THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK

no starch press

58

SOCKETS: FUNDAMENTALS OF TCP/IP NETWORKS

This chapter provides an introduction to computer networking concepts and the TCP/IP networking protocols. An understanding of these topics is necessary to make effective use of Internet domain sockets, which are described in the next chapter.

Starting in this chapter, we begin mentioning various *Request for Comments* (RFC) documents. Each of the networking protocols discussed in this book is formally described in an RFC. We provide further information about RFCs, as well as a list of RFCs of particular relevance to the material covered in this book, in Section 58.7.

58.1 Internets

An *internetwork* or, more commonly, *internet* (with a lowercase *i*), connects different computer networks, allowing hosts on all of the networks to communicate with one another. In other words, an internet is a network of computer networks. The term *subnetwork*, or *subnet*, is used to refer to one of the networks composing an internet. An internet aims to hide the details of different physical networks in order to present a unified network architecture to all hosts on the connected networks. This means, for example, that a single address format is used to identify all hosts in the internet.

Although various internetworking protocols have been devised, TCP/IP has become the dominant protocol suite, supplanting even the proprietary networking protocols that were formerly common on local and wide area networks. The term *Internet* (with an uppercase *I*) is used to refer to the TCP/IP internet that connects millions of computers globally.

The first widespread implementation of TCP/IP appeared with 4.2BSD in 1983. Several implementations of TCP/IP are derived directly from the BSD code; other implementations, including the Linux implementation, are written from scratch, taking the operation of the BSD code as a reference standard defining the operation of TCP/IP.

> TCP/IP grew out of a project sponsored by the US Department of Defense Advanced Research Projects Agency (ARPA, later DARPA, with the *D* for Defense) to devise a computer networking architecture to be used in the ARPANET, an early wide area network. During the 1970s, a new family of protocols was designed for the ARPANET. Accurately, these protocols are known as the DARPA Internet protocol suite, but more usually they are known as the TCP/IP protocol suite, or simply TCP/IP.
>
> The web page *http://www.isoc.org/internet/history/brief.shtml* provides a brief history of the Internet and TCP/IP.

Figure 58-1 shows a simple internet. In this diagram, the machine `tekapo` is an example of a *router*, a computer whose function is to connect one subnetwork to another, transferring data between them. As well as understanding the internet protocol being used, a router must also understand the (possibly) different data-link-layer protocols used on each of the subnets that it connects.

A router has multiple network interfaces, one for each of the subnets to which it is connected. The more general term *multihomed host* is used for any host—not necessarily a router—with multiple network interfaces. (Another way of describing a router is to say that it is a multihomed host that forwards packets from one subnet to another.) A multihomed host has a different network address for each of its interfaces (i.e., a different address on each of the subnets to which it is connected).

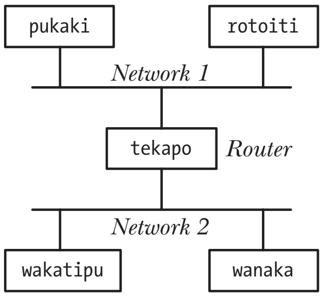


Figure 58-1: An internet using a router to connect two networks

58.2 Networking Protocols and Layers

A *networking protocol* is a set of rules defining how information is to be transmitted across a network. Networking protocols are generally organized as a series of *layers*, with each layer building on the layer below it to add features that are made available to higher layers.

The *TCP/IP protocol suite* is a layered networking protocol (Figure 58-2). It includes the *Internet Protocol* (IP) and various protocols layered above it. (The code that implements these various layers is commonly referred to as a *protocol stack*.) The name TCP/IP derives from the fact that the *Transmission Control Protocol* (TCP) is the most heavily used transport-layer protocol.

> We have omitted a range of other TCP/IP protocols from Figure 58-2 because they are not relevant to this chapter. The *Address Resolution Protocol* (ARP) is concerned with mapping Internet addresses to hardware (e.g., Ethernet) addresses. The *Internet Control Message Protocol* (ICMP) is used to convey error and control information across the network. (ICMP is used by the *ping* program, which is frequently employed to check whether a particular host is alive and visible on a TCP/IP network, and by *traceroute*, which traces the path of an IP packet through the network.) The *Internet Group Management Protocol* (IGMP) is used by hosts and routers that support multicasting of IP datagrams.

