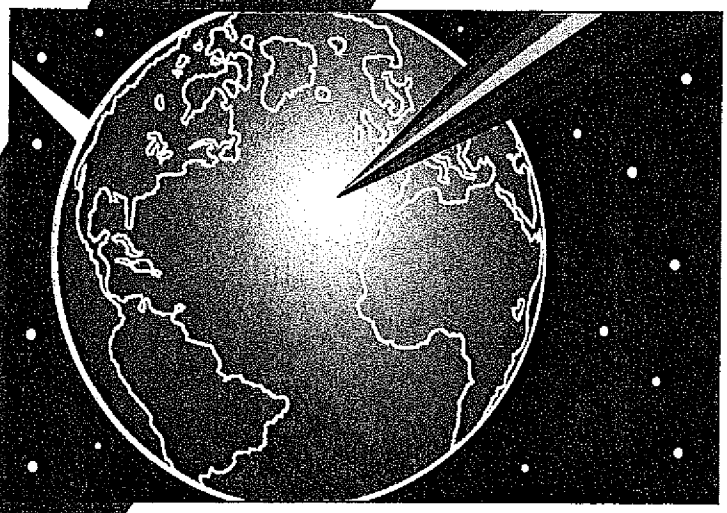
TCP/IP Illustrated Volume

The Protocols

W. Richard Stevens



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

TCP/IP Illustrated, Volume 1

The Protocols

W. Richard Stevens

ADDISON–WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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 0201633469

19 2021222324 MA 04 03 02 01

19th Printing July 2001

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.

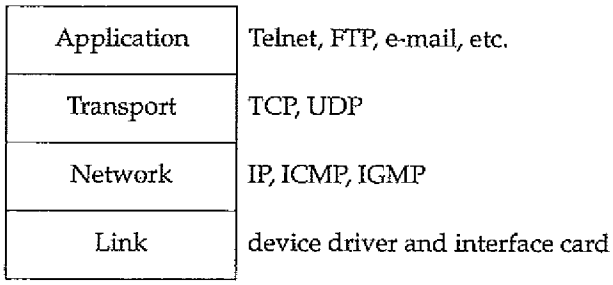


Figure 1.1 The four layers of the TCP/IP protocol suite.

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.

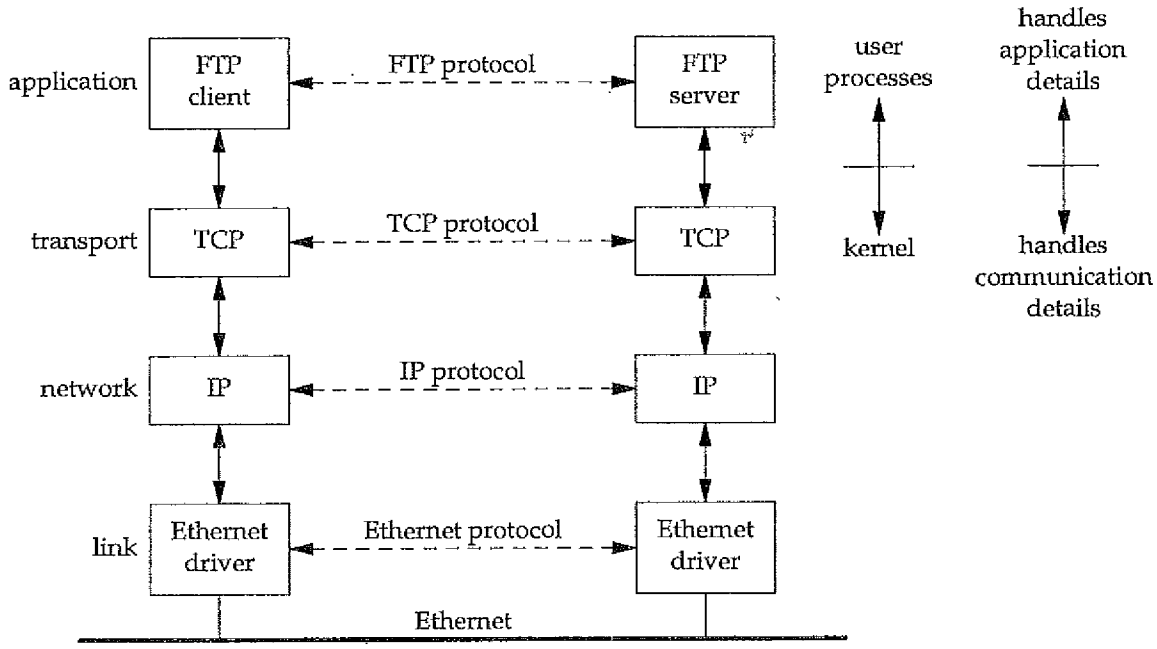


Figure 1.2 Two hosts on a LAN running FTP.

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.

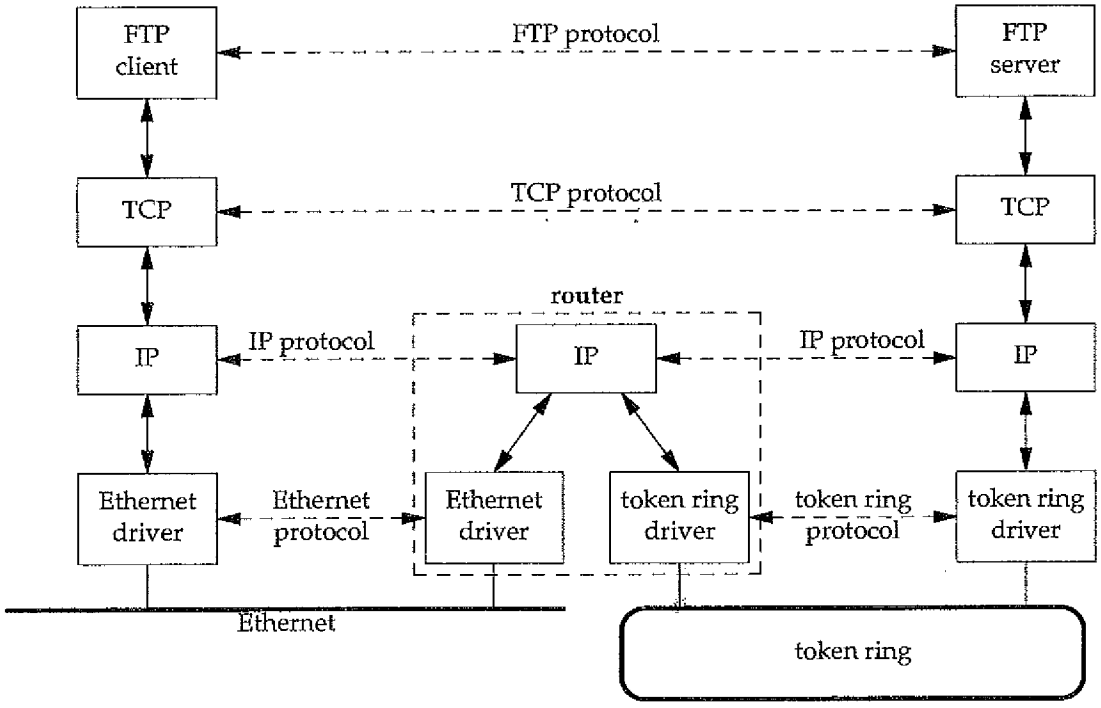


Figure 1.3 Two networks connected with a router.

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.

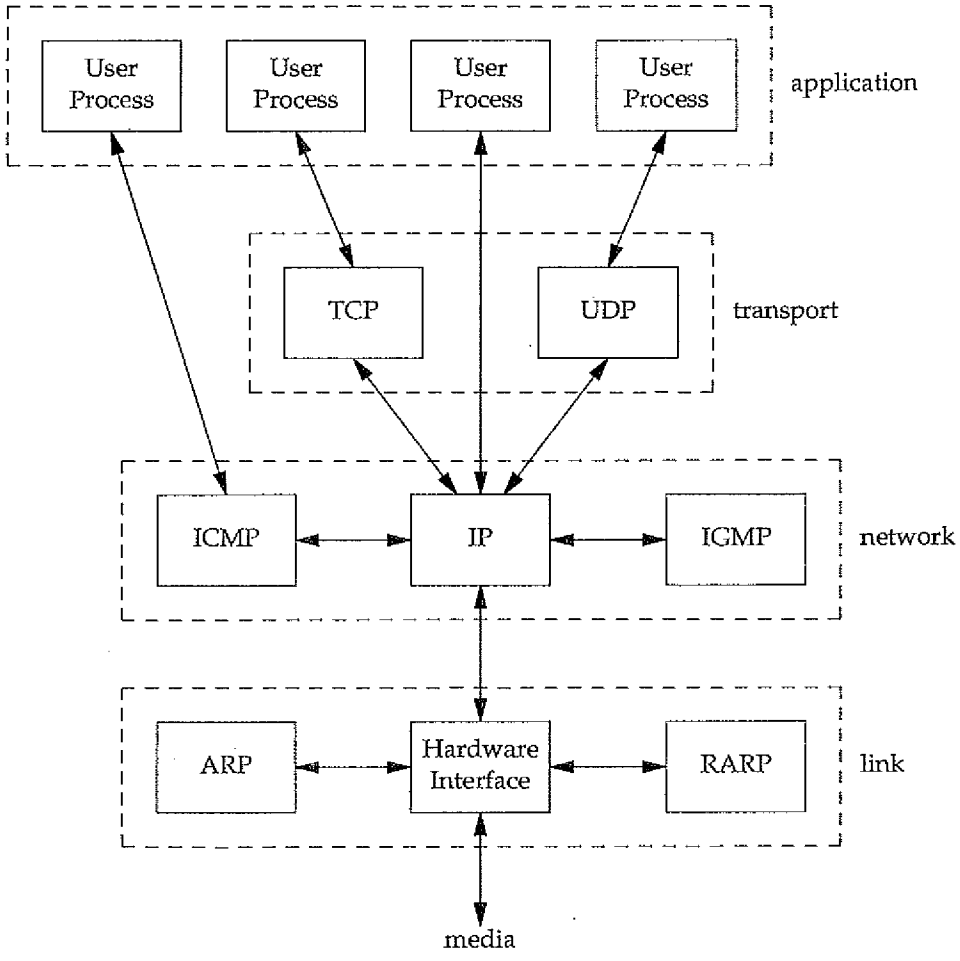


Figure 1.4 Various protocols at the different layers in the TCP/IP protocol suite.

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.

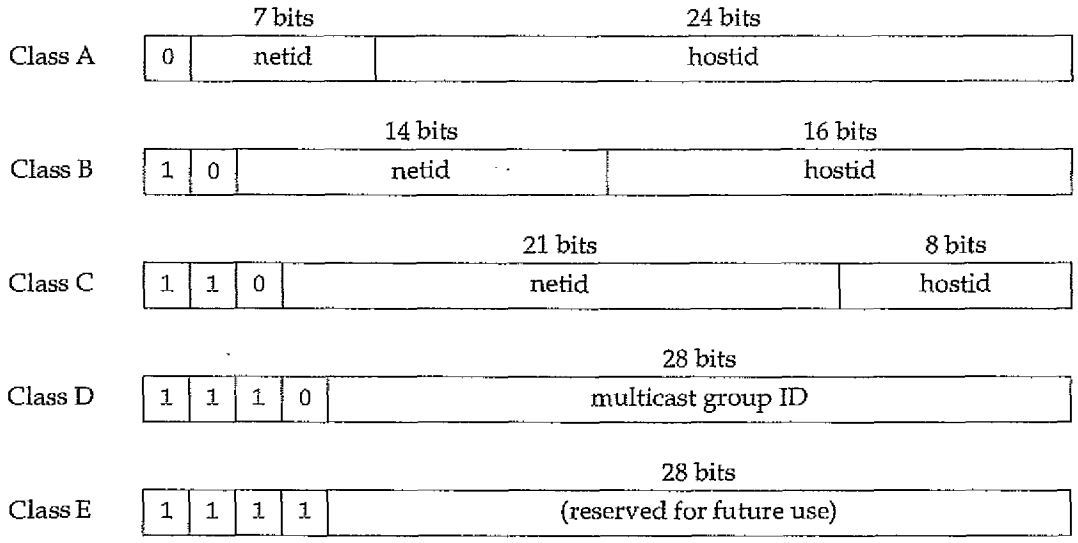


Figure 1.5 The five different classes of Internet addresses.

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 **255**.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.

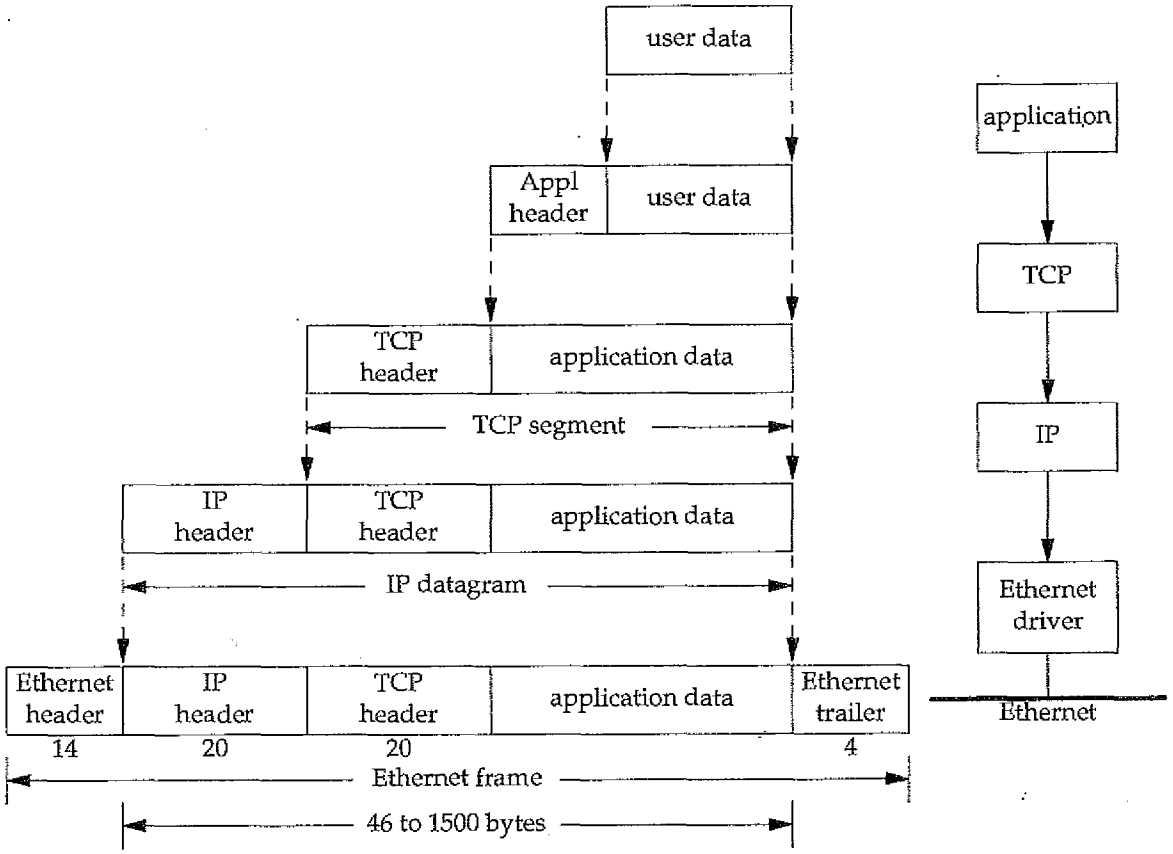


Figure 1.7 Encapsulation of data as it goes down the protocol stack.

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.

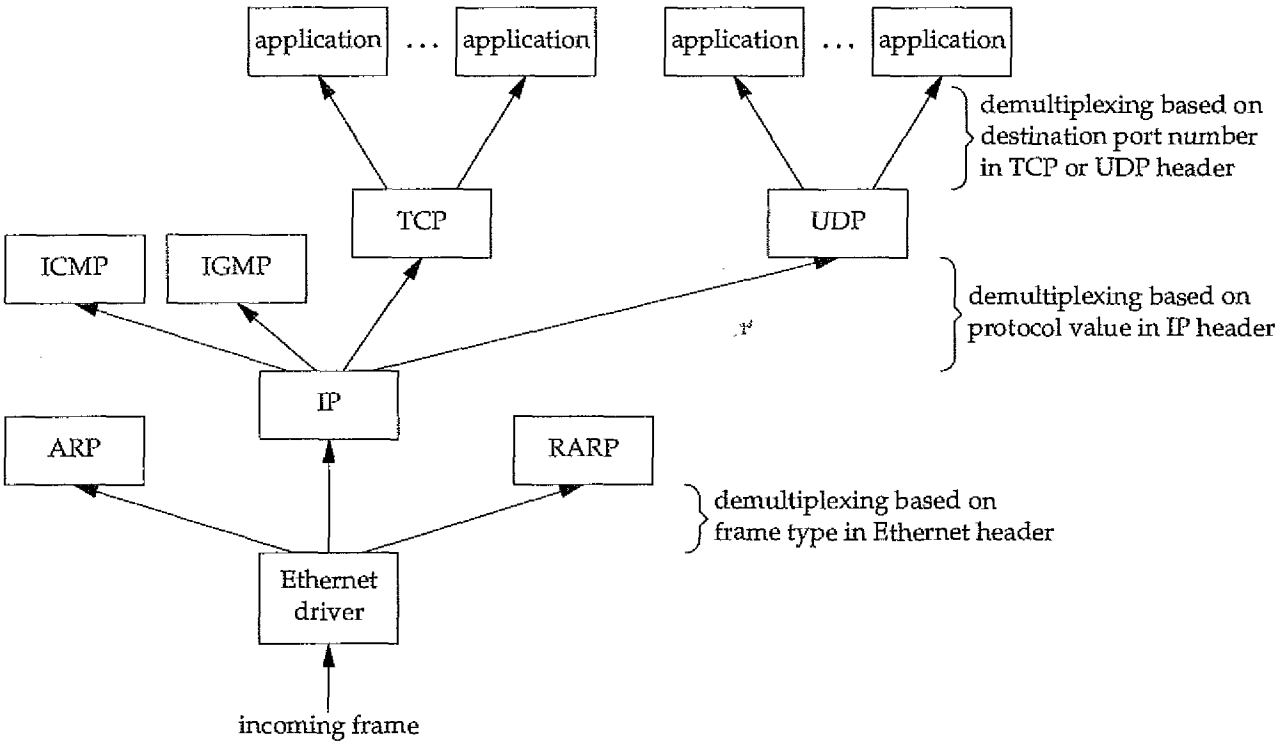


Figure 1.8 The demultiplexing of a received Ethernet frame.

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!

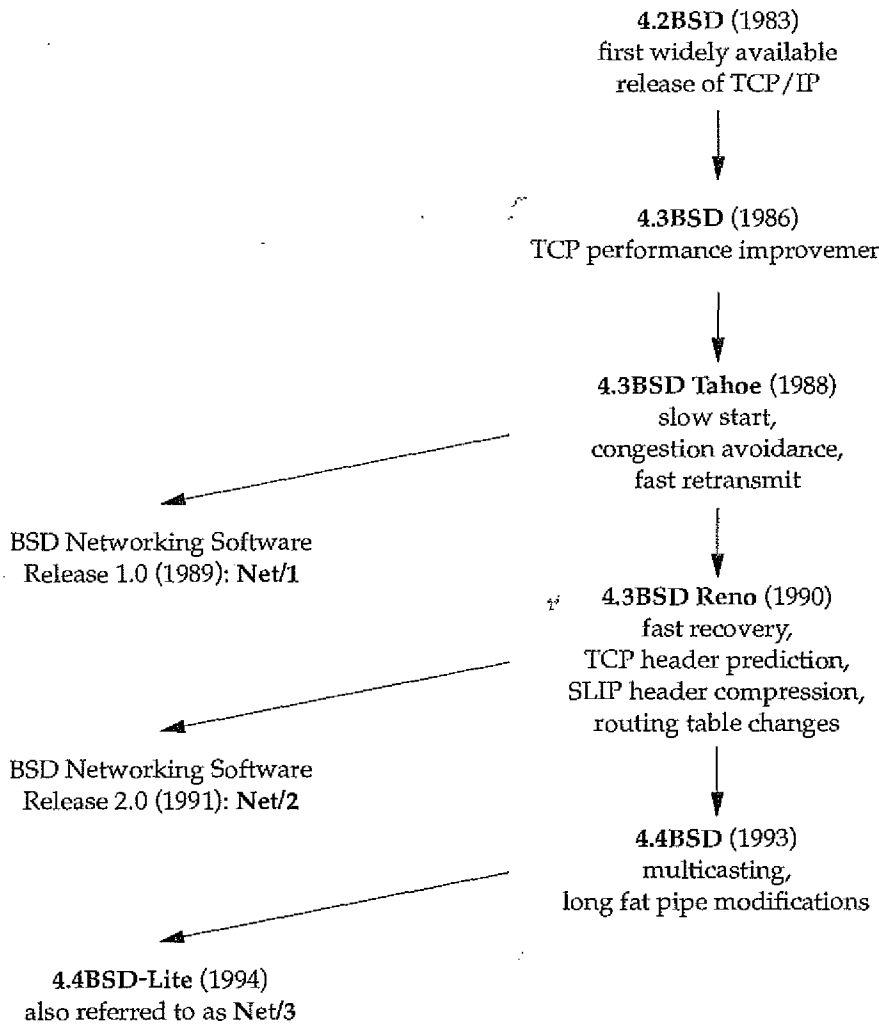


Figure 1.10 Various BSD releases with important TCP/IP features.

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.

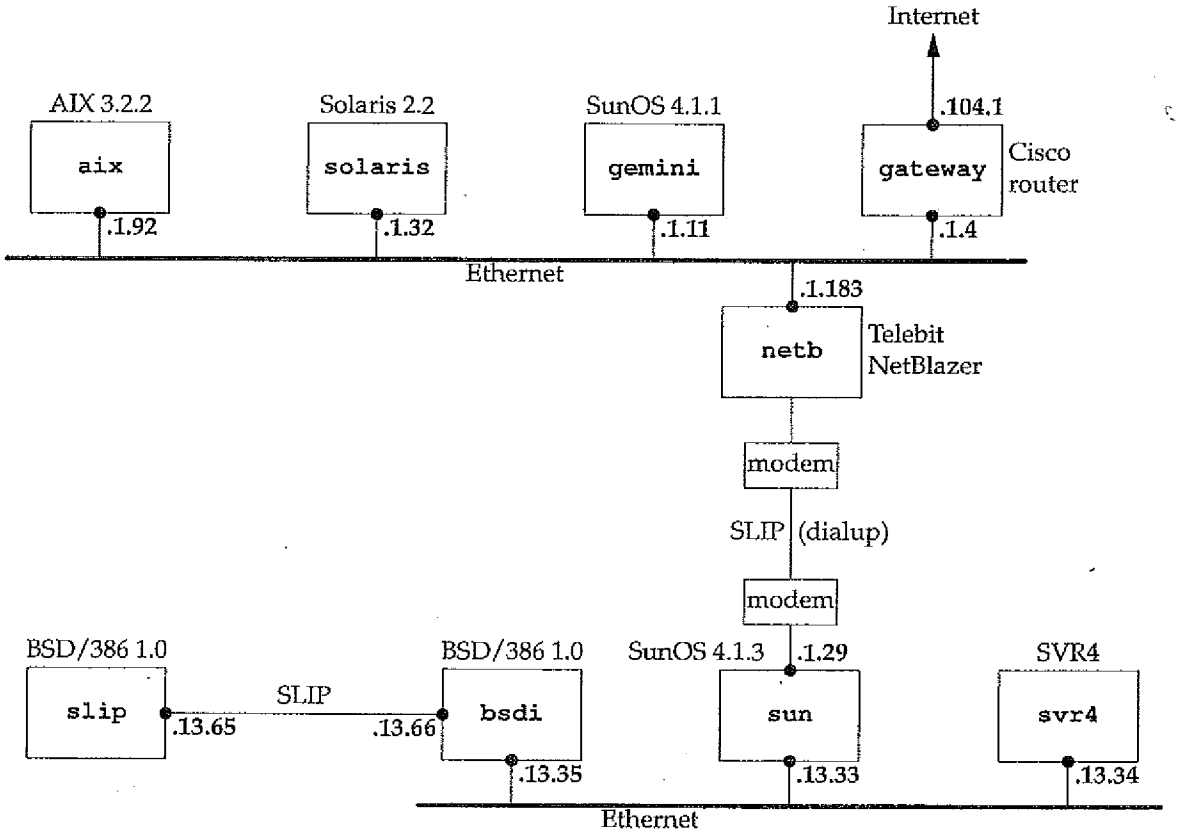


Figure 1.11 Test network used for all the examples in the text. All IP addresses begin with 140.252.

