

Chapter 12

Network Programming

Network applications are everywhere. Any time you browse the Web, send an email message, or pop up an X window, 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. We 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 small but functional Web server that can serve both static and dynamic content with text and graphics to real Web browsers.

12.1 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 12.1). A client-server trans-

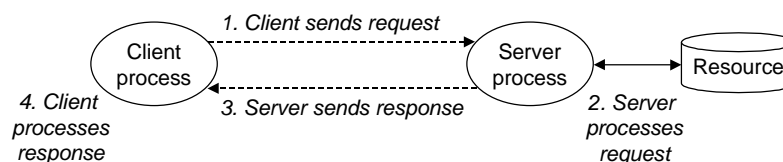


Figure 12.1: A client-server transaction.

action 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.
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.

Aside: Client-server transactions vs. 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. **End Aside.**

12.2 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 12.2. An adapter plugged into an expansion slot on the I/O bus provides the physical interface to the network. Data received from the network is 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 1 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 12.3. 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 non-volatile 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

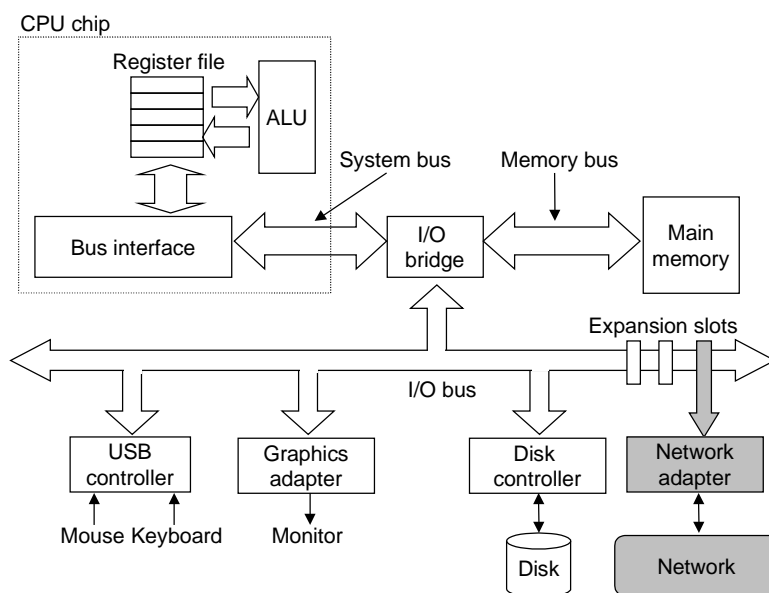


Figure 12.2: **Hardware organization of a network host.**

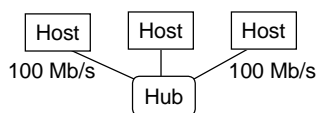


Figure 12.3: **Ethernet segment.**

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 12.4. 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.

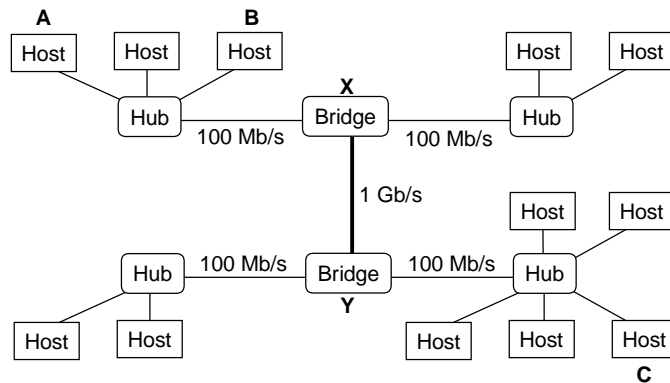


Figure 12.4: **Bridged Ethernet segments.**

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 bridge C's segment.

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 12.5.

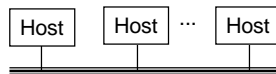


Figure 12.5: **Conceptual view of a LAN.**

At a higher level in the hierarchy, multiple incompatible LANs can be connected by specialized computers called *routers* to form an *internet* (interconnected network).

Aside: Internet vs. internet.

We will always use lower-case *internet* to denote the general concept, and upper-case *Internet* to denote a specific implementation, namely the global IP Internet. **End Aside.**

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 12.6 shows an example internet with a pair of LANs and WANs connected by three routers.

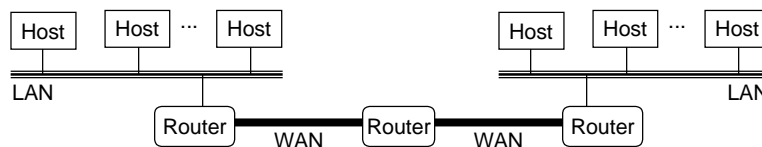


Figure 12.6: **A small internet.** Two LANs and two WANs are connected by three routers.

