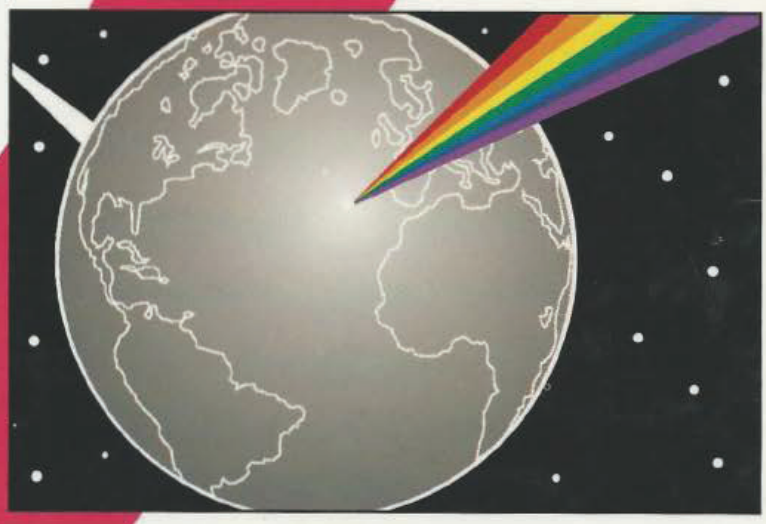
TCP/IP Illustrated, Volume 1

The Protocols

W. Richard Stevens



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

APPLE-1020

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

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

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

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

Copyright © 1994 by Addison-Wesley Publishing Company

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

ISBN 0-201-63346-9
Text printed on recycled paper.
2 3 4 5 6 7 8 9 CRW 97969594
Second Printing, February 1994

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.

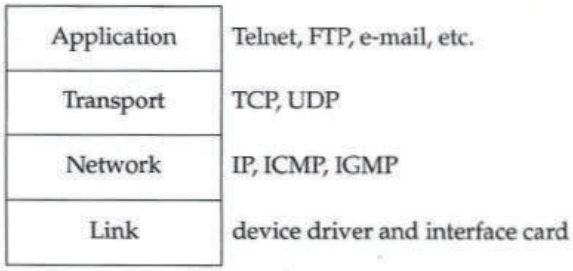


Figure 1.1 The four layers of the TCP/IP protocol suite.

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.

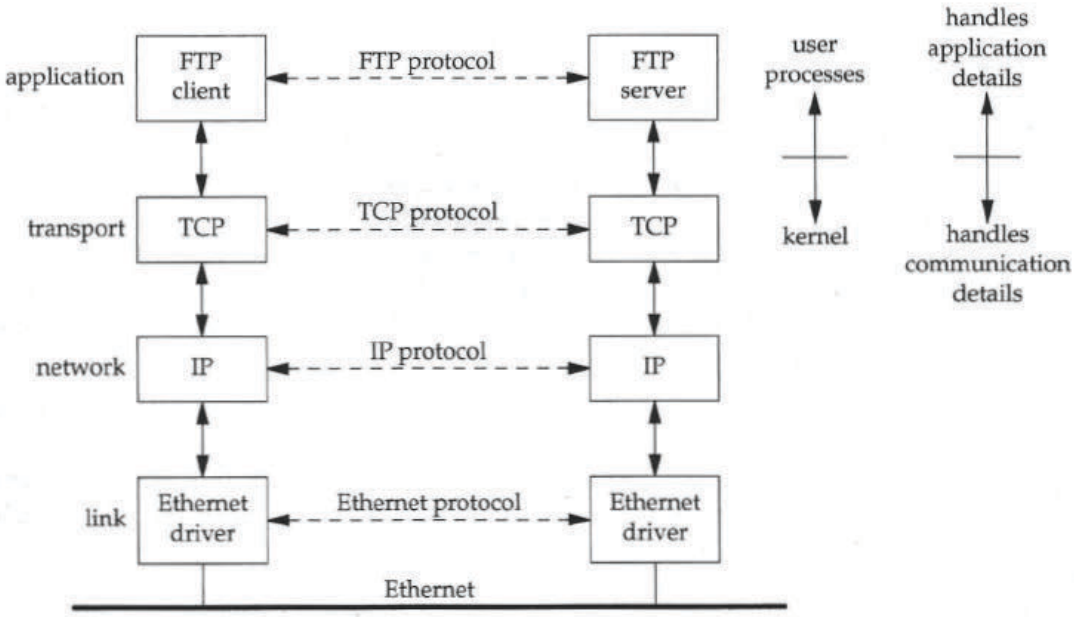


Figure 1.2 Two hosts on a LAN running FTP.

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.

