TCP/IP Illustrated, Volume 1

The Protocols

W. Richard Stevens

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

TCP/IP Illustrated, Volume 1

The Protocols

W. Richard Stevens

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

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

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

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

Text printed on recycled and acid-free paper

ISBN 0-201-63346-9
7 8 9 10 11 12 13 14 15-MA-99989796
Seventh printing, March 1996

TCP/IP Illustrated, Volume 1

Addison-Wesley Professional Computing Series

Brian W. Kernighan, Consulting Editor

Ken Arnold/John Peyton, *A C User's Guide to ANSI C*
Tom Cargill, *C++ Programming Style*
William R. Cheswick/Steven M. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*
David A. Curry, *UNIX® System Security: A Guide for Users and System Administrators*
Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*
John Lakos, *Large-Scale C++ Software Design*
Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*
Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*
Robert B. Murray, *C++ Strategies and Tactics*
David R. Musser/Atul Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*
John K. Ousterhout, *Tcl and the Tk Toolkit*
Craig Partridge, *Gigabit Networking*
J. Stephen Pendergrast Jr., *Desktop KornShell Graphical Programming*
Radia Perlman, *Interconnections: Bridges and Routers*
David M. Piscitello/A. Lyman Chapin, *Open Systems Networking: TCP/IP and OSI*
Stephen A. Rago, *UNIX® System V Network Programming*
Curt Schimmel, *UNIX® Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*
W. Richard Stevens, *Advanced Programming in the UNIX® Environment*
W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*
W. Richard Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*
Gary R. Wright/W. Richard Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*

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

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

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

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

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

— Steven Baker, Networking Columnist, *Unix Review*

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

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

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

— Walter Zintz, *UnixWorld*

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

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

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

— Elizabeth Zinkann, *Sys Admin*

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

— Scott Bradner, Consultant, Harvard University OIT/NSD

Contents

Preface

Introduction

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

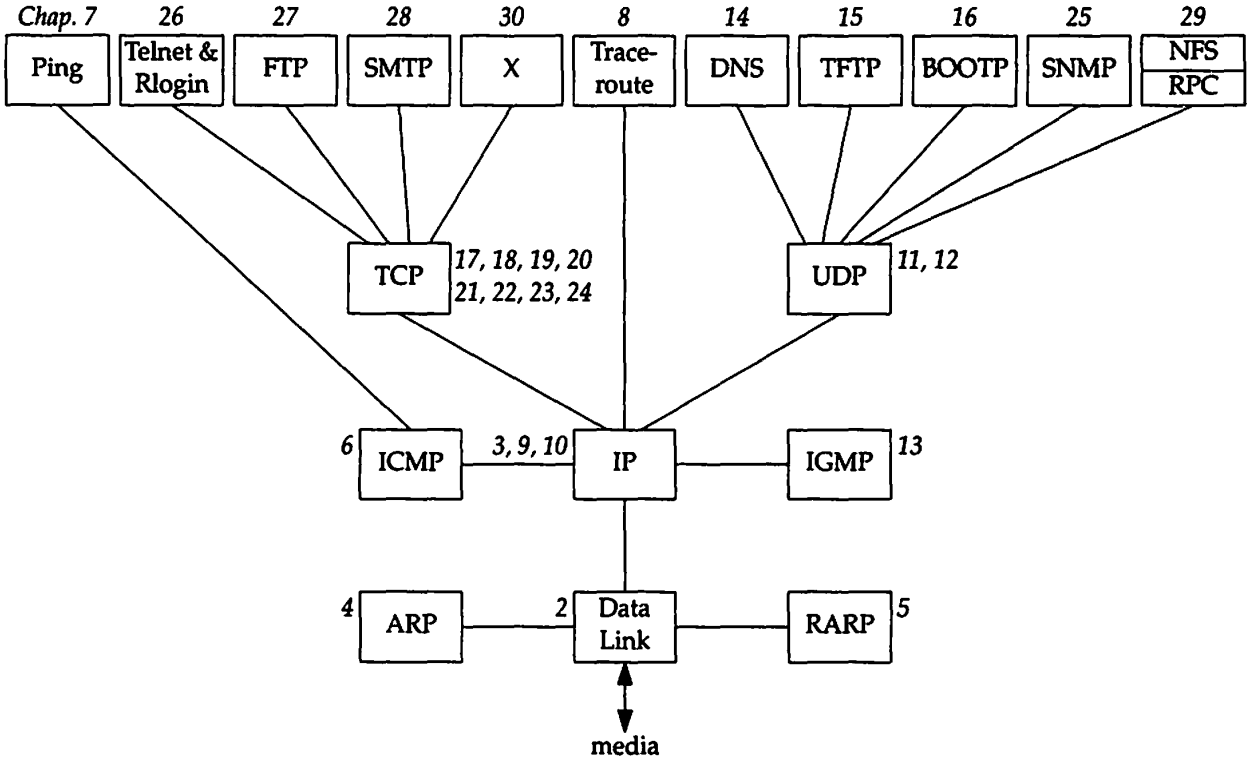
When networking protocols were being developed in the 1960s through the 1980s, expensive, dedicated hardware was required to see the packets going "across the wire." Extreme familiarity with the protocols was also required to comprehend the packets displayed by the hardware. Functionality of the hardware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

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

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

Organization of the Book

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



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

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

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

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

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

Readers

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

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

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

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

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

Systems Used for Testing

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

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

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

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

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

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

Typographical Conventions

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

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

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

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

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

Acknowledgments

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

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

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

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

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

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

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

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

Tucson, Arizona
October 1993

W. Richard Stevens
rstevens@noao.edu
http://www.noao.edu/~rstevens

1

Introduction

1.1 Introduction

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

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

1.2 Layering

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

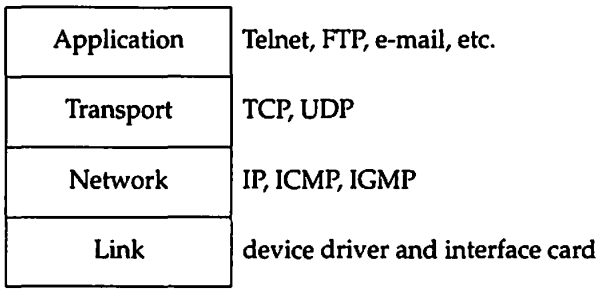


Figure 1.1 The four layers of the TCP/IP protocol suite.

Each layer has a different responsibility.

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

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

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

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

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

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

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

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

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

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

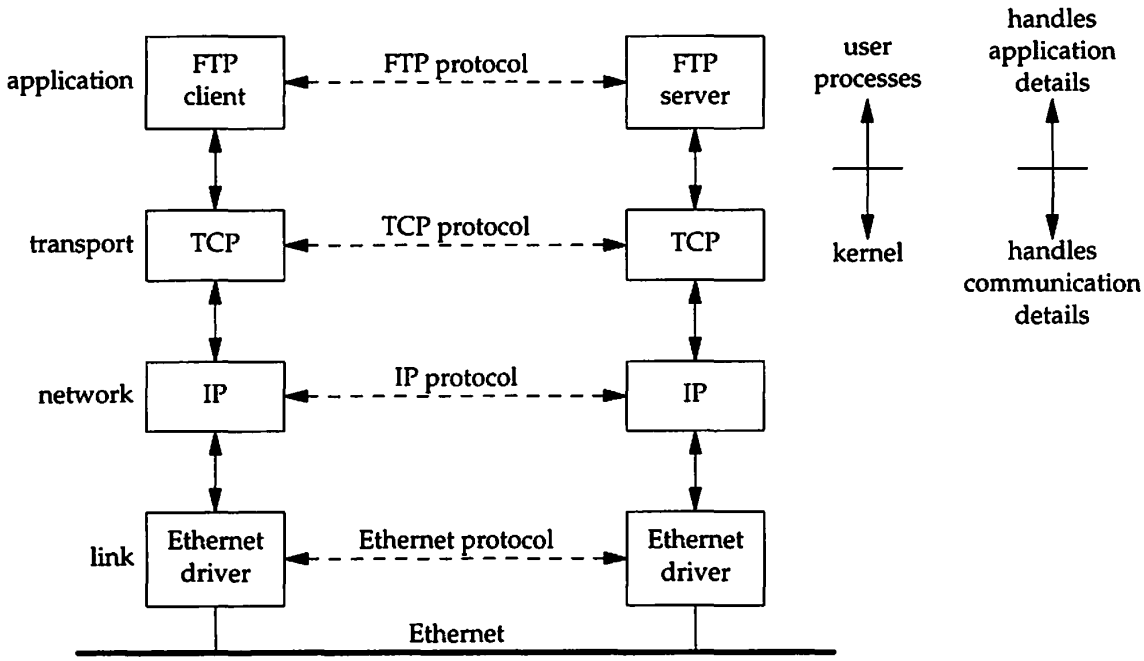


Figure 1.2 Two hosts on a LAN running FTP.

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

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

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

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

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

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

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

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

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

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

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

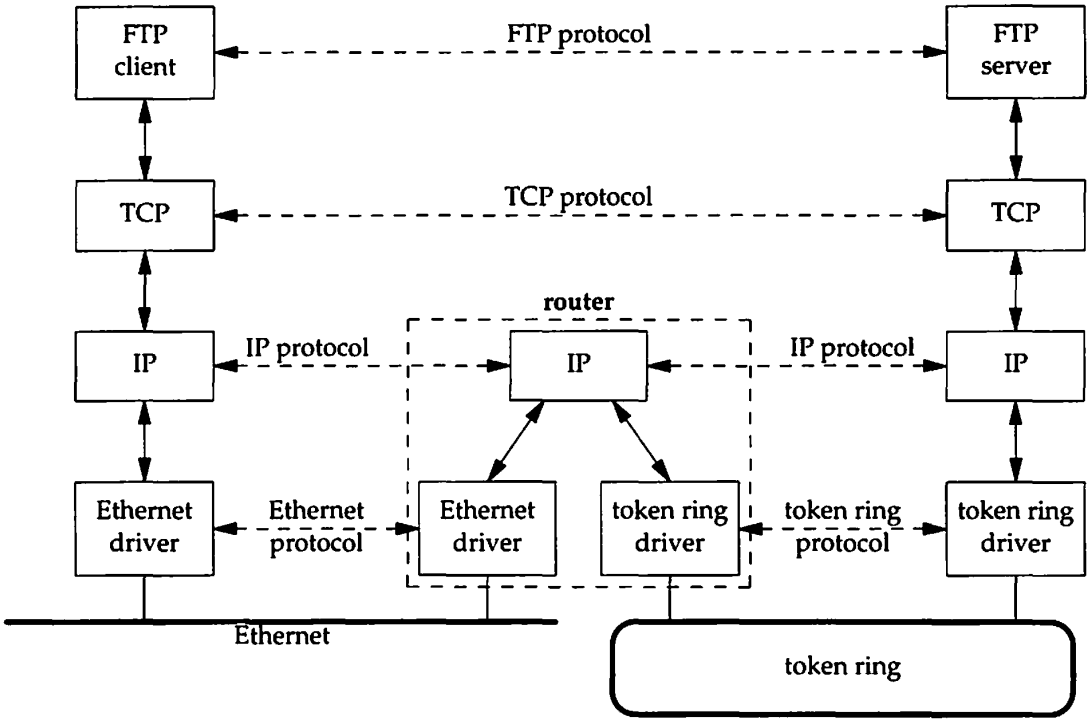


Figure 1.3 Two networks connected with a router.

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

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

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

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

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

1.3 TCP/IP Layering

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

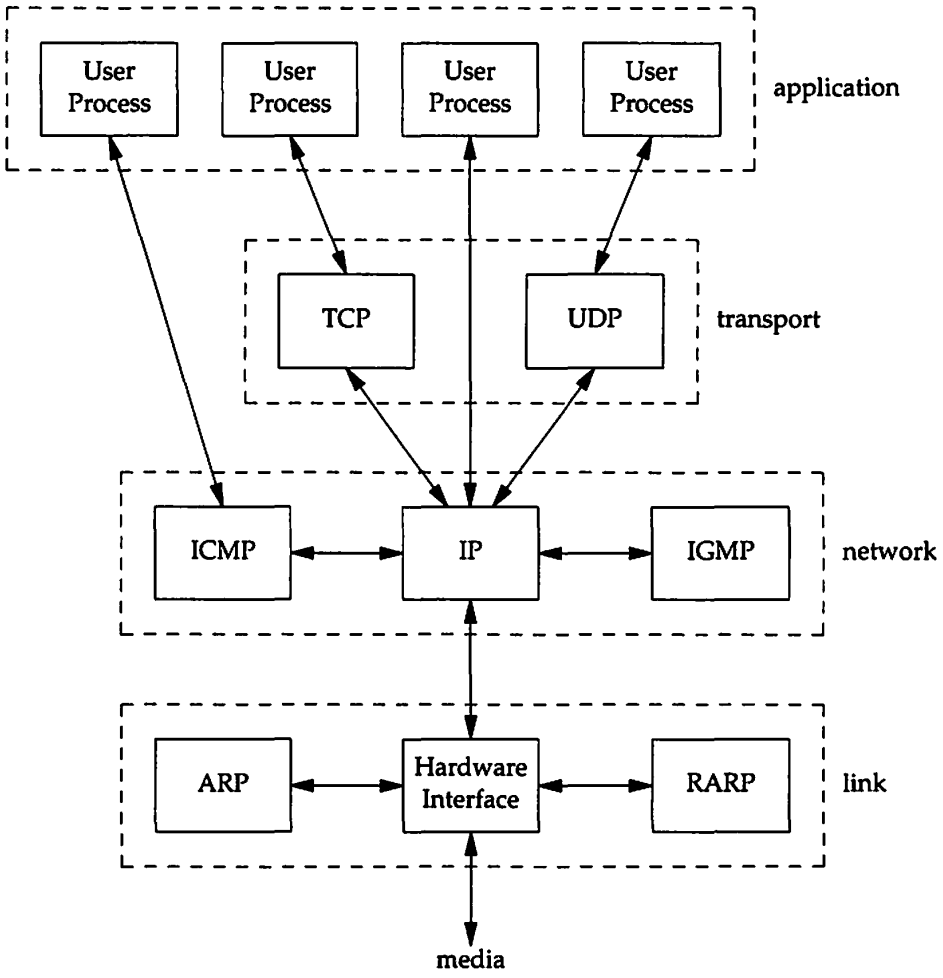


Figure 1.4 Various protocols at the different layers in the TCP/IP protocol suite.