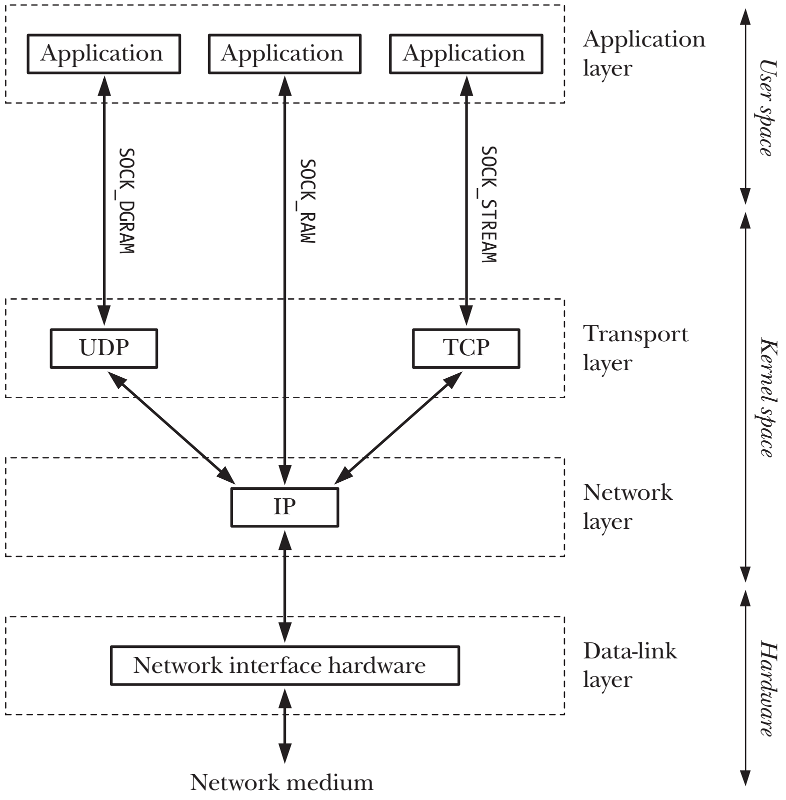


Figure 58-2: Protocols in the TCP/IP suite

One of the notions that lends great power and flexibility to protocol layering is *transparency*—each protocol layer shields higher layers from the operation and complexity of lower layers. Thus, for example, an application making use of TCP only needs to use the standard sockets API and to know that it is employing a reliable, byte-stream transport service. It doesn't need to understand the details of the operation of TCP. (When we look at socket options in Section 61.9, we'll see that this doesn't always strictly hold true; occasionally, an application does need to know some of the details of the operation of the underlying transport protocol.) Nor does the application need to know the details of the operation of IP or of the data-link layer. From the point of view of the applications, it is as though they are communicating directly with each other via the sockets API, as shown in Figure 58-3, where the dashed horizontal lines represent the virtual communication paths between corresponding application, TCP, and IP entities on the two hosts.

Encapsulation

Encapsulation is an important principle of a layered networking protocol. Figure 58-4 shows an example of encapsulation in the TCP/IP protocol layers. The key idea of encapsulation is that the information (e.g., application data, a TCP segment, or an IP datagram) passed from a higher layer to a lower layer is treated as opaque data by the lower layer. In other words, the lower layer makes no attempt to interpret information sent from the upper layer, but merely places that information inside whatever type of packet is used in the lower layer and adds its own layer-specific header before passing the packet down to the next lower layer. When data is passed up from a lower layer to a higher layer, a converse unpacking process takes place.

> We don't show it in Figure 58-4, but the concept of encapsulation also extends down into the data-link layer, where IP datagrams are encapsulated inside network frames. Encapsulation may also extend up into the application layer, where the application may perform its own packaging of data.

58.3 The Data-Link Layer

The lowest layer in Figure 58-2 is the *data-link layer*, which consists of the device driver and the hardware interface (network card) to the underlying physical medium (e.g., a telephone line, a coaxial cable, or a fiber-optic cable). The data-link layer is concerned with transferring data across a physical link in a network.

To transfer data, the data-link layer encapsulates datagrams from the network layer into units called *frames*. In addition to the data to be transmitted, each frame includes a header containing, for example, the destination address and frame size. The data-link layer transmits the frames across the physical link and handles acknowledgements from the receiver. (Not all data-link layers use acknowledgements.) This layer may perform error detection, retransmission, and flow control. Some data-link layers also split large network packets into multiple frames and reassemble them at the receiver.

From an application-programming point of view, we can generally ignore the data-link layer, since all communication details are handled in the driver and hardware.

One characteristic of the data-link layer that is important for our discussion of IP is the *maximum transmission unit* (MTU). A data-link layer's MTU is the upper limit that the layer places on the size of a frame. Different data-link layers have different MTUs.

> The command *netstat –i* displays a list of the system's network interfaces, along with their MTUs.

