CHAPTER 11

Network Programming

Network applications are everywhere. Any time you browse the Web, send an email message, or play an online game, you are using a network application. Interestingly, all network applications are based on the same basic programming model, have similar overall logical structures, and rely on the same programming interface.

Network applications rely on many of the concepts that you have already learned in our study of systems. For example, processes, signals, byte ordering, memory mapping, and dynamic storage allocation all play important roles. There are new concepts to master as well. You will need to understand the basic client-server programming model and how to write client-server programs that use the services provided by the Internet. At the end, we will tie all of these ideas together by developing a tiny but functional Web server that can serve both static and dynamic content with text and graphics to real Web browsers.

11.1 The Client-Server Programming Model

Every network application is based on the *client-server model*. With this model, an application consists of a *server* process and one or more *client* processes. A server manages some *resource*, and it provides some *service* for its clients by manipulating that resource. For example, a Web server manages a set of disk files that it retrieves and executes on behalf of clients. An FTP server manages a set of disk files that it stores and retrieves for clients. Similarly, an email server manages a spool file that it reads and updates for clients.

The fundamental operation in the client-server model is the *transaction* (Figure 11.1). A client-server transaction consists of four steps:

1. When a client needs service, it initiates a transaction by sending a *request* to the server. For example, when a Web browser needs a file, it sends a request to a Web server.
2. The server receives the request, interprets it, and manipulates its resources in the appropriate way. For example, when a Web server receives a request from a browser, it reads a disk file.
3. The server sends a *response* to the client and then waits for the next request. For example, a Web server sends the file back to a client.

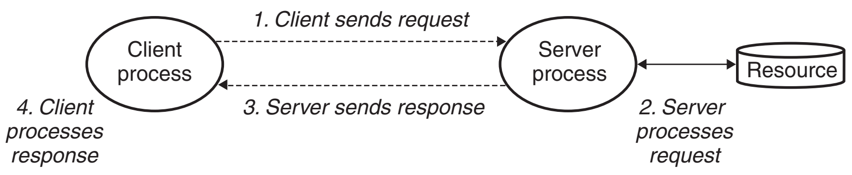


Figure 11.1 **A client-server transaction.**

Aside Client-server transactions versus database transactions

Client-server transactions are *not* database transactions and do not share any of their properties, such as atomicity. In our context, a transaction is simply a sequence of steps carried out by a client and a server.

4. The client receives the response and manipulates it. For example, after a Web browser receives a page from the server, it displays it on the screen.

It is important to realize that clients and servers are processes and not machines, or *hosts* as they are often called in this context. A single host can run many different clients and servers concurrently, and a client and server transaction can be on the same or different hosts. The client-server model is the same, regardless of the mapping of clients and servers to hosts.

11.2 Networks

Clients and servers often run on separate hosts and communicate using the hardware and software resources of a *computer network*. Networks are sophisticated systems, and we can only hope to scratch the surface here. Our aim is to give you a workable mental model from a programmer's perspective.

To a host, a network is just another I/O device that serves as a source and sink for data, as shown in Figure 11.2.

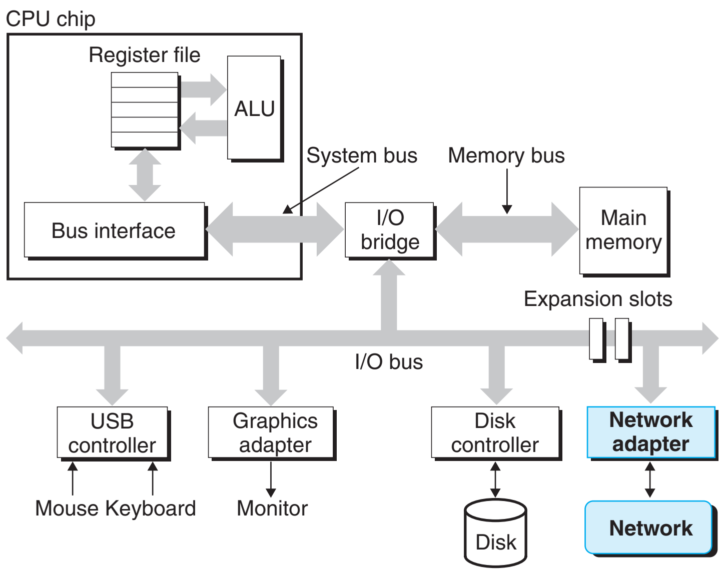


Figure 11.2
Hardware organization of a network host.

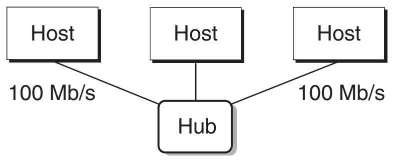


Figure 11.3
Ethernet segment.

An adapter plugged into an expansion slot on the I/O bus provides the physical interface to the network. Data received from the network are copied from the adapter across the I/O and memory buses into memory, typically by a DMA transfer. Similarly, data can also be copied from memory to the network.

Physically, a network is a hierarchical system that is organized by geographical proximity. At the lowest level is a LAN (local area network) that spans a building or a campus. The most popular LAN technology by far is *Ethernet*, which was developed in the mid-1970s at Xerox PARC. Ethernet has proven to be remarkably resilient, evolving from 3 Mb/s to 10 Gb/s.

An *Ethernet segment* consists of some wires (usually twisted pairs of wires) and a small box called a *hub*, as shown in Figure 11.3. Ethernet segments typically span small areas, such as a room or a floor in a building. Each wire has the same maximum bit bandwidth, typically 100 Mb/s or 1 Gb/s. One end is attached to an adapter on a host, and the other end is attached to a *port* on the hub. A hub slavishly copies every bit that it receives on each port to every other port. Thus, every host sees every bit.

Each Ethernet adapter has a globally unique 48-bit address that is stored in a nonvolatile memory on the adapter. A host can send a chunk of bits called a *frame* to any other host on the segment. Each frame includes some fixed number of *header* bits that identify the source and destination of the frame and the frame length, followed by a *payload* of data bits. Every host adapter sees the frame, but only the destination host actually reads it.

Multiple Ethernet segments can be connected into larger LANs, called *bridged Ethernets*, using a set of wires and small boxes called *bridges*, as shown in Figure 11.4. Bridged Ethernets can span entire buildings or campuses. In a bridged Ethernet, some wires connect bridges to bridges, and others connect bridges to hubs. The bandwidths of the wires can be different. In our example, the bridge–bridge wire has a 1 Gb/s bandwidth, while the four hub–bridge wires have bandwidths of 100 Mb/s.

Bridges make better use of the available wire bandwidth than hubs. Using a clever distributed algorithm, they automatically learn over time which hosts are reachable from which ports and then selectively copy frames from one port to another only when it is necessary. For example, if host A sends a frame to host B, which is on the segment, then bridge X will throw away the frame when it arrives at its input port, thus saving bandwidth on the other segments. However, if host A sends a frame to host C on a different segment, then bridge X will copy the frame only to the port connected to bridge Y, which will copy the frame only to the port connected to host C's segment.

Aside Internet versus internet

We will always use lowercase *internet* to denote the general concept, and uppercase *Internet* to denote a specific implementation—namely, the global IP Internet.

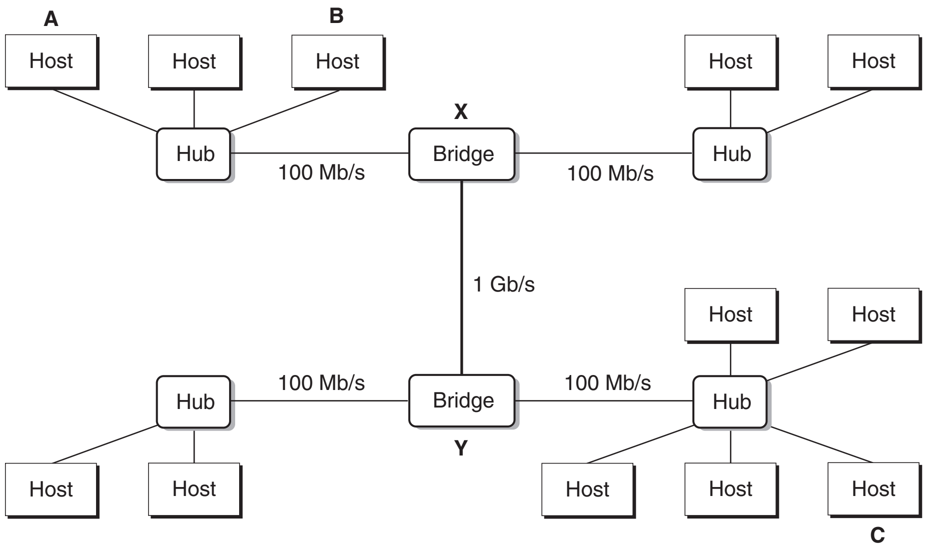
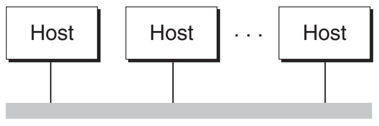


Figure 11.4 **Bridged Ethernet segments.**

Figure 11.5
Conceptual view of a LAN.



To simplify our pictures of LANs, we will draw the hubs and bridges and the wires that connect them as a single horizontal line, as shown in Figure 11.5.

At a higher level in the hierarchy, multiple incompatible LANs can be connected by specialized computers called *routers* to form an *internet* (interconnected network). Each router has an adapter (port) for each network that it is connected to. Routers can also connect high-speed point-to-point phone connections, which are examples of networks known as WANs (wide area networks), so called because they span larger geographical areas than LANs. In general, routers can be used to build internets from arbitrary collections of LANs and WANs. For example, Figure 11.6 shows an example internet with a pair of LANs and WANs connected by three routers.

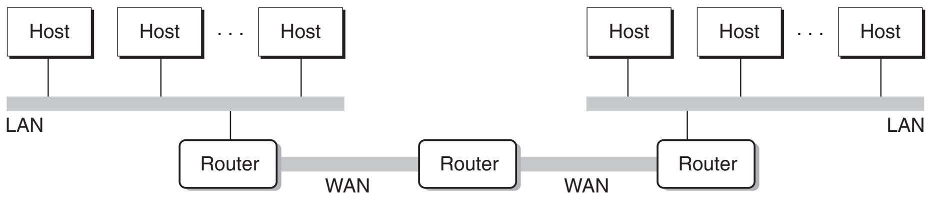


Figure 11.6 **A small internet.** Two LANs and two WANs are connected by three routers.

The crucial property of an internet is that it can consist of different LANs and WANs with radically different and incompatible technologies. Each host is physically connected to every other host, but how is it possible for some *source host* to send data bits to another *destination host* across all of these incompatible networks?

The solution is a layer of *protocol software* running on each host and router that smoothes out the differences between the different networks. This software implements a *protocol* that governs how hosts and routers cooperate in order to transfer data. The protocol must provide two basic capabilities:

Naming scheme. Different LAN technologies have different and incompatible ways of assigning addresses to hosts. The internet protocol smoothes these differences by defining a uniform format for host addresses. Each host is then assigned at least one of these *internet addresses* that uniquely identifies it.

Delivery mechanism. Different networking technologies have different and incompatible ways of encoding bits on wires and of packaging these bits into frames. The internet protocol smoothes these differences by defining a uniform way to bundle up data bits into discrete chunks called *packets*. A packet consists of a *header*, which contains the packet size and addresses of the source and destination hosts, and a *payload*, which contains data bits sent from the source host.

Figure 11.7 shows an example of how hosts and routers use the internet protocol to transfer data across incompatible LANs. The example internet consists of two LANs connected by a router. A client running on host A, which is attached to LAN1, sends a sequence of data bytes to a server running on host B, which is attached to LAN2. There are eight basic steps:

1. The client on host A invokes a system call that copies the data from the client's virtual address space into a kernel buffer.
2. The protocol software on host A creates a LAN1 frame by appending an internet header and a LAN1 frame header to the data. The internet header is addressed to internet host B. The LAN1 frame header is addressed to the router. It then passes the frame to the adapter. Notice that the payload of the LAN1 frame is an internet packet, whose payload is the actual user data. This kind of *encapsulation* is one of the fundamental insights of internetworking.

