

EX 1009

COMPUTER NETWORKS

Computer Networks

Third Edition

Other bestselling titles by Andrew S. Tanenbaum:

Operating Systems: Design and Implementation, 2nd edition

This now-classic text on operating systems is the only book covering both the principles of operating systems and their application to a real system. All the traditional operating systems topics are covered in detail. In addition, the principles are carefully illustrated by MINIX, a free UNIX-like operating system for personal computers. The second edition, which is expected in late 1996, will deal with the new POSIX-based MINIX 2.0 system. As with the first edition, the book will include a listing of the MINIX source code. New to the second edition is a free CD-ROM in each book containing the complete MINIX system (binary and source code).

Modern Operating Systems

This bestselling text presents the basics of both single processor and distributed computer systems. Tanenbaum covers traditional topics including processes, memory management, and files systems as well as key issues in distributed systems including the client-server model, remote procedure call, threads and distributed files servers. This practical guide uses UNIX, MS-DOS, Mach and Amoeba to illustrate operating system concepts.

Distributed Operating Systems

This text covers the fundamental concepts of distributed operating systems. Key topics include communication and synchronization, processes and processors, distributed shared memory, distributed file systems, and distributed real-time systems. The principles of distributed computing are illustrated in four detailed case studies using Mach, Amoeba, Chorus and DCE operating environments.

Structured Computer Organization, 3rd edition

This leading text looks at computer architecture as a series of levels. At the bottom is the hardware: transistors, gates, registers, adders, and other circuits. Then comes the microprogramming level. After that is the conventional machine level, with its ADD, MOVE, JUMP, and other instructions. On top of that is the operating system, which adds new facilities such as file management and virtual memory. The final chapter deals with two advanced topics: parallel computers and the design of RISC machines.

Computer Networks

Third Edition

Andrew S. Tanenbaum

*Vrije Universiteit
Amsterdam, The Netherlands*

For book and bookstore information



<http://www.prenhall.com>



*Prentice Hall PTR
Upper Saddle River, New Jersey 07458*

Library of Congress Cataloging in Publication Data

Tanenbaum, Andrew S. 1944-.

Computer networks / Andrew S. Tanenbaum. -- 3rd ed.

p. cm.

Includes bibliographical references and index.

ISBN 0-13-349945-6

1. Computer networks. I. Title.

TK5105.5.T36 1996

96-4121

004.6--dc20

CIP

Editorial/production manager: *Camille Trentacoste*

Interior design and composition: *Andrew S. Tanenbaum*

Cover design director: *Jerry Votta*

Cover designer: *Don Martinetti, DM Graphics, Inc.*

Cover concept: *Andrew S. Tanenbaum, from an idea by Marilyn Tremaine*

Interior graphics: *Hadel Studio*

Manufacturing manager: *Alexis R. Heydt*

Acquisitions editor: *Mary Franz*

Editorial Assistant: *Noreen Regina*



© 1996 by Prentice Hall PTR

Prentice-Hall, Inc.

A Simon & Schuster Company

Upper Saddle River, New Jersey 07458

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact:

Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, NJ 07458.

Phone: (800) 382-3419; Fax: (201) 236-7141. E-mail: corpsales@prenhall.com

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

All product names mentioned herein are the trademarks of their respective owners.

Printed in the United States of America

10 9 8 7 6 5

ISBN 0-13-349945-6

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Australia Pty. Limited, *Sydney*

Prentice-Hall Canada Inc., *Toronto*

Prentice-Hall Hispanoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall of Japan, Inc., *Tokyo*

Simon & Schuster Asia Pte. Ltd., *Singapore*

Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

To Suzanne, Barbara, Marvin, and Little Bram

CONTENTS

PREFACE

xv

1 INTRODUCTION

1

- 1.1 USES OF COMPUTER NETWORKS 3
 - 1.1.1 Networks for Companies 3
 - 1.1.2 Networks for People 4
 - 1.1.3 Social Issues 6
- 1.2 NETWORK HARDWARE 7
 - 1.2.1 Local Area Networks 9
 - 1.2.2 Metropolitan Area Networks 10
 - 1.2.3 Wide Area Networks 11
 - 1.2.4 Wireless Networks 13
 - 1.2.5 Internetworks 16
- 1.3 NETWORK SOFTWARE 16
 - 1.3.1 Protocol Hierarchies 17
 - 1.3.2 Design Issues for the Layers 21
 - 1.3.3 Interfaces and Services 22
 - 1.3.4 Connection-Oriented and Connectionless Services 23
 - 1.3.5 Service Primitives 25
 - 1.3.6 The Relationship of Services to Protocols 27
- 1.4 REFERENCE MODELS 28
 - 1.4.1 The OSI Reference Model 28
 - 1.4.2 The TCP/IP Reference Model 35
 - 1.4.3 A Comparison of the OSI and TCP/IP Reference Models 38
 - 1.4.4 A Critique of the OSI Model and Protocols 40
 - 1.4.5 A Critique of the TCP/IP Reference Model 43
- 1.5 EXAMPLE NETWORKS 44
 - 1.5.1 Novell NetWare 45
 - 1.5.2 The ARPANET 47
 - 1.5.3 NSFNET 50
 - 1.5.4 The Internet 52
 - 1.5.5 Gigabit Testbeds 54

1.6	EXAMPLE DATA COMMUNICATION SERVICES	56
1.6.1	SMDS—Switched Multimegabit Data Service	57
1.6.2	X.25 Networks	59
1.6.3	Frame Relay	60
1.6.4	Broadband ISDN and ATM	61
1.6.5	Comparison of Services	66
1.7	NETWORK STANDARDIZATION	66
1.7.1	Who's Who in the Telecommunications World	67
1.7.2	Who's Who in the International Standards World	69
1.7.3	Who's Who in the Internet Standards World	70
1.8	OUTLINE OF THE REST OF THE BOOK	72
1.9	SUMMARY	73

2 THE PHYSICAL LAYER

77

2.1	THE THEORETICAL BASIS FOR DATA COMMUNICATION	77
2.1.1	Fourier Analysis	78
2.1.2	Bandwidth-Limited Signals	78
2.1.3	The Maximum Data Rate of a Channel	81
2.2	TRANSMISSION MEDIA	82
2.2.1	Magnetic Media	82
2.2.2	Twisted Pair	83
2.2.3	Baseband Coaxial Cable	84
2.2.4	Broadband Coaxial Cable	85
2.2.5	Fiber Optics	87
2.3	WIRELESS TRANSMISSION	94
2.3.1	The Electromagnetic Spectrum	94
2.3.2	Radio Transmission	97
2.3.3	Microwave Transmission	98
2.3.4	Infrared and Millimeter Waves	100
2.3.5	Lightwave Transmission	100
2.4	THE TELEPHONE SYSTEM	102
2.4.1	Structure of the Telephone System	103
2.4.2	The Politics of Telephones	106
2.4.3	The Local Loop	108
2.4.4	Trunks and Multiplexing	118
2.4.5	Switching	130

- 2.5 NARROWBAND ISDN 139
 - 2.5.1 ISDN Services 140
 - 2.5.2 ISDN System Architecture 140
 - 2.5.3 The ISDN Interface 142
 - 2.5.4 Perspective on N-ISDN 143
- 2.6 BROADBAND ISDN AND ATM 144
 - 2.6.1 Virtual Circuits versus Circuit Switching 145
 - 2.6.2 Transmission in ATM Networks 146
 - 2.6.3 ATM Switches 147
- 2.7 CELLULAR RADIO 155
 - 2.7.1 Paging Systems 155
 - 2.7.2 Cordless Telephones 157
 - 2.7.3 Analog Cellular Telephones 157
 - 2.7.4 Digital Cellular Telephones 162
 - 2.7.5 Personal Communications Services 162
- 2.8 COMMUNICATION SATELLITES 163
 - 2.8.1 Geosynchronous Satellites 164
 - 2.8.2 Low-Orbit Satellites 167
 - 2.8.3 Satellites versus Fiber 168
- 2.9 SUMMARY 170

3 THE DATA LINK LAYER

How to send

175

- 3.1 DATA LINK LAYER DESIGN ISSUES 176
 - 3.1.1 Services Provided to the Network Layer 176
 - 3.1.2 Framing 179
 - 3.1.3 Error Control 182
 - 3.1.4 Flow Control 183
- 3.2 ERROR DETECTION AND CORRECTION 183
 - 3.2.1 Error-Correcting Codes 184
 - 3.2.2 Error-Detecting Codes 186
- 3.3 ELEMENTARY DATA LINK PROTOCOLS 190
 - 3.3.1 An Unrestricted Simplex Protocol 195
 - 3.3.2 A Simplex Stop-and-Wait Protocol 195
 - 3.3.3 A Simplex Protocol for a Noisy Channel 197

- 3.4 SLIDING WINDOW PROTOCOLS 202
 - 3.4.1 A One Bit Sliding Window Protocol 206
 - 3.4.2 A Protocol Using Go Back n 207
 - 3.4.3 A Protocol Using Selective Repeat 213
- 3.5 PROTOCOL SPECIFICATION AND VERIFICATION 219
 - 3.5.1 Finite State Machine Models 219
 - 3.5.2 Petri Net Models 223
- 3.6 EXAMPLE DATA LINK PROTOCOLS 225
 - 3.6.1 HDLC—High-level Data Link Control 225
 - 3.6.2 The Data Link Layer in the Internet 229
 - 3.6.3 The Data Link Layer in ATM 235
- 3.7. SUMMARY 239

4 THE MEDIUM ACCESS SUBLAYER *Medium Access* 243

- 4.1 THE CHANNEL ALLOCATION PROBLEM 244
 - 4.1.1 Static Channel Allocation in LANs and MANs 244
 - 4.1.2 Dynamic Channel Allocation in LANs and MANs 245
- 4.2 MULTIPLE ACCESS PROTOCOLS 246
 - 4.2.1 ALOHA 246
 - 4.2.2 Carrier Sense Multiple Access Protocols 250
 - 4.2.3 Collision-Free Protocols 254
 - 4.2.4 Limited-Contention Protocols 256
 - 4.2.5 Wavelength Division Multiple Access Protocols 260
 - 4.2.6 Wireless LAN Protocols 262
 - 4.2.7 Digital Cellular Radio 266
- 4.3 IEEE STANDARD 802 FOR LANS AND MANS 275
 - 4.3.1 IEEE Standard 802.3 and Ethernet 276
 - 4.3.2 IEEE Standard 802.4: Token Bus 287
 - 4.3.3 IEEE Standard 802.5: Token Ring 292
 - 4.3.4 Comparison of 802.3, 802.4, and 802.5 299
 - 4.3.5 IEEE Standard 802.6: Distributed Queue Dual Bus 301
 - 4.3.6 IEEE Standard 802.2: Logical Link Control 302

- 4.4 BRIDGES 304
 - 4.4.1 Bridges from 802.x to 802.y 307
 - 4.4.2 Transparent Bridges 310
 - 4.4.3 Source Routing Bridges 314
 - 4.4.4 Comparison of 802 Bridges 316
 - 4.4.5 Remote Bridges 317
- 4.5 HIGH-SPEED LANS 318
 - 4.5.1 FDDI 319
 - 4.5.2 Fast Ethernet 322
 - 4.5.3 HIPPI—High-Performance Parallel Interface 325
 - 4.5.4 Fibre Channel 326
- 4.6 SATELLITE NETWORKS 327
 - 4.6.1 Polling 328
 - 4.6.2 ALOHA 329
 - 4.6.3 FDM 330
 - 4.6.4 TDM 330
 - 4.6.5 CDMA 333
- 4.7 SUMMARY 333

5 THE NETWORK LAYER

339

- 5.1 NETWORK LAYER DESIGN ISSUES 339
 - 5.1.1 Services Provided to the Transport Layer 340
 - 5.1.2 Internal Organization of the Network Layer 342
 - 5.1.3 Comparison of Virtual Circuit and Datagram Subnets 344
- 5.2 ROUTING ALGORITHMS 345
 - 5.2.1 The Optimality Principle 347
 - 5.2.2 Shortest Path Routing 349
 - 5.2.3 Flooding 351
 - 5.2.4 Flow-Based Routing 353
 - 5.2.5 Distance Vector Routing 355
 - 5.2.6 Link State Routing 359
 - 5.2.7 Hierarchical Routing 365
 - 5.2.8 Routing for Mobile Hosts 367
 - 5.2.9 Broadcast Routing 370
 - 5.2.10 Multicast Routing 372

5.3	CONGESTION CONTROL ALGORITHMS	374
5.3.1	General Principles of Congestion Control	376
5.3.2	Congestion Prevention Policies	378
5.3.3	Traffic Shaping	379
5.3.4	Flow Specifications	384
5.3.5	Congestion Control in Virtual Circuit Subnets	386
5.3.6	Choke Packets	387
5.3.7	Load Shedding	390
5.3.8	Jitter Control	392
5.3.9	Congestion Control for Multicasting	393
5.4	INTERNETWORKING	396
5.4.1	How Networks Differ	399
5.4.2	Concatenated Virtual Circuits	401
5.4.3	Connectionless Internetworking	402
5.4.4	Tunneling	404
5.4.5	Internetwork Routing	405
5.4.6	Fragmentation	406
5.4.7	Firewalls	410
5.5	THE NETWORK LAYER IN THE INTERNET	412
5.5.1	The IP Protocol	413
5.5.2	IP Addresses	416
5.5.3	Subnets	417
5.5.4	Internet Control Protocols	419
5.5.5	The Interior Gateway Routing Protocol: OSPF	424
5.5.6	The Exterior Gateway Routing Protocol: BGP	429
5.5.7	Internet Multicasting	431
5.5.8	Mobile IP	432
5.5.9	CIDR—Classless InterDomain Routing	434
5.5.10	IPv6	437
5.6	THE NETWORK LAYER IN ATM NETWORKS	449
5.6.1	Cell Formats	450
5.6.2	Connection Setup	452
5.6.3	Routing and Switching	455
5.6.4	Service Categories	458
5.6.5	Quality of Service	460
5.6.6	Traffic Shaping and Policing	463
5.6.7	Congestion Control	467
5.6.8	ATM LANs	471
5.7	SUMMARY	473

6 THE TRANSPORT LAYER**479**

- 6.1 THE TRANSPORT SERVICE 479
 - 6.1.1 Services Provided to the Upper Layers 479
 - 6.1.2 Quality of Service 481
 - 6.1.3 Transport Service Primitives 483
- 6.2 ELEMENTS OF TRANSPORT PROTOCOLS 488
 - 6.2.1 Addressing 489
 - 6.2.2 Establishing a Connection 493
 - 6.2.3 Releasing a Connection 498
 - 6.2.4 Flow Control and Buffering 502
 - 6.2.5 Multiplexing 506
 - 6.2.6 Crash Recovery 508
- 6.3 A SIMPLE TRANSPORT PROTOCOL 510
 - 6.3.1 The Example Service Primitives 510
 - 6.3.2 The Example Transport Entity 512
 - 6.3.3 The Example as a Finite State Machine 519
- 6.4 THE INTERNET TRANSPORT PROTOCOLS (TCP AND UDP) 521
 - 6.4.1 The TCP Service Model 523
 - 6.4.2 The TCP Protocol 524
 - 6.4.3 The TCP Segment Header 526
 - 6.4.4 TCP Connection Management 529
 - 6.4.5 TCP Transmission Policy 533
 - 6.4.6 TCP Congestion Control 536
 - 6.4.7 TCP Timer Management 539
 - 6.4.8 UDP 542
 - 6.4.9 Wireless TCP and UDP 543
- 6.5 THE ATM AAL LAYER PROTOCOLS 545
 - 6.5.1 Structure of the ATM Adaptation Layer 546
 - 6.5.2 AAL 1 547
 - 6.5.3 AAL 2 549
 - 6.5.4 AAL 3/4 550
 - 6.5.5 AAL 5 552
 - 6.5.6 Comparison of AAL Protocols 554
 - 6.5.7 SSCOP—Service Specific Connection-Oriented Protocol 555
- 6.6 PERFORMANCE ISSUES 555
 - 6.6.1 Performance Problems in Computer Networks 556
 - 6.6.2 Measuring Network Performance 559

- 6.6.3 System Design for Better Performance 561
- 6.6.4 Fast TPDU Processing 565
- 6.6.5 Protocols for Gigabit Networks 568
- 6.7 SUMMARY 572

7 THE APPLICATION LAYER

577

- 7.1 NETWORK SECURITY 577
 - 7.1.1 Traditional Cryptography 580
 - 7.1.2 Two Fundamental Cryptographic Principles 585
 - 7.1.3 Secret-Key Algorithms 587
 - 7.1.4 Public-Key Algorithms 597
 - 7.1.5 Authentication Protocols 601
 - 7.1.6 Digital Signatures 613
 - 7.1.7 Social Issues 620
- 7.2 DNS—DOMAIN NAME SYSTEM 622
 - 7.2.1 The DNS Name Space 622
 - 7.2.2 Resource Records 624
 - 7.2.3 Name Servers 628
- 7.3 SNMP—SIMPLE NETWORK MANAGEMENT PROTOCOL 630
 - 7.3.1 The SNMP Model 631
 - 7.3.2 ASN.1—Abstract Syntax Notation 1 633
 - 7.3.3 SMI—Structure of Management Information 639
 - 7.3.4 The MIB—Management Information Base 641
 - 7.3.5 The SNMP Protocol 642
- 7.4 ELECTRONIC MAIL 643
 - 7.4.1 Architecture and Services 645
 - 7.4.2 The User Agent 646
 - 7.4.3 Message Formats 650
 - 7.4.4 Message Transfer 657
 - 7.4.5 Email Privacy 663
- 7.5 USENET NEWS 669
 - 7.5.1 The User View of USENET 670
 - 7.5.2 How USENET is Implemented 675

- 7.6 THE WORLD WIDE WEB 681
 - 7.6.1 The Client Side 682
 - 7.6.2 The Server Side 685
 - 7.6.3 Writing a Web Page in HTML 691
 - 7.6.4 Java 706
 - 7.6.5 Locating Information on the Web 720
- 7.7 MULTIMEDIA 723
 - 7.7.1 Audio 724
 - 7.7.2 Video 727
 - 7.7.3 Data Compression 730
 - 7.7.4 Video on Demand 744
 - 7.7.5 MBone—Multicast Backbone 756
- 7.8 SUMMARY 760

8 READING LIST AND BIBLIOGRAPHY

767

- 8.1 SUGGESTIONS FOR FURTHER READING 767
 - 8.1.1 Introduction and General Works 768
 - 8.1.2 The Physical Layer 769
 - 8.1.3 The Data Link Layer 770
 - 8.1.4 The Medium Access Control Sublayer 770
 - 8.1.5 The Network Layer 771
 - 8.1.6 The Transport Layer 772
 - 8.1.7 The Application Layer 772
- 8.2 ALPHABETICAL BIBLIOGRAPHY 775

INDEX 795

4.3.1. IEEE Standard 802.3 and Ethernet

The IEEE 802.3 standard is for a 1-persistent CSMA/CD LAN. To review the idea, when a station wants to transmit, it listens to the cable. If the cable is busy, the station waits until it goes idle; otherwise it transmits immediately. If two or more stations simultaneously begin transmitting on an idle cable, they will collide. All colliding stations then terminate their transmission, wait a random time, and repeat the whole process all over again.

The 802.3 standard has an interesting history. The real beginning was the ALOHA system constructed to allow radio communication between machines scattered over the Hawaiian Islands. Later, carrier sensing was added, and Xerox PARC built a 2.94-Mbps CSMA/CD system to connect over 100 personal workstations on a 1-km cable (Metcalf and Boggs, 1976). This system was called **Ethernet** after the *luminiferous ether*, through which electromagnetic radiation was once thought to propagate. (When the Nineteenth Century British physicist James Clerk Maxwell discovered that electromagnetic radiation could be described by a wave equation, scientists assumed that space must be filled with some ethereal medium in which the radiation was propagating. Only after the famous Michelson-Morley experiment in 1887, did physicists discover that electromagnetic radiation could propagate in a vacuum.)

The Xerox Ethernet was so successful that Xerox, DEC, and Intel drew up a standard for a 10-Mbps Ethernet. This standard formed the basis for 802.3. The published 802.3 standard differs from the Ethernet specification in that it describes a whole family of 1-persistent CSMA/CD systems, running at speeds from 1 to 10-Mbps on various media. Also, the one header field differs between the two (the 802.3 length field is used for packet type in Ethernet). The initial standard also gives the parameters for a 10 Mbps baseband system using 50-ohm coaxial cable. Parameter sets for other media and speeds came later.

Many people (incorrectly) use the name "Ethernet" in a generic sense to refer to all CSMA/CD protocols, even though it really refers to a specific product that almost implements 802.3. We will use the terms "802.3" and "CSMA/CD" except when specifically referring to the Ethernet product in the next few paragraphs.

802.3 Cabling

Since the name "Ethernet" refers to the cable (the ether), let us start our discussion there. Four types of cabling are commonly used, as shown in Fig. 4-17. Historically, **10Base5** cabling, popularly called **thick Ethernet**, came first. It resembles a yellow garden hose, with markings every 2.5 meters to show where the taps go. (The 802.3 standard does not actually *require* the cable to be yellow, but it does *suggest* it.) Connections to it are generally made using **vampire taps**, in which a pin is carefully forced halfway into the coaxial cable's core. The

notation 10Base5 means that it operates at 10 Mbps, uses baseband signaling, and can support segments of up to 500 meters.

Name	Cable	Max. segment	Nodes/seg.	Advantages
10Base5	Thick coax	500 m	100	Good for backbones
10Base2	Thin coax	200 m	30	Cheapest system
10Base-T	Twisted pair	100 m	1024	Easy maintenance
10Base-F	Fiber optics	2000 m	1024	Best between buildings

Fig. 4-17. The most common kinds of baseband 802.3 LANs.

Historically, the second cable type was **10Base2** or **thin Ethernet**, which, in contrast to the garden-hose-like thick Ethernet, bends easily. Connections to it are made using industry standard BNC connectors to form T junctions, rather than using vampire taps. These are easier to use and more reliable. Thin Ethernet is much cheaper and easier to install, but it can run for only 200 meters and can handle only 30 machines per cable segment.

Detecting cable breaks, bad taps, or loose connectors can be a major problem with both media. For this reason, techniques have been developed to track them down. Basically, a pulse of known shape is injected into the cable. If the pulse hits an obstacle or the end of the cable, an echo will be generated and sent back. By carefully timing the interval between sending the pulse and receiving the echo, it is possible to localize the origin of the echo. This technique is called **time domain reflectometry**.

The problems associated with finding cable breaks have driven systems toward a different kind of wiring pattern, in which all stations have a cable running to a central **hub**. Usually, these wires are telephone company twisted pairs, since most office buildings are already wired this way, and there are normally plenty of spare pairs available. This scheme is called **10Base-T**.

These three wiring schemes are illustrated in Fig. 4-18. For 10Base5, a **transceiver** is clamped securely around the cable so that its tap makes contact with the inner core. The transceiver contains the electronics that handle carrier detection and collision detection. When a collision is detected, the transceiver also puts a special invalid signal on the cable to ensure that all other transceivers also realize that a collision has occurred.

With 10Base5, a **transceiver cable** connects the transceiver to an interface board in the computer. The transceiver cable may be up to 50 meters long and contains five individually shielded twisted pairs. Two of the pairs are for data in and data out, respectively. Two more are for control signals in and out. The fifth pair, which is not always used, allows the computer to power the transceiver electronics. Some transceivers allow up to eight nearby computers to be attached to them, to reduce the number of transceivers needed.

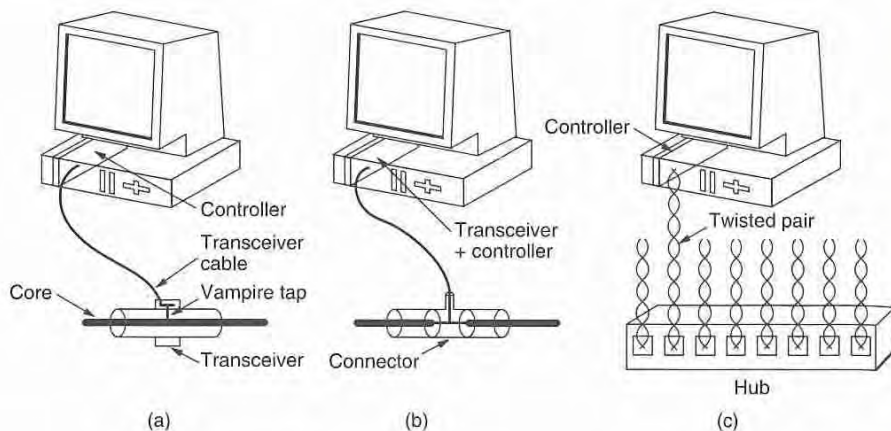


Fig. 4-18. Three kinds of 802.3 cabling. (a) 10Base5. (b) 10Base2. (c) 10Base-T.

The transceiver cable terminates on an interface board inside the computer. The interface board contains a controller chip that transmits frames to, and receives frames from, the transceiver. The controller is responsible for assembling the data into the proper frame format, as well as computing checksums on outgoing frames and verifying them on incoming frames. Some controller chips also manage a pool of buffers for incoming frames, a queue of buffers to be transmitted, DMA transfers with the host computers, and other aspects of network management.

With 10Base2, the connection to the cable is just a passive BNC T-junction connector. The transceiver electronics are on the controller board, and each station always has its own transceiver.

With 10Base-T, there is no cable at all, just the hub (a box full of electronics). Adding or removing a station is simpler in this configuration, and cable breaks can be detected easily. The disadvantage of 10Base-T is that the maximum cable run from the hub is only 100 meters, maybe 150 meters if high-quality (category 5) twisted pairs are used. Also, a large hub costs thousands of dollars. Still, 10Base-T is becoming steadily more popular due to the ease of maintenance that it offers. A faster version of 10Base-T (100Base-T) will be discussed later in this chapter.

A fourth cabling option for 802.3 is **10Base-F**, which uses fiber optics. This alternative is expensive due to the cost of the connectors and terminators, but it has excellent noise immunity and is the method of choice when running between buildings or widely separated hubs.

Figure 4-19 shows different ways of wiring up a building. In Fig. 4-19(a), a single cable is snaked from room to room, with each station tapping onto it at the nearest point. In Fig. 4-19(b), a vertical spine runs from the basement to the roof,

with horizontal cables on each floor connected to it by special amplifiers (repeaters). In some buildings the horizontal cables are thin, and the backbone is thick. The most general topology is the tree, as in Fig. 4-19(c), because a network with two paths between some pairs of stations would suffer from interference between the two signals.

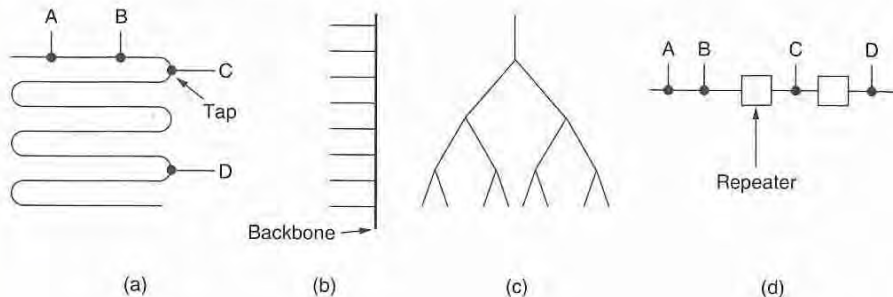


Fig. 4-19. Cable topologies. (a) Linear. (b) Spine. (c) Tree. (d) Segmented.

Each version of 802.3 has a maximum cable length per segment. To allow larger networks, multiple cables can be connected by **repeaters**, as shown in Fig. 4-19(d). A repeater is a physical layer device. It receives, amplifies, and retransmits signals in both directions. As far as the software is concerned, a series of cable segments connected by repeaters is no different than a single cable (except for some delay introduced by the repeaters). A system may contain multiple cable segments and multiple repeaters, but no two transceivers may be more than 2.5 km apart and no path between any two transceivers may traverse more than four repeaters.

Manchester Encoding

None of the versions of 802.3 use straight binary encoding with 0 volts for a 0 bit and 5 volts for a 1 bit because it leads to ambiguities. If one station sends the bit string 0001000, others might falsely interpret it as 10000000 or 01000000 because they cannot tell the difference between an idle sender (0 volts) and a 0 bit (0 volts).

What is needed is a way for receivers to unambiguously determine the start, end, or middle of each bit without reference to an external clock. Two such approaches are called **Manchester encoding** and **differential Manchester encoding**. With Manchester encoding, each bit period is divided into two equal intervals. A binary 1 bit is sent by having the voltage set high during the first interval and low in the second one. A binary 0 is just the reverse: first low and then high. This scheme ensures that every bit period has a transition in the middle, making it easy for the receiver to synchronize with the sender. A

disadvantage of Manchester encoding is that it requires twice as much bandwidth as straight binary encoding, because the pulses are half the width. Manchester encoding is shown in Fig. 4-20(b).

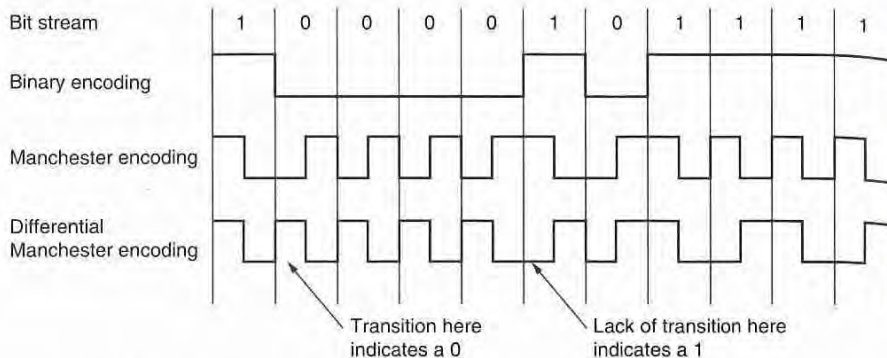


Fig. 4-20. (a) Binary encoding. (b) Manchester encoding. (c) Differential Manchester encoding.

Differential Manchester encoding, shown in Fig. 4-20(c), is a variation of basic Manchester encoding. In it, a 1 bit is indicated by the absence of a transition at the start of the interval. A 0 bit is indicated by the presence of a transition at the start of the interval. In both cases, there is a transition in the middle as well. The differential scheme requires more complex equipment but offers better noise immunity. All 802.3 baseband systems use Manchester encoding due to its simplicity. The high signal is +0.85 volts and the low signal is -0.85 volts, giving a DC value of 0 volts.

The 802.3 MAC Sublayer Protocol

The 802.3 (IEEE, 1985a) frame structure is shown in Fig. 4-21. Each frame starts with a *Preamble* of 7 bytes, each containing the bit pattern 10101010. The Manchester encoding of this pattern produces a 10-MHz square wave for 5.6 μsec to allow the receiver's clock to synchronize with the sender's. Next comes a *Start of frame* byte containing 10101011 to denote the start of the frame itself.

The frame contains two addresses, one for the destination and one for the source. The standard allows 2-byte and 6-byte addresses, but the parameters defined for the 10-Mbps baseband standard use only the 6-byte addresses. The high-order bit of the destination address is a 0 for ordinary addresses and 1 for group addresses. Group addresses allow multiple stations to listen to a single address. When a frame is sent to a group address, all the stations in the group receive it. Sending to a group of stations is called **multicast**. The address consisting of all 1 bits is reserved for **broadcast**. A frame containing all 1s in the destination field is delivered to all stations on the network.

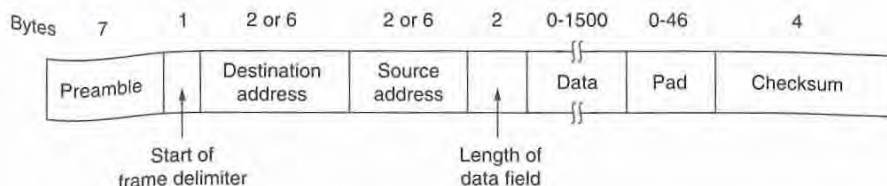


Fig. 4-21. The 802.3 frame format.

Another interesting feature of the addressing is the use of bit 46 (adjacent to the high-order bit) to distinguish local from global addresses. Local addresses are assigned by each network administrator and have no significance outside the local network. Global addresses, in contrast, are assigned by IEEE to ensure that no two stations anywhere in the world have the same global address. With $48 - 2 = 46$ bits available, there are about 7×10^{13} global addresses. The idea is that any station can uniquely address any other station by just giving the right 48-bit number. It is up to the network layer to figure out how to locate the destination.