3

IP: Internet Protocol

3.1 Introduction

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

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

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

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

3.2 IP Header

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

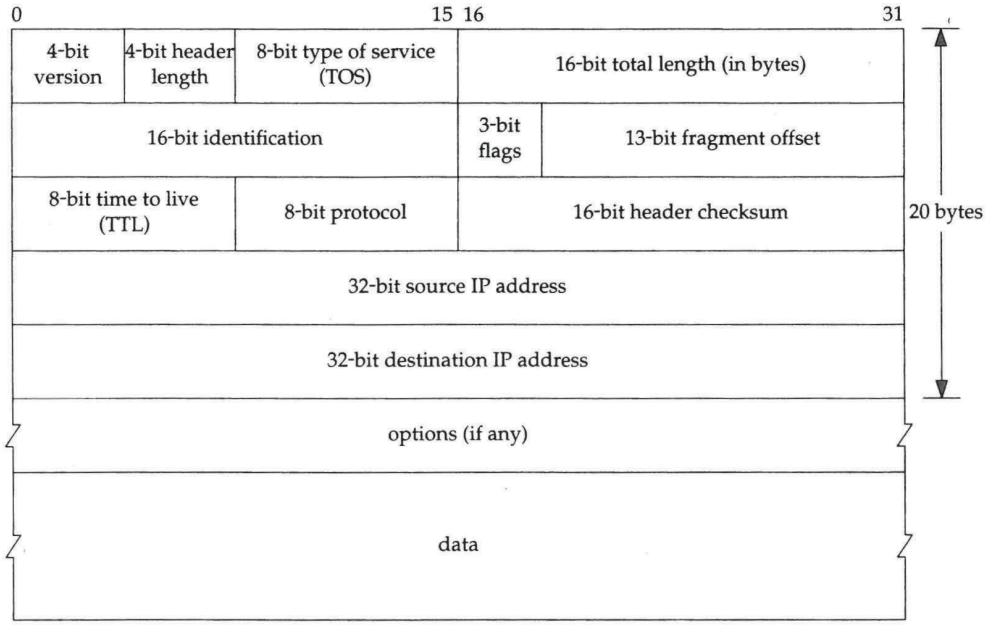


Figure 3.1 IP datagram, showing the fields in the IP header.

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

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

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

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

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

3.7 A Subnet Example

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

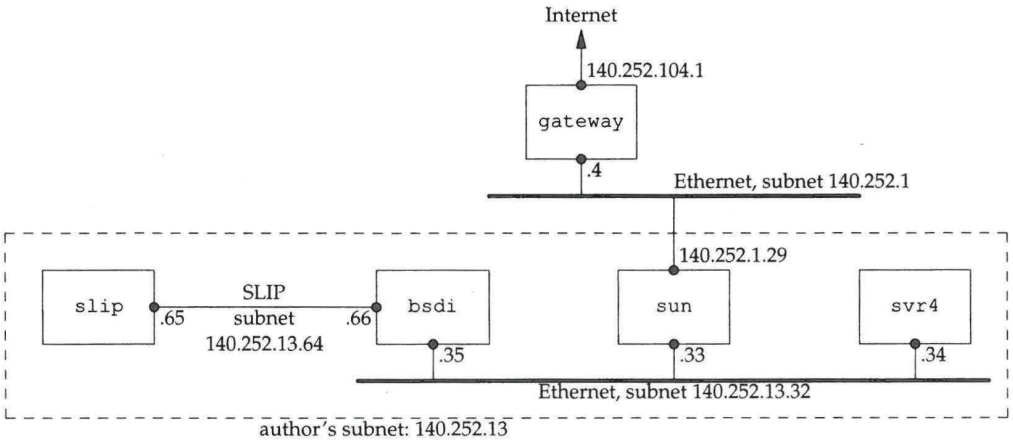


Figure 3.10 Arrangement of hosts and networks for author's subnet.

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

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

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

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

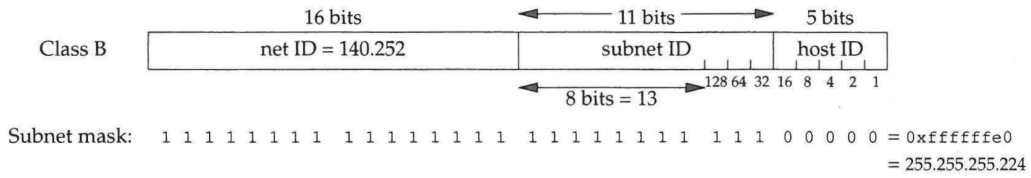


Figure 3.11 Using variable-length subnets.

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

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

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

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

Figure 3.12 IP addresses on author's subnet.