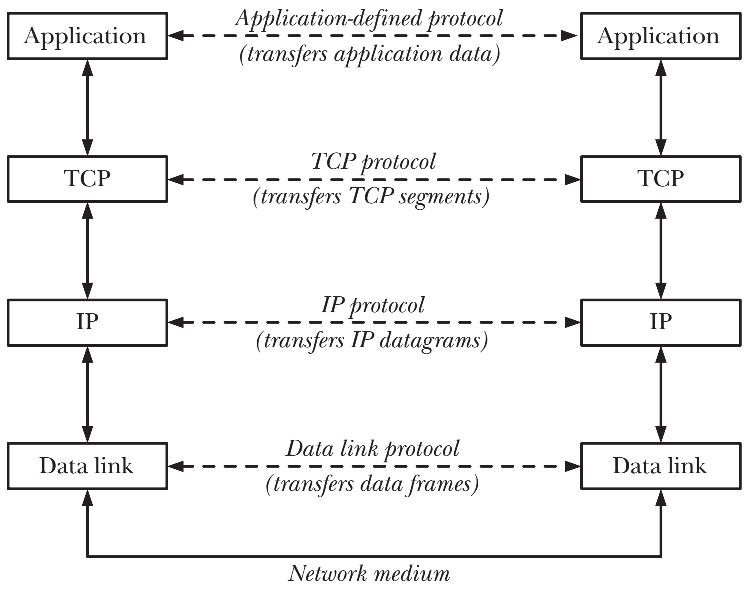
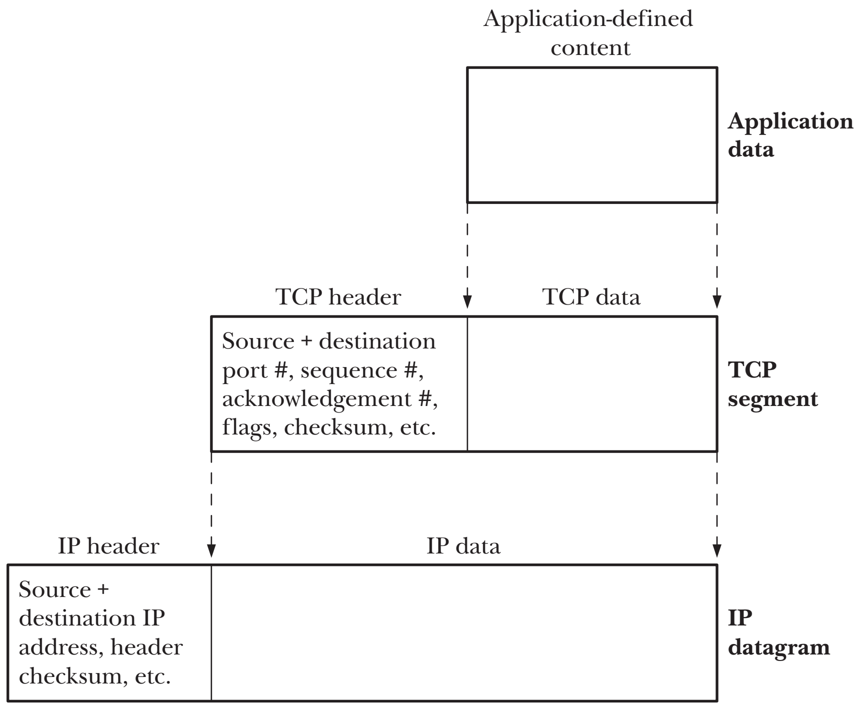


Figure 58-3: Layered communication via the TCP/IP protocols



Application-defined content
Application data
TCP header
TCP data
Source + destination port #, sequence #, acknowledgement #, flags, checksum, etc.
TCP segment
IP header
IP data
Source + destination IP address, header checksum, etc.
IP datagram

Figure 58-4: Encapsulation within the TCP/IP protocol layers

58.4 The Network Layer: IP

Above the data-link layer is the *network layer*, which is concerned with delivering packets (data) from the source host to the destination host. This layer performs a variety of tasks, including:

- breaking data into fragments small enough for transmission via the data-link layer (if necessary);
- routing data across the internet; and
- providing services to the transport layer.

In the TCP/IP protocol suite, the principal protocol in the network layer is IP. The version of IP that appeared in the 4.2BSD implementation was IP version 4 (IPv4). In the early 1990s, a revised version of IP was devised: IP version 6 (IPv6). The most notable difference between the two versions is that IPv4 identifies subnets and hosts using 32-bit addresses, while IPv6 uses 128-bit addresses, thus providing a much larger range of addresses to be assigned to hosts. Although IPv4 is still the predominant version of IP in use on the Internet, in coming years, it should be supplanted by IPv6. Both IPv4 and IPv6 support the higher UDP and TCP transport-layer protocols (as well as many other protocols).

> Although a 32-bit address space theoretically permits billions of IPv4 network addresses to be assigned, the manner in which addresses were structured and allocated meant that the practical number of available addresses was far lower. The possible exhaustion of the IPv4 address space was one of the primary motivations for the creation of IPv6.
>
> A short history of IPv6 can be found at *http://www.laynetworks.com/IPv6.htm*.
>
> The existence of IPv4 and IPv6 begs the question, "What about IPv5?" There never was an IPv5 as such. Each IP datagram header includes a 4-bit version number field (thus, IPv4 datagrams always have the number 4 in this field), and the version number 5 was assigned to an experimental protocol, *Internet Stream Protocol*. (Version 2 of this protocol, abbreviated as ST-II, is described in RFC 1819.) Initially conceived in the 1970s, this connection-oriented protocol was designed to support voice and video transmission, and distributed simulation. Since the IP datagram version number 5 was already assigned, the successor to IPv4 was assigned the version number 6.

Figure 58-2 shows a *raw* socket type (`SOCK_RAW`), which allows an application to communicate directly with the IP layer. We don't describe the use of raw sockets, since most applications employ sockets over one of the transport-layer protocols (TCP or UDP). Raw sockets are described in Chapter 28 of [Stevens et al., 2004]. One instructive example of the use of raw sockets is the *sendip* program (*http://www.earth.li/projectpurple/progs/sendip.html*), which is a command-line-driven tool that allows the construction and transmission of IP datagrams with arbitrary contents (including options to construct UDP datagrams and TCP segments).

IP transmits datagrams

IP transmits data in the form of datagrams (packets). Each datagram sent between two hosts travels independently across the network, possibly taking a different route. An IP datagram includes a header, which ranges in size from 20 to 60 bytes. The header contains the address of the target host, so that the datagram can be routed through the network to its destination, and also includes the originating address of the packet, so that the receiving host knows the origin of the datagram.

> It is possible for a sending host to spoof the originating address of a packet, and this forms the basis of a TCP denial-of-service attack known as SYN-flooding. [Lemon, 2002] describes the details of this attack and the measures used by modern TCP implementations to deal with it.