The *Length* field tells how many bytes are present in the data field, from a minimum of 0 to a maximum of 1500. While a data field of 0 bytes is legal, it causes a problem. When a transceiver detects a collision, it truncates the current frame, which means that stray bits and pieces of frames appear on the cable all the time. To make it easier to distinguish valid frames from garbage, 802.3 states that valid frames must be at least 64 bytes long, from destination address to checksum. If the data portion of a frame is less than 46 bytes, the pad field is used to fill out the frame to the minimum size.

Another (and more important) reason for having a minimum length frame is to prevent a station from completing the transmission of a short frame before the first bit has even reached the far end of the cable, where it may collide with another frame. This problem is illustrated in Fig. 4-22. At time 0, station A, at one end of the network, sends off a frame. Let us call the propagation time for this frame to reach the other end τ . Just before the frame gets to the other end (i.e., at time $\tau - \epsilon$) the most distant station, B, starts transmitting. When B detects that it is receiving more power than it is putting out, it knows that a collision has occurred, so it aborts its transmission and generates a 48-bit noise burst to warn all other stations. At about time 2τ , the sender sees the noise burst and aborts its transmission, too. It then waits a random time before trying again.

If a station tries to transmit a very short frame, it is conceivable that a collision occurs, but the transmission completes before the noise burst gets back at 2τ . The sender will then incorrectly conclude that the frame was successfully sent. To prevent this situation from occurring, all frames must take more than 2τ to send. For a 10-Mbps LAN with a maximum length of 2500 meters and four

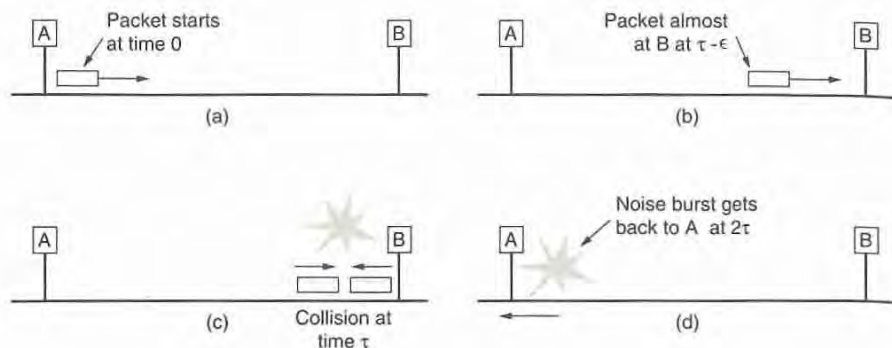


Fig. 4-22. Collision detection can take as long as 2τ .

repeaters (from the 802.3 specification), the minimum allowed frame must take 51.2 μsec . This time corresponds to 64 bytes. Frames with fewer bytes are padded out to 64 bytes.

As the network speed goes up, the minimum frame length must go up or the maximum cable length must come down, proportionally. For a 2500-meter LAN operating at 1 Gbps, the minimum frame size would have to be 6400 bytes. Alternatively, the minimum frame size could be 640 bytes and the maximum distance between any two stations 250 meters. These restrictions are becoming increasingly painful as we move toward gigabit networks.

The final 802.3 field is the *Checksum*. It is effectively a 32-bit hash code of the data. If some data bits are erroneously received (due to noise on the cable), the checksum will almost certainly be wrong, and the error will be detected. The checksum algorithm is a cyclic redundancy check of the kind discussed in Chap. 3.

The Binary Exponential Backoff Algorithm

Let us now see how randomization is done when a collision occurs. The model is that of Fig. 4-5. After a collision, time is divided up into discrete slots whose length is equal to the worst case round-trip propagation time on the ether (2τ). To accommodate the longest path allowed by 802.3 (2.5 km and four repeaters), the slot time has been set to 512 bit times, or 51.2 μsec .

After the first collision, each station waits either 0 or 1 slot times before trying again. If two stations collide and each one picks the same random number, they will collide again. After the second collision, each one picks either 0, 1, 2, or 3 at random and waits that number of slot times. If a third collision occurs (the probability of this happening is 0.25), then the next time the number of slots to wait is chosen at random from the interval 0 to $2^3 - 1$.

5

THE NETWORK LAYER

The network layer is concerned with getting packets from the source all the way to the destination. Getting to the destination may require making many hops at intermediate routers along the way. This function clearly contrasts with that of the data link layer, which has the more modest goal of just moving frames from one end of a wire to the other. Thus the network layer is the lowest layer that deals with end-to-end transmission. For more information about it, see (Huitema, 1995; and Perlman, 1992).

To achieve its goals, the network layer must know about the topology of the communication subnet (i.e., the set of all routers) and choose appropriate paths through it. It must also take care to choose routes to avoid overloading some of the communication lines and routers while leaving others idle. Finally, when the source and destination are in different networks, it is up to the network layer to deal with these differences and solve the problems that result from them. In this chapter we will study all these issues and illustrate them with our two running examples, the Internet and ATM.

5.1. NETWORK LAYER DESIGN ISSUES

In the following sections we will provide an introduction to some of the issues that the designers of the network layer must grapple with. These issues include the service provided to the transport layer and the internal design of the subnet.

5.1.1. Services Provided to the Transport Layer

The network layer provides services to the transport layer at the network layer/transport layer interface. This interface is often especially important for another reason: it frequently is the interface between the carrier and the customer, that is, the boundary of the subnet. The carrier often has control of the protocols and interfaces up to and including the network layer. Its job is to deliver packets given to it by its customers. For this reason, this interface must be especially well defined.

The network layer services have been designed with the following goals in mind.

1. The services should be independent of the subnet technology.
2. The transport layer should be shielded from the number, type, and topology of the subnets present.
3. The network addresses made available to the transport layer should use a uniform numbering plan, even across LANs and WANs.

Given these goals, the designers of the network layer have a lot of freedom in writing detailed specifications of the services to be offered to the transport layer. This freedom often degenerates into a raging battle between two warring factions. The discussion centers on the question of whether the network layer should provide connection-oriented service or connectionless service.

One camp (represented by the Internet community) argues that the subnet's job is moving bits around and nothing else. In their view (based on nearly 30 years of actual experience with a real, working computer network), the subnet is inherently unreliable, no matter how it is designed. Therefore, the hosts should accept the fact that it is unreliable and do error control (i.e., error detection and correction) and flow control themselves.

This viewpoint leads quickly to the conclusion that the network service should be connectionless, with primitives `SEND PACKET` and `RECEIVE PACKET`, and little else. In particular, no packet ordering and flow control should be done, because the hosts are going to do that anyway, and there is probably little to be gained by doing it twice. Furthermore, each packet must carry the full destination address, because each packet sent is carried independently of its predecessors, if any.

The other camp (represented by the telephone companies) argues that the subnet should provide a (reasonably) reliable, connection-oriented service. They claim 100 years of successful experience with the worldwide telephone system is a good guide. In this view, connections should have the following properties:

1. Before sending data, a network layer process on the sending side must set up a connection to its peer on the receiving side. This connection, which is given a special identifier, is then used until all the data have been sent, at which time it is explicitly released.

2. When a connection is set up, the two processes can enter into a negotiation about the parameters, quality, and cost of the service to be provided.
3. Communication is in both directions, and packets are delivered in sequence.
4. Flow control is provided automatically to prevent a fast sender from dumping packets into the pipe at a higher rate than the receiver can take them out, thus leading to overflow.

Other properties, such as guaranteed delivery, explicit confirmation of delivery, and high priority packets are optional. As we pointed out in Chap. 1, connectionless service is like the postal system, and connection-oriented service is like the telephone system.

The argument between connection-oriented and connectionless service really has to do with where to put the complexity. In the connection-oriented service, it is in the network layer (subnet); in the connectionless service, it is in the transport layer (hosts). Supporters of connectionless service say that user computing power has become cheap, so that there is no reason not to put the complexity in the hosts. Furthermore, they argue that the subnet is a major (inter)national investment that will last for decades, so it should not be cluttered up with features that may become obsolete quickly but will have to be calculated into the price structure for many years. Furthermore, some applications, such as digitized voice and real-time data collection may regard *speedy* delivery as much more important than *accurate* delivery.

On the other hand, supporters of connection-oriented service say that most users are not interested in running complex transport layer protocols in their machines. What they want is reliable, trouble-free service, and this service can be best provided with network layer connections. Furthermore, some services, such as real time audio and video are much easier to provide on top of a connection-oriented network layer than on top of a connectionless network layer.

Although it is rarely discussed in these terms, two separate issues are involved here. First, whether the network is connection-oriented (setup required) or connectionless (no setup required). Second, whether it is reliable (no lost, duplicated, or garbled packets) or unreliable (packets can be lost, duplicated, or garbled). In theory, all four combinations exist, but the dominant combinations are reliable connection-oriented and unreliable connectionless, so the other two tend to get lost in the noise.

These two camps are represented by our two running examples. The Internet has a connectionless network layer, and ATM networks have a connection-oriented network layer. An obvious question arises about how the Internet works when it runs over an ATM-based, carrier-provided subnet. The answer is that the source host first establishes an ATM network layer connection to the destination

host and then sends independent (IP) packets over it, as shown in Fig. 5-1. Although this approach works, it is inefficient because certain functionality is in both layers. For example, the ATM network layer guarantees that packets are always delivered in order, but the TCP code still contains the full mechanism for managing and reordering out-of-order packets. For more information about how to run IP over ATM, see RFC 1577 and (Armitage and Adams, 1995).

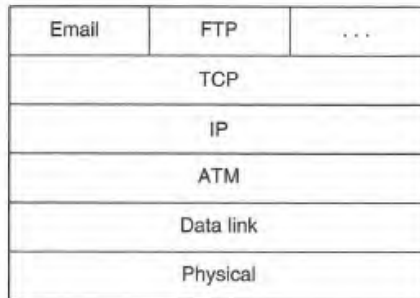


Fig. 5-1. Running TCP/IP over an ATM subnet.

5.1.2. Internal Organization of the Network Layer

Having looked at the two classes of service the network layer can provide to its users, it is time to see how it works inside. There are basically two different philosophies for organizing the subnet, one using connections and the other working connectionless. In the context of the *internal* operation of the subnet, a connection is usually called a **virtual circuit**, in analogy with the physical circuits set up by the telephone system. The independent packets of the connectionless organization are called **datagrams**, in analogy with telegrams.

Virtual circuits are generally used in subnets whose primary service is connection-oriented, so we will describe them in that context. The idea behind virtual circuits is to avoid having to choose a new route for every packet or cell sent. Instead, when a connection is established, a route from the source machine to the destination machine is chosen as part of the connection setup and remembered. That route is used for all traffic flowing over the connection, exactly the same way that the telephone system works. When the connection is released, the virtual circuit is also terminated.

In contrast, with a datagram subnet no routes are worked out in advance, even if the service is connection-oriented. Each packet sent is routed independently of its predecessors. Successive packets may follow different routes. While datagram subnets have to do more work, they are also generally more robust and adapt to failures and congestion more easily than virtual circuit subnets. We will discuss the pros and cons of the two approaches later.

If packets flowing over a given virtual circuit always take the same route through the subnet, each router must remember where to forward packets for each of the currently open virtual circuits passing through it. Every router must maintain a table with one entry per open virtual circuit passing through it. Each packet traveling through the subnet must contain a virtual circuit number field in its header, in addition to sequence numbers, checksums, and the like. When a packet arrives at a router, the router knows on which line it arrived and what the virtual circuit number is. Based on only this information, the packet must be forwarded on the correct output line.

When a network connection is set up, a virtual circuit number not already in use on that machine is chosen as the connection identifier. Since each machine chooses virtual circuit numbers independently, these numbers have only local significance. If they were globally significant over the whole network, it is likely that two virtual circuits bearing the same global virtual circuit number might pass through some intermediate router, leading to ambiguities.

Because virtual circuits can be initiated from either end, a problem occurs when call setups are propagating in both directions at once along a chain of routers. At some point they have arrived at adjacent routers. Each router must now pick a virtual circuit number to use for the (full-duplex) circuit it is trying to establish. If they have been programmed to choose the lowest number not already in use on the link, they will pick the same number, causing two unrelated virtual circuits over the same physical line to have the same number. When a data packet arrives later, the receiving router has no way of telling whether it is a forward packet on one circuit or a reverse packet on the other. If circuits are simplex, there is no ambiguity.

Note that every process must be required to indicate when it is through using a virtual circuit, so that the virtual circuit can be purged from the router tables to recover the space. In public networks, the motivation is the stick rather than the carrot: users are invariably charged for connect time as well as for data transported. In addition, some provision must be made for dealing with machines that terminate their virtual circuits by crashing rather than politely releasing them when done.

So much for the use of virtual circuits internal to the subnet. The other possibility is to use datagrams internally, in which case the routers do not have a table with one entry for each open virtual circuit. Instead, they have a table telling which outgoing line to use for each possible destination router. These tables are also needed when virtual circuits are used internally, to determine the route for a setup packet.

Each datagram must contain the full destination address. For a large network, these addresses can be quite long (e.g., a dozen bytes or more). When a packet comes in, the router looks up the outgoing line to use and sends the packet on its way. Also, the establishment and release of network or transport layer connections do not require any special work on the part of the routers.

5.1.3. Comparison of Virtual Circuit and Datagram Subnets

Both virtual circuits and datagrams have their supporters and their detractors. We will now attempt to summarize the arguments both ways. The major issues are listed in Fig. 5-2, although purists could probably find a counterexample for everything in the figure.

Issue	Datagram subnet	VC subnet
Circuit setup	Not needed	Required
Addressing	Each packet contains the full source and destination address	Each packet contains a short VC number
State information	Subnet does not hold state information	Each VC requires subnet table space
Routing	Each packet is routed independently	Route chosen when VC is set up; all packets follow this route
Effect of router failures	None, except for packets lost during the crash	All VCs that passed through the failed router are terminated
Congestion control	Difficult	Easy if enough buffers can be allocated in advance for each VC

Fig. 5-2. Comparison of datagram and virtual circuit subnets.

Inside the subnet, several trade-offs exist between virtual circuits and datagrams. One trade-off is between router memory space and bandwidth. Virtual circuits allow packets to contain circuit numbers instead of full destination addresses. If the packets tend to be fairly short, a full destination address in every packet may represent a significant amount of overhead, and hence wasted bandwidth. The price paid for using virtual circuits internally is the table space within the routers. Depending upon the relative cost of communication circuits versus router memory, one or the other may be cheaper.

Another trade-off is setup time versus address parsing time. Using virtual circuits requires a setup phase, which takes time and consumes resources. However, figuring out what to do with a data packet in a virtual circuit subnet is easy: the router just uses the circuit number to index into a table to find out where the packet goes. In a datagram subnet, a more complicated procedure is required to determine where the packet goes.

Virtual circuits have some advantages in avoiding congestion within the

subnet because resources can be reserved in advance, when the connection is established. Once the packets start arriving, the necessary bandwidth and router capacity will be there. With a datagram subnet, congestion avoidance is more difficult.

For transaction processing systems (e.g., stores calling up to verify credit card purchases), the overhead required to set up and clear a virtual circuit may easily dwarf the use of the circuit. If the majority of the traffic is expected to be of this kind, the use of switched virtual circuits inside the subnet makes little sense. On the other hand, permanent virtual circuits, which are set up manually and last for months or years, may be useful here.

Virtual circuits also have a vulnerability problem. If a router crashes and loses its memory, even if it comes back up a second later, all the virtual circuits passing through it will have to be aborted. In contrast, if a datagram router goes down, only those users whose packets were queued up in the router at the time will suffer, and maybe not even all those, depending upon whether they have already been acknowledged or not. The loss of a communication line is fatal to virtual circuits using it but can be easily compensated for if datagrams are used. Datagrams also allow the routers to balance the traffic throughout the subnet, since routes can be changed halfway through a connection.

It is worth explicitly pointing out that the service offered (connection-oriented or connectionless) is a separate issue from the subnet structure (virtual circuit or datagram). In theory, all four combinations are possible. Obviously, a virtual circuit implementation of a connection-oriented service and a datagram implementation of a connectionless service are reasonable. Implementing connections using datagrams also makes sense when the subnet is trying to provide a highly robust service.

The fourth possibility, a connectionless service on top of a virtual circuit subnet, seems strange but certainly occurs. The obvious example is running IP over an ATM subnet. Here it is desired to run an existing connectionless protocol over a new connection-oriented network layer. As mentioned earlier, this is more of an ad hoc solution to a problem than a good design. In a new system designed to run over an ATM subnet, one would not normally put a connectionless protocol like IP over a connection-oriented network layer like ATM and then layer a connection-oriented transport protocol on top of the connectionless protocol. Examples of all four cases are shown in Fig. 5-3.

5.2. ROUTING ALGORITHMS

The main function of the network layer is routing packets from the source machine to the destination machine. In most subnets, packets will require multiple hops to make the journey. The only notable exception is for broadcast

Upper layer	Type of subnet	
	Datagram	Virtual circuit
Connectionless	UDP over IP	UDP over IP over ATM
Connection-oriented	TCP over IP	ATM AAL1 over ATM

Fig. 5-3. Examples of different combinations of service and subnet structure.

networks, but even here routing is an issue if the source and destination are not on the same network. The algorithms that choose the routes and the data structures that they use are a major area of network layer design.

The **routing algorithm** is that part of the network layer software responsible for deciding which output line an incoming packet should be transmitted on. If the subnet uses datagrams internally, this decision must be made anew for every arriving data packet since the best route may have changed since last time. If the subnet uses virtual circuits internally, routing decisions are made only when a new virtual circuit is being set up. Thereafter, data packets just follow the previously established route. The latter case is sometimes called **session routing**, because a route remains in force for an entire user session (e.g., a login session at a terminal or a file transfer).

Regardless of whether routes are chosen independently for each packet or only when new connections are established, there are certain properties that are desirable in a routing algorithm: correctness, simplicity, robustness, stability, fairness, and optimality. Correctness and simplicity hardly require comment, but the need for robustness may be less obvious at first. Once a major network comes on the air, it may be expected to run continuously for years without systemwide failures. During that period there will be hardware and software failures of all kinds. Hosts, routers, and lines will go up and down repeatedly, and the topology will change many times. The routing algorithm should be able to cope with changes in the topology and traffic without requiring all jobs in all hosts to be aborted and the network to be rebooted every time some router crashes.

Stability is also an important goal for the routing algorithm. There exist routing algorithms that never converge to equilibrium, no matter how long they run. Fairness and optimality may sound obvious—surely no one would oppose them—but as it turns out, they are often contradictory goals. As a simple example of this conflict, look at Fig. 5-4. Suppose that there is enough traffic between A and A' , between B and B' , and between C and C' to saturate the horizontal links. To maximize the total flow, the X to X' traffic should be shut off altogether.

Unfortunately, X and X' may not see it that way. Evidently, some compromise between global efficiency and fairness to individual connections is needed.

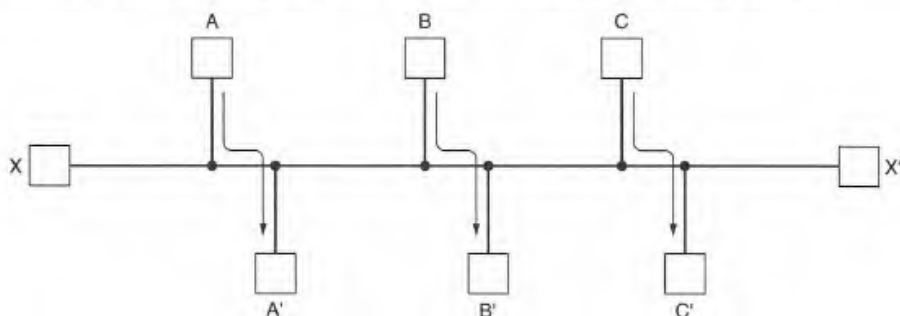


Fig. 5-4. Conflict between fairness and optimality.

Before we can even attempt to find trade-offs between fairness and optimality, we must decide what it is we seek to optimize. Minimizing mean packet delay is an obvious candidate, but so is maximizing total network throughput. Furthermore, these two goals are also in conflict, since operating any queueing system near capacity implies a long queueing delay. As a compromise, many networks attempt to minimize the number of hops a packet must make, because reducing the number of hops tends to improve the delay and also reduce the amount of bandwidth consumed, which tends to improve the throughput as well.

Routing algorithms can be grouped into two major classes: nonadaptive and adaptive. **Nonadaptive algorithms** do not base their routing decisions on measurements or estimates of the current traffic and topology. Instead, the choice of the route to use to get from I to J (for all I and J) is computed in advance, off-line, and downloaded to the routers when the network is booted. This procedure is sometimes called **static routing**.

Adaptive algorithms, in contrast, change their routing decisions to reflect changes in the topology, and usually the traffic as well. Adaptive algorithms differ in where they get their information (e.g., locally, from adjacent routers, or from all routers), when they change the routes (e.g., every ΔT sec, when the load changes, or when the topology changes), and what metric is used for optimization (e.g., distance, number of hops, or estimated transit time). In the following sections we will discuss a variety of routing algorithms, both static and dynamic.

5.2.1. The Optimality Principle

Before getting into specific algorithms, it may be helpful to note that one can make a general statement about optimal routes without regard to network topology or traffic. This statement is known as the **optimality principle**. It states that if router J is on the optimal path from router I to router K , then the optimal path

from J to K also falls along the same route. To see this, call the part of the route from I to J r_1 and the rest of the route r_2 . If a route better than r_2 existed from J to K , it could be concatenated with r_1 to improve the route from I to K , contradicting our statement that $r_1 r_2$ is optimal.

As a direct consequence of the optimality principle, we can see that the set of optimal routes from all sources to a given destination form a tree rooted at the destination. Such a tree is called a **sink tree** and is illustrated in Fig. 5-5 where the distance metric is the number of hops. Note that a sink tree is not necessarily unique; other trees with the same path lengths may exist. The goal of all routing algorithms is to discover and use the sink trees for all routers.

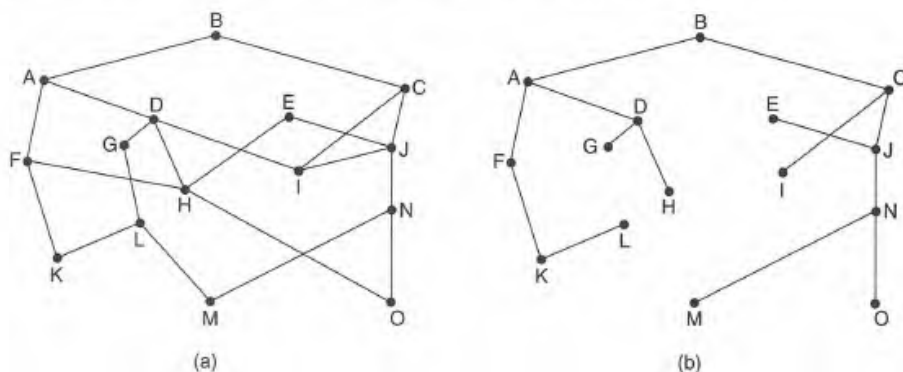


Fig. 5-5. (a) A subnet. (b) A sink tree for router B .

Since a sink tree is indeed a tree, it does not contain any loops, so each packet will be delivered within a finite and bounded number of hops. In practice, life is not quite this easy. Links and routers can go down and come back up during operation, so different routers may have different ideas about the current topology. Also, we have quietly finessed the issue of whether each router has to individually acquire the information on which to base its sink tree computation, or whether this information is collected by some other means. We will come back to these issues shortly. Nevertheless, the optimality principle and the sink tree provide a benchmark against which other routing algorithms can be measured.

In the next three sections, we will look at three different static routing algorithms. After that we will move on to adaptive ones.

5.2.2. Shortest Path Routing

Let us begin our study of routing algorithms with a technique that is widely used in many forms because it is simple and easy to understand. The idea is to build a graph of the subnet, with each node of the graph representing a router and

each arc of the graph representing a communication line (often called a link). To choose a route between a given pair of routers, the algorithm just finds the shortest path between them on the graph.

The concept of a **shortest path** deserves some explanation. One way of measuring path length is the number of hops. Using this metric, the paths *ABC* and *ABE* in Fig. 5-6 are equally long. Another metric is the geographic distance in kilometers, in which case *ABC* is clearly much longer than *ABE* (assuming the figure is drawn to scale).

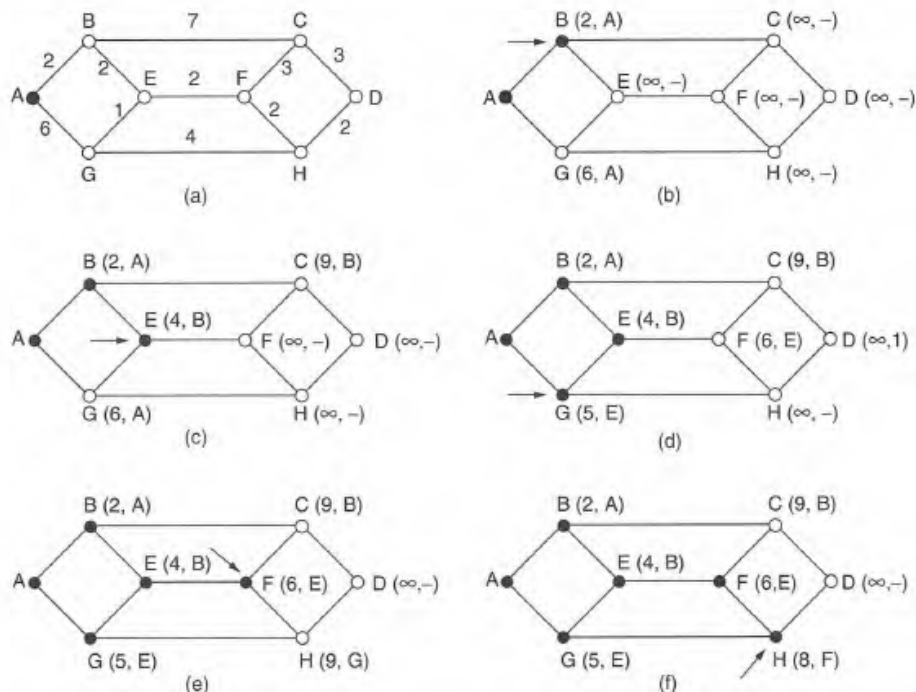


Fig. 5-6. The first five steps used in computing the shortest path from A to D. The arrows indicate the working node.

However, many other metrics are also possible besides hops and physical distance. For example, each arc could be labeled with the mean queueing and transmission delay for some standard test packet as determined by hourly test runs. With this graph labeling, the shortest path is the fastest path, rather than the path with the fewest arcs or kilometers.

In the most general case, the labels on the arcs could be computed as a function of the distance, bandwidth, average traffic, communication cost, mean queue length, measured delay, and other factors. By changing the weighting function,

the algorithm would then compute the "shortest" path measured according to any one of a number of criteria, or a combination of criteria.

Several algorithms for computing the shortest path between two nodes of a graph are known. This one is due to Dijkstra (1959). Each node is labeled (in parentheses) with its distance from the source node along the best known path. Initially, no paths are known, so all nodes are labeled with infinity. As the algorithm proceeds and paths are found, the labels may change, reflecting better paths. A label may be either tentative or permanent. Initially, all labels are tentative. When it is discovered that a label represents the shortest possible path from the source to that node, it is made permanent and never changed thereafter.

To illustrate how the labeling algorithm works, look at the weighted, undirected graph of Fig. 5-6(a), where the weights represent, for example, distance. We want to find the shortest path from *A* to *D*. We start out by marking node *A* as permanent, indicated by a filled in circle. Then we examine, in turn, each of the nodes adjacent to *A* (the working node), relabeling each one with the distance to *A*. Whenever a node is relabeled, we also label it with the node from which the probe was made, so we can reconstruct the final path later. Having examined each of the nodes adjacent to *A*, we examine all the tentatively labeled nodes in the whole graph and make the one with the smallest label permanent, as shown in Fig. 5-6(b). This one becomes the new working node.

We now start at *B*, and examine all nodes adjacent to it. If the sum of the label on *B* and the distance from *B* to the node being considered is less than the label on that node, we have a shorter path, so the node is relabeled.

After all the nodes adjacent to the working node have been inspected and the tentative labels changed if possible, the entire graph is searched for the tentatively labeled node with the smallest value. This node is made permanent and becomes the working node for the next round. Figure 5-6 shows the first five steps of the algorithm.

To see why the algorithm works, look at Fig. 5-6(c). At that point we have just made *E* permanent. Suppose that there were a shorter path than *ABE*, say *AXYZE*. There are two possibilities: either node *Z* has already been made permanent, or it has not been. If it has, then *E* has already been probed (on the round following the one when *Z* was made permanent), so the *AXYZE* path has not escaped our attention.

Now consider the case where *Z* is still tentatively labeled. Either the label at *Z* is greater than or equal to that at *E*, in which case *AXYZE* cannot be a shorter path than *ABE*, or it is less than that of *E*, in which case *Z* and not *E* will become permanent first, allowing *E* to be probed from *Z*.

This algorithm is given in Fig. 5-7. The only difference between the program and the algorithm described above is that in Fig. 5-7, we compute the shortest path starting at the terminal node, *t*, rather than at the source node, *s*. Since the shortest path from *t* to *s* in an undirected graph is the same as the shortest path from *s* to *t*, it does not matter at which end we begin (unless there are several shortest paths,

in which case reversing the search might discover a different one). The reason for searching backward is that each node is labeled with its predecessor rather than its successor. When copying the final path into the output variable, *path*, the path is thus reversed. By reversing the search, the two effects cancel, and the answer is produced in the correct order.

5.2.3. Flooding

Another static algorithm is **flooding**, in which every incoming packet is sent out on every outgoing line except the one it arrived on. Flooding obviously generates vast numbers of duplicate packets, in fact, an infinite number unless some measures are taken to damp the process. One such measure is to have a hop counter contained in the header of each packet, which is decremented at each hop, with the packet being discarded when the counter reaches zero. Ideally, the hop counter should be initialized to the length of the path from source to destination. If the sender does not know how long the path is, it can initialize the counter to the worst case, namely, the full diameter of the subnet.

An alternative technique for damming the flood is to keep track of which packets have been flooded, to avoid sending them out a second time. One way to achieve this goal is to have the source router put a sequence number in each packet it receives from its hosts. Each router then needs a list per source router telling which sequence numbers originating at that source have already been seen. If an incoming packet is on the list, it is not flooded.

To prevent the list from growing without bound, each list should be augmented by a counter, k , meaning that all sequence numbers through k have been seen. When a packet comes in, it is easy to check if the packet is a duplicate; if so, it is discarded. Furthermore, the full list below k is not needed, since k effectively summarizes it.

A variation of flooding that is slightly more practical is **selective flooding**. In this algorithm the routers do not send every incoming packet out on every line, only on those lines that are going approximately in the right direction. There is usually little point in sending a westbound packet on an eastbound line unless the topology is extremely peculiar.

Flooding is not practical in most applications, but it does have some uses. For example, in military applications, where large numbers of routers may be blown to bits at any instant, the tremendous robustness of flooding is highly desirable. In distributed database applications, it is sometimes necessary to update all the databases concurrently, in which case flooding can be useful. A third possible use of flooding is as a metric against which other routing algorithms can be compared. Flooding always chooses the shortest path, because it chooses every possible path in parallel. Consequently, no other algorithm can produce a shorter delay (if we ignore the overhead generated by the flooding process itself).

```

#define MAX_NODES 1024          /* maximum number of nodes */
#define INFINITY 1000000000     /* a number larger than every maximum path */
int n, dist[MAX_NODES][MAX_NODES]; /* dist[i][j] is the distance from i to j */

void shortest_path(int s, int t, int path[])
{ struct state {                /* the path being worked on */
    int predecessor;            /* previous node */
    int length;                 /* length from source to this node */
    enum {permanent, tentative} label; /* label state */
} state[MAX_NODES];

int i, k, min;
struct state *
    p;
for (p = &state[0]; p < &state[n]; p++) { /* initialize state */
    p->predecessor = -1;
    p->length = INFINITY;
    p->label = tentative;
}
state[t].length = 0; state[t].label = permanent;
k = t;                                /* k is the initial working node */
do {                                  /* Is there a better path from k? */
    for (i = 0; i < n; i++)           /* this graph has n nodes */
        if (dist[k][i] != 0 && state[i].label == tentative) {
            if (state[k].length + dist[k][i] < state[i].length) {
                state[i].predecessor = k;
                state[i].length = state[k].length + dist[k][i];
            }
        }

    /* Find the tentatively labeled node with the smallest label. */
    k = 0; min = INFINITY;
    for (i = 0; i < n; i++)
        if (state[i].label == tentative && state[i].length < min) {
            min = state[i].length;
            k = i;
        }
    state[k].label = permanent;
} while (k != s);

/* Copy the path into the output array. */
i = 0; k = s;
do {path[i++] = k; k = state[k].predecessor; } while (k >= 0);
}

```

Fig. 5-7. Dijkstra's algorithm to compute the shortest path through a graph.