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

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

3.8 `ifconfig` Command

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

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

Figure 6.3 ICMP message types.

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

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

7

Ping Program

7.1 Introduction

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

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

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

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

7.2 Ping Program

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

11

UDP: User Datagram Protocol

11.1 Introduction

UDP is a simple, datagram-oriented, transport layer protocol: each output operation by a process produces exactly one UDP datagram, which causes one IP datagram to be sent. This is different from a stream-oriented protocol such as TCP where the amount of data written by an application may have little relationship to what actually gets sent in an IP datagram.

Figure 11.1 shows the encapsulation of a UDP datagram as an IP datagram.

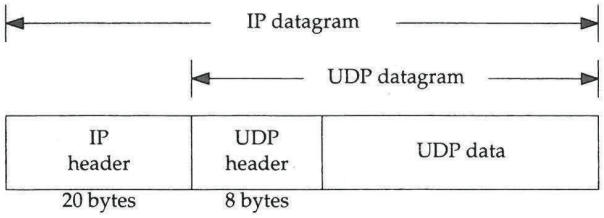


Figure 11.1 UDP encapsulation.

RFC 768 [Postel 1980] is the official specification of UDP.

UDP provides no reliability: it sends the datagrams that the application writes to the IP layer, but there is no guarantee that they ever reach their destination. Given this lack of reliability, we are tempted to think we should avoid UDP and always use a reliable protocol such as TCP. After we describe TCP in Chapter 17 we'll return to this topic and see what types of applications can utilize UDP.

The application needs to worry about the size of the resulting IP datagram. If it exceeds the network's MTU (Section 2.8), the IP datagram is fragmented. This applies to each network that the datagram traverses from the source to the destination, not just the first network connected to the sending host. (We defined this as the *path MTU* in Section 2.9.) We examine IP fragmentation in Section 11.5.

11.2 UDP Header

Figure 11.2 shows the fields in the UDP header.

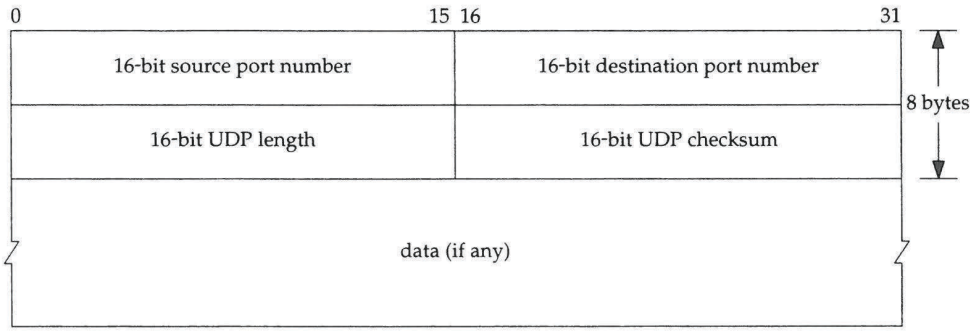


Figure 11.2 UDP header.

The *port numbers* identify the sending process and the receiving process. In Figure 1.8 we showed that TCP and UDP use the destination port number to demultiplex incoming data from IP. Since IP has already demultiplexed the incoming IP datagram to either TCP or UDP (based on the protocol value in the IP header), this means the TCP port numbers are looked at by TCP, and the UDP port numbers by UDP. The TCP port numbers are independent of the UDP port numbers.

> Despite this independence, if a well-known service is provided by both TCP and UDP, the port number is normally chosen to be the same for both transport layers. This is purely for convenience and is not required by the protocols.

The *UDP length* field is the length of the UDP header and the UDP data in bytes. The minimum value for this field is 8 bytes. (Sending a UDP datagram with 0 bytes of data is OK.) This UDP length is redundant. The IP datagram contains its total length in bytes (Figure 3.1), so the length of the UDP datagram is this total length minus the length of the IP header (which is specified by the header length field in Figure 3.1).

11.3 UDP Checksum

The *UDP checksum* covers the UDP header and the UDP data. Recall that the checksum in the IP header only covers the IP header—it does not cover any data in the IP

17

TCP: Transmission Control Protocol

17.1 Introduction

In this chapter we provide a description of the services provided by TCP for the application layer. We also look at the fields in the TCP header. In the chapters that follow we examine all of these header fields in more detail, as we see how TCP operates.

Our description of TCP starts in this chapter and continues in the next seven chapters. Chapter 18 describes how a TCP connection is established and terminated, and Chapters 19 and 20 look at the normal transfer of data, both for interactive use (remote login) and bulk data (file transfer). Chapter 21 provides the details of TCP's timeout and retransmission, followed by two other TCP timers in Chapters 22 and 23. Finally Chapter 24 takes a look at newer TCP features and TCP performance.