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

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

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

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

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

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

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

1.4 Internet Addresses

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

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

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

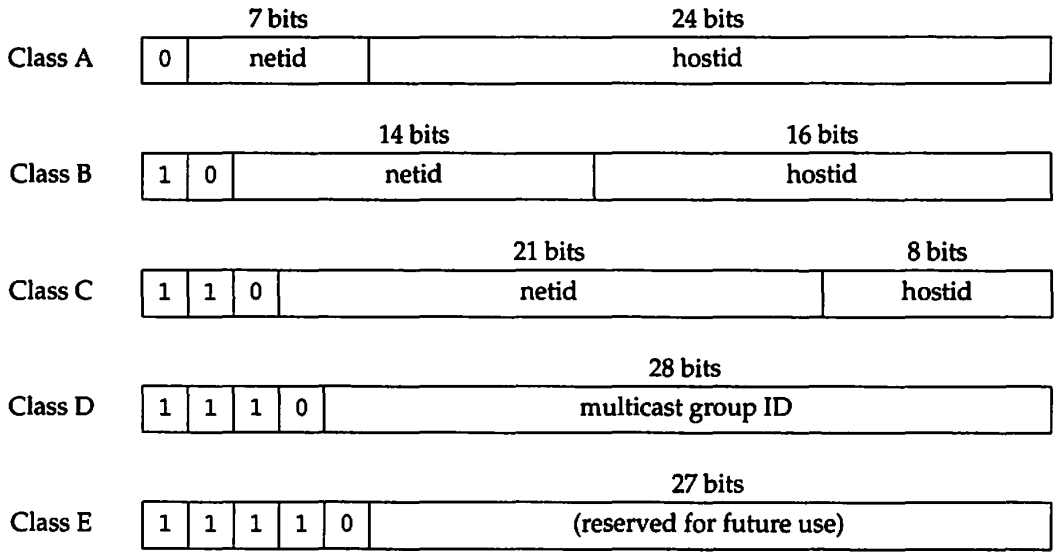


Figure 1.5 The five different classes of Internet addresses.

Class	Range
A	0.0.0.0 to 127.255.255.255
B	128.0.0.0 to 191.255.255.255
C	192.0.0.0 to 223.255.255.255
D	224.0.0.0 to 239.255.255.255
E	240.0.0.0 to 247.255.255.255

Figure 1.6 Ranges for different classes of IP addresses.

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

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

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

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

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

1.5 The Domain Name System

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

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

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

1.6 Encapsulation

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

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

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

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

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

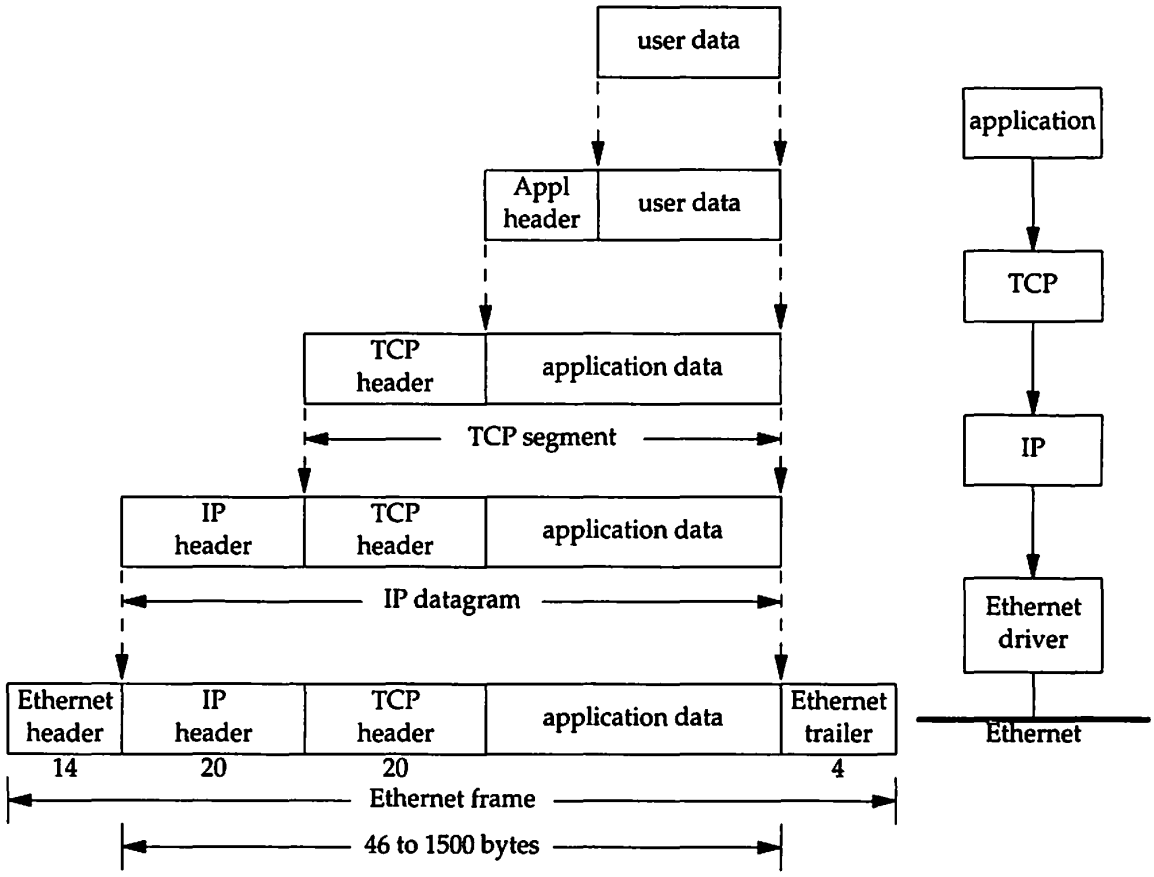


Figure 1.7 Encapsulation of data as it goes down the protocol stack.

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

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

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

1.7 Demultiplexing

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

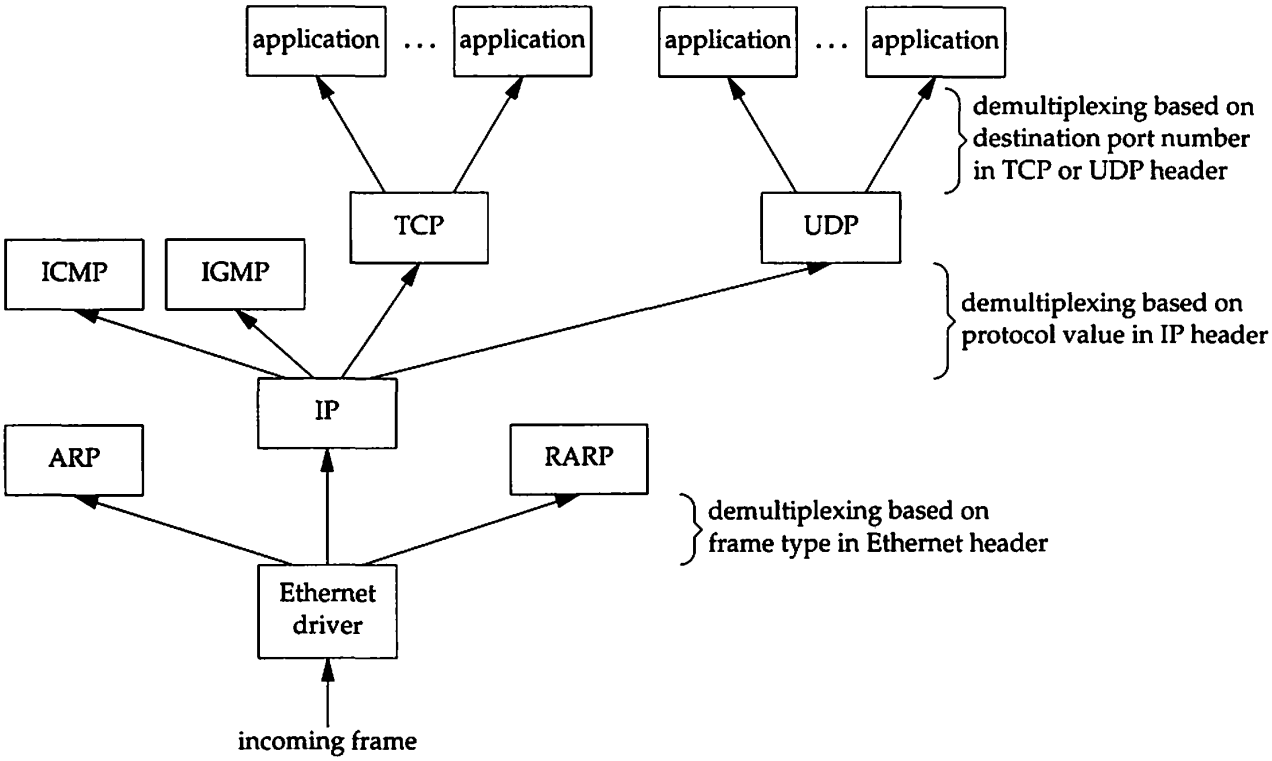


Figure 1.8 The demultiplexing of a received Ethernet frame.

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

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

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

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

1.8 Client–Server Model

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

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

I1. Wait for a client request to arrive.

I2. Process the client request.

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

I4. Go back to step I1.

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

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

C1. Wait for a client request to arrive.

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

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

C3. Go back to step C1.

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

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

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

1.9 Port Numbers

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

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

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

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

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

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

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

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

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

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

Reserved Ports

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

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

1.10 Standardization Process

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

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

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

1.11 RFCs

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

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

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

help: ways_to_get_rfcs
```

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

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

There are a few important RFCs.

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

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

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

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

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

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

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

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

1.12 Standard, Simple Services

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

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

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

Figure 1.9 Standard, simple services provided by most implementations.

1.13 The Internet

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

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

1.14 Implementations

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

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

Throughout the text we'll use the term *Berkeley-derived implementation* to refer to vendor implementations such as SunOS 4.x, SVR4, and AIX 3.2 that were originally developed from the Berkeley sources. These implementations have much in common, often including the same bugs!

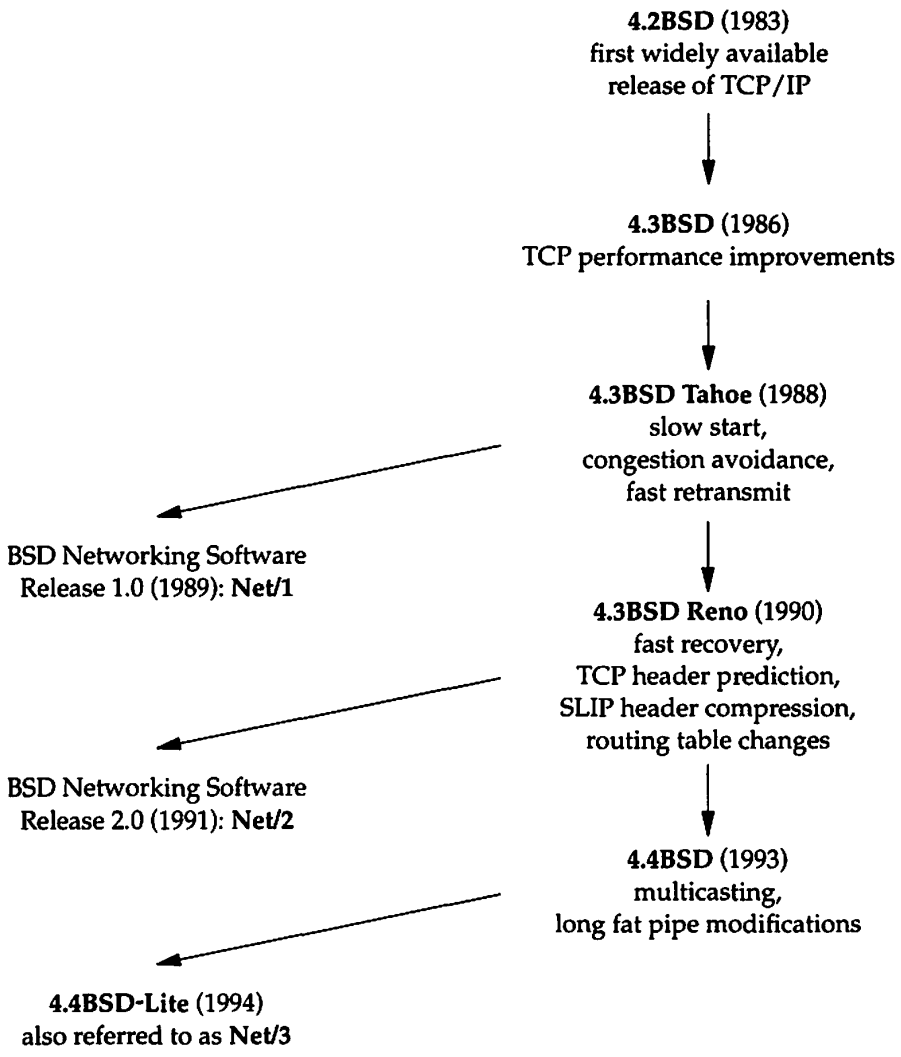


Figure 1.10 Various BSD releases with important TCP/IP features.

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

1.15 Application Programming Interfaces

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

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

1.16 Test Network

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

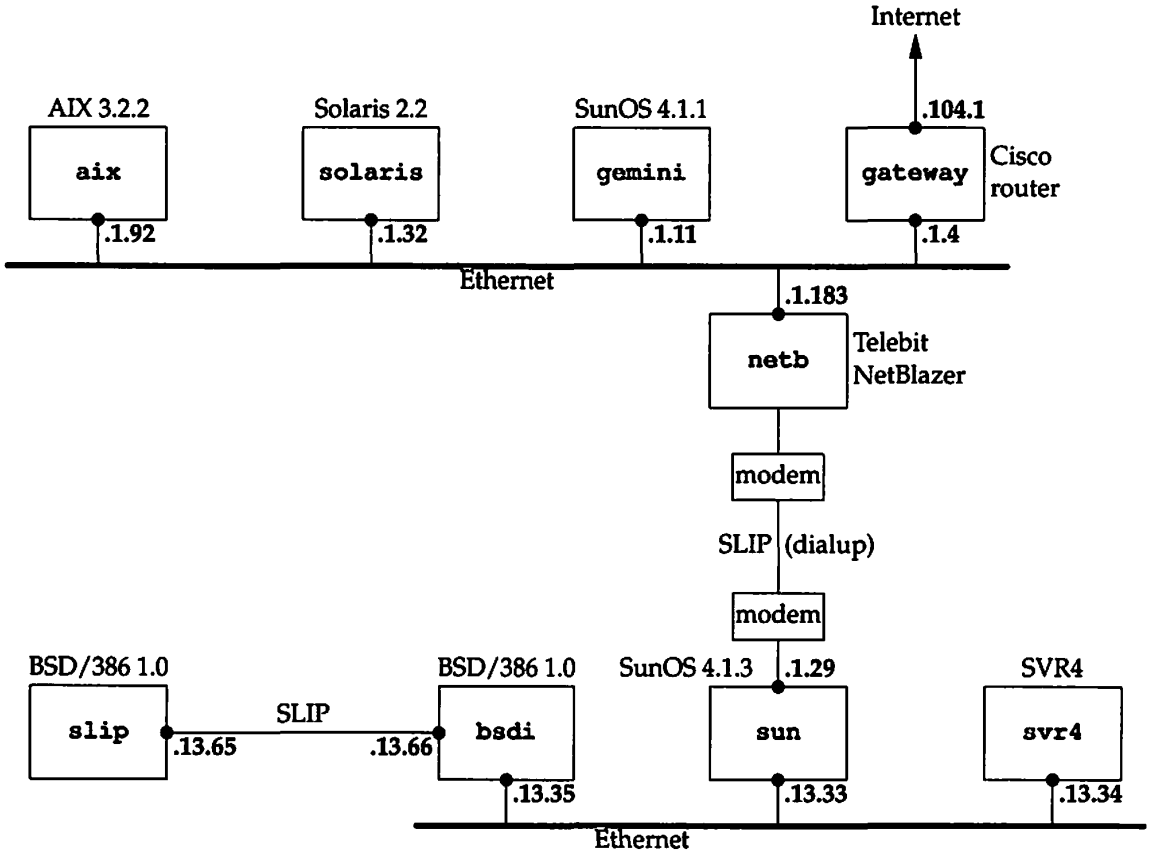


Figure 1.11 Test network used for all the examples in the text. All IP addresses begin with 140.252.

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

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

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

1.17 Summary

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

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

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

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

Exercises

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

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

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

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

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

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

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

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

20

TCP Bulk Data Flow

20.1 Introduction

In Chapter 15 we saw that TFTP uses a stop-and-wait protocol. The sender of a data block required an acknowledgment for that block before the next block was sent. In this chapter we'll see that TCP uses a different form of flow control called a *sliding window* protocol. It allows the sender to transmit multiple packets before it stops and waits for an acknowledgment. This leads to faster data transfer, since the sender doesn't have to stop and wait for an acknowledgment each time a packet is sent.

We also look at TCP's PUSH flag, something we've seen in many of the previous examples. We also look at slow start, the technique used by TCP for getting the flow of data established on a connection, and then we examine bulk data throughput.

20.2 Normal Data Flow

Let's start with a one-way transfer of 8192 bytes from the host `svr4` to the host `bsdi`. We run our `sock` program on `bsdi` as the server:

```
bsdi % sock -i -s 7777
```

The `-i` and `-s` flags tell the program to run as a "sink" server (read from the network and discard the data), and the server's port number is specified as 7777. The corresponding client is then run as:

```
svr4 % sock -i -n8 bsdi 7777
```

This causes the client to perform eight 1024-byte writes to the network. Figure 20.1 shows the time line for this exchange. We have left the first three segments in the output to show the MSS values for each end.

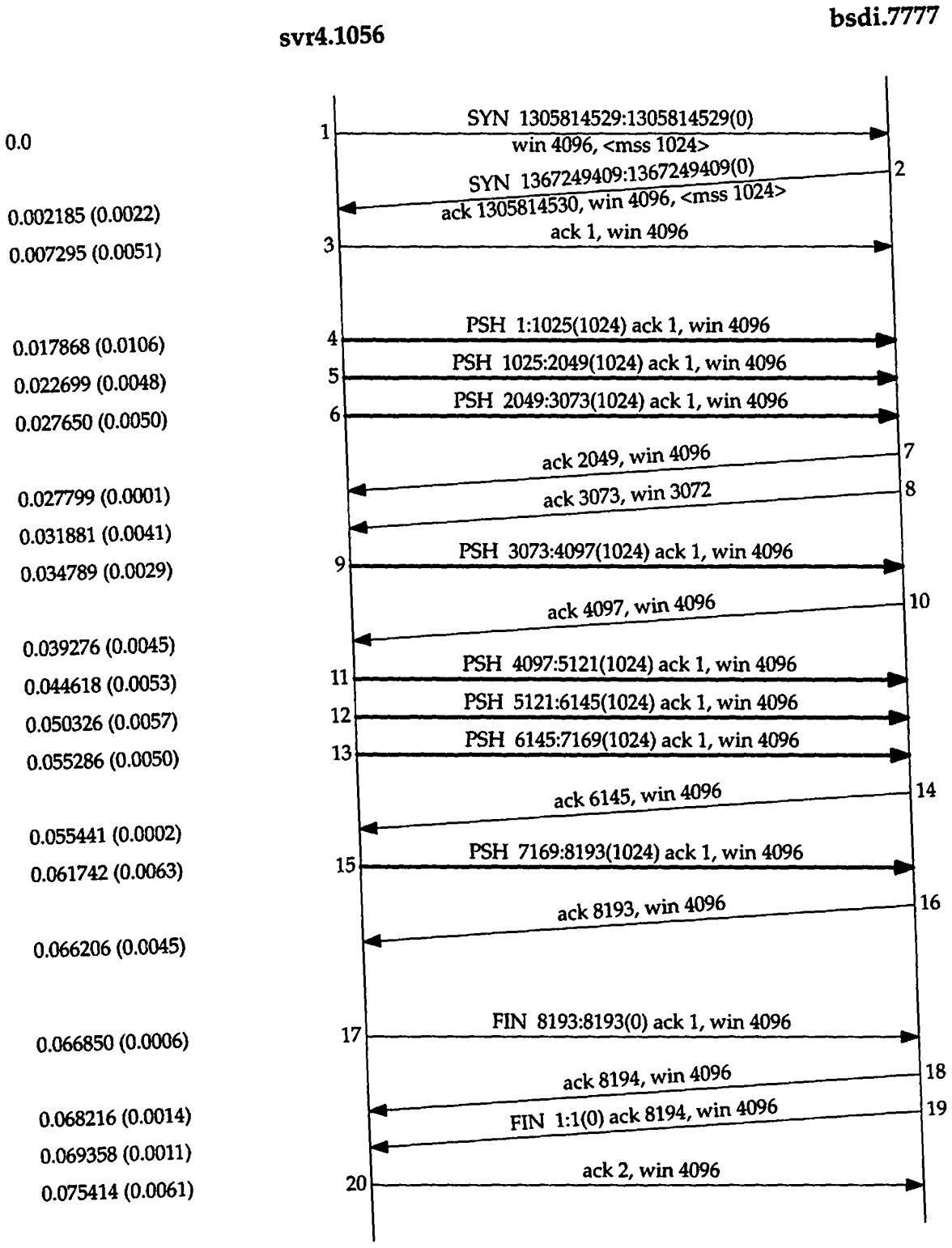


Figure 20.1 Transfer of 8192 bytes from `svr4` to `bsdi`.

The sender transmits three data segments (4–6) first. The next segment (7) acknowledges the first two data segments only. We know this because the acknowledged sequence number is 2049, not 3073.

Segment 7 specifies an ACK of 2049 and not 3073 for the following reason. When a packet arrives it is initially processed by the device driver's interrupt service routine and then placed onto IP's input queue. The three segments 4, 5, and 6 arrive one after the other and are placed onto IP's input queue in the received order. IP will pass them to TCP in the same order. When TCP processes segment 4, the connection is marked to generate a delayed ACK. TCP processes the next segment (5) and since TCP now has two outstanding segments to ACK, the ACK of 2049 is generated (segment 7), and the delayed ACK flag for this connection is turned off. TCP processes the next input segment (6) and the connection is again marked for a delayed ACK. Before segment 9 arrives, however, it appears the delayed ACK timer goes off, and the ACK of 3073 (segment 8) is generated. Segment 8 advertises a window of 3072 bytes, implying that there are still 1024 bytes of data in the TCP receive buffer that the application has not read.

Segments 11–16 show the "ACK every other segment" strategy that is common. Segments 11, 12, and 13 arrive and are placed on IP's input queue. When segment 11 is processed by TCP the connection is marked for a delayed ACK. When segment 12 is processed, an ACK is generated (segment 14) for segments 11 and 12, and the delayed ACK flag for this connection is turned off. Segment 13 causes the connection to be marked again for a delayed ACK but before the timer goes off, segment 15 is processed, causing the ACK (segment 16) to be sent immediately.

It is important to notice that the ACK in segments 7, 14, and 16 acknowledge two received segments. With TCP's sliding-window protocol the receiver does not have to acknowledge every received packet. With TCP, the ACKs are cumulative—they acknowledge that the receiver has correctly received all bytes up through the acknowledged sequence number minus one. In this example three of the ACKs acknowledge 2048 bytes of data and two acknowledge 1024 bytes of data. (This ignores the ACKs in the connection establishment and termination.)

What we are watching with `tcpdump` are the dynamics of TCP in action. The ordering of the packets that we see on the wire depends on many factors, most of which we have no control over: the sending TCP implementation, the receiving TCP implementation, the reading of data by the receiving process (which depends on the process scheduling by the operating system), and the dynamics of the network (i.e., Ethernet collisions and backoffs). There is no single correct way for two TCPs to exchange a given amount of data.

To show how things can change, Figure 20.2 shows another time line for the same exchange of data between the same two hosts, captured a few minutes after the one in Figure 20.1.

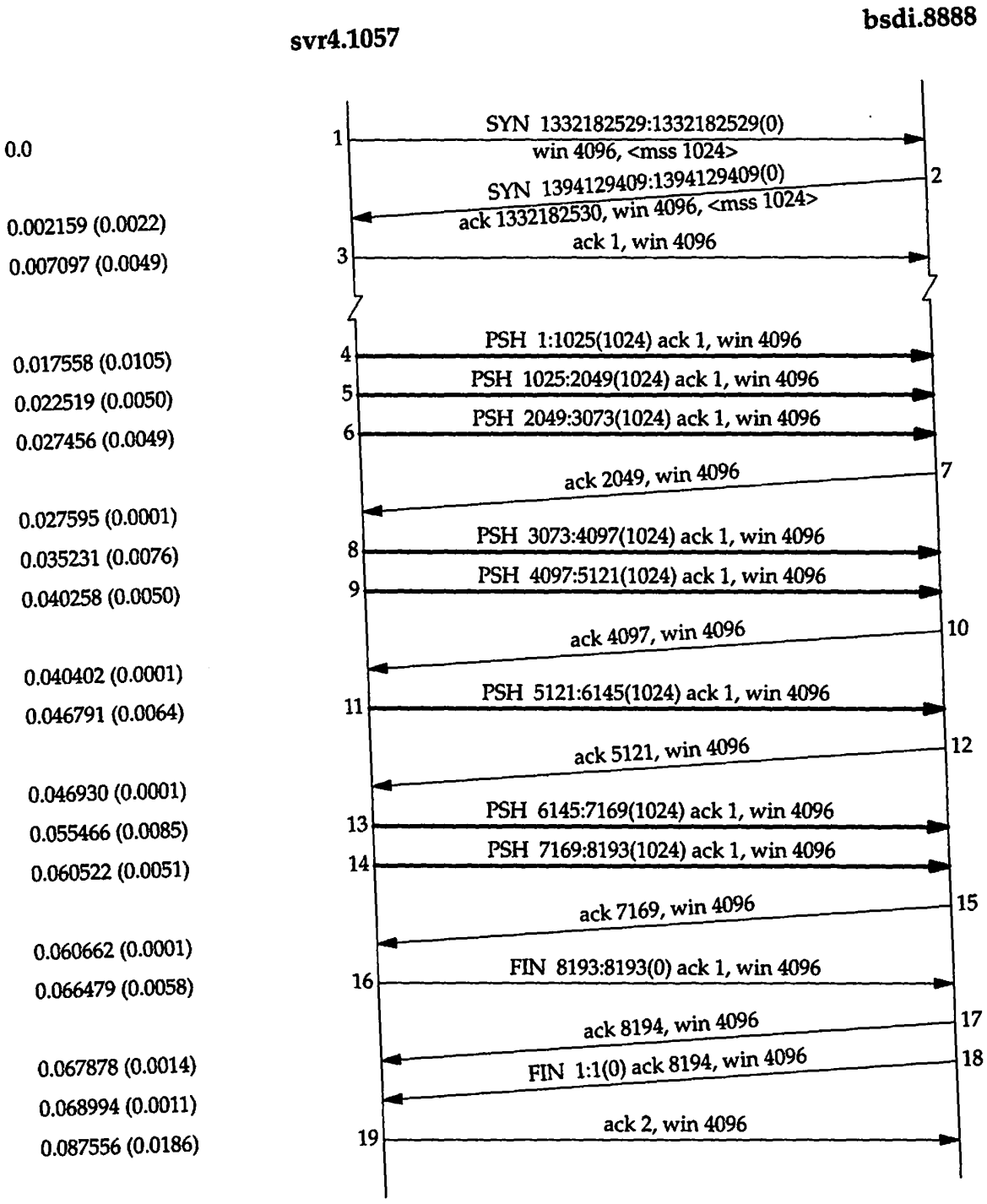


Figure 20.2 Another transfer of 8192 bytes from `svr4` to `bsdi`.

A few things have changed. This time the receiver does not send an ACK of 3073; instead it waits and sends the ACK of 4097. The receiver sends only four ACKs (segments 7, 10, 12, and 15): three of these are for 2048 bytes and one for 1024 bytes. The ACK of the final 1024 bytes of data appears in segment 17, along with the ACK of the FIN. (Compare segment 17 in this figure with segments 16 and 18 in Figure 20.1.)

Fast Sender, Slow Receiver

Figure 20.3 shows another time line, this time from a fast sender (a Sparc) to a slow receiver (an 80386 with a slow Ethernet card). The dynamics are different again.

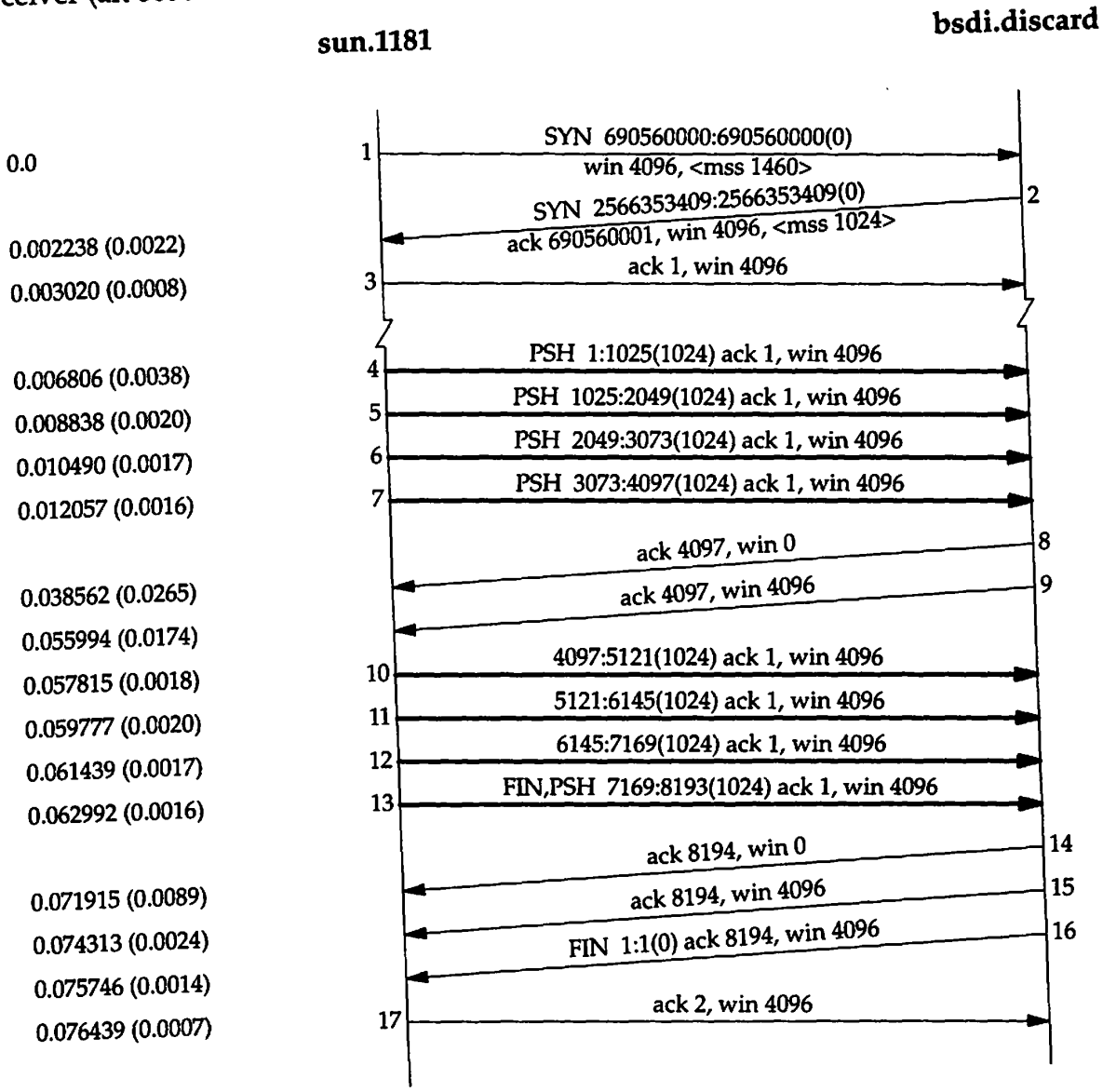


Figure 20.3 Sending 8192 bytes from a fast sender to a slow receiver.

The sender transmits four back-to-back data segments (4–7) to fill the receiver's window. The sender then stops and waits for an ACK. The receiver sends the ACK (segment 8) but the advertised window is 0. This means the receiver has all the data, but it's all in the receiver's TCP buffers, because the application hasn't had a chance to read the data. Another ACK (called a *window update*) is sent 17.4 ms later, announcing that the receiver can now receive another 4096 bytes. Although this looks like an ACK, it is called a window update because it does not acknowledge any new data, it just advances the right edge of the window.

The sender transmits its final four segments (10–13), again filling the receiver's window. Notice that segment 13 contains two flag bits: PUSH and FIN. This is followed by another two ACKs from the receiver. Both of these acknowledge the final 4096 bytes of data (bytes 4097 through 8192) and the FIN (numbered 8193).

20.3 Sliding Windows

The sliding window protocol that we observed in the previous section can be visualized as shown in Figure 20.4.

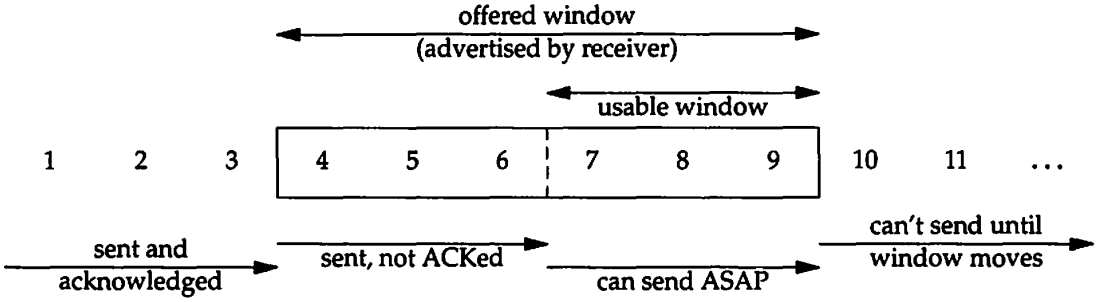


Figure 20.4 Visualization of TCP sliding window.

In this figure we have numbered the bytes 1 through 11. The window advertised by the receiver is called the *offered window* and covers bytes 4 through 9, meaning that the receiver has acknowledged all bytes up through and including number 3, and has advertised a window size of 6. Recall from Chapter 17 that the window size is relative to the acknowledged sequence number. The sender computes its *usable window*, which is how much data it can send immediately.

Over time this sliding window moves to the right, as the receiver acknowledges data. The relative motion of the two ends of the window increases or decreases the size of the window. Three terms are used to describe the movement of the right and left edges of the window.

1. The window *closes* as the left edge advances to the right. This happens when data is sent and acknowledged.
2. The window *opens* when the right edge moves to the right, allowing more data to be sent. This happens when the receiving process on the other end reads acknowledged data, freeing up space in its TCP receive buffer.
3. The window *shrinks* when the right edge moves to the left. The Host Requirements RFC strongly discourages this, but TCP must be able to cope with a peer that does this. Section 22.3 shows an example when one side would like to shrink the window by moving the right edge to the left, but cannot.

Figure 20.5 shows these three terms. The left edge of the window cannot move to the left, because this edge is controlled by the acknowledgment number received from

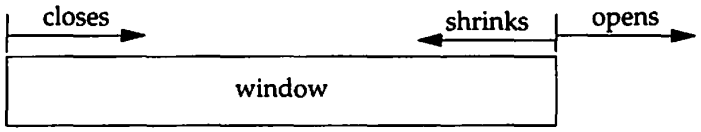


Figure 20.5 Movement of window edges.

the other end. If an ACK were received that implied moving the left edge to the left, it is a duplicate ACK, and discarded.

If the left edge reaches the right edge, it is called a *zero window*. This stops the sender from transmitting any data.

An Example

Figure 20.6 shows the dynamics of TCP's sliding window protocol for the data transfer in Figure 20.1.

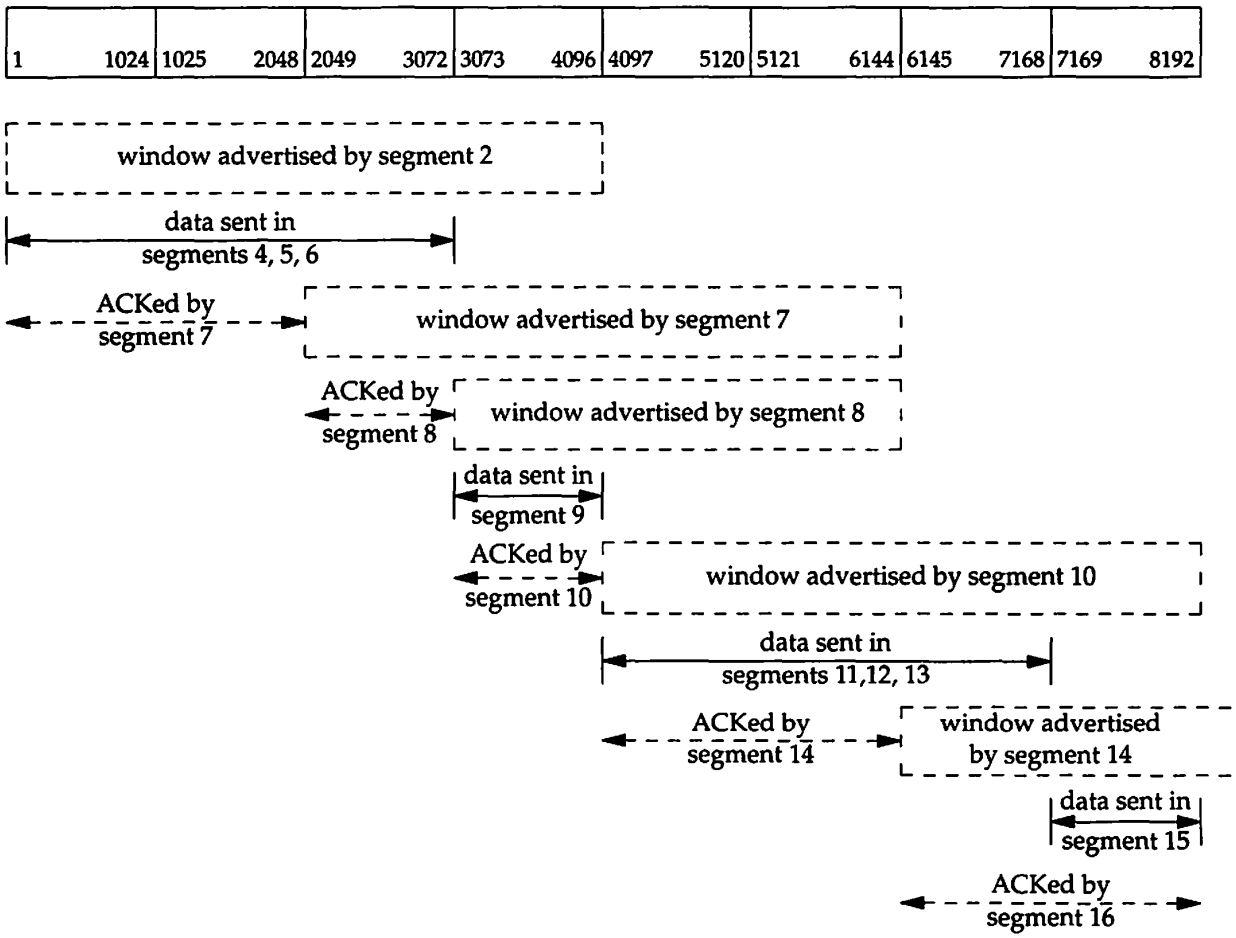


Figure 20.6 Sliding window protocol for Figure 20.1.

There are numerous points that we can summarize using this figure as an example.

1. The sender does not have to transmit a full window's worth of data.
2. One segment from the receiver acknowledges data and slides the window to the right. This is because the window size is relative to the acknowledged sequence number.
3. The size of the window can decrease, as shown by the change from segment 7 to segment 8, but the right edge of the window must not move leftward.
4. The receiver does not have to wait for the window to fill before sending an ACK. We saw earlier that many implementations send an ACK for every two segments that are received.

We'll see more examples of the dynamics of the sliding window protocol in later examples.

20.4 Window Size

The size of the window offered by the receiver can usually be controlled by the receiving process. This can affect the TCP performance.

> 4.2BSD defaulted the send buffer and receive buffer to 2048 bytes each. With 4.3BSD both were increased to 4096 bytes. As we can see from all the examples so far in this text, SunOS 4.1.3, BSD/386, and SVR4 still use this 4096-byte default. Other systems, such as Solaris 2.2, 4.4BSD, and AIX 3.2, use larger default buffer sizes, such as 8192 or 16384 bytes.
>
> The sockets API allows a process to set the sizes of the send buffer and the receive buffer. The size of the receive buffer is the maximum size of the advertised window for that connection. Some applications change the socket buffer sizes to increase performance.

[Mogul 1993] shows some results for file transfer between two workstations on an Ethernet, with varying sizes for the transmit buffer and receive buffer. (For a one-way flow of data such as file transfer, it is the size of the transmit buffer on the sending side and the size of the receive buffer on the receiving side that matters.) The common default of 4096 bytes for both is not optimal for an Ethernet. An approximate 40% increase in throughput is seen by just increasing both buffers to 16384 bytes. Similar results are shown in [Papadopoulos and Parulkar 1993].

In Section 20.7 we'll see how to calculate the minimum buffer size, given the bandwidth of the communication media and the round-trip time between the two ends.

An Example

We can control the sizes of these buffers with our `sock` program. We invoke the server as:

```
bsdi % sock -i -s -R6144 5555
```

which sets the size of the receive buffer (-R option) to 6144 bytes. We then start the client on the host sun and have it perform one write of 8192 bytes:

```
sun % sock -i -n1 -w8192 bsdi 5555
```

Figure 20.7 shows the results.

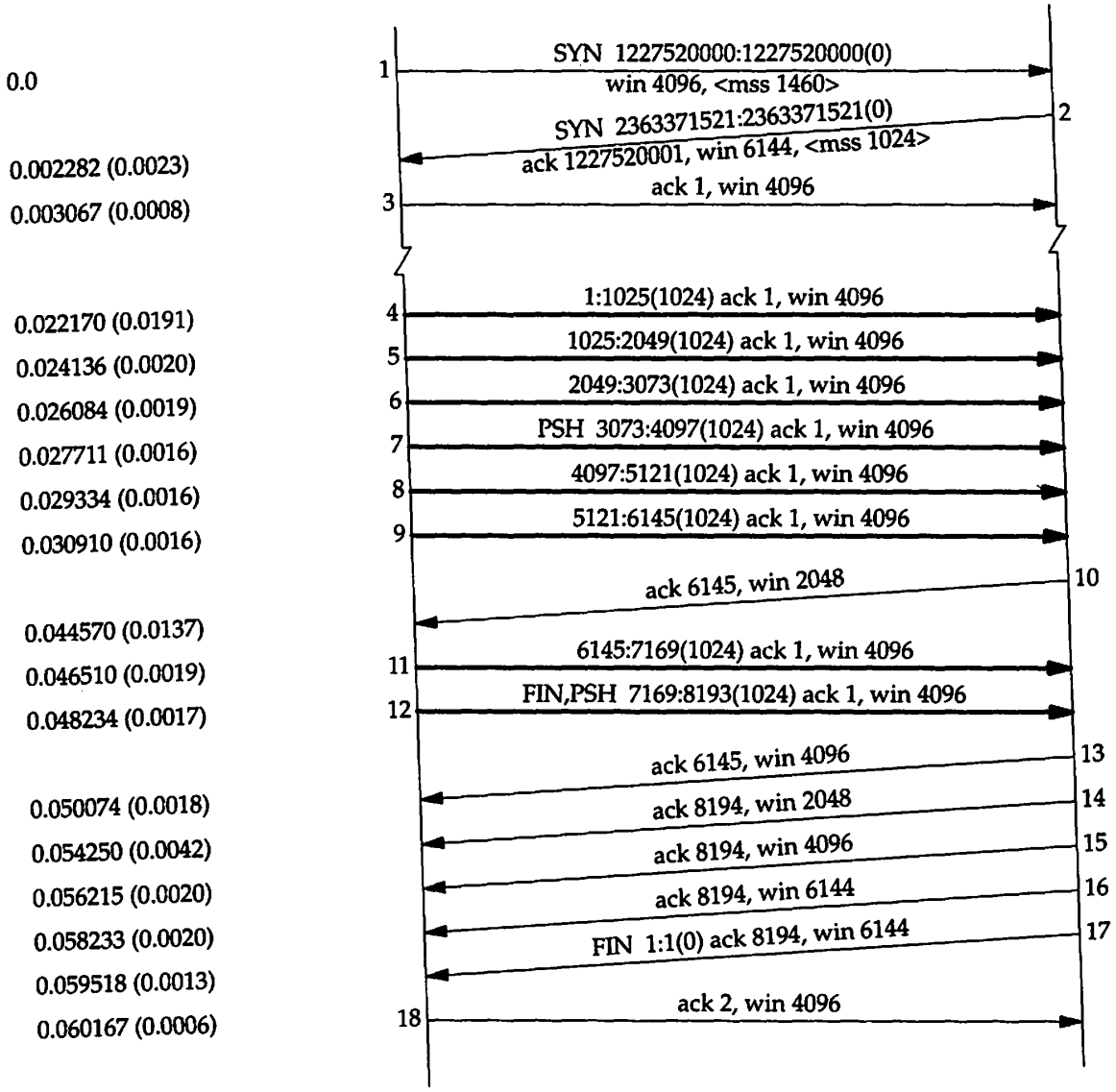


Figure 20.7 Data transfer with receiver offering a window size of 6144 bytes.

First notice that the receiver's window size is offered as 6144 bytes in segment 2. Because of this larger window, the client sends six segments immediately (segments 4–9), and then stops. Segment 10 acknowledges all the data (bytes 1 through 6144) but offers a window of only 2048, probably because the receiving application hasn't had a chance to read more than 2048 bytes. Segments 11 and 12 complete the data transfer from the client, and this final data segment also carries the FIN flag.

The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four. This provides an exponential increase.

At some point the capacity of the internet can be reached, and an intermediate router will start discarding packets. This tells the sender that its congestion window has gotten too large. When we talk about TCP's timeout and retransmission algorithms in the next chapter, we'll see how this is handled, and what happens to the congestion window. For now, let's watch slow start in action.

An Example

Figure 20.8 shows data being sent from the host `sun` to the host `vangogh.cs.berkeley.edu`. The data traverses a slow SLIP link, which should be the bottleneck. (We have removed the connection establishment from this time line.)

We see the sender transmit one segment with 512 bytes of data and then wait for its ACK. The ACK is received 716 ms later, which is an indicator of the round-trip time. The congestion window is then increased to two segments, and two segments are sent. When the ACK in segment 5 is received, the congestion window is increased to three segments. Although three more could be sent, only two are sent before another ACK is received.

We'll return to slow start in Section 21.6 and see how it's normally implemented with another technique called *congestion avoidance*.

20.7 Bulk Data Throughput

Let's look at the interaction of the window size, the windowed flow control, and slow start on the throughput of a TCP connection carrying bulk data.

Figure 20.9 shows the steps over time of a connection between a sender on the left and a receiver on the right. Sixteen units of time are shown. We show only discrete units of time in this figure, for simplicity. We show segments carrying data going from the left to right in the top half of each picture, numbered 1, 2, 3, and so on. The ACKs go in the other direction in the bottom half of each picture. We draw the ACKs smaller, and show the segment number being acknowledged.

which sets the size of the receive buffer (-R option) to 6144 bytes. We then start the client on the host sun and have it perform one write of 8192 bytes:

```
sun % sock -i -n1 -w8192 bsdi 5555
```

Figure 20.7 shows the results.

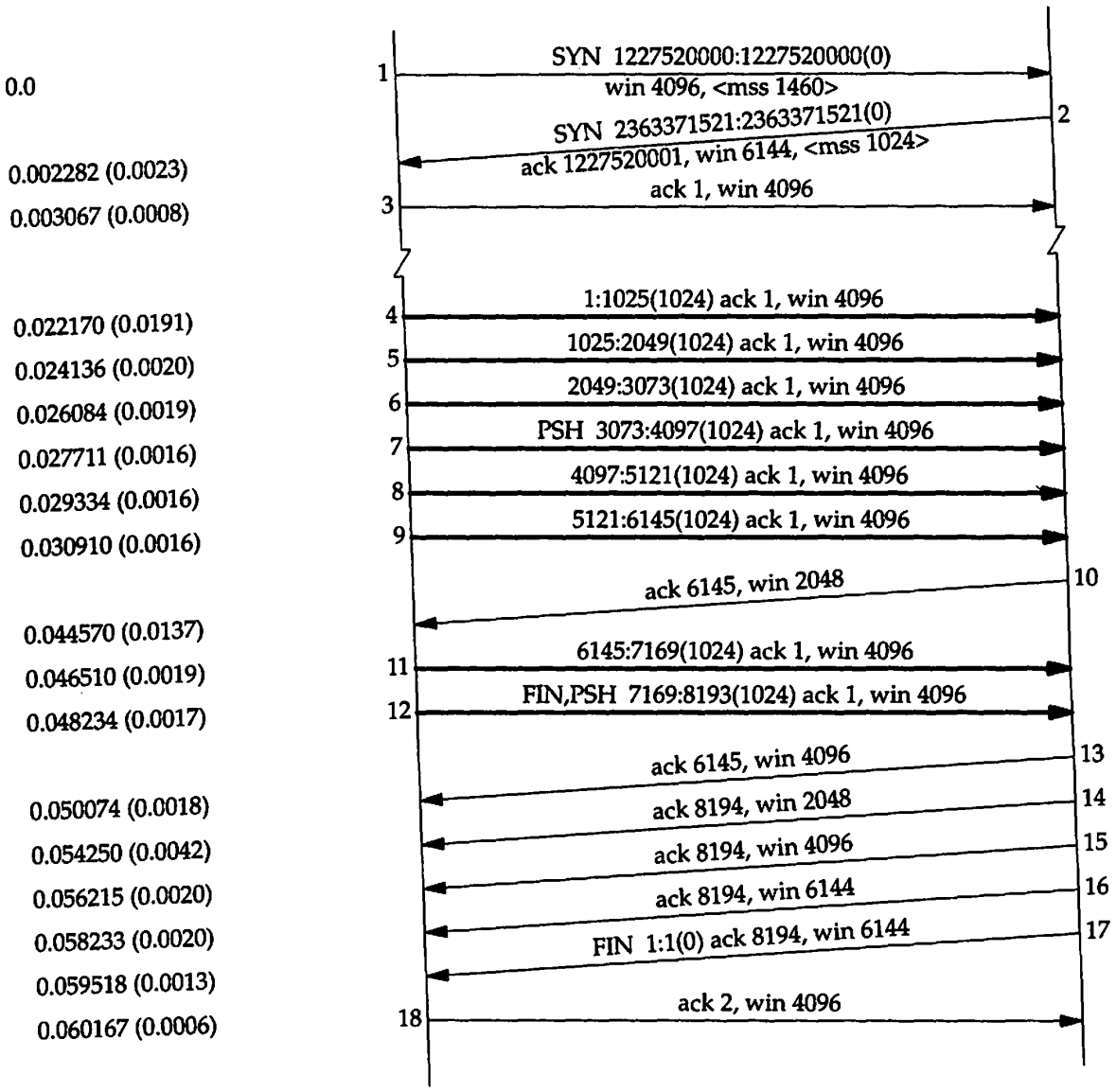


Figure 20.7 Data transfer with receiver offering a window size of 6144 bytes.

First notice that the receiver's window size is offered as 6144 bytes in segment 2. Because of this larger window, the client sends six segments immediately (segments 4–9), and then stops. Segment 10 acknowledges all the data (bytes 1 through 6144) but offers a window of only 2048, probably because the receiving application hasn't had a chance to read more than 2048 bytes. Segments 11 and 12 complete the data transfer from the client, and this final data segment also carries the FIN flag.

Segment 13 contains the same acknowledgment sequence number as segment 10, but advertises a larger window. Segment 14 acknowledges the final 2048 bytes of data and the FIN, and segments 15 and 16 just advertise a larger window. Segments 17 and 18 complete the normal close.

20.5 PUSH Flag

We've seen the PUSH flag in every one our TCP examples, but we've never described its use. It's a notification from the sender to the receiver for the receiver to pass all the data that it has to the receiving process. This data would consist of whatever is in the segment with the PUSH flag, along with any other data the receiving TCP has collected for the receiving process.

In the original TCP specification, it was assumed that the programming interface would allow the sending process to tell its TCP when to set the PUSH flag. In an interactive application, for example, when the client sent a command to the server, the client would set the PUSH flag and wait for the server's response. (In Exercise 19.1 we could imagine the client setting the PUSH flag when the 12-byte request is written.) By allowing the client application to tell its TCP to set the flag, it was a notification to the client's TCP that the client process didn't want the data to hang around in the TCP buffer, waiting for additional data, before sending a segment to the server. Similarly, when the server's TCP received the segment with the PUSH flag, it was a notification to pass the data to the server process and not wait to see if any additional data arrives.

Today, however, most APIs don't provide a way for the application to tell its TCP to set the PUSH flag. Indeed, many implementors feel the need for the PUSH flag is outdated, and a good TCP implementation can determine when to set the flag by itself.

Most Berkeley-derived implementations automatically set the PUSH flag if the data in the segment being sent empties the send buffer. This means we normally see the PUSH flag set for each application write, because data is usually sent when it's written.

> A comment in the code indicates this algorithm is to please those implementations that only pass received data to the application when a buffer fills or a segment is received with the PUSH flag.
>
> It is not possible using the sockets API to tell TCP to turn on the PUSH flag or to tell whether the PUSH flag was set in received data.

Berkeley-derived implementations ignore a received PUSH flag because they normally never delay the delivery of received data to the application.

Examples

In Figure 20.1 (p. 276) we see the PUSH flag turned on for all eight data segments (4–6, 9, 11–13, and 15). This is because the client did eight writes of 1024 bytes, and each write emptied the send buffer.

Look again at Figure 20.7 (p. 283). We expect the PUSH flag to be set on segment 12, since that is the final data segment. Why was the PUSH flag set on segment 7, when the

sender knew there were still more bytes to send? The reason is that the size of the sender's send buffer is 4096 bytes, even though we specified a single write of 8192 bytes.

Another point to note in Figure 20.7 concerns the three consecutive ACKs, segments 14, 15, and 16. We saw two consecutive ACKs in Figure 20.3, but that was because the receiver had advertised a window of 0 (stopping the sender) so when the window opened up, another ACK was required, with the nonzero window, to restart the sender. In Figure 20.7, however, the window never reaches 0. Nevertheless, when the size of the window increases by 2048 bytes, another ACK is sent (segments 15 and 16) to provide this window update to the other end. (These two window updates in segments 15 and 16 are not needed, since the FIN has been received from the other end, preventing it from sending any more data.) Many implementations send this window update if the window increases by either two maximum sized segments (2048 bytes in this example, with an MSS of 1024) or 50% of the maximum possible window (2048 bytes in this example, with a maximum window of 4096). We'll see this again in Section 22.3 when we examine the silly window syndrome in detail.

As another example of the PUSH flag, look again at Figure 20.3 (p. 279). The reason we see the flag on for the first four data segments (4–7) is because each one caused a segment to be generated by TCP and passed to the IP layer. But then TCP had to stop, waiting for an ACK to move the 4096-byte window. While waiting for the ACK, TCP takes the final 4096 bytes of data from the application. When the window opens up (segment 9) the sending TCP knows it has four segments that it can send immediately, so it only turns on the PUSH flag for the final segment (13).

20.6 Slow Start

In all the examples we've seen so far in this chapter, the sender starts off by injecting multiple segments into the network, up to the window size advertised by the receiver. While this is OK when the two hosts are on the same LAN, if there are routers and slower links between the sender and the receiver, problems can arise. Some intermediate router must queue the packets, and it's possible for that router to run out of space. [Jacobson 1988] shows how this naive approach can reduce the throughput of a TCP connection drastically.

TCP is now required to support an algorithm called *slow start*. It operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgments are returned by the other end.

Slow start adds another window to the sender's TCP: the *congestion window*, called *cwnd*. When a new connection is established with a host on another network, the congestion window is initialized to one segment (i.e., the segment size announced by the other end). Each time an ACK is received, the congestion window is increased by one segment. (*cwnd* is maintained in bytes, but slow start always increments it by the segment size.) The sender can transmit up to the minimum of the congestion window and the advertised window. The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver.

The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four. This provides an exponential increase.

At some point the capacity of the internet can be reached, and an intermediate router will start discarding packets. This tells the sender that its congestion window has gotten too large. When we talk about TCP's timeout and retransmission algorithms in the next chapter, we'll see how this is handled, and what happens to the congestion window. For now, let's watch slow start in action.

An Example

Figure 20.8 shows data being sent from the host `sun` to the host `vangogh.cs.berkeley.edu`. The data traverses a slow SLIP link, which should be the bottleneck. (We have removed the connection establishment from this time line.)

We see the sender transmit one segment with 512 bytes of data and then wait for its ACK. The ACK is received 716 ms later, which is an indicator of the round-trip time. The congestion window is then increased to two segments, and two segments are sent. When the ACK in segment 5 is received, the congestion window is increased to three segments. Although three more could be sent, only two are sent before another ACK is received.

We'll return to slow start in Section 21.6 and see how it's normally implemented with another technique called *congestion avoidance*.

20.7 Bulk Data Throughput

Let's look at the interaction of the window size, the windowed flow control, and slow start on the throughput of a TCP connection carrying bulk data.

Figure 20.9 shows the steps over time of a connection between a sender on the left and a receiver on the right. Sixteen units of time are shown. We show only discrete units of time in this figure, for simplicity. We show segments carrying data going from the left to right in the top half of each picture, numbered 1, 2, 3, and so on. The ACKs go in the other direction in the bottom half of each picture. We draw the ACKs smaller, and show the segment number being acknowledged.

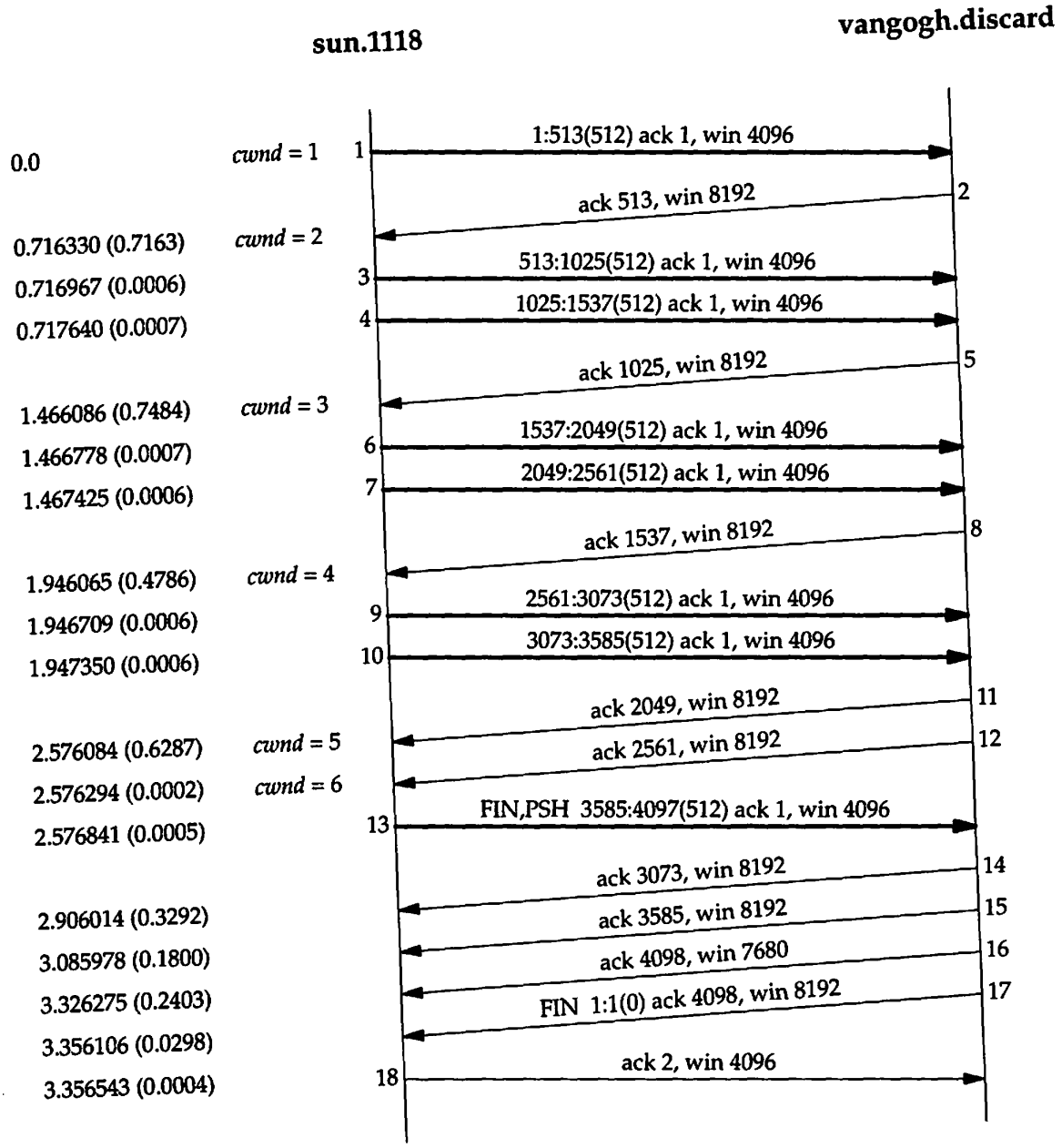


Figure 20.8 Example of slow start.

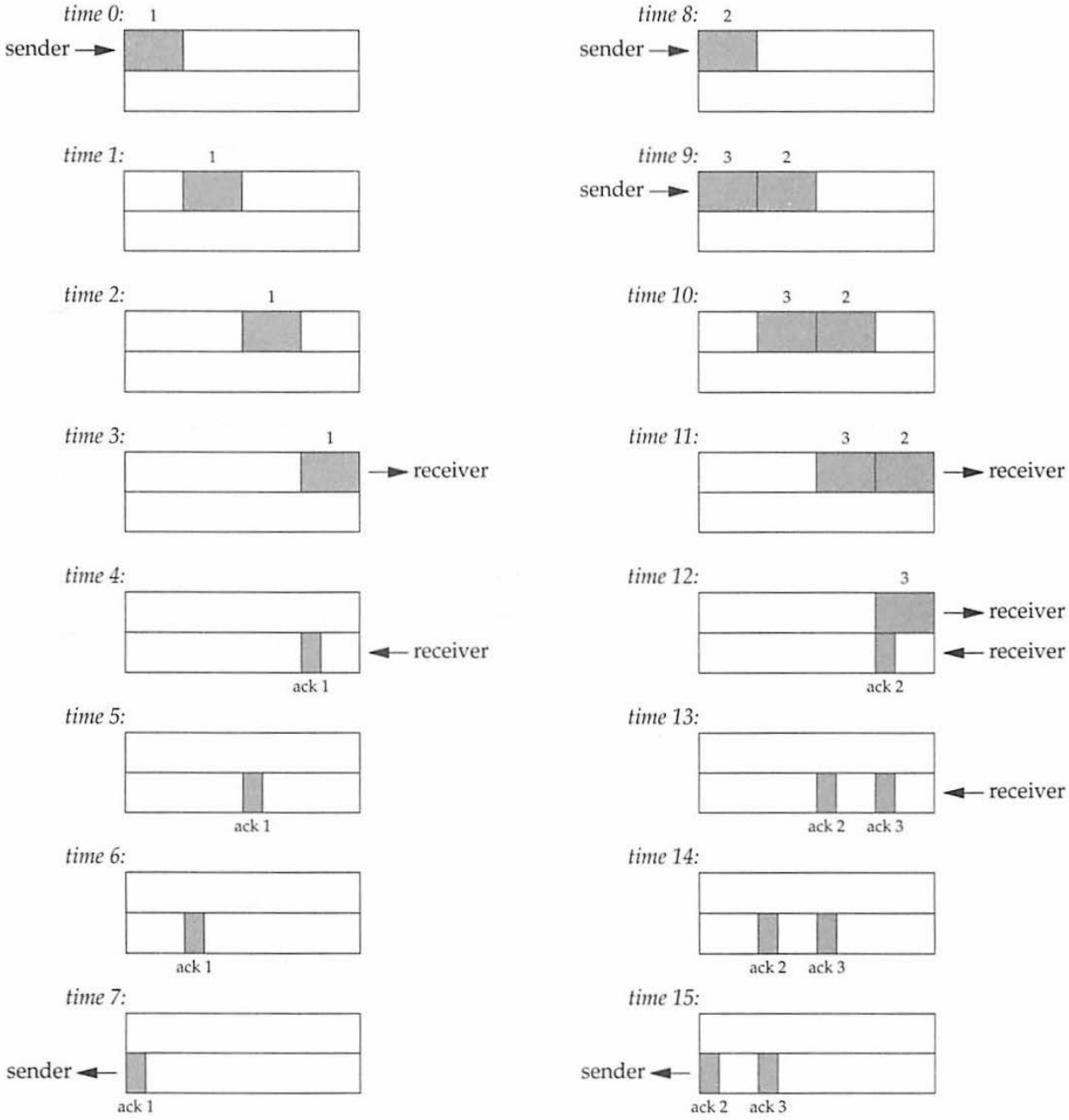


Figure 20.9 Times 0–15 for bulk data throughput example.

At time 0 the sender transmits one segment. Since the sender is in slow start (its congestion window is one segment), it must wait for the acknowledgment of this segment before continuing.

At times 1, 2, and 3 the segment moves one unit of time to the right. At time 4 the receiver reads the segment and generates the acknowledgment. At times 5, 6, and 7 the ACK moves to the left one unit, back to the sender. We have a round-trip time (RTT) of 8 units of time.

We have purposely drawn the ACK segment smaller than the data segment, since it's normally just an IP header and a TCP header. We're showing only a unidirectional

flow of data here. Also, we assume that the ACK moves at the same speed as the data segment, which isn't always true.

> In general the time to send a packet depends on two factors: a propagation delay (caused by the finite speed of light, latencies in transmission equipment, etc.) and a transmission delay that depends on the speed of the media (how many bits per second the media can transmit). For a given path between two nodes the propagation delay is fixed while the transmission delay depends on the packet size. At lower speeds the transmission delay dominates (e.g., Exercise 7.2 where we didn't even consider the propagation delay), whereas at gigabit speeds the propagation delay dominates (e.g., Figure 24.6).