5.2.4. Flow-Based Routing

The algorithms studied so far take only the topology into account. They do not consider the load. If, for example, there is always a huge amount of traffic from *A* to *B*, in Fig. 5-6, then it may be better to route traffic from *A* to *C* via *AGEFC*, even though this path is much longer than *ABC*. In this section we will study a static algorithm that uses both topology and load for routing. It is called **flow-based routing**.

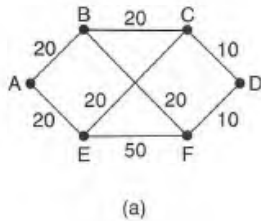
In some networks, the mean data flow between each pair of nodes is relatively stable and predictable. For example, in a corporate network for a retail store chain, each store might send orders, sales reports, inventory updates, and other well-defined types of messages to known sites in a predefined pattern, so that the total volume of traffic varies little from day to day. Under conditions in which the average traffic from *i* to *j* is known in advance and, to a reasonable approximation, constant in time, it is possible to analyze the flows mathematically to optimize the routing.

The basic idea behind the analysis is that for a given line, if the capacity and average flow are known, it is possible to compute the mean packet delay on that line from queueing theory. From the mean delays on all the lines, it is straightforward to calculate a flow-weighted average to get the mean packet delay for the whole subnet. The routing problem then reduces to finding the routing algorithm that produces the minimum average delay for the subnet.

To use this technique, certain information must be known in advance. First the subnet topology must be known. Second, the traffic matrix, F_{ij} , must be given. Third, the line capacity matrix, C_{ij} , specifying the capacity of each line in bps must be available. Finally, a (possibly tentative) routing algorithm must be chosen.

As an example of this method, consider the full-duplex subnet of Fig. 5-8(a). The weights on the arcs give the capacities, C_{ij} , in each direction measured in kbps. The matrix of Fig. 5-8(b) has an entry for each source-destination pair. The entry for source *i* to destination *j* shows the route to be used for *i-j* traffic, and also the number of packets/sec to be sent from source *i* to destination *j*. For example, 3 packets/sec go from *B* to *D*, and they use route *BFD* to get there. Notice that some routing algorithm has already been applied to derive the routes shown in the matrix.

Given this information, it is straightforward to calculate the total in line *i*, λ_i . For example, the *B-D* traffic contributes 3 packets/sec to the *BF* line and also 3 packets/sec to the *FD* line. Similarly, the *A-D* traffic contributes 1 packet/sec to each of three lines. The total traffic in each eastbound line is shown in the λ_i column of Fig. 5-9. In this example, all the traffic is symmetric, that is, the *XY* traffic is identical to the *YX* traffic, for all *X* and *Y*. In real networks this condition does not always hold. The figure also shows the mean number of packets/sec on each line, μC_i assuming a mean packet size of $1/\mu = 800$ bits.



		Destination					
		A	B	C	D	E	F
Source	A		9 AB	4 ABC	1 ABFD	7 AE	4 AEF
	B	9 BA		8 BC	3 BFD	2 BFE	4 BF
	C	4 CBA	8 CB		3 CD	3 CE	2 CEF
	D	1 DFBA	3 DFB	3 DC		3 DCE	4 DF
	E	7 EA	2 EFB	3 EC	3 ECD		5 EF
	F	4 FEA	4 FB	2 FEC	4 FD	5 FE	

(b)

Fig. 5-8. (a) A subnet with line capacities shown in kbps. (b) The traffic in packets/sec and the routing matrix.

The next-to-last column of Fig. 5-9 gives the mean delay for each line derived from the queueing theory formula

$$T = \frac{1}{\mu C - \lambda}$$

where $1/\mu$ is the mean packet size in bits, C is the capacity in bps, and λ is the mean flow in packets/sec. For example, with a capacity $\mu C = 25$ packets/sec and an actual flow $\lambda = 14$ packets/sec, the mean delay is 91 msec. Note that with $\lambda = 0$, the mean delay is still 40 msec, because the capacity is 25 packets/sec. In other words, the “delay” includes both queueing and service time.

To compute the mean delay time for the entire subnet, we take the weighted sum of each of the eight lines, with the weight being the fraction of the total traffic using that line. In this example, the mean turns out to be 86 msec.

To evaluate a different routing algorithm, we can repeat the entire process, only with different flows to get a new average delay. If we restrict ourselves to only single path routing algorithms, as we have done so far, there are only a finite number of ways to route packets from each source to each destination. It is always possible to write a program to simply try them all, one after another, and find out which one has the smallest mean delay. Since this calculation can be done off-line in advance, the fact that it may be time consuming is not necessarily a serious problem. This one is then the best routing algorithm. Bertsekas and Gallager (1992) discuss flow-based routing in detail.

i	Line	λ_i (pkts/sec)	C_i (kbps)	μC_i (pkts/sec)	T_i (msec)	Weight
1	AB	14	20	25	91	0.171
2	BC	12	20	25	77	0.146
3	CD	6	10	12.5	154	0.073
4	AE	11	20	25	71	0.134
5	EF	13	50	62.5	20	0.159
6	FD	8	10	12.5	222	0.098
7	BF	10	20	25	67	0.122
8	EC	8	20	25	59	0.098

Fig. 5-9. Analysis of the subnet of Fig. 5-8 using a mean packet size of 800 bits. The reverse traffic (*BA*, *CB*, etc.) is the same as the forward traffic.

5.2.5. Distance Vector Routing

Modern computer networks generally use dynamic routing algorithms rather than the static ones described above. Two dynamic algorithms in particular, distance vector routing and link state routing, are the most popular. In this section we will look at the former algorithm. In the following one we will study the latter one.

Distance vector routing algorithms operate by having each router maintain a table (i.e., a vector) giving the best known distance to each destination and which line to use to get there. These tables are updated by exchanging information with the neighbors.

The distance vector routing algorithm is sometimes called by other names, including the distributed **Bellman-Ford** routing algorithm and the **Ford-Fulkerson** algorithm, after the researchers who developed it (Bellman, 1957; and Ford and Fulkerson, 1962). It was the original ARPANET routing algorithm and was also used in the Internet under the name RIP and in early versions of DECnet and Novell's IPX. AppleTalk and Cisco routers use improved distance vector protocols.

In distance vector routing, each router maintains a routing table indexed by, and containing one entry for, each router in the subnet. This entry contains two parts: the preferred outgoing line to use for that destination, and an estimate of the time or distance to that destination. The metric used might be number of hops, time delay in milliseconds, total number of packets queued along the path, or something similar.

The router is assumed to know the "distance" to each of its neighbors. If the metric is hops, the distance is just one hop. If the metric is queue length, the router simply examines each queue. If the metric is delay, the router can measure

it directly with special ECHO packets that the receiver just timestamps and sends back as fast as it can.

As an example, assume that delay is used as a metric and that the router knows the delay to each of its neighbors. Once every T msec each router sends to each neighbor a list of its estimated delays to each destination. It also receives a similar list from each neighbor. Imagine that one of these tables has just come in from neighbor X , with X_i being X 's estimate of how long it takes to get to router i . If the router knows that the delay to X is m msec, it also knows that it can reach router i via X in $X_i + m$ msec via X . By performing this calculation for each neighbor, a router can find out which estimate seems the best and use that estimate and the corresponding line in its new routing table. Note that the old routing table is not used in the calculation.

This updating process is illustrated in Fig. 5-10. Part (a) shows a subnet. The first four columns of part (b) show the delay vectors received from the neighbors of router J . A claims to have a 12-msec delay to B , a 25-msec delay to C , a 40-msec delay to D , etc. Suppose that J has measured or estimated its delay to its neighbors, A , I , H , and K as 8, 10, 12, and 6 msec, respectively.

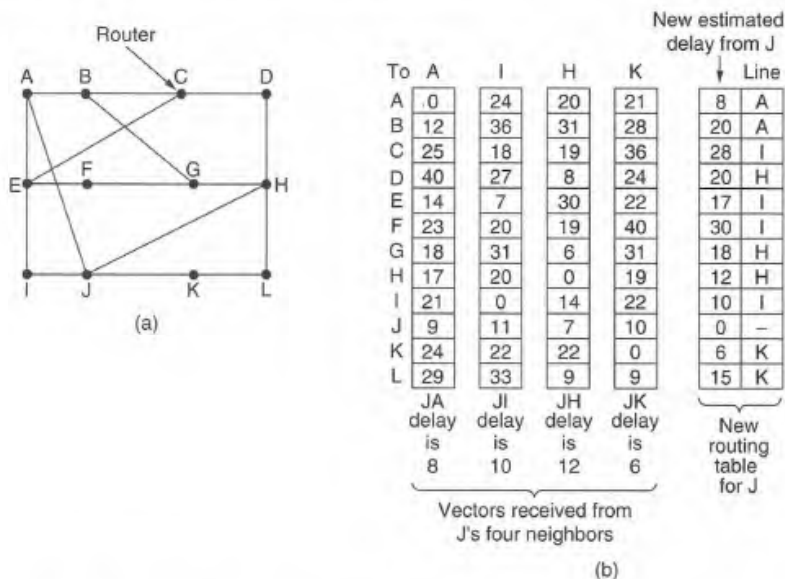


Fig. 5-10. (a) A subnet. (b) Input from A , I , H , K , and the new routing table for J .

Consider how J computes its new route to router G . It knows that it can get to A in 8 msec, and A claims to be able to get to G in 18 msec, so J knows it can count on a delay of 26 msec to G if it forwards packets bound for G to A .

Similarly, it computes the delay to *G* via *I*, *H*, and *K* as 41 (31 + 10), 18 (6 + 12), and 37 (31 + 6) msec respectively. The best of these values is 18, so it makes an entry in its routing table that the delay to *G* is 18 msec, and that the route to use is via *H*. The same calculation is performed for all the other destinations, with the new routing table shown in the last column of the figure.

The Count-to-Infinity Problem

Distance vector routing works in theory but has a serious drawback in practice: although it converges to the correct answer, it may do so slowly. In particular, it reacts rapidly to good news, but leisurely to bad news. Consider a router whose best route to destination *X* is large. If on the next exchange neighbor *A* suddenly reports a short delay to *X*, the router just switches over to using the line to *A* to send traffic to *X*. In one vector exchange, the good news is processed.

To see how fast good news propagates, consider the five-node (linear) subnet of Fig. 5-11, where the delay metric is the number of hops. Suppose *A* is down initially and all the other routers know this. In other words, they have all recorded the delay to *A* as infinity.

A	B	C	D	E	
•	•	•	•	•	Initially
	∞	∞	∞	∞	After 1 exchange
	1	∞	∞	∞	After 2 exchanges
	1	2	∞	∞	After 3 exchanges
	1	2	3	∞	After 4 exchanges
	1	2	3	4	After 5 exchanges
					(a)

A	B	C	D	E	
•	•	•	•	•	Initially
	1	2	3	4	After 1 exchange
	3	2	3	4	After 2 exchanges
	3	4	3	4	After 3 exchanges
	5	4	5	4	After 4 exchanges
	5	6	5	6	After 5 exchanges
	7	6	7	6	After 6 exchanges
	7	8	7	8	After 7 exchanges
					⋮
	∞	∞	∞	∞	(b)

Fig. 5-11. The count-to-infinity problem.

When *A* comes up, the other routers learn about it via the vector exchanges. For simplicity we will assume that there is a gigantic gong somewhere that is struck periodically to initiate a vector exchange at all routers simultaneously. At the time of the first exchange, *B* learns that its left neighbor has zero delay to *A*. *B* now makes an entry in its routing table that *A* is one hop away to the left. All the other routers still think that *A* is down. At this point, the routing table entries for *A* are as shown in the second row of Fig. 5-11(a). On the next exchange, *C* learns that *B* has a path of length 1 to *A*, so it updates its routing table to indicate a path of length 2, but *D* and *E* do not hear the good news until later. Clearly, the good news is spreading at the rate of one hop per exchange. In a subnet whose longest

path is of length N hops, within N exchanges everyone will know about newly revived lines and routers.

Now let us consider the situation of Fig. 5-11(b), in which all the lines and routers are initially up. Routers B , C , D , and E have distances to A of 1, 2, 3, and 4, respectively. Suddenly A goes down, or alternatively, the line between A and B is cut, which is effectively the same thing from B 's point of view.

At the first packet exchange, B does not hear anything from A . Fortunately, C says "Do not worry. I have a path to A of length 2." Little does B know that C 's path runs through B itself. For all B knows, C might have ten outgoing lines all with independent paths to A of length 2. As a result, B now thinks it can reach A via C , with a path length of 3. D and E do not update their entries for A on the first exchange.

On the second exchange, C notices that each of its neighbors claims to have a path to A of length 3. It picks one of the them at random and makes its new distance to A 4, as shown in the third row of Fig. 5-11(b). Subsequent exchanges produce the history shown in the rest of Fig. 5-11(b).

From this figure, it should be clear why bad news travels slowly: no router ever has a value more than one higher than the minimum of all its neighbors. Gradually, all the routers work their way up to infinity, but the number of exchanges required depends on the numerical value used for infinity. For this reason, it is wise to set infinity to the longest path plus 1. If the metric is time delay, there is no well-defined upper bound, so a high value is needed to prevent a path with a long delay from being treated as down. Not entirely surprisingly, this problem is known as the **count-to-infinity** problem.

The Split Horizon Hack

Many ad hoc solutions to the count-to-infinity problem have been proposed in the literature, each one more complicated and less useful than the one before it. We will describe just one of them here and then tell why it, too, fails. The **split horizon** algorithm works the same way as distance vector routing, except that the distance to X is not reported on the line that packets for X are sent on (actually, it is reported as infinity). In the initial state of Fig. 5-11(b), for example, C tells D the truth about the distance to A , but C tells B that its distance to A is infinite. Similarly, D tells the truth to E but lies to C .

Now let us see what happens when A goes down. On the first exchange, B discovers that the direct line is gone, and C is reporting an infinite distance to A as well. Since neither of its neighbors can get to A , B sets its distance to infinity as well. On the next exchange, C hears that A is unreachable from both of its neighbors, so it marks A as unreachable too. Using split horizon, the bad news propagates one hop per exchange. This rate is much better than without split horizon.

The real bad news is that split horizon, although widely used, sometimes fails.

Consider, for example, the four-node subnet of Fig. 5-12. Initially, both *A* and *B* have a distance 2 to *D*, and *C* has a distance 1 there.

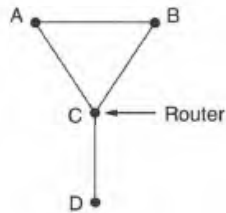


Fig. 5-12. An example where split horizon fails.

Now suppose that the *CD* line goes down. Using split horizon, both *A* and *B* tell *C* that they cannot get to *D*. Thus *C* immediately concludes that *D* is unreachable and reports this to both *A* and *B*. Unfortunately, *A* hears that *B* has a path of length 2 to *D*, so it assumes it can get to *D* via *B* in 3 hops. Similarly, *B* concludes it can get to *D* via *A* in 3 hops. On the next exchange, they each set their distance to *D* to 4. Both of them gradually count to infinity, precisely the behavior we were trying to avoid.

5.2.6. Link State Routing

Distance vector routing was used in the ARPANET until 1979, when it was replaced by link state routing. Two primary problems caused its demise. First, since the delay metric was queue length, it did not take line bandwidth into account when choosing routes. Initially, all the lines were 56 kbps, so line bandwidth was not an issue, but after some lines had been upgraded to 230 kbps and others to 1.544 Mbps, not taking bandwidth into account was a major problem. Of course, it would have been possible to change the delay metric to factor in line bandwidth, but a second problem also existed, namely, the algorithm often took too long to converge, even with tricks like split horizon. For these reasons, it was replaced by an entirely new algorithm now called **link state routing**. Variants of link state routing are now widely used.

The idea behind link state routing is simple and can be stated as five parts. Each router must

1. Discover its neighbors and learn their network addresses.
2. Measure the delay or cost to each of its neighbors.
3. Construct a packet telling all it has just learned.
4. Send this packet to all other routers.
5. Compute the shortest path to every other router.

In effect, the complete topology and all delays are experimentally measured and distributed to every router. Then Dijkstra's algorithm can be used to find the shortest path to every other router. Below we will consider each of these five steps in more detail.

Learning about the Neighbors

When a router is booted, its first task is to learn who its neighbors are. It accomplishes this goal by sending a special HELLO packet on each point-to-point line. The router on the other end is expected to send back a reply telling who it is. These names must be globally unique because when a distant router later hears that three routers are all connected to *F*, it is essential that it can determine whether or not all three mean the same *F*.

When two or more routers are connected by a LAN, the situation is slightly more complicated. Fig. 5-13(a) illustrates a LAN to which three routers, *A*, *C*, and *F*, are directly connected. Each of these routers is connected to one or more additional routers, as shown.

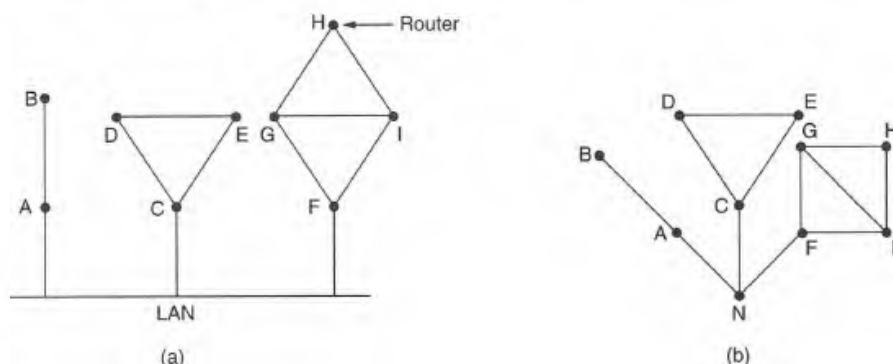


Fig. 5-13. (a) Nine routers and a LAN. (b) A graph model of (a).

One way to model the LAN is to consider it as a node itself, as shown in Fig. 5-13(b). Here we have introduced a new, artificial node, *N*, to which *A*, *C*, and *F* are connected. The fact that it is possible to go from *A* to *C* on the LAN is represented by the path *ANC* here.

Measuring Line Cost

The link state routing algorithm requires each router to know, or at least have a reasonable estimate, of the delay to each of its neighbors. The most direct way to determine this delay is to send a special ECHO packet over the line that the other

side is required to send back immediately. By measuring the round-trip time and dividing it by two, the sending router can get a reasonable estimate of the delay. For even better results, the test can be conducted several times, and the average used.

An interesting issue is whether or not to take the load into account when measuring the delay. To factor the load in, the round-trip timer must be started when the ECHO packet is queued. To ignore the load, the timer should be started when the ECHO packet reaches the front of the queue.

Arguments can be made both ways. Including traffic-induced delays in the measurements means that when a router has a choice between two lines with the same bandwidth, one of which is heavily loaded all the time and one of which is not, it will regard the route over the unloaded line as a shorter path. This choice will result in better performance.

Unfortunately, there is also an argument against including the load in the delay calculation. Consider the subnet of Fig. 5-14, which is divided up into two parts, East and West, connected by two lines, *CF* and *EI*. Suppose that most of the traffic between East and West is using line *CF*, and as a result, this line is heavily loaded with long delays. Including queueing delay in the shortest path calculation will make *EI* more attractive. After the new routing tables have been installed, most of the East-West traffic will now go over *EI*, overloading this line. Consequently, in the next update, *CF* will appear to be the shortest path. As a result, the routing tables may oscillate wildly, leading to erratic routing and many potential problems. If load is ignored and only bandwidth is considered, this problem does not occur. Alternatively, the load can be spread over both lines, but this solution does not fully utilize the best path.

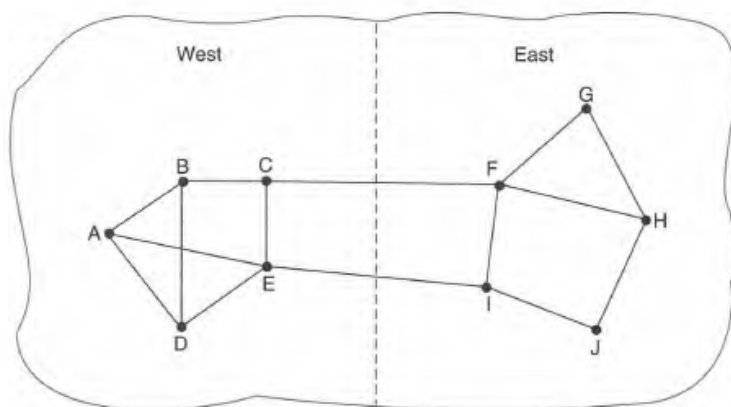


Fig. 5-14. A subnet in which the East and West parts are connected by two lines.

Building Link State Packets

Once the information needed for the exchange has been collected, the next step is for each router to build a packet containing all the data. The packet starts with the identity of the sender, followed by a sequence number and age (to be described later), and a list of neighbors. For each neighbor, the delay to that neighbor is given. An example subnet is given in Fig. 5-15(a) with delays shown in the lines. The corresponding link state packets for all six routers are shown in Fig. 5-15(b).

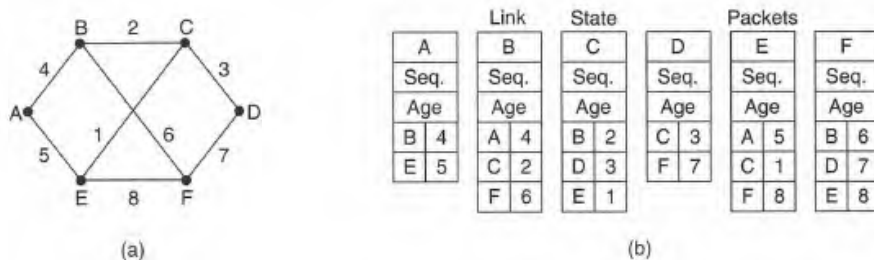


Fig. 5-15. (a) A subnet. (b) The link state packets for this subnet.

Building the link state packets is easy. The hard part is determining when to build them. One possibility is to build them periodically, that is, at regular intervals. Another possibility is when some significant event occurs, such as a line or neighbor going down or coming back up again, or changing its properties appreciably.

Distributing the Link State Packets

The trickiest part of the algorithm is distributing the link state packets reliably. As the packets are distributed and installed, the routers getting the first ones will change their routes. Consequently, the different routers may be using different versions of the topology, which can lead to inconsistencies, loops, unreachable machines, and other problems.

First we will describe the basic distribution algorithm. Later we will give some refinements. The fundamental idea is to use flooding to distribute the link state packets. To keep the flood in check, each packet contains a sequence number that is incremented for each new packet sent. Routers keep track of all the (source router, sequence) pairs they see. When a new link state packet comes in, it is checked against the list of packets already seen. If it is new, it is forwarded on all lines except the one it arrived on. If it is a duplicate, it is discarded.

If a packet with a sequence number lower than the highest one seen so far ever arrives, it is rejected as being obsolete.

This algorithm has a few problems, but they are manageable. First, if the sequence numbers wrap around, confusion will reign. The solution here is to use a 32-bit sequence number. With one link state packet per second, it would take 137 years to wrap around, so this possibility can be ignored.

Second, if a router ever crashes, it will lose track of its sequence number. If it starts again at 0, the next packet will be rejected as a duplicate.

Third, if a sequence number is ever corrupted and 65,540 is received instead of 4 (a 1-bit error), packets 5 through 65,540 will be rejected as obsolete, since the current sequence number is thought to be 65,540.

The solution to all these problems is to include the age of each packet after the sequence number and decrement it once per second. When the age hits zero, the information from that router is discarded. Normally, a new packet comes in, say, every 10 minutes, so router information only times out when a router is down (or six consecutive packets have been lost, an unlikely event). The age field is also decremented by each router during the initial flooding process, to make sure no packet can get lost and live for an indefinite period of time (a packet whose age is zero is discarded).

Some refinements to this algorithm make it more robust. When a link state packet comes in to a router for flooding, it is not queued for transmission immediately. Instead it is put in a holding area to wait a short while first. If another link state packet from the same source comes in before it is transmitted, their sequence numbers are compared. If they are equal, the duplicate is discarded. If they are different, the older one is thrown out. To guard against errors on the router-router lines, all link state packets are acknowledged. When a line goes idle, the holding area is scanned in round robin order to select a packet or acknowledgement to send.

The data structure used by router *B* for the subnet shown in Fig. 5-15(a) is depicted in Fig. 5-16. Each row here corresponds to a recently arrived, but as yet not fully processed, link state packet. The table records where the packet originated, its sequence number and age, and the data. In addition, there are send and acknowledgement flags for each of *B*'s three lines (to *A*, *C*, and *F*, respectively). The send flags mean that the packet must be sent on the indicated line. The acknowledgement flags mean that it must be acknowledged there.

In Fig. 5-16, the link state packet from *A* arrived directly, so it must be sent to *C* and *F* and acknowledged to *A*, as indicated by the flag bits. Similarly, the packet from *F* has to be forwarded to *A* and *C* and acknowledged to *F*.

However, the situation with the third packet, from *E*, is different. It arrived twice, once via *EAB* and once via *EFB*. Consequently, it has to be sent only to *C*, but acknowledged to both *A* and *F*, as indicated by the bits.

If a duplicate arrives while the original is still in the buffer, bits have to be changed. For example, if a copy of *C*'s state arrives from *F* before the fourth

Source	Seq.	Age	Send flags			ACK flags			Data
			A	C	F	A	C	F	
A	21	60	0	1	1	1	0	0	
F	21	60	1	1	0	0	0	1	
E	21	59	0	1	0	1	0	1	
C	20	60	1	0	1	0	1	0	
D	21	59	1	0	0	0	1	1	

Fig. 5-16. The packet buffer for router *B* in Fig. 5-15.

entry in the table has been forwarded, the six bits will be changed to 100011 to indicate that the packet must be acknowledged to *F* but not sent there.

Computing the New Routes

Once a router has accumulated a full set of link state packets, it can construct the entire subnet graph because every link is represented. Every link is, in fact, represented twice, once for each direction. The two values can be averaged or used separately.

Now Dijkstra's algorithm can be run locally to construct the shortest path to all possible destinations. The results of this algorithm can be installed in the routing tables, and normal operation resumed.

For a subnet with n routers, each of which has k neighbors, the memory required to store the input data is proportional to kn . For large subnets, this can be a problem. Also, the computation time can be an issue. Nevertheless, in many practical situations, link state routing works well.

However, problems with the hardware or software can wreak havoc with this algorithm (also with other ones). For example, if a router claims to have a line it does not have, or forgets a line it does have, the subnet graph will be incorrect. If a router fails to forward packets, or corrupts them while forwarding them, trouble will arise. Finally, if it runs out of memory or does the routing calculation wrong, bad things will happen. As the subnet grows into the range of tens or hundreds of thousands of nodes, the probability of some router failing occasionally becomes nonnegligible. The trick is to try to arrange to limit the damage when the inevitable happens. Perlman (1988) discusses these problems and their solutions in detail.

Link state routing is widely used in actual networks, so a few words about some example protocols using it are in order. The OSPF protocol, which is

increasingly being used in the Internet, uses a link state algorithm. We will describe OSPF in Sec. 5.5.5.

Another important link state protocol is **IS-IS (Intermediate System-Intermediate System)**, which was designed for DECnet and later adopted by ISO for use with its connectionless network layer protocol, CLNP. Since then it has been modified to handle other protocols as well, most notably, IP. IS-IS is used in numerous Internet backbones (including the old NSFNET backbone), and in some digital cellular systems such as CDPD. Novell NetWare uses a minor variant of IS-IS (NLSP) for routing IPX packets.

Basically IS-IS distributes a picture of the router topology, from which the shortest paths are computed. Each router announces, in its link state information, which network layer addresses it can reach directly. These addresses can be IP, IPX, AppleTalk, or any other addresses. IS-IS can even support multiple network layer protocols at the same time.

Many of the innovations designed for IS-IS were adopted by OSPF (OSPF was designed several years after IS-IS). These include a self-stabilizing method of flooding link state updates, the concept of a designated router on a LAN, and the method of computing and supporting path splitting and multiple metrics. As a consequence, there is very little difference between IS-IS and OSPF. The most important difference is that IS-IS is encoded in such a way that it is easy and natural to simultaneously carry information about multiple network layer protocols, a feature OSPF does not have. This advantage is especially valuable in large multiprotocol environments.

5.2.7. Hierarchical Routing

As networks grow in size, the router routing tables grow proportionally. Not only is router memory consumed by ever increasing tables, but more CPU time is needed to scan them and more bandwidth is needed to send status reports about them. At a certain point the network may grow to the point where it is no longer feasible for every router to have an entry for every other router, so the routing will have to be done hierarchically, as it is in the telephone network.

When hierarchical routing is used, the routers are divided into what we will call **regions**, with each router knowing all the details about how to route packets to destinations within its own region, but knowing nothing about the internal structure of other regions. When different networks are connected together, it is natural to regard each one as a separate region in order to free the routers in one network from having to know the topological structure of the other ones.

For huge networks, a two-level hierarchy may be insufficient; it may be necessary to group the regions into clusters, the clusters into zones, the zones into groups, and so on, until we run out of names for aggregations. As an example of a multilevel hierarchy, consider how a packet might be routed from Berkeley, California to Malindi, Kenya. The Berkeley router would know the detailed topology

within California but would send all out-of-state traffic to the Los Angeles router. The Los Angeles router would be able to route traffic to other domestic routers, but would send foreign traffic to New York. The New York router would be programmed to direct all traffic to the router in the destination country responsible for handling foreign traffic, say in Nairobi. Finally, the packet would work its way down the tree in Kenya until it got to Malindi.

Figure 5-17 gives a quantitative example of routing in a two-level hierarchy with five regions. The full routing table for router 1A has 17 entries, as shown in Fig. 5-17(b). When routing is done hierarchically, as in Fig. 5-17(c), there are entries for all the local routers as before, but all other regions have been condensed into a single router, so all traffic for region 2 goes via the 1B–2A line, but the rest of the remote traffic goes via the 1C–3B line. Hierarchical routing has reduced the table from 17 to 7 entries. As the ratio of the number of regions to the number of routers per region grows, the savings in table space increase.

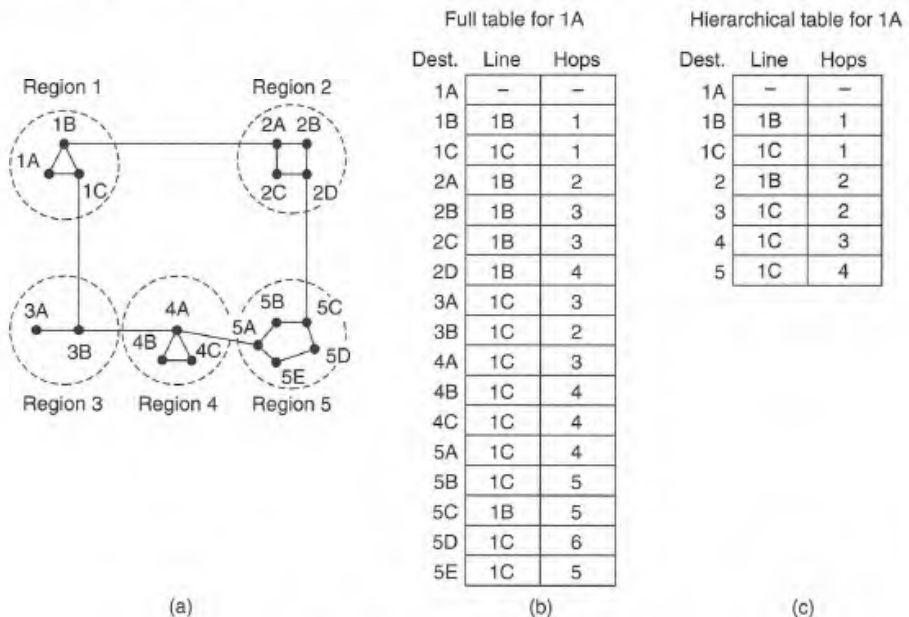


Fig. 5-17. Hierarchical routing.

Unfortunately, these gains in space are not free. There is a penalty to be paid, and this penalty is in the form of increased path length. For example, the best route from 1A to 5C is via region 2, but with hierarchical routing all traffic to region 5 goes via region 3, because that is better for most destinations in region 5.

When a single network becomes very large, an interesting question is: How many levels should the hierarchy have? For example, consider a subnet with 720

routers. If there is no hierarchy, each router needs 720 routing table entries. If the subnet is partitioned into 24 regions of 30 routers each, each router needs 30 local entries plus 23 remote entries for a total of 53 entries. If a three-level hierarchy is chosen, with eight clusters, each containing 9 regions of 10 routers, each router needs 10 entries for local routers, 8 entries for routing to other regions within its own cluster, and 7 entries for distant clusters, for a total of 25 entries. Kamoun and Kleinrock (1979) have discovered that the optimal number of levels for an N router subnet is $\ln N$, requiring a total of $e \ln N$ entries per router. They have also shown that the increase in effective mean path length caused by hierarchical routing is sufficiently small that it is usually acceptable.