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 smooths 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 smooths 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 12.7 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.
3. The LAN1 adapter copies the frame to the network.

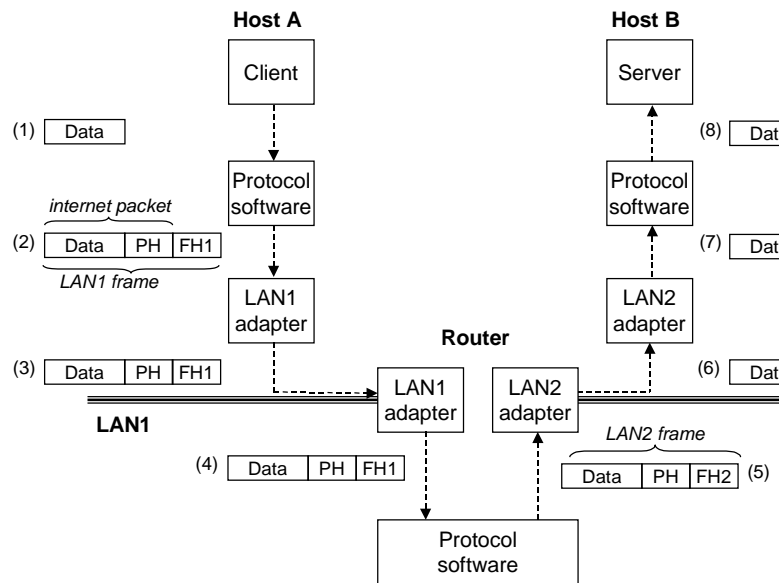


Figure 12.7: **How data travels from one host to another on an internet.** Key: PH: internet packet header, FH1: frame header for LAN1, FH2: frame header for LAN2.

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.

12.3 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 12.8 shows the basic hardware and software organization of an Internet client-server application.

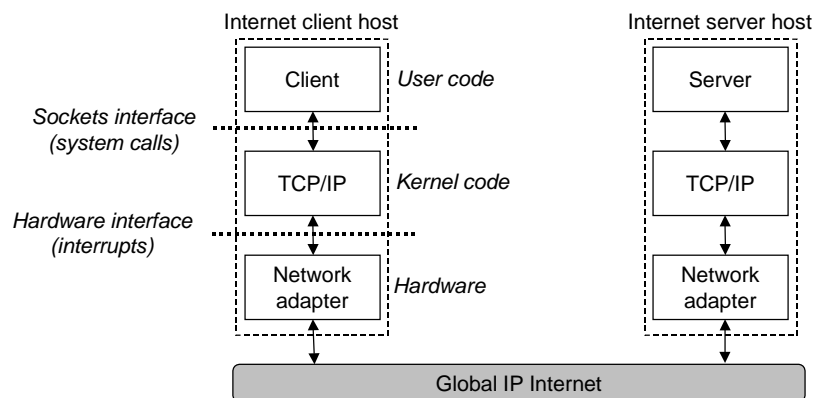


Figure 12.8: **Hardware and software organization of an Internet application.**

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 12.4.) 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, the IP protocol provides the basic naming scheme and a delivery mechanism that can send packets, known as *datagrams*, from one Internet host to any another 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 packets 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*.
- 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 next three sections discuss these fundamental Internet ideas in more detail.

12.3.1 IP Addresses

An IP address is an unsigned 32-bit integer. Network programs store IP addresses in the *IP address structure* shown in Figure 12.9.

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

netinet/in.h

Figure 12.9: IP address structure.

Aside: Why store the scalar IP address in a structure?

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. **End Aside.**

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.

<pre>#include <netinet/in.h> unsigned long int htonl(unsigned long int hostlong); unsigned short int htons(unsigned short int hostshort); unsigned long int ntohl(unsigned long int netlong); unsigned short int ntohs(unsigned short int netshort);</pre>	
	Return: value in network byte order
	Return: value in host byte order

The `htonl` function converts a 32-bit integer from host byte order to network byte order. The `ntohl` function converts a 32-bit integer from network byte order to host byte order. The `htons` and `ntohs` functions perform corresponding conversions for 16-bit integers.