When the sender receives the ACK it can transmit two more segments (which we've numbered 2 and 3), at times 8 and 9. Its congestion window is now two segments. These two segments move right toward the receiver, where the ACKs are generated at times 12 and 13. The spacing of the ACKs returned to the sender is identical to the spacing of the data segments. This is called the *self-clocking* behavior of TCP. Since the receiver can only generate ACKs when the data arrives, the spacing of the ACKs at the sender identifies the arrival rate of the data at the receiver. (In actuality, however, queueing on the return path can change the arrival rate of the ACKs.)

Figure 20.10 shows the next 16 time units. The arrival of the two ACKs increases the congestion window from two to four segments, and these four segments are sent at times 16–19. The first of the ACKs returns at time 23. The four ACKs increase the congestion window from four to eight segments, and these eight segments are transmitted at times 24–31.

At time 31, and at all successive times, the pipe between the sender and receiver is full. It cannot hold any more data, regardless of the congestion window or the window advertised by the receiver. Each unit of time a segment is removed from the network by the receiver, and another is placed into the network by the sender. However many data segments fill the pipe, there are an equal number of ACKs making the return trip. This is the ideal steady state of the connection.

Bandwidth-Delay Product

We can now answer the question: how big should the window be? In our example, the sender needs to have eight segments outstanding and unacknowledged at any time, for maximum throughput. The receiver's advertised window must be that large, since that limits how much the sender can transmit.

We can calculate the capacity of the pipe as

$$\textit{capacity}\ (\text{bits}) = \textit{bandwidth}\ (\text{bits/sec}) \times \textit{round-trip time}\ (\text{sec})$$

This is normally called the *bandwidth-delay product*. This value can vary widely, depending on the network speed and the RTT between the two ends. For example, a T1 telephone line (1,544,000 bits/sec) across the United States (about a 60-ms RTT) gives a bandwidth-delay product of 11,580 bytes. This is reasonable in terms of the buffer sizes we talked about in Section 20.4, but a T3 telephone line (45,000,000 bits/sec) across the United States gives a bandwidth-delay product of 337,500 bytes, which is bigger than the maximum allowable TCP window advertisement (65535 bytes). We describe the

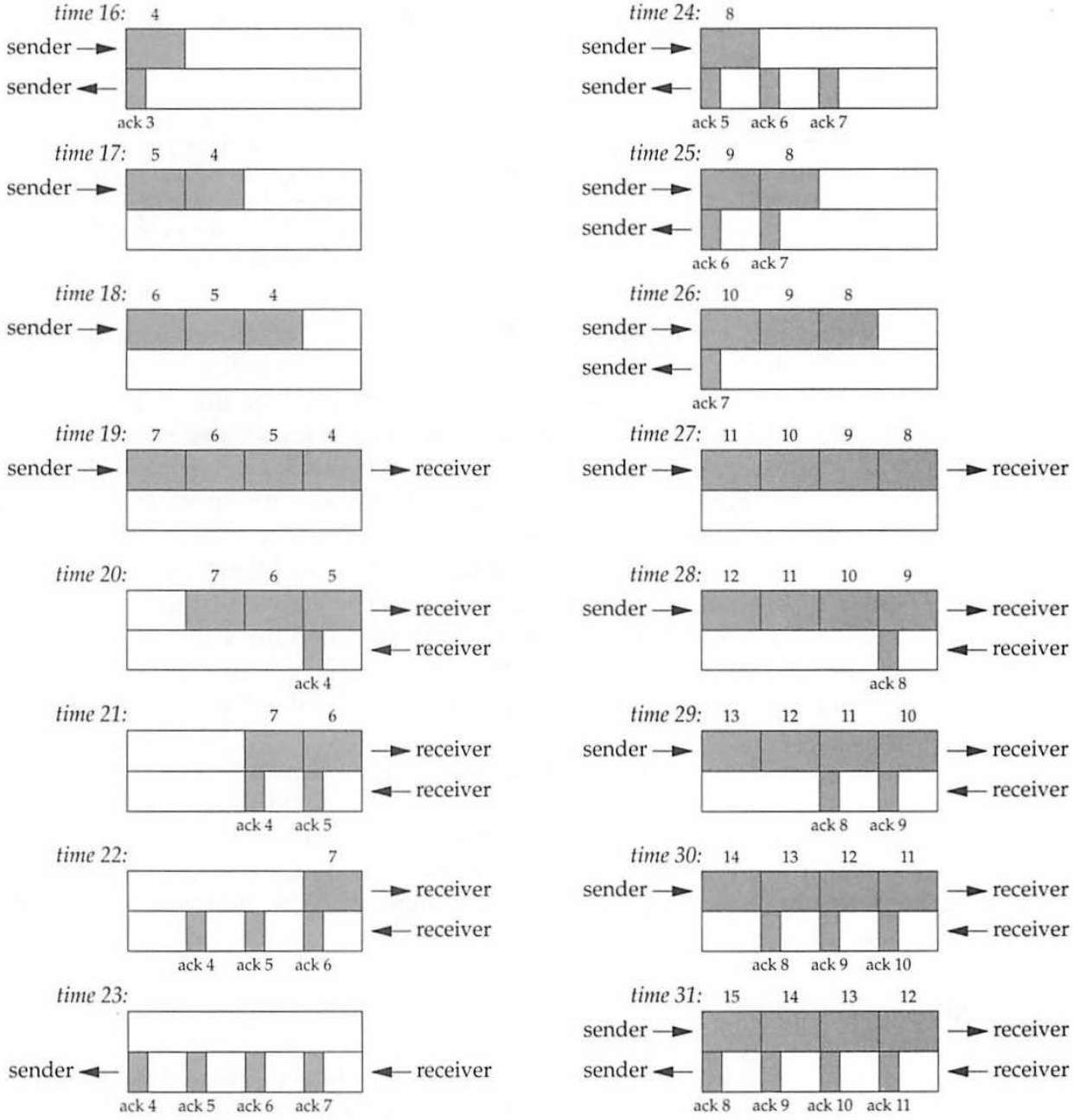


Figure 20.10 Times 16–31 for bulk data throughput example.

new TCP window scale option in Section 24.4 that gets around this current limitation of TCP.

> The value 1,544,000 bits/sec for a T1 phone line is the raw bit rate. The data rate is actually 1,536,000 bits/sec, since 1 bit in 193 is used for framing. The raw bit rate of a T3 phone line is actually 44,736,000 bits/sec, and the data rate can reach 44,210,000 bits/sec. For our discussion we'll use 1.544 Mbits/sec and 45 Mbits/sec.

Either the bandwidth or the delay can affect the capacity of the pipe between the sender and receiver. In Figure 20.11 we show graphically how a doubling of the RTT doubles the capacity of the pipe.

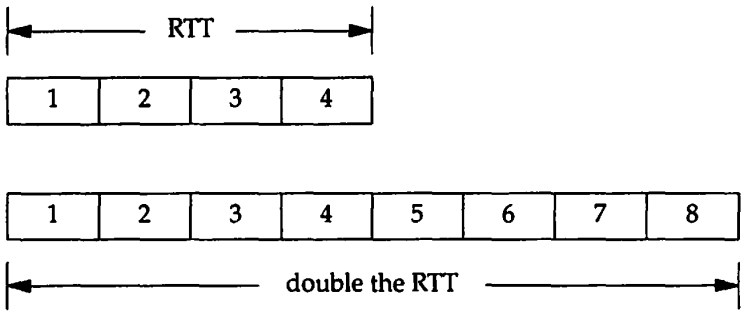


Figure 20.11 Doubling the RTT doubles the capacity of the pipe.

In the lower illustration of Figure 20.11, with the longer RTT, the pipe can hold eight segments, instead of four.

Similarly, Figure 20.12 shows that doubling the bandwidth also doubles the capacity of the pipe.

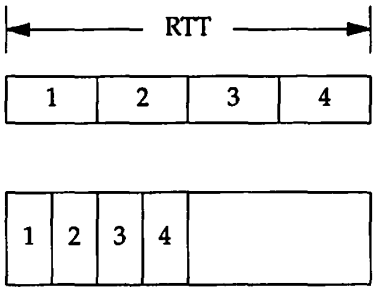


Figure 20.12 Doubling the bandwidth doubles the capacity of the pipe.

In the lower illustration of Figure 20.12, we assume that the network speed has doubled, allowing us to send four segments in half the time as in the top picture. Again, the capacity of the pipe has doubled. (We assume that the segments in the top half of this figure have the same area, that is the same number of bits, as the segments in the bottom half.)

Congestion

Congestion can occur when data arrives on a big pipe (a fast LAN) and gets sent out a smaller pipe (a slower WAN). Congestion can also occur when multiple input streams arrive at a router whose output capacity is less than the sum of the inputs.

Figure 20.13 shows a typical scenario with a big pipe feeding a smaller pipe. We say this is typical because most hosts are connected to LANs, with an attached router that is connected to a slower WAN. (Again, we are assuming the areas of all the data segments (9–20) in the top half of the figure are all the same, and the areas of all the acknowledgments in the bottom half are all the same.)

In this figure we have labeled the router R1 as the "bottleneck," because it is the congestion point. It can receive packets from the LAN on its left faster than they can be

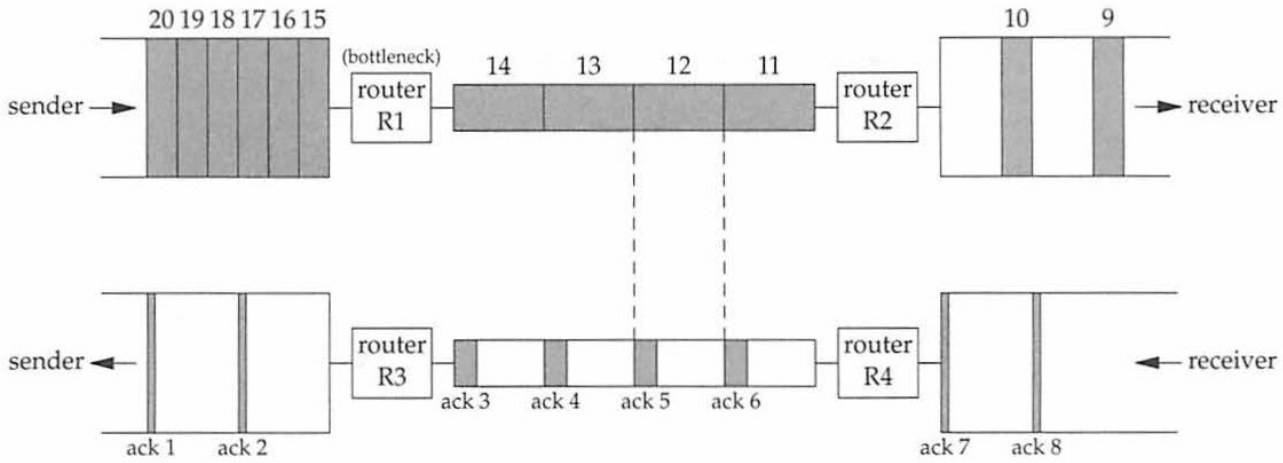


Figure 20.13 Congestion caused by a bigger pipe feeding a smaller pipe.

sent out the WAN on its right. (Commonly R1 and R3 are the same router, as are R2 and R4, but that's not required; asymmetrical paths can occur.) When router R2 puts the received packets onto the LAN on its right, they maintain the same spacing as they did on the WAN on its left, even though the bandwidth of the LAN is higher. Similarly, the spacing of the ACKs on their way back is the same as the spacing of the slowest link in the path.

In Figure 20.13 we have assumed that the sender did not use slow start, and sent the segments we've numbered 1–20 as fast as the LAN could take them. (This assumes the receiving host advertised a window of at least 20 segments.) The spacing of the ACKs will correspond to the bandwidth of the slowest link, as we show. We are assuming the bottleneck router has adequate buffering for all 20 segments. This is not guaranteed, and can lead to that router discarding packets. We'll see how to avoid this when we talk about congestion avoidance in Section 21.6.

20.8 Urgent Mode

TCP provides what it calls *urgent mode*, allowing one end to tell the other end that "urgent data" of some form has been placed into the normal stream of data. The other end is notified that this urgent data has been placed into the data stream, and it's up to the receiving end to decide what to do.

The notification from one end to the other that urgent data exists in the data stream is done by setting two fields in the TCP header (Figure 17.2, p. 225). The URG bit is turned on and the 16-bit *urgent pointer* is set to a positive offset that must be added to the sequence number field in the TCP header to obtain the sequence number of the last byte of urgent data.

There is continuing debate about whether the urgent pointer points to the last byte of urgent data, or to the byte following the last byte of urgent data. The original TCP specification gave

both interpretations but the Host Requirements RFC identifies which is correct: the urgent pointer points to the last byte of urgent data.

The problem, however, is that most implementations (i.e., the Berkeley-derived implementations) continue to use the wrong interpretation. An implementation that follows the specification in the Host Requirements RFC might be compliant, but might not communicate correctly with most other hosts.

TCP must inform the receiving process when an urgent pointer is received and one was not already pending on the connection, or if the urgent pointer advances in the data stream. The receiving application can then read the data stream and must be able to tell when the urgent pointer is encountered. As long as data exists from the receiver's current read position until the urgent pointer, the application is considered to be in an "urgent mode." After the urgent pointer is passed, the application returns to its normal mode.

TCP itself says little more about urgent data. There is no way to specify where the urgent data starts in the data stream. The only information sent across the connection by TCP is that urgent mode has begun (the URG bit in the TCP header) and the pointer to the last byte of urgent data. Everything else is left to the application.

Unfortunately many implementations incorrectly call TCP's urgent mode *out-of-band* data. If an application really wants a separate out-of-band channel, a second TCP connection is the easiest way to accomplish this. (Some transport layers do provide what most people consider true out-of-band data: a logically separate data path using the same connection as the normal data path. This is not what TCP provides.)

The confusion between TCP's urgent mode and out-of-band data is also because the predominant programming interface, the sockets API, maps TCP's urgent mode into what sockets calls out-of-band data.

What is urgent mode used for? The two most commonly used applications are Telnet and Rlogin, when the interactive user types the interrupt key, and we show examples of this use of urgent mode in Chapter 26. Another is FTP, when the interactive user aborts a file transfer, and we show an example of this in Chapter 27.

Telnet and Rlogin use urgent mode from the server to the client because it's possible for this direction of data flow to be stopped by the client TCP (i.e., it advertises a window of 0). But if the server process enters urgent mode, the server TCP immediately sends the urgent pointer and the URG flag, even though it can't send any data. When the client TCP receives this notification, it in turn notifies the client process, so the client can read its input from the server, to open the window, and let the data flow.

What happens if the sender enters urgent mode multiple times before the receiver processes all the data up through the first urgent pointer? The urgent pointer just advances in the data stream, and its previous position at the receiver is lost. There is only one urgent pointer at the receiver and its value is overwritten when a new value for the urgent pointer arrives from the other end. This means if the contents of the data stream that are written by the sender when it enters urgent mode are important to the receiver, these data bytes must be specially marked (somehow) by the sender. We'll see that Telnet marks all of its command bytes in the data stream by prefixing them with a byte of 255.

An Example

Let's watch how TCP sends urgent data, even when the receiver's window is closed. We'll start our `sock` program on the host `bsdi` and have it pause for 10 seconds after the connection is established (the `-P` option), before it reads from the network. This lets the other end fill the send window.