5.2.8. Routing for Mobile Hosts

Millions of people have portable computers nowadays, and they generally want to read their email and access their normal file systems wherever in the world they may be. These mobile hosts introduce a new complication: to route a packet to a mobile host, the network first has to find it. The subject of incorporating mobile hosts into a network is very young, but in this section we will sketch some of the issues here and give a possible solution.

The model of the world that network designers typically use is shown in Fig. 5-18. Here we have a WAN consisting of routers and hosts. Connected to the WAN are LANs, MANs, and wireless cells of the type we studied in Chap. 2.

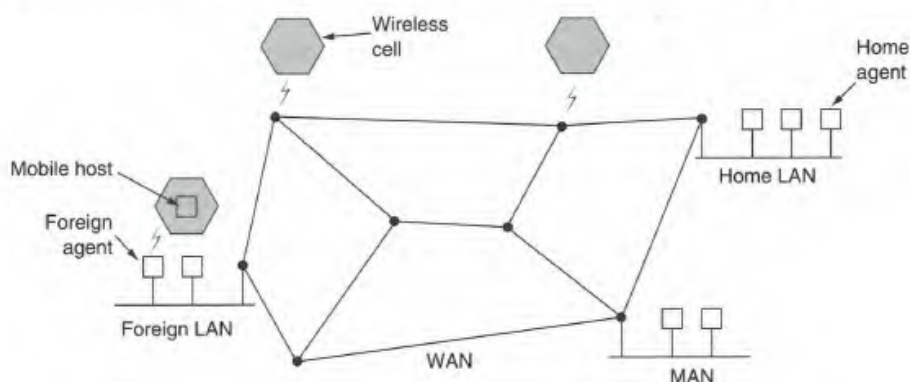


Fig. 5-18. A WAN to which LANs, MANs, and wireless cells are attached.

Users who never move are said to be stationary. They are connected to the network by copper wires or fiber optics. In contrast, we can distinguish two other kinds of users. Migratory users are basically stationary users who move from one fixed site to another from time to time but use the network only when they are

physically connected to it. Roaming users actually compute on the run and want to maintain their connections as they move around. We will use the term **mobile users** to mean either of the latter two categories, that is, all users who are away from home.

All users are assumed to have a permanent **home location** that never changes. Users also have a permanent home address that can be used to determine their home locations, analogous to the way the telephone number 1-212-5551212 indicates the United States (country code 1) and Manhattan (212). The routing goal in systems with mobile users is to make it possible to send packets to mobile users using their home addresses, and have the packets efficiently reach them wherever they may be. The trick, of course, is to find them.

In the model of Fig. 5-18, the world is divided up (geographically) into small units. Let us call them areas, where an area is typically a LAN or wireless cell. Each area has one or more **foreign agents**, which keep track of all mobile users visiting the area. In addition, each area has a **home agent**, which keeps track of users whose home is in the area, but who are currently visiting another area.

When a new user enters an area, either by connecting to it (e.g., plugging into the LAN), or just wandering into the cell, his computer must register itself with the foreign agent there. The registration procedure typically works like this:

1. Periodically, each foreign agent broadcasts a packet announcing its existence and address. A newly arrived mobile host may wait for one of these messages, but if none arrives quickly enough, the mobile host can broadcast a packet saying: "Are there any foreign agents around?"
2. The mobile host registers with the foreign agent, giving its home address, current data link layer address, and some security information.
3. The foreign agent contacts the mobile host's home agent and says: "One of your hosts is over here." The message from the foreign agent to the home agent contains the foreign agent's network address. It also includes the security information, to convince the home agent that the mobile host is really there.
4. The home agent examines the security information, which contains a timestamp, to prove that it was generated within the past few seconds. If it is happy, it tells the foreign agent to proceed.
5. When the foreign agent gets the acknowledgement from the home agent, it makes an entry in its tables and informs the mobile host that it is now registered.

Ideally, when a user leaves an area, that, too, should be announced to allow deregistration, but many users abruptly turn off their computers when done.

When a packet is sent to a mobile user, it is routed to the user's home LAN because that is what the address says should be done, as illustrated in step 1 of Fig. 5-19. Packets sent to the mobile user on its home LAN are intercepted by the home agent. The home agent then looks up the mobile user's new (temporary) location and finds the address of the foreign agent handling the mobile user. The home agent then does two things. First, it encapsulates the packet in the payload field of an outer packet and sends the latter to the foreign agent (step 2 in Fig. 5-19). This mechanism is called tunneling; we will look at it in more detail later. After getting the encapsulated packet, the foreign agent removes the original packet from the payload field and sends it to the mobile user as a data link frame.

Second, the home agent tells the sender to henceforth send packets to the mobile host by encapsulating them in the payload of packets explicitly addressed to the foreign agent, instead of just sending them to the mobile user's home address (step 3). Subsequent packets can now be routed directly to the user via the foreign agent (step 4), bypassing the home location entirely.

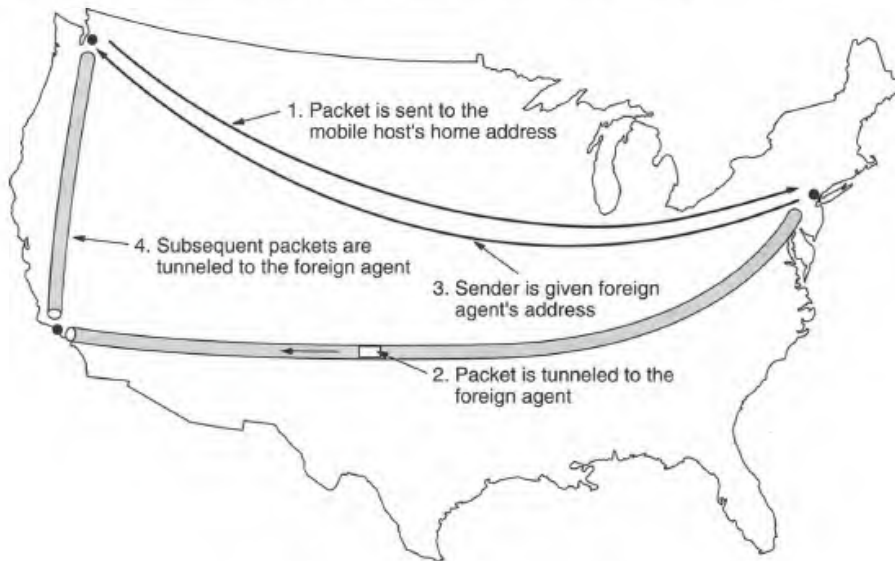


Fig. 5-19. Packet routing for mobile users.

The various schemes that have been proposed differ in several ways. First, there is the issue of how much of this protocol is carried out by the routers and how much by the hosts, and in the latter case, by which layer in the hosts. Second, in a few schemes, routers along the way record mapped addresses so they can intercept and redirect traffic even before it gets to the home location. Third, in some schemes each visitor is given a unique temporary address; in others, the temporary address refers to an agent that handles traffic for all visitors.

Fourth, the schemes differ in how they actually manage to arrange for packets that are addressed to one destination to be delivered to a different one. One choice is changing the destination address and just retransmitting the modified packet. Alternatively, the whole packet, home address and all, can be encapsulated inside the payload of another packet sent to the temporary address. Finally, the schemes differ in their security aspects. In general, when a host or router gets a message of the form "Starting right now, please send all of Cayla's mail to me," it might have a couple of questions about whom it was talking to and whether or not this is a good idea. Several mobile host protocols are discussed and compared in (Ioannidis and Maguire, 1993; Myles and Skellern, 1993; Perkins, 1993; Teraoka et al., 1993; and Wada et al., 1993).

5.2.9. Broadcast Routing

For some applications, hosts need to send messages to many or all other hosts. For example, a service distributing weather reports, stock market updates, or live radio programs might work best by broadcasting to all machines and letting those that are interested read the data. Sending a packet to all destinations simultaneously is called **broadcasting**; various methods have been proposed for doing it.

One broadcasting method that requires no special features from the subnet is for the source to simply send a distinct packet to each destination. Not only is the method wasteful of bandwidth, but it also requires the source to have a complete list of all destinations. In practice this may be the only possibility, but it is the least desirable of the methods.

Flooding is another obvious candidate. Although flooding is ill-suited for ordinary point-to-point communication, for broadcasting it might rate serious consideration, especially if none of the methods described below are applicable. The problem with flooding as a broadcast technique is the same problem it has as a point-to-point routing algorithm: it generates too many packets and consumes too much bandwidth.

A third algorithm is **multidestination routing**. If this method is used, each packet contains either a list of destinations or a bit map indicating the desired destinations. When a packet arrives at a router, the router checks all the destinations to determine the set of output lines that will be needed. (An output line is needed if it is the best route to at least one of the destinations.) The router generates a new copy of the packet for each output line to be used and includes in each packet only those destinations that are to use the line. In effect, the destination set is partitioned among the output lines. After a sufficient number of hops, each packet will carry only one destination and can be treated as a normal packet. Multidestination routing is like separately addressed packets, except that when several packets must follow the same route, one of them pays full fare and the rest ride free.

A fourth broadcast algorithm makes explicit use of the sink tree for the router initiating the broadcast, or any other convenient spanning tree for that matter. A

spanning tree is a subset of the subnet that includes all the routers but contains no loops. If each router knows which of its lines belong to the spanning tree, it can copy an incoming broadcast packet onto all the spanning tree lines except the one it arrived on. This method makes excellent use of bandwidth, generating the absolute minimum number of packets necessary to do the job. The only problem is that each router must have knowledge of some spanning tree for it to be applicable. Sometimes this information is available (e.g., with link state routing) but sometimes it is not (e.g., with distance vector routing).

Our last broadcast algorithm is an attempt to approximate the behavior of the previous one, even when the routers do not know anything at all about spanning trees. The idea is remarkably simple once it has been pointed out. When a broadcast packet arrives at a router, the router checks to see if the packet arrived on the line that is normally used for sending packets *to* the source of the broadcast. If so, there is an excellent chance that the broadcast packet itself followed the best route from the router and is therefore the first copy to arrive at the router. This being the case, the router forwards copies of it onto all lines except the one it arrived on. If, however, the broadcast packet arrived on a line other than the preferred one for reaching the source, the packet is discarded as a likely duplicate.

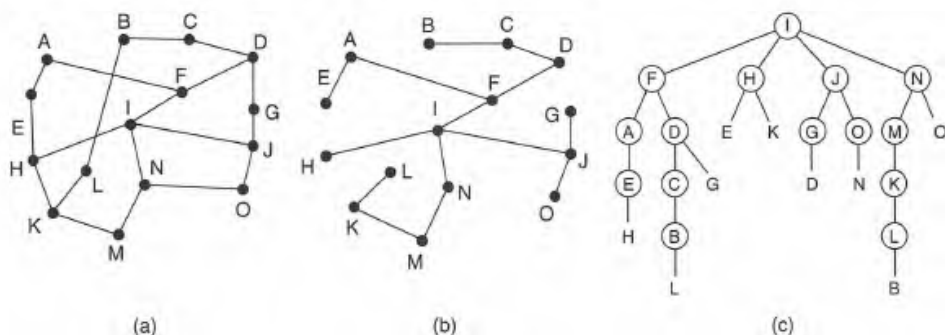


Fig. 5-20. Reverse path forwarding. (a) A subnet. (b) A spanning tree. (c) The tree built by reverse path forwarding.

An example of the algorithm, called **reverse path forwarding**, is shown in Fig. 5-20. Part (a) shows a subnet, part (b) shows a sink tree for router *I* of that subnet, and part (c) shows how the reverse path algorithm works. On the first hop, *I* sends packets to *F*, *H*, *J*, and *N*, as indicated by the second row of the tree. Each of these packets arrives on the preferred path to *I* (assuming that the preferred path falls along the sink tree) and is so indicated by a circle around the letter. On the second hop, eight packets are generated, two by each of the routers that received a packet on the first hop. As it turns out, all eight of these arrive at previously unvisited routers, and five of these arrive along the preferred line. Of the six packets generated on the third hop, only three arrive on the preferred path

(at C , E , and K); the others are duplicates. After five hops and 23 packets, the broadcasting terminates, compared with four hops and 14 packets had the sink tree been followed exactly.

The principal advantage of reverse path forwarding is that it is both reasonably efficient and easy to implement. It does not require routers to know about spanning trees, nor does it have the overhead of a destination list or bit map in each broadcast packet as does multidestination addressing. Nor does it require any special mechanism to stop the process, as flooding does (either a hop counter in each packet and a priori knowledge of the subnet diameter, or a list of packets already seen per source).

5.2.10. Multicast Routing

For some applications, widely-separated processes work together in groups, for example, a group of processes implementing a distributed database system. It frequently is necessary for one process to send a message to all the other members of the group. If the group is small, it can just send each other member a point-to-point message. If the group is large, this strategy is expensive. Sometimes broadcasting can be used, but using broadcasting to inform 1000 machines on a million-node network is inefficient because most receivers are not interested in the message (or worse yet, they are definitely interested but are not supposed to see it). Thus we need a way to send messages to well-defined groups that are numerically large in size but small compared to the network as a whole.

Sending a message to such a group is called **multicasting**, and its routing algorithm is called **multicast routing**. In this section we will describe one way of doing multicast routing. For additional information, see (Deering and Cheriton, 1990; Deering et al., 1994; and Rajagopalan, 1992).

To do multicasting, group management is required. Some way is needed to create and destroy groups, and for processes to join and leave groups. How these tasks are accomplished is not of concern to the routing algorithm. What is of concern is that when a process joins a group, it informs its host of this fact. It is important that routers know which of their hosts belong to which groups. Either hosts must inform their routers about changes in group membership, or routers must query their hosts periodically. Either way, routers learn about which of their hosts are in which groups. Routers tell their neighbors, so the information propagates through the subnet.

To do multicast routing, each router computes a spanning tree covering all other routers in the subnet. For example, in Fig. 5-21(a) we have a subnet with two groups, 1 and 2. Some routers are attached to hosts that belong to one or both of these groups, as indicated in the figure. A spanning tree for the leftmost router is shown in Fig. 5-21(b).

When a process sends a multicast packet to a group, the first router examines its spanning tree and prunes it, removing all lines that do not lead to hosts that are

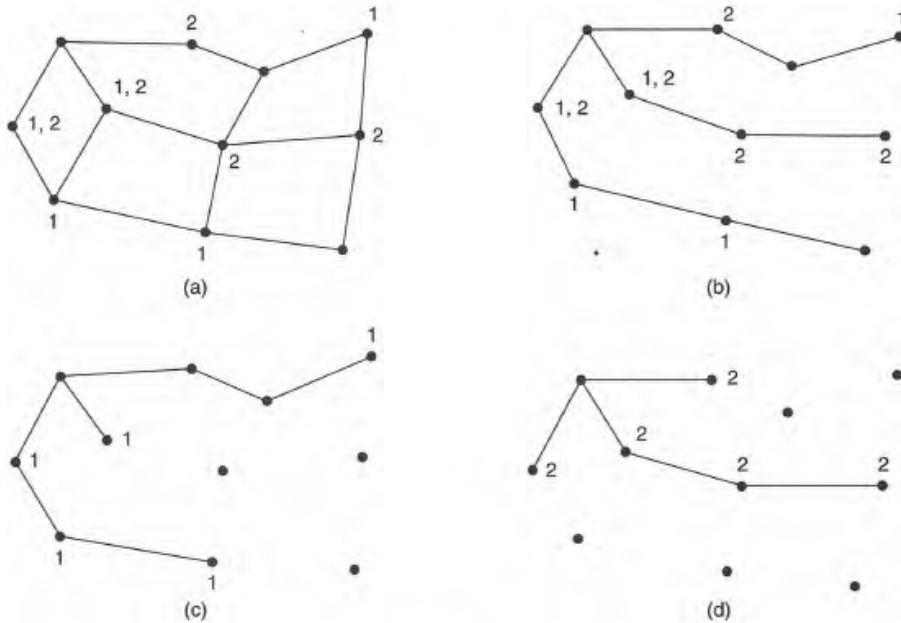


Fig. 5-21. (a) A subnet. (b) A spanning tree for the leftmost router. (c) A multicast tree for group 1. (d) A multicast tree for group 2.

members of the group. In our example, Fig. 5-21(c) shows the pruned spanning tree for group 1. Similarly, Fig. 5-21(d) shows the pruned spanning tree for group 2. Multicast packets are forwarded only along the appropriate spanning tree.

Various ways of pruning the spanning tree are possible. The simplest one can be used if link state routing is used, and each router is aware of the complete subnet topology, including which hosts belong to which groups. Then the spanning tree can be pruned by starting at the end of each path and working toward the root, removing all routers that do not belong to the group in question.

With distance vector routing, a different pruning strategy can be followed. The basic algorithm is reverse path forwarding. However, whenever a router with no hosts interested in a particular group and no connections to other routers receives a multicast message for that group, it responds with a PRUNE message, telling the sender not to send it any more multicasts for that group. When a router with no group members among its own hosts has received such messages on all its lines, it, too, can respond with a PRUNE message. In this way, the subnet is recursively pruned.

One potential disadvantage of this algorithm is that it scales poorly to large networks. Suppose that a network has n groups, each with an average of m

members. For each group, m pruned spanning trees must be stored, for a total of mn trees. When many large groups exist, considerable storage is needed to store all the trees.

An alternative design uses **core-base trees** (Ballardie et al., 1993). Here, a single spanning tree per group is computed, with the root (the core) near the middle of the group. To send a multicast message, a host sends it to the core, which then does the multicast along the spanning tree. Although this tree will not be optimal for all sources, the reduction in storage costs from m trees to one tree per group is a major saving.

5.3. CONGESTION CONTROL ALGORITHMS

When too many packets are present in (a part of) the subnet, performance degrades. This situation is called **congestion**. Figure 5-22 depicts the symptom. When the number of packets dumped into the subnet by the hosts is within its carrying capacity, they are all delivered (except for a few that are afflicted with transmission errors), and the number delivered is proportional to the number sent. However, as traffic increases too far, the routers are no longer able to cope, and they begin losing packets. This tends to make matters worse. At very high traffic, performance collapses completely, and almost no packets are delivered.

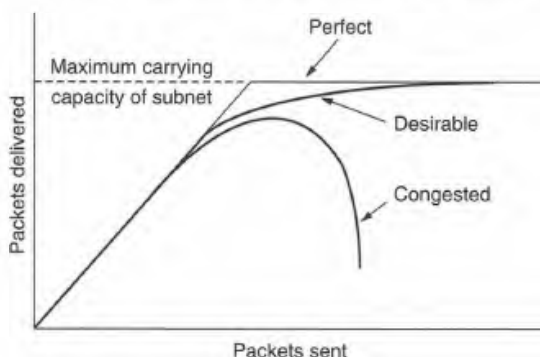


Fig. 5-22. When too much traffic is offered, congestion sets in and performance degrades sharply.

Congestion can be brought about by several factors. If all of a sudden, streams of packets begin arriving on three or four input lines and all need the same output line, a queue will build up. If there is insufficient memory to hold all of them, packets will be lost. Adding more memory may help up to a point, but Nagle (1987) discovered that if routers have an infinite amount of memory,

congestion gets worse, not better, because by the time packets get to the front of the queue, they have already timed out (repeatedly), and duplicates have been sent. All these packets will be dutifully forwarded to the next router, increasing the load all the way to the destination.

Slow processors can also cause congestion. If the routers' CPUs are slow at performing the bookkeeping tasks required of them (queueing buffers, updating tables, etc.), queues can build up, even though there is excess line capacity. Similarly, low-bandwidth lines can also cause congestion. Upgrading the lines but not changing the processors, or vice versa, often helps a little, but frequently just shifts the bottleneck. Also, upgrading part, but not all, of the system, often just moves the bottleneck somewhere else. The real problem is frequently a mismatch between parts of the system. This problem will persist until all the components are in balance.

Congestion tends to feed upon itself and become worse. If a router has no free buffers, it must ignore newly arriving packets. When a packet is discarded, the sending router (a neighbor) may time out and retransmit it, perhaps ultimately many times. Since it cannot discard the packet until it has been acknowledged, congestion at the receiver's end forces the sender to refrain from releasing a buffer it would have normally freed. In this manner, congestion backs up, like cars approaching a toll booth.

It is worth explicitly pointing out the difference between congestion control and flow control, as the relationship is subtle. Congestion control has to do with making sure the subnet is able to carry the offered traffic. It is a global issue, involving the behavior of all the hosts, all the routers, the store-and-forwarding processing within the routers, and all the other factors that tend to diminish the carrying capacity of the subnet.

Flow control, in contrast, relates to the point-to-point traffic between a given sender and a given receiver. Its job is to make sure that a fast sender cannot continually transmit data faster than the receiver can absorb it. Flow control nearly always involves some direct feedback from the receiver to the sender to tell the sender how things are doing at the other end.

To see the difference between these two concepts, consider a fiber optic network with a capacity of 1000 gigabits/sec on which a supercomputer is trying to transfer a file to a personal computer at 1 Gbps. Although there is no congestion (the network itself is not in trouble), flow control is needed to force the supercomputer to stop frequently to give the personal computer a chance to breathe.

At the other extreme, consider a store-and-forward network with 1-Mbps lines and 1000 large computers, half of which are trying to transfer files at 100 kbps to the other half. Here the problem is not that of fast senders overpowering slow receivers, but simply that the total offered traffic exceeds what the network can handle.

The reason congestion control and flow control are often confused is that some congestion control algorithms operate by sending messages back to the

various sources telling them to slow down when the network gets into trouble. Thus a host can get a "slow down" message either because the receiver cannot handle the load, or because the network cannot handle it. We will come back to this point later.

We will start our study of congestion control by looking at a general model for dealing with it. Then we will look at broad approaches to preventing it in the first place. After that, we will look at various dynamic algorithms for coping with it once it has set in.

5.3.1. General Principles of Congestion Control

Many problems in complex systems, such as computer networks, can be viewed from a control theory point of view. This approach leads to dividing all solutions into two groups: open loop and closed loop. Open loop solutions attempt to solve the problem by good design, in essence, to make sure it does not occur in the first place. Once the system is up and running, midcourse corrections are not made.

Tools for doing open-loop control include deciding when to accept new traffic, deciding when to discard packets and which ones, and making scheduling decisions at various points in the network. All of these have in common the fact that they make decisions without regard to the current state of the network.

In contrast, closed loop solutions are based on the concept of a feedback loop. This approach has three parts when applied to congestion control:

1. Monitor the system to detect when and where congestion occurs.
2. Pass this information to places where action can be taken.
3. Adjust system operation to correct the problem.

Various metrics can be used to monitor the subnet for congestion. Chief among these are the percentage of all packets discarded for lack of buffer space, the average queue lengths, the number of packets that time out and are retransmitted, the average packet delay, and the standard deviation of packet delay. In all cases, rising numbers indicate growing congestion.

The second step in the feedback loop is to transfer the information about the congestion from the point where it is detected to the point where something can be done about it. The obvious way is for the router detecting the congestion to send a packet to the traffic source or sources, announcing the problem. Of course, these extra packets increase the load at precisely the moment that more load is not needed, namely, when the subnet is congested.

However, other possibilities also exist. For example, a bit or field can be reserved in every packet for routers to fill in whenever congestion gets above some threshold level. When a router detects this congested state, it fills in the field in all outgoing packets, to warn the neighbors.

Still another approach is to have hosts or routers send probe packets out periodically to explicitly ask about congestion. This information can then be used to route traffic around problem areas. Some radio stations have helicopters flying around their cities to report on road congestion in the hope that their listeners will route their packets (cars) around hot spots.

In all feedback schemes, the hope is that knowledge of congestion will cause the hosts to take appropriate action to reduce the congestion. To work correctly, the time scale must be adjusted carefully. If every time two packets arrive in a row, a router yells STOP, and every time a router is idle for 20 μ sec it yells GO, the system will oscillate wildly and never converge. On the other hand, if it waits 30 minutes to make sure before saying anything, the congestion control mechanism will react too sluggishly to be of any real use. To work well, some kind of averaging is needed, but getting the time constant right is a nontrivial matter.

Many congestion control algorithms are known. To provide a way to organize them in a sensible way, Yang and Reddy (1995) have developed a taxonomy for congestion control algorithms. They begin by dividing all algorithms into open loop or closed loop, as described above. They further divide the open loop algorithms into ones that act at the source versus ones that act at the destination. The closed loop algorithms are also divided into two subcategories: explicit feedback versus implicit feedback. In explicit feedback algorithms, packets are sent back from the point of congestion to warn the source. In implicit algorithms, the source deduces the existence of congestion by making local observations, such as the time needed for acknowledgements to come back.

The presence of congestion means that the load is (temporarily) greater than the resources (in part of the system) can handle. Two solutions come to mind: increase the resources or decrease the load. For example, the subnet may start using dial-up telephone lines to temporarily increase the bandwidth between certain points. In systems like SMDS (see Chap. 1), it may ask the carrier for additional bandwidth for a while. On satellite systems, increasing transmission power often gives higher bandwidth. Splitting traffic over multiple routes instead of always using the best one may also effectively increase the bandwidth. Finally, spare routers that are normally used only as backups (to make the system fault tolerant) can be put on-line to give more capacity when serious congestion appears.

However, sometimes it is not possible to increase the capacity, or it has already been increased to the limit. The only way then to beat back the congestion is to decrease the load. Several ways exist to reduce the load, including denying service to some users, degrading service to some or all users, and having users schedule their demands in a more predictable way.

Some of these methods, which we will study shortly, can best be applied to virtual circuits. For subnets that use virtual circuits internally, these methods can be used at the network layer. For datagram subnets, they can nevertheless sometimes be used on transport layer connections. In this chapter, we will focus on

their use in the network layer. In the next one, we will see what can be done at the transport layer to manage congestion.

5.3.2. Congestion Prevention Policies

Let us begin our study of methods to control congestion by looking at open loop systems. These systems are designed to minimize congestion in the first place, rather than letting it happen and reacting after the fact. They try to achieve their goal by using appropriate policies at various levels. In Fig. 5-23 we see different data link, network, and transport policies that can affect congestion (Jain, 1990).

Layer	Policies
Transport	• Retransmission policy • Out-of-order caching policy • Acknowledgement policy • Flow control policy • Timeout determination
Network	• Virtual circuits versus datagram inside the subnet • Packet queueing and service policy • Packet discard policy • Routing algorithm • Packet lifetime management
Data link	• Retransmission policy • Out-of-order caching policy • Acknowledgement policy • Flow control policy

Fig. 5-23. Policies that affect congestion.

Let us start at the data link layer and work our way upward. The retransmission policy deals with how fast a sender times out and what it transmits upon timeout. A jumpy sender that times out quickly and retransmits all outstanding packets using go back n will put a heavier load on the system than a leisurely sender that uses selective repeat. Closely related to this is caching policy. If receivers routinely discard all out-of-order packets, these packets will have to be transmitted again later, creating extra load.

Acknowledgement policy also affects congestion. If each packet is acknowledged immediately, the acknowledgement packets generate extra traffic. However, if acknowledgements are saved up to piggyback onto reverse traffic, extra timeouts and retransmissions may result. A tight flow control scheme (e.g., a small window) reduces the data rate and thus helps fight congestion.

At the network layer, the choice between virtual circuits and datagrams affects congestion, since many congestion control algorithms work only with virtual circuit subnets. Packet queueing and service policy relates to whether routers have one queue per input line, one queue per output line, or both. It also relates to the order packets are processed (e.g., round robin, or priority based). Discard policy is the rule telling which packet is dropped when there is no space. A good policy can help alleviate congestion and a bad one can make it worse.

The routing algorithm can help avoid congestion by spreading the traffic over all the lines, whereas a bad one can send too much traffic over already congested lines. Finally, packet lifetime management deals with how long a packet may live before being discarded. If it is too long, lost packets may clog up the works for a long time, but if it is too short, packets may sometimes time out before reaching their destination, thus inducing retransmissions.

In the transport layer, the same issues occur as in the data link layer, but in addition, determining the timeout interval is harder because the transit time across the network is less predictable than the transit time over a wire between two routers. If it is too short, extra packets will be sent unnecessarily. If it is too long, congestion will be reduced, but the response time will suffer whenever a packet is lost.

5.3.3. Traffic Shaping

One of the main causes of congestion is that traffic is often bursty. If hosts could be made to transmit at a uniform rate, congestion would be less common. Another open loop method to help manage congestion is forcing the packets to be transmitted at a more predictable rate. This approach to congestion management is widely used in ATM networks and is called **traffic shaping**.

Traffic shaping is about regulating the average *rate* (and burstiness) of data transmission. In contrast, the sliding window protocols we studied earlier limit the amount of data in transit at once, not the rate at which it is sent. When a virtual circuit is set up, the user and the subnet (i.e., the customer and the carrier) agree on a certain traffic pattern (i.e., shape) for that circuit. As long as the customer fulfills her part of the bargain and only sends packets according to the agreed upon contract, the carrier promises to deliver them all in a timely fashion. Traffic shaping reduces congestion and thus helps the carrier live up to its promise. Such agreements are not so important for file transfers but are of great importance for real-time data, such as audio and video connections, which do not tolerate congestion well.

In effect, with traffic shaping the customer says to the carrier: "My transmission pattern will look like this. Can you handle it?" If the carrier agrees, the issue arises of how the carrier can tell if the customer is following the agreement, and what to do if the customer is not. Monitoring a traffic flow is called **traffic policing**. Agreeing to a traffic shape and policing it afterward are easier with virtual

circuit subnets than with datagram subnets. However, even with datagram subnets, the same ideas can be applied to transport layer connections.

The Leaky Bucket Algorithm

Imagine a bucket with a small hole in the bottom, as illustrated in Fig. 5-24(a). No matter at what rate water enters the bucket, the outflow is at a constant rate, ρ , when there is any water in the bucket, and zero when the bucket is empty. Also, once the bucket is full, any additional water entering it spills over the sides and is lost (i.e., does not appear in the output stream under the hole).

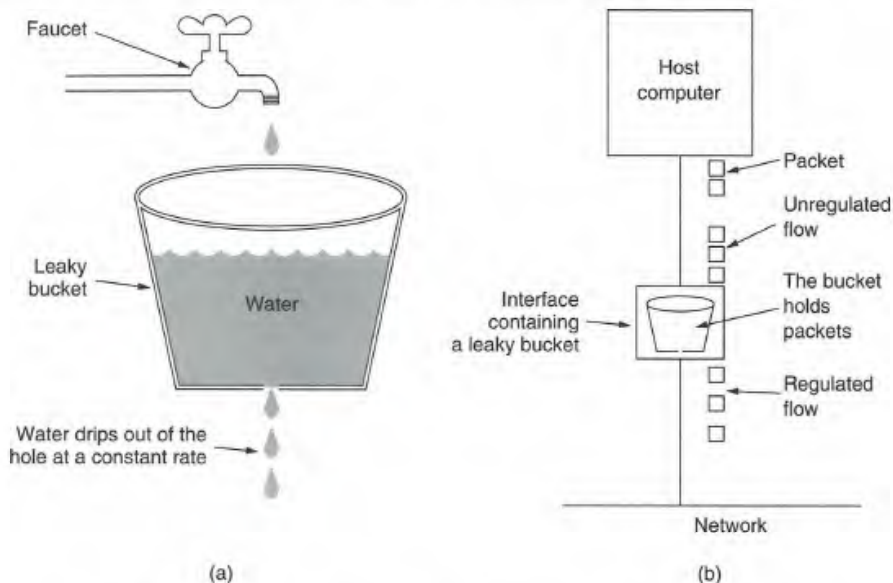


Fig. 5-24. (a) A leaky bucket with water. (b) A leaky bucket with packets.

The same idea can be applied to packets, as shown in Fig. 5-24(b). Conceptually, each host is connected to the network by an interface containing a leaky bucket, that is, a finite internal queue. If a packet arrives at the queue when it is full, the packet is discarded. In other words, if one or more processes within the host try to send a packet when the maximum number are already queued, the new packet is unceremoniously discarded. This arrangement can be built into the hardware interface or simulated by the host operating system. It was first proposed by Turner (1986) and is called the **leaky bucket algorithm**. In fact, it is nothing other than a single-server queueing system with constant service time.

The host is allowed to put one packet per clock tick onto the network. Again, this can be enforced by the interface card or by the operating system. This

mechanism turns an uneven flow of packets from the user processes inside the host into an even flow of packets onto the network, smoothing out bursts and greatly reducing the chances of congestion.

When the packets are all the same size (e.g., ATM cells), this algorithm can be used as described. However, when variable-sized packets are being used, it is often better to allow a fixed number of bytes per tick, rather than just one packet. Thus if the rule is 1024 bytes per tick, a single 1024-byte packet can be admitted on a tick, two 512-byte packets, four 256-byte packets, and so on. If the residual byte count is too low, the next packet must wait until the next tick.