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`.

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.

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

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

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

Exercises

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

3.2 IP Header

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

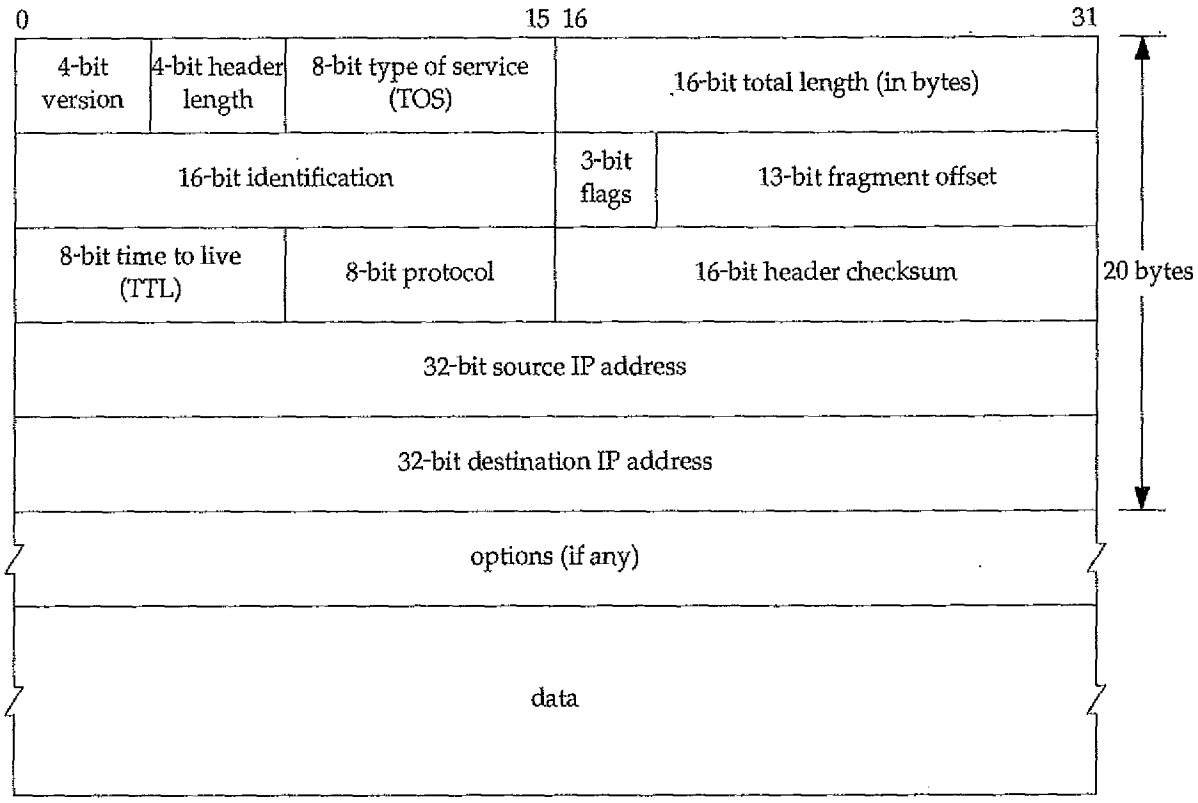


Figure 3.1 IP datagram, showing the fields in the IP header.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 IP Routing

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

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

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

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

Each entry in the routing table contains the following information:

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

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

IP routing performs the following actions:

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

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

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

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

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

Examples

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

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

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

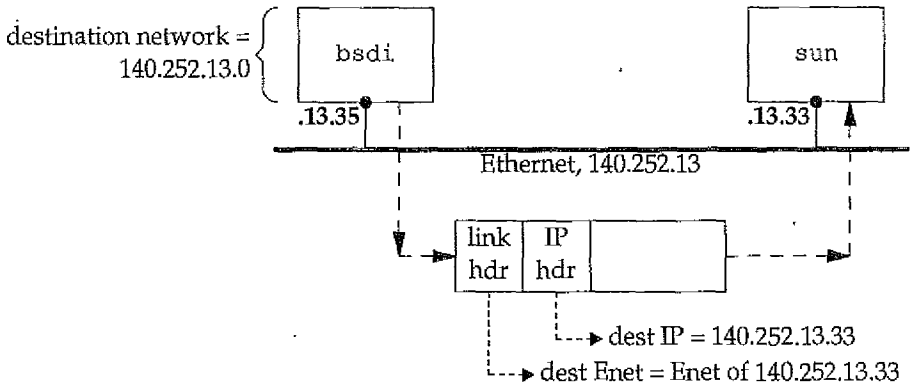


Figure 3.3 Delivery of IP datagram from `bsdi` to `sun`.

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

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

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

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

A few key points come out in this example.

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

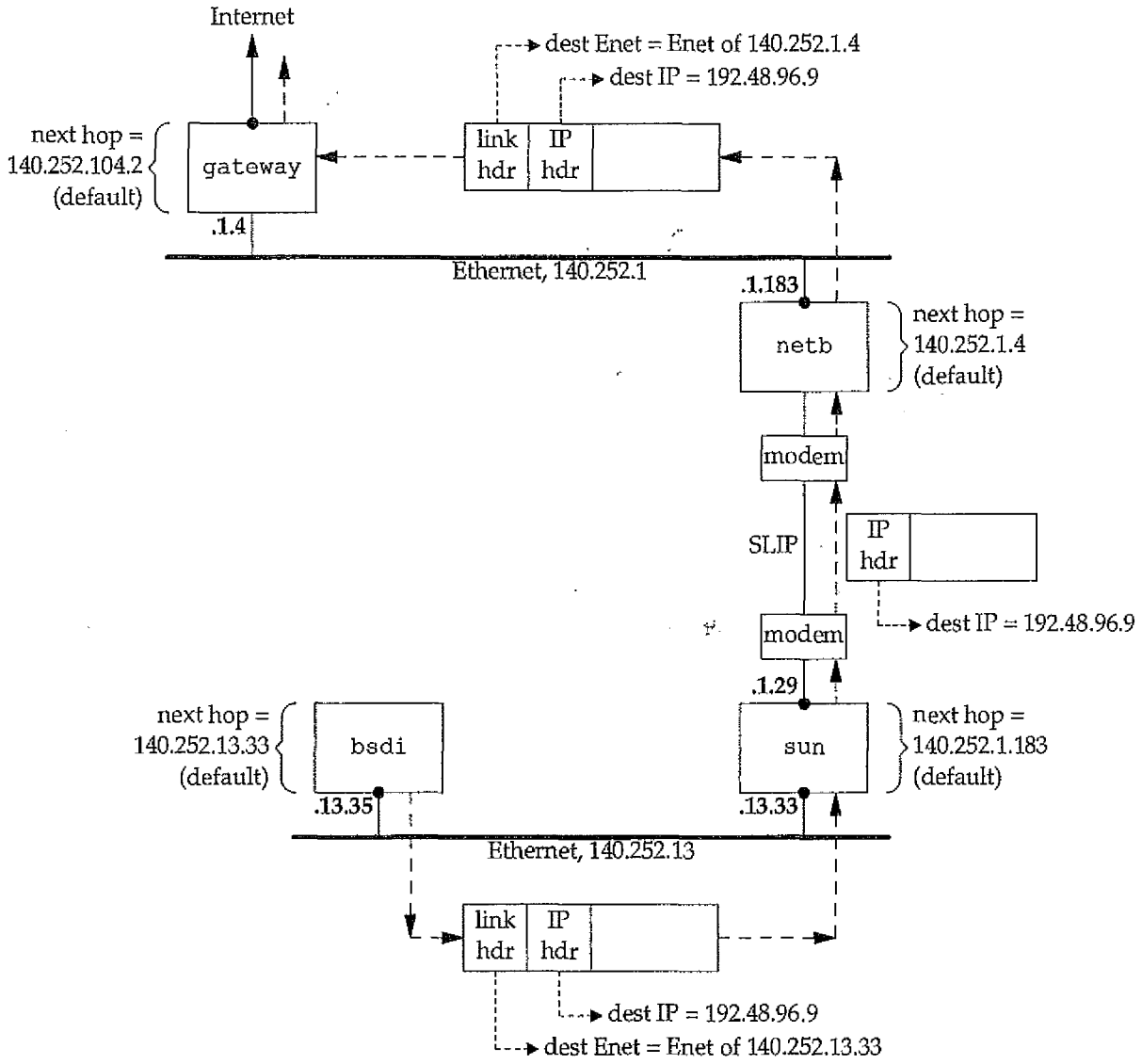


Figure 3.4 Initial path of datagram from `bsdi` to `ftp.uu.net` (192.48.96.9).

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

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

3.4 Subnet Addressing

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

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

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

Figure 3.5 Subnetting a class B address.

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

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

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

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

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

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

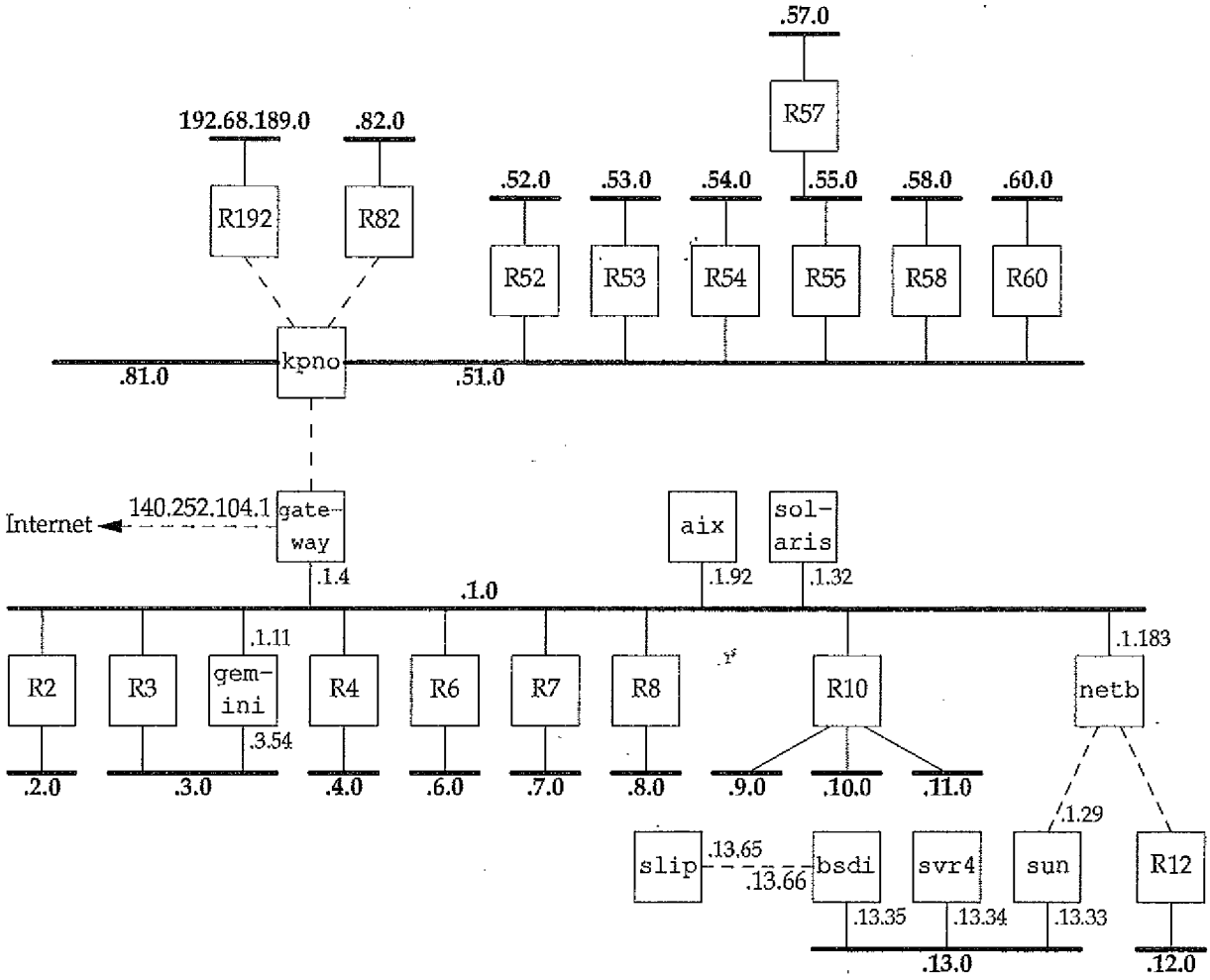


Figure 3.6 Arrangement of most of the noao.edu 140.252 subnets.

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

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

3.5 Subnet Mask

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

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

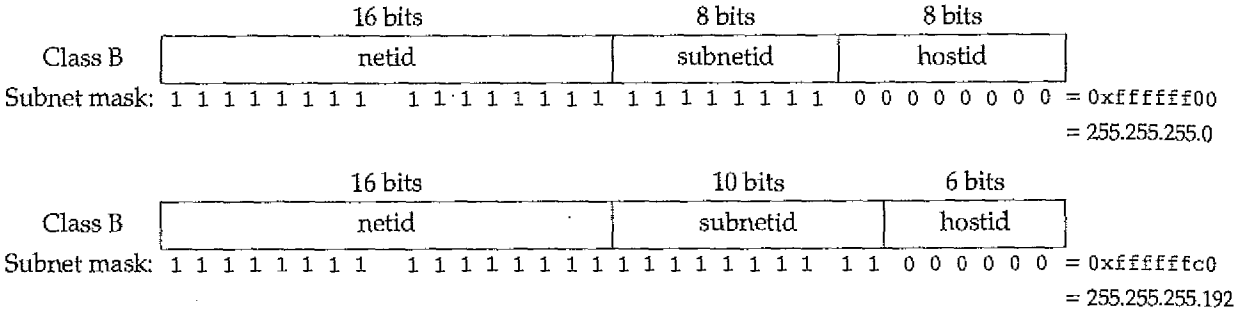


Figure 3.7 Example subnet masks for two different class B subnet arrangements.

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

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

Example

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

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

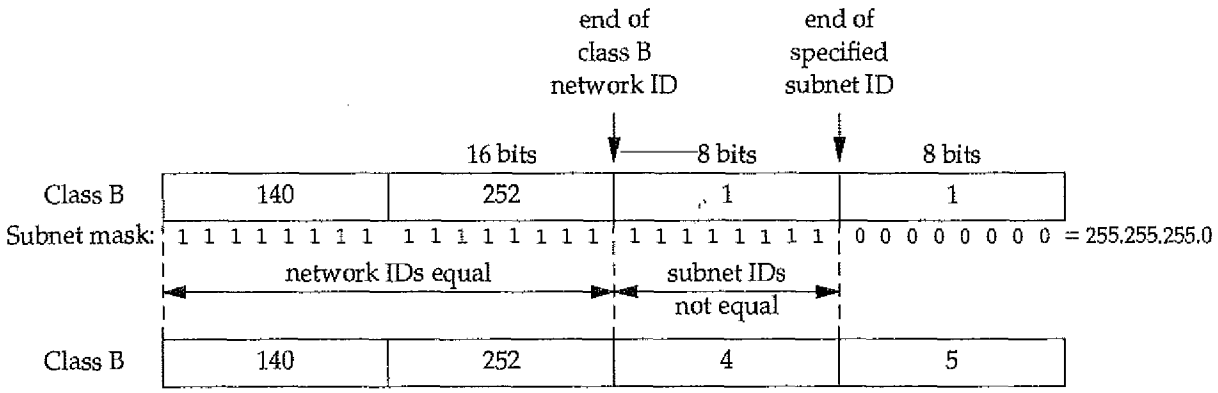


Figure 3.8 Comparison of two class B addresses using a subnet mask.

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

3.6 Special Case IP Addresses

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

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

Figure 3.9 Special case IP addresses.

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

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

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

3.7 A Subnet Example

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

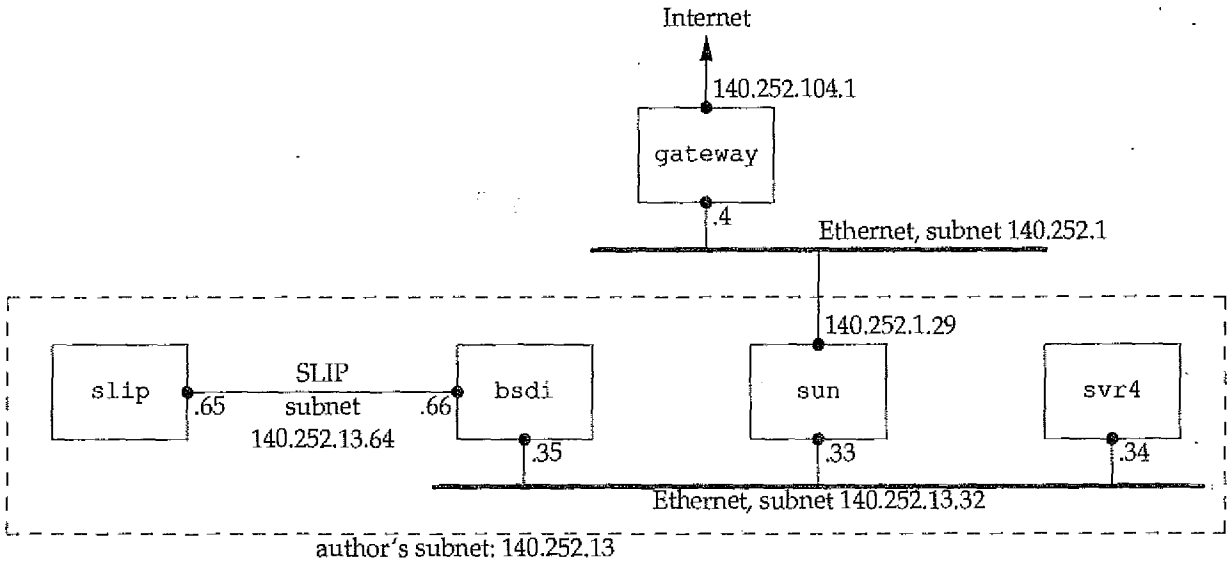


Figure 3.10 Arrangement of hosts and networks for author's subnet.

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

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

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

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

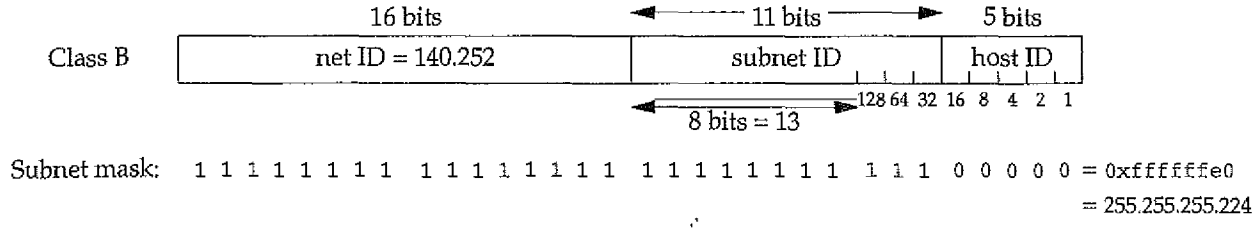


Figure 3.11 Using variable-length subnets.

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

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

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

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

Figure 3.12 IP addresses on author's subnet.