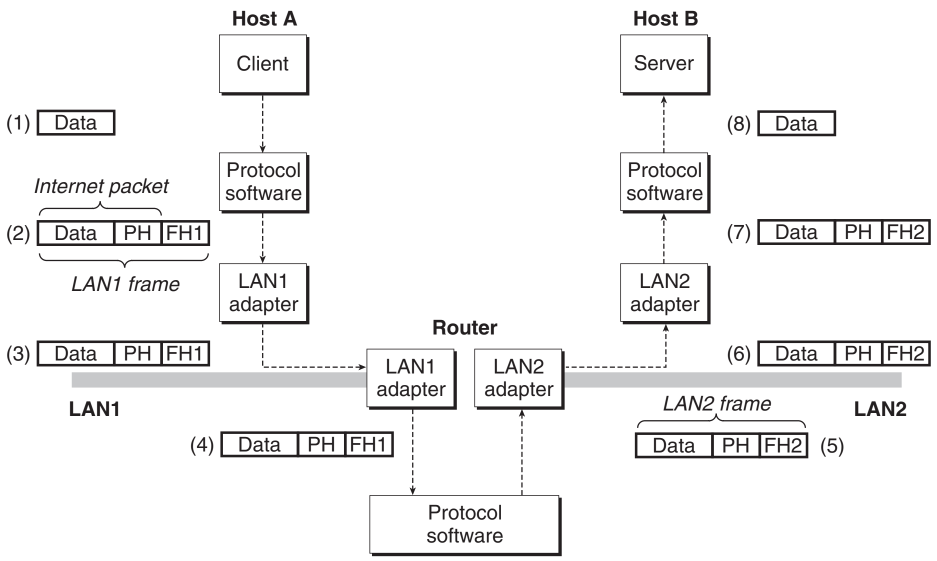


Figure 11.7 How data travel from one host to another on an internet. PH: internet packet header; FH1: frame header for LAN1; FH2: frame header for LAN2.

3. The LAN1 adapter copies the frame to the network.
4. When the frame reaches the router, the router's LAN1 adapter reads it from the wire and passes it to the protocol software.
5. The router fetches the destination internet address from the internet packet header and uses this as an index into a routing table to determine where to forward the packet, which in this case is LAN2. The router then strips off the old LAN1 frame header, prepends a new LAN2 frame header addressed to host B, and passes the resulting frame to the adapter.
6. The router's LAN2 adapter copies the frame to the network.
7. When the frame reaches host B, its adapter reads the frame from the wire and passes it to the protocol software.
8. Finally, the protocol software on host B strips off the packet header and frame header. The protocol software will eventually copy the resulting data into the server's virtual address space when the server invokes a system call that reads the data.

Of course, we are glossing over many difficult issues here. What if different networks have different maximum frame sizes? How do routers know where to forward frames? How are routers informed when the network topology changes? What if a packet gets lost? Nonetheless, our example captures the essence of the internet idea, and encapsulation is the key.

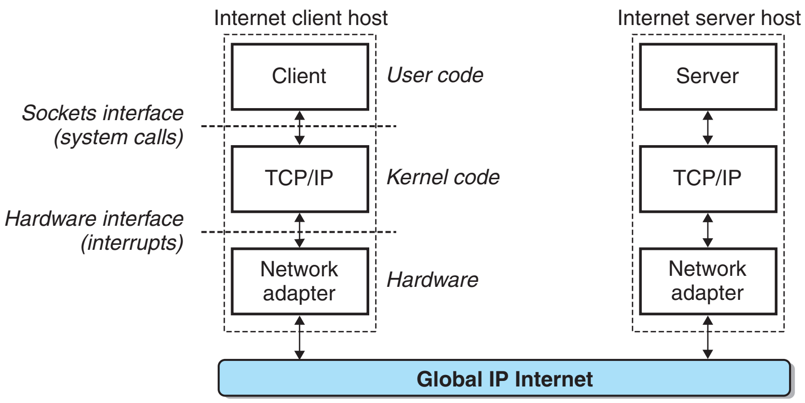


Figure 11.8
Hardware and software organization of an Internet application.

11.3 The Global IP Internet

The global IP Internet is the most famous and successful implementation of an internet. It has existed in one form or another since 1969. While the internal architecture of the Internet is complex and constantly changing, the organization of client-server applications has remained remarkably stable since the early 1980s. Figure 11.8 shows the basic hardware and software organization of an Internet client-server application.

Each Internet host runs software that implements the *TCP/IP* protocol (*Transmission Control Protocol/Internet Protocol*), which is supported by almost every modern computer system. Internet clients and servers communicate using a mix of *sockets interface* functions and Unix I/O functions. (We will describe the sockets interface in Section 11.4.) The sockets functions are typically implemented as system calls that trap into the kernel and call various kernel-mode functions in TCP/IP.

TCP/IP is actually a family of protocols, each of which contributes different capabilities. For example, IP provides the basic naming scheme and a delivery mechanism that can send packets, known as *datagrams*, from one Internet host to any other host. The IP mechanism is unreliable in the sense that it makes no effort to recover if datagrams are lost or duplicated in the network. UDP (Unreliable Datagram Protocol) extends IP slightly, so that datagrams can be transferred from process to process, rather than host to host. TCP is a complex protocol that builds on IP to provide reliable full duplex (bidirectional) *connections* between processes. To simplify our discussion, we will treat TCP/IP as a single monolithic protocol. We will not discuss its inner workings, and we will only discuss some of the basic capabilities that TCP and IP provide to application programs. We will not discuss UDP.

From a programmer's perspective, we can think of the Internet as a worldwide collection of hosts with the following properties:

- The set of hosts is mapped to a set of 32-bit *IP addresses.*

Aside IPv4 and IPv6

The original Internet protocol, with its 32-bit addresses, is known as Internet Protocol Version 4 (IPv4). In 1996, the Internet Engineering Task Force (IETF) proposed a new version of IP, called Internet Protocol Version 6 (IPv6), that uses 128-bit addresses and that was intended as the successor to IPv4. However, as of 2015, almost 20 years later, the vast majority of Internet traffic is still carried by IPv4 networks. For example, only 4 percent of users access Google services using IPv6 [42].

Because of its low adoption rate, we will not discuss IPv6 in any detail in this book and will focus exclusively on the concepts behind IPv4. When we talk about the Internet, what we mean is the Internet based on IPv4. Nonetheless, the techniques for writing clients and servers that we will teach you later in this chapter are based on modern interfaces that are independent of any particular protocol.

- The set of IP addresses is mapped to a set of identifiers called *Internet domain names*.
- A process on one Internet host can communicate with a process on any other Internet host over a *connection*.

The following sections discuss these fundamental Internet ideas in more detail.

11.3.1 IP Addresses

An IP address is an unsigned 32-bit integer. Network programs store IP addresses in the *IP address structure* shown in Figure 11.9.

Storing a scalar address in a structure is an unfortunate artifact from the early implementations of the sockets interface. It would make more sense to define a scalar type for IP addresses, but it is too late to change now because of the enormous installed base of applications.

Because Internet hosts can have different host byte orders, TCP/IP defines a uniform *network byte order* (big-endian byte order) for any integer data item, such as an IP address, that is carried across the network in a packet header. Addresses in IP address structures are always stored in (big-endian) network byte order, even if the host byte order is little-endian. Unix provides the following functions for converting between network and host byte order.

code/netp/netpfragments.c

```
/* IP address structure */
struct in_addr {
    uint32_t  s_addr; /* Address in network byte order (big-endian) */
};
```

code/netp/netpfragments.c

Figure 11.9 **IP address structure.**

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
                                        Returns: value in network byte order

uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(unit16_t netshort);
                                           Returns: value in host byte order
```

The `htonl` function converts an unsigned 32-bit integer from host byte order to network byte order. The `ntohl` function converts an unsigned 32-bit integer from network byte order to host byte order. The `htons` and `ntohs` functions perform corresponding conversions for unsigned 16-bit integers. Note that there are no equivalent functions for manipulating 64-bit values.

IP addresses are typically presented to humans in a form known as *dotted-decimal notation*, where each byte is represented by its decimal value and separated from the other bytes by a period. For example, `128.2.194.242` is the dotted-decimal representation of the address `0x8002c2f2`. On Linux systems, you can use the HOSTNAME command to determine the dotted-decimal address of your own host:

```
linux> hostname -i
128.2.210.175
```

Application programs can convert back and forth between IP addresses and dotted-decimal strings using the functions `inet_pton` and `inet_ntop`.

```
#include <arpa/inet.h>

int inet_pton(AF_INET, const char *src, void *dst);
                Returns: 1 if OK, 0 if src is invalid dotted decimal, −1 on error

const char *inet_ntop(AF_INET, const void *src, char *dst,
                      socklen_t size);
              Returns: pointer to a dotted-decimal string if OK, NULL on error
```

In these function names, the “`n`” stands for *network* and the “`p`” stands for *presentation*. They can manipulate either 32-bit IPv4 addresses (`AF_INET`), as shown here, or 128-bit IPv6 addresses (`AF_INET6`), which we do not cover.

The `inet_pton` function converts a dotted-decimal string (`src`) to a binary IP address in network byte order (`dst`). If `src` does not point to a valid dotted-decimal string, then it returns 0. Any other error returns −1 and sets `errno`. Similarly, the `inet_ntop` function converts a binary IP address in network byte order (`src`) to the corresponding dotted-decimal representation and copies at most `size` bytes of the resulting null-terminated string to `dst`.

Practice Problem 11.1 **(solution page 1002)**

Complete the following table:

Dotted-decimal address	Hex address
`107.212.122.205`	
`64.12.149.13`	
`107.212.96.29`	
	`0x00000080`
	`0xFFFFFF00`
	`0x0A010140`

Practice Problem 11.2 **(solution page 1003)**

Write a program `hex2dd.c` that converts its 16-bit hex argument to a 16-bit network byte order and prints the result. For example

```
linux> ./hex2dd 0x400
1024
```

Practice Problem 11.3 **(solution page 1003)**

Write a program `dd2hex.c` that converts its 16-bit network byte order to a 16-bit hex number and prints the result. For example,

```
linux> ./dd2hex 1024
0x400
```

11.3.2 Internet Domain Names

Internet clients and servers use IP addresses when they communicate with each other. However, large integers are difficult for people to remember, so the Internet also defines a separate set of more human-friendly *domain names*, as well as a mechanism that maps the set of domain names to the set of IP addresses. A domain name is a sequence of words (letters, numbers, and dashes) separated by periods, such as `whaleshark.ics.cs.cmu.edu`.

The set of domain names forms a hierarchy, and each domain name encodes its position in the hierarchy. An example is the easiest way to understand this. Figure 11.10 shows a portion of the domain name hierarchy.

The hierarchy is represented as a tree. The nodes of the tree represent domain names that are formed by the path back to the root. Subtrees are referred to as *subdomains*. The first level in the hierarchy is an unnamed root node. The next level is a collection of *first-level domain names* that are defined by a nonprofit organization called ICANN (Internet Corporation for Assigned Names and Numbers). Common first-level domains include `com`, `edu`, `gov`, `org`, and `net`.

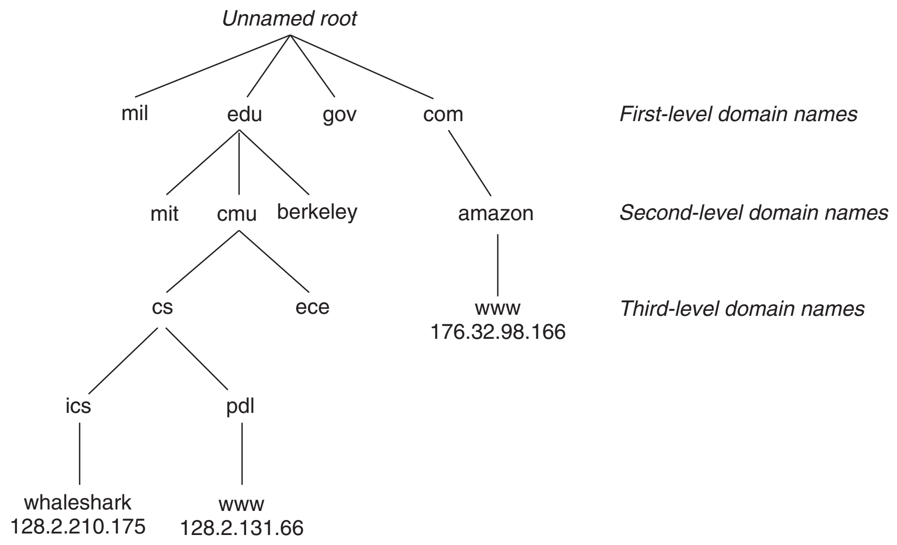


Figure 11.10 Subset of the Internet domain name hierarchy.

At the next level are *second-level* domain names such as `cmu.edu`, which are assigned on a first-come first-serve basis by various authorized agents of ICANN. Once an organization has received a second-level domain name, then it is free to create any other new domain name within its subdomain, such as `cs.cmu.edu`.

The Internet defines a mapping between the set of domain names and the set of IP addresses. Until 1988, this mapping was maintained manually in a single text file called `HOSTS.TXT`. Since then, the mapping has been maintained in a distributed worldwide database known as *DNS (Domain Name System)*. Conceptually, the DNS database consists of millions of *host entries*, each of which defines the mapping between a set of domain names and a set of IP addresses. In a mathematical sense, think of each host entry as an equivalence class of domain names and IP addresses. We can explore some of the properties of the DNS mappings with the Linux NSLOOKUP program, which displays the IP addresses associated with a domain name.[1]

Each Internet host has the locally defined domain name `localhost`, which always maps to the *loopback address* `127.0.0.1`:

```
linux> nslookup localhost
Address: 127.0.0.1
```

The `localhost` name provides a convenient and portable way to reference clients and servers that are running on the same machine, which can be especially useful

1. We've reformatted the output of NSLOOKUP to improve readability.

for debugging. We can use HOSTNAME to determine the real domain name of our local host:

```
linux> hostname
whaleshark.ics.cs.cmu.edu
```

In the simplest case, there is a one-to-one mapping between a domain name and an IP address:

```
linux> nslookup whaleshark.ics.cs.cmu.edu
Address: 128.2.210.175
```

However, in some cases, multiple domain names are mapped to the same IP address:

```
linux> nslookup cs.mit.edu
Address: 18.62.1.6

linux> nslookup eecs.mit.edu
Address: 18.62.1.6
```

In the most general case, multiple domain names are mapped to the same set of multiple IP addresses:

```
linux> nslookup www.twitter.com
Address: 199.16.156.6
Address: 199.16.156.70
Address: 199.16.156.102
Address: 199.16.156.230

linux> nslookup twitter.com
Address: 199.16.156.102
Address: 199.16.156.230
Address: 199.16.156.6
Address: 199.16.156.70
```

Finally, we notice that some valid domain names are not mapped to any IP address:

```
linux> nslookup edu
*** Can't find edu: No answer
linux> nslookup ics.cs.cmu.edu
*** Can't find ics.cs.cmu.edu: No answer
```

11.3.3 Internet Connections

Internet clients and servers communicate by sending and receiving streams of bytes over *connections*. A connection is *point-to-point* in the sense that it connects a pair of processes. It is *full duplex* in the sense that data can flow in both directions

Aside How many Internet hosts are there?

Twice a year since 1987, the Internet Systems Consortium conducts the *Internet Domain Survey*. The survey, which estimates the number of Internet hosts by counting the number of IP addresses that have been assigned a domain name, reveals an amazing trend. Since 1987, when there were about 20,000 Internet hosts, the number of hosts has been increasing exponentially. By 2015, there were over 1,000,000,000 Internet hosts!

at the same time. And it is *reliable* in the sense that—barring some catastrophic failure such as a cable cut by the proverbial careless backhoe operator—the stream of bytes sent by the source process is eventually received by the destination process in the same order it was sent.

A *socket* is an end point of a connection. Each socket has a corresponding *socket address* that consists of an Internet address and a 16-bit integer *port*[2] and is denoted by the notation `address:port`.

The port in the client's socket address is assigned automatically by the kernel when the client makes a connection request and is known as an *ephemeral port*. However, the port in the server's socket address is typically some *well-known port* that is permanently associated with the service. For example, Web servers typically use port 80, and email servers use port 25. Associated with each service with a well-known port is a corresponding *well-known service name*. For example, the well-known name for the Web service is `http`, and the well-known name for email is `smtp`. The mapping between well-known names and well-known ports is contained in a file called `/etc/services`.

A connection is uniquely identified by the socket addresses of its two end points. This pair of socket addresses is known as a *socket pair* and is denoted by the tuple

$$(cliaddr\colon cliport,\ servaddr\colon servport)$$

where *cliaddr* is the client's IP address, *cliport* is the client's port, *servaddr* is the server's IP address, and *servport* is the server's port. For example, Figure 11.11 shows a connection between a Web client and a Web server.

In this example, the Web client's socket address is

```
128.2.194.242:51213
```

where port `51213` is an ephemeral port assigned by the kernel. The Web server's socket address is

```
208.216.181.15:80
```

2. These software ports have no relation to the hardware ports in network switches and routers.

Aside Origins of the Internet

The Internet is one of the most successful examples of government, university, and industry partnership. Many factors contributed to its success, but we think two are particularly important: a sustained 30-year investment by the United States government and a commitment by passionate researchers to what Dave Clarke at MIT has dubbed "rough consensus and working code."

The seeds of the Internet were sown in 1957, when, at the height of the Cold War, the Soviet Union shocked the world by launching Sputnik, the first artificial earth satellite. In response, the United States government created the Advanced Research Projects Agency (ARPA), whose charter was to reestablish the US lead in science and technology. In 1967, Lawrence Roberts at ARPA published plans for a new network called the ARPANET. The first ARPANET nodes were up and running by 1969. By 1971, there were 13 ARPANET nodes, and email had emerged as the first important network application.

In 1972, Robert Kahn outlined the general principles of internetworking: a collection of interconnected networks, with communication between the networks handled independently on a "best-effort basis" by black boxes called "routers." In 1974, Kahn and Vinton Cerf published the first details of TCP/IP, which by 1982 had become the standard internetworking protocol for ARPANET. On January 1, 1983, every node on the ARPANET switched to TCP/IP, marking the birth of the global IP Internet.

In 1985, Paul Mockapetris invented DNS, and there were over 1,000 Internet hosts. The next year, the National Science Foundation (NSF) built the NSFNET backbone connecting 13 sites with 56 Kb/s phone lines. It was upgraded to 1.5 Mb/s T1 links in 1988 and 45 Mb/s T3 links in 1991. By 1988, there were more than 50,000 hosts. In 1989, the original ARPANET was officially retired. In 1995, when there were almost 10,000,000 Internet hosts, NSF retired NSFNET and replaced it with the modern Internet architecture based on private commercial backbones connected by public network access points.

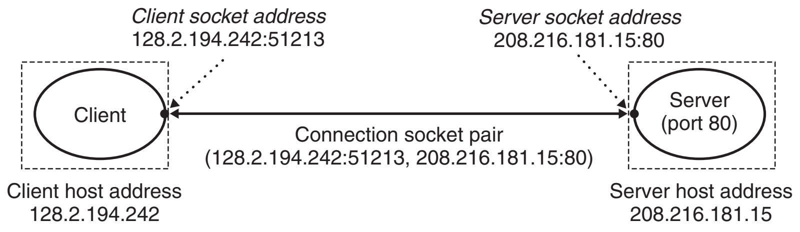


Figure 11.11
Anatomy of an Internet connection.

where port 80 is the well-known port associated with Web services. Given these client and server socket addresses, the connection between the client and server is uniquely identified by the socket pair

```
(128.2.194.242:51213, 208.216.181.15:80)
```

Aside Origins of the sockets interface

The original sockets interface was developed by researchers at University of California, Berkeley, in the early 1980s. For this reason, it is often referred to as *Berkeley sockets*. The Berkeley researchers developed the sockets interface to work with any underlying protocol. The first implementation was for TCP/IP, which they included in the Unix 4.2BSD kernel and distributed to numerous universities and labs. This was an important event in Internet history. Almost overnight, thousands of people had access to TCP/IP and its source codes. It generated tremendous excitement and sparked a flurry of new research in networking and internetworking.

11.4 The Sockets Interface

The *sockets interface* is a set of functions that are used in conjunction with the Unix I/O functions to build network applications. It has been implemented on most modern systems, including all Unix variants as well as Windows and Macintosh systems. Figure 11.12 gives an overview of the sockets interface in the context of a typical client-server transaction. You should use this picture as a road map when we discuss the individual functions.

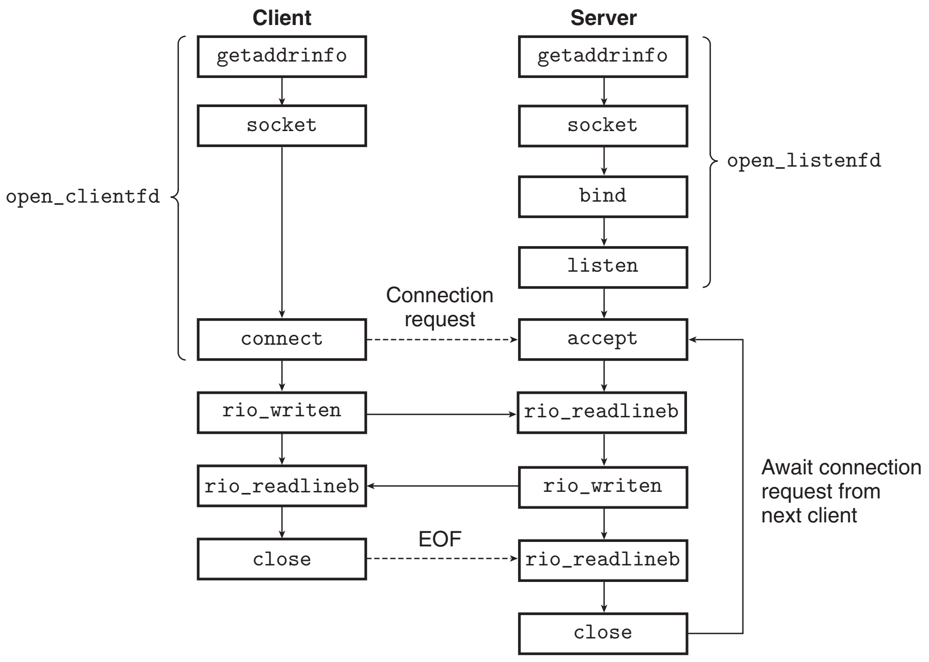


Figure 11.12 **Overview of network applications based on the sockets interface.**

Aside What does the `_in` suffix mean?

The `_in` suffix is short for *internet*, not *input*.

code/netp/netpfragments.c

```
/* IP socket address structure */
struct sockaddr_in  {
    uint16_t        sin_family;  /* Protocol family (always AF_INET) */
    uint16_t        sin_port;    /* Port number in network byte order */
    struct in_addr  sin_addr;    /* IP address in network byte order */
    unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};

/* Generic socket address structure (for connect, bind, and accept) */
struct sockaddr {
    uint16_t  sa_family;    /* Protocol family */
    char      sa_data[14];  /* Address data  */
};
```

code/netp/netpfragments.c

Figure 11.13 **Socket address structures.**

11.4.1 Socket Address Structures

From the perspective of the Linux kernel, a socket is an end point for communication. From the perspective of a Linux program, a socket is an open file with a corresponding descriptor.

Internet socket addresses are stored in 16-byte structures having the type `sockaddr_in`, shown in Figure 11.13. For Internet applications, the `sin_family` field is AF_INET, the `sin_port` field is a 16-bit port number, and the `sin_addr` field contains a 32-bit IP address. The IP address and port number are always stored in network (big-endian) byte order.

The `connect`, `bind`, and `accept` functions require a pointer to a protocol-specific socket address structure. The problem faced by the designers of the sockets interface was how to define these functions to accept any kind of socket address structure. Today, we would use the generic `void *` pointer, which did not exist in C at that time. Their solution was to define sockets functions to expect a pointer to a generic `sockaddr` structure (Figure 11.13) and then require applications to cast any pointers to protocol-specific structures to this generic structure. To simplify our code examples, we follow Stevens's lead and define the following type:

```
typedef struct sockaddr SA;
```

We then use this type whenever we need to cast a `sockaddr_in` structure to a generic `sockaddr` structure.

11.4.2 The `socket` Function

Clients and servers use the `socket` function to create a *socket descriptor*.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Returns: nonnegative descriptor if OK, −1 on error

If we wanted the socket to be the end point for a connection, then we could call `socket` with the following hardcoded arguments:

```
    clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

where AF_INET indicates that we are using 32-bit IP addresses and SOCK_STREAM indicates that the socket will be an end point for a connection. However, the best practice is to use the `getaddrinfo` function (Section 11.4.7) to generate these parameters automatically, so that the code is protocol-independent. We will show you how to use `getaddrinfo` with the `socket` function in Section 11.4.8.