Implementing the original leaky bucket algorithm is easy. The leaky bucket consists of a finite queue. When a packet arrives, if there is room on the queue it is appended to the queue; otherwise, it is discarded. At every clock tick, one packet is transmitted (unless the queue is empty).

The byte-counting leaky bucket is implemented almost the same way. At each tick, a counter is initialized to n . If the first packet on the queue has fewer bytes than the current value of the counter, it is transmitted, and the counter is decremented by that number of bytes. Additional packets may also be sent, as long as the counter is high enough. When the counter drops below the length of the next packet on the queue, transmission stops until the next tick, at which time the residual byte count is overwritten and lost.

As an example of a leaky bucket, imagine that a computer can produce data at 25 million bytes/sec (200 Mbps) and that the network also runs at this speed. However, the routers can handle this data rate only for short intervals. For long intervals, they work best at rates not exceeding 2 million bytes/sec. Now suppose data comes in 1-million-byte bursts, one 40-msec burst every second. To reduce the average rate to 2 MB/sec, we could use a leaky bucket with $\rho = 2$ MB/sec and a capacity, C , of 1 MB. This means that bursts of up to 1 MB can be handled without data loss, and that such bursts are spread out over 500 msec, no matter how fast they come in.

In Fig. 5-25(a) we see the input to the leaky bucket running at 25 MB/sec for 40 msec. In Fig. 5-25(b) we see the output draining out at a uniform rate of 2 MB/sec for 500 msec.

The Token Bucket Algorithm

The leaky bucket algorithm enforces a rigid output pattern at the average rate, no matter how bursty the traffic is. For many applications, it is better to allow the output to speed up somewhat when large bursts arrive, so a more flexible algorithm is needed, preferably one that never loses data. One such algorithm is the **token bucket algorithm**. In this algorithm, the leaky bucket holds tokens, generated by a clock at the rate of one token every ΔT sec. In Fig. 5-26(a) we see a bucket holding three tokens, with five packets waiting to be transmitted. For a packet to be transmitted, it must capture and destroy one token. In Fig. 5-26(b)

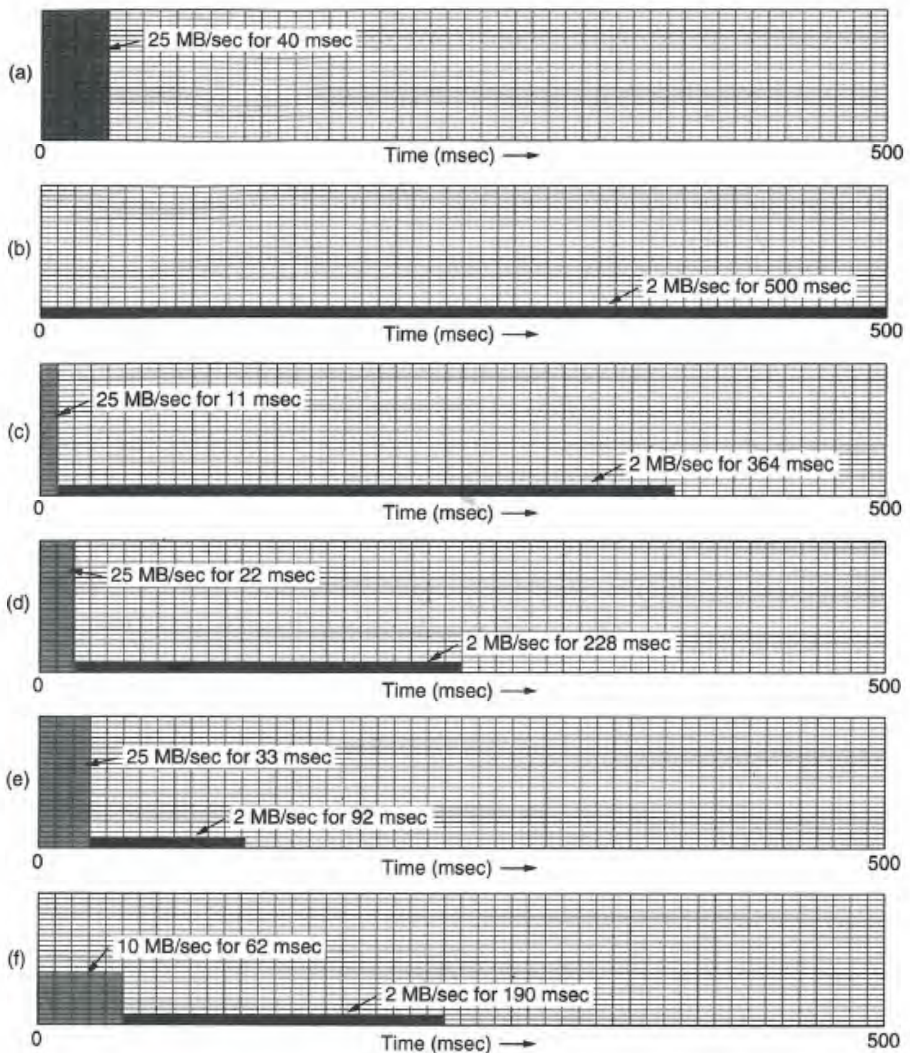


Fig. 5-25. (a) Input to a leaky bucket. (b) Output from a leaky bucket. (c) - (e) Output from a token bucket with capacities of 250KB, 500KB, and 750KB. (f) Output from a 500KB token bucket feeding a 10 MB/sec leaky bucket.

we see that three of the five packets have gotten through, but the other two are stuck waiting for two more tokens to be generated.

The token bucket algorithm provides a different kind of traffic shaping than the leaky bucket algorithm. The leaky bucket algorithm does not allow idle hosts

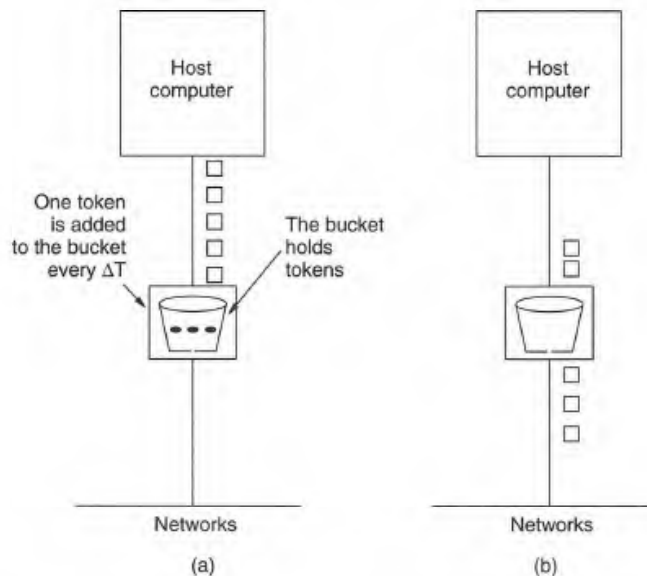


Fig. 5-26. The token bucket algorithm. (a) Before. (b) After.

to save up permission to send large bursts later. The token bucket algorithm does allow saving, up to the maximum size of the bucket, n . This property means that bursts of up to n packets can be sent at once, allowing some burstiness in the output stream and giving faster response to sudden bursts of input.

Another difference between the two algorithms is that the token bucket algorithm throws away tokens when the bucket fills up but never discards packets. In contrast, the leaky bucket algorithm discards packets when the bucket fills up.

Here too, a minor variant is possible, in which each token represents the right to send not one packet, but k bytes. A packet can only be transmitted if enough tokens are available to cover its length in bytes. Fractional tokens are kept for future use.

The leaky bucket and token bucket algorithms can also be used to smooth traffic between routers, as well as being used to regulate host output as in our examples. However, one clear difference is that a token bucket regulating a host can make the host stop sending when the rules say it must. Telling a router to stop sending while its input keeps pouring in may result in lost data.

The implementation of the basic token bucket algorithm is just a variable that counts tokens. The counter is incremented by one every ΔT and decremented by one whenever a packet is sent. When the counter hits zero, no packets may be sent. In the byte-count variant, the counter is incremented by k bytes every ΔT and decremented by the length of each packet sent.

Essentially what the token bucket does is allow bursts, but up to a regulated maximum length. Look at Fig. 5-25(c) for example. Here we have a token bucket with a capacity of 250 KB. Tokens arrive at a rate allowing output at 2 MB/sec. Assuming the token bucket is full when the 1-MB burst arrives, the bucket can drain at the full 25 MB/sec for about 11 msec. Then it has to cut back to 2 MB/sec until the entire input burst has been sent.

Calculating the length of the maximum rate burst is slightly tricky. It is not just 1 MB divided by 25 MB/sec because while the burst is being output, more tokens arrive. If we call the burst length S sec, the token bucket capacity C bytes, the token arrival rate ρ bytes/sec, and the maximum output rate M bytes/sec, we see that an output burst contains a maximum of $C + \rho S$ bytes. We also know that the number of bytes in a maximum-speed burst of length S seconds is MS . Hence we have

$$C + \rho S = MS$$

We can solve this equation to get $S = C/(M - \rho)$. For our parameters of $C = 250$ KB, $M = 25$ MB/sec, and $\rho = 2$ MB/sec, we get a burst time of about 11 msec. Figure 5-25(d) and Fig. 5-25(e) show the token bucket for capacities of 500-KB and 750 KB, respectively.

A potential problem with the token bucket algorithm is that it allows large bursts again, even though the maximum burst interval can be regulated by careful selection of ρ and M . Frequently it is desirable to reduce the peak rate, but without going back to the low value of the original leaky bucket.

One way to get smoother traffic is to put a leaky bucket after the token bucket. The rate of the leaky bucket should be higher than the token bucket's ρ but lower than the maximum rate of the network. Figure 5-25(f) shows the output for a 500 KB token bucket followed by a 10-MB/sec leaky bucket.

Policing all these schemes can be a bit tricky. Essentially, the network has to simulate the algorithm and make sure that no more packets or bytes are being sent than are permitted. Excess packets are then discarded or downgraded, as discussed later.

5.3.4. Flow Specifications

Traffic shaping is most effective when the sender, receiver, and subnet all agree to it. To get agreement, it is necessary to specify the traffic pattern in a precise way. Such an agreement is called a **flow specification**. It consists of a data structure that describes both the pattern of the injected traffic and the quality of service desired by the applications. A flow specification can apply either to the packets sent on a virtual circuit, or to a sequence of datagrams sent between a source and a destination (or even to multiple destinations).

In this section we will describe an example flow specification designed by Partridge (1992). It is shown in Fig. 5-27. The idea is that before a connection is

established or before a sequence of datagrams are sent, the source gives the flow specification to the subnet for approval. The subnet can either accept it, reject it, or come back with a counterproposal ("I cannot give you 100 msec average delay; can you live with 150 msec?"). Once the sender and subnet have struck a deal, the sender can ask the receiver if it, too, agrees.

Characteristics of the Input	Service Desired
Maximum packet size (bytes)	Loss sensitivity (bytes)
Token bucket rate (bytes/sec)	Loss interval (μ sec)
Token bucket size (bytes)	Burst loss sensitivity (packets)
Maximum transmission rate (bytes/sec)	Minimum delay noticed (μ sec)
	Maximum delay variation (μ sec)
	Quality of guarantee

Fig. 5-27. An example flow specification.

Let us now examine the parameters of our example flow specification starting with the traffic specification. The *Maximum packet size* tells how big packets may be. The next two parameters implicitly assume that traffic will be shaped by the token bucket algorithm working in bytes. They tell how many bytes are put into the token bucket per second, and how big the bucket is. If the rate is r bytes/sec and the bucket size is b bytes, then in any arbitrary time interval Δt , the maximum number of bytes that may be sent is $b + r\Delta t$. Here the first term represents the maximum possible contents of the bucket at the start of the interval and the second one represents the new credits that come in during the interval. The *Maximum transmission rate* is the top rate the host is capable of producing under any conditions and implicitly specifies the shortest time interval in which the token bucket could be emptied.

The second column specifies what the application wants from the subnet. The first and second parameters represent the numerator and denominator of a fraction giving the maximum acceptable loss rate (e.g., 1 byte per hour). Alternatively, they can indicate that the flow is insensitive to packet loss. The *Burst loss sensitivity* tells how many consecutive lost packets can be tolerated.

The next two service parameters deal with delay. The *Minimum delay noticed* says how long a packet can be delayed without the application noticing. For a file transfer, it might be a second, but for an audio stream 3 msec might be the limit. The *Maximum delay variation* tries to quantify the fact that some applications are not sensitive to the actual delay but are highly sensitive to the **jitter**, that is, the amount of variation in the end-to-end packet transit time. It is two times the number of microseconds a packet's delay may vary from the average. Thus a value of 2000 means that a packet may be up to 1 msec early or late, but no more.

Finally, the *Quality of guarantee* indicates whether or not the application really means it. On the one hand, the loss and delay characteristics might be ideal goals, but no harm is done if they are not met. On the other hand, they might be so important that if they cannot be met, the application simply terminates. Intermediate positions are also possible.

Although we have looked at the flow specification as a request from the application to the subnet, it can also be a return value telling what the subnet can do. Thus it can potentially be used for an extended negotiation about the service level.

A problem inherent with any flow specification is that the application may not know what it really wants. For example, an application program running in New York might be quite happy with a delay of 200 msec to Sydney, but most unhappy with the same 200-msec delay to Boston. Here the “minimum service” is clearly a function of what is thought to be possible.

5.3.5. Congestion Control in Virtual Circuit Subnets

The congestion control methods described above are basically open loop: they try to prevent congestion from occurring in the first place, rather than dealing with it after the fact. In this section we will describe some approaches to dynamically controlling congestion in virtual circuit subnets. In the next two, we will look at techniques that can be used in any subnet.

One technique that is widely used to keep congestion that has already started from getting worse is **admission control**. The idea is simple: once congestion has been signaled, no more virtual circuits are set up until the problem has gone away. Thus, attempts to set up new transport layer connections fail. Letting more people in just makes matters worse. While this approach is crude, it is simple and easy to carry out. In the telephone system, when a switch gets overloaded, it also practices admission control, by not giving dial tones.

An alternative approach is to allow new virtual circuits but carefully route all new virtual circuits around problem areas. For example, consider the subnet of Fig. 5-28(a), in which two routers are congested, as indicated.

Suppose that a host attached to router *A* wants to set up a connection to a host attached to router *B*. Normally, this connection would pass through one of the congested routers. To avoid this situation, we can redraw the subnet as shown in Fig. 5-28(b), omitting the congested routers and all of their lines. The dashed line shows a possible route for the virtual circuit that avoids the congested routers.

Another strategy relating to virtual circuits is to negotiate an agreement between the host and subnet when a virtual circuit is set up. This agreement normally specifies the volume and shape of the traffic, quality of service required, and other parameters. To keep its part of the agreement, the subnet will typically reserve resources along the path when the circuit is set up. These resources can include table and buffer space in the routers and bandwidth on the lines. In this

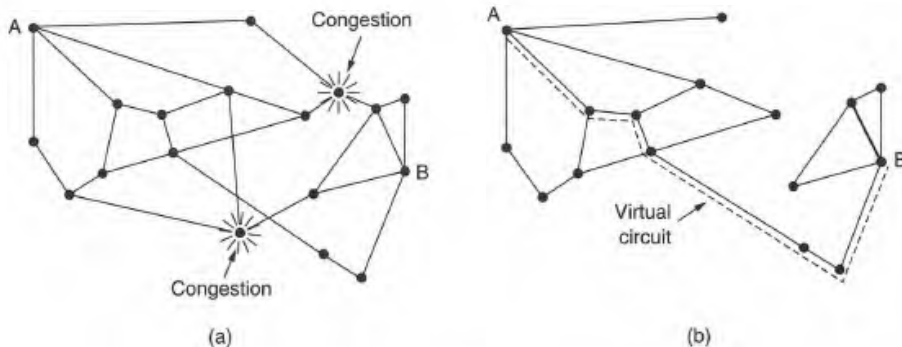


Fig. 5-28. (a) A congested subnet. (b) A redrawn subnet that eliminates the congestion and a virtual circuit from A to B.

way, congestion is unlikely to occur on the new virtual circuits because all the necessary resources are guaranteed to be available.

This kind of reservation can be done all the time as standard operating procedure, or only when the subnet is congested. A disadvantage of doing it all the time is that it tends to waste resources. If six virtual circuits that might use 1 Mbps all pass through the same physical 6-Mbps line, the line has to be marked as full, even though it may rarely happen that all six virtual circuits are transmitting at the same time. Consequently, the price of the congestion control is unused bandwidth.

5.3.6. Choke Packets

Let us now turn to an approach that can be used in both virtual circuit and datagram subnets. Each router can easily monitor the utilization of its output lines and other resources. For example, it can associate with each line a real variable, u , whose value, between 0.0 and 1.0, reflects the recent utilization of that line. To maintain a good estimate of u , a sample of the instantaneous line utilization, f (either 0 or 1), can be made periodically and u updated according to

$$u_{\text{new}} = au_{\text{old}} + (1 - a)f$$

where the constant a determines how fast the router forgets recent history.

Whenever u moves above the threshold, the output line enters a "warning" state. Each newly arriving packet is checked to see if its output line is in warning state. If so, the router sends a **choke packet** back to the source host, giving it the destination found in the packet. The original packet is tagged (a header bit is turned on) so that it will not generate any more choke packets further along the path and is then forwarded in the usual way.

When the source host gets the choke packet, it is required to reduce the traffic sent to the specified destination by X percent. Since other packets aimed at the same destination are probably already under way and will generate yet more choke packets, the host should ignore choke packets referring to that destination for a fixed time interval. After that period has expired, the host listens for more choke packets for another interval. If one arrives, the line is still congested, so the host reduces the flow still more and begins ignoring choke packets again. If no choke packets arrive during the listening period, the host may increase the flow again. The feedback implicit in this protocol can help prevent congestion yet not throttle any flow unless trouble occurs.

Hosts can reduce traffic by adjusting their policy parameters, for example, window size or leaky bucket output rate. Typically, the first choke packet causes the data rate to be reduced to 0.50 of its previous rate, the next one causes a reduction to 0.25, and so on. Increases are done in smaller increments to prevent congestion from reoccurring quickly.

Several variations on this congestion control algorithm have been proposed. For one, the routers can maintain several thresholds. Depending on which threshold has been crossed, the choke packet can contain a mild warning, a stern warning, or an ultimatum.

Another variation is to use queue lengths or buffer utilization instead of line utilization as the trigger signal. The same exponential weighting can be used with this metric as with u , of course.

Weighted Fair Queueing

A problem with using choke packets is that the action to be taken by the source hosts is voluntary. Suppose that a router is being swamped by packets from four sources, and it sends choke packets to all of them. One of them cuts back, as it is supposed to, but the other three just keep blasting away. The result is that the honest host gets an even smaller share of the bandwidth than it had before.

To get around this problem, and thus make compliance more attractive, Nagle (1987) proposed a **fair queueing** algorithm. The essence of the algorithm is that routers have multiple queues for each output line, one for each source. When a line becomes idle, the router scans the queues round robin, taking the first packet on the next queue. In this way, with n hosts competing for a given output line, each host gets to send one out of every n packets. Sending more packets will not improve this fraction. Some ATM switches use this algorithm.

Although a start, the algorithm has a problem: it gives more bandwidth to hosts that use large packets than to hosts that use small packets. Demers et al. (1990) suggested an improvement in which the round robin is done in such a way as to simulate a byte-by-byte round robin, instead of a packet-by-packet round

robin. In effect, it scans the queues repeatedly, byte-for-byte, until it finds the tick on which each packet will be finished. The packets are then sorted in order of their finishing and sent in that order. The algorithm is illustrated in Fig. 5-29.

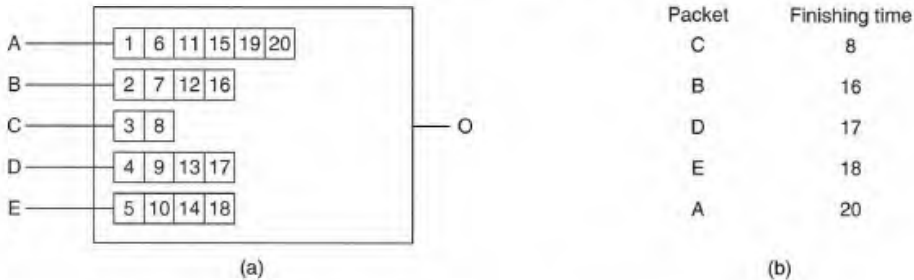


Fig. 5-29. (a) A router with five packets queued for line O. (b) Finishing times for the five packets.

In Fig. 5-29(a) we see packets of length 2 to 6 bytes. At (virtual) clock tick 1, the first byte of the packet on line A is sent. Then goes the first byte of the packet on line B, and so on. The first packet to finish is C, after eight ticks. The sorted order is given in Fig. 5-29(b). In the absence of new arrivals, the packets will be sent in the order listed, from C to A.

One problem with this algorithm is that it gives all hosts the same priority. In many situations, it is desirable to give the file and other servers more bandwidth than clients, so they can be given two or more bytes per tick. This modified algorithm is called **weighted fair queueing** and is widely used. Sometimes the weight is equal to the number of virtual circuits or flows coming out of a machine, so each process gets equal bandwidth. An efficient implementation of the algorithm is discussed in (Shreedhar and Varghese, 1995).

Hop-by-Hop Choke Packets

At high speeds and over long distances, sending a choke packet to the source hosts does not work well because the reaction is so slow. Consider, for example, a host in San Francisco (router A in Fig. 5-30) that is sending traffic to a host in New York (router D in Fig. 5-30) at 155 Mbps. If the New York host begins to run out of buffers, it will take about 30 msec for a choke packet to get back to San Francisco to tell it to slow down. The choke packet propagation is shown as the second, third, and fourth steps in Fig. 5-30(a). In those 30 msec, another 4.6 megabits (e.g., over 10,000 ATM cells) will have been sent. Even if the host in San Francisco completely shuts down immediately, the 4.6 megabits in the pipe will continue to pour in and have to be dealt with. Only in the seventh diagram in Fig. 5-30(a) will the New York router notice a slower flow.

An alternative approach is to have the choke packet take effect at every hop it passes through, as shown in the sequence of Fig. 5-30(b). Here, as soon as the choke packet reaches *F*, *F* is required to reduce the flow to *D*. Doing so will require *F* to devote more buffers to the flow, since the source is still sending away at full blast, but it gives *D* immediate relief, like a headache remedy in a television commercial. In the next step, the choke packet reaches *E*, which tells *E* to reduce the flow to *F*. This action puts a greater demand on *E*'s buffers but gives *F* immediate relief. Finally, the choke packet reaches *A* and the flow genuinely slows down.

The net effect of this hop-by-hop scheme is to provide quick relief at the point of congestion at the price of using up more buffers upstream. In this way congestion can be nipped in the bud without losing any packets. The idea is discussed in more detail and simulation results are given in (Mishra and Kanakia, 1992).

5.3.7. Load Shedding

When none of the above methods make the congestion disappear, routers can bring out the heavy artillery: load shedding. **Load shedding** is a fancy way of saying that when routers are being inundated by packets that they cannot handle, they just throw them away. The term comes from the world of electrical power generation where it refers to the practice of utilities intentionally blacking out certain areas to save the entire grid from collapsing on hot summer days when the demand for electricity greatly exceeds the supply.

A router drowning in packets can just pick packets at random to drop, but usually it can do better than that. Which packet to discard may depend on the applications running. For file transfer, an old packet is worth more than a new one because dropping packet 6 and keeping packets 7 through 10 will cause a gap at the receiver that may force packets 6 through 10 to be retransmitted (if the receiver routinely discards out-of-order packets). In a 12-packet file, dropping 6 may require 7 through 12 to be retransmitted, whereas dropping 10 may require only 10 through 12 to be retransmitted. In contrast, for multimedia, a new packet is more important than an old one. The former policy (old is better than new) is often called **wine** and the latter (new is better than old) is often called **milk**.

A step above this in intelligence requires cooperation from the senders. For many applications, some packets are more important than others. For example, certain algorithms for compressing video periodically transmit an entire frame and then send subsequent frames as differences from the last full frame. In this case, dropping a packet that is part of a difference is preferable to dropping one that is part of a full frame. As another example, consider transmitting a document containing ASCII text and pictures. Losing a line of pixels in some image is far less damaging than losing a line of readable text.

To implement an intelligent discard policy, applications must mark their packets in priority classes to indicate how important they are. If they do this, when

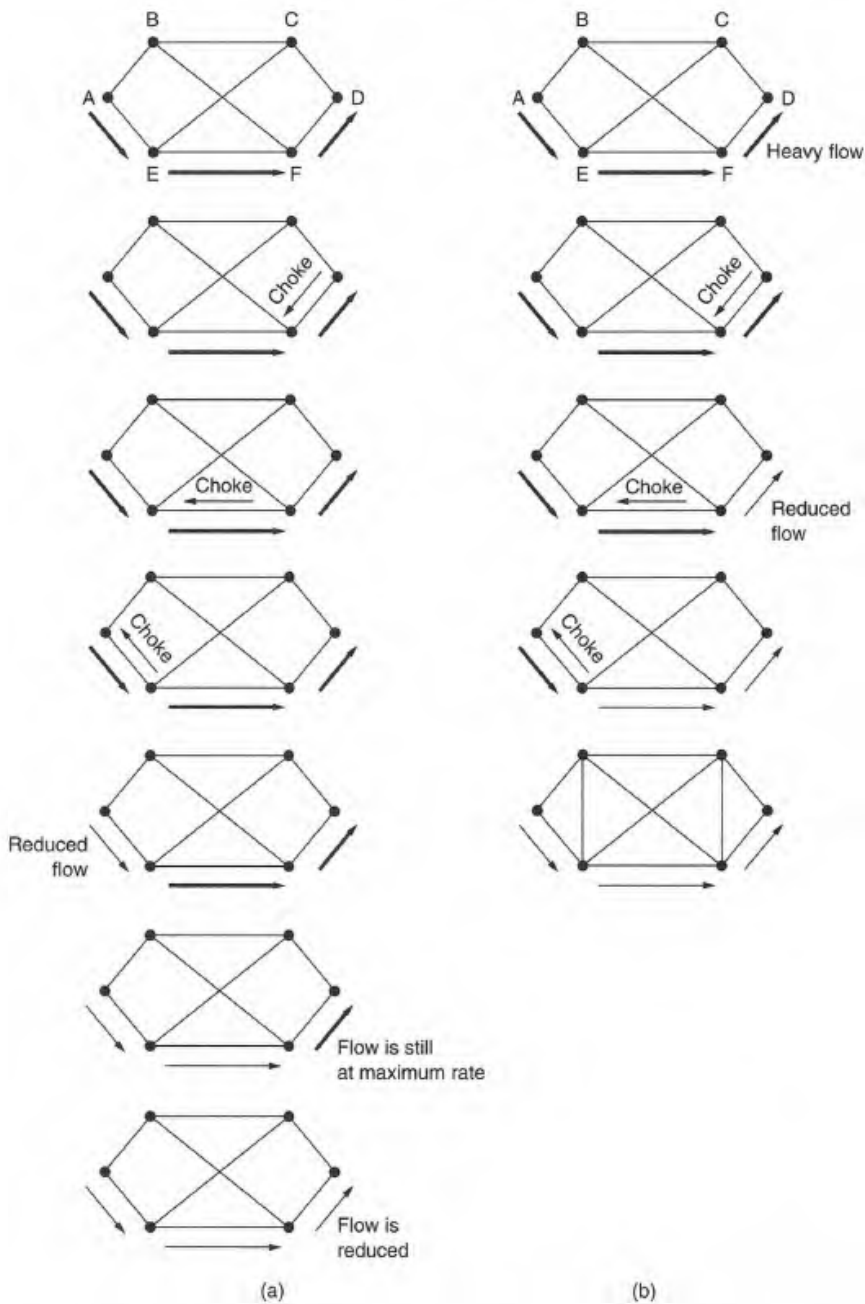


Fig. 5-30. (a) A choke packet that affects only the source. (b) A choke packet that affects each hop it passes through.

packets have to be discarded, routers can first drop packets from the lowest class, then the next lowest class, and so on. Of course, unless there is some significant incentive to mark packets as anything other than VERY IMPORTANT—NEVER, EVER DISCARD, nobody will do it.

The incentive might be in the form of money, with the low-priority packets being cheaper to send than the high-priority ones. Alternatively, priority classes could be coupled with traffic shaping. For example, there might be a rule saying that when the token bucket algorithm is being used and a packet arrives at a moment when no token is available, it may still be sent, provided that it is marked as the lowest possible priority, and thus subject to discard the instant trouble appears. Under conditions of light load, users might be happy to operate in this way, but as the load increases and packets actually begin to be discarded, they might cut back and only send packets when tokens are available.

Another option is to allow hosts to exceed the limits specified in the agreement negotiated when the virtual circuit was set up (e.g., use a higher bandwidth than allowed), but subject to the condition that all excess traffic be marked as low priority. Such a strategy is actually not a bad idea, because it makes more efficient use of idle resources, allowing hosts to use them as long as nobody else is interested, but without establishing a right to them when times get tough.

Marking packets by class requires one or more header bits in which to put the priority. ATM cells have 1 bit reserved in the header for this purpose, so every ATM cell is labeled either as low priority or high priority. ATM switches indeed use this bit when making discard decisions.

In some networks, packets are grouped together into larger units that are used for retransmission purposes. For example, in ATM networks, what we have been calling “packets” are fixed-length cells. These cells are just fragments of “messages.” When a cell is dropped, ultimately the entire “message” will be retransmitted, not just the missing cell. Under these conditions, a router that drops a cell might as well drop all the rest of the cells in that message, since transmitting them costs bandwidth and wins nothing—even if they get through they will still be retransmitted later.

Simulation results show that when a router senses trouble on the horizon, it is better off starting to discard packets early, rather than wait until it becomes completely clogged up (Floyd and Jacobson, 1993; Romanow and Floyd, 1994). Doing so may prevent the congestion from getting a foothold.

5.3.8. Jitter Control

For applications such as audio and video transmission, it does not matter much if the packets take 20 msec or 30 msec to be delivered, as long as the transit time is constant. Having some packets taking 20 msec and others taking 30 msec will give an uneven quality to the sound or image. Thus the agreement might be that 99 percent of the packets be delivered with a delay in the range of 24.5 msec

to 25.5 msec. The mean value chosen must be feasible, of course. In other words, an average amount of congestion must be calculated in.

The jitter can be bounded by computing the expected transit time for each hop along the path. When a packet arrives at a router, the router checks to see how much the packet is behind or ahead of its schedule. This information is stored in the packet and updated at each hop. If the packet is ahead of schedule, it is held just long enough to get it back on schedule. If it is behind schedule, the router tries to get it out the door quickly. In fact, the algorithm for determining which of several packets competing for an output line should go next can always choose the packet furthest behind in its schedule. In this way, packets that are ahead of schedule get slowed down and packets that are behind schedule get speeded up, in both cases reducing the amount of jitter.

5.3.9. Congestion Control for Multicasting

All of the congestion control algorithms discussed so far deal with messages from a single source to a single destination. In this section we will describe a way of managing multicast flows from multiple sources to multiple destinations. For example, imagine several closed-circuit television stations transmitting audio and video streams to a group of receivers, each of whom can view one or more stations at once and are free to switch from station to station at will. An application of this technology might be a video conference, in which each participant could focus on the current speaker or on the boss' expression, as desired.

In many multicast applications, groups can change membership dynamically, for example, as people enter a video conference or get bored and switch to a soap opera. Under these conditions, the approach of having the senders reserve bandwidth in advance does not work well, as it would require each sender to track all entries and exits of its audience and regenerate the spanning tree at each change. For a system designed to transmit cable television, with millions of subscribers, it would not work at all.

RSVP—Resource reSerVation Protocol

One interesting solution that can handle this environment is the **RSVP** protocol (Zhang et al., 1993). It allows multiple senders to transmit to multiple groups of receivers, permits individual receivers to switch channels freely, and optimizes bandwidth use while at the same time eliminating congestion.

In its simplest form, the protocol uses multicast routing using spanning trees, as discussed earlier. Each group is assigned a group address. To send to a group, a sender puts the group's address in its packets. The standard multicast routing algorithm then builds a spanning tree covering all group members. The routing

algorithm is not part of RSVP. The only difference with normal multicasting is a little extra information that is multicast to the group periodically to tell the routers along the tree to maintain certain data structures in their memories.

As an example, consider the network of Fig. 5-31(a). Hosts 1 and 2 are multicast senders, and hosts 3, 4, and 5 are multicast receivers. In this example, the senders and receivers are disjoint, but in general, the two sets may overlap. The multicast trees for hosts 1 and 2 are shown in Fig. 5-31(b) and Fig. 5-31(c), respectively.