The original specification for TCP is RFC 793 [Postel 1981c], although some errors in that RFC are corrected in the Host Requirements RFC.

17.2 TCP Services

Even though TCP and UDP use the same network layer (IP), TCP provides a totally different service to the application layer than UDP does. TCP provides a connection-oriented, reliable, byte stream service.

The term *connection-oriented* means the two applications using TCP (normally considered a client and a server) must establish a TCP connection with each other before they can exchange data. The typical analogy is dialing a telephone number, waiting for the other party to answer the phone and say "hello," and then saying who's calling. In Chapter 18 we look at how a connection is established, and disconnected some time later when either end is done.

There are exactly two end points communicating with each other on a TCP connection. Concepts that we talked about in Chapter 12, broadcasting and multicasting, aren't applicable to TCP.

TCP provides *reliability* by doing the following:

- The application data is broken into what TCP considers the best sized chunks to send. This is totally different from UDP, where each write by the application generates a UDP datagram of that size. The unit of information passed by TCP to IP is called a *segment*. (See Figure 1.7, p. 10.) In Section 18.4 we'll see how TCP decides what this segment size is.
- When TCP sends a segment it maintains a timer, waiting for the other end to acknowledge reception of the segment. If an acknowledgment isn't received in time, the segment is retransmitted. In Chapter 21 we'll look at TCP's adaptive timeout and retransmission strategy.
- When TCP receives data from the other end of the connection, it sends an acknowledgment. This acknowledgment is not sent immediately, but normally delayed a fraction of a second, as we discuss in Section 19.3.
- TCP maintains a checksum on its header and data. This is an end-to-end checksum whose purpose is to detect any modification of the data in transit. If a segment arrives with an invalid checksum, TCP discards it and doesn't acknowledge receiving it. (It expects the sender to time out and retransmit.)
- Since TCP segments are transmitted as IP datagrams, and since IP datagrams can arrive out of order, TCP segments can arrive out of order. A receiving TCP resequences the data if necessary, passing the received data in the correct order to the application.
- Since IP datagrams can get duplicated, a receiving TCP must discard duplicate data.
- TCP also provides flow control. Each end of a TCP connection has a finite amount of buffer space. A receiving TCP only allows the other end to send as much data as the receiver has buffers for. This prevents a fast host from taking all the buffers on a slower host.

A stream of 8-bit bytes is exchanged across the TCP connection between the two applications. There are no record markers automatically inserted by TCP. This is what we called a *byte stream service*. If the application on one end writes 10 bytes, followed by a write of 20 bytes, followed by a write of 50 bytes, the application at the other end of the connection cannot tell what size the individual writes were. The other end may read the 80 bytes in four reads of 20 bytes at a time. One end puts a stream of bytes into TCP and the same, identical stream of bytes appears at the other end.

Also, TCP does not interpret the contents of the bytes at all. TCP has no idea if the data bytes being exchanged are binary data, ASCII characters, EBCDIC characters, or whatever. The interpretation of this byte stream is up to the applications on each end of the connection.

This treatment of the byte stream by TCP is similar to the treatment of a file by the Unix operating system. The Unix kernel does no interpretation whatsoever of the bytes that an application reads or write—that is up to the applications. There is no distinction to the Unix kernel between a binary file or a file containing lines of text.

17.3 TCP Header

Recall that TCP data is encapsulated in an IP datagram, as shown in Figure 17.1.

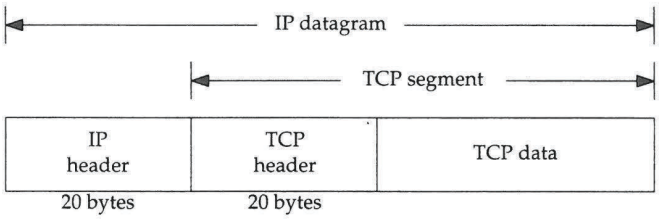


Figure 17.1 Encapsulation of TCP data in an IP datagram.

Figure 17.2 shows the format of the TCP header. Its normal size is 20 bytes, unless options are present.