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

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

3.8 `ifconfig` Command

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
svr4 % /usr/sbin/ifconfig emd0
emd0: flags=23<UP,BROADCAST,NOTRAILERS>
        inet 140.252.13.34 netmask fffffffe0 broadcast 140.252.13.63
```

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

3.9 `netstat` Command

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

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

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

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

3.10 IP Futures

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

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

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

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

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

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

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

3.11 Summary

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

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

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

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

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

Exercises

3.1 Must the loopback address be 127.0.0.1?

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

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

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

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

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

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

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 a single IP datagram.

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

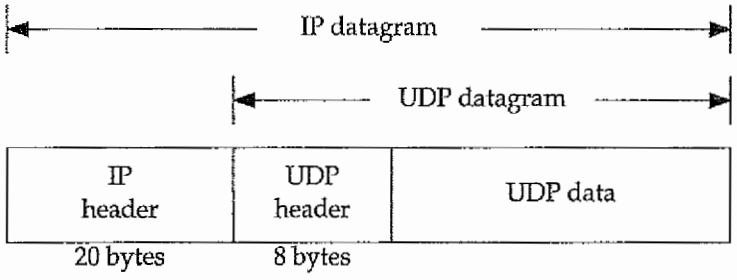


Figure 11.1 UDP encapsulation.

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.

10.3 The OSPF packet format has a checksum field, but the RIP packet does not. Why?

10.4 What effect does load balancing, as done by OSPF, have on a transport layer?

10.5 Read RFC 1058 for additional details on the implementation of RIP. In Figure 10.8 each router advertises only the routes that it provides, and none of the other routes that it learned about through the other router's broadcasts on the 140.252.1 network. What is this technique called?

10.6 In Section 3.4 we said there are more than 100 hosts on the 140.252.1 subnet in addition to the eight routers we show in Figure 10.7. What do these 100 hosts do with the eight broadcasts that arrive every 30 seconds (Figure 10.8)?

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.

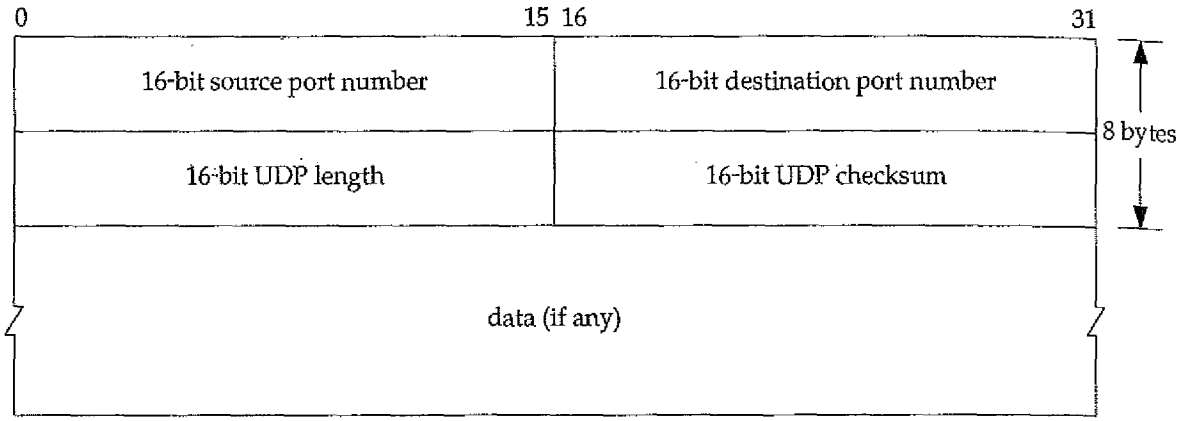


Figure 11.2 UDP header.

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

datagram. Both UDP and TCP have checksums in their headers to cover their header and their data. With UDP the checksum is optional, while with TCP it is mandatory.

Although the basics for calculating the UDP checksum are similar to what we described in Section 3.2 for the IP header checksum (the ones complement sum of 16-bit words), there are differences. First, the length of the UDP datagram can be an odd number of bytes, while the checksum algorithm adds 16-bit words. The solution is to append a pad byte of 0 to the end, if necessary, just for the checksum computation. (That is, this possible pad byte is not transmitted.)

Next, both UDP and TCP include a 12-byte pseudo-header with the UDP datagram (or TCP segment) just for the checksum computation. This pseudo-header includes certain fields from the IP header. The purpose is to let UDP double-check that the data has arrived at the correct destination (i.e., that IP has not accepted a datagram that is not addressed to this host, and that IP has not given UDP a datagram that is for another upper layer). Figure 11.3 shows the pseudo-header along with a UDP datagram.

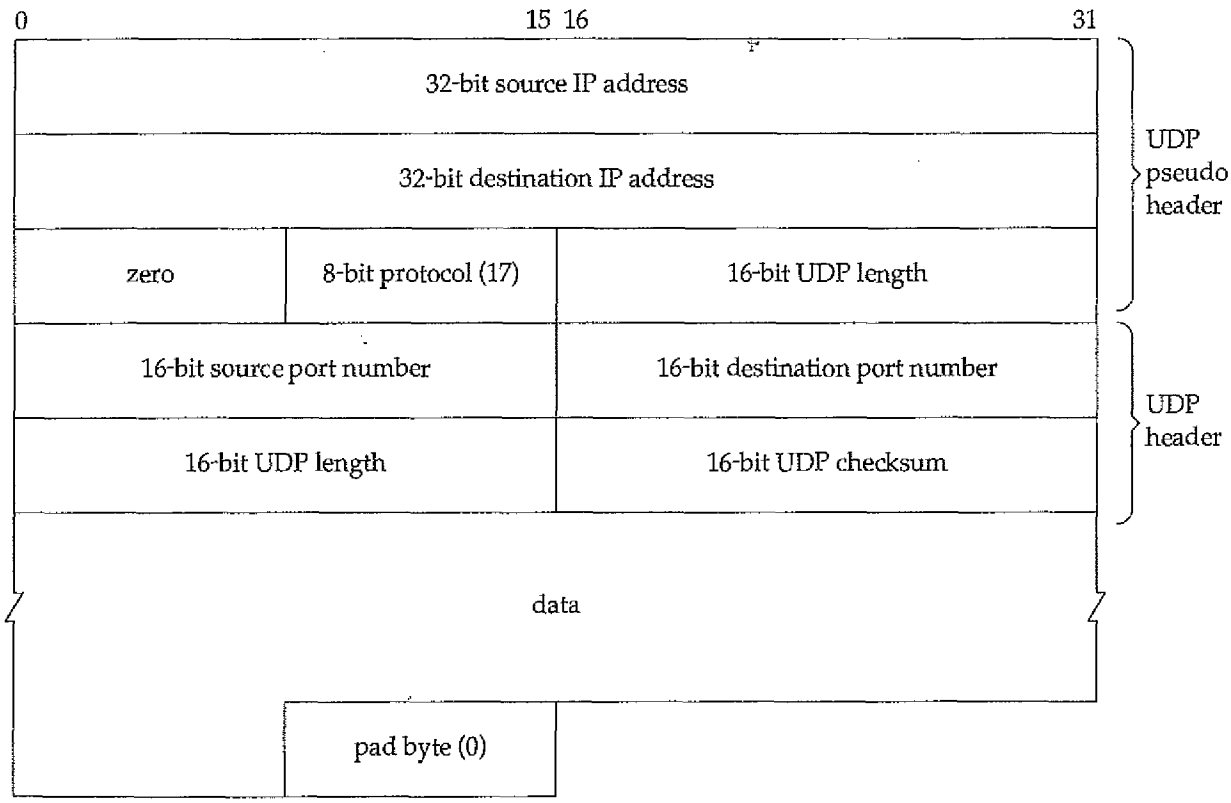


Figure 11.3 Fields used for computation of UDP checksum.

In this figure we explicitly show a datagram with an odd length, requiring a pad byte for the checksum computation. Notice that the length of the UDP datagram appears twice in the checksum computation.

If the calculated checksum is 0, it is stored as all one bits (65535), which is equivalent in ones-complement arithmetic. If the transmitted checksum is 0, it indicates that the sender did not compute the checksum.

If the sender did compute a checksum and the receiver detects a checksum error, the UDP datagram is silently discarded. No error message is generated. (This is what happens if an IP header checksum error is detected by IP.)

This UDP checksum is an end-to-end checksum. It is calculated by the sender, and then verified by the receiver. It is designed to catch any modification of the UDP header or data anywhere between the sender and receiver.

Despite UDP checksums being optional, they should *always* be enabled. During the 1980s some computer vendors turned off UDP checksums by default, to speed up their implementation of Sun's Network File System (NFS), which uses UDP. While this *might* be acceptable on a single LAN, where the cyclic redundancy check on the data-link frame (e.g., Ethernet or token ring frame) can detect most corruption of the frame, when the datagrams pass through routers, all bets are off. Believe it or not, there have been routers with software and hardware bugs that have modified bits in the datagrams being routed. These errors are undetectable in a UDP datagram if the end-to-end UDP checksum is disabled. Also realize that some data-link protocols (e.g., SLIP) don't have any form of data-link checksum.

> The Host Requirements RFC requires that UDP checksums be enabled by default. It also states that an implementation must verify a received checksum if the sender calculated one (i.e., the received checksum is nonzero). Many implementations violate this, however, and only verify a received checksum if outgoing checksums are enabled.

`tcpdump` Output

It is hard to detect whether a particular system has UDP checksums enabled. It is normally impossible for an application to obtain the checksum field in a received UDP header. To get around this, the author added another option to the `tcpdump` program that prints the received UDP checksum. If this printed value is 0, it means the sending host did not calculate the checksum.

Figure 11.4 shows the output to and from three different systems on our test network (see the figure on the inside front cover). We ran our `sock` program (Appendix C), sending a single UDP datagram with 9 bytes of data to the standard echo server.

```
1   0.0                 sun.1900 > gemini.echo: udp 9 (UDP cksum=6e90)
2   0.303755 ( 0.3038)  gemini.echo > sun.1900: udp 9 (UDP cksum=0)

3  17.392480 (17.0887)  sun.1904 > aix.echo: udp 9 (UDP cksum=6e3b)
4  17.614371 ( 0.2219)  aix.echo > sun.1904: udp 9 (UDP cksum=6e3b)

5  32.092454 (14.4781)  sun.1907 > solaris.echo: udp 9 (UDP cksum=6e74)
6  32.314378 ( 0.2219)  solaris.echo > sun.1907: udp 9 (UDP cksum=6e74)
```

Figure 11.4 `tcpdump` output to see whether other hosts enable UDP checksum.

We can see from this that two of the three systems have UDP checksums enabled.

Also notice that for this simple example the outgoing datagram has the same checksum as the incoming datagram (lines 3 and 4, 5 and 6). Looking at Figure 11.3 we see that the two IP addresses are swapped, as are the two port numbers. The other fields in the pseudo-header and the UDP header are the same, as is the data being echoed. This

reiterates that the UDP checksums (indeed, all the checksums in the TCP/IP protocol suite) are simple 16-bit sums. They cannot detect an error that swaps two of the 16-bit values.

> The author also directed a DNS query at each of the eight root name servers described in Section 14.2. The DNS uses UDP primarily, and only two of the eight had UDP checksums enabled!

Some Statistics

[Mogul 1992] provides counts of various checksum errors on a busy NFS (Network File System) server that had been up for 40 days. Figure 11.5 summarizes these numbers.

Layer	Number of checksum errors	Approximate total number of packets
Ethernet	446	170,000,000
IP	14	170,000,000
UDP	5	140,000,000
TCP	350	30,000,000

Figure 11.5 Counts of corrupted packets detected by various checksums.

The final column is only the approximate total for each row, since other protocols are in use at the Ethernet and IP layers. For example, not all the Ethernet frames are IP datagrams, since minimally ARP is also used on an Ethernet. Not all IP datagrams are UDP or TCP, since ICMP also uses IP.

Note the much higher percentage of TCP checksum errors compared to UDP checksum errors. This is probably because the TCP connections on this system tended to be "long distance" (traversing many routers, bridges, etc.) while the UDP traffic was local.

The bottom line is not to trust the data-link (e.g., Ethernet, token ring, etc.) CRC completely. You should enable the end-to-end checksums all the time. Also, if your data is valuable, you might not want to trust either the UDP or the TCP checksum completely, since these are simple checksums and were not meant to catch all possible errors.

11.4 A Simple Example

We'll use our `sock` program to generate some UDP datagrams that we can watch with `tcpdump`:

```
bsdi % sock -v -u -i -n4 svr4 discard
connected on 140.252.13.35.1108 to 140.252.13.34.9

bsdi % sock -v -u -i -n4 -w0 svr4 discard
connected on 140.252.13.35.1110 to 140.252.13.34.9
```

The first time we execute the program we specify the verbose mode (`-v`) to see the ephemeral port numbers, specify UDP (`-u`) instead of the default TCP, and use the

source mode (`-i`) to send data instead of trying to read and write standard input and output. The `-n4` option says to output 4 datagrams (instead of the default 1024) and the destination host is `svr4`. We described the discard service in Section 1.12. We use the default output size of 1024 bytes per write.

The second time we run the program we specify `-w0`, causing 0-length datagrams to be written. Figure 11.6 shows the `tcpdump` output for both commands.

```
1   0.0                 bsdi.1108 > svr4.discard: udp 1024
2   0.002424 ( 0.0024)  bsdi.1108 > svr4.discard: udp 1024
3   0.006210 ( 0.0038)  bsdi.1108 > svr4.discard: udp 1024
4   0.010276 ( 0.0041)  bsdi.1108 > svr4.discard: udp 1024

5  41.720114 (41.7098)  bsdi.1110 > svr4.discard: udp 0
6  41.721072 ( 0.0010)  bsdi.1110 > svr4.discard: udp 0
7  41.722094 ( 0.0010)  bsdi.1110 > svr4.discard: udp 0
8  41.723070 ( 0.0010)  bsdi.1110 > svr4.discard: udp 0
```

Figure 11.6 `tcpdump` output when UDP datagrams are sent in one direction.

This output shows the four 1024-byte datagrams, followed by the four 0-length datagrams. Each datagram followed the previous by a few milliseconds. (It took 41 seconds to type in the second command.)

There is no communication between the sender and receiver before the first datagram is sent. (We'll see in Chapter 17 that TCP must establish a connection with the other end before the first byte of data can be sent.) Also, there are no acknowledgments by the receiver when the data is received. The sender, in this example, has no idea whether the other end receives the datagrams.

Finally note that the source UDP port number changes each time the program is run. First it is 1108 and then it is 1110. We mentioned in Section 1.9 that the ephemeral port numbers used by clients are typically in the range 1024 through 5000, as we see here.

11.5 IP Fragmentation

As we described in Section 2.8, the physical network layer normally imposes an upper limit on the size of the frame that can be transmitted. Whenever the IP layer receives an IP datagram to send, it determines which local interface the datagram is being sent on (routing), and queries that interface to obtain its MTU. IP compares the MTU with the datagram size and performs fragmentation, if necessary. Fragmentation can take place either at the original sending host or at an intermediate router.

When an IP datagram is fragmented, it is not reassembled until it reaches its final destination. (This handling of reassembly differs from some other networking protocols that require reassembly to take place at the next hop, not at the final destination.) The IP layer at the destination performs the reassembly. The goal is to make fragmentation and reassembly transparent to the transport layer (TCP and UDP), which it is, except for possible performance degradation. It is also possible for the fragment of a datagram to

again be fragmented (possibly more than once). The information maintained in the IP header for fragmentation and reassembly provides enough information to do this.

Recalling the IP header (Figure 3.1, p. 34), the following fields are used in fragmentation. The *identification* field contains a unique value for each IP datagram that the sender transmits. This number is copied into each fragment of a particular datagram. (We now see the use for this field.) The *flags* field uses one bit as the "more fragments" bit. This bit is turned on for each fragment comprising a datagram except the final fragment. The *fragment offset* field contains the offset (in 8-byte units) of this fragment from the beginning of the original datagram. Also, when a datagram is fragmented the *total length* field of each fragment is changed to be the size of that fragment.

Finally, one of the bits in the flags field is called the "don't fragment" bit. If this is turned on, IP will not fragment the datagram. Instead the datagram is thrown away and an ICMP error ("fragmentation needed but don't fragment bit set," Figure 6.3) is sent to the originator. We'll see an example of this error in the next section.

When an IP datagram is fragmented, each fragment becomes its own packet, with its own IP header, and is routed independently of any other packets. This makes it possible for the fragments of a datagram to arrive at the final destination out of order, but there is enough information in the IP header to allow the receiver to reassemble the fragments correctly.

Although IP fragmentation looks transparent, there is one feature that makes it less than desirable: if one fragment is lost the entire datagram must be retransmitted. To understand why this happens, realize that IP itself has no timeout and retransmission—that is the responsibility of the higher layers. (TCP performs timeout and retransmission, UDP doesn't. Some UDP applications perform timeout and retransmission themselves.) When a fragment is lost that came from a TCP segment, TCP will time out and retransmit the entire TCP segment, which corresponds to an IP datagram. There is no way to resend only one fragment of a datagram. Indeed, if the fragmentation was done by an intermediate router, and not the originating system, there is no way for the originating system to know how the datagram was fragmented. For this reason alone, fragmentation is often avoided. [Kent and Mogul 1987] provide arguments for avoiding fragmentation.

Using UDP it is easy to generate IP fragmentation. (We'll see later that TCP tries to avoid fragmentation and that it is nearly impossible for an application to force TCP to send segments large enough to require fragmentation.) We can use our `sock` program and increase the size of the datagram until fragmentation occurs. On an Ethernet the maximum amount of data in a frame is 1500 bytes (Figure 2.1), which leaves 1472 bytes for our data, assuming 20 bytes for the IP header and 8 bytes for the UDP header. We'll run our `sock` program, with data sizes of 1471, 1472, 1473, and 1474 bytes. We expect the last two to cause fragmentation:

```
bsdi % sock -u -i -n1 -w1471 svr4 discard
bsdi % sock -u -i -n1 -w1472 svr4 discard
bsdi % sock -u -i -n1 -w1473 svr4 discard
bsdi % sock -u -i -n1 -w1474 svr4 discard
```

Figure 11.7 shows the corresponding `tcpdump` output.

```
1   0.0                 bsdi.1112 > svr4.discard: udp 1471
2  21.008303 (21.0083)  bsdi.1114 > svr4.discard: udp 1472
3  50.449704 (29.4414)  bsdi.1116 > svr4.discard: udp 1473 (frag 26304:1480@0+)
4  50.450040 ( 0.0003)  bsdi > svr4: (frag 26304:1@1480)
5  75.328650 (24.8786)  bsdi.1118 > svr4.discard: udp 1474 (frag 26313:1480@0+)
6  75.328982 ( 0.0003)  bsdi > svr4: (frag 26313:2@1480)
```

Figure 11.7 Watching fragmentation of UDP datagrams.

The first two UDP datagrams (lines 1 and 2) fit into Ethernet frames, and are not fragmented. But the length of the IP datagram corresponding to the write of 1473 bytes is 1501, which must be fragmented (lines 3 and 4). Similarly the datagram generated by the write of 1474 bytes is 1502, and is also fragmented (lines 5 and 6).

When the IP datagram is fragmented, `tcpdump` prints additional information. First, the output `frag 26304` (lines 3 and 4) and `frag 26313` (lines 5 and 6) specify the value of the identification field in the IP header.

The next number in the fragmentation information, the 1480 between the colon and the at sign in line 3, is the size, excluding the IP header. The first fragment of both datagrams contains 1480 bytes of data: 8 bytes for the UDP header and 1472 bytes of user data. (The 20-byte IP header makes the packet exactly 1500 bytes.) The second fragment of the first datagram (line 4) contains 1 byte of data—the remaining byte of user data. The second fragment of the second datagram (line 6) contains the remaining 2 bytes of user data.

Fragmentation requires that the data portion of the generated fragments (that is, everything excluding the IP header) be a multiple of 8 bytes for all fragments other than the final one. In this example, 1480 is a multiple of 8.

The number following the at sign is the offset of the data in the fragment, from the start of the datagram. The first fragment of both datagrams starts at 0 (lines 3 and 5) and the second fragment of both datagrams starts at byte offset 1480 (lines 4 and 6). The plus sign following this offset that is printed for the first fragment of both datagrams means there are more fragments comprising this datagram. This plus sign corresponds to the "more fragments" bit in the 3-bit flags in the IP header. The purpose of this bit is to let the receiver know when it has completed the reassembly of all the fragments for a datagram.

Finally, notice that lines 4 and 6 (fragments other than the first) omit the protocol (UDP) and the source and destination ports. The protocol could be printed, since it's in the IP header that's copied into the fragments. The port numbers, however, are in the UDP header, which only occurs in the first fragment.

Figure 11.8 shows what's happening with the third datagram that is sent (with 1473 bytes of user data). It reiterates that any transport layer header appears only in the first fragment.

Also note the terminology: an *IP datagram* is the unit of end-to-end transmission at the IP layer (before fragmentation and after reassembly), and a *packet* is the unit of data passed between the IP layer and the link layer. A packet can be a complete IP datagram or a fragment of an IP datagram.

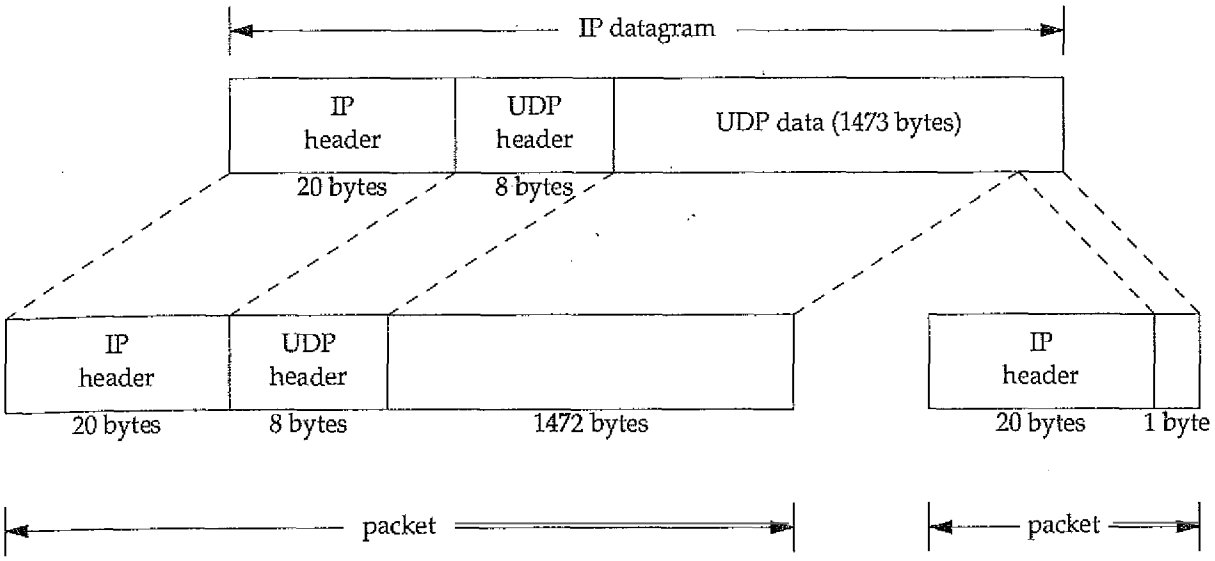


Figure 11.8 Example of UDP fragmentation.

11.6 ICMP Unreachable Error (Fragmentation Required)

Another variation of the ICMP unreachable error occurs when a router receives a datagram that requires fragmentation, but the don't fragment (DF) flag is turned on in the IP header. This error can be used by a program that needs to determine the smallest MTU in the path to a destination—called the *path MTU discovery* mechanism (Section 2.9).

Figure 11.9 shows the format of the ICMP unreachable error for this case. This differs from Figure 6.10 because bits 16–31 of the second 32-bit word can provide the MTU of the next hop, instead of being 0.

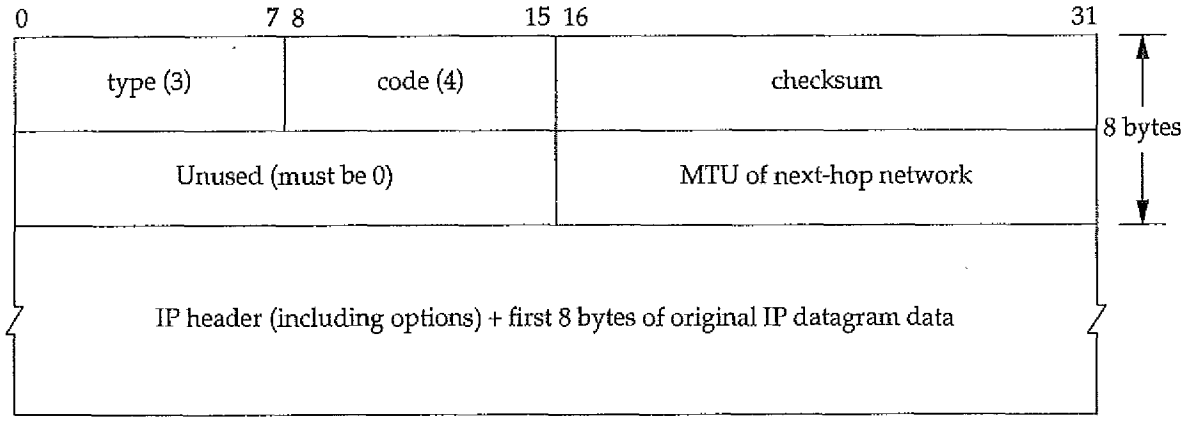


Figure 11.9 ICMP unreachable error when fragmentation required but don't fragment bit set.

If a router doesn't provide this newer format ICMP error, the next-hop MTU is set to 0.

The new Router Requirements RFC [Almquist 1993] states that a router must generate this newer form when originating this ICMP unreachable error.

Example

A problem encountered by the author involving fragmentation and this ICMP error is trying to determine the MTU on the dialup SLIP link from the router `netb` to the host `sun`. We know the MTU of this link from `sun` to `netb`: it's part of the SLIP configuration process when SLIP was installed in the host `sun`, plus we saw it with the `netstat` command in Section 3.9. We want to determine the MTU in the other direction also. (In Chapter 25 we'll see how to determine this using SNMP.) On a point-to-point link, it is not required that the MTU be the same in both directions.

The technique used was to run `ping` on the host `solaris`, to the host `bsdi`, increasing the size of the data packets until fragmentation was seen on the incoming packets. This is shown in Figure 11.10.

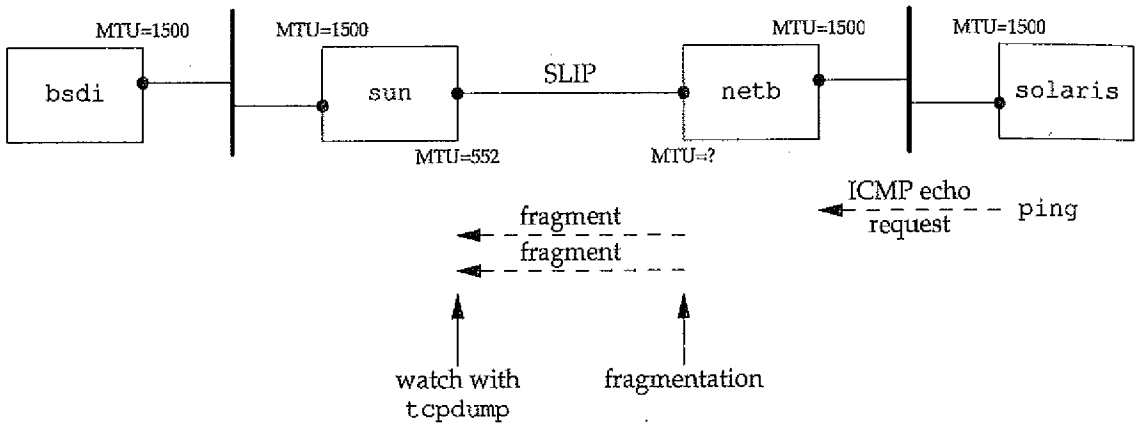


Figure 11.10 Systems being used to determine MTU of SLIP link from `netb` to `sun`.

`tcpdump` was run on the host `sun`, watching the SLIP link, to see when fragmentation occurred. No fragmentation was observed and everything was fine until the size of the data portion of the `ping` packet was increased from 500 to 600 bytes. The incoming echo requests were seen (there was still no fragmentation), but the echo replies disappeared.

To track this down, `tcpdump` was also run on `bsdi`, to see what it was receiving and sending. Figure 11.11 shows the output.