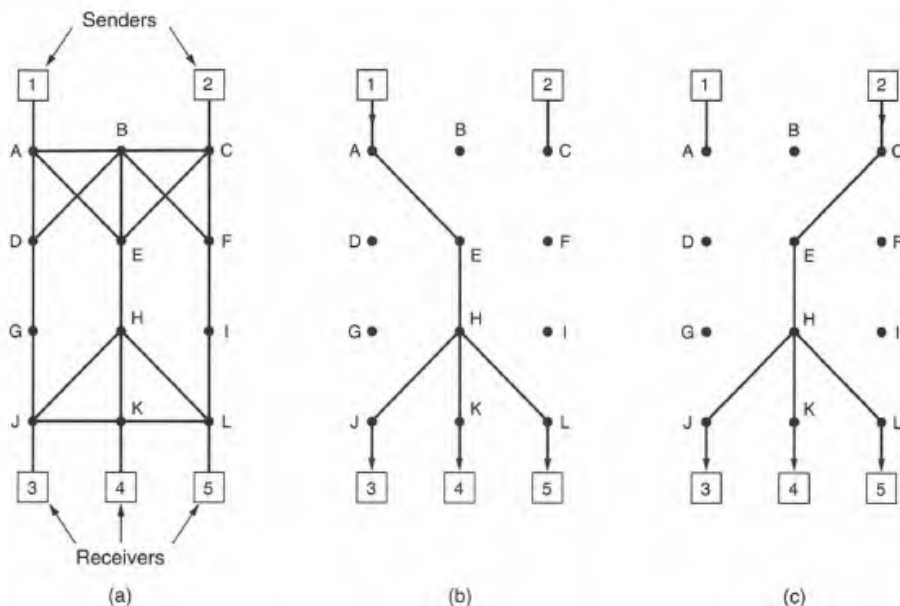


Fig. 5-31. (a) A network. (b) The multicast spanning tree for host 1. (c) The multicast spanning tree for host 2.

To get better reception and eliminate congestion, any of the receivers in a group can send a reservation message up the tree to the sender. The message is propagated using the reverse path forwarding algorithm discussed earlier. At each hop, the router notes the reservation and reserves the necessary bandwidth. If insufficient bandwidth is available, it reports back failure. By the time the message gets back to the source, bandwidth has been reserved all the way from the sender to the receiver making the reservation request along the spanning tree.

An example of such a reservation is shown in Fig. 5-32(a). Here host 3 has requested a channel to host 1. Once it has been established, packets can flow from 1 to 3 without congestion. Now consider what happens if host 3 next reserves a channel to the other sender, host 2, so the user can watch two television

programs at once. A second path is reserved, as illustrated in Fig. 5-32(b). Note that two separate channels are needed from host 3 to router *E* because two independent streams are being transmitted.

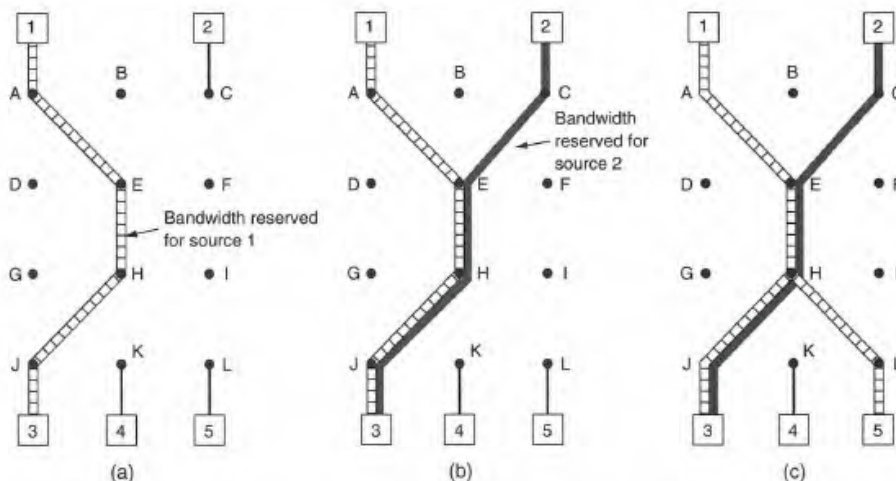


Fig. 5-32. (a) Host 3 requests a channel to host 1. (b) Host 3 then requests a second channel, to host 2. (c) Host 5 requests a channel to host 1.

Finally, in Fig. 5-32(c), host 5 decides to watch the program being transmitted by host 1 and also makes a reservation. First, dedicated bandwidth is reserved as far as router *H*. However, this router sees that it already has a feed from host 1, so if the necessary bandwidth has already been reserved, it does not have to reserve any more. Note that hosts 3 and 5 might have asked for different amounts of bandwidth (e.g., 3 has a black-and-white television set, so it does not want the color information), so the capacity reserved must be large enough to satisfy the greediest receiver.

When making a reservation, a receiver can (optionally) specify one or more sources that it wants to receive from. It can also specify whether these choices are fixed for the duration of the reservation, or whether the receiver wants to keep open the option of changing sources later. The routers use this information to optimize bandwidth planning. In particular, two receivers are only set up to share a path if they both agree not to change sources later on.

The reason for this strategy in the fully dynamic case is that reserved bandwidth is decoupled from the choice of source. Once a receiver has reserved bandwidth, it can switch to another source and keep that portion of the existing path that is valid for the new source. If host 2 is transmitting several video streams, for example, host 3 may switch between them at will without changing its reservation: the routers do not care what program the receiver is watching.

5.4.7. Firewalls

The ability to connect any computer, anywhere, to any other computer, anywhere, is a mixed blessing. For individuals at home, wandering around the Internet is lots of fun. For corporate security managers, it is a nightmare. Most companies have large amounts of confidential information on-line—trade secrets, product development plans, marketing strategies, financial analyses, etc. Disclosure of this information to a competitor could have dire consequences.

In addition to the danger of information leaking out, there is also a danger of information leaking in. In particular, viruses, worms, and other digital pests (Kaufman et al., 1995) can breach security, destroy valuable data, and waste large amounts of administrators' time trying to clean up the mess they leave. Often they are imported by careless employees who want to play some nifty new game.

Consequently, mechanisms are needed to keep “good” bits in and “bad” bits out. One method is to use encryption. This approach protects data in transit between secure sites. We will study it in Chap. 7. However, encryption does nothing to keep digital pests and hackers out. To accomplish this goal, we need to look at firewalls (Chapman and Zwicky, 1995; and Cheswick and Bellovin, 1994).

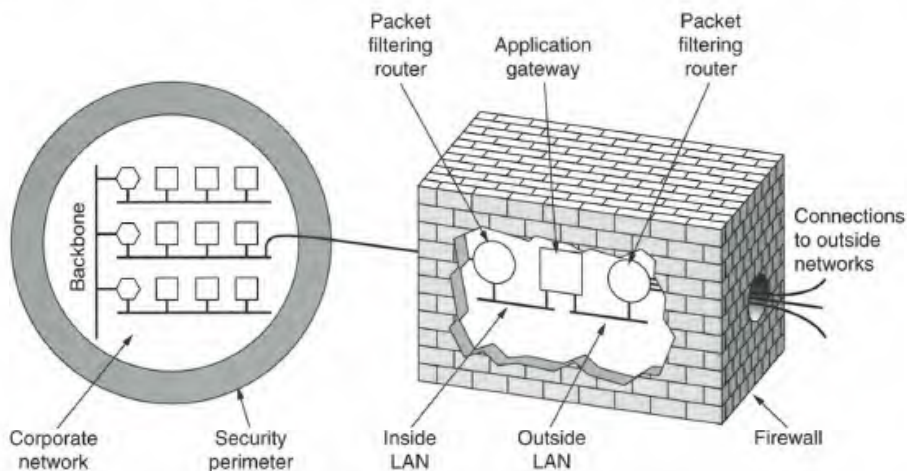


Fig. 5-43. A firewall consisting of two packet filters and an application gateway.

Firewalls are just a modern adaptation of that old medieval security standby: digging a deep moat around your castle. This design forced everyone entering or leaving the castle to pass over a single drawbridge, where they could be inspected by the I/O police. With networks, the same trick is possible: a company can have many LANs connected in arbitrary ways, but all traffic to or from the company is forced through an electronic drawbridge (firewall), as shown in Fig. 5-43.

The firewall in this configuration has two components: two routers that do packet filtering and an application gateway. Simpler configurations also exist, but the advantage of this design is that every packet must transit two filters and an application gateway to go in or out. No other route exists. Readers who think that one security checkpoint is enough clearly have not made an international flight on a scheduled airline recently.

Each **packet filter** is a standard router equipped with some extra functionality. The extra functionality allows every incoming or outgoing packet to be inspected. Packets meeting some criterion are forwarded normally. Those that fail the test are dropped.

In Fig. 5-43, most likely the packet filter on the inside LAN checks outgoing packets and the one on the outside LAN checks incoming packets. Packets crossing the first hurdle go to the application gateway for further examination. The point of putting the two packet filters on different LANs is to ensure that no packet gets in or out without having to pass through the application gateway: there is no path around it.

Packet filters are typically driven by tables configured by the system administrator. These tables list sources and destinations that are acceptable, sources and destinations that are blocked, and default rules about what to do with packets coming from or going to other machines.

In the common case of a UNIX setting, a source or destination consists of an IP address and a port. Ports indicate which service is desired. For example, port 23 is for Telnet, port 79 is for Finger, and port 119 is for USENET news. A company could block incoming packets for all IP addresses combined with one of these ports. In this way, no one outside the company could log in via Telnet, or look up people using the Finger daemon. Furthermore, the company would be spared from having employees spend all day reading USENET news.

Blocking outgoing packets is trickier because although most sites stick to the standard port naming conventions, they are not forced to do so. Furthermore, for some important services, such as FTP (File Transfer Protocol), port numbers are assigned dynamically. In addition, although blocking TCP connections is difficult, blocking UDP packets is even harder because so little is known a priori about what they will do. Many packet filters simply ban UDP traffic altogether.

The second half of the firewall mechanism is the **application gateway**. Rather than just looking at raw packets, the gateway operates at the application level. A mail gateway, for example, can be set up to examine each message going in or coming out. For each one it makes a decision to transmit or discard it based on header fields, message size, or even the content (e.g., at a military installation, the presence of words like "nuclear" or "bomb" might cause some special action to be taken).

Installations are free to set up one or more application gateways for specific applications, but it is not uncommon for suspicious organizations to permit email in and out, and perhaps use of the World Wide Web, but ban everything else as

too dicey. Combined with encryption and packet filtering, this arrangement offers a limited amount of security at the cost of some inconvenience.

One final note concerns wireless communication and firewalls. It is easy to design a system that is logically completely secure, but which, in practice, leaks like a sieve. This situation can occur if some of the machines are wireless and use radio communication, which passes right over the firewall in both directions.

5.5. THE NETWORK LAYER IN THE INTERNET

At the network layer, the Internet can be viewed as a collection of subnetworks or **Autonomous Systems (ASes)** that are connected together. There is no real structure, but several major backbones exist. These are constructed from high-bandwidth lines and fast routers. Attached to the backbones are regional (midlevel) networks, and attached to these regional networks are the LANs at many universities, companies, and Internet service providers. A sketch of this quasihierarchical organization is given in Fig. 5-44.

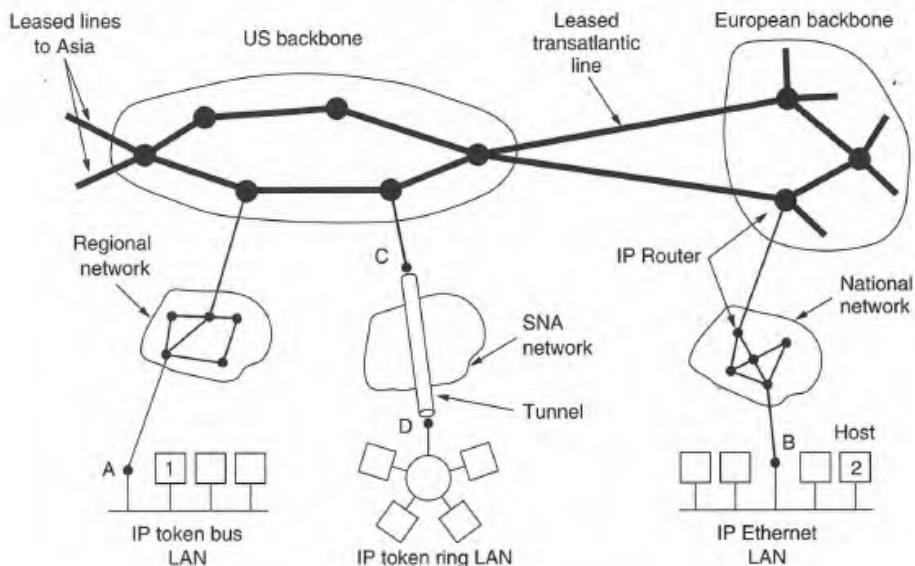


Fig. 5-44. The Internet is an interconnected collection of many networks.

The glue that holds the Internet together is the network layer protocol, **IP (Internet Protocol)**. Unlike most older network layer protocols, it was designed from the beginning with internetworking in mind. A good way to think of the network layer is this. Its job is to provide a best-efforts way to transport datagrams

from source to destination, without regard to whether or not these machines are on the same network, or whether or not there are other networks in between them.

Communication in the Internet works as follows. The transport layer takes data streams and breaks them up into datagrams. In theory, datagrams can be up to 64 Kbytes each, but in practice they are usually around 1500 bytes. Each datagram is transmitted through the Internet, possibly being fragmented into smaller units as it goes. When all the pieces finally get to the destination machine, they are reassembled by the network layer into the original datagram. This datagram is then handed to the transport layer, which inserts it into the receiving process' input stream.

5.5.1. The IP Protocol

An appropriate place to start our study of the network layer in the Internet is the format of the IP datagrams themselves. An IP datagram consists of a header part and a text part. The header has a 20-byte fixed part and a variable length optional part. The header format is shown in Fig. 5-45. It is transmitted in big endian order: from left to right, with the high-order bit of the *Version* field going first. (The SPARC is big endian; the Pentium is little endian.) On little endian machines, software conversion is required on both transmission and reception.

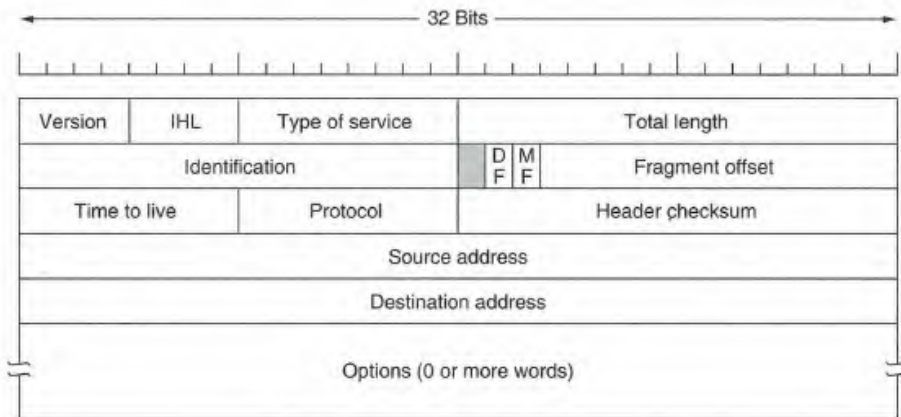


Fig. 5-45. The IP (Internet Protocol) header.

The *Version* field keeps track of which version of the protocol the datagram belongs to. By including the version in each datagram, it becomes possible to have the transition between versions take months, or even years, with some machines running the old version and others running the new one.

Since the header length is not constant, a field in the header, *IHL*, is provided to tell how long the header is, in 32-bit words. The minimum value is 5, which

applies when no options are present. The maximum value of this 4-bit field is 15, which limits the header to 60 bytes, and thus the options field to 40 bytes. For some options, such as one that records the route a packet has taken, 40 bytes is far too small, making the option useless.

The *Type of service* field allows the host to tell the subnet what kind of service it wants. Various combinations of reliability and speed are possible. For digitized voice, fast delivery beats accurate delivery. For file transfer, error-free transmission is more important than fast transmission.

The field itself contains (from left to right), a three-bit *Precedence* field, three flags, *D*, *T*, and *R*, and 2 unused bits. The *Precedence* field is a priority, from 0 (normal) to 7 (network control packet). The three flag bits allow the host to specify what it cares most about from the set {Delay, Throughput, Reliability}. In theory, these fields allow routers to make choices between, for example, a satellite link with high throughput and high delay or a leased line with low throughput and low delay. In practice, current routers ignore the *Type of Service* field altogether.

The *Total length* includes everything in the datagram—both header and data. The maximum length is 65,535 bytes. At present, this upper limit is tolerable, but with future gigabit networks larger datagrams may be needed.

The *Identification* field is needed to allow the destination host to determine which datagram a newly arrived fragment belongs to. All the fragments of a datagram contain the same *Identification* value.

Next comes an unused bit and then two 1-bit fields. *DF* stands for Don't Fragment. It is an order to the routers not to fragment the datagram because the destination is incapable of putting the pieces back together again. For example, when a computer boots, its ROM might ask for a memory image to be sent to it as a single datagram. By marking the datagram with the *DF* bit, the sender knows it will arrive in one piece, even if this means that the datagram must avoid a small-packet network on the best path and take a suboptimal route. All machines are required to accept fragments of 576 bytes or less.

MF stands for More Fragments. All fragments except the last one have this bit set. It is needed to know when all fragments of a datagram have arrived.

The *Fragment offset* tells where in the current datagram this fragment belongs. All fragments except the last one in a datagram must be a multiple of 8 bytes, the elementary fragment unit. Since 13 bits are provided, there is a maximum of 8192 fragments per datagram, giving a maximum datagram length of 65,536 bytes, one more than the *Total length* field.

The *Time to live* field is a counter used to limit packet lifetimes. It is supposed to count time in seconds, allowing a maximum lifetime of 255 sec. It must be decremented on each hop and is supposed to be decremented multiple times when queued for a long time in a router. In practice, it just counts hops. When it hits zero, the packet is discarded and a warning packet is sent back to the source host. This feature prevents datagrams for wandering around forever, something that otherwise might happen if the routing tables ever become corrupted.

When the network layer has assembled a complete datagram, it needs to know what to do with it. The *Protocol* field tells it which transport process to give it to. TCP is one possibility, but so are UDP and some others. The numbering of protocols is global across the entire Internet and is defined in RFC 1700.

The *Header checksum* verifies the header only. Such a checksum is useful for detecting errors generated by bad memory words inside a router. The algorithm is to add up all the 16-bit halfwords as they arrive, using one's complement arithmetic and then take the one's complement of the result. For purposes of this algorithm, the *Header checksum* is assumed to be zero upon arrival. This algorithm is more robust than using a normal add. Note that the *Header checksum* must be recomputed at each hop, because at least one field always changes (the *Time to live* field), but tricks can be used to speed up the computation.

The *Source address* and *Destination address* indicate the network number and host number. We will discuss Internet addresses in the next section. The *Options* field was designed to provide an escape to allow subsequent versions of the protocol to include information not present in the original design, to permit experimenters to try out new ideas, and to avoid allocating header bits to information that is rarely needed. The options are variable length. Each begins with a 1-byte code identifying the option. Some options are followed by a 1-byte option length field, and then one or more data bytes. The *Options* field is padded out to a multiple of four bytes. Currently five options are defined, as listed in Fig. 5-46, but not all routers support all of them.

Option	Description
Security	Specifies how secret the datagram is
Strict source routing	Gives the complete path to be followed
Loose source routing	Gives a list of routers not to be missed
Record route	Makes each router append its IP address
Timestamp	Makes each router append its address and timestamp

Fig. 5-46. IP options.

The *Security* option tells how secret the information is. In theory, a military router might use this field to specify not to route through certain countries the military considers to be "bad guys." In practice, all routers ignore it, so its only practical function is to help spies find the good stuff more easily.

The *Strict source routing* option gives the complete path from source to destination as a sequence of IP addresses. The datagram is required to follow that exact route. It is most useful for system managers to send emergency packets when the routing tables are corrupted, or for making timing measurements.

The *Loose source routing* option requires the packet to traverse the list of routers specified, and in the order specified, but it is allowed to pass through other

routers on the way. Normally, this option would only provide a few routers, to force a particular path. For example, to force a packet from London to Sydney to go west instead of east, this option might specify routers in New York, Los Angeles, and Honolulu. This option is most useful when political or economic considerations dictate passing through or avoiding certain countries.

The *Record route* option tells the routers along the path to append their IP address to the option field. This allows system managers to track down bugs in the routing algorithms ("Why are packets from Houston to Dallas all visiting Tokyo first?") When the ARPANET was first set up, no packet ever passed through more than nine routers, so 40 bytes of option was ample. As mentioned above, now it is too small.

Finally, the *Timestamp* option is like the *Record route* option, except that in addition to recording its 32-bit IP address, each router also records a 32-bit timestamp. This option, too, is mostly for debugging routing algorithms.

5.5.2. IP Addresses

Every host and router on the Internet has an IP address, which encodes its network number and host number. The combination is unique; no two machines have the same IP address. All IP addresses are 32 bits long and are used in the *Source address* and *Destination address* fields of IP packets. The formats used for IP address are shown in Fig. 5-47. Those machines connected to multiple networks have a different IP address on each network.

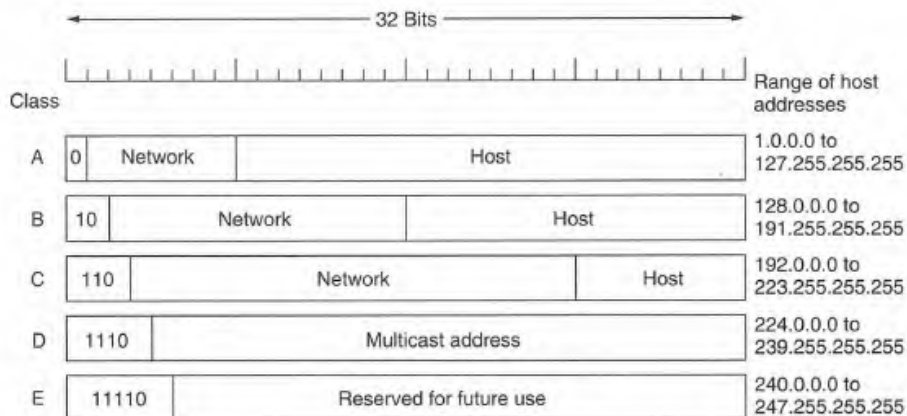


Fig. 5-47. IP address formats.

The class A, B, C, and D formats allow for up to 126 networks with 16 million hosts each, 16,382 networks with up to 64K hosts, 2 million networks, (e.g.,

LANs), with up to 254 hosts each, and multicast, in which a datagram is directed to multiple hosts. Addresses beginning with 11110 are reserved for future use. Tens of thousands of networks are now connected to the Internet, and the number doubles every year. Network numbers are assigned by the NIC (**Network Information Center**) to avoid conflicts.

Network addresses, which are 32-bit numbers, are usually written in **dotted decimal notation**. In this format, each of the 4 bytes is written in decimal, from 0 to 255. For example, the hexadecimal address C0290614 is written as 192.41.6.20. The lowest IP address is 0.0.0.0 and the highest is 255.255.255.255.

The values 0 and -1 have special meanings, as shown in Fig. 5-48. The value 0 means this network or this host. The value of -1 is used as a broadcast address to mean all hosts on the indicated network.

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	This host	
0 0 ... 0 0	Host	A host on this network
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	Broadcast on the local network	
Network	1 1 1 1 ... 1 1 1 1	Broadcast on a distant network
127	(Anything)	Loopback

Fig. 5-48. Special IP addresses.

The IP address 0.0.0.0 is used by hosts when they are being booted but is not used afterward. IP addresses with 0 as network number refer to the current network. These addresses allow machines to refer to their own network without knowing its number (but they have to know its class to know how many 0s to include). The address consisting of all 1s allows broadcasting on the local network, typically a LAN. The addresses with a proper network number and all 1s in the host field allow machines to send broadcast packets to distant LANs anywhere in the Internet. Finally, all addresses of the form 127.xx.yy.zz are reserved for loopback testing. Packets sent to that address are not put out onto the wire; they are processed locally and treated as incoming packets. This allows packets to be sent to the local network without the sender knowing its number. This feature is also used for debugging network software.

5.5.3. Subnets

As we have seen, all the hosts in a network must have the same network number. This property of IP addressing can cause problems as networks grow. For example, consider a company that starts out with one class C LAN on the

Internet. As time goes on, it might acquire more than 254 machines, and thus need a second class C address. Alternatively, it might acquire a second LAN of a different type and want a separate IP address for it (the LANs could be bridged to form a single IP network, but bridges have their own problems). Eventually, it might end up with many LANs, each with its own router and each with its own class C network number.

As the number of distinct local networks grows, managing them can become a serious headache. Every time a new network is installed the system administrator has to contact NIC to get a new network number. Then this number must be announced worldwide. Furthermore, moving a machine from one LAN to another requires it to change its IP address, which in turn may mean modifying its configuration files and also announcing the new IP address to the world. If some other machine is given the newly-released IP address, that machine will get email and other data intended for the original machine until the address has propagated all over the world.

The solution to these problems is to allow a network to be split into several parts for internal use but still act like a single network to the outside world. In the Internet literature, these parts are called **subnets**. As we mentioned in Chap. 1, this usage conflicts with “subnet” to mean the set of all routers and communication lines in a network. Hopefully it will be clear from the context which meaning is intended. In this section, the new definition will be the one used. If our growing company started up with a class B address instead of a class C address, it could start out just numbering the hosts from 1 to 254. When the second LAN arrived, it could decide, for example, to split the 16-bit host number into a 6-bit subnet number and a 10-bit host number, as shown in Fig. 5-49. This split allows 62 LANs (0 and -1 are reserved), each with up to 1022 hosts.

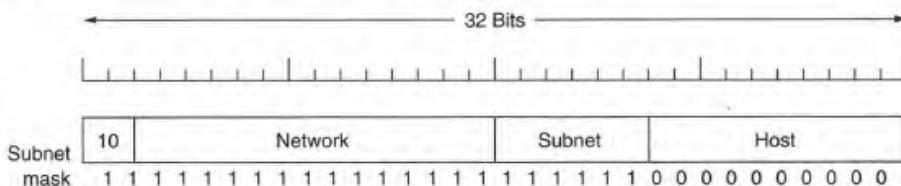


Fig. 5-49. One of the ways to subnet a class B network.

Outside the network, the subnetting is not visible, so allocating a new subnet does not require contacting NIC or changing any external databases. In this example, the first subnet might use IP addresses starting at 130.50.4.1, the second subnet might start at 130.50.8.1, and so on.

To see how subnets work, it is necessary to explain how IP packets are processed at a router. Each router has a table listing some number of (network, 0) IP addresses and some number of (this-network, host) IP addresses. The first kind

tells how to get to distant networks. The second kind tells how to get to local hosts. Associated with each table is the network interface to use to reach the destination, and certain other information.

When an IP packet arrives, its destination address is looked up in the routing table. If the packet is for a distant network, it is forwarded to the next router on the interface given in the table. If it is a local host (e.g., on the router's LAN), it is sent directly to the destination. If the network is not present, the packet is forwarded to a default router with more extensive tables. This algorithm means that each router only has to keep track of other networks and local hosts, not (network, host) pairs, greatly reducing the size of the routing table.

When subnetting is introduced, the routing tables are changed, adding entries of the form (this-network, subnet, 0) and (this-network, this-subnet, host). Thus a router on subnet k knows how to get to all the other subnets and also how to get to all the hosts on subnet k . It does not have to know the details about hosts on other subnets. In fact, all that needs to be changed is to have each router do a Boolean AND with the network's **subnet mask** (see Fig. 5-49) to get rid of the host number and look up the resulting address in its tables (after determining which network class it is). For example, a packet addressed to 130.50.15.6 and arriving at a router on subnet 5 is ANDed with the subnet mask of Fig. 5-49 to give the address 130.50.12.0. This address is looked up in the routing tables to find out how to get to hosts on subnet 3. The router on subnet 5 is thus spared the work of keeping track of the data link addresses of hosts other than those on subnet 5. Subnetting thus reduces router table space by creating a three-level hierarchy.

5.5.4. Internet Control Protocols

In addition to IP, which is used for data transfer, the Internet has several control protocols used in the network layer, including ICMP, ARP, RARP, and BOOTP. In this section we will look at each of these in turn.

The Internet Control Message Protocol

The operation of the Internet is monitored closely by the routers. When something unexpected occurs, the event is reported by the **ICMP (Internet Control Message Protocol)**, which is also used to test the Internet. About a dozen types of ICMP messages are defined. The most important ones are listed in Fig. 5-50. Each ICMP message type is encapsulated in an IP packet.

The **DESTINATION UNREACHABLE** message is used when the subnet or a router cannot locate the destination, or a packet with the *DF* bit cannot be delivered because a "small-packet" network stands in the way.

The **TIME EXCEEDED** message is sent when a packet is dropped due to its counter reaching zero. This event is a symptom that packets are looping, that there is enormous congestion, or that the timer values are being set too low.

Message type	Description
Destination unreachable	Packet could not be delivered
Time exceeded	Time to live field hit 0
Parameter problem	Invalid header field
Source quench	Choke packet
Redirect	Teach a router about geography
Echo request	Ask a machine if it is alive
Echo reply	Yes, I am alive
Timestamp request	Same as Echo request, but with timestamp
Timestamp reply	Same as Echo reply, but with timestamp

Fig. 5-50. The principal ICMP message types.

The **PARAMETER PROBLEM** message indicates that an illegal value has been detected in a header field. This problem indicates a bug in the sending host's IP software, or possibly in the software of a router transited.

The **SOURCE QUENCH** message was formerly used to throttle hosts that were sending too many packets. When a host received this message, it was expected to slow down. It is rarely used any more because when congestion occurs, these packets tend to add more fuel to the fire. Congestion control in the Internet is now done largely in the transport layer and will be studied in detail in Chap. 6.

The **REDIRECT** message is used when a router notices that a packet seems to be routed wrong. It is used by the router to tell the sending host about the probable error.

The **ECHO REQUEST** and **ECHO REPLY** messages are used to see if a given destination is reachable and alive. Upon receiving the **ECHO** message, the destination is expected to send an **ECHO REPLY** message back. The **TIMESTAMP REQUEST** and **TIMESTAMP REPLY** messages are similar, except that the arrival time of the message and the departure time of the reply are recorded in the reply. This facility is used to measure network performance.

In addition to these messages, there are four others that deal with Internet addressing, to allow hosts to discover their network numbers and to handle the case of multiple LANs sharing a single IP address. ICMP is defined in RFC 792.

The Address Resolution Protocol

Although every machine on the Internet has one (or more) IP addresses, these cannot actually be used for sending packets because the data link layer hardware does not understand Internet addresses. Nowadays, most hosts are attached to a

LAN by an interface board that only understands LAN addresses. For example, every Ethernet board ever manufactured comes equipped with a 48-bit Ethernet address. Manufacturers of Ethernet boards request a block of addresses from a central authority to ensure that no two boards have the same address (to avoid conflicts should the two boards ever appear on the same LAN). The boards send and receive frames based on 48-bit Ethernet addresses. They know nothing at all about 32-bit IP addresses.

The question now arises: How do IP addresses get mapped onto data link layer addresses, such as Ethernet? To explain how this works, let us use the example of Fig. 5-51, in which a small university with several class C networks is illustrated. Here we have two Ethernets, one in the Computer Science department, with IP address 192.31.65.0 and one in Electrical Engineering, with IP address 192.31.63.0. These are connected by a campus FDDI ring with IP address 192.31.60.0. Each machine on an Ethernet has a unique Ethernet address, labeled *E1* through *E6*, and each machine on the FDDI ring has an FDDI address, labeled *F1* through *F3*.

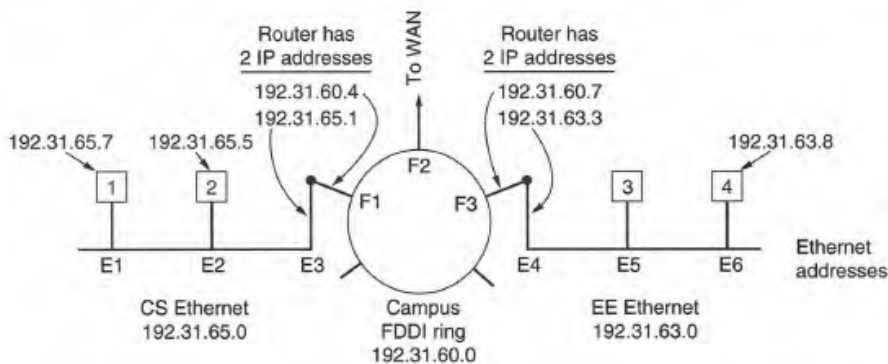


Fig. 5-51. Three interconnected class C networks: two Ethernets and an FDDI ring.