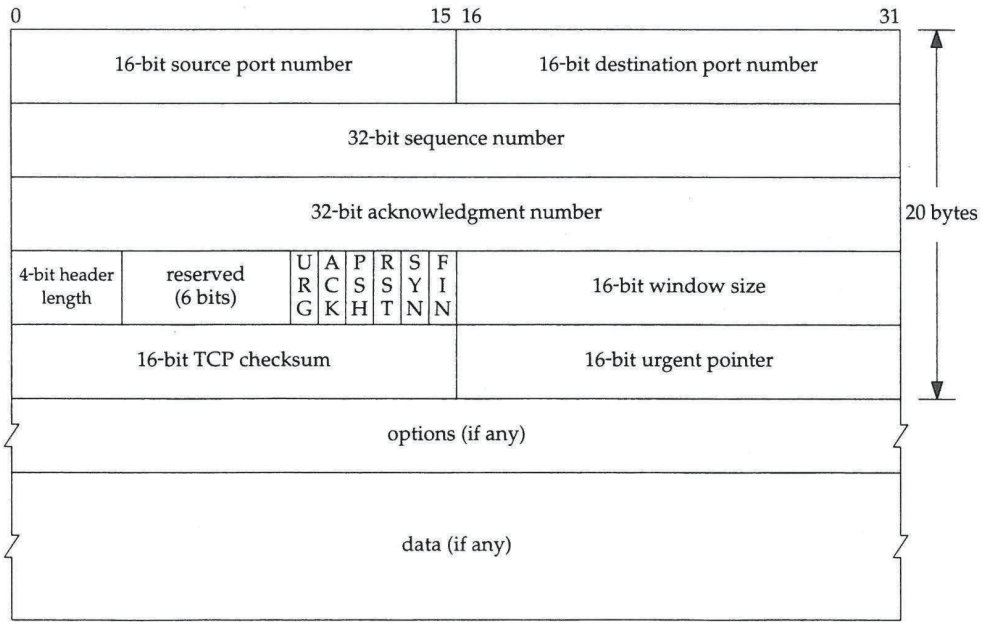


Figure 17.2 TCP header.

Each TCP segment contains the source and destination *port number* to identify the sending and receiving application. These two values, along with the source and destination IP addresses in the IP header, uniquely identify each *connection*.

The combination of an IP address and a port number is sometimes called a *socket*. This term appeared in the original TCP specification (RFC 793), and later it also became used as the name of the Berkeley-derived programming interface (Section 1.15). It is the *socket pair* (the 4-tuple consisting of the client IP address, client port number, server IP address, and server port number) that specifies the two end points that uniquely identifies each TCP connection in an internet.

The *sequence number* identifies the byte in the stream of data from the sending TCP to the receiving TCP that the first byte of data in this segment represents. If we consider the stream of bytes flowing in one direction between two applications, TCP numbers each byte with a sequence number. This sequence number is a 32-bit unsigned number that wraps back around to 0 after reaching $2^{32} - 1$.

When a new connection is being established, the SYN flag is turned on. The *sequence number field* contains the *initial sequence number* (ISN) chosen by this host for this connection. The sequence number of the first byte of data sent by this host will be the ISN plus one because the SYN flag consumes a sequence number. (We describe additional details on exactly how a connection is established and terminated in the next chapter where we'll see that the FIN flag consumes a sequence number also.)

Since every byte that is exchanged is numbered, the *acknowledgment number* contains the next sequence number that the sender of the acknowledgment expects to receive. This is therefore the sequence number plus 1 of the last successfully received byte of data. This field is valid only if the ACK flag (described below) is on.

Sending an ACK costs nothing because the 32-bit acknowledgment number field is always part of the header, as is the ACK flag. Therefore we'll see that once a connection is established, this field is always set and the ACK flag is always on.

TCP provides a *full-duplex* service to the application layer. This means that data can be flowing in each direction, independent of the other direction. Therefore, each end of a connection must maintain a sequence number of the data flowing in each direction.

TCP can be described as a sliding-window protocol without selective or negative acknowledgments. (The sliding window protocol used for data transmission is described in Section 20.3.) We say that TCP lacks selective acknowledgments because the acknowledgment number in the TCP header means that the sender has successfully received up through but not including that byte. There is currently no way to acknowledge selected pieces of the data stream. For example, if bytes 1–1024 are received OK, and the next segment contains bytes 2049–3072, the receiver cannot acknowledge this new segment. All it can send is an ACK with 1025 as the acknowledgment number. There is no means for negatively acknowledging a segment. For example, if the segment with bytes 1025–2048 did arrive, but had a checksum error, all the receiving TCP can send is an ACK with 1025 as the acknowledgment number. In Section 21.7 we'll see how duplicate acknowledgments can help determine that packets have been lost.

The *header length* gives the length of the header in 32-bit words. This is required because the length of the options field is variable. With a 4-bit field, TCP is limited to a 60-byte header. Without options, however, the normal size is 20 bytes.

There are six flag bits in the TCP header. One or more of them can be turned on at the same time. We briefly mention their use here and discuss each flag in more detail in later chapters.

URG The *urgent pointer* is valid (Section 20.8).

ACK The *acknowledgment number* is valid.

PSH The receiver should pass this data to the application as soon as possible (Section 20.5).