An IP implementation may place an upper limit on the size of datagrams that it supports. All IP implementations must permit datagrams at least as large as the limit specified by IP's *minimum reassembly buffer size*. In IPv4, this limit is 576 bytes; in IPv6, it is 1500 bytes.

IP is connectionless and unreliable

IP is described as a *connectionless* protocol, since it doesn't provide the notion of a virtual circuit connecting two hosts. IP is also an *unreliable* protocol: it makes a "best effort" to transmit datagrams from the sender to the receiver, but doesn't guarantee that packets will arrive in the order they were transmitted, that they won't be duplicated, or even that they will arrive at all. Nor does IP provide error recovery (packets with header errors are silently discarded). Reliability must be provided either by using a reliable transport-layer protocol (e.g., TCP) or within the application itself.

> IPv4 provides a checksum for the IP header, which allows the detection of errors in the header, but doesn't provide any error detection for the data transmitted within the packet. IPv6 doesn't provide a checksum in the IP header, relying on higher-layer protocols to provide error checking and reliability as required. (UDP checksums are optional with IPv4, but generally enabled; UDP checksums are mandatory with IPv6. TCP checksums are mandatory with both IPv4 and IPv6.)
>
> Duplication of IP datagrams may occur because of techniques employed by some data-link layers to ensure reliability or when IP datagrams are tunneled through some non-TCP/IP network that employs retransmission.

IP may fragment datagrams

IPv4 datagrams can be up to 65,535 bytes. By default, IPv6 allows datagrams of up to 65,575 bytes (40 bytes for the header, 65,535 bytes for data), and provides an option for larger datagrams (so-called *jumbograms*).

We noted earlier that most data-link layers impose an upper limit (the MTU) on the size of data frames. For example, this upper limit is 1500 bytes on the commonly used Ethernet network architecture (i.e., much smaller than the maximum size of an IP datagram). IP also defines the notion of the *path MTU*. This is the minimum MTU on all of the data-link layers traversed on the route from the source to the destination. (In practice, the Ethernet MTU is often the minimum MTU in a path.)

When an IP datagram is larger than the MTU, IP fragments (breaks up) the datagram into suitably sized units for transmission across the network. These fragments are then reassembled at the final destination to re-create the original datagram.

(Each IP fragment is itself an IP datagram that contains an offset field giving the location of that fragment within the original datagram.)

IP fragmentation occurs transparently to higher protocol layers, but nevertheless is generally considered undesirable ([Kent & Mogul, 1987]). The problem is that, because IP doesn't perform retransmission, and a datagram can be reassembled at the destination only if all fragments arrive, the entire datagram is unusable if any fragment is lost or contains transmission errors. In some cases, this can lead to significant rates of data loss (for higher protocol layers that don't perform retransmission, such as UDP) or degraded transfer rates (for higher protocol layers that do perform retransmission, such as TCP). Modern TCP implementations employ algorithms (*path MTU discovery*) to determine the MTU of a path between hosts, and accordingly break up the data they pass to IP, so that IP is not asked to transmit datagrams that exceed this size. UDP provides no such mechanism, and we consider how UDP-based applications can deal with the possibility of IP fragmentation in Section 58.6.2.

58.5 IP Addresses

An IP address consists of two parts: a network ID, which specifies the network on which a host resides, and a host ID, which identifies the host within that network.

IPv4 addresses

An IPv4 address consists of 32 bits (Figure 58-5). When expressed in human-readable form, these addresses are normally written in *dotted-decimal notation*, with the 4 bytes of the address being written as decimal numbers separated by dots, as in `204.152.189.116`.

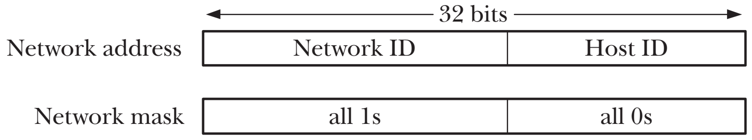


Figure 58-5: An IPv4 network address and corresponding network mask

When an organization applies for a range of IPv4 addresses for its hosts, it receives a 32-bit network address and a corresponding 32-bit *network mask*. In binary form, this mask consists of a sequence of 1s in the leftmost bits, followed by a sequence of 0s to fill out the remainder of the mask. The 1s indicate which part of the address contains the assigned network ID, while the 0s indicate which part of the address is available to the organization to assign as unique host IDs on its network. The size of the network ID part of the mask is determined when the address is assigned. Since the network ID component always occupies the leftmost part of the mask, the following notation is sufficient to specify the range of assigned addresses:

```
204.152.189.0/24
```

The `/24` indicates that the network ID part of the assigned address consists of the leftmost 24 bits, with the remaining 8 bits specifying the host ID. Alternatively, we could say that the network mask in this case is `255.255.255.0` in dotted-decimal notation.

An organization holding this address can assign 254 unique Internet addresses to its computers—`204.152.189.1` through `204.152.189.254`. Two addresses can't be assigned. One of these is the address whose host ID is all 0 bits, which is used to identify the network itself. The other is the address whose host ID is all 1 bits—`204.152.189.255` in this example—which is the *subnet broadcast address*.

Certain IPv4 addresses have special meanings. The special address `127.0.0.1` is normally defined as the *loopback address*, and is conventionally assigned the hostname `localhost`. (Any address on the network `127.0.0.0/8` can be designated as the IPv4 loopback address, but `127.0.0.1` is the usual choice.) A datagram sent to this address never actually reaches the network, but instead automatically loops back to become input to the sending host. Using this address is convenient for testing client and server programs on the same host. For use in a C program, the integer constant `INADDR_LOOPBACK` is defined for this address.