The `clientfd` descriptor returned by `socket` is only partially opened and cannot yet be used for reading and writing. How we finish opening the socket depends on whether we are a client or a server. The next section describes how we finish opening the socket if we are a client.

11.4.3 The `connect` Function

A client establishes a connection with a server by calling the `connect` function.

```
#include <sys/socket.h>

int connect(int clientfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

Returns: 0 if OK, −1 on error

The `connect` function attempts to establish an Internet connection with the server at socket address `addr`, where `addrlen` is `sizeof(sockaddr_in)`. The `connect` function blocks until either the connection is successfully established or an error occurs. If successful, the `clientfd` descriptor is now ready for reading and writing, and the resulting connection is characterized by the socket pair

```
(x:y, addr.sin_addr:addr.sin_port)
```

where `x` is the client's IP address and `y` is the ephemeral port that uniquely identifies the client process on the client host. As with `socket`, the best practice is to use `getaddrinfo` to supply the arguments to `connect` (see Section 11.4.8).

11.4.4 The `bind` Function

The remaining sockets functions—`bind`, `listen`, and `accept`—are used by servers to establish connections with clients.

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

Returns: 0 if OK, −1 on error

The `bind` function asks the kernel to associate the server's socket address in `addr` with the socket descriptor `sockfd`. The `addrlen` argument is `sizeof(sockaddr_in)`. As with `socket` and `connect`, the best practice is to use `getaddrinfo` to supply the arguments to `bind` (see Section 11.4.8).

11.4.5 The `listen` Function

Clients are active entities that initiate connection requests. Servers are passive entities that wait for connection requests from clients. By default, the kernel assumes that a descriptor created by the `socket` function corresponds to an *active socket* that will live on the client end of a connection. A server calls the `listen` function to tell the kernel that the descriptor will be used by a server instead of a client.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Returns: 0 if OK, −1 on error

The `listen` function converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients. The `backlog` argument is a hint about the number of outstanding connection requests that the kernel should queue up before it starts to refuse requests. The exact meaning of the `backlog` argument requires an understanding of TCP/IP that is beyond our scope. We will typically set it to a large value, such as 1,024.

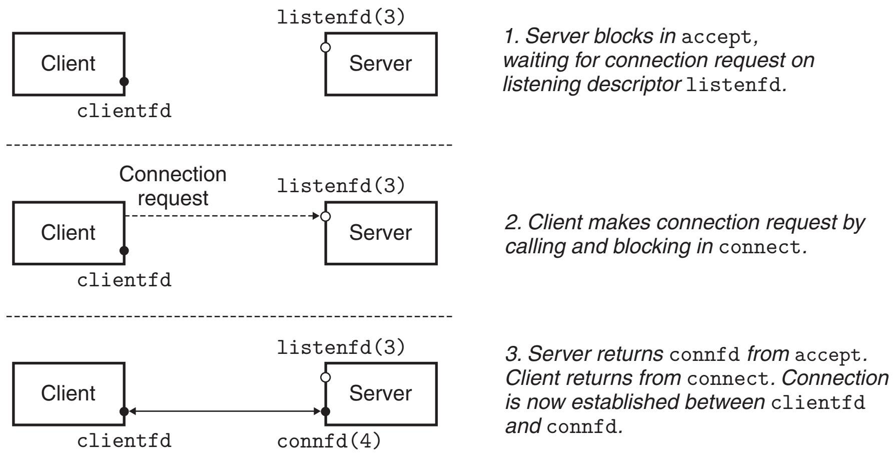

listenfd(3)
Client
Server
clientfd
1. Server blocks in accept, waiting for connection request on listening descriptor listenfd.
Connection request
listenfd(3)
Client
Server
clientfd
2. Client makes connection request by calling and blocking in connect.
listenfd(3)
Client
Server
clientfd
connfd(4)
3. Server returns connfd from accept. Client returns from connect. Connection is now established between clientfd and connfd.

Figure 11.14 **The roles of the listening and connected descriptors.**

11.4.6 The `accept` Function

Servers wait for connection requests from clients by calling the `accept` function.

```
#include <sys/socket.h>

int accept(int listenfd, struct sockaddr *addr, int *addrlen);
```
Returns: nonnegative connected descriptor if OK, −1 on error

The `accept` function waits for a connection request from a client to arrive on the listening descriptor `listenfd`, then fills in the client's socket address in `addr`, and returns a *connected descriptor* that can be used to communicate with the client using Unix I/O functions.

The distinction between a listening descriptor and a connected descriptor confuses many students. The listening descriptor serves as an end point for client connection requests. It is typically created once and exists for the lifetime of the server. The connected descriptor is the end point of the connection that is established between the client and the server. It is created each time the server accepts a connection request and exists only as long as it takes the server to service a client.

Figure 11.14 outlines the roles of the listening and connected descriptors. In step 1, the server calls `accept`, which waits for a connection request to arrive on the listening descriptor, which for concreteness we will assume is descriptor 3. Recall that descriptors 0–2 are reserved for the standard files.

In step 2, the client calls the `connect` function, which sends a connection request to `listenfd`. In step 3, the `accept` function opens a new connected descriptor `connfd` (which we will assume is descriptor 4), establishes the connection between `clientfd` and `connfd`, and then returns `connfd` to the application. The

Aside Why the distinction between listening and connected descriptors?

You might wonder why the sockets interface makes a distinction between listening and connected descriptors. At first glance, it appears to be an unnecessary complication. However, distinguishing between the two turns out to be quite useful, because it allows us to build concurrent servers that can process many client connections simultaneously. For example, each time a connection request arrives on the listening descriptor, we might fork a new process that communicates with the client over its connected descriptor. You'll learn more about concurrent servers in Chapter 12.

client also returns from the `connect`, and from this point, the client and server can pass data back and forth by reading and writing `clientfd` and `connfd`, respectively.

11.4.7 Host and Service Conversion

Linux provides some powerful functions, called `getaddrinfo` and `getnameinfo`, for converting back and forth between binary socket address structures and the string representations of hostnames, host addresses, service names, and port numbers. When used in conjunction with the sockets interface, they allow us to write network programs that are independent of any particular version of the IP protocol.

The `getaddrinfo` Function

The `getaddrinfo` function converts string representations of hostnames, host addresses, service names, and port numbers into socket address structures. It is the modern replacement for the obsolete `gethostbyname` and `getservbyname` functions. Unlike these functions, it is reentrant (see Section 12.7.2) and works with any protocol.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *host, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **result);
```

Returns: 0 if OK, nonzero error code on error

```
void freeaddrinfo(struct addrinfo *result);
```

Returns: nothing

```
const char *gai_strerror(int errcode);
```

Returns: error message

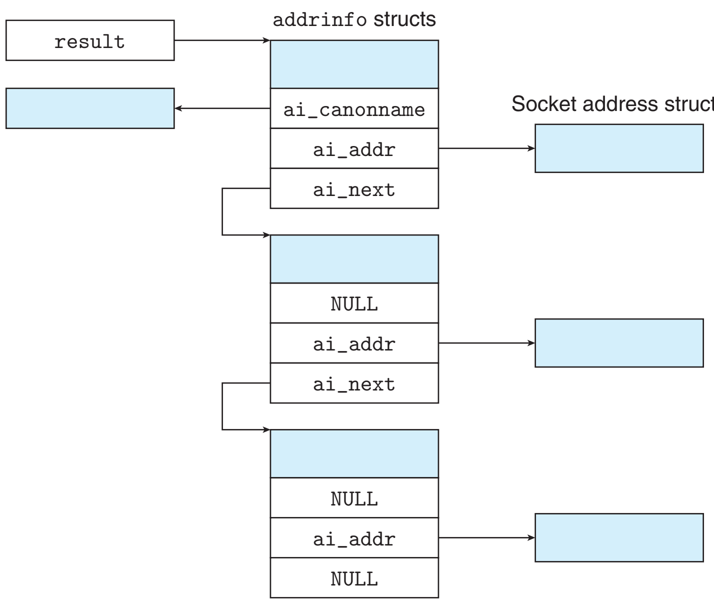


Figure 11.15
Data structure returned by `getaddrinfo`.

Given `host` and `service` (the two components of a socket address), `getaddrinfo` returns a `result` that points to a linked list of `addrinfo` structures, each of which points to a socket address structure that corresponds to `host` and `service` (Figure 11.15).

After a client calls `getaddrinfo`, it walks this list, trying each socket address in turn until the calls to `socket` and `connect` succeed and the connection is established. Similarly, a server tries each socket address on the list until the calls to `socket` and `bind` succeed and the descriptor is bound to a valid socket address. To avoid memory leaks, the application must eventually free the list by calling `freeaddrinfo`. If `getaddrinfo` returns a nonzero error code, the application can call `gai_strerror` to convert the code to a message string.

The `host` argument to `getaddrinfo` can be either a domain name or a numeric address (e.g., a dotted-decimal IP address). The `service` argument can be either a service name (e.g., `http`) or a decimal port number. If we are not interested in converting the hostname to an address, we can set `host` to NULL. The same holds for `service`. However, at least one of them must be specified.

The optional `hints` argument is an `addrinfo` structure (Figure 11.16) that provides finer control over the list of socket addresses that `getaddrinfo` returns. When passed as a `hints` argument, only the `ai_family`, `ai_socktype`, `ai_protocol`, and `ai_flags` fields can be set. The other fields must be set to zero (or NULL). In practice, we use `memset` to zero the entire structure and then set a few selected fields:

- By default, `getaddrinfo` can return both IPv4 and IPv6 socket addresses. Setting `ai_family` to AF_INET restricts the list to IPv4 addresses. Setting it to AF_INET6 restricts the list to IPv6 addresses.

code/netp/netpfragments.c

```
struct addrinfo {
    int              ai_flags;     /* Hints argument flags */
    int              ai_family;    /* First arg to socket function */
    int              ai_socktype;  /* Second arg to socket function */
    int              ai_protocol;  /* Third arg to socket function  */
    char            *ai_canonname; /* Canonical hostname */
    size_t           ai_addrlen;   /* Size of ai_addr struct */
    struct sockaddr *ai_addr;      /* Ptr to socket address structure */
    struct addrinfo *ai_next;      /* Ptr to next item in linked list */
};
```

code/netp/netpfragments.c

Figure 11.16 The `addrinfo` structure used by `getaddrinfo`.

- By default, for each unique address associated with `host`, the `getaddrinfo` function can return up to three `addrinfo` structures, each with a different `ai_socktype` field: one for connections, one for datagrams (not covered), and one for raw sockets (not covered). Setting `ai_socktype` to SOCK_STREAM restricts the list to at most one `addrinfo` structure for each unique address, one whose socket address can be used as the end point of a connection. This is the desired behavior for all of our example programs.
- The `ai_flags` field is a bit mask that further modifies the default behavior. You create it by ORing combinations of various values. Here are some that we find useful:

  AI_ADDRCONFIG. This flag is recommended if you are using connections [34]. It asks `getaddrinfo` to return IPv4 addresses only if the local host is configured for IPv4. Similarly for IPv6.

  AI_CANONNAME. By default, the `ai_canonname` field is NULL. If this flag is set, it instructs `getaddrinfo` to point the `ai_canonname` field in the first `addrinfo` structure in the list to the canonical (official) name of `host` (see Figure 11.15).

  AI_NUMERICSERV. By default, the `service` argument can be a service name or a port number. This flag forces the `service` argument to be a port number.