RST Reset the connection (Section 18.7).

SYN Synchronize sequence numbers to initiate a connection. This flag and the next are described in Chapter 18.

FIN The sender is finished sending data.

TCP's flow control is provided by each end advertising a *window size*. This is the number of bytes, starting with the one specified by the acknowledgment number field, that the receiver is willing to accept. This is a 16-bit field, limiting the window to 65535 bytes. In Section 24.4 we'll look at the new window scale option that allows this value to be scaled, providing larger windows.

The *checksum* covers the TCP segment: the TCP header and the TCP data. This is a mandatory field that must be calculated and stored by the sender, and then verified by the receiver. The TCP checksum is calculated similar to the UDP checksum, using a pseudo-header as described in Section 11.3.

The *urgent pointer* is valid only if the URG flag is set. This pointer is a positive offset that must be added to the sequence number field of the segment to yield the sequence number of the last byte of urgent data. TCP's urgent mode is a way for the sender to transmit emergency data to the other end. We'll look at this feature in Section 20.8.

The most common *option* field is the maximum segment size option, called the *MSS*. Each end of a connection normally specifies this option on the first segment exchanged (the one with the SYN flag set to establish the connection). It specifies the maximum sized segment that the sender wants to receive. We describe the MSS option in more detail in Section 18.4, and some of the other TCP options in Chapter 24.

In Figure 17.2 we note that the data portion of the TCP segment is optional. We'll see in Chapter 18 that when a connection is established, and when a connection is terminated, segments are exchanged that contain only the TCP header with possible options. A header without any data is also used to acknowledge received data, if there is no data to be transmitted in that direction. There are also some cases dealing with timeouts when a segment can be sent without any data.

17.4 Summary

TCP provides a reliable, connection-oriented, byte stream, transport layer service. We looked briefly at all the fields in the TCP header and will examine them in detail in the following chapters.

In line 1, the field `1415531521:1415531521(0)` means the sequence number of the packet was 1415531521 and the number of data bytes in the segment was 0. `tcpdump` displays this by printing the starting sequence number, a colon, the implied ending sequence number, and the number of data bytes in parentheses. The advantage of displaying both the sequence number and the implied ending sequence number is to see what the implied ending sequence number is, when the number of bytes is greater than 0. This field is output only if (1) the segment contains one or more bytes of data or (2) the SYN, FIN, or RST flag was on. Lines 1, 2, 4, and 6 in Figure 18.1 display this field because of the flag bits—we never exchange any data in this example.

In line 2 the field `ack 1415531522` shows the acknowledgment number. This is printed only if the ACK flag in the header is on.

The field `win 4096` in every line of output shows the window size being advertised by the sender. In these examples, where we are not exchanging any data, the window size never changes from its default of 4096. (We examine TCP's window size in Section 20.4.)

The final field that is output in Figure 18.1, `<mss 1024>` shows the *maximum segment size* (MSS) option specified by the sender. The sender does not want to receive TCP segments larger than this value. This is normally to avoid fragmentation (Section 11.5). We discuss the maximum segment size in Section 18.4, and show the format of the various TCP options in Section 18.10.

Time Line

Figure 18.3 shows the time line for this sequence of packets. (We described some general features of these time lines when we showed the first one in Figure 6.11, p. 80.) This figure shows which end is sending packets. We also expand some of the `tcpdump` output (e.g., printing SYN instead of `S`). In this time line we have also removed the window size values, since they add nothing to the discussion.

Connection Establishment Protocol

Now let's return to the details of the TCP protocol that are shown in Figure 18.3. To establish a TCP connection:

1. The requesting end (normally called the *client*) sends a SYN segment specifying the port number of the *server* that the client wants to connect to, and the client's *initial sequence number* (ISN, 1415531521 in this example). This is segment 1.
2. The server responds with its own SYN segment containing the server's initial sequence number (segment 2). The server also acknowledges the client's SYN by ACKing the client's ISN plus one. A SYN consumes one sequence number.
3. The client must acknowledge this SYN from the server by ACKing the server's ISN plus one (segment 3).

These three segments complete the connection establishment. This is often called the *three-way handshake*.