The constant `INADDR_ANY` is the so-called IPv4 *wildcard address*. The wildcard IP address is useful for applications that bind Internet domain sockets on multihomed hosts. If an application on a multihomed host binds a socket to just one of its host's IP addresses, then that socket can receive only UDP datagrams or TCP connection requests sent to that IP address. However, we normally want an application on a multihomed host to be able to receive datagrams or connection requests that specify any of the host's IP addresses, and binding the socket to the wildcard IP address makes this possible. SUSv3 doesn't specify any particular value for `INADDR_ANY`, but most implementations define it as `0.0.0.0` (all zeros).

Typically, IPv4 addresses are *subnetted*. Subnetting divides the host ID part of an IPv4 address into two parts: a subnet ID and a host ID (Figure 58-6). (The choice of how the bits of the host ID are divided is made by the local network administrator.) The rationale for subnetting is that an organization often doesn't attach all of its hosts to a single network. Instead, the organization may operate a set of subnetworks (an "internal internetwork"), with each subnetwork being identified by the combination of the network ID plus the subnet ID. This combination is usually referred to as the *extended network ID*. Within a subnet, the subnet mask serves the same role as described earlier for the network mask, and we can use a similar notation to indicate the range of addresses assigned to a particular subnet.

For example, suppose that our assigned network ID is `204.152.189.0/24`, and we choose to subnet this address range by splitting the 8 bits of the host ID into a 4-bit subnet ID and a 4-bit host ID. Under this scheme, the subnet mask would consist of 28 leading ones, followed by 4 zeros, and the subnet with the ID of 1 would be designated as `204.152.189.16/28`.

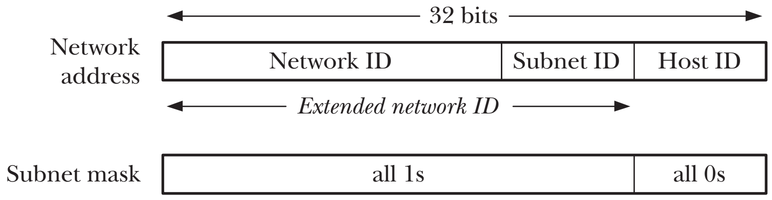


Figure 58-6: IPv4 subnetting

IPv6 addresses

The principles of IPv6 addresses are similar to IPv4 addresses. The key difference is that IPv6 addresses consist of 128 bits, and the first few bits of the address are a *format prefix*, indicating the address type. (We won't go into the details of these address types; see Appendix A of [Stevens et al., 2004] and RFC 3513 for details.)

IPv6 addresses are typically written as a series of 16-bit hexadecimal numbers separated by colons, as in the following:

```
F000:0:0:0:0:0:A:1
```

IPv6 addresses often include a sequence of zeros and, as a notational convenience, two colons (`::`) can be employed to indicate such a sequence. Thus, the above address can be rewritten as:

```
F000::A:1
```

Only one instance of the double-colon notation can appear in an IPv6 address; more than one instance would be ambiguous.

IPv6 also provides equivalents of the IPv4's loopback address (127 zeros, followed by a one, thus `::1`) and wildcard address (all zeros, written as either `0::0` or `::`).

In order to allow IPv6 applications to communicate with hosts supporting only IPv4, IPv6 provides so-called *IPv4-mapped IPv6 addresses*. The format of these addresses is shown in Figure 58-7.

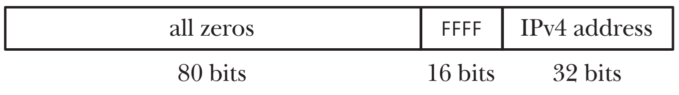


Figure 58-7: Format of an IPv4-mapped IPv6 address

When writing an IPv4-mapped IPv6 address, the IPv4 part of the address (i.e., the last 4 bytes) is written in IPv4 dotted-decimal notation. Thus, the IPv4-mapped IPv6 address equivalent to `204.152.189.116` is `::FFFF:204.152.189.116`.

58.6 The Transport Layer

There are two widely used transport-layer protocols in the TCP/IP suite:

- *User Datagram Protocol* (UDP) is the protocol used for datagram sockets.
- *Transmission Control Protocol* (TCP) is the protocol used for stream sockets.

Before considering these protocols, we first need to describe port numbers, a concept used by both protocols.

58.6.1 Port Numbers

The task of the transport protocol is to provide an end-to-end communication service to applications residing on different hosts (or sometimes on the same host). In order to do this, the transport layer requires a method of differentiating the applications on a host. In TCP and UDP, this differentiation is provided by a 16-bit *port number*.

Well-known, registered, and privileged ports

Some *well-known port numbers* are permanently assigned to specific applications (also known as *services*). For example, the *ssh* (secure shell) daemon uses the well-known port 22, and HTTP (the protocol used for communication between web servers and browsers) uses the well-known port 80. Well-known ports are assigned numbers in the range 0 to 1023 by a central authority, the Internet Assigned Numbers Authority (IANA, *http://www.iana.org/*). Assignment of a well-known port number is contingent on an approved network specification (typically in the form of an RFC).

IANA also records *registered ports*, which are allocated to application developers on a less stringent basis (which also means that an implementation doesn't need to guarantee the availability of these ports for their registered purpose). The range of IANA registered ports is 1024 to 41951. (Not all port numbers in this range are registered.)