```
bsdi % sock -i -s -P10 5555
```

We then start the client on the host `sun` telling it to use a send buffer of 8192 bytes (`-S` option) and perform six 1024-byte writes to the network (`-n` option). We also specify `-U5` telling it to write 1 byte of data and enter urgent mode before writing the fifth buffer to the network. We specify the verbose flag to see the order of the writes:

```
sun % sock -v -i -n6 -S8192 -U5 bsdi 5555
connected on 140.252.13.33.1305 to 140.252.13.35.5555
SO_SNDBUF = 8192
TCP_MAXSEG = 1024
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1 byte of urgent data
wrote 1024 bytes
wrote 1024 bytes
```

We set the send buffer size to 8192 bytes, to let the sending application immediately write all of its data. Figure 20.14 shows the `tcpdump` output for this exchange. (We have removed the connection establishment.) Lines 1–5 show the sender filling the receiver's window with four 1024-byte segments. The sender is then stopped because the receiver's window is full. (The ACK on line 4 acknowledges data, but does not move the right edge of the window.)

After the fourth application write of normal data, the application writes 1 byte of data and enters urgent mode. Line 6 is the result of this application write. The urgent pointer is set to 4098. The urgent pointer is sent with the URG flag even though the sender cannot send any data.

Five of these ACKs are sent in about 13 ms (lines 6–10). The first is sent when the application writes 1 byte and enters urgent mode. The next two are sent when the application does the final two writes of 1024 bytes. (Even though TCP can't send these 2048 bytes of data, each time the application performs a write, the TCP output function is called, and when it sees that urgent mode has been entered, sends another urgent notification.) The fourth of these ACKs occurs when the application closes its end of the connection. (The TCP output function is again called.) The sending application terminates milliseconds after it starts—before the receiving application has issued its first write. TCP queues all the data and sends it when it can. (This is why we specified a send buffer size of 8192—so all the data can fit in the buffer.) The fifth of these ACKs is probably generated by the reception of the ACK on line 4. The sending TCP has probably already queued its fourth segment for output (line 5) before this ACK arrives. The receipt of this ACK from the other end also causes the TCP output routine to be called.