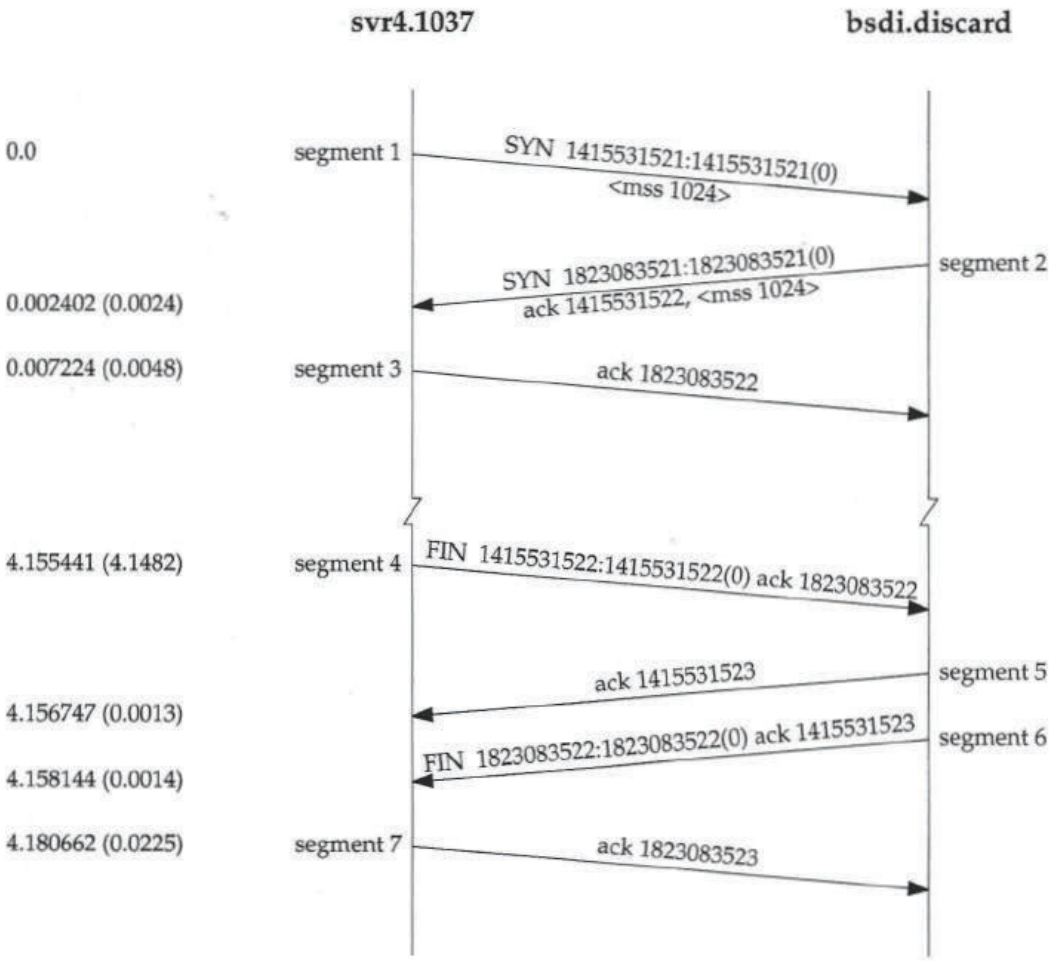


Figure 18.3 Time line of connection establishment and connection termination.

The side that sends the first SYN is said to perform an *active open*. The other side, which receives this SYN and sends the next SYN, performs a *passive open*. (In Section 18.8 we describe a simultaneous open where both sides can do an active open.)

When each end sends its SYN to establish the connection, it chooses an initial sequence number for that connection. The ISN should change over time, so that each connection has a different ISN. RFC 793 [Postel 1981c] specifies that the ISN should be viewed as a 32-bit counter that increments by one every 4 microseconds. The purpose in these sequence numbers is to prevent packets that get delayed in the network from being delivered later and then misinterpreted as part of an existing connection.

> How are the sequence numbers chosen? In 4.4BSD (and most Berkeley-derived implementations) when the system is initialized the initial send sequence number is initialized to 1. This practice violates the Host Requirements RFC. (A comment in the code acknowledges that this is wrong.) This variable is then incremented by 64,000 every half-second, and will cycle back to 0 about every 9.5 hours. (This corresponds to a counter that is incremented every 8

```
bsdi % sock -L0 svr4 8888          this is the client; server shown later
hello, world                       type one line of input that's sent to other end
^D                                 type end-of-file character to terminate client
```

Figure 18.15 shows the `tcpdump` output for this example. (We have deleted all the window advertisements in this figure, since they add nothing to the discussion.)

```
1  0.0                  bsdi.1099 > svr4.8888: S 671112193:671112193(0)
                                                 <mss 1024>
2  0.004975 (0.0050)    svr4.8888 > bsdi.1099: S 3224959489:3224959489(0)
                                                 ack 671112194 <mss 1024>
3  0.006656 (0.0017)    bsdi.1099 > svr4.8888: . ack 1

4  4.833073 (4.8264)    bsdi.1099 > svr4.8888: P 1:14(13) ack 1
5  5.026224 (0.1932)    svr4.8888 > bsdi.1099: . ack 14

6  9.527634 (4.5014)    bsdi.1099 > svr4.8888: R 14:14(0) ack 1
```

Figure 18.15 Aborting a connection with a reset (RST) instead of a FIN.