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
128.2.194.242
```

Internet programs convert back and forth between IP addresses and dotted-decimal strings using the functions `inet_aton` and `inet_ntoa`:

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

int inet_aton(const char *cp, struct in_addr *inp);
                                                    Returns: 1 if OK, 0 on error

char *inet_ntoa(struct in_addr in);
                                                    Returns: pointer to a dotted-decimal string
```

The `inet_aton` function converts a dotted-decimal string (`cp`) to an IP address in network byte order (`inp`). Similarly, the `inet_ntoa` function converts an IP address in network byte order to its corresponding dotted-decimal string. Notice that a call to `inet_aton` passes a pointer to a structure, while a call to `inet_ntoa` passes the structure itself.

Aside: What do `ntoa` and `aton` mean?

The “n” denotes *network* representation. The “a” denotes *application* representation. The “to” means *to*. **End Aside.**

Practice Problem 12.1:

Complete the following table.

Hex address	Dotted-decimal address
0x0	
0xffffffff	
0xef000001	
	205.188.160.121
	64.12.149.13
	205.188.146.23

Practice Problem 12.2:

Write a program `hex2dd.c` that converts its hex argument to a dotted-decimal string and prints the result. For example,

```
unix> ./hex2dd 0x8002c2f2
128.2.194.242
```

Practice Problem 12.3:

Write a program `dd2hex.c` that converts its dotted-decimal argument to a hex number and prints the result. For example,

```
unix> ./dd2hex 128.2.194.242
0x8002c2f2
```

12.3.2 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 example,

kittyhawk.cmcl.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 12.10 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

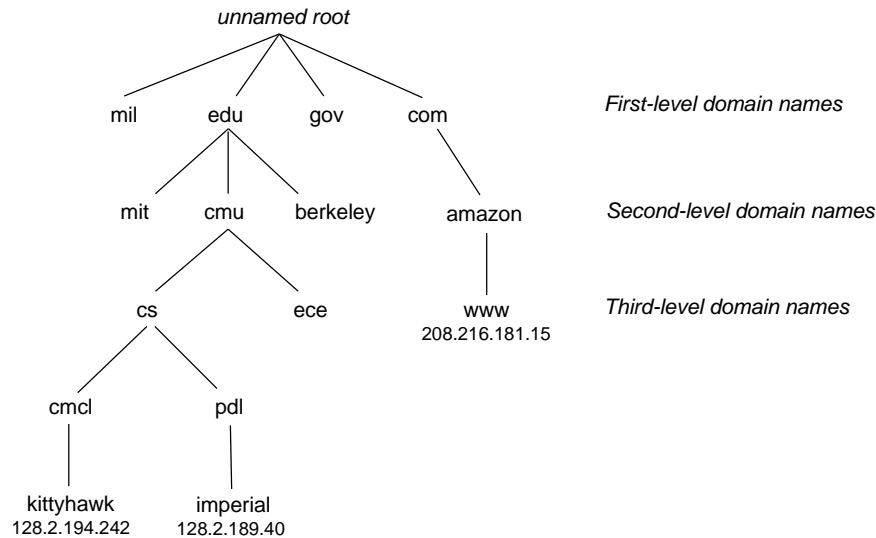


Figure 12.10: **Subset of the Internet domain name hierarchy.**

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.

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.

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 world-wide database known as *DNS* (Domain Naming System). Conceptually,

the DNS database consists of millions of the *host entry structures* shown in Figure 12.11, each of which defines the mapping between a set of domain names (an official name and a list of aliases) and a set of IP addresses. In a mathematical sense, you can think of each host entry as an equivalence class of domain names and IP addresses.

```

/* DNS host entry structure */
struct hostent {
    char    *h_name;           /* official domain name of host */
    char    **h_aliases;       /* null-terminated array of domain names */
    int     h_addrtype;        /* host address type (AF_INET) */
    int     h_length;          /* length of an address, in bytes */
    char    **h_addr_list;     /* null-terminated array of in_addr structs */
};

```

netdb.h

Figure 12.11: **DNS host entry structure.**

Internet applications retrieve arbitrary host entries from the DNS database by calling the `gethostbyname` and `gethostbyaddr` functions.

```

#include <netdb.h>

struct hostent *gethostbyname(const char *name);
                                     Returns: non-NULL pointer if OK, NULL pointer on error with h_errno set
struct hostent *gethostbyaddr(const char *addr, int len, int type);
                                     Returns: non-NULL pointer if OK, NULL pointer on error with h_errno set

```

The `gethostbyname` function returns the host entry associated with the domain name `name`. The `gethostbyaddr` function returns the host entry associated with the IP address `addr`. The second argument gives the length in bytes of an IP address, which for the current Internet is always four bytes. For our purposes, the third argument is always zero.