```
1  0.0                    solaris > bsdi: icmp: echo request (DF)
2  0.000000 (0.0000)      bsdi > solaris: icmp: echo reply (DF)
3  0.000000 (0.0000)      sun > bsdi: icmp: solaris unreachable -
                          need to frag, mtu = 0 (DF)

4  0.738400 (0.7384)      solaris > bsdi: icmp: echo request (DF)
5  0.748800 (0.0104)      bsdi > solaris: icmp: echo reply (DF)
6  0.748800 (0.0000)      sun > bsdi: icmp: solaris unreachable -
                          need to frag, mtu = 0 (DF)
```

Figure 11.11 `tcpdump` output for `ping` of `bsdi` from `solaris` with 600-byte IP datagram.

First, the notation `(DF)` in each line means the don't fragment bit is turned on in the IP header. It turns out that Solaris 2.2 normally turns this bit on, as part of its implementation of the path MTU discovery mechanism.

Line 1 shows that the echo request got through the router `netb` to `sun` without being fragmented, and with the DF bit set, so we know that the SLIP MTU of `netb` has not been reached yet.

Next, notice in line 2 that the DF flag is copied into the echo reply. This is what causes the problem. The echo reply is the same size as the echo request (just over 600 bytes), but the MTU on `sun`'s outgoing SLIP interface is 552. The echo reply needs to be fragmented, but the DF flag is set. This causes `sun` to generate the ICMP unreachable error back to `bsdi` (where it's discarded).

This is why we never saw any echo replies on `solaris`. The replies never got past `sun`. Figure 11.12 shows the path of the packets.

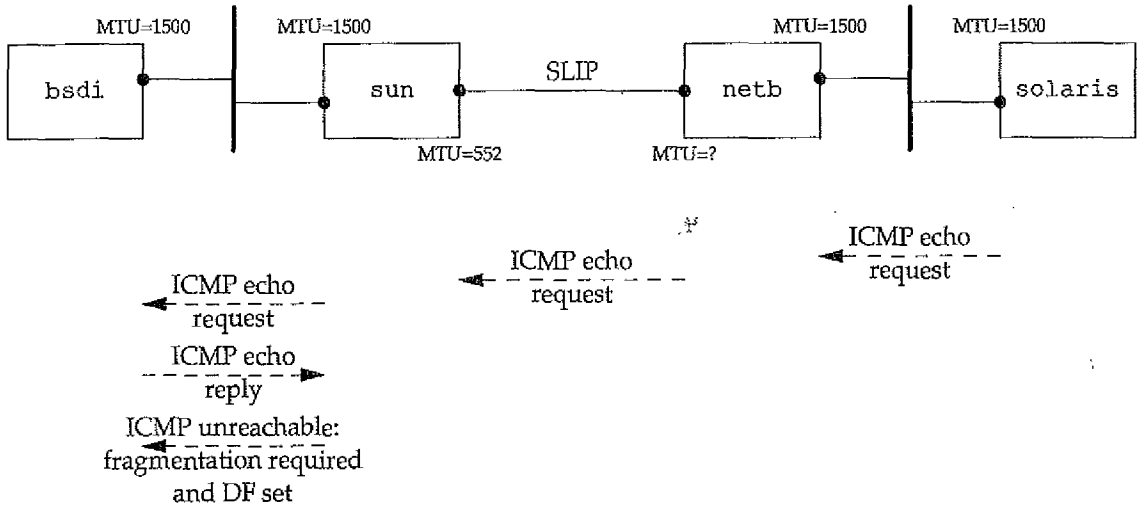


Figure 11.12 Packets exchanged in example.

Finally, the notation `mtu=0` in lines 3 and 6 of Figure 11.11 indicates that `sun` does not return the MTU of the outgoing interface in the ICMP unreachable message, as shown in Figure 11.9. (In Section 25.9 we return to this problem and use SNMP to determine that the MTU of the SLIP interface on `netb` is 1500.)

11.7 Determining the Path MTU Using Traceroute

Although most systems don't support the path MTU discovery feature, we can easily modify a version of `traceroute` (Chapter 8) to let us determine the path MTU. What we'll do is send packets with the "don't fragment" bit set. The size of the first packet we send will equal the MTU of the outgoing interface, and whenever we receive an ICMP "can't fragment" error (which we described in the previous section) we'll reduce the size of the packet. If the router sending the ICMP error sends the newer version that includes the MTU of the outgoing interface, we'll use that value; otherwise we'll try the next smallest MTU. As RFC 1191 [Mogul and Deering 1990] states, there are a limited number of MTUs, so our program has a table of the likely values and moves to the next smallest value.

Let's first try it from our host `sun` to the host `slip`, knowing that the SLIP link has an MTU of 296:

```
sun % traceroute.pmtu slip
traceroute to slip (140.252.13.65), 30 hops max
outgoing MTU = 1500
 1  bsdi (140.252.13.35)  15 ms  6 ms  6 ms
 2  bsdi (140.252.13.35)  6 ms
fragmentation required and DF set, trying new MTU = 1492
fragmentation required and DF set, trying new MTU = 1006
fragmentation required and DF set, trying new MTU = 576
fragmentation required and DF set, trying new MTU = 552
fragmentation required and DF set, trying new MTU = 544
fragmentation required and DF set, trying new MTU = 512
fragmentation required and DF set, trying new MTU = 508
fragmentation required and DF set, trying new MTU = 296
 2  slip (140.252.13.65)  377 ms  377 ms  377 ms
```

In this example the router `bsdi` does not return the MTU of the outgoing interface in the ICMP error, so we step through the likely values for the MTU. The first line of output for a TTL of 2 prints a hostname of `bsdi`, but that's because it's the router returning the ICMP error. The final line of output for a TTL of 2 is what we're looking for.

It's not hard to modify the ICMP code on `bsdi` to return the MTU of the outgoing interface, and if we do that and rerun our program, we get the following output:

```
sun % traceroute.pmtu slip
traceroute to slip (140.252.13.65), 30 hops max
outgoing MTU = 1500
 1  bsdi (140.252.13.35)  53 ms  6 ms  6 ms
 2  bsdi (140.252.13.35)  6 ms
fragmentation required and DF set, next hop MTU = 296
 2  slip (140.252.13.65)  377 ms  378 ms  377 ms
```

Here we don't have to try eight different values for the MTU before finding the right one—the router returns the correct value.

The Worldwide Internet

As an experiment, this modified version of `traceroute` was run numerous times to various hosts around the world. Fifteen countries (including Antarctica) were reached and various transatlantic and transpacific links were used. Before doing this, however, the MTU of the dialup SLIP link between the author's subnet and the router `netb` (Figure 11.12) was increased to 1500, the same as an Ethernet.

Out of 18 runs, only 2 had a path MTU of less than 1500. One of the transatlantic links had an MTU of 572 (a value not even listed as a likely value in RFC 1191) and the router did return the newer format ICMP error. Another link, between two routers in Japan, wouldn't handle a 1500-byte frame, and the router did not return the newer format ICMP error. Setting the MTU down to 1006 did work.

The conclusion we can make from this experiment is that many, but not all, WANs today can handle packets larger than 512 bytes. Using the path MTU discovery feature will allow applications to take advantage of these larger MTUs.

11.8 Path MTU Discovery with UDP

Let's examine the interaction between an application using UDP and the path MTU discovery mechanism. We want to see what happens when the application writes datagrams that are too big for some intermediate link.

Example

Since the only system that we've been using that supports the path MTU discovery mechanism is Solaris 2.x, we'll use it as the source host to send 650-byte datagrams to `slip`. Since our host `slip` sits behind a SLIP link with an MTU of 296, any UDP datagram greater than 268 bytes (296 – 20 – 8) with the "don't fragment" bit set should cause the router `bsdi` to generate the ICMP "can't fragment" error. Figure 11.13 shows the topology and the MTUs.

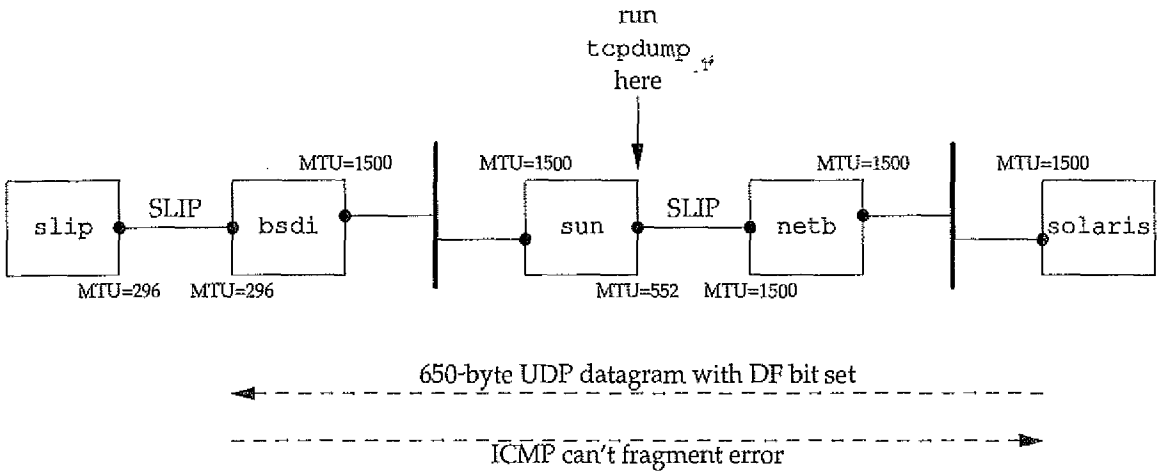


Figure 11.13 Systems used for path MTU discovery using UDP.

The following command generates ten 650-byte UDP datagrams, with a 5-second pause between each datagram:

```
solaris % sock -u -i -n10 -w650 -p5 slip discard
```

Figure 11.14 shows the `tcpdump` output. When this example was run, the router `bsdi` was set to not return the next-hop MTU as part of the ICMP "can't fragment" error.

The first datagram is sent with the DF bit set (line 1) and generates the expected error from the router `bsdi` (line 2). What's puzzling is that the next datagram is also sent with the DF bit set (line 3) and generates the same ICMP error (line 4). We would expect this datagram to be sent with the DF bit off.

On line 5 it appears IP has finally learned that datagrams to this destination should not be sent with the DF bit set, so IP goes ahead and fragments the datagrams at the source host. This is different from earlier examples where IP sends the datagram that is passed to it by UDP and allows the router with the smaller MTU (`bsdi` in this case) to

```
 1   0.0                 solaris.38196 > slip.discard: udp 650 (DF)
 2   0.004218 (0.0042)   bsdi > solaris: icmp:
                             slip unreachable - need to frag, mtu = 0 (DF)

 3   4.980528 (4.9763)   solaris.38196 > slip.discard: udp 650 (DF)
 4   4.984503 (0.0040)   bsdi > solaris: icmp:
                             slip unreachable - need to frag, mtu = 0 (DF)

 5   9.870407 (4.8859)   solaris.38196 > slip.discard: udp 650 (frag 47942:552@0+)
 6   9.960056 (0.0896)   solaris > slip: (frag 47942:106@552)

 7  14.940338 (4.9803)   solaris.38196 > slip.discard: udp 650 (DF)
 8  14.944466 (0.0041)   bsdi > solaris: icmp:
                             slip unreachable - need to frag, mtu = 0 (DF)

 9  19.890015 (4.9455)   solaris.38196 > slip.discard: udp 650 (frag 47944:552@0+)
10  19.950463 (0.0604)   solaris > slip: (frag 47944:106@552)

11  24.870401 (4.9199)   solaris.38196 > slip.discard: udp 650 (frag 47945:552@0+)
12  24.960038 (0.0896)   solaris > slip: (frag 47945:106@552)

13  29.880182 (4.9201)   solaris.38196 > slip.discard: udp 650 (frag 47946:552@0+)
14  29.940498 (0.0603)   solaris > slip: (frag 47946:106@552)

15  34.860607 (4.9201)   solaris.38196 > slip.discard: udp 650 (frag 47947:552@0+)
16  34.950051 (0.0894)   solaris > slip: (frag 47947:106@552)

17  39.870216 (4.9202)   solaris.38196 > slip.discard: udp 650 (frag 47948:552@0+)
18  39.930443 (0.0602)   solaris > slip: (frag 47948:106@552)

19  44.940485 (5.0100)   solaris.38196 > slip.discard: udp 650 (DF)
20  44.944432 (0.0039)   bsdi > solaris: icmp:
                             slip unreachable - need to frag, mtu = 0 (DF)
```

Figure 11.14 Path MTU discovery using UDP.

do the fragmentation. Since the ICMP "can't fragment" message didn't specify the next-hop MTU, it appears that IP guesses that an MTU of 576 is OK. The first fragment (line 5) contains 544 bytes of UDP data, the 8-byte UDP header, and the 20-byte IP header, for a total IP datagram size of 572 bytes. The second fragment (line 6) contains the remaining 106 bytes of UDP data and a 20-byte IP header.

Unfortunately the next datagram, line 7, has its DF bit set, so it's discarded by `bsdi` and the ICMP error returned. What has happened here is that an IP timer has expired telling IP to see if the path MTU has increased by setting the DF bit again. We see this happen again on lines 19 and 20. Comparing the times on lines 7 and 19 it appears that IP turns on the DF bit, to see if the path MTU has increased, every 30 seconds.

> This 30-second timer value is way too small. RFC 1191 recommends a value of 10 minutes. It can be changed by modifying the parameter `ip_ire_pathmtu_interval` (Section E.4). Also there is no way in Solaris 2.2 to turn off this path MTU discovery for a single UDP application or for all UDP applications. It can only be enabled or disabled on a systemwide basis by changing the parameter `ip_path_mtu_discovery`. As we can see from this example, enabling path MTU discovery when UDP applications write datagrams that will probably be fragmented can cause datagrams to be discarded.

The maximum datagram size assumed by the IP layer on `solaris` (576 bytes) is not right. In Figure 11.13 we see that the real MTU is 296 bytes. This means the fragments generated by `solaris` will be fragmented again by `bsdi`. Figure 11.15 shows the `tcpdump` output collected on the destination host (`slip`) for the first datagram that arrives (lines 5 and 6 from Figure 11.14).

```
1  0.0                 solaris.38196 > slip.discard: udp 650 (frag 47942:272@0+)
2  0.304513 (0.3045)   solaris > slip: (frag 47942:272@272+)
3  0.334651 (0.0301)   solaris > slip: (frag 47942:8@544+)
4  0.466642 (0.1320)   solaris > slip: (frag 47942:106@552)
```

Figure 11.15 First datagram arriving at host `slip` from `solaris`.

In this example the host `solaris` should not fragment the outgoing datagrams but should turn off the DF bit and let the router with the smaller MTU do the fragmentation.

Now we'll run the same example but modify the router `bsdi` to return the next-hop MTU in the ICMP "can't fragment" error. Figure 11.16 shows the first six lines of the `tcpdump` output.

```
1  0.0                 solaris.37974 > slip.discard: udp 650 (DF)
2  0.004199 (0.0042)   bsdi > solaris: icmp:
                             slip unreachable - need to frag, mtu = 296 (DF)

3  4.950193 (4.9460)   solaris.37974 > slip.discard: udp 650 (DF)
4  4.954325 (0.0041)   bsdi > solaris: icmp:
                             slip unreachable - need to frag, mtu = 296 (DF)

5  9.779855 (4.8255)   solaris.37974 > slip.discard: udp 650 (frag 35278:272@0+)
6  9.930018 (0.1502)   solaris > slip: (frag 35278:272@272+)
7  9.990170 (0.0602)   solaris > slip: (frag 35278:114@544)
```

Figure 11.16 Path MTU discovery using UDP.

Again, the first two datagrams are sent with the DF bit set, and both elicit the ICMP error. The ICMP error now specifies the next-hop MTU of 296.

In lines 5, 6, and 7 we see the source host perform fragmentation, similar to Figure 11.14. But knowing the next-hop MTU, only three fragments are generated, compared to the four fragments generated by the router `bsdi` in Figure 11.15.

11.9 Interaction Between UDP and ARP

Using UDP we can see an interesting (and often unmentioned) interaction with UDP and typical implementations of ARP.

We use our `sock` program to generate a single UDP datagram with 8192 bytes of data. We expect this to generate six fragments on an Ethernet (see Exercise 11.3). We also assure that the ARP cache is empty before running the program, so that an ARP request and reply must be exchanged before the first fragment is sent.

```
bsdi % arp -a                                    verify ARP cache is empty
bsdi % sock -u -i -n1 -w8192 svr4 discard
```

We expect the first fragment to cause an ARP request to be sent. Five more fragments are generated by IP and this presents two timing questions that we'll need to use `tcpdump` to answer: are the remaining fragments ready to be sent before the ARP reply is received, and if so, what does ARP do with multiple packets to a given destination when it's waiting for an ARP reply? Figure 11.17 shows the `tcpdump` output.

```
 1  0.0                  arp who-has svr4 tell bsdi
 2  0.001234 (0.0012)    arp who-has svr4 tell bsdi
 3  0.001941 (0.0007)    arp who-has svr4 tell bsdi
 4  0.002775 (0.0008)    arp who-has svr4 tell bsdi
 5  0.003495 (0.0007)    arp who-has svr4 tell bsdi
 6  0.004319 (0.0008)    arp who-has svr4 tell bsdi
 7  0.008772 (0.0045)    arp reply svr4 is-at 0:0:c0:c2:9b:26
 8  0.009911 (0.0011)    arp reply svr4 is-at 0:0:c0:c2:9b:26
 9  0.011127 (0.0012)    bsdi > svr4: (frag 10863:800@7400)
10  0.011255 (0.0001)    arp reply svr4 is-at 0:0:c0:c2:9b:26
11  0.012562 (0.0013)    arp reply svr4 is-at 0:0:c0:c2:9b:26
12  0.013458 (0.0009)    arp reply svr4 is-at 0:0:c0:c2:9b:26
13  0.014526 (0.0011)    arp reply svr4 is-at 0:0:c0:c2:9b:26
14  0.015583 (0.0011)    arp reply svr4 is-at 0:0:c0:c2:9b:26
```

Figure 11.17 Packet exchange when an 8192-byte UDP datagram is sent on an Ethernet.

There are a few surprises in this output. First, six ARP requests are generated before the first ARP reply is returned. What we guess is happening is that IP generates the six fragments rapidly, and each one causes an ARP request.

Next, when the first ARP reply is received (line 7) only the last fragment is sent (line 9)! It appears that the first five fragments have been discarded. Indeed, this is the normal operation of ARP. Most implementations keep only the *last* packet sent to a given destination while waiting for an ARP reply.

> The Host Requirements RFC requires an implementation to prevent this type of *ARP flooding* (repeatedly sending an ARP request for the same IP address at a high rate). The recommended maximum rate is one per second. Here we see six ARP requests in 4.3 ms.
>
> The Host Requirements RFC states that ARP should save at least one packet, and this should be the latest packet. That's what we see here.

Another unexplained anomaly in this output is that `svr4` sends back seven ARP replies, not six.

The final point worth mentioning is that `tcpdump` was left to run for 5 minutes after the final ARP reply was returned, waiting to see if `svr4` sent back an ICMP "time exceeded during reassembly" error. The ICMP error was never sent. (We showed the format of this message in Figure 8.2. A *code* of 1 indicates that the time was exceeded during the reassembly of a datagram.)

The IP layer must start a timer when the first fragment of a datagram appears. Here "first" means the first arrival of any fragment for a given datagram, not the first fragment (with a fragment offset of 0). A normal timeout value is 30 or 60 seconds. If all the

fragments for this datagram have not arrived when the timer expires, all these fragments are discarded. If this were not done, fragments that never arrive (as we see in this example) could eventually cause the receiver to run out of buffers.

There are two reasons we don't see the ICMP message here. First, most Berkeley-derived implementations never generate this error! These implementations do set a timer, and do discard all fragments when the timer expires, but the ICMP error is never generated. Second, the first fragment—the one with an offset of 0 containing the UDP header—was never received. (It was the first of the five packets discarded by ARP.) An implementation is not required to generate the ICMP error unless this first fragment has been received. The reason is that the receiver of the ICMP error couldn't tell which user process sent the datagram that was discarded, because the transport layer header is not available. It's assumed that the upper layer (either TCP or the application using UDP) will eventually time out and retransmit.

In this section we've used IP fragmentation to see this interaction between UDP and ARP. We can also see this interaction if the sender quickly transmits multiple UDP datagrams. We chose to use fragmentation because the packets get generated quickly by IP, faster than multiple datagrams can be generated by a user process.

As unlikely as this example might seem, it occurs regularly. NFS sends UDP datagrams whose length just exceeds 8192 bytes. On an Ethernet these are fragmented as we've indicated, and if the appropriate ARP cache entry times out, you can see what we've shown here. NFS will time out and retransmit, but the first IP datagram can still be discarded because of ARP's limited queue.

11.10 Maximum UDP Datagram Size

Theoretically, the maximum size of an IP datagram is 65535 bytes, imposed by the 16-bit total length field in the IP header (Figure 3.1). With an IP header of 20 bytes and a UDP header of 8 bytes, this leaves a maximum of 65507 bytes of user data in a UDP datagram. Most implementations, however, provide less than this maximum.

There are two limits we can encounter. First the application program may be limited by its programming interface. The sockets API (Section 1.15) provides a function that the application can call to set the size of the receive buffer and the send buffer. For a UDP socket, this size is directly related to the maximum size UDP datagram the application can read or write. Most systems today provide a default of just over 8192 bytes for the maximum size of a UDP datagram that can be read or written. (This default is because 8192 is the amount of user data that NFS reads and writes by default.)

The next limitation comes from the kernel's implementation of TCP/IP. There may be implementation features (or bugs) that limit the size of an IP datagram to less than 65535 bytes.

> The author experimented with various UDP datagram sizes, using the `sock` program. Using the loopback interface under SunOS 4.1.3, the maximum size IP datagram was 32767 bytes. All higher values failed. But going across an Ethernet from BSD/386 to SunOS 4.1.3, the maximum size IP datagram the Sun could accept was 32786 (that is, 32758 bytes of user data). Using the loopback interface under Solaris 2.2, the maximum 65535-byte IP datagram could be sent and received. From Solaris 2.2 to AIX 3.2.2, the maximum 65535-byte IP datagram could be transferred. Obviously this limit depends on the source and destination implementations.

We mentioned in Section 3.2 that a host is required to receive at least a 576-byte IP datagram. Many UDP applications are designed to restrict their application data to 512 bytes or less, to stay below this limit. We saw this in Section 10.4, for example, where the Routing Information Protocol always sent less than 512 bytes of data per datagram. We'll encounter this same limit with other UDP applications: the DNS (Chapter 14), TFTP (Chapter 15), BOOTP (Chapter 16), and SNMP (Chapter 25).

Datagram Truncation

Just because IP is capable of sending and receiving a datagram of a given size doesn't mean the receiving application is prepared to read that size. UDP programming interfaces allow the application to specify the maximum number of bytes to return each time. What happens if the received datagram exceeds the size the application is prepared to deal with?

Unfortunately the answer depends on the programming interface and the implementation.

> The traditional Berkeley version of the sockets API truncates the datagram, discarding any excess data. Whether the application is notified depends on the version. (4.3BSD Reno and later can notify the application that the datagram was truncated.)
>
> The sockets API under SVR4 (including Solaris 2.x) does not truncate the datagram. Any excess data is returned in subsequent reads. The application is not notified that multiple reads are being fulfilled from a single UDP datagram.
>
> The TLI API does not discard the data. Instead a flag is returned indicating that more data is available, and subsequent reads by the application return the rest of the datagram.

When we discuss TCP we'll see that it provides a continuous stream of bytes to the application, without any message boundaries. TCP passes the data to the application in whatever size reads the application asks for—there is never any data loss across this interface.

11.11 ICMP Source Quench Error

Using UDP we are also able to generate the ICMP "source quench" error. This is an error that may be generated by a system (router or host) when it receives datagrams at a rate that is too fast to be processed. Note the qualifier "may." A system is not required to send a source quench, even if it runs out of buffers and throws datagrams away.

Figure 11.18 shows the format of the ICMP source quench error. We have a perfect scenario with our test network for generating this error. We can send datagrams from `bsdi` to the router `sun` across the Ethernet that must be routed across the dialup SLIP link. Since the SLIP link is about 1000 times slower than the Ethernet, we should easily be able to overrun its buffer space. The following command sends 100 1024-byte datagrams from the host `bsdi` through the router `sun` to `solaris`. We send the datagrams to the standard discard service, where they'll be ignored:

```
bsdi % sock -u -i -w1024 -n100 solaris discard
```

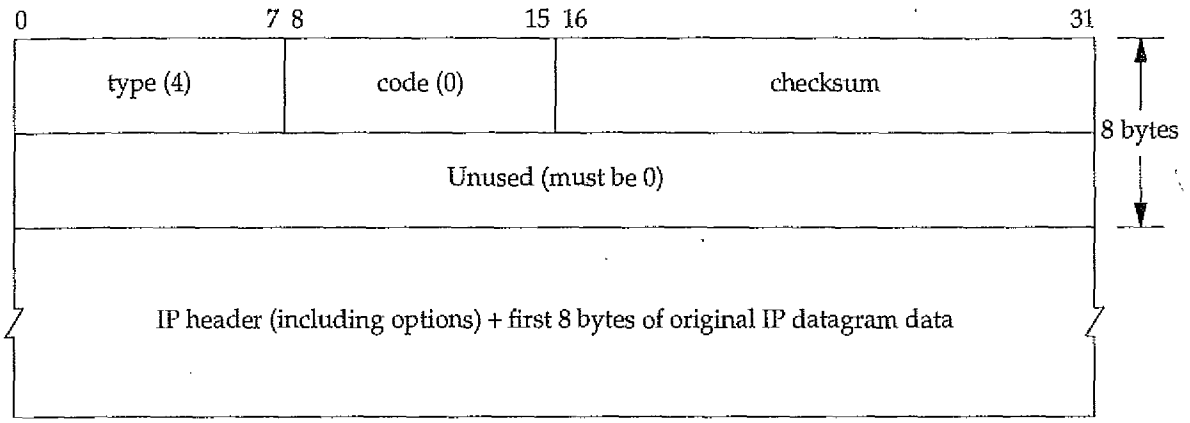


Figure 11.18 ICMP source quench error.

Figure 11.19 shows the `tcpdump` output corresponding to this command.

