8

Networking and Sockets

The IPC mechanisms in Chapters 6 and 7 have their uses, of course, but many modern applications need to send data between machines, not just between processes on the same machine. By "between machines," I don't just mean machines across the room or even in the same building. I mean anywhere in the world, using the Internet.

The basic mechanism for going between machines—networking—is called *sockets*. There are eight basic socket system calls, five of which are unique to sockets: `socket`, `bind`, `listen`, `accept`, `connect`, `read`, `write`, and `close`. However, because sockets, and especially the underlying communication protocols, can get complicated, there are 60 or so other system calls involved, all of which are described in this chapter.

While there's enough here to get you started, along with lots of examples, including a Web browser and server, you'll probably need more advanced references as you get deeper into UNIX networking. The most complete book by far is [Ste2003]. You'll also need to be familiar with your system's documentation and with details of the protocol you're using, especially if it's more exotic than TCP/IP.

Here's how we'll proceed: First, I'll explain the basic socket-related system calls and give some simple client/server examples. Then I'll talk about socket addresses and socket options. I'll introduce a simple interface that hides a lot of the complexity and use it to implement a socket-based version of the Simple Messaging Interface (SMI) that I discussed extensively in Chapter 7. Next come more advanced topics: connectionless sockets, out-of-band data, network database functions, and some other miscellaneous functions. Finally, I'll say a bit about the challenge of building servers that can handle thousands of clients at once.

8.1 Socket Basics

This section introduces sockets and describes the basic socket system calls.

8.1.1 How Sockets Work

Sockets are complicated, so let's start with something we already know. Recall from Section 7.2 that a process can open a FIFO like this:

```
fd = open("MyFifo", O_RDONLY);
```

Now consider what this system call does underneath the covers:

1. It creates an endpoint for I/O and allocates a file descriptor for it.
2. It binds the file descriptor to the external name "MyFifo."
3. It waits until there's a writer.
4. It returns with the file descriptor, which can be used in a `read` system call.

Sockets work similarly, except that each step is broken down into a separate system call:

`socket` Creates an endpoint and allocates a file descriptor.

`bind` Associates the socket with an external name so other processes can refer to it.

`listen` Marks the socket as being able to accept connections from other sockets.

`accept` Blocks waiting for a connection.

`connect` Connects to a socket that is blocked in an `accept`.

FIFOs are symmetrical between client and server in the sense that both execute exactly the same system call, `open`, but with different flags (e.g., `O_RDONLY` vs. `O_WRONLY`). But connected sockets are usually asymmetrical because client and server use a different sequence of system calls, as shown in Figure 8.1.

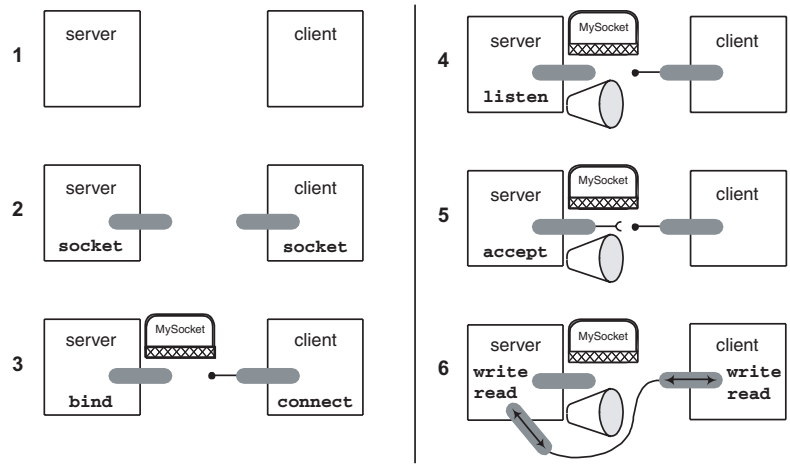

Figure 8.1 Setting Up Connected Sockets.

The server:

- Calls `socket` to create the endpoint and create a file descriptor (Step 2).
- Calls `bind` to bind the socket to a name (Step 3).
- Calls `listen` to mark it as accepting connections (Step 4).
- Calls `accept` to block until a connection is made. Then `accept` creates a second socket, with a new file descriptor (Step 5).
- Uses the new file descriptor to read and write data on the second socket (Step 6).

The client:

- Calls `socket` to create the endpoint and create a file descriptor (Step 2).
- Calls `connect`, with the server's bound name as an argument, to block until the connection is accepted by the server (Step 3, although it need not be synchronized with the server's Step 3).
- Uses the socket's file descriptor to read and write data (Step 6).