  AI_PASSIVE. By default, `getaddrinfo` returns socket addresses that can be used by clients as active sockets in calls to `connect`. This flag instructs it to return socket addresses that can be used by servers as listening sockets. In this case, the `host` argument should be NULL. The address field in the resulting socket address structure(s) will be the *wildcard address*, which tells the kernel that this server will accept requests to any of the IP addresses for this host. This is the desired behavior for all of our example servers.

When getaddrinfo creates an addrinfo structure in the output list, it fills in each field except for ai_flags. The ai_addr field points to a socket address structure, the ai_addrlen field gives the size of this socket address structure, and the ai_next field points to the next addrinfo structure in the list. The other fields describe various attributes of the socket address.

One of the elegant aspects of getaddrinfo is that the fields in an addrinfo structure are opaque, in the sense that they can be passed directly to the functions in the sockets interface without any further manipulation by the application code. For example, ai_family, ai_socktype, and ai_protocol can be passed directly to socket. Similarly, ai_addr and ai_addrlen can be passed directly to connect and bind. This powerful property allows us to write clients and servers that are independent of any particular version of the IP protocol.

The getnameinfo Function

The getnameinfo function is the inverse of getaddrinfo. It converts a socket address structure to the corresponding host and service name strings. It is the modern replacement for the obsolete gethostbyaddr and getservbyport functions, and unlike those functions, it is reentrant and protocol-independent.

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *service, size_t servlen, int flags);
```

Returns: 0 if OK, nonzero error code on error

The sa argument points to a socket address structure of size salen bytes, host to a buffer of size hostlen bytes, and service to a buffer of size servlen bytes. The getnameinfo function converts the socket address structure sa to the corresponding host and service name strings and copies them to the host and service buffers. If getnameinfo returns a nonzero error code, the application can convert it to a string by calling gai_strerror.

If we don't want the hostname, we can set host to NULL and hostlen to zero. The same holds for the service fields. However, one or the other must be set.

The flags argument is a bit mask that modifies the default behavior. You create it by ORing combinations of various values. Here are a couple of useful ones:

NI_NUMERICHOST. By default, getnameinfo tries to return a domain name in host. Setting this flag will cause it to return a numeric address string instead.

NI_NUMERICSERV. By default, getnameinfo will look in /etc/services and if possible, return a service name instead of a port number. Setting this flag forces it to skip the lookup and simply return the port number.

code/netp/hostinfo.c

```
1   #include "csapp.h"
2
3   int main(int argc, char **argv)
4   {
5       struct addrinfo *p, *listp, hints;
6       char buf[MAXLINE];
7       int rc, flags;
8
9       if (argc != 2) {
10          fprintf(stderr, "usage: %s <domain name>\n", argv[0]);
11          exit(0);
12      }
13
14      /* Get a list of addrinfo records */
15      memset(&hints, 0, sizeof(struct addrinfo));
16      hints.ai_family = AF_INET;       /* IPv4 only */
17      hints.ai_socktype = SOCK_STREAM; /* Connections only */
18      if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
19          fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
20          exit(1);
21      }
22
23      /* Walk the list and display each IP address */
24      flags = NI_NUMERICHOST; /* Display address string instead of domain name */
25      for (p = listp; p; p = p->ai_next) {
26          Getnameinfo(p->ai_addr, p->ai_addrlen, buf, MAXLINE, NULL, 0, flags);
27          printf("%s\n", buf);
28      }
29
30      /* Clean up */
31      Freeaddrinfo(listp);
32
33      exit(0);
34  }
```

code/netp/hostinfo.c

Figure 11.17 HOSTINFO **displays the mapping of a domain name to its associated IP addresses.**

Figure 11.17 shows a simple program, called HOSTINFO, that uses `getaddrinfo` and `getnameinfo` to display the mapping of a domain name to its associated IP addresses. It is similar to the NSLOOKUP program from Section 11.3.2.

First, we initialize the `hints` structure so that `getaddrinfo` returns the addresses we want. In this case, we are looking for 32-bit IP addresses (line 16) that can be used as end points of connections (line 17). Since we are only asking `getaddrinfo` to convert domain names, we call it with a NULL `service` argument.

After the call to `getaddrinfo`, we walk the list of `addrinfo` structures, using `getnameinfo` to convert each socket address to a dotted-decimal address string. After walking the list, we are careful to free it by calling `freeaddrinfo` (although for this simple program it is not strictly necessary).

When we run HOSTINFO, we see that `twitter.com` maps to four IP addresses, which is what we saw using NSLOOKUP in Section 11.3.2.

```
linux> ./hostinfo twitter.com
199.16.156.102
199.16.156.230
199.16.156.6
199.16.156.70
```

Practice Problem 11.4 (solution page 1004)

The `getaddrinfo` and `getnameinfo` functions subsume the functionality of `inet_pton` and `inet_ntop`, respectively, and they provide a higher-level of abstraction that is independent of any particular address format. To convince yourself how handy this is, write a version of HOSTINFO (Figure 11.17) that uses `inet_ntop` instead of `getnameinfo` to convert each socket address to a dotted-decimal address string.

11.4.8 Helper Functions for the Sockets Interface

The `getaddrinfo` function and the sockets interface can seem somewhat daunting when you first learn about them. We find it convenient to wrap them with higher-level helper functions, called `open_clientfd` and `open_listenfd`, that clients and servers can use when they want to communicate with each other.

The `open_clientfd` Function

A client establishes a connection with a server by calling `open_clientfd`.

```
#include "csapp.h"

int open_clientfd(char *hostname, char *port);
```

Returns: descriptor if OK, −1 on error

The `open_clientfd` function establishes a connection with a server running on host `hostname` and listening for connection requests on port number `port`. It returns an open socket descriptor that is ready for input and output using the Unix I/O functions. Figure 11.18 shows the code for `open_clientfd`.

We call `getaddrinfo`, which returns a list of `addrinfo` structures, each of which points to a socket address structure that is suitable for establishing a con-

code/src/csapp.c

```
int open_clientfd(char *hostname, char *port) {
    int clientfd;
    struct addrinfo hints, *listp, *p;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;  /* Open a connection */
    hints.ai_flags = AI_NUMERICSERV;  /* ... using a numeric port arg. */
    hints.ai_flags |= AI_ADDRCONFIG;  /* Recommended for connections */
    Getaddrinfo(hostname, port, &hints, &listp);

    /* Walk the list for one that we can successfully connect to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((clientfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
            continue; /* Socket failed, try the next */

        /* Connect to the server */
        if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
            break; /* Success */
        Close(clientfd); /* Connect failed, try another */
    }

    /* Clean up */
    Freeaddrinfo(listp);
    if (!p) /* All connects failed */
        return -1;
    else    /* The last connect succeeded */
        return clientfd;
}
```

code/src/csapp.c

Figure 11.18 `open_clientfd`: **Helper function that establishes a connection with a server.** It is reentrant and protocol-independent.

nection with a server running on `hostname` and listening on `port`. We then walk the list, trying each list entry in turn, until the calls to `socket` and `connect` succeed. If the `connect` fails, we are careful to close the socket descriptor before trying the next entry. If the `connect` succeeds, we free the list memory and return the socket descriptor to the client, which can immediately begin using Unix I/O to communicate with the server.

Notice how there is no dependence on any particular version of IP anywhere in the code. The arguments to `socket` and `connect` are generated for us automatically by `getaddrinfo`, which allows our code to be clean and portable.

The `open_listenfd` Function

A server creates a listening descriptor that is ready to receive connection requests by calling the `open_listenfd` function.

```
#include "csapp.h"

int open_listenfd(char *port);
```

Returns: descriptor if OK, −1 on error

The `open_listenfd` function returns a listening descriptor that is ready to receive connection requests on port `port`. Figure 11.19 shows the code for `open_listenfd`.

The style is similar to `open_clientfd`. We call `getaddrinfo` and then walk the resulting list until the calls to `socket` and `bind` succeed. Note that in line 20 we use the `setsockopt` function (not described here) to configure the server so that it can be terminated, be restarted, and begin accepting connection requests immediately. By default, a restarted server will deny connection requests from clients for approximately 30 seconds, which seriously hinders debugging.

Since we have called `getaddrinfo` with the AI_PASSIVE flag and a NULL `host` argument, the address field in each socket address structure is set to the wildcard address, which tells the kernel that this server will accept requests to any of the IP addresses for this host.

Finally, we call the `listen` function to convert `listenfd` to a listening descriptor and return it to the caller. If the `listen` fails, we are careful to avoid a memory leak by closing the descriptor before returning.

11.4.9 Example Echo Client and Server

The best way to learn the sockets interface is to study example code. Figure 11.20 shows the code for an echo client. After establishing a connection with the server, the client enters a loop that repeatedly reads a text line from standard input, sends the text line to the server, reads the echo line from the server, and prints the result to standard output. The loop terminates when `fgets` encounters EOF on standard input, either because the user typed Ctrl+D at the keyboard or because it has exhausted the text lines in a redirected input file.

After the loop terminates, the client closes the descriptor. This results in an EOF notification being sent to the server, which it detects when it receives a return code of zero from its `rio_readlineb` function. After closing its descriptor, the client terminates. Since the client's kernel automatically closes all open descriptors when a process terminates, the `close` in line 24 is not necessary. However, it is good programming practice to explicitly close any descriptors that you have opened.

Figure 11.21 shows the main routine for the echo server. After opening the listening descriptor, it enters an infinite loop. Each iteration waits for a connection request from a client, prints the domain name and port of the connected client, and then calls the `echo` function that services the client. After the echo routine returns, *code/src/csapp.c*

```c
int open_listenfd(char *port)
{
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;             /* Accept connections */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG; /* ... on any IP address */
    hints.ai_flags |= AI_NUMERICSERV;            /* ... using port number */
    Getaddrinfo(NULL, port, &hints, &listp);

    /* Walk the list for one that we can bind to */
    for (p = listp; p; p = p->ai_next) {
        /* Create a socket descriptor */
        if ((listenfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
            continue;  /* Socket failed, try the next */

        /* Eliminates "Address already in use" error from bind */
        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int));

        /* Bind the descriptor to the address */
        if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
            break; /* Success */
        Close(listenfd); /* Bind failed, try the next */
    }

    /* Clean up */
    Freeaddrinfo(listp);
    if (!p) /* No address worked */
        return -1;

    /* Make it a listening socket ready to accept connection requests */
    if (listen(listenfd, LISTENQ) < 0) {
        Close(listenfd);
        return -1;
    }
    return listenfd;
}
```

code/src/csapp.c

Figure 11.19 `open_listenfd`: **Helper function that opens and returns a listening descriptor.** It is reentrant and protocol-independent.

code/netp/echoclient.c

```c
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;

    if (argc != 3) {
        fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
        exit(0);
    }
    host = argv[1];
    port = argv[2];

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

code/netp/echoclient.c

Figure 11.20 Echo client main routine.

the main routine closes the connected descriptor. Once the client and server have closed their respective descriptors, the connection is terminated.

The `clientaddr` variable in line 9 is a socket address structure that is passed to `accept`. Before `accept` returns, it fills in `clientaddr` with the socket address of the client on the other end of the connection. Notice how we declare `clientaddr` as type `struct sockaddr_storage` rather than `struct sockaddr_in`. By definition, the `sockaddr_storage` structure is large enough to hold any type of socket address, which keeps the code protocol-independent.

Notice that our simple echo server can only handle one client at a time. A server of this type that iterates through clients, one at a time, is called an *iterative server*. In Chapter 12, we will learn how to build more sophisticated *concurrent servers* that can handle multiple clients simultaneously.