```
 1  0.0                  bsdi.1403 > solaris.discard: udp 1024
                                 26 lines that we don't show
27  0.10 (0.00)          bsdi.1403 > solaris.discard: udp 1024
28  0.11 (0.01)          sun > bsdi: icmp: source quench
29  0.11 (0.00)          bsdi.1403 > solaris.discard: udp 1024
30  0.11 (0.00)          sun > bsdi: icmp: source quench
                                 142 lines that we don't show
173  0.71 (0.06)         bsdi.1403 > solaris.discard: udp 1024
174  0.71 (0.00)         sun > bsdi: icmp: source quench
```

Figure 11.19 ICMP source quench from the router sun.

We have removed lots of lines from this output; there is a pattern. The first 26 datagrams are received without an error; we show the output only for the first. Starting with our 27th datagram, however, every time we send a datagram, we receive a source quench in return. There are a total of $26 + (74 \times 2) = 174$ lines of output.

From our serial line throughput calculations in Section 2.10, it takes just over 1 second to transfer a 1024-byte datagram at 9600 bits/sec. (In our example it should take longer than this since the $20 + 8 + 1024$ byte datagram will be fragmented because the MTU of the SLIP link from `sun` to `netb` is 552 bytes.) But we can see from the timing in Figure 11.19 that the router `sun` receives all 100 datagrams in less than 1 second, before the first one is through the SLIP link. It's not surprising that we used up many of its buffers.

> Although RFC 1009 [Braden and Postel 1987] requires a router to generate source quenches when it runs out of buffers, the new Router Requirements RFC [Almquist 1993] changes this and says that a router must not originate source quench errors. The current feeling is to deprecate the source quench error, since it consumes network bandwidth and is an ineffective and unfair fix for congestion.

Another point to make regarding this example is that our `sock` program either never received a notification that the source quenches were being received, or if it did, it

appears to have ignored them. It turns out that BSD implementations normally ignore received source quenches if the protocol is UDP. (TCP is notified, and slows down the data transfer on the connection that generated the source quench, as we discuss in Section 21.10.) Part of the problem is that the process that generated the data that caused the source quench may have already terminated when the source quench is received. Indeed, if we use the Unix `time` program to measure how long our `sock` program takes to run, it only executes for about 0.5 seconds. But from Figure 11.19 we see that some of the source quenches are received 0.71 seconds after the first datagram was sent, after the process has terminated. What is happening is that our program writes 100 datagrams and terminates. But not all 100 datagrams have been sent—some are queued for output.

This example reiterates that UDP is an unreliable protocol and illustrates the value of end-to-end flow control. Even though our `sock` program successfully wrote 100 datagrams to its network, only 26 were really sent to the destination. The other 74 were probably discarded by the intermediate router. Unless we build some form of acknowledgment into the application, the sender has no idea whether the receiver really got the data.

11.12 UDP Server Design

There are some implications in using UDP that affect the design and implementation of a server. The design and implementation of clients is usually easier than that of servers, which is why we talk about server design and not client design. Servers typically interact with the operating system and most servers need a way to handle multiple clients at the same time.

Normally a client starts, immediately communicates with a single server, and is done. Servers, on the other hand, start and then go to sleep, waiting for a client's request to arrive. In the case of UDP, the server wakes up when a client's datagram arrives, probably containing a request message of some form from the client.

Our interest here is not in the programming aspects of clients and servers ([Stevens 1990] covers all those details), but in the protocol features of UDP that affect the design and implementation of a server using UDP. (We examine the details of TCP server design in Section 18.11.) Although some of the features we describe depend on the implementation of UDP being used, the features are common to most implementations.

Client IP Address and Port Number

What arrives from the client is a UDP datagram. The IP header contains the source and destination IP addresses, and the UDP header contains the source and destination UDP port numbers. When an application receives a UDP datagram, it must be told by the operating system who sent the message—the source IP address and port number.

This feature allows an iterative UDP server to handle multiple clients. Each reply is sent back to the client that sent the request.

Destination IP Address

Some applications need to know who the datagram was sent to, that is, the destination IP address. For example, the Host Requirements RFC states that a TFTP server should ignore received datagrams that are sent to a broadcast address. (We describe broadcasting in Chapter 12 and TFTP in Chapter 15.)

This requires the operating system to pass the destination IP address from the received UDP datagram to the application. Unfortunately, not all implementations provide this capability.

> The sockets API provides this capability with the `IP_RECVDSTADDR` socket option. Of the systems used in the text, only BSD/386, 4.4BSD, and AIX 3.2.2 support this option. SVR4, SunOS 4.x, and Solaris 2.x don't support it.

UDP Input Queue

We said in Section 1.8 that most UDP servers are iterative servers. This means a single server process handles all the client requests on a single UDP port (the server's well-known port).

Normally there is a limited size input queue associated with each UDP port that an application is using. This means that requests that arrive at about the same time from different clients are automatically queued by UDP. The received UDP datagrams are passed to the application (when it asks for the next one) in the order they were received.

It is possible, however, for this queue to overflow, causing the kernel's UDP module to discard incoming datagrams. We can see this with the following experiment. We start our `sock` program on the host `bsdi` running as a UDP server:

```
bsdi % sock -s -u -v -E -R256 -r256 -P30 6666
from 140.252.13.33, to 140.252.13.63: 1111111111          from sun, to broadcast address
from 140.252.13.34, to 140.252.13.35: 4444444444444       from svr4, to unicast address
```

We specify the following flags: `-s` to run as a server, `-u` for UDP, `-v` to print the client's IP address, and `-E` to print the destination IP address (which is supported by this system). Additionally we set the UDP receive buffer for this port to 256 bytes (`-R`), along with the size of each application read (`-r`). The flag `-P30` tells it to pause for 30 seconds after creating the UDP port, before reading the first datagram. This gives us time to start the clients on two other hosts, send some datagrams, and see how the receive queueing works.

Once the server is started, and is in its 30-second pause, we start one client on the host `sun` and send three datagrams:

```
sun % sock -u -v 140.252.13.63 6666          to Ethernet broadcast address
connected on 140.252.13.33.1252 to 140.252.13.63.6666
1111111111                    11 bytes of data (with newline)
222222222                     10 bytes of data (with newline)
33333333333                   12 bytes of data (with newline)
```

The destination address is the broadcast address (140.252.13.63). We also start a second client on the host `svr4` and send another three datagrams:

The gateway item is an example of a variable-length item. The length is always a multiple of 4, and the values are the 32-bit IP addresses of one or more gateways (routers) for the client to use. The first one returned must be the preferred gateway.

There are 14 other items defined in RFC 1533. Probably the most important is the IP address of a DNS name server, with a tag value of 6. Other items return the IP address of a printer server, the IP address of a time server, and so on. Refer to the RFC for all the details.

Returning to our example in Figure 16.3, we never saw an ICMP address mask request (Section 6.3) that would have been broadcast by the client to find its subnet mask. Although it wasn't output by `tcpdump`, we can probably assume that the client's subnet mask was returned in the vendor-specific area of the BOOTP reply.

> The Host Requirements RFC recommends that a system using BOOTP obtain its subnet mask using BOOTP, not ICMP.

The size of the vendor-specific area is limited to 64 bytes. This is a constraint for some applications. A new protocol named DHCP (Dynamic Host Configuration Protocol) is built on, but replaces, BOOTP. DHCP extends this area to 312 bytes and is defined in RFC 1541 [Droms 1993].

16.7 Summary

BOOTP uses UDP and is intended as an alternative to RARP for bootstrapping a diskless system to find its IP address. BOOTP can also return additional information, such as the IP address of a router, the client's subnet mask, and the IP address of a name server.

Since BOOTP is used in the bootstrap process, a diskless system needs the following protocols implemented in read-only memory: BOOTP, TFTP, UDP, IP, and a device driver for the local network.

The implementation of a BOOTP server is easier than an RARP server, since BOOTP requests and replies are in UDP datagrams, not special link-layer frames. A router can also serve as a proxy agent for a real BOOTP server, forwarding client requests to the real server on a different network.

Exercises

16.1 We've said that one advantage of BOOTP over RARP is that BOOTP can work through routers, whereas RARP, which is a link-layer broadcast, cannot. Yet in Section 16.5 we had to define special ways for BOOTP to work through a router. What would happen if a capability were added to routers allowing them to forward RARP requests?

16.2 We said that a BOOTP client must use the transaction ID to match responses with requests, in case there are multiple clients bootstrapping at the same time from a server that broadcasts replies. But in Figure 16.3 the transaction ID is 0, implying that this client ignores the transaction ID. How do you think this client matches the responses with its requests?

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.

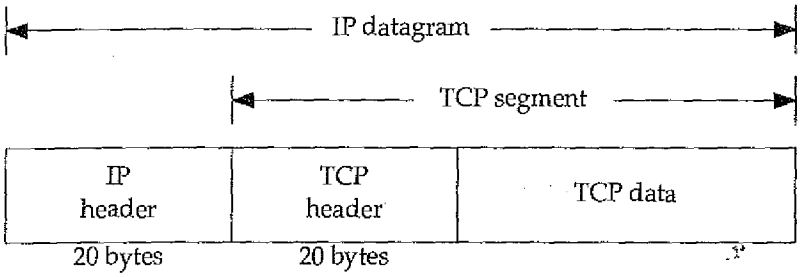


Figure 17.1 Encapsulation of TCP data in an IP datagram.

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

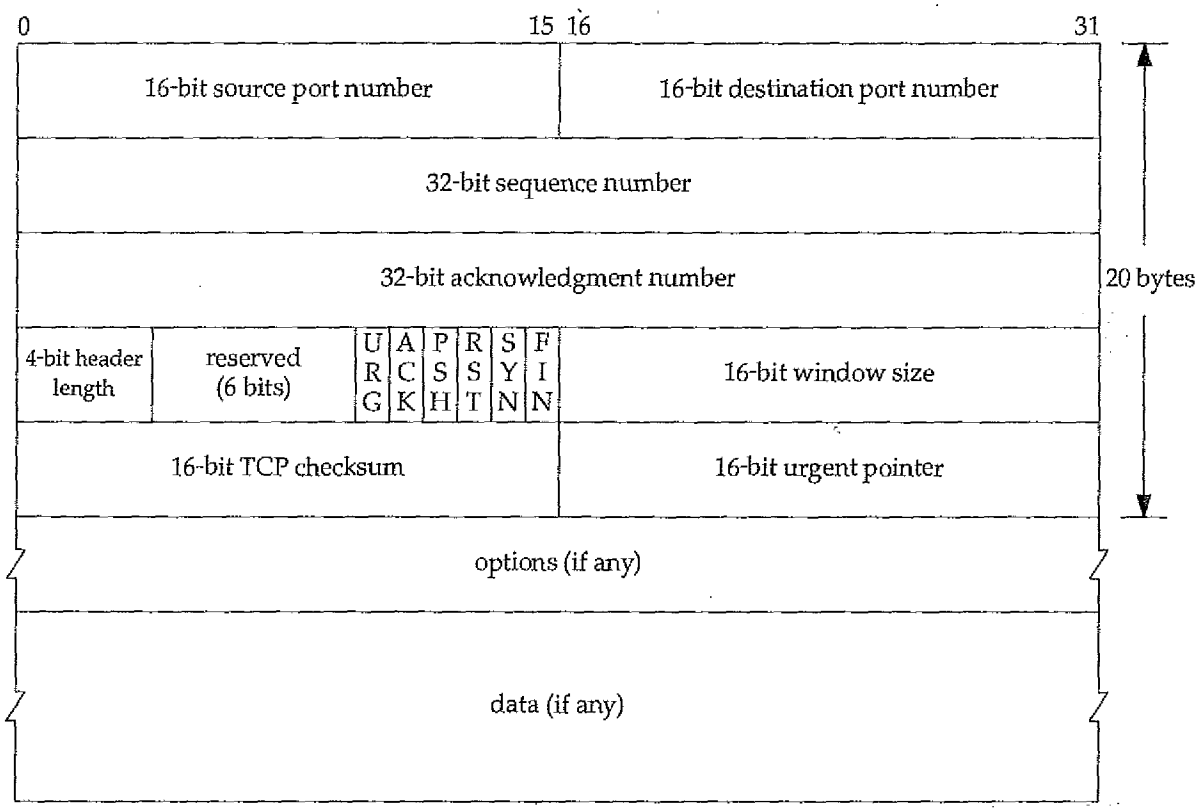


Figure 17.2 TCP header.

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.

TCP packetizes the user data into segments, sets a timeout any time it sends data, acknowledges data received by the other end, reorders out-of-order data, discards duplicate data, provides end-to-end flow control, and calculates and verifies a mandatory end-to-end checksum.

TCP is used by many of the popular applications, such as Telnet, Rlogin, FTP, and electronic mail (SMTP).

Exercises

17.1 We've covered the following packet formats, each of which has a checksum in its corresponding header: IP, ICMP, IGMP, UDP, and TCP. For each one, describe what portion of an IP datagram the checksum covers and whether the checksum is mandatory or optional.

17.2 Why do all the Internet protocols that we've discussed (IP, ICMP, IGMP, UDP, TCP) quietly discard a packet that arrives with a checksum error?

17.3 TCP provides a byte-stream service where record boundaries are not maintained between the sender and receiver. How can applications provide their own record markers?

17.4 Why are the source and destination port numbers at the beginning of the TCP header?

17.5 Why does the TCP header have a header length field while the UDP header (Figure 11.2, p. 144) does not?

18

TCP Connection Establishment and Termination

18.1 Introduction

TCP is a *connection-oriented* protocol. Before either end can send data to the other, a *connection* must be established between them. In this chapter we take a detailed look at how a TCP connection is established and later terminated.

This establishment of a connection between the two ends differs from a *connectionless* protocol such as UDP. We saw in Chapter 11 that with UDP one end just sends a datagram to the other end, without any preliminary handshaking.

18.2 Connection Establishment and Termination

To see what happens when a TCP connection is established and then terminated, we type the following command on the system `svr4`:

```
svr4 % telnet bsdi discard
Trying 140.252.13.35 ...
Connected to bsdi.
Escape character is '^]'.
^]                                  type Control, right bracket to talk to the Telnet client
telnet> quit                        terminate the connection
Connection closed.
```

The `telnet` command establishes a TCP connection with the host `bsdi` on the port corresponding to the discard service (Section 1.12). This is exactly the type of service we need to see what happens when a connection is established and terminated, without having the server initiate any data exchange.

tcpdump Output

Figure 18.1 shows the tcpdump output for the segments generated by this command.

```
1  0.0                  svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                  win 4096 <mss 1024>

2  0.002402 (0.0024)    bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                  ack 1415531522 win 4096
                                                  <mss 1024>

3  0.007224 (0.0048)    svr4.1037 > bsdi.discard: . ack 1823083522 win 4096

4  4.155441 (4.1482)    svr4.1037 > bsdi.discard: F 1415531522:1415531522(0)
                                                  ack 1823083522 win 4096

5  4.156747 (0.0013)    bsdi.discard > svr4.1037: . ack 1415531523 win 4096

6  4.158144 (0.0014)    bsdi.discard > svr4.1037: F 1823083522:1823083522(0)
                                                  ack 1415531523 win 4096

7  4.180662 (0.0225)    svr4.1037 > bsdi.discard: . ack 1823083523 win 4096
```

Figure 18.1 tcpdump output for TCP connection establishment and termination.

These seven TCP segments contain TCP headers only. No data is exchanged.

For TCP segments, each output line begins with

source > *destination: flags*

where *flags* represents four of the six flag bits in the TCP header (Figure 17.2). Figure 18.2 shows the five different characters that can appear in the *flags* output.

flag	3-character abbreviation	Description
S	SYN	synchronize sequence numbers
F	FIN	sender is finished sending data
R	RST	reset connection
P	PSH	push data to receiving process as soon as possible
.		none of above four flags is on

Figure 18.2 *flag* characters output by tcpdump for flag bits in TCP header.

In this example we see the S, F, and period. We'll see the other two *flags* (R and P) later. The other two TCP header flag bits—ACK and URG—are printed specially by tcpdump.

It's possible for more than one of the four flag bits in Figure 18.2 to be on in a single segment, but we normally see only one on at a time.

RFC 1025 [Postel 1987], the *TCP and IP Bake Off*, calls a segment with the maximum combination of allowable flag bits turned on at once (SYN, URG, PSH, FIN, and 1 byte of data) a Kamikaze packet. It's also known as a nastygram, Christmas tree packet, and lamp test segment.

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*.

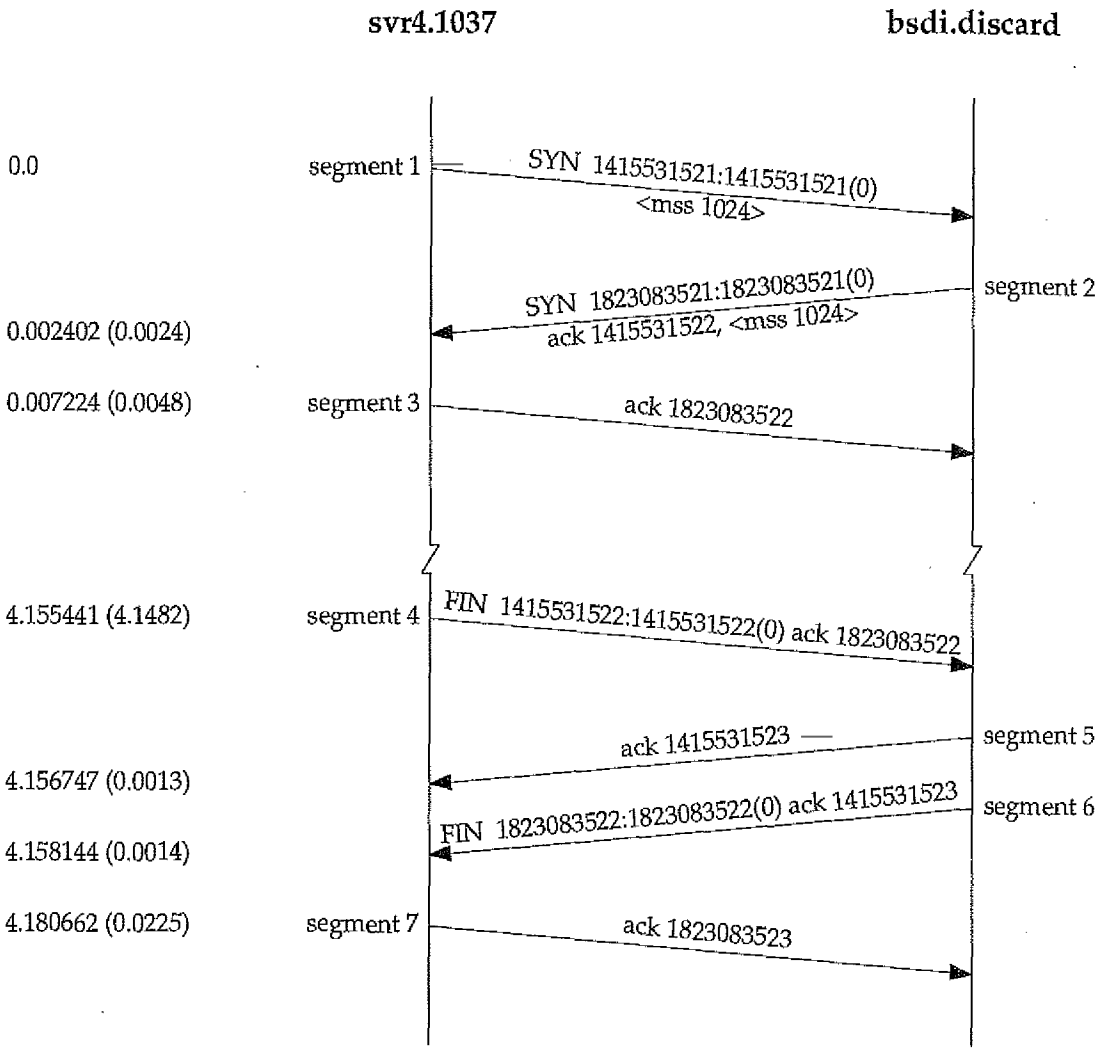


Figure 18.3 Time line of connection establishment and connection termination.

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

microseconds, not every 4 microseconds.) Additionally, each time a connection is established, this variable is incremented by 64,000.

The 4.1-second gap between segments 3 and 4 is the time between establishing the connection and typing the `quit` command to `telnet` to terminate the connection.

Connection Termination Protocol

While it takes three segments to establish a connection, it takes four to terminate a connection. This is caused by TCP's *half-close*. Since a TCP connection is full-duplex (that is, data can be flowing in each direction independently of the other direction), each direction must be shut down independently. The rule is that either end can send a FIN when it is done sending data. When a TCP receives a FIN, it must notify the application that the other end has terminated that direction of data flow. The sending of a FIN is normally the result of the application issuing a close.

The receipt of a FIN only means there will be no more data flowing in that direction. A TCP can still send data after receiving a FIN. While it's possible for an application to take advantage of this half-close, in practice few TCP applications use it. The normal scenario is what we show in Figure 18.3. We describe the half-close in more detail in Section 18.5.

We say that the end that first issues the close (e.g., sends the first FIN) performs the *active close* and the other end (that receives this FIN) performs the *passive close*. Normally one end does the active close and the other does the passive close, but we'll see in Section 18.9 how both ends can do an active close.

Segment 4 in Figure 18.3 initiates the termination of the connection and is sent when the Telnet client closes its connection. This happens when we type `quit`. This causes the client TCP to send a FIN, closing the flow of data from the client to the server.

When the server receives the FIN it sends back an ACK of the received sequence number plus one (segment 5). A FIN consumes a sequence number, just like a SYN. At this point the server's TCP also delivers an end-of-file to the application (the discard server). The server then closes its connection, causing its TCP to send a FIN (segment 6), which the client TCP must ACK by incrementing the received sequence number by one (segment 7).

Figure 18.4 shows the typical sequence of segments that we've described for the termination of a connection. We omit the sequence numbers. In this figure sending the FINs is caused by the applications closing their end of the connection, whereas the ACKs of these FINs are automatically generated by the TCP software.

Connections are normally initiated by the client, with the first SYN going from the client to the server. Either end can actively close the connection (i.e., send the first FIN). Often, however, it is the client that determines when the connection should be terminated, since client processes are often driven by an interactive user, who enters something like "quit" to terminate. In Figure 18.4 we can switch the labels at the top, calling the left side the server and the right side the client, and everything still works fine as shown. (The first example in Section 14.4, for example, shows the daytime server closing the connection.)

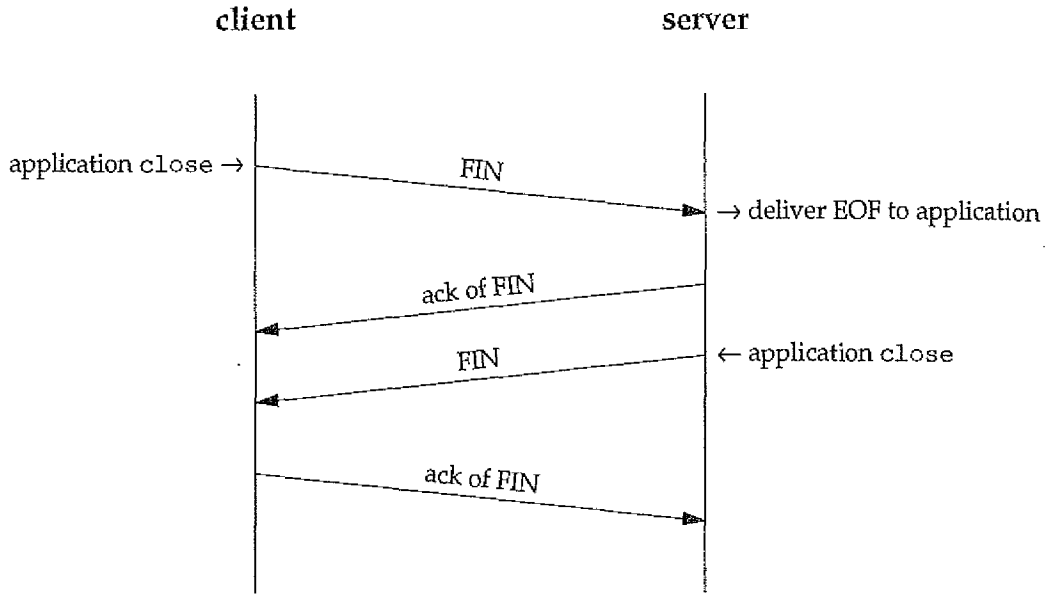


Figure 18.4 Normal exchange of segments during connection termination.

Normal tcpdump Output

Having to sort through all the huge sequence numbers is cumbersome, so the default tcpdump output shows the complete sequence numbers only on the SYN segments, and shows all following sequence numbers as relative offsets from the original sequence numbers. (To generate the output for Figure 18.1 we had to specify the -S option.) The normal tcpdump output corresponding to Figure 18.1 is shown in Figure 18.5.