```
 1  0.0                  sun.1305 > bsdi.5555: P 1:1025(1024) ack 1 win 4096
 2  0.073743 (0.0737)    sun.1305 > bsdi.5555: P 1025:2049(1024) ack 1 win 4096
 3  0.096969 (0.0232)    sun.1305 > bsdi.5555: P 2049:3073(1024) ack 1 win 4096
 4  0.157514 (0.0605)    bsdi.5555 > sun.1305: . ack 3073 win 1024
 5  0.164267 (0.0068)    sun.1305 > bsdi.5555: P 3073:4097(1024) ack 1 win 4096

 6  0.167961 (0.0037)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 7  0.171969 (0.0040)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 8  0.176196 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
 9  0.180373 (0.0042)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098
10  0.180768 (0.0004)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

11  0.367533 (0.1868)    bsdi.5555 > sun.1305: . ack 4097 win 0
12  0.368478 (0.0009)    sun.1305 > bsdi.5555: . ack 1 win 4096 urg 4098

13  9.829712 (9.4612)    bsdi.5555 > sun.1305: . ack 4097 win 2048
14  9.831578 (0.0019)    sun.1305 > bsdi.5555: . 4097:5121(1024) ack 1 win 4096
                                                 urg 4098
15  9.833303 (0.0017)    sun.1305 > bsdi.5555: . 5121:6145(1024) ack 1 win 4096

16  9.835089 (0.0018)    bsdi.5555 > sun.1305: . ack 4097 win 4096
17  9.835913 (0.0008)    sun.1305 > bsdi.5555: FP 6145:6146(1) ack 1 win 4096
18  9.840264 (0.0044)    bsdi.5555 > sun.1305: . ack 6147 win 2048
19  9.842386 (0.0021)    bsdi.5555 > sun.1305: . ack 6147 win 4096
20  9.843622 (0.0012)    bsdi.5555 > sun.1305: F 1:1(0) ack 6147 win 4096
21  9.844320 (0.0007)    sun.1305 > bsdi.5555: . ack 2 win 4096
```

Figure 20.14 `tcpdump` output for TCP urgent mode.

The receiver then acknowledges the final 1024 bytes of data (line 11) but also advertises a window of 0. The sender responds with another segment containing the urgent notification.

The receiver advertises a window of 2048 bytes in line 13, when the application wakes up and reads some of the data from the receive buffer. The next two 1024-byte segments are sent (lines 14 and 15). The first segment has the urgent notification set, since the urgent pointer is within this segment. The second segment has turned the urgent notification off.

When the receiver opens the window again (line 16) the sender transmits the final byte of data (numbered 6145) and also initiates the normal connection termination.

Figure 20.15 shows the sequence numbers of the 6145 bytes of data that are sent. We see that the sequence number of the byte written when urgent mode was entered is 4097, but the value of the urgent pointer in Figure 20.14 is 4098. This confirms that this implementation (SunOS 4.1.3) sets the urgent pointer to 1 byte beyond the last byte of urgent data.

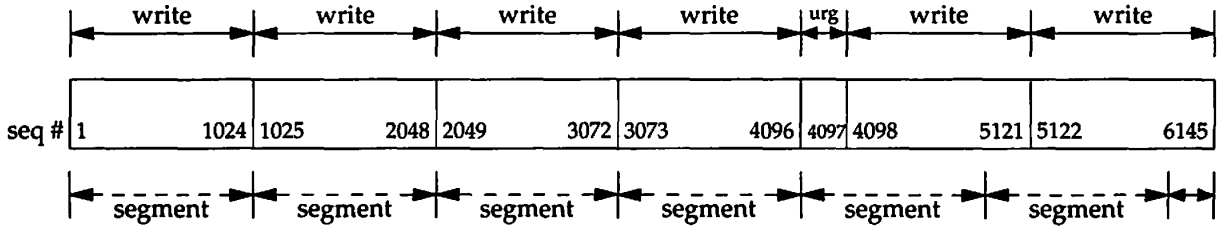


Figure 20.15 Application writes and TCP segments for urgent mode example.

This figure also lets us see how TCP repacketizes the data that the application wrote. The single byte that was output when urgent mode was entered is sent along with the next 1023 bytes of data in the buffer. The next segment also contains 1024 bytes of data, and the final segment contains 1 byte of data.

20.9 Summary

As we said early in the chapter, there is no single way to exchange bulk data using TCP. It is a dynamic process that depends on many factors, some of which we can control (e.g., send and receive buffer sizes) and some of which we have no control over (e.g., network congestion, implementation features). In this chapter we've examined many TCP transfers, explaining all the characteristics and algorithms that we could see.

Fundamental to the efficient transfer of bulk data is TCP's sliding window protocol. We then looked at what it takes for TCP to get the fastest transfer possible by keeping the pipe between the sender and receiver full. We measured the capacity of this pipe as the bandwidth-delay product, and saw the relationship between this and the window size. We return to this concept in Section 24.8 when we look at TCP performance.

We also looked at TCP's PUSH flag, since we'll always see it in trace output, but we have no control over its setting. The final topic was TCP's urgent data, which is often mistakenly called "out-of-band data." TCP's urgent mode is just a notification from the sender to the receiver that urgent data has been sent, along with the sequence number of the final byte of urgent data. The programming interface for the application to use with urgent data is often less than optimal, which leads to much confusion.

Exercises

20.1 In Figure 20.6 (p. 281) we could have shown a byte numbered 0 and a byte numbered 8193. What do these 2 bytes designate?

20.2 Look ahead to Figure 22.1 (p. 324) and explain the setting of the PUSH flag by the host `bsdi`.

20.3 In a Usenet posting someone complained about a throughput of 120,000 bits/sec on a 256,000 bits/sec link with a 128-ms delay between the United States and Japan (47% utilization), and a throughput of 33,000 bits/sec when the link was routed over a satellite (13% utilization). What does the window size appear to be for both cases? (Assume a 500-ms delay for the satellite link.) How big should the window be for the satellite link?

20.4 If the API provided a way for a sending application to tell its TCP to turn on the PUSH flag, and a way for the receiver to tell if the PUSH flag was on in a received segment, could the flag then be used as a record marker?

20.5 In Figure 20.3 why aren't segments 15 and 16 combined?

20.6 In Figure 20.13 we assume that the ACKs come back nicely spaced, corresponding to the spacing of the data segments. What happens if the ACKs are queued somewhere on the return path, causing a bunch of them to arrive at the same time at the sender?

ACK	acknowledgment flag, TCP header, p. 227
API	application program interface, p. 17
ARP	Address Resolution Protocol, p. 53
ARPANET	Advanced Research Projects Agency network, p. 548
AS	autonomous system, p. 128
ASCII	American Standard Code for Information Interchange, p. 401
ASN.1	Abstract Syntax Notation One, p. 386
BER	Basic Encoding Rules, p. 386
BGP	Border Gateway Protocol, p. 138
BIND	Berkeley Internet Name Domain, p. 188
BOOTP	Bootstrap Protocol, p. 215
BPF	BSD Packet Filter, p. 491
BSD	Berkeley Software Distribution, p. 16
CIDR	classless interdomain routing, p. 140
CIX	Commercial Internet Exchange, p. 119
CLNP	Connectionless Network Protocol, p. 50
CRC	cyclic redundancy check, p. 22
CSLIP	compressed SLIP, p. 25
CSMA	carrier sense multiple access, p. 21
DCE	Distributed Computing Environment, p. 462
DDN	Defense Data Network, p. 8
DF	don't fragment flag, IP header, p. 149
DHCP	Dynamic Host Configuration Protocol, p. 222
DLPI	Data Link Provider Interface, p. 494
DNS	Domain Name System, p. 187
DSAP	Destination Service Access Point, p. 22
DTS	Distributed Time Service, p. 77
DVMRP	Distance–Vector Multicast Routing Protocol, p. 185
EBONE	European IP Backbone, p. 119
EGP	Exterior Gateway Protocol, p. 128
EOL	end of option list, p. 93
FCS	frame check sequence, p. 22
FDDI	Fiber Distributed Data Interface, p. 4
FIFO	first in, first out, p. 259
FIN	finish flag, TCP header, p. 227
FQDN	fully qualified domain name, p. 189
FTP	File Transfer Protocol, p. 419
HDLC	high-level data link control, p. 26
HELLO	routing protocol, p. 128
IAB	Internet Architecture Board, p. 14
IANA	Internet Assigned Number Authority, p. 13
ICMP	Internet Control Message Protocol, p. 69
IDRP	Interdomain Routing Protocol, p. 141
IEEE	Institute of Electrical and Electronics Engineers, p. 21
IEN	Internet Experiment Notes, p. 172
IESG	Internet Engineering Steering Group, p. 14
IETF	Internet Engineering Task Force, p. 14
IGMP	Internet Group Management Protocol, p. 179
IGP	interior gateway protocol, p. 128
IP	Internet Protocol, p. 33
IRTF	Internet Research Task Force, p. 14
IS–IS	Intermediate System to Intermediate System Protocol, p. 141
ISN	initial sequence number, p. 226
ISO	International Organization for Standardization, p. 26
ISOC	Internet Society, p. 14
LAN	local area network, p. 3
LBX	low bandwidth X, p. 490
LCP	link control protocol, p. 26
LFN	long fat network, p. 344
LIFO	last in, first out, p. 259
LLC	logical link control, p. 22

LSRR	loose source and record route, p. 104
MBONE	multicast backbone, p. 186
MIB	management information base, p. 365
MILNET	Military Network, p. 483
MIME	multipurpose Internet mail extensions, p. 456
MSL	maximum segment lifetime, p. 242
MSS	maximum segment size, p. 236
MTA	message transfer agent, p. 442
MTU	maximum transmission unit, p. 29
NCP	Network Control Protocol, p. 15
NFS	Network File System, p. 461
NIC	Network Information Center, p. 8
NIT	network interface tap, p. 493
NNTP	Network News Transfer Protocol, p. 35
NOAO	National Optical Astronomy Observatories, p. 18
NOP	no operation, p. 93
NSFNET	National Science Foundation network, p. 103
NSI	NASA Science Internet, p. 103
NTP	Network Time Protocol, p. 77
NVT	network virtual terminal, p. 401
OSF	Open Software Foundation, p. 462
OSI	open systems interconnection, p. 26
OSPF	open shortest path first, p. 137
PAWS	protection against wrapped sequence numbers, p. 351
PDU	protocol data unit, p. 362
POSIX	Portable Operating System Interface, p. 479
PPP	Point-to-Point Protocol, p. 26
PSH	push flag, TCP header, p. 227
RARP	Reverse Address Resolution Protocol, p. 65
RFC	Request for Comment, p. 14
RIP	Routing Information Protocol, p. 129
RPC	remote procedure call, p. 461
RR	resource record, p. 201
RST	reset flag, TCP header, p. 246
RTO	retransmission time out, p. 299
RTT	round-trip time, p. 299
SACK	selective acknowledgment, p. 345
SLIP	Serial Line Internet Protocol, p. 24
SMI	structure of management information, p. 363
SMTP	Simple Mail Transfer Protocol, p. 441
SNMP	Simple Network Management Protocol, p. 359
SSAP	source service access point, p. 22
SSRR	strict source and record route, p. 104
SWS	silly window syndrome, p. 325
SYN	synchronize sequence numbers flag, TCP header, p. 231
TCP	Transmission Control Protocol, p. 223
TFTP	Trivial File Transfer Protocol, p. 209
TLI	Transport Layer Interface, p. 17
TOS	type-of-service, p. 34
TTL	time-to-live, p. 36
TUBA	TCP and UDP with bigger addresses, p. 50
Telnet	remote terminal protocol, p. 401
UDP	User Datagram Protocol, p. 143
URG	urgent pointer flag, TCP header, p. 292
UTC	Coordinated Universal Time, p. 74
UUCP	Unix-to-Unix Copy, p. 201
WAN	wide area network, p. 1
WWW	World Wide Web, p. 486
XDR	external data representation, p. 465
XID	transaction ID, p. 463
XTI	X/Open Transport Layer Interface, p. 17

TCP/IP Illustrated, Volume 1
Stevens
61346

Networking, TCP/IP

TCP/IP Illustrated, Volume 1

"The word *illustrated* distinguishes this book from its many rivals. Stevens uses the Lawrence Berkeley Laboratories `tcpdump` program to capture packets in promiscuous mode under a variety of OS and TCP/IP implementations. Studying `tcpdump` output helps you understand how the various protocols work."

—Stan Kelly-Bootle, *Unix Review*

TCP/IP Illustrated is a complete and detailed guide to the entire TCP/IP protocol suite—with an important difference from other books on the subject. Rather than just describing what the RFCs say the protocol suite should do, this unique book uses a popular diagnostic tool so you may actually watch the protocols in action.

By forcing various conditions to occur—such as connection establishment, timeout and retransmission, and fragmentation—and then displaying the results, *TCP/IP Illustrated* gives you a much greater understanding of these concepts than words alone could provide. Whether you are new to TCP/IP or you have read other books on the subject, you will come away with an increased understanding of how and why TCP/IP works the way it does, as well as enhanced skill at developing applications that run over TCP/IP.

With this unique approach, *TCP/IP Illustrated* presents the structure and function of TCP/IP from the link layer up through the network, transport, and application layers. You will learn about the protocols that belong to each of these layers and how they operate under numerous implementations, including SunOS™ 4.1.3, Solaris® 2.2, UNIX® System V Release 4, BSD/386™, AIX® 3.2.2, and 4.4BSD.

In *TCP/IP Illustrated* you will find the most thorough coverage of TCP available–8 entire chapters. You will also find coverage of the newest TCP/IP features, including multicasting, path MTU discovery, and long fat pipes.

W. Richard Stevens is the highly-respected author of three other bestselling books, *TCP/IP Illustrated, Volume 2*—with Gary R. Wright (Addison-Wesley, 1995), *Advanced Programming in the UNIX Environment* (Addison-Wesley, 1992), and *UNIX Network Programming* (Prentice-Hall, 1990). He is also a popular tutorials instructor and consultant.

Cover illustration by C. Shane Sykes
♻ Text printed on recycled paper
Corporate & Professional Publishing Group
Addison-Wesley Publishing Company