Finally, Figure 11.22 shows the code for the `echo` routine, which repeatedly reads and writes lines of text until the `rio_readlineb` function encounters EOF in line 10.

code/netp/echoserveri.c

```
#include "csapp.h"

void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;  /* Enough space for any address */
    char client_hostname[MAXLINE], client_port[MAXLINE];

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen, client_hostname, MAXLINE,
                    client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

code/netp/echoserveri.c

Figure 11.21 **Iterative echo server main routine.**

code/netp/echo.c

```
#include "csapp.h"

void echo(int connfd)
{
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        Rio_writen(connfd, buf, n);
    }
}
```

code/netp/echo.c

Figure 11.22 `echo` **function that reads and echoes text lines.**

Aside What does EOF on a connection mean?

The idea of EOF is often confusing to students, especially in the context of Internet connections. First, we need to understand that there is no such thing as an EOF character. Rather, EOF is a condition that is detected by the kernel. An application finds out about the EOF condition when it receives a zero return code from the `read` function. For disk files, EOF occurs when the current file position exceeds the file length. For Internet connections, EOF occurs when a process closes its end of the connection. The process at the other end of the connection detects the EOF when it attempts to read past the last byte in the stream.

11.5 Web Servers

So far we have discussed network programming in the context of a simple echo server. In this section, we will show you how to use the basic ideas of network programming to build your own small, but quite functional, Web server.

11.5.1 Web Basics

Web clients and servers interact using a text-based application-level protocol known as *HTTP (hypertext transfer protocol)*. HTTP is a simple protocol. A Web client (known as a *browser*) opens an Internet connection to a server and requests some *content*. The server responds with the requested content and then closes the connection. The browser reads the content and displays it on the screen.

What distinguishes Web services from conventional file retrieval services such as FTP? The main difference is that Web content can be written in a language known as *HTML (hypertext markup language)*. An HTML program (page) contains instructions (tags) that tell the browser how to display various text and graphical objects in the page. For example, the code

```
<b> Make me bold! </b>
```

tells the browser to print the text between the `<b>` and `</b>` tags in boldface type. However, the real power of HTML is that a page can contain pointers (hyperlinks) to content stored on any Internet host. For example, an HTML line of the form

```
<a href="http://www.cmu.edu/index.html">Carnegie Mellon</a>
```

tells the browser to highlight the text object `Carnegie Mellon` and to create a hyperlink to an HTML file called `index.html` that is stored on the CMU Web server. If the user clicks on the highlighted text object, the browser requests the corresponding HTML file from the CMU server and displays it.

Aside Origins of the World Wide Web

The World Wide Web was invented by Tim Berners-Lee, a software engineer working at CERN, a Swiss physics lab. In 1989, Berners-Lee wrote an internal memo proposing a distributed hypertext system that would connect a "web of notes with links." The intent of the proposed system was to help CERN scientists share and manage information. Over the next two years, after Berners-Lee implemented the first Web server and Web browser, the Web developed a small following within CERN and a few other sites. A pivotal event occurred in 1993, when Marc Andreesen (who later founded Netscape and Andreessen Horowitz) and his colleagues at NCSA released a graphical browser called MOSAIC for all three major platforms: Linux, Windows, and Macintosh. After the release of MOSAIC, interest in the Web exploded, with the number of Web sites increasing at an exponential rate. By 2015, there were over 975,000,000 sites worldwide. (Source: Netcraft Web Survey)

MIME type	Description
`text/html`	HTML page
`text/plain`	Unformatted text
`application/postscript`	Postscript document
`image/gif`	Binary image encoded in GIF format
`image/png`	Binary image encoded in PNG format
`image/jpeg`	Binary image encoded in JPEG format

Figure 11.23 **Example MIME types.**

11.5.2 Web Content

To Web clients and servers, *content* is a sequence of bytes with an associated *MIME (multipurpose internet mail extensions)* type. Figure 11.23 shows some common MIME types.

Web servers provide content to clients in two different ways:

- Fetch a disk file and return its contents to the client. The disk file is known as *static content* and the process of returning the file to the client is known as *serving static content*.
- Run an executable file and return its output to the client. The output produced by the executable at run time is known as *dynamic content*, and the process of running the program and returning its output to the client is known as *serving dynamic content*.

Every piece of content returned by a Web server is associated with some file that it manages. Each of these files has a unique name known as a *URL (universal resource locator)*. For example, the URL

```
http://www.google.com:80/index.html
```

identifies an HTML file called `/index.html` on Internet host `www.google.com` that is managed by a Web server listening on port 80. The port number is optional and defaults to the well-known HTTP port 80. URLs for executable files can include program arguments after the filename. A '`?`' character separates the filename from the arguments, and each argument is separated by an '`&`' character. For example, the URL

```
http://bluefish.ics.cs.cmu.edu:8000/cgi-bin/adder?15000&213
```

identifies an executable called `/cgi-bin/adder` that will be called with two argument strings: `15000` and `213`. Clients and servers use different parts of the URL during a transaction. For instance, a client uses the prefix

```
http://www.google.com:80
```

to determine what kind of server to contact, where the server is, and what port it is listening on. The server uses the suffix

```
/index.html
```

to find the file on its filesystem and to determine whether the request is for static or dynamic content.

There are several points to understand about how servers interpret the suffix of a URL:

- There are no standard rules for determining whether a URL refers to static or dynamic content. Each server has its own rules for the files it manages. A classic (old-fashioned) approach is to identify a set of directories, such as `cgi-bin`, where all executables must reside.
- The initial '`/`' in the suffix does *not* denote the Linux root directory. Rather, it denotes the home directory for whatever kind of content is being requested. For example, a server might be configured so that all static content is stored in directory `/usr/httpd/html` and all dynamic content is stored in directory `/usr/httpd/cgi-bin`.
- The minimal URL suffix is the '`/`' character, which all servers expand to some default home page such as `/index.html`. This explains why it is possible to fetch the home page of a site by simply typing a domain name to the browser. The browser appends the missing '`/`' to the URL and passes it to the server, which expands the '`/`' to some default filename.

11.5.3 HTTP Transactions

Since HTTP is based on text lines transmitted over Internet connections, we can use the Linux TELNET program to conduct transactions with any Web server on the Internet. The TELNET program has been largely supplanted by SSH as a remote login tool, but it is very handy for debugging servers that talk to clients with text lines over connections. For example, Figure 11.24 uses TELNET to request the home page from the AOL Web server.

```
1  linux> telnet www.aol.com 80          Client: open connection to server
2  Trying 205.188.146.23...              Telnet prints 3 lines to the terminal
3  Connected to aol.com.
4  Escape character is '^]'.
5  GET / HTTP/1.1                        Client: request line
6  Host: www.aol.com                     Client: required HTTP/1.1 header
7                                        Client: empty line terminates headers
8  HTTP/1.0 200 OK                       Server: response line
9  MIME-Version: 1.0                     Server: followed by five response headers
10 Date: Mon, 8 Jan 2010 4:59:42 GMT
11 Server:  Apache-Coyote/1.1
12 Content-Type: text/html               Server: expect HTML in the response body
13 Content-Length: 42092                 Server: expect 42,092 bytes in the response body
14                                       Server: empty line terminates response headers
15 <html>                                Server: first HTML line in response body
16 ...                                   Server: 766 lines of HTML not shown
17 </html>                               Server: last HTML line in response body
18 Connection closed by foreign host.    Server: closes connection
19 linux>                                Client: closes connection and terminates
```

Figure 11.24 Example of an HTTP transaction that serves static content.

In line 1, we run TELNET from a Linux shell and ask it to open a connection to the AOL Web server. TELNET prints three lines of output to the terminal, opens the connection, and then waits for us to enter text (line 5). Each time we enter a text line and hit the `enter` key, TELNET reads the line, appends carriage return and line feed characters (‘`\r\n`’ in C notation), and sends the line to the server. This is consistent with the HTTP standard, which requires every text line to be terminated by a carriage return and line feed pair. To initiate the transaction, we enter an HTTP request (lines 5–7). The server replies with an HTTP response (lines 8–17) and then closes the connection (line 18).

HTTP Requests

An *HTTP request* consists of a *request line* (line 5), followed by zero or more *request headers* (line 6), followed by an empty text line that terminates the list of headers (line 7). A request line has the form

method URI version

HTTP supports a number of different *methods*, including GET, POST, OPTIONS, HEAD, PUT, DELETE, and TRACE. We will only discuss the workhorse GET method, which accounts for a majority of HTTP requests. The GET method instructs the server to generate and return the content identified by the *URI* *(uniform resource identifier)*. The URI is the suffix of the corresponding URL that includes the filename and optional arguments.[3]

The *version* field in the request line indicates the HTTP version to which the request conforms. The most recent HTTP version is HTTP/1.1 [37]. HTTP/1.0 is an earlier, much simpler version from 1996 [6]. HTTP/1.1 defines additional headers that provide support for advanced features such as caching and security, as well as a mechanism that allows a client and server to perform multiple transactions over the same *persistent connection*. In practice, the two versions are compatible because HTTP/1.0 clients and servers simply ignore unknown HTTP/1.1 headers.

To summarize, the request line in line 5 asks the server to fetch and return the HTML file `/index.html`. It also informs the server that the remainder of the request will be in HTTP/1.1 format.

Request headers provide additional information to the server, such as the brand name of the browser or the MIME types that the browser understands. Request headers have the form

header-name: *header-data*

For our purposes, the only header to be concerned with is the `Host` header (line 6), which is required in HTTP/1.1 requests, but not in HTTP/1.0 requests. The `Host` header is used by *proxy caches*, which sometimes serve as intermediaries between a browser and the *origin server* that manages the requested file. Multiple proxies can exist between a client and an origin server in a so-called proxy chain. The data in the `Host` header, which identifies the domain name of the origin server, allow a proxy in the middle of a proxy chain to determine if it might have a locally cached copy of the requested content.

Continuing with our example in Figure 11.24, the empty text line in line 7 (generated by hitting `enter` on our keyboard) terminates the headers and instructs the server to send the requested HTML file.

HTTP Responses

HTTP responses are similar to HTTP requests. An *HTTP response* consists of a *response line* (line 8), followed by zero or more *response headers* (lines 9–13), followed by an empty line that terminates the headers (line 14), followed by the *response body* (lines 15–17). A response line has the form

version status-code status-message

The *version* field describes the HTTP version that the response conforms to. The *status-code* is a three-digit positive integer that indicates the disposition of the request. The *status-message* gives the English equivalent of the error code. Figure 11.25 lists some common status codes and their corresponding messages.

3. Actually, this is only true when a browser requests content. If a proxy server requests content, then the URI must be the complete URL.

Aside Passing arguments in HTTP POST requests

Arguments for HTTP POST requests are passed in the request body rather than in the URI.

Status code	Status message	Description
200	OK	Request was handled without error.
301	Moved permanently	Content has moved to the hostname in the Location header.
400	Bad request	Request could not be understood by the server.
403	Forbidden	Server lacks permission to access the requested file.
404	Not found	Server could not find the requested file.
501	Not implemented	Server does not support the request method.
505	HTTP version not supported	Server does not support version in request.

Figure 11.25 **Some HTTP status codes.**

The response headers in lines 9–13 provide additional information about the response. For our purposes, the two most important headers are `Content-Type` (line 12), which tells the client the MIME type of the content in the response body, and `Content-Length` (line 13), which indicates its size in bytes.

The empty text line in line 14 that terminates the response headers is followed by the response body, which contains the requested content.

11.5.4 Serving Dynamic Content

If we stop to think for a moment how a server might provide dynamic content to a client, certain questions arise. For example, how does the client pass any program arguments to the server? How does the server pass these arguments to the child process that it creates? How does the server pass other information to the child that it might need to generate the content? Where does the child send its output? These questions are addressed by a de facto standard called *CGI (common gateway interface)*.

How Does the Client Pass Program Arguments to the Server?

Arguments for GET requests are passed in the URI. As we have seen, a '`?`' character separates the filename from the arguments, and each argument is separated by an '`&`' character. Spaces are not allowed in arguments and must be represented with the `%20` string. Similar encodings exist for other special characters.

How Does the Server Pass Arguments to the Child?

After a server receives a request such as