```
1  0.0                  svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                    win 4096 <mss 1024>
2  0.002402 (0.0024)    bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                    ack 1415531522
                                                    win 4096 <mss 1024>
3  0.007224 (0.0048)    svr4.1037 > bsdi.discard: . ack 1 win 4096
4  4.155441 (4.1482)    svr4.1037 > bsdi.discard: F 1:1(0) ack 1 win 4096
5  4.156747 (0.0013)    bsdi.discard > svr4.1037: . ack 2 win 4096
6  4.158144 (0.0014)    bsdi.discard > svr4.1037: F 1:1(0) ack 2 win 4096
7  4.180662 (0.0225)    svr4.1037 > bsdi.discard: . ack 2 win 4096
```

Figure 18.5 Normal tcpdump output for connection establishment and termination.

Unless we need to show the complete sequence numbers, we'll use this form of output in all following examples.

18.3 Timeout of Connection Establishment

There are several instances when the connection cannot be established. In one example the server host is down. To simulate this scenario we issue our `telnet` command after disconnecting the Ethernet cable from the server's host. Figure 18.6 shows the `tcpdump` output.

```
1   0.0                  bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
2   5.814797 ( 5.8148)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
3  29.815436 (24.0006)   bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                    win 4096 <mss 1024>
                                                    [tos 0x10]
```

Figure 18.6 `tcpdump` output for connection establishment that times out.

The interesting point in this output is how frequently the client's TCP sends a SYN to try to establish the connection. The second segment is sent 5.8 seconds after the first, and the third is sent 24 seconds after the second.

> As a side note, this example was run about 38 minutes after the client was rebooted. This corresponds with the initial sequence number of 291,008,001 (approximately $38 \times 60 \times 64000 \times 2$). Recall earlier in this chapter we said that typical Berkeley-derived systems initialize the initial sequence number to 1 and then increment it by 64,000 every half-second.
>
> Also, this is the first TCP connection since the system was bootstrapped, which is why the client's port number is 1024.

What isn't shown in Figure 18.6 is how long the client's TCP keeps retransmitting before giving up. To see this we have to time the `telnet` command:

```
bsdi % date ; telnet svr4 discard ; date
Thu Sep 24 16:24:11 MST 1992
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection timed out
Thu Sep 24 16:25:27 MST 1992
```

The time difference is 76 seconds. Most Berkeley-derived systems set a time limit of 75 seconds on the establishment of a new connection. We'll see in Section 21.4 that the third packet sent by the client would have timed out around 16:25:29, 48 seconds after it was sent, had the client not given up after 75 seconds.

First Timeout Period

One puzzling item in Figure 18.6 is that the first timeout period, 5.8 seconds, is close to 6 seconds, but not exact, while the second period is almost exactly 24 seconds. Ten more

of these tests were run and the first timeout period took on various values between 5.59 seconds and 5.93 seconds. The second timeout period, however, was always 24.00 (to two decimal places).

What's happening here is that BSD implementations of TCP run a timer that goes off every 500 ms. This 500-ms timer is used for various TCP timeouts, all of which we cover in later chapters. When we type in the `telnet` command, an initial 6-second timer is established (12 clock ticks), but it may expire anywhere between 5.5 and 6 seconds in the future. Figure 18.7 shows what's happening.

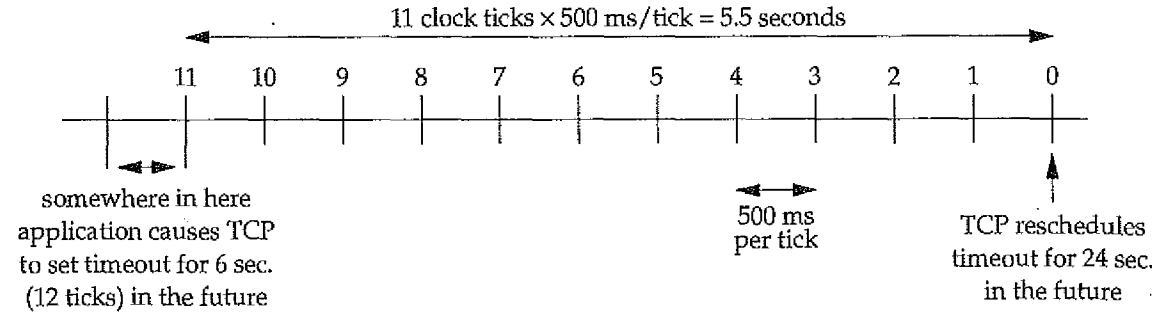


Figure 18.7 TCP 500-ms timer.

Although the timer is initialized to 12 ticks, the first decrement of the timer can occur between 0 and 500 ms after it is set. From that point on the timer is decremented about every 500 ms, but the first period can be variable. (We use the qualifier "about" because the time when TCP gets control every 500 ms can be preempted by other interrupts being handled by the kernel.)

When that 6-second timer expires at the tick labeled 0 in Figure 18.7, the timer is reset for 24 seconds (48 ticks) in the future. This next timer will be close to 24 seconds, since it was set at a time when the TCP's 500-ms timer handler was called by the kernel.

Type-of-Service Field

In Figure 18.6, the notation `[tos 0x10]` appears. This is the type-of-service (TOS) field in the IP datagram (Figure 3.2). The BSD/386 Telnet client sets the field for minimum delay.

18.4 Maximum Segment Size

The maximum segment size (MSS) is the largest "chunk" of data that TCP will send to the other end. When a connection is established, each end can announce its MSS. The values we've seen have all been 1024. The resulting IP datagram is normally 40 bytes larger: 20 bytes for the TCP header and 20 bytes for the IP header.

Some texts refer to this as a "negotiated" option. It is not negotiated in any way. When a connection is established, each end has the option of announcing the MSS it

expects to receive. (An MSS option can only appear in a SYN segment.) If one end does not receive an MSS option from the other end, a default of 536 bytes is assumed. (This default allows for a 20-byte IP header and a 20-byte TCP header to fit into a 576-byte IP datagram.)

In general, the larger the MSS the better, until fragmentation occurs. (This may not always be true. See Figures 24.3 and 24.4 for a counterexample.) A larger segment size allows more data to be sent in each segment, amortizing the cost of the IP and TCP headers. When TCP sends a SYN segment, either because a local application wants to initiate a connection, or when a connection request is received from another host, it can send an MSS value up to the outgoing interface's MTU, minus the size of the fixed TCP and IP headers. For an Ethernet this implies an MSS of up to 1460 bytes. Using IEEE 802.3 encapsulation (Section 2.2), the MSS could go up to 1452 bytes.

The values of 1024 that we've seen in this chapter, for connections involving BSD/386 and SVR4, are because many BSD implementations require the MSS to be a multiple of 512. Other systems, such as SunOS 4.1.3, Solaris 2.2, and AIX 3.2.2, all announce an MSS of 1460 when both ends are on a local Ethernet. Measurements in [Mogul 1993] show how an MSS of 1460 provides better performance on an Ethernet than an MSS of 1024.

If the destination IP address is "nonlocal," the MSS normally defaults to 536. While it's easy to say that a destination whose IP address has the same network ID and the same subnet ID as ours is local, and a destination whose IP address has a totally different network ID from ours is nonlocal, a destination with the same network ID but a different subnet ID could be either local or nonlocal. Most implementations provide a configuration option (Appendix E and Figure E.1) that lets the system administrator specify whether different subnets are local or nonlocal. The setting of this option determines whether the announced MSS is as large as possible (up to the outgoing interface's MTU) or the default of 536.

The MSS lets a host limit the size of datagrams that the other end sends it. When combined with the fact that a host can also limit the size of the datagrams that it sends, this lets a host avoid fragmentation when the host is connected to a network with a small MTU.

Consider our host `slip`, which has a SLIP link with an MTU of 296 to the router `bsdi`. Figure 18.8 shows these systems and the host `sun`.

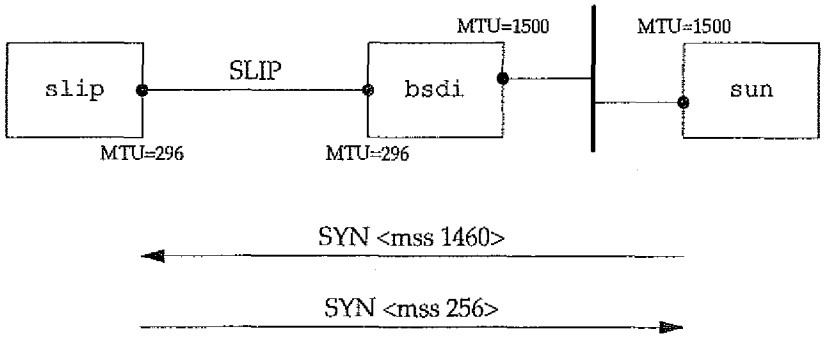


Figure 18.8 TCP connection from `sun` to `slip` showing MSS values.

We initiate a TCP connection from sun to slip and watch the segments using tcpdump. Figure 18.9 shows only the connection establishment (with the window size advertisements removed).

```
1  0.0                  sun.1093 > slip.discard: S 517312000:517312000(0)
                                                 <mss 1460>
2  0.10 (0.00)          slip.discard > sun.1093: S 509556225:509556225(0)
                                                 ack 517312001 <mss 256>
3  0.10 (0.00)          sun.1093 > slip.discard: . ack 1
```

Figure 18.9 tcpdump output for connection establishment from sun to slip.

The important fact here is that sun cannot send a segment with more than 256 bytes of data, since it received an MSS option of 256 (line 2). Furthermore, since slip knows that the outgoing interface's MTU is 296, even though sun announced an MSS of 1460, it will never send more than 256 bytes of data, to avoid fragmentation. It's OK for a system to send *less* than the MSS announced by the other end.

This avoidance of fragmentation works only if either host is directly connected to a network with an MTU of less than 576. If both hosts are connected to Ethernets, and both announce an MSS of 536, but an intermediate network has an MTU of 296, fragmentation will occur. The only way around this is to use the path MTU discovery mechanism (Section 24.2).

18.5 TCP Half-Close

TCP provides the ability for one end of a connection to terminate its output, while still receiving data from the other end. This is called a *half-close*. Few applications take advantage of this capability, as we mentioned earlier.

To use this feature the programming interface must provide a way for the application to say "I am done sending data, so send an end-of-file (FIN) to the other end, but I still want to receive data from the other end, until it sends me an end-of-file (FIN)."

> The sockets API supports the half-close, if the application calls shutdown with a second argument of 1, instead of calling close. Most applications, however, terminate both directions of the connection by calling close.

Figure 18.10 shows a typical scenario for a half-close. We show the client on the left side initiating the half-close, but either end can do this. The first two segments are the same: a FIN by the initiator, followed by an ACK of the FIN by the recipient. But it then changes from Figure 18.4, because the side that receives the half-close can still send data. We show only one data segment, followed by an ACK, but any number of data segments can be sent. (We talk more about the exchange of data segments and acknowledgments in Chapter 19.) When the end that received the half-close is done sending data, it closes its end of the connection, causing a FIN to be sent, and this delivers an end-of-file to the application that initiated the half-close. When this second FIN is acknowledged, the connection is completely closed.

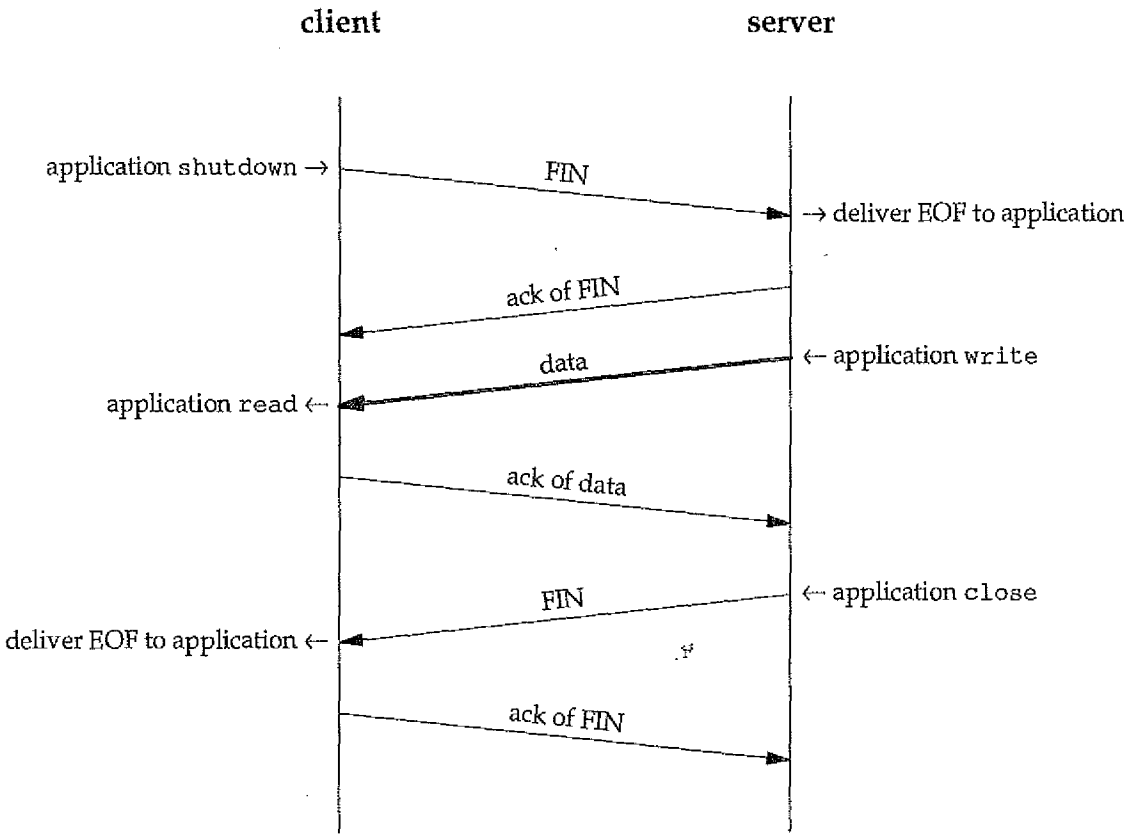


Figure 18.10 Example of TCP's half-close.

Why is there a half-close? One example is the Unix `rsh`(1) command, which executes a command on another system. The command

```
sun % rsh bsdi sort < datafile
```

executes the `sort` command on the host `bsdi` with standard input for the `rsh` command being read from the file named `datafile`. A TCP connection is created by `rsh` between itself and the program being executed on the other host. The operation of `rsh` is then simple: it copies standard input (`datafile`) to the connection, and copies from the connection to standard output (our terminal). Figure 18.11 shows the setup. (Remember that a TCP connection is full-duplex.)

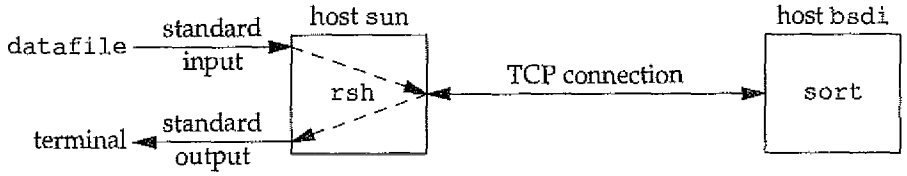


Figure 18.11 The command: `rsh bsdi sort < datafile`.

On the remote host `bsdi` the `rshd` server executes the `sort` program so that its standard input and standard output are both the TCP connection. Chapter 14 of [Stevens 1990] details the Unix process structure involved, but what concerns us here is the use of the TCP connection and the required use of TCP's half-close.

The `sort` program cannot generate any output until all of its input has been read. All the initial data across the connection is from the `rsh` client to the `sort` server, sending the file to be sorted. When the end-of-file is reached on the input (`datafile`), the `rsh` client performs a half-close on the TCP connection. The `sort` server then receives an end-of-file on its standard input (the TCP connection), sorts the file, and writes the result to its standard output (the TCP connection). The `rsh` client continues reading its end of the TCP connection, copying the sorted file to its standard output.

Without a half-close, some other technique is needed to let the client tell the server that the client is finished sending data, but still let the client receive data from the server. Two connections could be used as an alternative, but a single connection with a half-close is better.

18.6 TCP State Transition Diagram

We've described numerous rules regarding the initiation and termination of a TCP connection. These rules can be summarized in a state transition diagram, which we show in Figure 18.12.

The first thing to note in this diagram is that a subset of the state transitions is "typical." We've marked the normal client transitions with a darker solid arrow, and the normal server transitions with a darker dashed arrow.

Next, the two transitions leading to the ESTABLISHED state correspond to opening a connection, and the two transitions leading from the ESTABLISHED state are for the termination of a connection. The ESTABLISHED state is where data transfer can occur between the two ends in both directions. Later chapters describe what happens in this state.

We've collected the four boxes in the lower left of this diagram within a dashed box and labeled it "active close." Two other boxes (CLOSE_WAIT and LAST_ACK) are collected in a dashed box with the label "passive close."

The names of the 11 states (CLOSED, LISTEN, SYN_SENT, etc.) in this figure were purposely chosen to be identical to the states output by the `netstat` command. The `netstat` names, in turn, are almost identical to the names originally described in RFC 793. The state CLOSED is not really a state, but is the imaginary starting point and ending point for the diagram.

The state transition from LISTEN to SYN_SENT is legal but is not supported in Berkeley-derived implementations.

The transition from SYN_RCVD back to LISTEN is valid only if the SYN_RCVD state was entered from the LISTEN state (the normal scenario), not from the SYN_SENT state (a simultaneous open). This means if we perform a passive open (enter LISTEN), receive a SYN, send a SYN with an ACK (enter SYN_RCVD), and then receive a reset instead of an ACK, the end point returns to the LISTEN state and waits for another connection request to arrive.

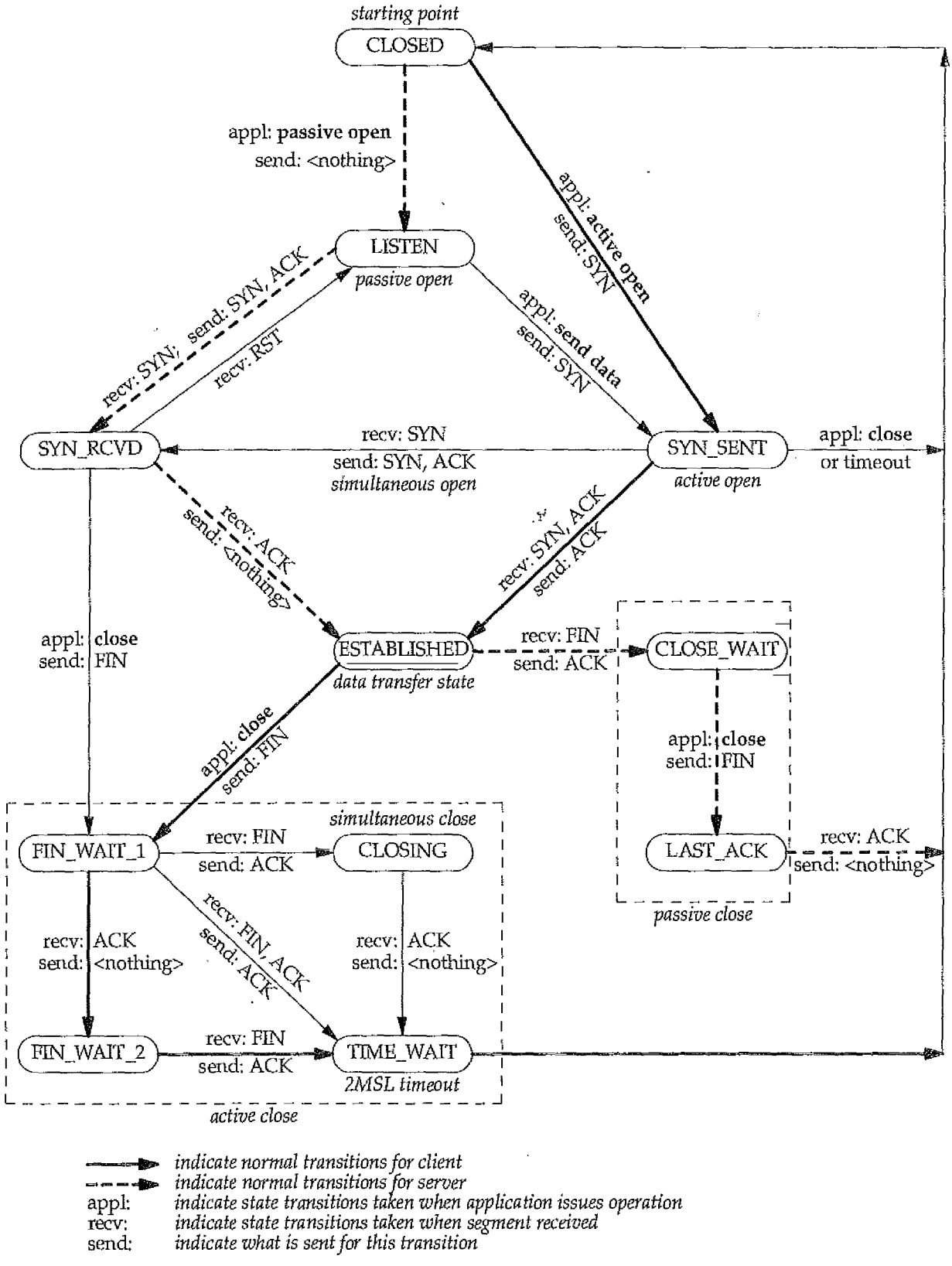


Figure 18.12 TCP state transition diagram.

Figure 18.13 shows the normal TCP connection establishment and termination, detailing the different states through which the client and server pass. It is a redo of Figure 18.3 showing only the states.

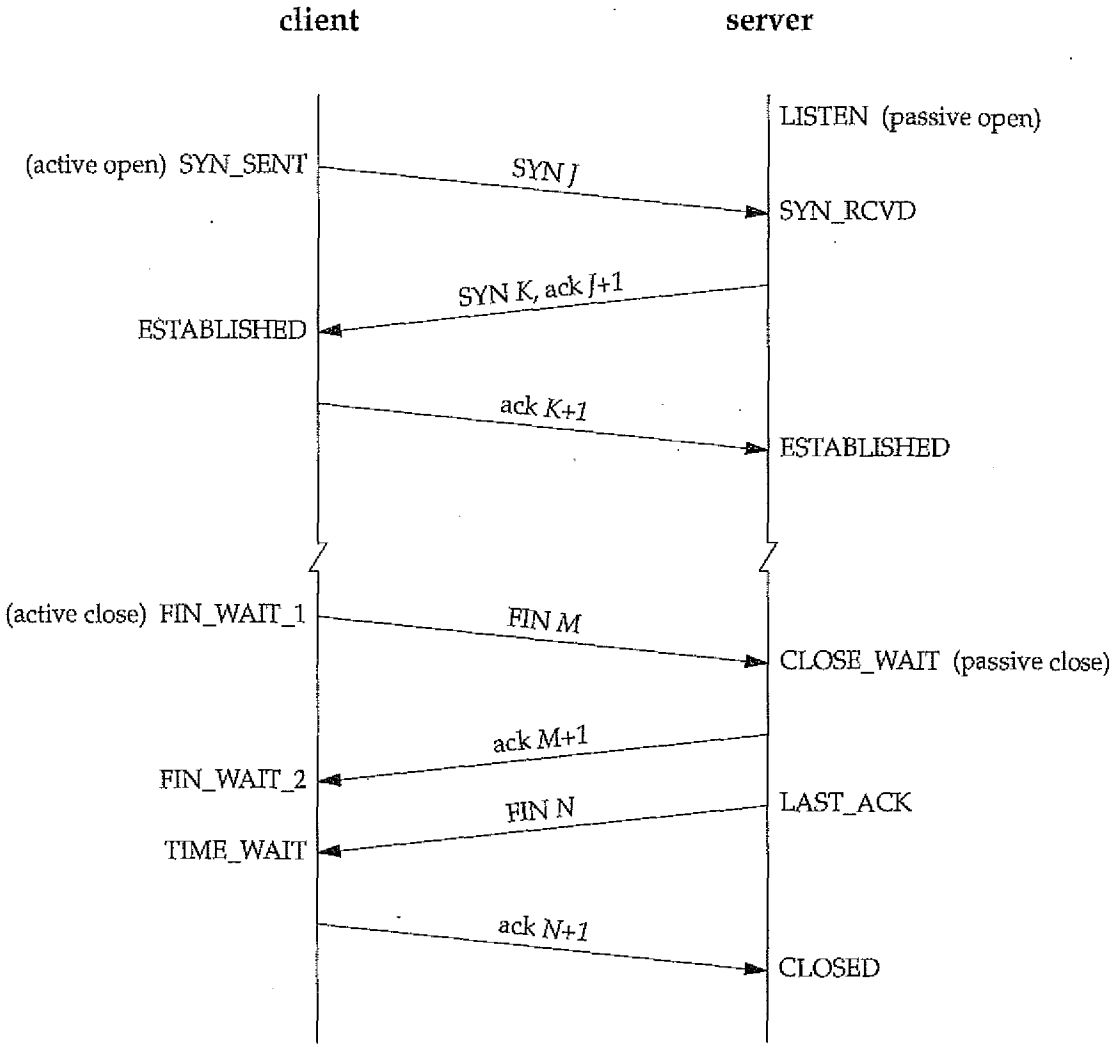


Figure 18.13 TCP states corresponding to normal connection establishment and termination.

We assume in Figure 18.13 that the client on the left side does an active open, and the server on the right side does a passive open. Although we show the client doing the active close, as we mentioned earlier, either side can do the active close.

You should follow through the state changes in Figure 18.13 using the state transition diagram in Figure 18.12, making certain you understand why each state change takes place.

2MSL Wait State

The TIME_WAIT state is also called the 2MSL wait state. Every implementation must choose a value for the *maximum segment lifetime* (MSL). It is the maximum amount of

time any segment can exist in the network before being discarded. We know this time limit is bounded, since TCP segments are transmitted as IP datagrams, and the IP datagram has the TTL field that limits its lifetime.

> RFC 793 [Postel 1981c] specifies the MSL as 2 minutes. Common implementation values, however, are 30 seconds, 1 minute, or 2 minutes.

Recall from Chapter 8 that the real-world limit on the lifetime of the IP datagram is based on the number of hops, not a timer.

Given the MSL value for an implementation, the rule is: when TCP performs an active close, and sends the final ACK, that connection must stay in the TIME_WAIT state for twice the MSL. This lets TCP resend the final ACK in case this ACK is lost (in which case the other end will time out and retransmit its final FIN).

Another effect of this 2MSL wait is that while the TCP connection is in the 2MSL wait, the socket pair defining that connection (client IP address, client port number, server IP address, and server port number) cannot be reused. That connection can only be reused when the 2MSL wait is over.

Unfortunately most implementations (i.e., the Berkeley-derived ones) impose a more stringent constraint. By default a local port number cannot be reused while that port number is the local port number of a socket pair that is in the 2MSL wait. We'll see examples of this common constraint below.

> Some implementations and APIs provide a way to bypass this restriction. With the sockets API, the `SO_REUSEADDR` socket option can be specified. It lets the caller assign itself a local port number that's in the 2MSL wait, but we'll see that the rules of TCP still prevent this port number from being part of a connection that is in the 2MSL wait.

Any delayed segments that arrive for a connection while it is in the 2MSL wait are discarded. Since the connection defined by the socket pair in the 2MSL wait cannot be reused during this time period, when we do establish a valid connection we know that delayed segments from an earlier incarnation of this connection cannot be misinterpreted as being part of the new connection. (A connection is defined by a socket pair. New instances of a connection are called *incarnations* of that connection.)

As we said with Figure 18.13, it is normally the client that does the active close and enters the TIME_WAIT state. The server usually does the passive close, and does not go through the TIME_WAIT state. The implication is that if we terminate a client, and restart the same client immediately, that new client cannot reuse the same local port number. This isn't a problem, since clients normally use ephemeral ports, and don't care what the local ephemeral port number is.

With servers, however, this changes, since servers use well-known ports. If we terminate a server that has a connection established, and immediately try to restart the server, the server cannot assign its well-known port number to its end point, since that port number is part of a connection that is in a 2MSL wait. It may take from 1 to 4 minutes before the server can be restarted.

We can see this scenario using our `sock` program. We start the server, connect to it from a client, and then terminate the server:

```
sun % sock -v -s 6666                    start as server, listening on port 6666
                                         (execute client on bsdi that connects to this port)
connection on 140.252.13.33.6666 from 140.252.13.35.1081
^?                                       then type interrupt key to terminate server

sun % sock -s 6666                       and immediately try to restart server on same port
can't bind local address: Address already in use

sun % netstat                            let's check the state of the connection
Active Internet connections
Proto Recv-Q Send-Q  Local Address   Foreign Address     (state)
tcp        0      0  sun.6666        bsdi.1081          TIME_WAIT
                                         many more lines that are deleted
```

When we try to restart the server, the program outputs an error message indicating it cannot bind its well-known port number, because it's already in use (i.e., it's in a 2MSL wait).

We then immediately execute `netstat` to see the state of the connection, and verify that it is indeed in the TIME_WAIT state.

> If we continually try to restart the server, and measure the time until it succeeds, we can measure the 2MSL value. On SunOS 4.1.3, SVR4, BSD/386, and AIX 3.2.2, it takes 1 minute to restart the server, meaning the MSL is 30 seconds. Under Solaris 2.2 it takes 4 minutes to restart the server, implying an MSL of 2 minutes.

We can see the same error from a client, if the client tries to allocate a port that is part of a connection in the 2MSL wait (something clients normally don't do):

```
sun % sock -v bsdi echo                  start as client, connect to echo server
connected on 140.252.13.33.1162 to 140.252.13.35.7
hello there                              type this line
hello there                              and it's echoed by the server
^D                                       type end-of-file character to terminate client

sun % sock -b1162 bsdi echo
can't bind local address: Address already in use
```

The first time we execute the client we specify the `-v` option to see what the local port number is (1162). The second time we execute the client we specify the `-b` option, telling the client to assign itself 1162 as its local port number. As we expect, the client can't do this, since that port number is part of a connection that is in a 2MSL wait.

We need to reemphasize one effect of the 2MSL wait because we'll encounter it in Chapter 27 with FTP, the File Transfer Protocol. As we said earlier, it is a socket pair (that is, the 4-tuple consisting of a local IP address, local port, remote IP address and remote port) that remains in the 2MSL wait. Although many implementations allow a process to reuse a port number that is part of a connection that is in the 2MSL wait (normally with an option named `SO_REUSEADDR`), TCP *cannot* allow a new connection to be created with the same socket pair. We can see this with the following experiment:

```
sun % sock -v -s 6666                    start as server, listening on port 6666
                                         (execute client on bsdi that connects to this port)
```

```
connection on 140.252.13.33.6666 from 140.252.13.35.1098
^?                                        then type interrupt key to terminate server

sun % sock -b6666 bsdi 1098               try to start as client with local port 6666
can't bind local address: Address already in use

sun % sock -A -b6666 bsdi 1098            try again, this time with -A option
active open error: Address already in use
```

The first time we run our `sock` program, we run it as a server on port 6666 and connect to it from a client on the host `bsdi`. The client's ephemeral port number is 1098. We terminate the server so it does the active close. This causes the 4-tuple of 140.252.13.33 (local IP address), 6666 (local port number), 140.252.13.35 (foreign IP address), and 1098 (foreign port number) to enter the 2MSL wait on the server host.

The second time we run the program, we run it as a client and try to specify the local port number as 6666 and connect to host `bsdi` on port 1098. But the program gets an error when it tries to assign itself the local port number of 6666, because that port number is part of the 4-tuple that is in the 2MSL wait state.

To try and get around this error we run the program again, specifying the `-A` option, which enables the `SO_REUSEADDR` option that we mentioned. This lets the program assign itself the port number 6666, but we then get an error when it tries to issue the active open. Even though it can assign itself the port number 6666, it cannot create a connection to port 1098 on the host `bsdi`, because the socket pair defining that connection is in the 2MSL wait state.

What if we try to establish the connection from the other host? First we must restart the server on `sun` with the `-A` flag, since the local port it needs (6666) is part of a connection that is in the 2MSL wait:

```
sun % sock -A -s 6666                     start as server, listening on port 6666
```

Then, before the 2MSL wait is over on `sun`, we start the client on `bsdi`:

```
bsdi % sock -b1098 sun 6666
connected on 140.252.13.35.1098 to 140.252.13.33.6666
```

Unfortunately it works! This is a violation of the TCP specification, but is supported by most Berkeley-derived implementations. These implementations allow a new connection request to arrive for a connection that is in the TIME_WAIT state, if the new sequence number is greater than the final sequence number from the previous incarnation of this connection. In this case the ISN for the new incarnation is set to the final sequence number from the previous incarnation plus 128,000. The appendix of RFC 1185 [Jacobson, Braden, and Zhang 1990] shows the pitfalls still possible with this technique.

This implementation feature lets a client and server continually reuse the same port number at each end for successive incarnations of the same connection, but only if the server does the active close. We'll see another example of this 2MSL wait condition in Figure 27.8, with FTP. See Exercise 18.5 also.

Quiet Time Concept

The 2MSL wait provides protection against delayed segments from an earlier incarnation of a connection from being interpreted as part of a new connection that uses the same local and foreign IP addresses and port numbers. But this works only if a host with connections in the 2MSL wait does not crash.

What if a host with ports in the 2MSL wait crashes, reboots within MSL seconds, and immediately establishes new connections using the same local and foreign IP addresses and port numbers corresponding to the local ports that were in the 2MSL wait before the crash? In this scenario, delayed segments from the connections that existed before the crash can be misinterpreted as belonging to the new connections created after the reboot. This can happen regardless of how the initial sequence number is chosen after the reboot.

To protect against this scenario, RFC 793 states that TCP should not create any connections for MSL seconds after rebooting. This is called the *quiet time*.

> Few implementations abide by this since most hosts take longer than MSL seconds to reboot after a crash.

FIN_WAIT_2 State

In the FIN_WAIT_2 state we have sent our FIN and the other end has acknowledged it. Unless we have done a half-close, we are waiting for the application on the other end to recognize that it has received an end-of-file notification and close its end of the connection, which sends us a FIN. Only when the process at the other end does this close will our end move from the FIN_WAIT_2 to the TIME_WAIT state.

This means our end of the connection can remain in this state forever. The other end is still in the CLOSE_WAIT state, and can remain there forever, until the application decides to issue its close.

> Many Berkeley-derived implementations prevent this infinite wait in the FIN_WAIT_2 state as follows. If the application that does the active close does a complete close, not a half-close indicating that it expects to receive data, then a timer is set. If the connection is idle for 10 minutes plus 75 seconds, TCP moves the connection into the CLOSED state. A comment in the code acknowledges that this implementation feature violates the protocol specification.

18.7 Reset Segments

We've mentioned a bit in the TCP header named RST for "reset." In general, a reset is sent by TCP whenever a segment arrives that doesn't appear correct for the referenced connection. (We use the term "referenced connection" to mean the connection specified by the destination IP address and port number, and the source IP address and port number. This is what RFC 793 calls a socket.)

Connection Request to Nonexistent Port

A common case for generating a reset is when a connection request arrives and no process is listening on the destination port. In the case of UDP, we saw in Section 6.5 that an ICMP port unreachable was generated when a datagram arrived for a destination port that was not in use. TCP uses a reset instead.

This example is trivial to generate—we use the Telnet client and specify a port number that's not in use on the destination:

```
bsdi % telnet svr4 20000              port 20000 should not be in use
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection refused
```

This error message is output by the Telnet client immediately. Figure 18.14 shows the packet exchange corresponding to this command.

```
1  0.0                    bsdi.1087 > svr4.20000: S 297416193:297416193(0)
                                                   win 4096 <mss 1024>
                                                   [tos 0x10]
2  0.003771 (0.0038)      svr4.20000 > bsdi.1087: R 0:0(0) ack 297416194 win 0
```

Figure 18.14 Reset generated by attempt to open connection to nonexistent port.

The values we need to examine in this figure are the sequence number field and acknowledgment number field in the reset. Because the ACK bit was not on in the arriving segment, the sequence number of the reset is set to 0 and the acknowledgment number is set to the incoming ISN plus the number of data bytes in the segment. Although there is no real data in the arriving segment, the SYN bit logically occupies 1 byte of sequence number space; therefore, in this example the acknowledgment number in the reset is set to the ISN, plus the data length (0), plus one for the SYN bit.

Aborting a Connection

We saw in Section 18.2 that the normal way to terminate a connection is for one side to send a FIN. This is sometimes called an *orderly release* since the FIN is sent after all previously queued data has been sent, and there is normally no loss of data. But it's also possible to abort a connection by sending a reset instead of a FIN. This is sometimes called an *abortive release*.

Aborting a connection provides two features to the application: (1) any queued data is thrown away and the reset is sent immediately, and (2) the receiver of the RST can tell that the other end did an abort instead of a normal close. The API being used by the application must provide a way to generate the abort instead of a normal close.

We can watch this abort sequence happen using our `sock` program. The sockets API provides this capability by using the "linger on close" socket option (`SO_LINGER`). We specify the `-L` option with a linger time of 0. This causes the abort to be sent when the connection is closed, instead of the normal FIN. We'll connect to a server version of our `sock` program on `svr4` and type one line of input:

```
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

(line 6 in Figure 18.16). This elicits a response from the server, but note that since the server was rebooted, its ARP cache is empty, so an ARP request and reply are required (lines 7 and 8). Then the reset is sent in line 9. The client receives the reset and outputs that the connection was terminated by the foreign host. (The final message output by the Telnet client is not as informative as it could be.)

18.8 Simultaneous Open

It is possible, although improbable, for two applications to both perform an active open to each other at the same time. Each end must transmit a SYN, and the SYNs must pass each other on the network. It also requires each end to have a local port number that is well known to the other end. This is called a *simultaneous open*.

For example, one application on host A could have a local port of 7777 and perform an active open to port 8888 on host B. The application on host B would have a local port of 8888 and perform an active open to port 7777 on host A.

This is *not* the same as connecting a Telnet client on host A to the Telnet server on host B, at the same time that a Telnet client on host B is connecting to the Telnet server on host A. In this Telnet scenario, both Telnet servers perform passive opens, not active opens, and the Telnet clients assign themselves an ephemeral port number, not a port number that is well known to the other Telnet server.

TCP was purposely designed to handle simultaneous opens and the rule is that only one connection results from this, not two connections. (Other protocol suites, notably the OSI transport layer, create two connections in this scenario, not one.)

When a simultaneous open occurs the state transitions differ from those shown in Figure 18.13. Both ends send a SYN at about the same time, entering the SYN_SENT state. When each end receives the SYN, the state changes to SYN_RCVD (Figure 18.12), and each end resends the SYN and acknowledges the received SYN. When each end receives the SYN plus the ACK, the state changes to ESTABLISHED. These state changes are summarized in Figure 18.17.

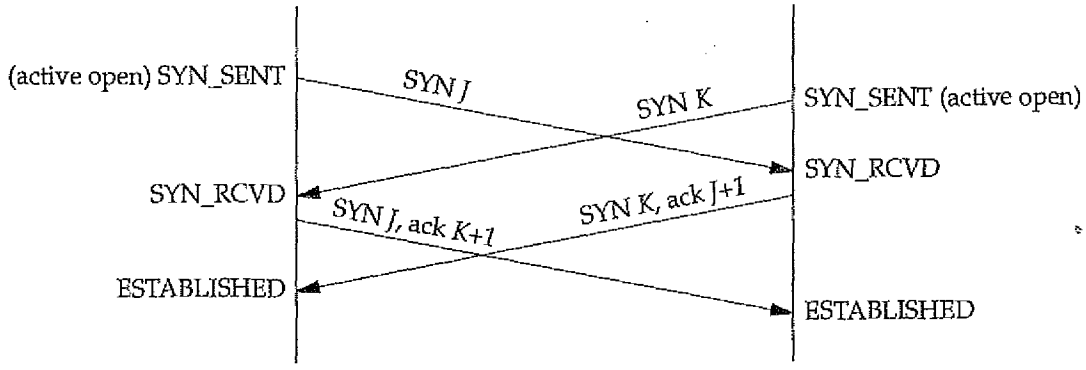


Figure 18.17 Segments exchanged during simultaneous open.

A simultaneous open requires the exchange of four segments, one more than the normal three-way handshake. Also notice that we don't call either end a client or a server, because both ends act as client and server.

An Example

It is possible, though hard, to generate a simultaneous open. The two ends must be started at about the same time, so that the SYNs cross each other. Having a long round-trip time between the two ends helps, to let the SYNs cross. To do this we'll execute one end on our host `bsdi`, and the other end on the host `vangogh.cs.berkeley.edu`. Since there is a dialup SLIP link between them, the round-trip time should be long enough (a few hundred milliseconds) to let the SYNs cross.

One end (`bsdi`) assigns itself a local port of 8888 (the `-b` command-line option) and performs an active open to port 7777 on the other host:

```
bsdi % sock -v -b8888 vangogh.cs.berkeley.edu 7777
connected on 140.252.13.35.8888 to 128.32.130.2.7777
TCP_MAXSEG = 512
hello, world                          we type this line
and hi there                          this line was typed on other end
connection closed by peer             this is output when FIN received
```

The other end is started at about the same time, assigns itself a local port of 7777, and performs an active open to port 8888:

```
vangogh % sock -v -b7777 bsdi.tuc.noao.edu 8888
connected on 128.32.130.2.7777 to 140.252.13.35.8888
TCP_MAXSEG = 512
hello, world                          this is typed on the other end
and hi there                          we type this line
^D                                    and then type our EOF character
```

We specify the `-v` flag to our `sock` program to verify the IP address and port numbers on each end of the connection. This flag also prints the MSS used by each end of the connection. We also type in one line on each end, which is sent to the other end and printed, to verify that both ends are indeed talking to each other.

Figure 18.18 shows the exchange of segments across the connection. (We have deleted some new TCP options that appear in the original SYN from `vangogh`, a 4.4BSD system. We describe these newer options in Section 18.10.) Notice the two SYNs (lines 1 and 2) followed by the two SYNs with ACKs (lines 3 and 4). These perform the simultaneous open.

Line 5 shows the input line "hello, world" going from `bsdi` to `vangogh`, with the acknowledgment in line 6. Lines 7 and 8 correspond to the line "and hi there" going in the other direction. Lines 9–12 show the normal connection termination.

> Many Berkeley-derived implementations do not support the simultaneous open correctly. On these systems, if you can get the SYNs to cross, you end up with an infinite exchange of segments, each with a SYN and an ACK, in each direction. The transition from the SYN_SENT state to the SYN_RCVD state in Figure 18.12 is not always tested in many implementations.

```
 1   0.0                  bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                     win 4096 <mss 512>
 2   0.213782 (0.2138)    vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                     win 8192 <mss 512>
 3   0.215399 (0.0016)    bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                     ack 1058199042 win 4096
                                                     <mss 512>
 4   0.340405 (0.1250)    vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                     ack 91904002 win 8192
                                                     <mss 512>
 5   5.633142 (5.2927)    bsdi.8888 > vangogh.7777: P 1:14(13) ack 1 win 4096
 6   6.100366 (0.4672)    vangogh.7777 > bsdi.8888: . ack 14 win 8192
 7   9.640214 (3.5398)    vangogh.7777 > bsdi.8888: P 1:14(13) ack 14 win 8192
 8   9.796417 (0.1562)    bsdi.8888 > vangogh.7777: . ack 14 win 4096
 9  13.060395 (3.2640)    vangogh.7777 > bsdi.8888: F 14:14(0) ack 14 win 8192
10  13.061828 (0.0014)    bsdi.8888 > vangogh.7777: . ack 15 win 4096
11  13.079769 (0.0179)    bsdi.8888 > vangogh.7777: F 14:14(0) ack 15 win 4096
12  13.299940 (0.2202)    vangogh.7777 > bsdi.8888: . ack 15 win 8192
```

Figure 18.18 Exchange of segments during simultaneous open.

18.9 Simultaneous Close

We said earlier that one side (often, but not always, the client) performs the active close, causing the first FIN to be sent. It's also possible for both sides to perform an active close, and the TCP protocol allows for this *simultaneous close*.

In terms of Figure 18.12, both ends go from ESTABLISHED to FIN_WAIT_1 when the application issues the close. This causes both FINs to be sent, and they probably pass each other somewhere in the network. When the FIN is received, each end transitions from FIN_WAIT_1 to the CLOSING state, and each state sends its final ACK. When each end receives the final ACK, the state changes to TIME_WAIT. Figure 18.19 summarizes these state changes.

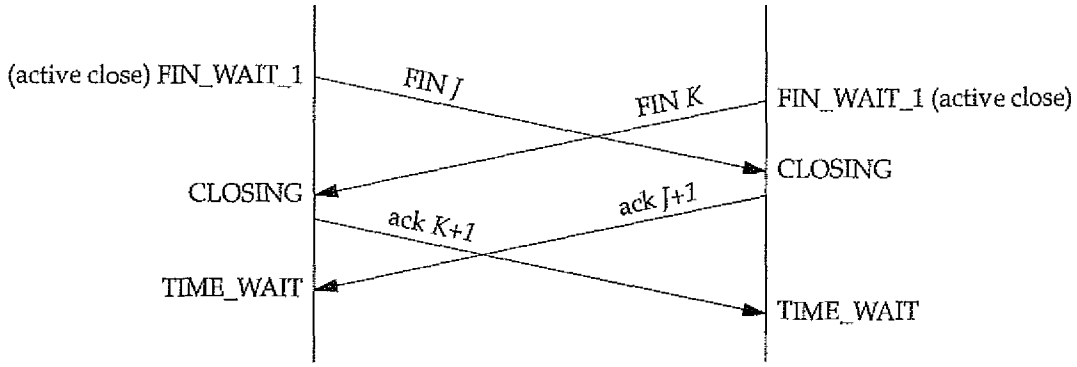


Figure 18.19 Segments exchanged during simultaneous close.