Lines 1–3 show the normal connection establishment. Line 4 sends the data line that we typed (12 characters plus the Unix newline character), and line 5 is the acknowledgment of the received data.

Line 6 corresponds to our typing the end-of-file character (Control-D) to terminate the client. Since we specified an abort instead of a normal close (the `-L0` command-line option), the TCP on `bsdi` sends an RST instead of the normal FIN. The RST segment contains a sequence number and acknowledgment number. Also notice that the RST segment elicits no response from the other end—it is not acknowledged at all. The receiver of the reset aborts the connection and advises the application that the connection was reset.

We get the following error on the server for this exchange:

```
svr4 % sock -s 8888                run as server, listen on port 8888
hello, world                       this is what the client sent over
read error: Connection reset by peer
```

This server reads from the network and copies whatever it receives to standard output. It normally ends by receiving an end-of-file notification from its TCP, but here we see that it receives an error when the RST arrives. The error is what we expect: the connection was reset by the peer.

Detecting Half-Open Connections

A TCP connection is said to be *half-open* if one end has closed or aborted the connection without the knowledge of the other end. This can happen any time one of the two hosts crashes. As long as there is no attempt to transfer data across a half-open connection, the end that's still up won't detect that the other end has crashed.

Another common cause of a half-open connection is when a client host is powered off, instead of terminating the client application and then shutting down the client host. This happens when PCs are being used to run Telnet clients, for example, and the users

power off the PC at the end of the day. If there was no data transfer going on when the PC was powered off, the server will never know that the client disappeared. When the user comes in the next morning, powers on the PC, and starts a new Telnet client, a new occurrence of the server is started on the server host. This can lead to many half-open TCP connections on the server host. (In Chapter 23 we'll see a way for one end of a TCP connection to discover that the other end has disappeared using TCP's keepalive option.)

We can easily create a half-open connection. We'll execute the Telnet client on `bsdi`, connecting to the discard server on `svr4`. We type one line of input, and watch it go across with `tcpdump`, and then disconnect the Ethernet cable on the server's host, and reboot the server host. This simulates the server host crashing. (We disconnect the Ethernet cable before rebooting the server to prevent it from sending a FIN out the open connections, which some TCPs do when they are shut down.) After the server has rebooted, we reconnect the cable, and try to send another line from the client to the server. Since the server's TCP has rebooted, and lost all memory of the connections that existed before it was rebooted, it knows nothing about the connection that the data segment references. The rule of TCP is that the receiver responds with a reset.

```
bsdi % telnet svr4 discard          start the client
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
hi there                            this line is sent OK
                                    here is where we reboot the server host
another line                        and this one elicits a reset
Connection closed by foreign host.
```

Figure 18.16 shows the `tcpdump` output for this example. (We have removed from this output the window advertisements, the type-of-service information, and the MSS announcements, since they add nothing to the discussion.)

```
1    0.0                 bsdi.1102 > svr4.discard: S 1591752193:1591752193(0)
2    0.004811 (  0.0048) svr4.discard > bsdi.1102: S 26368001:26368001(0)
                                                    ack 1591752194
3    0.006516 (  0.0017) bsdi.1102 > svr4.discard: . ack 1

4    5.167679 (  5.1612) bsdi.1102 > svr4.discard: P 1:11(10) ack 1
5    5.201662 (  0.0340) svr4.discard > bsdi.1102: . ack 11

6  194.909929 (189.7083) bsdi.1102 > svr4.discard: P 11:25(14) ack 1
7  194.914957 (  0.0050) arp who-has bsdi tell svr4
8  194.915678 (  0.0007) arp reply bsdi is-at 0:0:c0:6f:2d:40
9  194.918225 (  0.0025) svr4.discard > bsdi.1102: R 26368002:26368002(0)
```

Figure 18.16 Reset in response to data segment on a half-open connection.

Lines 1–3 are the normal connection establishment. Line 4 sends the line "hi there" to the discard server, and line 5 is the acknowledgment.

At this point we disconnect the Ethernet cable from `svr4`, reboot it, and reconnect the cable. This takes almost 190 seconds. We then type the next line of input to the client ("another line") and when we type the return key the line is sent to the server

Networking

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 author of the bestselling *Advanced Programming in the UNIX Environment* (Addison-Wesley, 1992) and *UNIX Network Programming* (Prentice-Hall, 1990). He currently consults for companies, helping them with UNIX, TCP/IP, and other networking issues.

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