Let us start out by seeing how a user on host 1 sends a packet to a user on host 2. Let us assume the sender knows the name of the intended receiver, possibly something like *mary@eagle.cs.uni.edu*. The first step is to find the IP address for host 2, known as *eagle.cs.uni.edu*. This lookup is performed by the Domain Name System, which we will study in Chap. 7. For the moment, we will just assume that DNS returns the IP address for host 2 (192.31.65.5).

The upper layer software on host 1 now builds a packet with 192.31.65.5 in the *Destination address* field and gives it to the IP software to transmit. The IP software can look at the address and see that the destination is on its own network, but it needs a way to find the destination's Ethernet address. One solution is to have a configuration file somewhere in the system that maps IP addresses onto

Ethernet addresses. This solution is certainly possible, but for organizations with thousands of machines, keeping these files up to date is an error-prone, time-consuming job.

A better solution is for host 1 to output a broadcast packet onto the Ethernet asking: "Who owns IP address 192.31.65.5?" The broadcast will arrive at every machine on Ethernet 192.31.65.0, and each one will check its IP address. Host 2 alone will respond with its Ethernet address (*E2*). In this way host 1 learns that IP address 192.31.65.5 is on the host with Ethernet address *E2*. The protocol for asking this question and getting the reply is called **ARP (Address Resolution Protocol)**. Almost every machine on the Internet runs it. It is defined in RFC 826.

The advantage of using ARP over configuration files is the simplicity. The system manager does not have to do much except assign each machine an IP address and decide about subnet masks. ARP does the rest.

At this point, the IP software on host 1 builds an Ethernet frame addressed to *E2*, puts the IP packet (addressed to 192.31.65.5) in the payload field, and dumps it onto the Ethernet. The Ethernet board of host 2 detects this frame, recognizes it as a frame for itself, scoops it up, and causes an interrupt. The Ethernet driver extracts the IP packet from the payload and passes it to the IP software, which sees that it is correctly addressed, and processes it.

Various optimizations are possible to make ARP more efficient. To start with, once a machine has run ARP, it caches the result in case it needs to contact the same machine shortly. Next time it will find the mapping in its own cache, thus eliminating the need for a second broadcast. In many cases host 2 will need to send back a reply, forcing it, too, to run ARP to determine the sender's Ethernet address. This ARP broadcast can be avoided by having host 1 include its IP to Ethernet mapping in the ARP packet. When ARP broadcast arrives at host 2, the pair (192.31.65.7, *E1*) is entered into host 2's ARP cache for future use. In fact, all machines on the Ethernet can enter this mapping into their ARP caches.

Yet another optimization is to have every machine broadcast its mapping when it boots. This broadcast is generally done in the form of an ARP looking for its own IP address. There should not be a response, but a side effect of the broadcast is to make any entry in everyone's ARP cache. If a response does arrive, two machines have been assigned the same IP address. The new one should inform the system manager and not boot.

To allow mappings to change, for example, when an Ethernet board breaks and is replaced with a new one (and thus a new Ethernet address), entries in the ARP cache should time out after a few minutes.

Now let us look at Fig. 5-51 again, only this time host 1 wants to send a packet to host 4 (192.31.63.8). Using ARP will fail because host 4 will not see the broadcast (routers do not forward Ethernet-level broadcasts). There are two solutions. First, the CS router could be configured to respond to ARP requests for network 192.31.63.0 (and possibly other local networks). In this case, host 1 will make an ARP cache entry of (192.31.63.8, *E3*) and happily send all traffic for host

4 to the local router. This solution is called **proxy ARP**. The second solution is to have host 1 immediately see that the destination is on a remote network and just send all such traffic to a default Ethernet address that handles all remote traffic, in this case *E3*. This solution does not require having the CS router know which remote networks it is serving.

Either way, what happens is that host 1 packs the IP packet into the payload field of an Ethernet frame addressed to *E3*. When the CS router gets the Ethernet frame, it removes the IP packet from the payload field and looks up the IP address in its routing tables. It discovers that packets for network 192.31.63.0 are supposed to go to router 192.31.60.7. If it does not already know the FDDI address of 192.31.60.7, it broadcasts an ARP packet onto the ring and learns that its ring address is *F3*. It then inserts the packet into the payload field of an FDDI frame addressed to *F3* and puts it on the ring.

At the EE router, the FDDI driver removes the packet from the payload field and gives it to the IP software, which sees that it needs to send the packet to 192.31.63.8. If this IP address is not in its ARP cache, it broadcasts an ARP request on the EE Ethernet and learns that the destination address is *E6* so it builds an Ethernet frame addressed to *E6*, puts the packet in the payload field, and sends it over the Ethernet. When the Ethernet frame arrives at host 4, the packet is extracted from the frame and passed to the IP software for processing.

Going from host 1 to a distant network over a WAN works essentially the same way, except that this time the CS router's tables tell it to use the WAN router whose FDDI address is *F2*.

The Reverse Address Resolution Protocol

ARP solves the problem of finding out which Ethernet address corresponds to a given IP address. Sometimes the reverse problem has to be solved: Given an Ethernet address, what is the corresponding IP address? In particular, this problem occurs when booting a diskless workstation. Such a machine will normally get the binary image of its operating system from a remote file server. But how does it learn its IP address?

The solution is to use the **RARP (Reverse Address Resolution Protocol)** (defined in RFC 903). This protocol allows a newly-booted workstation to broadcast its Ethernet address and say: "My 48-bit Ethernet address is 14.04.05.18.01.25. Does anyone out there know my IP address?" The RARP server sees this request, looks up the Ethernet address in its configuration files, and sends back the corresponding IP address.

Using RARP is better than embedding an IP address in the memory image because it allows the same image to be used on all machines. If the IP address were buried inside the image, each workstation would need its own image.

A disadvantage of RARP is that it uses a destination address of all 1s (limited broadcasting) to reach the RARP server. However, such broadcasts are not

forwarded by routers, so a RARP server is needed on each network. To get around this problem, an alternative bootstrap protocol called **BOOTP** has been invented (see RFCs 951, 1048, and 1084). Unlike RARP, it uses UDP messages, which are forwarded over routers. It also provides a diskless workstation with additional information, including the IP address of the file server holding the memory image, the IP address of the default router, and the subnet mask to use. BOOTP is described in RFC 951.

5.5.5. The Interior Gateway Routing Protocol: OSPF

As we mentioned earlier, the Internet is made up of a large number of autonomous systems. Each AS is operated by a different organization and can use its own routing algorithm inside. For example, the internal networks of companies X, Y, and Z would usually be seen as three ASes if all three were on the Internet. All three may use different routing algorithms internally. Nevertheless, having standards, even for internal routing, simplifies the implementation at the boundaries between ASes and allows reuse of code. In this section we will study routing within an AS. In the next one, we will look at routing between ASes. A routing algorithm within an AS is called an **interior gateway protocol**; an algorithm for routing between ASes is called an **exterior gateway protocol**.

The original Internet interior gateway protocol was a distance vector protocol (RIP) based on the Bellman-Ford algorithm. It worked well in small systems, but less well as ASes got larger. It also suffered from the count-to-infinity problem and generally slow convergence, so it was replaced in May 1979 by a link state protocol. In 1988, the Internet Engineering Task Force began work on a successor. That successor, called **OSPF (Open Shortest Path First)** became a standard in 1990. Many router vendors are now supporting it, and it will become the main interior gateway protocol in the near future. Below we will give a sketch of how OSPF works. For the complete story, see RFC 1247.

Given the long experience with other routing protocols, the group designing the new protocol had a long list of requirements that had to be met. First, the algorithm had to be published in the open literature, hence the "O" in OSPF. A proprietary solution owned by one company would not do. Second, the new protocol had to support a variety of distance metrics, including physical distance, delay, and so on. Third, it had to be a dynamic algorithm, one that adapted to changes in the topology automatically and quickly.

Fourth, and new for OSPF, it had to support routing based on type of service. The new protocol had to be able to route real-time traffic one way and other traffic a different way. The IP protocol has a *Type of Service* field, but no existing routing protocol used it.

Fifth, and related to the above, the new protocol had to do load balancing, splitting the load over multiple lines. Most previous protocols sent all packets

over the best route. The second-best route was not used at all. In many cases, splitting the load over multiple lines gives better performance.

Sixth, support for hierarchical systems was needed. By 1988, the Internet had grown so large that no router could be expected to know the entire topology. The new routing protocol had to be designed so that no router would have to.

Seventh, some modicum of security was required to prevent fun-loving students from spoofing routers by sending them false routing information. Finally, provision was needed for dealing with routers that were connected to the Internet via a tunnel. Previous protocols did not handle this well.

OSPF supports three kinds of connections and networks:

1. Point-to-point lines between exactly two routers.
2. Multiaccess networks with broadcasting (e.g., most LANs).
3. Multiaccess networks without broadcasting (e.g., most packet-switched WANs).

A **multiaccess** network is one that can have multiple routers on it, each of which can directly communicate with all the others. All LANs and WANs have this property. Figure 5-52(a) shows an AS containing all three kinds of networks. Note that hosts do not generally play a role in OSPF.

OSPF works by abstracting the collection of actual networks, routers, and lines into a directed graph in which each arc is assigned a cost (distance, delay, etc.). It then computes the shortest path based on the weights on the arcs. A serial connection between two routers is represented by a pair of arcs, one in each direction. Their weights may be different. A multiaccess network is represented by a node for the network itself plus a node for each router. The arcs from the network node to the routers have weight 0 and are omitted from the graph.

Figure 5-52(b) shows the graph representation of the network of Fig. 5-52(a). What OSPF fundamentally does is represent the actual network as a graph like this and then compute the shortest path from every router to every other router.

Many of the ASes in the Internet are themselves large and nontrivial to manage. OSPF allows them to be divided up into numbered **areas**, where an area is a network or a set of contiguous networks. Areas do not overlap but need not be exhaustive, that is, some routers may belong to no area. An area is a generalization of a subnet. Outside an area, its topology and details are not visible.

Every AS has a **backbone** area, called area 0. All areas are connected to the backbone, possibly by tunnels, so it is possible to go from any area in the AS to any other area in the AS via the backbone. A tunnel is represented in the graph as an arc and has a cost. Each router that is connected to two or more areas is part of the backbone. As with other areas, the topology of the backbone is not visible outside the backbone.

Within an area, each router has the same link state database and runs the same shortest path algorithm. Its main job is to calculate the shortest path from itself to

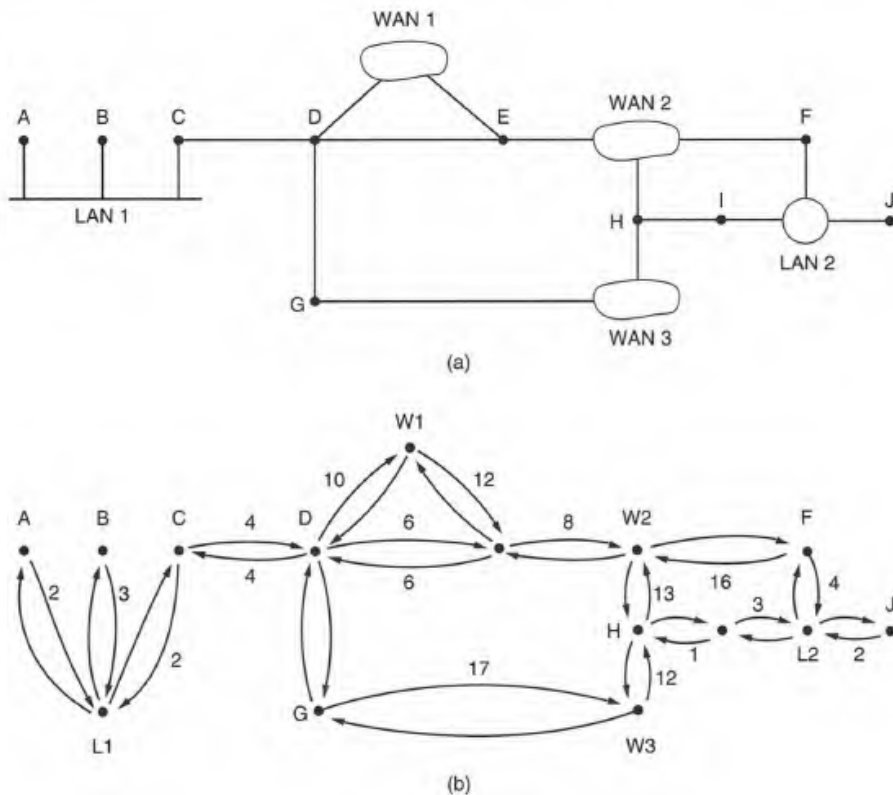


Fig. 5-52. (a) An autonomous system. (b) A graph representation of (a).

every other router in the area, including the router that is connected to the backbone, of which there must be at least one. A router that connects to two areas needs the databases for both areas and must run the shortest path algorithm for each one separately.

The way OSPF handles type of service routing is to have multiple graphs, one labeled with the costs when delay is the metric, one labeled with the costs when throughput is the metric, and one labeled with the costs when reliability is the metric. Although this triples the computation needed, it allows separate routes for optimizing delay, throughput, and reliability.

During normal operation, three kinds of routes may be needed: intra-area, interarea, and interAS. Intra-area routes are the easiest, since the source router already knows the shortest path to the destination router. Interarea routing always proceeds in three steps: go from the source to the backbone; go across the backbone to the destination area; go to the destination. This algorithm forces a star

configuration on OSPF with the backbone being the hub and the other areas being spokes. Packets are routed from source to destination “as is.” They are not encapsulated or tunneled, unless going to an area whose only connection to the backbone is a tunnel. Figure 5-53 shows part of the Internet with ASes and areas.

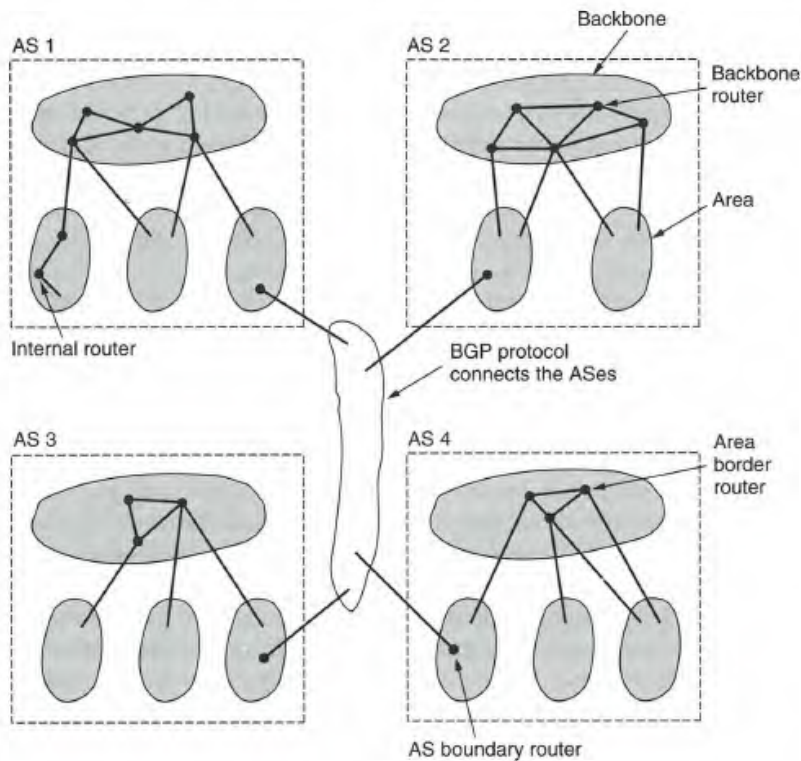


Fig. 5-53. The relation between ASes, backbones, and areas in OSPF.

OSPF distinguishes four classes of routers:

1. Internal routers are wholly within one area.
2. Area border routers connect two or more areas.
3. Backbone routers are on the backbone.
4. AS boundary routers talk to routers in other ASes.

These classes are allowed to overlap. For example, all the border routers are automatically part of the backbone. In addition, a router that is in the backbone

but not part of any other area is also an internal router. Examples of all four classes of routers are illustrated in Fig. 5-53.

When a router boots, it sends HELLO messages on all of its point-to-point lines and multicasts them on LANs to the group consisting of all the other routers. On WANs, it needs some configuration information to know who to contact. From the responses, each router learns who its neighbors are.

OSPF works by exchanging information between **adjacent** routers, which is not the same as between neighboring routers. In particular, it is inefficient to have every router on a LAN talk to every other router on the LAN. To avoid this situation, one router is elected as the **designated router**. It is said to be adjacent to all the other routers, and exchanges information with them. Neighboring routers that are not adjacent do not exchange information with each other. A backup designated router is always kept up to date to ease the transition should the primary designated router crash.

During normal operation, each router periodically floods LINK STATE UPDATE messages to each of its adjacent routers. This message gives its state and provides the costs used in the topological database. The flooding messages are acknowledged, to make them reliable. Each message has a sequence number, so a router can see whether an incoming LINK STATE UPDATE is older or newer than what it currently has. Routers also send these messages when a line goes up or down or its cost changes.

DATABASE DESCRIPTION messages give the sequence numbers of all the link state entries currently held by the sender. By comparing its own values with those of the sender, the receiver can determine who has the most recent values. These messages are used when a line is brought up.

Either partner can request link state information from the other one using LINK STATE REQUEST messages. The net result of this algorithm is that each pair of adjacent routers checks to see who has the most recent data, and new information is spread throughout the area this way. All these messages are sent as raw IP packets. The five kinds of messages are summarized in Fig. 5-54.

Message type	Description
Hello	Used to discover who the neighbors are
Link state update	Provides the sender's costs to its neighbors
Link state ack	Acknowledges link state update
Database description	Announces which updates the sender has
Link state request	Requests information from the partner

Fig. 5-54. The five types of OSPF messages.

Finally, we can put all the pieces together. Using flooding, each router informs all the other routers in its area of its neighbors and costs. This

information allows each router to construct the graph for its area(s) and compute the shortest path. The backbone area does this too. In addition, the backbone routers accept information from the area border routers in order to compute the best route from each backbone router to every other router. This information is propagated back to the area border routers, which advertise it within their areas. Using this information, a router about to send an interarea packet can select the best exit router to the backbone.

5.5.6. The Exterior Gateway Routing Protocol: BGP

Within a single AS, the recommended routing protocol on the Internet is OSPF (although it is certainly not the only one in use). Between ASes, a different protocol, **BGP (Border Gateway Protocol)**, is used. A different protocol is needed between ASes because the goals of an interior gateway protocol and an exterior gateway protocol are not the same. All an interior gateway protocol has to do is move packets as efficiently as possible from the source to the destination. It does not have to worry about politics.

Exterior gateway protocol routers have to worry about politics a great deal. For example, a corporate AS might want the ability to send packets to any Internet site and receive packets from any Internet site. However, it might be unwilling to carry transit packets originating in a foreign AS and ending in a different foreign AS, even if its own AS was on the shortest path between the two foreign ASes ("That's their problem, not ours"). On the other hand, it might be willing to carry transit traffic for its neighbors, or even for specific other ASes that paid it for this service. Telephone companies, for example, might be happy to act as a carrier for their customers, but not for others. Exterior gateway protocols in general, and BGP in particular, have been designed to allow many kinds of routing policies to be enforced in the interAS traffic.

Typical policies involve political, security, or economic considerations. A few examples of routing constraints are

1. No transit traffic through certain ASes.
2. Never put Iraq on a route starting at the Pentagon.
3. Do not use the United States to get from British Columbia to Ontario.
4. Only transit Albania if there is no alternative to the destination.
5. Traffic starting or ending at IBM® should not transit Microsoft®.

Policies are manually configured into each BGP router. They are not part of the protocol itself.

From the point of view of a BGP router, the world consists of other BGP routers and the lines connecting them. Two BGP routers are considered connected if they share a common network. Given BGP's special interest in transit

traffic, networks are grouped into one of three categories. The first category is the **stub networks**, which have only one connection to the BGP graph. These cannot be used for transit traffic because there is no one on the other side. Then come the **multiconnected networks**. These could be used for transit traffic, except that they refuse. Finally, there are the **transit networks**, such as backbones, which are willing to handle third-party packets, possibly with some restrictions.

Pairs of BGP routers communicate with each other by establishing TCP connections. Operating this way provides reliable communication and hides all the details of the network being passed through.

BGP is fundamentally a distance vector protocol, but quite different from most others such as RIP. Instead of maintaining just the cost to each destination, each BGP router keeps track of the exact path used. Similarly, instead of periodically giving each neighbor its estimated cost to each possible destination, each BGP router tells its neighbors the exact path it is using.

As an example, consider the BGP routers shown in Fig. 5-55(a). In particular, consider *F*'s routing table. Suppose that it uses the path *FGCD* to get to *D*. When the neighbors give it routing information, they provide their complete paths, as shown in Fig. 5-55(b) (for simplicity, only destination *D* is shown here).

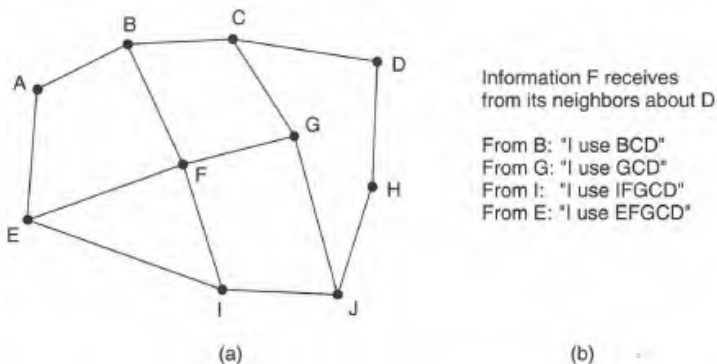


Fig. 5-55. (a) A set of BGP routers. (b) Information sent to *F*.

After all the paths come in from the neighbors, *F* examines them to see which is the best. It quickly discards the paths from *I* and *E*, since these paths pass through *F* itself. The choice is then between using *B* and *G*. Every BGP router contains a module that examines routes to a given destination and scores them, returning a number for the "distance" to that destination for each route. Any route violating a policy constraint automatically gets a score of infinity. The router then adopts the route with the shortest distance. The scoring function is not part of the BGP protocol and can be any function the system managers want.

BGP easily solves the count-to-infinity problem that plagues other distance vector routing algorithms. For example, suppose *G* crashes or the line *FG* goes

down. *F* then receives routes from its three remaining neighbors. These routes are *BCD*, *IFGCD*, and *EFGCD*. It can immediately see that the two latter routes are pointless, since they pass through *F* itself, so it chooses *FBCD* as its new route. Other distance vector algorithms often make the wrong choice because they cannot tell which of their neighbors have independent routes to the destination, and which do not. The current definition of BGP is in RFC 1654. Additional useful information can be found in RFC 1268.

5.5.7. Internet Multicasting

Normal IP communication is between one sender and one receiver. However, for some applications it is useful for a process to be able to send to a large number of receivers simultaneously. Examples are updating replicated, distributed databases, transmitting stock quotes to multiple brokers, and handling digital conference (i.e., multiparty) telephone calls.

IP supports multicasting, using class D addresses. Each class D address identifies a group of hosts. Twenty-eight bits are available for identifying groups, so over 250 million groups can exist at the same time. When a process sends a packet to a class D address, a best-efforts attempt is made to deliver it to all the members of the group addressed, but no guarantees are given. Some members may not get the packet.

Two kinds of group addresses are supported: permanent addresses and temporary ones. A permanent group is always there and does not have to be set up. Each permanent group has a permanent group address. Some examples of permanent group addresses are

- 224.0.0.1 All systems on a LAN
- 224.0.0.2 All routers on a LAN
- 224.0.0.5 All OSPF routers on a LAN
- 224.0.0.6 All designated OSPF routers on a LAN

Temporary groups must be created before they can be used. A process can ask its host to join a specific group. It can also ask its host to leave the group. When the last process on a host leaves a group, that group is no longer present on the host. Each host keeps track of which groups its processes currently belong to.

Multicasting is implemented by special multicast routers, which may or may not be colocated with the standard routers. About once a minute, each multicast router sends a hardware (i.e., data link layer) multicast to the hosts on its LAN (address 224.0.0.1) asking them to report back on the groups their processes currently belong to. Each host sends back responses for all the class D addresses it is interested in.

These query and response packets use a protocol called **IGMP (Internet Group Management Protocol)**, which is vaguely analogous to ICMP. It has only two kinds of packets: query and response, each with a simple fixed format

containing some control information in the first word of the payload field and a class D address in the second word. It is described in RFC 1112.

Multicast routing is done using spanning trees. Each multicast router exchanges information with its neighbors using a modified distance vector protocol in order for each one to construct a spanning tree per group covering all group members. Various optimizations are used to prune the tree to eliminate routers and networks not interested in particular groups. The protocol makes heavy use of tunneling to avoid bothering nodes not in a spanning tree.

5.5.8. Mobile IP

Many users of the Internet have portable computers and want to stay connected to the Internet when they visit a distant Internet site and even on the road in between. Unfortunately, the IP addressing system makes working far from home easier said than done. In this section we will examine the problem and the solution. A more detailed description is given in (Johnson, 1995).

The real villain is the addressing scheme itself. Every IP address contains three fields: the class, the network number, and the host number. For example, consider the machine with IP address 160.80.40.20. The 160.80 gives the class (B) and network number (8272); the 40.20 is the host number (10260). Routers all over the world have routing tables telling which line to use to get to network 160.80. Whenever a packet comes in with a destination IP address of the form 160.80.xxx.yyy, it goes out on that line.

If all of a sudden, the machine with that address is carted off to some distant site, the packets for it will continue to be routed to its home LAN (or router). The owner will no longer get email, and so on. Giving the machine a new IP address corresponding to its new location is unattractive because large numbers of people, programs, and databases would have to be informed of the change.

Another approach is to have the routers use complete IP addresses for routing, instead of just the class and network. However, this strategy would require each router to have millions of table entries, at astronomical cost to the Internet.

When people began demanding the ability to have mobile hosts, the IETF set up a Working Group to find a solution. The Working Group quickly formulated a number of goals considered desirable in any solution. The major ones were

1. Each mobile host must be able to use its home IP address anywhere.
2. Software changes to the fixed hosts were not permitted.
3. Changes to the router software and tables were not permitted.
4. Most packets for mobile hosts should not make detours on the way.
5. No overhead should be incurred when a mobile host is at home.

The solution chosen was the one described in Sec. 5.2.8. To review it briefly, every site that wants to allow its users to roam has to create a home agent. Every

site that wants to allow visitors has to create a foreign agent. When a mobile host shows up at a foreign site, it contacts the foreign host there and registers. The foreign host then contacts the user's home agent and gives it a **care-of address**, normally the foreign agent's own IP address.

When a packet arrives at the user's home LAN, it comes in at some router attached to the LAN. The router then tries to locate the host in the usual way, by broadcasting an ARP packet asking, for example: "What is the Ethernet address of 160.80.40.20?" The home agent responds to this query by giving its own Ethernet address. The router then sends packets for 160.80.40.20 to the home agent. It, in turn, tunnels them to the care-of address by encapsulating them in the payload field of an IP packet addressed to the foreign agent. The foreign agent then decapsulates and delivers them to the data link address of the mobile host. In addition, the home agent gives the care-of address to the sender, so future packets can be tunnelled directly to the foreign agent. This solution meets all the requirements stated above.

One small detail is probably worth mentioning. At the time the mobile host moves, the router probably has its (soon-to-be-invalid) Ethernet address cached. To replace that Ethernet address with the home agent's, a trick called **gratuitous ARP** is used. This is a special, unsolicited message to the router that causes it to replace a specific cache entry, in this case, that of the mobile host about to leave. When the mobile host returns later, the same trick is used to update the router's cache again.

Nothing in the design prevents a mobile host from being its own foreign agent, but that approach only works if the mobile host (in its capacity as foreign agent) is logically connected to the Internet at its current site. Also, it must be able to acquire a (temporary) care-of IP address to use. That IP address must belong to the LAN to which it is currently attached.

The IETF solution for mobile hosts solves a number of other problems not mentioned so far. For example, how are agents located? The solution is for each agent to periodically broadcast its address and the type of services it is willing to provide (e.g., home, foreign, or both). When a mobile host arrives somewhere, it can just listen for these broadcasts, called **advertisements**. Alternatively, it can broadcast a packet announcing its arrival and hope that the local foreign agent responds to it.

Another problem that had to be solved is what to do about impolite mobile hosts that leave without saying goodbye. The solution is to make registration valid only for a fixed time interval. If it is not refreshed periodically, it times out, so the foreign host can clear its tables.

Yet another issue is security. When a home agent gets a message asking it to please forward all of Nora's packets to some IP address, it had better not comply unless it is convinced that Nora is the source of this request, and not somebody trying to impersonate her. Cryptographic authentication protocols are used for this purpose. We will study such protocols in Chap. 7.

A final point addressed by the Working Group relates to levels of mobility. Imagine an airplane with an on-board Ethernet used by the navigation and avionics computers. On this Ethernet is a standard router that talks to the wired Internet on the ground over a radio link. One fine day, some clever marketing executive gets the idea to install Ethernet connectors in all the arm rests so passengers with mobile computers can also plug in.

Now we have two levels of mobility: the aircraft's own computers, which are stationary with respect to the Ethernet, and the passengers' computers, which are mobile with respect to it. In addition, the on-board router is mobile with respect to routers on the ground. Being mobile with respect to a system that is itself mobile can be handled using recursive tunneling.

5.5.9. CIDR—Classless InterDomain Routing

IP has been in heavy use for over a decade. It has worked extremely well, as demonstrated by the exponential growth of the Internet. Unfortunately, IP is rapidly becoming a victim of its own popularity: it is running out of addresses. This looming disaster has sparked a great deal of discussion and controversy within the Internet community about what to do about it. In this section we will describe both the problem and several proposed solutions. A more complete description is given in (Huitema, 1996).

Back in 1987, a few visionaries predicted that some day the Internet might grow to 100,000 networks. Most experts pooh-poohed this as being decades in the future, if ever. The 100,000th network was connected in 1996. The problem, simply stated, is that the Internet is rapidly running out of IP addresses. In principle, over 2 billion addresses exist, but the practice of organizing the address space by classes (see Fig. 5-47), wastes millions of them. In particular, the real villain is the class B network. For most organizations, a class A network, with 16 million addresses is too big, and a class C network, with 256 addresses is too small. A class B network, with 65,536, is just right. In Internet folklore, this situation is known as the **three bears problem** (as in *Goldilocks and the Three Bears*).

In reality, a class B address is far too large for most organizations. Studies have shown that more than half of all class B networks have fewer than 50 hosts. A class C network would have done the job, but no doubt every organization that asked for a class B address thought that one day it would outgrow the 8-bit host field. In retrospect, it might have been better to have had class C networks use 10 bits instead of eight for the host number, allowing 1022 hosts per network. Had this been the case, most organizations would have probably settled for a class C network, and there would have been half a million of them (versus only 16,384 class B networks).

However, then another problem would have emerged more quickly: the routing table explosion. From the point of view of the routers, the IP address space is

a two-level hierarchy, with network numbers and host numbers. Routers do not have to know about all the hosts, but they do have to know about all the networks. If half a million class C networks were in use, every router in the entire Internet would need a table with half a million entries, one per network, telling which line to use to get to that network, as well as other information.

The actual physical storage of half a million entry tables is probably doable, although expensive for critical routers that keep the tables in static RAM on I/O boards. A more serious problem is that the complexity of various algorithms relating to management of the tables grows faster than linear. Worse yet, much of the existing router software and firmware was designed at a time when the Internet had 1000 connected networks and 10,000 networks seemed decades away. Design choices made then often are far from optimal now.

In addition, various routing algorithms require each router to transmit its tables periodically. The larger the tables, the more likely some parts will get lost underway, leading to incomplete data at the other end and possibly routing instabilities.

The routing table problem could have been solved by going to a deeper hierarchy. For example, having each IP address contain a country, state, city, network, and host field might work. Then each router would only need to know how to get to each country, the states or provinces in its own country, the cities in its state or province, and the networks in its city. Unfortunately, this solution would require considerably more than 32 bits for IP addresses and would use addresses inefficiently (Liechtenstein would have as many bits as the United States).

In short, most solutions solve one problem but create a new one. One solution that is now being implemented and which will give the Internet a bit of extra breathing room is **CIDR (Classless InterDomain Routing)**. The basic idea behind CIDR, which is described in RFC 1519, is to allocate the remaining class C networks, of which there are almost two million, in variable-sized blocks. If a site needs, say, 2000 addresses, it is given a block of 2048 addresses (eight contiguous class C networks), and not a full class B address. Similarly, a site needing 8000 addresses gets 8192 addresses (32 contiguous class C networks).