```
GET /cgi-bin/adder?15000&213 HTTP/1.1
```

Environment variable	Description
QUERY_STRING	Program arguments
SERVER_PORT	Port that the parent is listening on
REQUEST_METHOD	GET or POST
REMOTE_HOST	Domain name of client
REMOTE_ADDR	Dotted-decimal IP address of client
CONTENT_TYPE	POST only: MIME type of the request body
CONTENT_LENGTH	POST only: Size in bytes of the request body

Figure 11.26 Examples of CGI environment variables.

it calls `fork` to create a child process and calls `execve` to run the `/cgi-bin/adder` program in the context of the child. Programs like the `adder` program are often referred to as *CGI programs* because they obey the rules of the CGI standard. Before the call to `execve`, the child process sets the CGI environment variable QUERY_STRING to `15000&213`, which the `adder` program can reference at run time using the Linux `getenv` function.

How Does the Server Pass Other Information to the Child?

CGI defines a number of other environment variables that a CGI program can expect to be set when it runs. Figure 11.26 shows a subset.

Where Does the Child Send Its Output?

A CGI program sends its dynamic content to the standard output. Before the child process loads and runs the CGI program, it uses the Linux `dup2` function to redirect standard output to the connected descriptor that is associated with the client. Thus, anything that the CGI program writes to standard output goes directly to the client.

Notice that since the parent does not know the type or size of the content that the child generates, the child is responsible for generating the `Content-type` and `Content-length` response headers, as well as the empty line that terminates the headers.

Figure 11.27 shows a simple CGI program that sums its two arguments and returns an HTML file with the result to the client. Figure 11.28 shows an HTTP transaction that serves dynamic content from the `adder` program.

Practice Problem 11.5 (solution page 1005)

Assume that a CGI program needs to send dynamic content to the client. This is typically done by making the CGI program send its content to the standard output. Explain how this content is sent to the client.

Aside Passing arguments in HTTP POST requests to CGI programs

For POST requests, the child would also need to redirect standard input to the connected descriptor. The CGI program would then read the arguments in the request body from standard input.

code/netp/tiny/cgi-bin/adder.c

```
#include "csapp.h"

int main(void) {
    char *buf, *p;
    char arg1[MAXLINE], arg2[MAXLINE], content[MAXLINE];
    int n1=0, n2=0;

    /* Extract the two arguments */
    if ((buf = getenv("QUERY_STRING")) != NULL) {
        p = strchr(buf, '&');
        *p = '\0';
        strcpy(arg1, buf);
        strcpy(arg2, p+1);
        n1 = atoi(arg1);
        n2 = atoi(arg2);
    }

    /* Make the response body */
    sprintf(content, "QUERY_STRING=%s", buf);
    sprintf(content, "Welcome to add.com: ");
    sprintf(content, "%sTHE Internet addition portal.\r\n<p>", content);
    sprintf(content, "%sThe answer is: %d + %d = %d\r\n<p>",
            content, n1, n2, n1 + n2);
    sprintf(content, "%sThanks for visiting!\r\n", content);

    /* Generate the HTTP response */
    printf("Connection: close\r\n");
    printf("Content-length: %d\r\n", (int)strlen(content));
    printf("Content-type: text/html\r\n\r\n");
    printf("%s", content);
    fflush(stdout);

    exit(0);
}
```

code/netp/tiny/cgi-bin/adder.c

Figure 11.27 CGI program that sums two integers.

```
linux> telnet kittyhawk.cmcl.cs.cmu.edu 8000  Client: open connection
Trying 128.2.194.242...
Connected to kittyhawk.cmcl.cs.cmu.edu.
Escape character is '^]'.
GET /cgi-bin/adder?15000&213 HTTP/1.0  Client: request line
                                       Client: empty line terminates headers
HTTP/1.0 200 OK                        Server: response line
Server: Tiny Web Server                Server: identify server
Content-length: 115                    Adder: expect 115 bytes in response body
Content-type: text/html                Adder: expect HTML in response body
                                       Adder: empty line terminates headers
Welcome to add.com: THE Internet addition portal. Adder: first HTML line
<p>The answer is: 15000 + 213 = 15213  Adder: second HTML line in response body
<p>Thanks for visiting!                Adder: third HTML line in response body
Connection closed by foreign host.     Server: closes connection
linux>                                 Client: closes connection and terminates
```

Figure 11.28 An HTTP transaction that serves dynamic HTML content.

11.6 Putting It Together: The Tiny Web Server

We conclude our discussion of network programming by developing a small but functioning Web server called Tiny. Tiny is an interesting program. It combines many of the ideas that we have learned about, such as process control, Unix I/O, the sockets interface, and HTTP, in only 250 lines of code. While it lacks the functionality, robustness, and security of a real server, it is powerful enough to serve both static and dynamic content to real Web browsers. We encourage you to study it and implement it yourself. It is quite exciting (even for the authors!) to point a real browser at your own server and watch it display a complicated Web page with text and graphics.

The Tiny `main` Routine

Figure 11.29 shows Tiny's main routine. Tiny is an iterative server that listens for connection requests on the port that is passed in the command line. After opening a listening socket by calling the `open_listenfd` function, Tiny executes the typical infinite server loop, repeatedly accepting a connection request (line 32), performing a transaction (line 36), and closing its end of the connection (line 37).

The `doit` Function

The `doit` function in Figure 11.30 handles one HTTP transaction. First, we read and parse the request line (lines 11–14). Notice that we are using the `rio_readlineb` function from Figure 10.8 to read the request line.

Tiny supports only the GET method. If the client requests another method (such as POST), we send it an error message and return to the main routine

code/netp/tiny/tiny.c

```c
/*
 * tiny.c - A simple, iterative HTTP/1.0 Web server that uses the
 *     GET method to serve static and dynamic content
 */
#include "csapp.h"

void doit(int fd);
void read_requesthdrs(rio_t *rp);
int parse_uri(char *uri, char *filename, char *cgiargs);
void serve_static(int fd, char *filename, int filesize);
void get_filetype(char *filename, char *filetype);
void serve_dynamic(int fd, char *filename, char *cgiargs);
void clienterror(int fd, char *cause, char *errnum,
                 char *shortmsg, char *longmsg);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    char hostname[MAXLINE], port[MAXLINE];
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    /* Check command-line args */
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(1);
    }

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen, hostname, MAXLINE,
                    port, MAXLINE, 0);
        printf("Accepted connection from (%s, %s)\n", hostname, port);
        doit(connfd);
        Close(connfd);
    }
}
```

code/netp/tiny/tiny.c

Figure 11.29 **The TINY Web server.**

code/netp/tiny/tiny.c

```
void doit(int fd)
{
    int is_static;
    struct stat sbuf;
    char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
    char filename[MAXLINE], cgiargs[MAXLINE];
    rio_t rio;

    /* Read request line and headers */
    Rio_readinitb(&rio, fd);
    Rio_readlineb(&rio, buf, MAXLINE);
    printf("Request headers:\n");
    printf("%s", buf);
    sscanf(buf, "%s %s %s", method, uri, version);
    if (strcasecmp(method, "GET")) {
       clienterror(fd, method, "501", "Not implemented",
                "Tiny does not implement this method");
        return;
    }
    read_requesthdrs(&rio);

    /* Parse URI from GET request */
    is_static = parse_uri(uri, filename, cgiargs);
    if (stat(filename, &sbuf) < 0) {
        clienterror(fd, filename, "404", "Not found",
                    "Tiny couldn't find this file");
        return;
    }

    if (is_static) { /* Serve static content */
        if (!(S_ISREG(sbuf.st_mode)) || !(S_IRUSR & sbuf.st_mode)) {
            clienterror(fd, filename, "403", "Forbidden",
                        "Tiny couldn't read the file");
            return;
        }
        serve_static(fd, filename, sbuf.st_size);
    }
    else { /* Serve dynamic content */
        if (!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode)) {
            clienterror(fd, filename, "403", "Forbidden",
                        "Tiny couldn't run the CGI program");
            return;
        }
        serve_dynamic(fd, filename, cgiargs);
    }
}
```

code/netp/tiny/tiny.c

Figure 11.30 TINY `doit` **handles one HTTP transaction.**

(lines 15–19), which then closes the connection and awaits the next connection request. Otherwise, we read and (as we shall see) ignore any request headers (line 20).

Next, we parse the URI into a filename and a possibly empty CGI argument string, and we set a flag that indicates whether the request is for static or dynamic content (line 23). If the file does not exist on disk, we immediately send an error message to the client and return.

Finally, if the request is for static content, we verify that the file is a regular file and that we have read permission (line 31). If so, we serve the static content (line 36) to the client. Similarly, if the request is for dynamic content, we verify that the file is executable (line 39), and, if so, we go ahead and serve the dynamic content (line 44).

The `clienterror` Function

TINY lacks many of the error-handling features of a real server. However, it does check for some obvious errors and reports them to the client. The `clienterror` function in Figure 11.31 sends an HTTP response to the client with the appropriate

code/netp/tiny/tiny.c

```
void clienterror(int fd, char *cause, char *errnum,
                 char *shortmsg, char *longmsg)
{
    char buf[MAXLINE], body[MAXBUF];

    /* Build the HTTP response body */
    sprintf(body, "<html><title>Tiny Error</title>");
    sprintf(body, "%s<body bgcolor=""ffffff"">\r\n", body);
    sprintf(body, "%s%s: %s\r\n", body, errnum, shortmsg);
    sprintf(body, "%s<p>%s: %s\r\n", body, longmsg, cause);
    sprintf(body, "%s<hr><em>The Tiny Web server</em>\r\n", body);

    /* Print the HTTP response */
    sprintf(buf, "HTTP/1.0 %s %s\r\n", errnum, shortmsg);
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Content-type: text/html\r\n");
    Rio_writen(fd, buf, strlen(buf));
    sprintf(buf, "Content-length: %d\r\n\r\n", (int)strlen(body));
    Rio_writen(fd, buf, strlen(buf));
    Rio_writen(fd, body, strlen(body));
}
```

code/netp/tiny/tiny.c

Figure 11.31 TINY `clienterror` **sends an error message to the client.**

code/netp/tiny/tiny.c

```
1   void read_requesthdrs(rio_t *rp)
2   {
3       char buf[MAXLINE];
4
5       Rio_readlineb(rp, buf, MAXLINE);
6       while(strcmp(buf, "\r\n")) {
7           Rio_readlineb(rp, buf, MAXLINE);
8           printf("%s", buf);
9       }
10      return;
11  }
```

code/netp/tiny/tiny.c

Figure 11.32 TINY `read_requesthdrs` **reads and ignores request headers.**

status code and status message in the response line, along with an HTML file in the response body that explains the error to the browser's user.

Recall that an HTML response should indicate the size and type of the content in the body. Thus, we have opted to build the HTML content as a single string so that we can easily determine its size. Also, notice that we are using the robust `rio_writen` function from Figure 10.4 for all output.

The `read_requesthdrs` Function

TINY does not use any of the information in the request headers. It simply reads and ignores them by calling the `read_requesthdrs` function in Figure 11.32. Notice that the empty text line that terminates the request headers consists of a carriage return and line feed pair, which we check for in line 6.

The `parse_uri` Function

TINY assumes that the home directory for static content is its current directory and that the home directory for executables is `./cgi-bin`. Any URI that contains the string `cgi-bin` is assumed to denote a request for dynamic content. The default filename is `./home.html`.

The `parse_uri` function in Figure 11.33 implements these policies. It parses the URI into a filename and an optional CGI argument string. If the request is for static content (line 5), we clear the CGI argument string (line 6) and then convert the URI into a relative Linux pathname such as `./index.html` (lines 7–8). If the URI ends with a '`/`' character (line 9), then we append the default filename (line 10). On the other hand, if the request is for dynamic content (line 13), we extract any CGI arguments (lines 14–20) and convert the remaining portion of the URI to a relative Linux filename (lines 21–22).

code/netp/tiny/tiny.c

```
int parse_uri(char *uri, char *filename, char *cgiargs)
{
    char *ptr;

    if (!strstr(uri, "cgi-bin")) {  /* Static content */
        strcpy(cgiargs, "");
        strcpy(filename, ".");
        strcat(filename, uri);
        if (uri[strlen(uri)-1] == '/')
            strcat(filename, "home.html");
        return 1;
    }
    else {  /* Dynamic content */
        ptr = index(uri, '?');
        if (ptr) {
            strcpy(cgiargs, ptr+1);
            *ptr = '\0';
        }
        else
            strcpy(cgiargs, "");
        strcpy(filename, ".");
        strcat(filename, uri);
        return 0;
    }
}
```

code/netp/tiny/tiny.c

Figure 11.33 Tiny `parse_uri` **parses an HTTP URI.**

The `serve_static` Function

Tiny serves five common types of static content: HTML files, unformatted text files, and images encoded in GIF, PNG, and JPEG formats.

The `serve_static` function in Figure 11.34 sends an HTTP response whose body contains the contents of a local file. First, we determine the file type by inspecting the suffix in the filename (line 7) and then send the response line and response headers to the client (lines 8–13). Notice that a blank line terminates the headers.

Next, we send the response body by copying the contents of the requested file to the connected descriptor `fd`. The code here is somewhat subtle and needs to be studied carefully. Line 18 opens `filename` for reading and gets its descriptor. In line 19, the Linux `mmap` function maps the requested file to a virtual memory area. Recall from our discussion of `mmap` in Section 9.8 that the call to `mmap` maps the

code/netp/tiny/tiny.c