We can explore some of the properties of the DNS mapping with the `hostinfo` program in Figure 12.12, which reads a domain name or dotted-decimal address from the command line and displays the corresponding host entry. Each Internet host has the locally defined domain name `localhost`, which always maps to the *loopback address* `127.0.0.1`:

```

unix> ./hostinfo localhost
official hostname: localhost
alias: localhost.localdomain
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 for debugging. We can use `HOSTNAME` to determine the real domain name of our local host:

code/netp/hostinfo.c

```
1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     char **pp;
6     struct in_addr addr;
7     struct hostent *hostp;
8
9     if (argc != 2) {
10         fprintf(stderr, "usage: %s <domain name or dotted-decimal>\n",
11             argv[0]);
12         exit(0);
13     }
14
15     if (inet_aton(argv[1], &addr) != 0)
16         hostp = Gethostbyaddr((const char *)&addr, sizeof(addr), AF_INET);
17     else
18         hostp = Gethostbyname(argv[1]);
19
20     printf("official hostname: %s\n", hostp->h_name);
21
22     for (pp = hostp->h_aliases; *pp != NULL; pp++)
23         printf("alias: %s\n", *pp);
24
25     for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
26         addr.s_addr = *((unsigned int *)*pp);
27         printf("address: %s\n", inet_ntoa(addr));
28     }
29     exit(0);
30 }
```

code/netp/hostinfo.c

Figure 12.12: Retrieves and prints a DNS host entry.

```
unix> hostname
kittyhawk.cmcl.cs.cmu.edu
```

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

```
unix> ./hostinfo kittyhawk.cmcl.cs.cmu.edu
official hostname: kittyhawk.cmcl.cs.cmu.edu
address: 128.2.194.242
```

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

```
unix> ./hostinfo cs.mit.edu
official hostname: EECS.MIT.EDU
alias: cs.mit.edu
address: 18.62.1.6
```

In the most general case, multiple domain names can be mapped to multiple IP addresses:

```
unix> ./hostinfo www.aol.com
official hostname: aol.com
alias: www.aol.com
address: 205.188.160.121
address: 64.12.149.13
address: 205.188.146.23
```

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

```
unix> ./hostinfo edu
Gethostbyname error: No address associated with name
unix> ./hostinfo cmcl.cs.cmu.edu
Gethostbyname error: No address associated with name
```

Aside: How many Internet hosts are there?

Twice a year since 1987, the Internet Software Consortium (www.isc.org) 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 roughly doubled each year. By June 2001, there were more than 120,000,000 Internet hosts.
End Aside.

Practice Problem 12.4:

Compile the HOSTINFO program from Figure 12.12. Then run `hostinfo aol.com` three times in a row on your system.

- A. What do you notice about the ordering of the IP addresses in the three host entries?
- B. How might this ordering be useful?

12.3.3 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 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*, and is denoted by `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 associated with the service. For example, Web servers typically use port 80, and email servers use port 25. On Unix machines, the file `/etc/services` contains a comprehensive list of the services provided on that machine, along with their well-known ports.

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:cliport, servaddr: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 12.13 shows a connection between a Web client and a Web server.

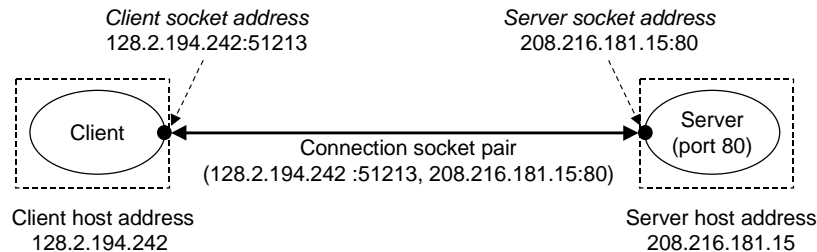


Figure 12.13: **Anatomy of an Internet connection**

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
```

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 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 Administration (ARPA), whose charter was to reestablish the U.S. 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 1000 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 later 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 a dozen or so public network access points. **End Aside.**

12.4 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, Windows, and Macintosh systems. Figure 12.14 gives an overview of the sockets interface in the context of a typical client-server transaction. You should use this picture as road map when we discuss the individual functions.

Aside: Origins of the sockets interface.

The 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. **End Aside.**

12.4.1 Socket Address Structures

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

Internet socket addresses are stored in 16-byte structures of the type `sockaddr_in` shown in Figure 12.15. For Internet applications, the `sin_family` member is `AF_INET`, the `sin_port` member is a 16-bit port number, and the `sin_addr` member is a 32-bit IP address. The IP address and port number are always stored in network (big-endian) byte order.