In addition to using blocks of contiguous class C networks as units, the allocation rules for the class C addresses were also changed in RFC 1519. The world was partitioned into four zones, and each one given a portion of the class C address space. The allocation was as follows:

Addresses 194.0.0.0 to 195.255.255.255 are for Europe

Addresses 198.0.0.0 to 199.255.255.255 are for North America

Addresses 200.0.0.0 to 201.255.255.255 are for Central and South America

Addresses 202.0.0.0 to 203.255.255.255 are for Asia and the Pacific

In this way, each region was given about 32 million addresses to allocate, with another 320 million class C addresses from 204.0.0.0 through 223.255.255.255

held in reserve for the future. The advantage of this allocation is that now any router outside of Europe that gets a packet addressed to 194.xx.yy.zz or 195.xx.yy.zz can just send it to its standard European gateway. In effect 32 million addresses have now been compressed into one routing table entry. Similarly for the other regions.

Of course, once a 194.xx.yy.zz packet gets to Europe, more detailed routing tables are needed. One possibility is to have 131,072 entries for networks 194.0.0.xx through 195.255.255.xx, but this is precisely this routing table explosion that we are trying to avoid. Instead, each routing table entry is extended by giving it a 32-bit mask. When a packet comes in, its destination address is first extracted. Then (conceptually) the routing table is scanned entry by entry, masking the destination address and comparing it to the table entry looking for a match.

To make this comparison process clearer, let us consider an example. Suppose that Cambridge University needs 2048 addresses and is assigned the addresses 194.24.0.0 through 194.24.7.255, along with mask 255.255.248.0. Next, Oxford University asks for 4096 addresses. Since a block of 4096 addresses must lie on a 4096-byte boundary, they cannot be given addresses starting at 194.24.8.0. Instead they get 194.24.16.0 through 194.24.31.255 along with mask 255.255.240.0. Now the University of Edinburgh asks for 1024 addresses and is assigned addresses 194.24.8.0 through 194.24.11.255 and mask 255.255.252.0.

The routing tables all over Europe are now updated with three entries, each one containing a base address and a mask. These entries (in binary) are:

Address	Mask
11000010 00011000 00000000 00000000	11111111 11111111 11111000 00000000
11000010 00011000 00010000 00000000	11111111 11111111 11110000 00000000
11000010 00011000 00001000 00000000	11111111 11111111 11111100 00000000

Now consider what happens when a packet comes in addressed to 194.24.17.4, which in binary is

```
11000010 00011000 00010001 00000100
```

First it is Boolean ANDed with the Cambridge mask to get

```
11000010 00011000 00010000 00000000
```

This value does not match the Cambridge base address, so the original address is next ANDed with the Oxford mask to get

```
11000010 00011000 00010000 00000000
```

This value does match the Oxford base, so the packet is sent to the Oxford router. In practice, the router entries are not tried sequentially; indexing tricks are used to speed up the search. Also, it is possible for two entries to match, in which case the one whose mask has the most 1 bits wins. Finally, the same idea can be applied to all addresses, not just the new class C addresses, so with CIDR, the old

class A, B, and C networks are no longer used for routing. This is why CIDR is called classless routing. CIDR is described in more detail in (Ford et al., 1993; and Huitema, 1995).

5.5.10. IPv6

While CIDR may buy a few more years' time, everyone realizes that the days of IP in its current form (IPv4) are numbered. In addition to these technical problems, there is another issue looming in the background. Up until recently, the Internet has been used largely by universities, high-tech industry, and the government (especially the Dept. of Defense). With the explosion of interest in the Internet starting in the mid 1990s, it is likely that in the next millenium, it will be used by a much larger group of people, especially people with different requirements. For one thing, millions of people with wireless portables may use it to keep in contact with their home bases. For another, with the impending convergence of the computer, communication, and entertainment industries, it may not be long before every television set in the world is an Internet node, producing a billion machines being used for video on demand. Under these circumstances, it became apparent that IP had to evolve and become more flexible.

Seeing these problems on the horizon, in 1990, IETF started work on a new version of IP, one which would never run out of addresses, would solve a variety of other problems, and be more flexible and efficient as well. Its major goals were to

1. Support billions of hosts, even with inefficient address space allocation.
2. Reduce the size of the routing tables.
3. Simplify the protocol, to allow routers to process packets faster.
4. Provide better security (authentication and privacy) than current IP.
5. Pay more attention to type of service, particularly for real-time data.
6. Aid multicasting by allowing scopes to be specified.
7. Make it possible for a host to roam without changing its address.
8. Allow the protocol to evolve in the future.
9. Permit the old and new protocols to coexist for years.

To find a protocol that met all these requirements, IETF issued a call for proposals and discussion in RFC 1550. Twenty-one responses were received, not all of them full proposals. By December 1992, seven serious proposals were on the table. They ranged from making minor patches to IP, to throwing it out altogether and replacing with a completely different protocol.

One proposal was to run TCP over CLNP, which, with its 160-bit addresses would have provided enough address space forever and would have unified two major network layer protocols. However, many people felt that this would have been an admission that something in the OSI world was actually done right, a statement considered Politically Incorrect in Internet circles. CLNP was patterned closely on IP, so the two are not really that different. In fact, the protocol ultimately chosen differs from IP far more than CLNP does. Another strike against CLNP was its poor support for service types, something required to transmit multimedia efficiently.

Three of the better proposals were published in *IEEE Network* (Deering, 1993; Francis, 1993; and Katz and Ford, 1993). After much discussion, revision, and jockeying for position, a modified combined version of the Deering and Francis proposals, by now called **SIPP (Simple Internet Protocol Plus)** was selected and given the designation **IPv6** (IPv5 was already in use for an experimental real-time stream protocol).

IPv6 meets the goals fairly well. It maintains the good features of IP, discards or deemphasizes the bad ones, and adds new ones where needed. In general, IPv6 is not compatible with IPv4, but it is compatible with all the other Internet protocols, including TCP, UDP, ICMP, IGMP, OSPF, BGP, and DNS, sometimes with small modifications being required (mostly to deal with longer addresses). The main features of IPv6 are discussed below. More information about it can be found in RFC 1883 through RFC 1887.

First and foremost, IPv6 has longer addresses than IPv4. They are 16 bytes long, which solves the problem that IPv6 was set out to solve: provide an effectively unlimited supply of Internet addresses. We will have more to say about addresses shortly.

The second major improvement of IPv6 is the simplification of the header. It contains only 7 fields (versus 13 in IPv4). This change allows routers to process packets faster and thus improve throughput. We will discuss the header shortly, too.

The third major improvement was better support for options. This change was essential with the new header because fields that previously were required are now optional. In addition, the way options are represented is different, making it simple for routers to skip over options not intended for them. This feature speeds up packet processing time.

A fourth area in which IPv6 represents a big advance is in security. IETF had its fill of newspaper stories about precocious 12-year-olds using their personal computers to break into banks and military bases all over the Internet. There was a strong feeling that something had to be done to improve security. Authentication and privacy are key features of the new IP.

Finally, more attention has been paid to type of service than in the past. IPv4 actually has an 8-bit field devoted to this matter, but with the expected growth in multimedia traffic in the future, much more is needed.

The Main IPv6 Header

The IPv6 header is shown in Fig. 5-56. The *Version* field is always 6 for IPv6 (and 4 for IPv4). During the transition period from IPv4, which will probably take a decade, routers will be able to examine this field to tell what kind of packet they have. As an aside, making this test wastes a few instructions in the critical path, so many implementations are likely to try to avoid it by using some field in the data link header to distinguish IPv4 packets from IPv6 packets. In this way, packets can be passed to the correct network layer handler directly. However, having the data link layer be aware of network packet types completely violates the design principle that each layer should not be aware of the meaning of the bits given to it from the layer above. The discussions between the “Do it right” and “Make it fast” camps will no doubt be lengthy and vigorous.

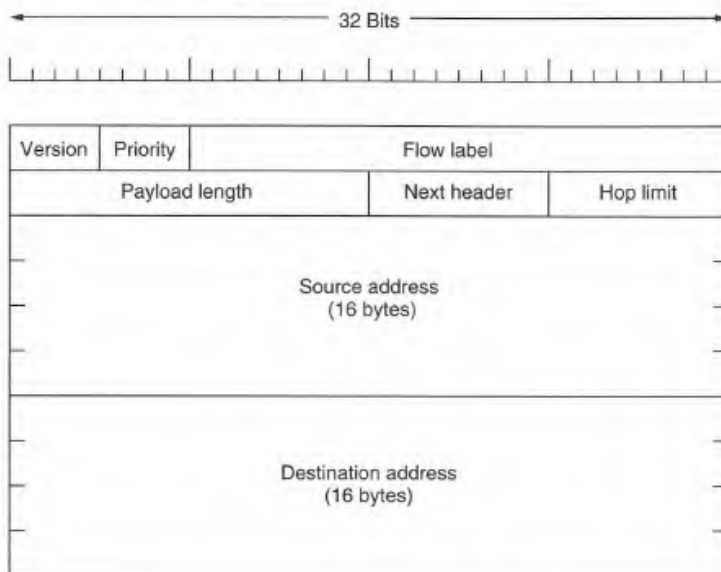


Fig. 5-56. The IPv6 fixed header (required).

The *Priority* field is used to distinguish between packets whose sources can be flow controlled and those that cannot. Values 0 through 7 are for transmissions that are capable of slowing down in the event of congestion. Values 8 through 15 are for real-time traffic whose sending rate is constant, even if all the packets are being lost. Audio and video fall into the latter category. This distinction allows routers to deal with packets better in the event of congestion. Within each group, lower-numbered packets are less important than higher-numbered ones. The IPv6 standard suggests, for example, to use 1 for news, 4 for FTP, and 6 for Telnet

connections, since delaying a news packet for a few seconds is not noticeable, but delaying a Telnet packet certainly is.

The *Flow label* field is still experimental but will be used to allow a source and destination to set up a pseudoconnection with particular properties and requirements. For example, a stream of packets from one process on a certain source host to a certain process on a certain destination host might have stringent delay requirements and thus need reserved bandwidth. The flow can be set up in advance and given an identifier. When a packet with a nonzero *Flow label* shows up, all the routers can look it up in internal tables to see what kind of special treatment it requires. In effect, flows are an attempt to have it both ways: the flexibility of a datagram subnet and the guarantees of a virtual circuit subnet.

Each flow is designated by the source address, destination address, and flow number, so many flows may be active at the same time between a given pair of IP addresses. Also, in this way, even if two flows coming from different hosts but with the same flow number pass through the same router, the router will be able to tell them apart using the source and destination addresses. It is expected that flow numbers will be chosen randomly, rather than assigned sequentially starting at 1, to make it easy for routers to hash them.

The *Payload length* field tells how many bytes follow the 40-byte header of Fig. 5-56. The name was changed from the IPv4 *Total length* field because the meaning was changed slightly: the 40 header bytes are no longer counted as part of the length as they used to be.

The *Next header* field lets the cat out of the bag. The reason the header could be simplified is that there can be additional (optional) extension headers. This field tells which of the (currently) six extension headers, if any, follows this one. If this header is the last IP header, the *Next header* field tells which transport protocol handler (e.g., TCP, UDP) to pass the packet to.

The *Hop limit* field is used to keep packets from living forever. It is, in practice, the same as the *Time to live* field in IPv4, namely, a field that is decremented on each hop. In theory, in IPv4 it was a time in seconds, but no router used it that way, so the name was changed to reflect the way it is actually used.

Next come the *Source address* and *Destination address* fields. Deering's original proposal, SIP, used 8-byte addresses, but during the review process many people felt that with 8-byte addresses IPv6 would run out of addresses within a few decades, whereas with 16-byte addresses it would never run out. Other people argued that 16 bytes was overkill, whereas still others favored using 20-byte addresses to be compatible with the OSI datagram protocol. Another faction wanted variable-sized addresses. After much discussion, it was decided that fixed-length 16-byte addresses were the best compromise.

The IPv6 address space is divided up as shown in Fig. 5-57. Addresses beginning with 80 zeros are reserved for IPv4 addresses. Two variants are supported, distinguished by the next 16 bits. These variants relate to how IPv6 packets will be tunneled over the existing IPv4 infrastructure.

Prefix (binary)	Usage	Fraction
0000 0000	Reserved (including IPv4)	1/256
0000 0001	Unassigned	1/256
0000 001	OSI NSAP addresses	1/128
0000 010	Novell NetWare IPX addresses	1/128
0000 011	Unassigned	1/128
0000 1	Unassigned	1/32
0001	Unassigned	1/16
001	Unassigned	1/8
010	Provider-based addresses	1/8
011	Unassigned	1/8
100	Geographic-based addresses	1/8
101	Unassigned	1/8
110	Unassigned	1/8
1110	Unassigned	1/16
1111 0	Unassigned	1/32
1111 10	Unassigned	1/64
1111 110	Unassigned	1/128
1111 1110 0	Unassigned	1/512
1111 1110 10	Link local use addresses	1/1024
1111 1110 11	Site local use addresses	1/1024
1111 1111	Multicast	1/256

Fig. 5-57. IPv6 addresses

The use of separate prefixes for provider-based and geographic-based addresses is a compromise between two different visions of the future of the Internet. Provider-based addresses make sense if you think that in the future there will be some number of companies providing Internet service to customers, analogous to AT&T, MCI, Sprint, British Telecom, and so on providing telephone service now. Each of these companies will be given some fraction of the address space. The first 5 bits following the 010 prefix are used to indicate which registry to look the provider up in. Currently three registries are operating, for North America, Europe, and Asia. Up to 29 new registries can be added later.

Each registry is free to divide up the remaining 15 bytes as it sees fit. It is expected that many of them will use a 3-byte provider number, giving about 16

million providers, in order to allow large companies to act as their own provider. Another possibility is to use 1 byte to indicate national providers and let them do further allocation. In this manner, additional levels of hierarchy can be introduced as needed.

The geographic model is the same as the current Internet, in which providers do not play a large role. In this way, IPv6 can handle both kinds of addresses.

The link and site local addresses have only a local significance. They can be reused at each organization without conflict. They cannot be propagated outside organizational boundaries, making them well suited to organizations that currently use firewalls to wall themselves off from the rest of the Internet.

Multicast addresses have a 4-bit flag field and a 4-bit scope field following the prefix, then a 112-bit group identifier. One of the flag bits distinguishes permanent from transient groups. The scope field allows a multicast to be limited to the current link, site, organization, or planet. These four scopes are spread out over the 16 values to allow new scopes to be added later. For example, the planetary scope is 14, so code 15 is available to allow future expansion of the Internet to other planets, solar systems, and galaxies.

In addition to supporting the standard unicast (point-to-point) and multicast addresses, IPv6 also supports a new kind of addressing: anycast. **Anycasting** is like multicasting in that the destination is a group of addresses, but instead of trying to deliver the packet to all of them, it tries to deliver it to just one, usually the nearest one. For example, when contacting a group of cooperating file servers, a client can use anycast to reach the nearest one, without having to know which one that is. Anycasting uses regular unicast addresses. It is up to the routing system to choose the lucky host that gets the packet.

A new notation has been devised for writing 16-byte addresses. They are written as eight groups of four hexadecimal digits with colons between the groups, like this:

```
8000:0000:0000:0000:0123:4567:89AB:CDEF
```

Since many addresses will have many zeros inside them, three optimizations have been authorized. First, leading zeros within a group can be omitted, so 0123 can be written as 123. Second, one or more groups of 16 zeros can be replaced by a pair of colons. Thus the above address now becomes

```
8000::123:4567:89AB:CDEF
```

Finally, IPv4 addresses can be written as a pair of colons and an old dotted decimal number, for example

```
::192.31.20.46
```

Perhaps it is unnecessary to be so explicit about it, but there are a lot of 16-byte addresses. Specifically, there are 2^{128} of them, which is approximately

3×10^{38} . If the entire earth, land and water, were covered with computers, IPv6 would allow 7×10^{23} IP addresses per square meter. Students of chemistry will notice that this number is larger than Avogadro's number. While it was not the intention to give every molecule on the surface of the earth its own IP address, we are not that far off.

In practice, the address space will not be used efficiently, just as the telephone number address space is not (the area code for Manhattan, 212, is nearly full, but that for Wyoming, 307, is nearly empty). In RFC 1715, Huitema calculated that using the allocation of telephone numbers as a guide, even in the most pessimistic scenario, there will still be well over 1000 IP addresses per square meter of the earth's surface (land and water). In any likely scenario, there will be trillions of them per square meter. In short, it seems unlikely that we will run out in the foreseeable future. It is also worth noting that only 28 percent of the address space has been allocated so far. The other 72 percent is available for future purposes not yet thought of.

It is instructive to compare the IPv4 header (Fig. 5-45) with the IPv6 header (Fig. 5-56) to see what has been left out in IPv6. The *IHL* field is gone because the IPv6 header has a fixed length. The *Protocol* field was taken out because the *Next header* field tells what follows the last IP header (e.g., a UDP or TCP segment).

All the fields relating to fragmentation were removed because IPv6 takes a different approach to fragmentation. To start with, all IPv6 conformant hosts and routers must support packets of 576 bytes. This rule makes fragmentation less likely to occur in the first place. In addition, when a host sends an IPv6 packet that is too large, instead of fragmenting it, the router that is unable to forward it sends back an error message. This message tells the host to break up all future packets to that destination. Having the host send packets that are the right size in the first place is ultimately much more efficient than having the routers fragment them on the fly.

Finally, the *Checksum* field is gone because calculating it greatly reduces performance. With the reliable networks now used, combined with the fact that the data link layer and transport layers normally have their own checksums, the value of yet another checksum was not worth the performance price it extracted. Removing all these features has resulted in a lean and mean network layer protocol. Thus the goal of IPv6—a fast, yet flexible, protocol with plenty of address space—has been met by this design.

Extension Headers

Nevertheless, some of the missing fields are occasionally still needed so IPv6 has introduced the concept of an (optional) **extension header**. These headers can be supplied to provide extra information, but encoded in an efficient way. Six

kinds of extension headers are defined at present, as listed in Fig. 5-58. Each one is optional, but if more than one is present, they must appear directly after the fixed header, and preferably in the order listed.

Extension header	Description
Hop-by-hop options	Miscellaneous information for routers
Routing	Full or partial route to follow
Fragmentation	Management of datagram fragments
Authentication	Verification of the sender's identity
Encrypted security payload	Information about the encrypted contents
Destination options	Additional information for the destination

Fig. 5-58. IPv6 extension headers.

Some of the headers have a fixed format; others contain a variable number of variable-length fields. For these, each item is encoded as a (Type, Length, Value) tuple. The *Type* is a 1-byte field telling which option this is. The *Type* values have been chosen so that the first 2 bits tell routers that do not know how to process the option what to do. The choices are: skip the option, discard the packet, discard the packet and send back an ICMP packet, and the same as the previous one, except do not send ICMP packets for multicast addresses (to prevent one bad multicast packet from generating millions of ICMP reports).

The *Length* is also a 1-byte field. It tells how long the value is (0 to 255 bytes). The *Value* is any information required, up to 255 bytes.

The hop-by-hop header is used for information that all routers along the path must examine. So far, one option has been defined: support of datagrams exceeding 64K. The format of this header is shown in Fig. 5-59.

Next header	0	194	0
Jumbo payload length			

Fig. 5-59. The hop-by-hop extension header for large datagrams (jumbograms).

As with all extension headers, this one starts out with a byte telling what kind of header comes next. This byte is followed by one telling how long the hop-by-hop header is in bytes, excluding the first 8 bytes, which are mandatory. The next 2 bytes indicate that this option defines the datagram size (code 194) as a 4-byte number. The last 4 bytes give the size of the datagram. Sizes less than 65,536 are not permitted and will result in the first router discarding the packet and sending back an ICMP error message. Datagrams using this header extension are called

jumbograms. The use of jumbograms is important for supercomputer applications that must transfer gigabytes of data efficiently across the Internet.

The routing header lists one or more routers that must be visited on the way to the destination. Both strict routing (the full path is supplied) and loose routing (only selected routers are supplied) are available, but they are combined. The format of the routing header is shown in Fig. 5-60.

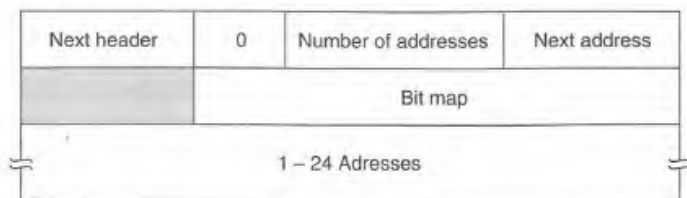


Fig. 5-60. The extension header for routing.

The first 4 bytes of the routing extension header contain four 1-byte integers: the next header type, the routing type (currently 0), the number of addresses present in this header (1 to 24), and the index of the next address to visit. The latter field starts at 0 and is incremented as each address is visited.

Then comes a reserved byte followed by a bit map with bits for each of the 24 potential IPv6 addresses following it. These bits tell whether each address must be visited directly after the one before it (strict source routing), or whether other routers may come in between (loose source routing).

The fragment header deals with fragmentation similarly to the way IPv4 does. The header holds the datagram identifier, fragment number, and a bit telling whether more fragments will follow. In IPv6, unlike in IPv4, only the source host can fragment a packet. Routers along the way may not do this. Although this change is a major philosophical break with the past, it simplifies the routers' work and makes routing go faster. As mentioned above, if a router is confronted with a packet that is too big, it discards the packet and sends an ICMP packet back to the source. This information allows the source host to fragment the packet into smaller pieces using this header and try again.

The authentication header provides a mechanism by which the receiver of a packet can be sure of who sent it. With IPv4, no such guarantee is present. The encrypted security payload makes it possible to encrypt the contents of a packet so that only the intended recipient can read it. These headers use cryptographic techniques to accomplish their missions. We will give brief descriptions below, but readers not already familiar with modern cryptography may not understand the full description now. They should come back after having read Chap. 7 (in particular, Sec. 7.1), which treats cryptographic protocols.

When a sender and receiver wish to communicate securely, they must first agree on one or more secret keys that only they know. How they do this is outside

the scope of IPv6. Each of these keys is assigned a unique 32-bit key number. The key numbers are global, so that if Alice is using key 4 to talk to Bob, she cannot also have a key 4 to talk to Carol. Associated with each key number are other parameters, such as key lifetime, and so on.

To send an authenticated message, the sender first constructs a packet consisting of all the IP headers and the payload and then replaces the fields that change underway (e.g., *Hop limit*) with zeros. The packet is then padded out with zeros to a multiple of 16 bytes. Similarly, the secret key to be used is also padded out with zeros to a multiple of 16 bytes. Now a cryptographic checksum is computed on the concatenation of the padded secret key, the padded packet, and the padded secret key again. Users may define their own cryptographic checksum algorithms, but cryptographically unsophisticated users should use the default algorithm, MD5.

Now we come to the role of the authentication header. Basically, it contains three parts. The first part consists of 4 bytes holding the next header number, the length of the authentication header, and 16 zero bits. Then comes the 32-bit key number. Finally, the MD5 (or other) checksum is included.

The receiver then uses the key number to find the secret key. The padded version of it is then prepended and appended to the padded payload, the variable header fields are zeroed out, and the checksum computed. If it agrees with the checksum included in the authentication header, the receiver can be sure that the packet came from the sender with whom the secret key is shared and also be sure that the packet was not tampered with underway. The properties of MD5 make it computationally infeasible for an intruder to forge the sender's identity or modify the packet in a way that escapes detection.

It is important to note that the payload of an authenticated packet is sent unencrypted. Any router along the way can read what it says. For many applications, secrecy is not really important, just authentication. For example, if a user instructs his bank to pay his telephone bill, there is probably no real need for secrecy, but there is a very real need for the bank to be absolutely sure it knows who sent the packet containing the payment order.

For packets that must be sent secretly, the encrypted security payload extension header is used. It starts out with a 32-bit key number, followed by the encrypted payload. The encryption algorithm is up to the sender and receiver, but DES in cipher block chaining mode is the default. When DES-CBC is used, the payload field starts out with the initialization vector (a multiple of 4 bytes), then the payload, then padding out to multiple of 8 bytes. If both encryption and authentication are desired, both headers are needed.

The destination options header is intended for fields that need only be interpreted at the destination host. In the initial version of IPv6, the only options defined are null options for padding this header out to a multiple of 8 bytes, so initially it will not be used. It was included to make sure that new routing and host software can handle it, in case someone thinks of a destination option some day.

Controversies

Given the open design process and the strongly-held opinions of many of the people involved, it should come as no surprise that many choices made for IPv6 were highly controversial. We will summarize a few of these below. For all the gory details, see (Huitema, 1996).

We have already mentioned the argument about the address length. The result was a compromise: 16-byte fixed-length addresses.

Another fight developed over the length of the *Hop limit* field. One camp felt strongly that limiting the maximum number of hops to 255 (implicit in using an 8-bit field) was a gross mistake. After all, paths of 32 hops are common now, and 10 years from now much longer paths may be common. These people argued that using a huge address size was farsighted but using a tiny hop count was shortsighted. In their view, the greatest sin a computer scientist can commit is to provide too few bits somewhere.

The response was that arguments could be made to increase every field, leading to a bloated header. Also, the function of the *Hop limit* field is to keep packets from wandering around for a long time and 65,535 hops is far too long. Finally, as the Internet grows, more and more long-distance links will be built, making it possible to get from any country to any other country in half a dozen hops at most. If it takes more than 125 hops to get from the source and destination to their respective international gateways, something is wrong with the national backbones. The 8-bitters won this one.

Another hot potato was the maximum packet size. The supercomputer community wanted packets in excess of 64 KB. When a supercomputer gets started transferring, it really means business and does not want to be interrupted every 64 KB. The argument against large packets is that if a 1-MB packet hits a 1.5-Mbps T1 line, that packet will tie the line up for over 5 seconds, producing a very noticeable delay for interactive users sharing the line. A compromise was reached here: normal packets are limited to 64 KB, but the hop-by-hop extension header can be used to permit jumbograms.

A third hot topic was removing the IPv4 checksum. Some people likened this move to removing the brakes from a car. Doing so makes the car lighter so it can go faster, but if an unexpected event happens, you have a problem.

The argument against checksums was that any application that really cares about data integrity has to have a transport layer checksum anyway, so having another one in IP (in addition to the data link layer checksum) is overkill. Furthermore, experience showed that computing the IP checksum was a major expense in IPv4. The antichecksum camp won this one, and IPv6 does not have a checksum.

Mobile hosts were also a point of contention. If a portable computer flies halfway around the world, can it continue operating at the destination with the same IPv6 address, or does it have to use a scheme with home agents and foreign

agents? Mobile hosts also introduce asymmetries into the routing system. It may well be the case that a small mobile computer can easily hear the powerful signal put out by a large stationary router, but the stationary router cannot hear the feeble signal put out by the mobile host. Consequently, some people wanted to build explicit support for mobile hosts into IPv6. That effort failed when no consensus could be found for any specific proposal.

Probably the biggest battle was about security. Everyone agreed it was needed. The war was about where and how. First where. The argument for putting it in the network layer is that it then becomes a standard service that all applications can use without any advance planning. The argument against it is that really secure applications generally want nothing less than end-to-end encryption, where the source application does the encryption and the destination application undoes it. With anything less, the user is at the mercy of potentially buggy network layer implementations over which he has no control. The response to this argument is that these applications can just refrain from using the IP security features and do the job themselves. The rejoinder to that is that the people who do not trust the network to do it right, do not want to have to pay the price of slow, bulky IP implementations that have this capability, even if it is disabled.

Another aspect of where to put security relates to the fact that many (but not all) countries have stringent export laws concerning cryptography. Some, notably France and Iraq, also greatly restrict its use domestically, so that people cannot have secrets from the police. As a result, any IP implementation that used a cryptographic system strong enough to be of much value could not be exported from the United States (and many other countries) to customers worldwide. Having to maintain two sets of software, one for domestic use and one for export, is something most computer vendors vigorously oppose.

One potential solution is for all vendors to move their cryptography shops to a country that does not regulate cryptography, such as Finland or Switzerland. Strong cryptographic software could be designed and manufactured there and then shipped legally to all countries except France and Iraq. The problem with this approach is that designing part of the router software in one country and part in another can lead to integration problems.

The final controversy concerning security relates to the choice of the default algorithms that all implementations must support. While MD5 was thought to be relatively secure, recent advances in cryptography may weaken it. No serious cryptographer believes that DES is secure against attacks by major governments, but it is probably good enough to foil even the most precocious 12-year-olds for the time being. The compromise was thus to mandate security in IPv6, use a state-of-the-art checksum algorithm for good authentication and a weakish algorithm for secrecy but give users the option of replacing these algorithms with their own.

One point on which there was no controversy is that no one expects the IPv4 Internet to be turned off on a Sunday morning and come back up as an IPv6

Internet Monday morning. Instead, isolated “islands” of IPv6 will be converted, initially communicating via tunnels. As the IPv6 islands grow, they will merge into bigger islands. Eventually, all the islands will merge, and the Internet will be fully converted. Given the massive investment in IPv4 routers currently deployed, the conversion process will probably take a decade. For this reason, an enormous amount of effort has gone into making sure that this transition will be as painless as possible.

5.6. THE NETWORK LAYER IN ATM NETWORKS

The layers of the ATM model (see Fig. 1-30) do not map onto the OSI layers especially well, which leads to ambiguities. The OSI data link layer deals with framing and transfer protocols between two machines on the same physical wire (or fiber). Data link layer protocols are single-hop protocols. They do not deal with end-to-end connections because switching and routing do not occur in the data link layer. About this there is no doubt.

The lowest layer that goes from source to destination, and thus involves routing and switching (i.e., is multihop), is the network layer. The ATM layer deals with moving cells from source to destination and definitely involves routing algorithms and protocols within the ATM switches. It also deals with global addressing. Thus functionally, the ATM layer performs the work expected of the network layer. The ATM layer is not guaranteed to be 100 percent reliable, but that is not required for a network layer protocol.

Also, the ATM layer resembles layer 3 of X.25, which everyone agrees is a network layer protocol. Depending on bit settings, the X.25 network layer protocol may or may not be reliable, but most implementations treat it as unreliable. Because the ATM layer has the functionality expected of the network layer, does not have the functionality expected of the data link layer, and is quite similar to existing network layer protocols, we will discuss the ATM layer in this chapter.

Confusion arises because many people in the ATM community regard the ATM layer as a data link layer, or when doing LAN emulation, even a physical layer. Many people in the Internet community also regard it as a data link layer because they want to put IP on top of it, and making the ATM layer a data link layer fits well with this idea. (Although following through with this line of reasoning, to the Internet community, *all* networks operate at the data link layer, no matter what their physical characteristics are.)

The only problem is that the ATM layer does not have the characteristics of a data link layer protocol: a single-hop protocol used by machines at the opposite ends of a wire, such as protocols 1 through 6 in Chap. 3. It has the characteristics of a network layer protocol: end-to-end virtual circuits, switching, and routing.

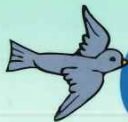
The author is reminded of an old riddle:

Q: How many legs would a mule have if you called the tail a leg?

A: Four. *Calling* the tail a leg does not *make* it a leg.

COMPUTER NETWORKING

THIRD EDITION



COMPUTER NETWORKS

ANDREW S. TANENBAUM

Computer Networks is the ideal introduction to today's and tomorrow's networks. This classic best-seller has been totally rewritten to reflect the networks of the late 1990s and beyond.

Author, educator, and researcher Andrew S. Tanenbaum, winner of the ACM Karl V. Karlstrom Outstanding Educator Award, carefully explains how networks work inside, from the hardware technology up through the most popular network applications. The book takes a structured approach to networking, starting at the bottom (the physical layer) and gradually working up to the top (the application layer). The topics covered include:

- Physical layer (e.g., copper, fiber, radio, and satellite communication)
- Data link layer (e.g., protocol principles, HDLC, SLIP, and PPP)
- MAC Sublayer (e.g., IEEE 802 LANs, bridges, new high-speed LANs)
- Network layer (e.g., routing, congestion control, internetworking, IPv6)
- Transport layer (e.g., transport protocol principles, TCP, network performance)
- Application layer (e.g., cryptography, email, news, the Web, Java, multimedia)

In each chapter, the necessary principles are described in detail, followed by extensive examples taken from the Internet, ATM networks, and wireless networks.

Other bestselling titles by Andrew S. Tanenbaum:

Operating Systems: Design and Implementation, 2nd edition

Modern Operating Systems

Distributed Operating Systems

Structured Computer Organization, 3rd edition

PRENTICE HALL
Upper Saddle River, NJ 07458

ISBN 0-13-349945-6



9 780133 499452

FORTINET EX1009 Page 120



TANENBAUM

COMPUTER NETWORKS

THIRD EDITION