The up-to-date list of IANA well-known and registered port assignments can be obtained online at *http://www.iana.org/assignments/port-numbers*.

In most TCP/IP implementations (including Linux), the port numbers in the range 0 to 1023 are also *privileged*, meaning that only privileged (`CAP_NET_BIND_SERVICE`) processes may bind to these ports. This prevents a normal user from implementing a malicious application that, for example, spoofs as *ssh* in order to obtain passwords. (Sometimes, privileged ports are referred to as *reserved* ports.)

Although TCP and UDP ports with the same number are distinct entities, the same well-known port number is usually assigned to a service under both TCP and UDP, even if, as is often the case, that service is available under only one of these protocols. This convention avoids confusion of port numbers across the two protocols.

Ephemeral ports

If an application doesn't select a particular port (i.e., in sockets terminology, it doesn't *bind()* its socket to a particular port), then TCP and UDP assign a unique *ephemeral port* (i.e., short-lived) number to the socket. In this case, the application—typically a client—doesn't care which port number it uses, but assigning a port is necessary so that the transport-layer protocols can identify the communication endpoints. It also has the result that the peer application at the other end of the communication channel knows how to communicate with this application. TCP and UDP also assign an ephemeral port number if we bind a socket to port 0.

IANA specifies the ports in the range 49152 to 65535 as *dynamic* or *private*, with the intention that these ports can be used by local applications and assigned as ephemeral ports. However, various implementations allocate ephemeral ports from different ranges. On Linux, the range is defined by (and can be modified via) two numbers contained in the file `/proc/sys/net/ipv4/ip_local_port_range`.

58.6.2 User Datagram Protocol (UDP)

UDP adds just two features to IP: port numbers and a data checksum to allow the detection of errors in the transmitted data.

Like IP, UDP is connectionless. Since it adds no reliability to IP, UDP is likewise unreliable. If an application layered on top of UDP requires reliability, then this must be implemented within the application. Despite this unreliability, we may sometimes prefer to use UDP instead of TCP, for the reasons detailed in Section 61.12.

The checksums used by both UDP and TCP are just 16 bits long, and are simple "add-up" checksums that can fail to detect certain classes of errors. Consequently, they do not provide extremely strong error detection. Busy Internet servers typically see an average of one undetected transmission error every few days ([Stone & Partridge, 2000]). Applications that need stronger assurances of data integrity can use the Secure Sockets Layer (SSL) protocol, which provides not only secure communication, but also much more rigorous detection of errors. Alternatively, an application could implement its own error-control scheme.

Selecting a UDP datagram size to avoid IP fragmentation

In Section 58.4, we described the IP fragmentation mechanism, and noted that it is usually best to avoid IP fragmentation. While TCP contains mechanisms for avoiding IP fragmentation, UDP does not. With UDP, we can easily cause IP fragmentation by transmitting a datagram that exceeds the MTU of the local data link.

A UDP-based application generally doesn't know the MTU of the path between the source and destination hosts. UDP-based applications that aim to avoid IP fragmentation typically adopt a conservative approach, which is to ensure that the transmitted IP datagram is less than the IPv4 minimum reassembly buffer size of 576 bytes. (This value is likely to be lower than the path MTU.) From these 576 bytes, 8 bytes are required by UDP's own header, and an additional minimum of 20 bytes are required for the IP header, leaving 548 bytes for the UDP datagram itself. In practice, many UDP-based applications opt for a still lower limit of 512 bytes for their datagrams ([Stevens, 1994]).

58.6.3 Transmission Control Protocol (TCP)

TCP provides a reliable, connection-oriented, bidirectional, byte-stream communication channel between two endpoints (i.e., applications), as shown in Figure 58-8. In order to provide these features, TCP must perform the tasks described in this section. (A detailed description of all of these features can be found in [Stevens, 1994].)

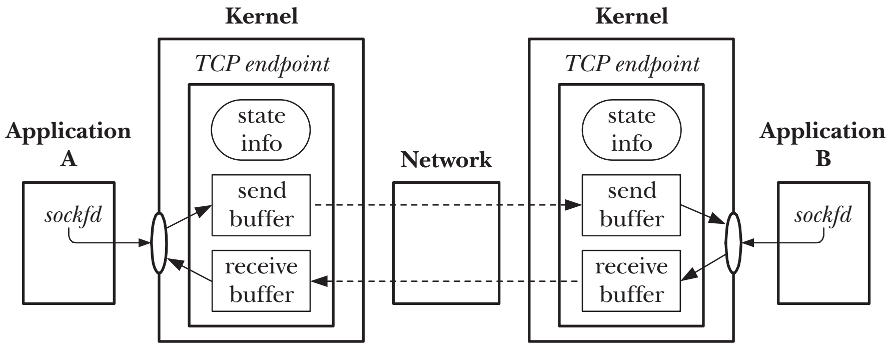


Figure 58-8: Connected TCP sockets

We use the term *TCP endpoint* to denote the information maintained by the kernel for one end of a TCP connection. (Often, we abbreviate this term further, for example, writing just "a TCP," to mean "a TCP endpoint," or "the client TCP" to mean "the TCP endpoint maintained for the client application.") This information includes the send and receive buffers for this end of the connection, as well as state information that is maintained in order to synchronize the operation of the two connected endpoints. (We describe this state information in further detail when we consider the TCP state transition diagram in Section 61.6.3.) In the remainder of this book, we use the terms *receiving TCP* and *sending TCP* to denote the TCP endpoints maintained for the receiving and sending applications on either end of a stream socket connection that is being used to transmit data in a particular direction.