```c
void serve_static(int fd, char *filename, int filesize)
{
    int srcfd;
    char *srcp, filetype[MAXLINE], buf[MAXBUF];

    /* Send response headers to client */
    get_filetype(filename, filetype);
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Tiny Web Server\r\n", buf);
    sprintf(buf, "%sConnection: close\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
    sprintf(buf, "%sContent-type: %s\r\n\r\n", buf, filetype);
    Rio_writen(fd, buf, strlen(buf));
    printf("Response headers:\n");
    printf("%s", buf);

    /* Send response body to client */
    srcfd = Open(filename, O_RDONLY, 0);
    srcp = Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
    Close(srcfd);
    Rio_writen(fd, srcp, filesize);
    Munmap(srcp, filesize);
}

/*
 * get_filetype - Derive file type from filename
 */
void get_filetype(char *filename, char *filetype)
{
    if (strstr(filename, ".html"))
        strcpy(filetype, "text/html");
    else if (strstr(filename, ".gif"))
        strcpy(filetype, "image/gif");
    else if (strstr(filename, ".png"))
        strcpy(filetype, "image/png");
    else if (strstr(filename, ".jpg"))
        strcpy(filetype, "image/jpeg");
    else
        strcpy(filetype, "text/plain");
}
```

code/netp/tiny/tiny.c

Figure 11.34 TINY `serve_static` **serves static content to a client.**

first `filesize` bytes of file `srcfd` to a private read-only area of virtual memory that starts at address `srcp`.

Once we have mapped the file to memory, we no longer need its descriptor, so we close the file (line 20). Failing to do this would introduce a potentially fatal memory leak. Line 21 performs the actual transfer of the file to the client. The `rio_writen` function copies the `filesize` bytes starting at location `srcp` (which of course is mapped to the requested file) to the client's connected descriptor. Finally, line 22 frees the mapped virtual memory area. This is important to avoid a potentially fatal memory leak.

The `serve_dynamic` Function

Tiny serves any type of dynamic content by forking a child process and then running a CGI program in the context of the child.

The `serve_dynamic` function in Figure 11.35 begins by sending a response line indicating success to the client, along with an informational `Server` header. The CGI program is responsible for sending the rest of the response. Notice that this is not as robust as we might wish, since it doesn't allow for the possibility that the CGI program might encounter some error.

After sending the first part of the response, we fork a new child process (line 11). The child initializes the QUERY_STRING environment variable with the CGI arguments from the request URI (line 13). Notice that a real server would

code/netp/tiny/tiny.c

```
1   void serve_dynamic(int fd, char *filename, char *cgiargs)
2   {
3       char buf[MAXLINE], *emptylist[] = { NULL };
4
5       /* Return first part of HTTP response */
6       sprintf(buf, "HTTP/1.0 200 OK\r\n");
7       Rio_writen(fd, buf, strlen(buf));
8       sprintf(buf, "Server: Tiny Web Server\r\n");
9       Rio_writen(fd, buf, strlen(buf));
10
11      if (Fork() == 0) { /* Child */
12          /* Real server would set all CGI vars here */
13          setenv("QUERY_STRING", cgiargs, 1);
14          Dup2(fd, STDOUT_FILENO);         /* Redirect stdout to client */
15          Execve(filename, emptylist, environ); /* Run CGI program */
16      }
17      Wait(NULL); /* Parent waits for and reaps child */
18  }
```

code/netp/tiny/tiny.c

Figure 11.35 Tiny `serve_dynamic` **serves dynamic content to a client.**

Aside Dealing with prematurely closed connections

Although the basic functions of a Web server are quite simple, we don't want to give you the false impression that writing a real Web server is easy. Building a robust Web server that runs for extended periods without crashing is a difficult task that requires a deeper understanding of Linux systems programming than we've learned here. For example, if a server writes to a connection that has already been closed by the client (say, because you clicked the "Stop" button on your browser), then the first such write returns normally, but the second write causes the delivery of a SIGPIPE signal whose default behavior is to terminate the process. If the SIGPIPE signal is caught or ignored, then the second write operation returns −1 with `errno` set to EPIPE. The `strerr` and `perror` functions report the EPIPE error as a "Broken pipe," a nonintuitive message that has confused generations of students. The bottom line is that a robust server must catch these SIGPIPE signals and check `write` function calls for EPIPE errors.

set the other CGI environment variables here as well. For brevity, we have omitted this step.

Next, the child redirects the child's standard output to the connected file descriptor (line 14) and then loads and runs the CGI program (line 15). Since the CGI program runs in the context of the child, it has access to the same open files and environment variables that existed before the call to the `execve` function. Thus, everything that the CGI program writes to standard output goes directly to the client process, without any intervention from the parent process. Meanwhile, the parent blocks in a call to `wait`, waiting to reap the child when it terminates (line 17).

11.7 Summary

Every network application is based on the client-server model. With this model, an application consists of a server and one or more clients. The server manages resources, providing a service for its clients by manipulating the resources in some way. The basic operation in the client-server model is a client-server transaction, which consists of a request from a client, followed by a response from the server.

Clients and servers communicate over a global network known as the Internet. From a programmer's point of view, we can think of the Internet as a worldwide collection of hosts with the following properties: (1) Each Internet host has a unique 32-bit name called its IP address. (2) The set of IP addresses is mapped to a set of Internet domain names. (3) Processes on different Internet hosts can communicate with each other over connections.

Clients and servers establish connections by using the sockets interface. A socket is an end point of a connection that is presented to applications in the form of a file descriptor. The sockets interface provides functions for opening and closing socket descriptors. Clients and servers communicate with each other by reading and writing these descriptors.

Web servers and their clients (such as browsers) communicate with each other using the HTTP protocol. A browser requests either static or dynamic content from the server. A request for static content is served by fetching a file from the server's disk and returning it to the client. A request for dynamic content is served by running a program in the context of a child process on the server and returning its output to the client. The CGI standard provides a set of rules that govern how the client passes program arguments to the server, how the server passes these arguments and other information to the child process, and how the child sends its output back to the client. A simple but functioning Web server that serves both static and dynamic content can be implemented in a few hundred lines of C code.

Bibliographic Notes

The official source of information for the Internet is contained in a set of freely available numbered documents known as *RFCs (requests for comments)*. A searchable index of RFCs is available on the Web at

http://rfc-editor.org

RFCs are typically written for developers of Internet infrastructure, and thus they are usually too detailed for the casual reader. However, for authoritative information, there is no better source. The HTTP/1.1 protocol is documented in RFC 2616. The authoritative list of MIME types is maintained at

http://www.iana.org/assignments/media-types

Kerrisk is the bible for all aspects of Linux programming and provides a detailed discussion of modern network programming [62]. There are a number of good general texts on computer networking [65, 84, 114]. The great technical writer W. Richard Stevens developed a series of classic texts on such topics as advanced Unix programming [111], the Internet protocols [109, 120, 107], and Unix network programming [108, 110]. Serious students of Unix systems programming will want to study all of them. Tragically, Stevens died on September 1, 1999. His contributions are greatly missed.

Homework Problems

11.6 ◆◆

A. Modify Tiny so that it echoes every request line and request header.

B. Use your favorite browser to make a request to Tiny for static content. Capture the output from Tiny in a file.

C. Inspect the output from Tiny to determine the version of HTTP your browser uses.

D. Consult the HTTP/1.1 standard in RFC 2616 to determine the meaning of each header in the HTTP request from your browser. You can obtain RFC 2616 from www.rfc-editor.org/rfc.html.

11.7 ◆◆

Extend Tiny so that it serves MPG video files. Check your work using a real browser.

11.8 ◆◆

Modify Tiny so that it reaps CGI children inside a SIGCHLD handler instead of explicitly waiting for them to terminate.

11.9 ◆◆

Modify Tiny so that when it serves static content, it copies the requested file to the connected descriptor using `malloc`, `rio_readn`, and `rio_writen`, instead of `mmap` and `rio_writen`.

11.10 ◆◆

A. Write an HTML form for the CGI `adder` function in Figure 11.27. Your form should include two text boxes that users fill in with the two numbers to be added together. Your form should request content using the GET method.

B. Check your work by using a real browser to request the form from Tiny, submit the filled-in form to Tiny, and then display the dynamic content generated by `adder`.

11.11 ◆◆

Extend Tiny to support the HTTP HEAD method. Check your work using TELNET as a Web client.

11.12 ◆◆◆

Extend Tiny so that it serves dynamic content requested by the HTTP POST method. Check your work using your favorite Web browser.

11.13 ◆◆◆

Modify Tiny so that it deals cleanly (without terminating) with the SIGPIPE signals and EPIPE errors that occur when the `write` function attempts to write to a prematurely closed connection.

Solutions to Practice Problems

Solution to Problem 11.1 (page 963)

Dotted-decimal address	Hex address
`107.212.122.205`	`0x6BD47ACD`
`64.12.149.13`	`0x400C950D`
`107.212.96.29`	`0x6BD4601D`
`[0.0].[0.128]`	`0x00000080`

Dotted-decimal address	Hex address
[255.255].[255.0]	0xFFFFFF00
[10.1].[1.64]	0x0A010140

Solution to Problem 11.2 (page 963)

code/netp/global-hex2dd.c

```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct in_addr inaddr;  /* Address in network byte order */
    uint16_t addr;          /* Address in host byte order */
    char buf[MAXBUF];       /* Buffer for dotted-decimal string */

    if (argc != 2) {
        fprintf(stderr, "usage: %s <hex number>\n", argv[0]);
        exit(0);
    }
    sscanf(argv[1], "%x", &addr);
    inaddr.s_addr = htons(addr);

    if (!inet_ntop(AF_INET, &inaddr, buf, MAXBUF))
        unix_error("inet_ntop");
    printf("%s\n", buf);

    exit(0);
}
```

code/net/global-hex2dd.c

Solution to Problem 11.3 (page 963)

code/netp/global-dd2hex.c

```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct in_addr inaddr;  /* Address in network byte order */
    int rc;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <network byte order>\n", argv[0]);
        exit(0);
    }

    rc = inet_pton(AF_INET, argv[1], &inaddr);
    if (rc == 0)
        app_error("inet_pton error: invalid network byte order");
    else if (rc < 0)
        unix_error("inet_pton error");

    printf("0x%x\n", ntohs(inaddr.s_addr));
    exit(0);
}
```

code/netp/global-dd2hex.c

Solution to Problem 11.4 (page 978)

Here's a solution. Notice how much more difficult it is to use `inet_ntop`, which requires messy casting and deep structure references. The `getnameinfo` function is much simpler because it does all of that work for us.

code/netp/hostinfo-ntop.c

```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct addrinfo *p, *listp, hints;
    struct sockaddr_in *sockp;
    char buf[MAXLINE];
    int rc;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <domain name>\n", argv[0]);
        exit(0);
    }

    /* Get a list of addrinfo records */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;       /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* Connections only */
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
        exit(1);
    }

    /* Walk the list and display each associated IP address */
    for (p = listp; p; p = p->ai_next) {
        sockp = (struct sockaddr_in *)p->ai_addr;
        Inet_ntop(AF_INET, &(sockp->sin_addr), buf, MAXLINE);
        printf("%s\n", buf);
    }

    /* Clean up */
    Freeaddrinfo(listp);

    exit(0);
}
```

code/netp/hostinfo-ntop.c

Solution to Problem 11.5 (page 990)

Before the process that runs the CGI program is loaded, a Linux `dup2` function is used to redirect standard output to the connected descriptor that is associated with the client. Thus, anything that the CGI program writes to standard output goes directly to the client.