With a simultaneous close the same number of segments are exchanged as in the normal close.

18.10 TCP Options

The TCP header can contain options (Figure 17.2). The only options defined in the original TCP specification are the end of option list, no operation, and the maximum segment size option. We have seen the MSS option in almost every SYN segment in our examples.

Newer RFCs, specifically RFC 1323 [Jacobson, Braden, and Borman 1992], define additional TCP options, most of which are found only in the latest implementations. (We describe these new options in Chapter 24.) Figure 18.20 shows the format of the current TCP options—those from RFC 793 and RFC 1323.

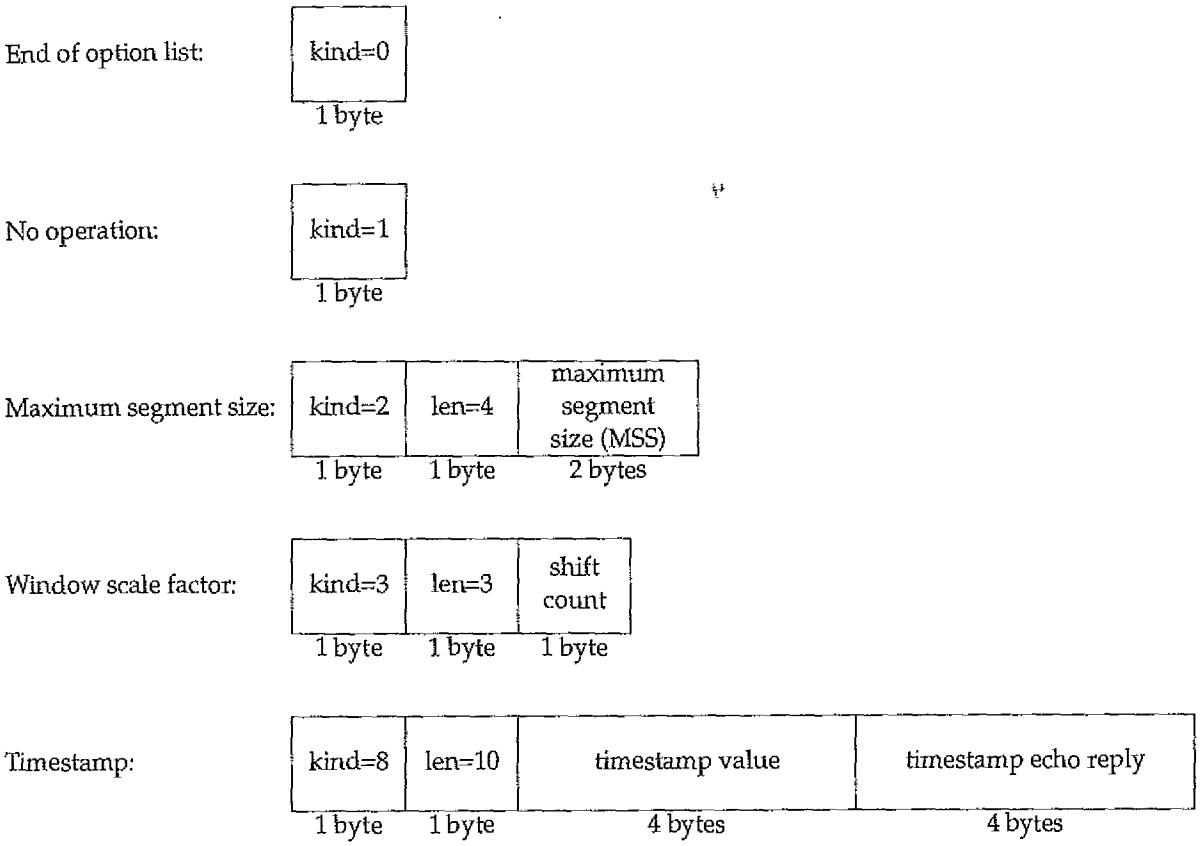


Figure 18.20 TCP options.

Every option begins with a 1-byte *kind* that specifies the type of option. The options with a *kind* of 0 and 1 occupy a single byte. The other options have a *len* byte that follows the *kind* byte. The length is the total length, including the *kind* and *len* bytes.

The reason for the no operation (NOP) option is to allow the sender to pad fields to a multiple of 4 bytes. If we initiate a TCP connection from a 4.4BSD system, the following TCP options are output by `tcpdump` on the initial SYN segment:

```
<mss 512,nop,wscale 0,nop,nop,timestamp 146647 0>
```

The MSS option is set to 512, followed by a NOP, followed by the window scale option. The reason for the first NOP is to pad the 3-byte window scale option to a 4-byte

boundary. Similarly, the 10-byte timestamp option is preceded by two NOPs, to occupy 12 bytes, placing the two 4-byte timestamps onto 4-byte boundaries.

Four other options have been proposed, with *kinds* of 4, 5, 6, and 7 called the selective-ACK and echo options. We don't show them in Figure 18.20 because the echo options have been replaced with the timestamp option, and selective ACKs, as currently defined, are still under discussion and were not included in RFC 1323. Also, the T/TCP proposal for TCP transactions (Section 24.7) specifies three options with *kinds* of 11, 12, and 13.

18.11 TCP Server Design

We said in Section 1.8 that most TCP servers are concurrent. When a new connection request arrives at a server, the server accepts the connection and invokes a new process to handle the new client. Depending on the operating system, various techniques are used to invoke the new server. Under Unix the common technique is to create a new process using the `fork` function. Lightweight processes (threads) can also be used, if supported.

What we're interested in is the interaction of TCP with concurrent servers. We need to answer the following questions: how are the port numbers handled when a server accepts a new connection request from a client, and what happens if multiple connection requests arrive at about the same time?

TCP Server Port Numbers

We can see how TCP handles the port numbers by watching any TCP server. We'll watch the Telnet server using the `netstat` command. The following output is on a system with no active Telnet connections. (We have deleted all the lines except the one showing the Telnet server.)

```
sun % netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  *.23                   *.*                    LISTEN
```

The `-a` flag reports on all network end points, not just those that are ESTABLISHED. The `-n` flag prints IP addresses as dotted-decimal numbers, instead of trying to use the DNS to convert the address to a name, and prints numeric port numbers (e.g., 23) instead of service names (e.g., Telnet). The `-f inet` option reports only TCP and UDP end points.

The local address is output as `*.23`, where the asterisk is normally called the *wildcard* character. This means that an incoming connection request (i.e., a SYN) will be accepted on any local interface. If the host were multihomed, we could specify a single IP address for the local IP address (one of the host's IP addresses), and only connections received on that interface would be accepted. (We'll see an example of this later in this section.) The local port is 23, the well-known port number for Telnet.

The foreign address is output as `*.*`, which means the foreign IP address and foreign port number are not known yet, because the end point is in the LISTEN state, waiting for a connection to arrive.

We now start a Telnet client on the host `slip` (140.252.13.65) that connects to this server. Here are the relevant lines from the `netstat` output:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23       140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

The first line for port 23 is the ESTABLISHED connection. All four elements of the local and foreign address are filled in for this connection: the local IP address and port number, and the foreign IP address and port number. The local IP address corresponds to the interface on which the connection request arrived (the Ethernet interface, 140.252.13.33).

The end point in the LISTEN state is left alone. This is the end point that the concurrent server uses to accept future connection requests. It is the TCP module in the kernel that creates the new end point in the ESTABLISHED state, when the incoming connection request arrives and is accepted. Also notice that the port number for the ESTABLISHED connection doesn't change: it's 23, the same as the LISTEN end point.

We now initiate another Telnet client from the same client (`slip`) to this server. Here is the relevant `netstat` output:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23       140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23       140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

We now have two ESTABLISHED connections from the same host to the same server. Both have a local port number of 23. This is not a problem for TCP since the foreign port numbers are different. They must be different because each of the Telnet clients uses an ephemeral port, and the definition of an ephemeral port is one that is not currently in use on that host (`slip`).

This example reiterates that TCP demultiplexes incoming segments using all four values that comprise the local and foreign addresses: destination IP address, destination port number, source IP address, and source port number. TCP cannot determine which process gets an incoming segment by looking at the destination port number only. Also, the only one of the three end points at port 23 that will receive incoming connection requests is the one in the LISTEN state. The end points in the ESTABLISHED state cannot receive SYN segments, and the end point in the LISTEN state cannot receive data segments.

Next we initiate a third Telnet client, from the host `solaris` that is across the SLIP link from sun, and not on its Ethernet.

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.23        140.252.1.32.34603     ESTABLISHED
tcp        0      0  140.252.13.33.23       140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23       140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

The local IP address of the first ESTABLISHED connection now corresponds to the interface address of SLIP link on the multihomed host sun (140.252.1.29).

Restricting Local IP Address

We can see what happens when the server does not wildcard its local IP address, setting it to one particular local interface address instead. If we specify an IP address (or hostname) to our `sock` program when we invoke it as a server, that IP address becomes the local IP address of the listening end point. For example

```
sun % sock -s 140.252.1.29 8888
```

restricts this server to connections arriving on the SLIP interface (140.252.1.29). The `netstat` output reflects this:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

If we connect to this server across the SLIP link, from the host `solaris`, it works.

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     140.252.1.32.34614     ESTABLISHED
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

But if we try to connect to this server from a host on the Ethernet (140.252.13), the connection request is not accepted by the TCP module. If we watch it with `tcpdump` the SYN is responded to with an RST, as we show in Figure 18.21.

```
1  0.0                  bsdi.1026 > sun.8888: S 3657920001:3657920001(0)
                                                win 4096 <mss 1024>
2  0.000859 (0.0009)    sun.8888 > bsdi.1026: R 0:0(0) ack 3657920002 win 0
```

Figure 18.21 Rejection of a connection request based on local IP address of server.

The server application never sees the connection request—the rejection is done by the kernel's TCP module, based on the local IP address specified by the application.

Restricting Foreign IP Address

In Section 11.12 we saw that a UDP server can normally specify the foreign IP address and foreign port, in addition to specifying the local IP address and local port. The interface functions shown in RFC 793 allow a server doing a passive open to have either a fully specified foreign socket (to wait for a particular client to issue an active open) or a unspecified foreign socket (to wait for any client).

Unfortunately, most APIs don't provide a way to do this. The server must leave the foreign socket unspecified, wait for the connection to arrive, and then examine the IP address and port number of the client.

Figure 18.22 summarizes the three types of address bindings that a TCP server can establish for itself. In all cases, *lport* is the server's well-known port and *localIP* must be the IP address of a local interface. The ordering of the three rows in the table is the order that the TCP module applies when trying to determine which local end point receives an incoming connection request. The most specific binding (the first row, if supported) is tried first, and the least specific (the last row with both IP addresses wildcarded) is tried last.

Local Address	Foreign Address	Description
localIP.lport	*foreignIP.fport*	restricted to one client (normally not supported)
localIP.lport	*.*	restricted to connections arriving on one local interface: *localIP*
*.*lport*	*.*	receives all connections sent to *lport*

Figure 18.22 Specification of local and foreign IP addresses and port number for TCP server.

Incoming Connection Request Queue

A concurrent server invokes a new process to handle each client, so the listening server should always be ready to handle the next incoming connection request. That's the underlying reason for using concurrent servers. But there is still a chance that multiple connection requests arrive while the listening server is creating a new process, or while the operating system is busy running other higher priority processes. How does TCP handle these incoming connection requests while the listening application is busy?

In Berkeley-derived implementations the following rules apply.

1. Each listening end point has a fixed length queue of connections that have been accepted by TCP (i.e., the three-way handshake is complete), but not yet accepted by the application.

   Be careful to differentiate between TCP accepting a connection and placing it on this queue, and the application taking the accepted connection off this queue.

2. The application specifies a limit to this queue, commonly called the backlog. This backlog must be between 0 and 5, inclusive. (Most applications specify the maximum value of 5.)

3. When a connection request arrives (i.e., the SYN segment), an algorithm is applied by TCP to the current number of connections already queued for this listening end point, to see whether to accept the connection or not. We would expect the backlog value specified by the application to be the maximum number of queued connections allowed for this end point, but it's not that simple. Figure 18.23 shows the relationship between the backlog value and the real maximum number of queued connections allowed by traditional Berkeley systems and Solaris 2.2.

Backlog value	Max # of queued connections	
	Traditional BSD	Solaris 2.2
0	1	0
1	2	1
2	4	2
3	5	3
4	7	4
5	8	5

Figure 18.23 Maximum number of accepted connections allowed for listening end point.

Keep in mind that this backlog value specifies only the maximum number of queued connections for one listening end point, all of which have already been accepted by TCP and are waiting to be accepted by the application. This backlog has no effect whatsoever on the maximum number of established connections allowed by the system, or on the number of clients that a concurrent server can handle concurrently.

The Solaris values in this figure are what we expect. The traditional BSD values are (for some unknown reason) the backlog value times 3, divided by 2, plus 1.

4. If there is room on this listening end point's queue for this new connection (based on Figure 18.23), the TCP module ACKs the SYN and completes the connection. The server application with the listening end point won't see this new connection until the third segment of the three-way handshake is received. Also, the client may think the server is ready to receive data when the client's active open completes successfully, before the server application has been notified of the new connection. (If this happens, the server's TCP just queues the incoming data.)
5. If there is not room on the queue for the new connection, TCP just ignores the received SYN. Nothing is sent back (i.e., no RST segment). If the listening server doesn't get around to accepting some of the already accepted connections that have filled its queue to the limit, the client's active open will eventually time out.

We can see this scenario take place with our `sock` program. We invoke it with a new option (`-O`) that tells it to pause after creating the listening end point, before accepting any connection requests. If we then invoke multiple clients during this pause period, it should cause the server's queue of accepted connections to fill, and we can see what happens with `tcpdump`.

```
bsdi % sock -s -v -q1 -O30 7777
```

The `-q1` option sets the backlog of the listening end point to 1, which for this traditional BSD system should allow two pending connection requests (Figure 18.23). The `-O30` option causes the program to sleep for 30 seconds before accepting any client connections. This gives us 30 seconds to start some clients, to fill the queue. We'll start four clients on the host `sun`.

Figure 18.24 shows the `tcpdump` output, starting with the first SYN from the first client. (We have removed the window size advertisements and MSS announcements. We have also marked the client port numbers in bold when the TCP connection is established—the three-way handshake.)

The first client's connection request from port 1090 is accepted by TCP (segments 1–3). The second client's connection request from port 1091 is also accepted by TCP (segments 4–6). The server application is still asleep, and has not accepted either connection yet. Everything has been done by the TCP module in the kernel. Also, the two clients have returned successfully from their active opens, since the three-way handshakes are complete.

```
 1   0.0                 sun.1090 > bsdi.7777: S 1617152000:1617152000(0)
 2   0.002310 ( 0.0023)  bsdi.7777 > sun.1090: S 4164096001:4164096001(0)
                                                 ack 1617152001
 3   0.003098 ( 0.0008)  sun.1090 > bsdi.7777: . ack 1

 4   4.291007 ( 4.2879)  sun.1091 > bsdi.7777: S 1617792000:1617792000(0)
 5   4.293349 ( 0.0023)  bsdi.7777 > sun.1091: S 4164672001:4164672001(0)
                                                 ack 1617792001
 6   4.294167 ( 0.0008)  sun.1091 > bsdi.7777: . ack 1

 7   7.131981 ( 2.8378)  sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
 8  10.556787 ( 3.4248)  sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
 9  12.695916 ( 2.1391)  sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
10  16.195772 ( 3.4999)  sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
11  24.695571 ( 8.4998)  sun.1092 > bsdi.7777: S 1618176000:1618176000(0)

12  28.195454 ( 3.4999)  sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
13  28.197810 ( 0.0024)  bsdi.7777 > sun.1093: S 4167808001:4167808001(0)
                                                 ack 1618688001
14  28.198639 ( 0.0008)  sun.1093 > bsdi.7777: . ack 1

15  48.694931 (20.4963)  sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
16  48.697292 ( 0.0024)  bsdi.7777 > sun.1092: S 4170496001:4170496001(0)
                                                 ack 1618176001
17  48.698145 ( 0.0009)  sun.1092 > bsdi.7777: . ack 1
```

Figure 18.24 `tcpdump` output for backlog example.

We try to start a third client in segment 7 (port 1092), and a fourth in segment 8 (port 1093). TCP ignores both SYNs since the queue for this listening end point is full. Both clients retransmit their SYNs in segments 9, 10, 11, 12, and 15. The fourth client's third retransmission is accepted (segments 12–14) because the server's 30-second pause is over, causing the server to remove the two connections that were accepted, emptying its queue. (The reason it appears this connection was accepted by the server at the time 28.19, and not at a time greater than 30, is because it took a few seconds to start the first client [segment 1, the starting time point in the output] after starting the server.) The third client's fourth retransmission is then accepted (segments 15–17). The fourth client connection (port 1093) is accepted by the server before the third client connection (port 1092) because of the timing interactions between the server's 30-second pause and the client's retransmissions.

> We would expect the queue of accepted connections to be passed to the application in FIFO (first-in, first-out) order. That is, after TCP accepts the connections on ports 1090 and 1091, we expect the application to receive the connection on port 1090 first, and then the connection on port 1091. But a bug has existed for years in many Berkeley-derived implementations causing them to be returned in a LIFO (last-in, first-out) order instead. Vendors have recently started fixing this bug, but it still exists in systems such as SunOS 4.1.3.

TCP ignores the incoming SYN when the queue is full, and doesn't respond with an RST, because this is a soft error, not a hard error. Normally the queue is full because the application or the operating system is busy, preventing the application from servicing incoming connections. This condition could change in a short while. But if the server's TCP responded with a reset, the client's active open would abort (which is what we saw

happen if the server wasn't started). By ignoring the SYN, the server forces the client TCP to retransmit the SYN later, hoping that the queue will then have room for the new connection.

A subtle point in this example, which is found in most TCP/IP implementations, is that TCP accepts an incoming connection request (i.e., a SYN) if there is room on the listener's queue, without giving the application a chance to see who it's from (the source IP address and source port number). This is not required by TCP, it's just the common implementation technique (i.e., the way the Berkeley sources have always done it). If an API such as TLI (Section 1.15) gives the application a way to learn when a connection request arrives, and then allows the application to choose whether to accept the connection or not, be aware that with TCP, when the application is supposedly told that the connection has just arrived, TCP's three-way handshake is over! Other transport layers may be implemented to provide this separation to the application between arrival and acceptance (i.e., the OSI transport layer) but not TCP.

> Solaris 2.2 provides an option that prevents TCP from accepting an incoming connection request until the application says so (`tcp_eager_listeners` in Section E.4).

This behavior also means that a TCP server has no way to cause a client's active open to fail. When a new client connection is passed to the server application, TCP's three-way handshake is over, and the client's active open has completed successfully. If the server then looks at the client's IP address and port number, and decides it doesn't want to service this client, all the server can do is either close the connection (causing a FIN to be sent) or reset the connection (causing an RST to be sent). In either case the client thought everything was OK when its active open completed, and may have already sent a request to the server.

18.12 Summary

Before two processes can exchange data using TCP, they must establish a connection between themselves. When they're done they terminate the connection. This chapter has provided a detailed look at how connections are established using a three-way handshake, and terminated using four segments.

We used `tcpdump` to show all the fields in the TCP header. We've also seen how a connection establishment can time out, how resets are sent, what happens with a half-open connection, and how TCP provides a half-close, simultaneous opens, and simultaneous closes.

Fundamental to understanding the operation of TCP is its state transition diagram. We've followed through the steps involved in connection establishment and termination, and the state transitions that take place. We also looked at the implications of TCP's connection establishment on the design of concurrent TCP servers.

A TCP connection is uniquely defined by a 4-tuple: the local IP address, local port number, foreign IP address, and foreign port number. Whenever a connection is terminated, one end must maintain knowledge of the connection, and we saw that the TIME_WAIT state handles this. The rule is that the end that does the active close enters this state for twice the implementation's MSL.

Exercises

18.1 In Section 18.2 we said that the initial sequence number (ISN) normally starts at 1 and is incremented by 64,000 every half-second and every time an active open is performed. This would imply that the low-order three digits of the ISN would always be 001. But in Figure 18.3 these low-order three digits are 521 in each direction. What's going on?

18.2 In Figure 18.15 we typed 12 characters and saw 13 bytes sent by TCP. In Figure 18.16 we typed eight characters but TCP sent 10 bytes. Why was 1 byte added in the first case, but 2 bytes in the second case?

18.3 What's the difference between a half-open connection and a half-closed connection?

18.4 If we start our `sock` program as a server, and then terminate it (without having a client connect to it), we can immediately restart the server. This implies that it doesn't go through the 2MSL wait state. Explain this in terms of the state transition diagram.

18.5 In Section 18.6 we showed that a client cannot reuse the same local port number while that port is part of a connection in the 2MSL wait. But if we run our `sock` program twice in a row as a client, connecting to the daytime server, we can reuse the same local port number. Additionally, we're able to create a new incarnation of a connection that should be in the 2MSL wait. What's going on?

```
sun % sock -v bsdi daytime
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:54:51 1993
connection closed by peer
sun % sock -v -b1163 bsdi daytime      reuse same local port number
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:55:01 1993
connection closed by peer
```

18.6 At the end of Section 18.6 when describing the FIN_WAIT_2 state, we mentioned that many implementations move a connection from this state into the CLOSED state if the application did a complete close (not a half-close) after just over 11 minutes. If the other end (in the CLOSE_WAIT state) waited 12 minutes before issuing its close (i.e., sending its FIN), what would its TCP get in response to the FIN?

18.7 Which end of a telephone conversation does the active open, and which does the passive open? Are simultaneous opens allowed? Are simultaneous closes allowed?

18.8 In Figure 18.6 we don't see an ARP request or an ARP reply. Obviously the hardware address for host `svr4` must be in the ARP cache on `bsdi`. What would change in this figure if this ARP cache entry was not present?

18.9 Explain the following `tcpdump` output. Compare it with Figure 18.13.

```
1  0:0                 solaris.32990 > bsdi.discard: S 40140288:40140288(0)
                                                     win 8760 <mss 1460>
2  0.003295 (0.0033)   bsdi.discard > solaris.32990: S 4208081409:4208081409(0)
                                                     ack 40140289 win 4096
                                                     <mss 1024>
3  0.419991 (0.4167)   solaris.32990 > bsdi.discard: P 1:257(256) ack 1 win 9216
4  0.449852 (0.0299)   solaris.32990 > bsdi.discard: F 257:257(0) ack 1 win 9216
5  0.451965 (0.0021)   bsdi.discard > solaris.32990: . ack 258 win 3840
6  0.464569 (0.0126)   bsdi.discard > solaris.32990: F 1:1(0) ack 258 win 4096
7  0.720031 (0.2555)   solaris.32990 > bsdi.discard: . ack 2 win 9216
```

18.10 Why doesn't the server in Figure 18.4 combine the ACK of the client's FIN with its own FIN, reducing the number of segments to three?

18.11 In Figure 18.16 why is the sequence number of the RST 26368002?

18.12 Does TCP's querying the link layer for the MTU violate the spirit of layering?

18.13 Assume in Figure 14.16 that each DNS query is issued using TCP instead of UDP. How many packets are exchanged?

18.14 With an MSL of 120 seconds, what is the maximum at which a system can initiate new connections and then do an active close?

18.15 Read RFC 793 to see what happens when an end point that is in the TIME_WAIT state receives a duplicate of the FIN that placed it into this state.

18.16 Read RFC 793 to see what happens when an end point that is in the TIME_WAIT state receives an RST.

18.17 Read the Host Requirements RFC to obtain the definition of a *half-duplex* TCP close.

18.18 In Figure 1.8 (p. 11) we said that incoming TCP segments are demultiplexed based on the destination TCP port number. Is that correct?