Connection establishment

Before communication can commence, TCP establishes a communication channel between the two endpoints. During connection establishment, the sender and receiver can exchange options to advertise parameters for the connection.

Packaging of data in segments

Data is broken into segments, each of which contains a checksum to allow the detection of end-to-end transmission errors. Each segment is transmitted in a single IP datagram.

Acknowledgements, retransmissions, and timeouts

When a TCP segment arrives at its destination without errors, the receiving TCP sends a positive acknowledgement to the sender, informing it of the successfully delivered data. If a segment arrives with errors, then it is discarded, and no acknowledgement is sent. To handle the possibility of segments that never arrive or are discarded, the sender starts a timer when each segment is transmitted. If an acknowledgement is not received before the timer expires, the segment is retransmitted.

> Since the time taken to transmit a segment and receive its acknowledgement varies according to the range of the network and the current traffic loading, TCP employs an algorithm to dynamically adjust the size of the retransmission timeout (RTO).
>
> The receiving TCP may not send acknowledgements immediately, but instead wait for a fraction of a second to see if the acknowledgement can be piggybacked inside any response that the receiver may send straight back to the sender. (Every TCP segment includes an acknowledgement field, allowing for such piggybacking.) The aim of this technique, called *delayed ACK*, is to save sending a TCP segment, thus decreasing the number of packets in the network and decreasing the load on the sending and receiving hosts.

Sequencing

Each byte that is transmitted over a TCP connection is assigned a logical sequence number. This number indicates the position of that byte in the data stream for the connection. (Each of the two streams in the connection has its own sequence numbering.) When a TCP segment is transmitted, it includes a field containing the sequence number of the first byte in the segment.

Attaching sequence numbers to each segment serves a variety of purposes:

- The sequence number allows TCP segments to be assembled in the correct order at the destination, and then passed as a byte stream to the application layer. (At any moment, multiple TCP segments may be in transit between sender and receiver, and these segments may arrive out of order.)

- The acknowledgement message passed from the receiver back to the sender can use the sequence number to identify which TCP segment was received.
- The receiver can use the sequence number to eliminate duplicate segments. Such duplicates may occur either because of the duplication of IP datagrams or because of TCP's own retransmission algorithm, which could retransmit a successfully delivered segment if the acknowledgement for that segment was lost or was not received in a timely fashion.

The initial sequence number (ISN) for a stream doesn't start at 0. Instead, it is generated via an algorithm that increases the ISN assigned to successive TCP connections (to prevent the possibility of old segments from a previous incarnation of the connection being confused with segments for this connection). This algorithm is also designed to make guessing the ISN difficult. The sequence number is a 32-bit value that is wrapped around to 0 when the maximum value is reached.

Flow control

Flow control prevents a fast sender from overwhelming a slow receiver. To implement flow control, the receiving TCP maintains a buffer for incoming data. (Each TCP advertises the size of this buffer during connection establishment.) Data accumulates in this buffer as it is received from the sending TCP, and is removed as the application reads data. With each acknowledgement, the receiver advises the sender of how much space is available in its incoming data buffer (i.e., how many bytes the sender can transmit). The TCP flow-control algorithm employs a so-called *sliding window* algorithm, which allows unacknowledged segments containing a total of up *N* (the offered window size) bytes to be in transit between the sender and receiver. If a receiving TCP's incoming data buffer fills completely, then the window is said to be closed, and the sending TCP stops transmitting.

> The receiver can override the default size for the incoming data buffer using the `SO_RCVBUF` socket option (see the *socket(7)* manual page).

Congestion control: slow-start and congestion-avoidance algorithms

TCP's congestion-control algorithms are designed to prevent a fast sender from overwhelming a network. If a sending TCP transmits packets faster than they can be relayed by an intervening router, that router will start dropping packets. This could lead to high rates of packet loss and, consequently, serious performance degradation, if the sending TCP kept retransmitting these dropped segments at the same rate. TCP's congestion-control algorithms are important in two circumstances:

- *After connection establishment*: At this time (or when transmission resumes on a connection that has been idle for some time), the sender could start by immediately injecting as many segments into the network as would be permitted by the window size advertised by the receiver. (In fact, this is what was done in early TCP implementations.) The problem here is that if the network can't handle this flood of segments, the sender risks overwhelming the network immediately.

- *When congestion is detected*: If the sending TCP detects that congestion is occurring, then it must reduce its transmission rate. TCP detects that congestion is occurring based on the assumption that segment loss because of transmission errors is very low; thus, if a packet is lost, the cause is assumed to be congestion.

TCP's congestion-control strategy employs two algorithms in combination: slow start and congestion avoidance.

The *slow-start* algorithm causes the sending TCP to initially transmit segments at a slow rate, but allows it to exponentially increase the rate as these segments are acknowledged by the receiving TCP. Slow start attempts to prevent a fast TCP sender from overwhelming a network. However, if unrestrained, slow start's exponential increase in the transmission rate could mean that the sender would soon overwhelm the network. TCP's *congestion-avoidance* algorithm prevents this, by placing a governor on the rate increase.

With congestion avoidance, at the beginning of a connection, the sending TCP starts with a small *congestion window*, which limits the amount of unacknowledged data that it can transmit. As the sender receives acknowledgements from the peer TCP, the congestion window initially grows exponentially. However, once the congestion window reaches a certain threshold believed to be close to the transmission capacity of the network, its growth becomes linear, rather than exponential. (An estimate of the capacity of the network is derived from a calculation based on the transmission rate that was in operation when congestion was detected, or is set at a fixed value after initial establishment of the connection.) At all times, the quantity of data that the sending TCP will transmit remains additionally constrained by the receiving TCP's advertised window and the local TCP's send buffer.

In combination, the slow-start and congestion-avoidance algorithms allow the sender to rapidly raise its transmission speed up to the available capacity of the network, without overshooting that capacity. The effect of these algorithms is to allow data transmission to quickly reach a state of equilibrium, where the sender transmits packets at the same rate as it receives acknowledgements from the receiver.

58.7 Requests for Comments (RFCs)

Each of the Internet protocols that we discuss in this book is defined in an RFC document—a formal protocol specification. RFCs are published by the *RFC Editor* (*http://www.rfc-editor.org/*), which is funded by the *Internet Society* (*http://www.isoc.org/*). RFCs that describe Internet standards are developed under the auspices of the *Internet Engineering Task Force* (IETF, *http://www.ietf.org/*), a community of network designers, operators, vendors, and researchers concerned with the evolution and smooth operation of the Internet. Membership of the IETF is open to any interested individual.

The following RFCs are of particular relevance to the material covered in this book:

- RFC 791, *Internet Protocol.* J. Postel (ed.), 1981.
- RFC 950, *Internet Standard Subnetting Procedure.* J. Mogul and J. Postel, 1985.

- RFC 793, *Transmission Control Protocol.* J. Postel (ed.), 1981.
- RFC 768, *User Datagram Protocol.* J. Postel (ed.), 1980.
- RFC 1122, *Requirements for Internet Hosts–Communication Layers*. R. Braden (ed.), 1989.

  RFC 1122 extends (and corrects) various earlier RFCs describing the TCP/IP protocols. It is one of a pair of RFCs that are often simply known as the *Host Requirements RFCs*. The other member of the pair is RFC 1123, which covers application-layer protocols such as *telnet*, FTP, and SMTP.

Among the RFCs that describe IPv6 are the following:

- RFC 2460, *Internet Protocol, Version 6*. S. Deering and R. Hinden, 1998.
- RFC 4291, *IP Version 6 Addressing Architecture*. R. Hinden and S. Deering, 2006.
- RFC 3493, *Basic Socket Interface Extensions for IPv6*. R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens, 2003.
- RFC 3542, *Advanced Sockets API for IPv6*. W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei, 2003.

A number of RFCs and papers provide improvements and extensions to the original TCP specification, including the following:

- *Congestion Avoidance and Control*. V. Jacobsen, 1988. This was the initial paper describing the congestion-control and slow-start algorithms for TCP. Originally published in *Proceedings of SIGCOMM '88*, a slightly revised version is available at *ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.* This paper is largely superseded by some of the following RFCs.
- RFC 1323, *TCP Extensions for High Performance*. V. Jacobson, R. Braden, and D. Borman, 1992.
- RFC 2018, *TCP Selective Acknowledgment Options*. M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, 1996.
- RFC 2581, *TCP Congestion Control*. M. Allman, V. Paxson, and W. Stevens, 1999.
- RFC 2861, *TCP Congestion Window Validation*. M. Handley, J. Padhye, and S. Floyd, 2000.
- RFC 2883, *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, 2000.
- RFC 2988, *Computing TCP's Retransmission Timer*. V. Paxson and M. Allman, 2000.
- RFC 3168, *The Addition of Explicit Congestion Notification (ECN) to IP*. K. Ramakrishnan, S. Floyd, and D. Black, 2001.
- RFC 3390, *Increasing TCP's Initial Window*. M. Allman, S. Floyd, and C. Partridge, 2002.

58.8 Summary

TCP/IP is a layered networking protocol suite. At the bottom layer of the TCP/IP protocol stack is the IP network-layer protocol. IP transmits data in the form of datagrams. IP is connectionless, meaning that datagrams transmitted between source and destination hosts may take different routes across the network. IP is unreliable, in that it provides no guarantee that datagrams will arrive in order or unduplicated, or even arrive at all. If reliability is required, then it must be provided via the use of a reliable higher-layer protocol (e.g., TCP), or within an application.

The original version of IP is IPv4. In the early 1990s, a new version of IP, IPv6, was devised. The most notable difference between IPv4 and IPv6 is that IPv4 uses 32 bits to represent a host address, while IPv6 uses 128 bits, thus allowing for a much larger number of hosts on the world-wide Internet. Currently, IPv4 remains the most widely used version of IP, although in coming years, it is likely to be supplanted by IPv6.

Various transport-layer protocols are layered on top of IP, of which the most widely used are UDP and TCP. UDP is an unreliable datagram protocol. TCP is a reliable, connection-oriented, byte-stream protocol. TCP handles all of the details of connection establishment and termination. TCP also packages data into segments for transmission by IP, and provides sequence numbering for these segments so that they can be acknowledged and assembled in the correct order by the receiver. In addition, TCP provides flow control, to prevent a fast sender from overwhelming a slow receiver, and congestion control, to prevent a fast sender from overwhelming the network.

Further information

Refer to the sources of further information listed in Section 59.15.