Before I formally introduce any socket system calls, let's look at an example program that forks to create a child process that acts as a client, while the parent acts as a server:

```
#define SOCKETNAME "MySocket"

int main(void)
{
    struct sockaddr_un sa;

    (void)unlink(SOCKETNAME);
    strcpy(sa.sun_path, SOCKETNAME);
    sa.sun_family = AF_UNIX;
    if (fork() == 0) { /* child -- client */
        int fd_skt;
        char buf[100];

        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
        while (connect(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) == -1)
            if (errno == ENOENT) {
                sleep(1);
                continue;
            }
            else
                EC_FAIL
        ec_neg1( write(fd_skt, "Hello!", 7 ) )
        ec_neg1( read(fd_skt, buf, sizeof(buf)) )
        printf("Client got \"%s\"\n", buf);
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    else { /* parent -- server */
        int fd_skt, fd_client;
        char buf[100];

        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
        ec_neg1( bind(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) )
        ec_neg1( listen(fd_skt, SOMAXCONN) )
        ec_neg1( fd_client = accept(fd_skt, NULL, 0) )
        ec_neg1( read(fd_client, buf, sizeof(buf)) )
        printf("Server got \"%s\"\n", buf);
        ec_neg1( write(fd_client, "Goodbye!", 9 ) )
        ec_neg1( close(fd_skt) )
        ec_neg1( close(fd_client) )
        exit(EXIT_SUCCESS);
    }

EC_CLEANUP_BGN
        exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The first thing we see is that the name passed to `bind` and `connect` isn't just a string, but a `sockaddr` structure that contains the string in the `sun_path` mem-

ber and an address family in the `sun_family` member. Socket addresses are a huge subject in themselves, which I'll get to, but for now just assume that `AF_UNIX` means "local communication within a single machine," which is all this example needs.

The socket file name is unlinked because, for `AF_UNIX`, `bind` won't reuse an existing name, and we want to ensure that it starts with a fresh one.

The server (parent process) follows the six-step sequence in Figure 8.1. (Ignore the mysterious arguments to `listen` and `accept` for now.) The client (child process) also follows the steps in Figure 8.1 but with a twist: If it gets to its `connect` before the server gets to its `bind`, `connect` will fail because the socket file isn't there yet. In that case we sleep and try again.

When the server returns from `accept`, it proceeds to read and write the file descriptor it got from `accept` (*not* the original socket's file descriptor). The client writes and reads its socket file descriptor (the only one it has) when it gets a return from `connect`. Remember that `accept` and `connect` are the two calls that block—the others just do their work and return right away.

The connection stays up as long as the server and client want it to, acting like a bidirectional pipe (Section 6.6).

The socket file really is a file of that type, as shown by the ls command:

```
$ ls -l MySocket
srwxr-xr-x   1 marc     sysadmin       0 Apr  4 10:03 MySocket
```

Here's the output I got when I ran the example program:

```
Server got "Hello!"
Client got "Goodbye!"
```

So, sockets are really pretty simple, except for some details about:

- Handling multiple clients from the same server
- Address families, including the interesting `AF_INET` family, for network communication
- Nonstream-oriented communication, involving datagrams instead of just streams of data
- Connectionless communication, in which processes send datagrams to named receivers, instead of setting up a semipermanent connection, as in my example

- Lots of options that fine-tune the behavior of the socket, especially for `AF_INET` and `AF_INET6`
- Passing binary data between machines with different ways of storing numbers
- Accessing databases of names (e.g., *www.basepath.com/aup*)

Well, actually, these aren't mere details. They're important topics that you need to know about to be able to build networking applications, and they fill the rest of this chapter.

8.1.2 Basic System Calls For Connected Sockets

As I indicated, there are connected sockets, in which a channel is set up by `accept` and `connect`, and connectionless sockets, in which datagrams are sent to named receivers (`listen` and `accept` aren't used, and `connect` is used in a different way). I'll confine my discussion in this section to connected sockets only, and defer connectionless sockets to Section 8.6.

This section covers just the basic five socket-specific system calls which I already informally introduced, starting with `socket`:

socket—create endpoint for communication

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

int socket(
    int domain,            /* domain (AF_UNIX, AF_INET, etc.) */
    int type,              /* SOCK_STREAM, SOCK_DGRAM, etc. */
    int protocol           /* specific to type; usually zero */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

The file descriptor returned from `socket` can be used for two different purposes:

- By the server, as an endpoint for accepting connections with `accept` (The actual I/O is done on the file descriptor returned from `accept`.)
- By a client, for direct I/O once `connect` has successfully connected the socket

The `domain`[1] and `type` are the same as for the socket address that will be used in a following call to `bind` or `connect`. The `protocol` argument is for making a

1. The macros starting with `AF_` are sometimes shown as starting with `PF_`, but the standard uses the `AF_` notation, which is what you should use.

selection if the type offers a choice, but in most cases the default will do, so it's zero.

Next, the server has to name the socket with `bind`:

bind—bind name to socket

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

int bind(
     int socket_fd,             /* socket file descriptor */
     const struct sockaddr *sa, /* socket address */
     socklen_t sa_len           /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Socket addresses get a whole section to themselves (Section 8.2). We've already seen an example for `AF_UNIX`, which is just for communication within a single machine, as were the IPC mechanisms in Chapter 7. Later we'll see how to set up addresses for other domains, including `AF_INET`.

Remember that on most systems, for `AF_UNIX`, `bind` will make a fresh socket file; it can't reuse an existing file.

The server also has to call `listen` to set the socket up for accepting connections:

listen—mark socket for accepting and set queue limit

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

int listen(
     int socket_fd,             /* socket file descriptor */
     int backlog                /* maximum connection queue length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Our example in the last section didn't show it, but a server can accept connections from lots of clients. Also, as it can only issue `accept`s at a certain rate, requests for connections can queue up. The second argument to `listen` limits the length of that queue. Usually an attempt by a client to `connect` when the queue is full causes its `connect` to return –1 with `errno` set to `ECONNREFUSED`. In my examples I use the constant `SOMAXCONN`, which is a system-defined maximum.

Next, the server accepts a connection:

accept—accept new connection on socket and create new socket

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

int accept(
     int socket_fd,                 /* socket file descriptor */
     struct sockaddr *sa,           /* socket address or NULL */
     socklen_t *sa_len              /* address length */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

Normally, `accept` blocks until a connection request arrives (from some other process's call to `connect`), and then it creates a new socket for the I/O on that connection and returns a new file descriptor. That file descriptor can be used with ordinary `read` and `write` system calls, although for more control over the I/O you can also use socket-specific calls—`send`, `sendto`, `sendmsg`, `recv`, `recvfrom`, and `recvmsg`—which I'll discuss in Sections 8.6.2, 8.6.3, and 8.9.1.

If the `O_NONBLOCK` flag is set for the socket file descriptor (with `fcntl`; see Section 3.8.3), `accept` doesn't wait for a connection but returns –1 immediately if no request is on the queue, with `errno` set to `EAGAIN` or `EWOULDBLOCK`.

The socket file descriptor can be used with `select` or `poll`, and it usually is, as we'll see in Section 8.1.3. Typically, the server sets up the `fd_set` for `select` (or the equivalent for `poll`) with the socket file descriptor and each file descriptor returned by `accept`. If `select` or `poll` says the socket file descriptor is ready, that means the server should do an `accept` (which won't block). If one of the others is ready, that means that data has arrived from a client and should be read.

If the `sa` argument is non-`NULL`, it's used to return the socket address that connected. You have to set `sa_len` on input to the size of the storage that `sa` points to, and on return it's set to the size of the actual address.

So much for the server. The client, after creating its socket, just calls `connect`, with the same socket address that the server used for `bind`:

connect—connect socket

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

int connect(
     int socket_fd,                 /* socket file descriptor */
     const struct sockaddr *sa,     /* socket address */
     socklen_t sa_len               /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Like `accept`, `connect` usually blocks until the connection request is accepted, but it doesn't return a file descriptor—the client does I/O on the socket file descriptor.

If the `O_NONBLOCK` flag is set, `connect` won't block waiting for a connection but will return –1 with `errno` set to `EINPROGRESS`. The connection request isn't abandoned but remains on the queue to be eventually satisfied. Further calls to `connect` during that period also return –1 but with `errno` now set to `EALREADY`. When the connection is made, the socket file descriptor can be used. If you want, you can use `select` or `poll` to wait for it to be ready for writing (not reading). A typical use for this would be an application that has some initialization to do. It issues a nonblocking `connect`, does the initialization, and then issues a `select` or `poll` to block.

8.1.3 Handling Multiple Clients

Now let's extend the example from Section 8.1.1 so the server can handle multiple clients. Here's a replacement for the server-side code in that example:

```
static bool run_server(struct sockaddr_un *sap)
{
    int fd_skt, fd_client, fd_hwm = 0, fd;
    char buf[100];
    fd_set set, read_set;
    ssize_t nread;

    ec_neg1( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
    ec_neg1( bind(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) )
    ec_neg1( listen(fd_skt, SOMAXCONN) )
    if (fd_skt > fd_hwm)
        fd_hwm = fd_skt;
    FD_ZERO(&set);
    FD_SET(fd_skt, &set);
    while (true) {
        read_set = set;
        ec_neg1( select(fd_hwm + 1, &read_set, NULL, NULL, NULL) )
        for (fd = 0; fd <= fd_hwm; fd++)
            if (FD_ISSET(fd, &read_set)) {
                if (fd == fd_skt) {
                    ec_neg1( fd_client = accept(fd_skt, NULL, 0) )
                    FD_SET(fd_client, &set);
                    if (fd_client > fd_hwm)
                        fd_hwm = fd_client;
                }
```

```
                else {
                    ec_neg1( nread = read(fd, buf, sizeof(buf)) )
                    if (nread == 0) {
                        FD_CLR(fd, &set);
                        if (fd == fd_hwm)
                            fd_hwm--;
                        ec_neg1( close(fd) )
                    }
                    else {
                        printf("Server got \"%s\"\n", buf);
                        ec_neg1( write(fd, "Goodbye!", 9 ) )
                    }
                }
            }
    }
    ec_neg1( close(fd_skt) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

Most of the new code has to do with setting up for and using `select`. I discussed it in Section 4.2.3, but here's a recap:

select—wait for I/O to be ready

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

int select(
    int nfds,                   /* highest fd + 1 */
    fd_set *readset,            /* read set or NULL */
    fd_set *writeset,           /* write set or NULL */
    fd_set *errorset,           /* error set or NULL */
    struct timeval *timeout     /* time-out (microseconds) or NULL */
);
/* Returns number of bits set or -1 on error (sets errno) */
```

There are two `fd_sets` in the function `run_server`: One, `set`, holds all the file descriptors of interest: the socket file descriptor, and one per client as returned by `accept`. We need the number of file descriptors in the set that `select` should look at, so the variable `fd_hwm` ("high-water-mark") is kept up-to-date as we get new file descriptors from `accept`. When we close the highest file descriptor, we remove it from `set` and decrement the high-water-mark. Removing it from `set` is extremely important; otherwise, `select` would keep reporting it as ready, meaning not that data is there to be read, but only that `read` will not block.

The second `fd_set`, `read_set`, is copied from `set` each time we call `select` and then is modified by `select` to indicate which file descriptors are ready.[2]

The function then loops forever, or until we kill it with a signal. Each time it gets a return from `select` it cycles through the returned set (the modified `read_set` variable) looking for a ready file descriptor. If `fd_skt` is ready, it does an `accept`; if anything else is ready, it means that a client has sent some data. We read it, display it, and send a little message back.

The code for a client is the same as in the earlier example, except that the message to the server includes the process ID:

```
static bool run_client(struct sockaddr_un *sap)
{
    if (fork() == 0) {
        int fd_skt;
        char buf[100];

        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_STREAM, 0) )
        while (connect(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) == -1)
            if (errno == ENOENT) {
                sleep(1);
                continue;
            }
            else
                EC_FAIL
        snprintf(buf, sizeof(buf), "Hello from %ld!",
          (long)getpid());
        ec_neg1( write(fd_skt, buf, strlen(buf) + 1 ) )
        ec_neg1( read(fd_skt, buf, sizeof(buf)) )
        printf("Client got \"%s\"\n", buf);
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    return true;

EC_CLEANUP_BGN
    /* only child gets here */
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Finally, there's a small `main` function that sets up the socket address, creates four client children, and runs the server in the parent:

2. A common mistake is to keep just one `fd_set` that gets passed to `select`. Since it clears the bits for unready file descriptors, they never get waited for.

```
#define SOCKETNAME "MySocket"

int main(void)
{
    struct sockaddr_un sa;
    int nclient;

    (void)unlink(SOCKETNAME);
    strcpy(sa.sun_path, SOCKETNAME);
    sa.sun_family = AF_UNIX;
    for (nclient = 1; nclient <= 4; nclient++)
        ec_false( run_client(&sa) )
    ec_false( run_server(&sa) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
        exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's the output we got:

```
Server got "Hello from 31786!"
Client got "Goodbye!"
Server got "Hello from 31785!"
Client got "Goodbye!"
Server got "Hello from 31784!"
Client got "Goodbye!"
Server got "Hello from 31787!"
Client got "Goodbye!"
```

We already have almost enough to implement a fairly sophisticated client-server system. What we need to know more about is how to set up socket addresses (`AF_UNIX` isn't that interesting) and how to set socket options. That's coming right up, after we deal with the byte-order issue.

8.1.4 Byte Order

There's never any problem passing strings like "Hello" and "Goodbye" between machines over a network because all communication preserves the order of bytes. However, there can be problems passing binary numbers because how the bytes in a number are arranged does indeed vary between machines.

To see this, look at Figure 8.2, which shows two ways of storing a two-byte integer whose hex representation is 0xD04C at location 106204 in memory.

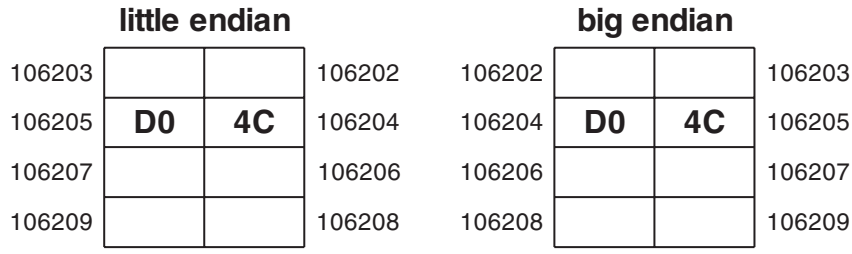

Figure 8.2 Little vs. Big Endian.

In a so-called "little-endian" machine, the address of a number is the address of its lowest-order byte; in a "big-endian" machine, it's the address of its highest-order byte.[3] Both are perfectly valid ways to address numbers, but the problem is that if one machine sends binary numbers to another machine that uses a different byte order, the numbers will be scrambled. That is, if the little-endian machine sends the number as bytes (the only way data is sent), they will go in the sequence 4C followed by D0, and if the receiving machine is big-endian, it will interpret the number as 4CD0 since it takes the first byte received as the high-order byte.

The solution is to send binary numbers in an agreed-upon network byte order. Thus, all senders must convert from host (local) byte order to network, and all receivers must convert from network byte order to host. But this is done only if the data isn't already structured in a standard way. For example, if you had a photograph encoded as a JPEG, you would send it as it stands—you would not attempt to convert any part of the JPEG to network byte order.

For most purposes you don't have to know either your host byte order or network byte order because there's a standard set of translation functions that do the conversion for 16-bit and 32-bit integers:

htons—convert 16-bit value from host to network byte order

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

uint16_t htons(
    uint16_t hostnum          /* 16-bit number in host byte order */
);
/* Returns number in network byte order (no error return) */
```

3. The terminology is self-explanatory, but the term "big-endian" actually comes from Jonathan Swift's *Gulliver's Travels*, which describes the controversy over whether to break eggs at the bigger or smaller end. In fact, "eleven thousand [Big-Endians] have, at several times, suffered Death, rather than submit to break their Eggs at the smaller End."

htonl—convert 32-bit value from host to network byte order

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

uint32_t htonl(
     uint32_t hostnum       /* 32-bit number in host byte order */
);
/* Returns number in network byte order (no error return) */
```

ntohs—convert 16-bit value from network to host byte order

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

uint16_t ntohs(
     uint16_t netnum        /* 16-bit number in network byte order */
);
/* Returns number in host byte order (no error return) */
```

ntohl—convert 32-bit value from network to host byte order

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

uint32_t ntohl(
     uint32_t netnum        /* 32-bit number in network byte order */
);
/* Returns number in host byte order (no error return) */
```

Unfortunately, the 16-bit functions use the suffix "s" for `short`, even though `short`s may have more than 16 bits, and, similarly, the 32-bit functions use the suffix "l" for `long`. In fact, these functions don't even operate on `short`s and `long`s; they operate on the types `uint16_t`, which is a 16-bit unsigned integer, and `uint32_t`, which is a 32-bit unsigned integer.

Just to show two of these functions in action, here's a program that converts 0xD04C to network byte order, displays the bytes, and converts it back:

```
int main(void)
{
    uint16_t nhost = 0xD04C, nnetwork;
    unsigned char *p;

    p = (unsigned char *)&nhost;
    printf("%x %x\n", *p, *(p + 1));
    nnetwork = htons(nhost);
    p = (unsigned char *)&nnetwork;
    printf("%x %x\n", *p, *(p + 1));
    exit(EXIT_SUCCESS);
}
```

The output shows that the Intel Pentium CPU on which it was run uses a byte order that's different from network:

```
4c d0
d0 4c
```

We can further conclude that the Pentium is little-endian, and network byte order is big-endian.[4]

Generally, you'll use the standard conversion functions when the protocol you're using requires you to send binary numbers, as we'll see in the next section. For your own purposes, try to design your applications to send characters instead of binary data, unless you're sending an object with a standardized format (JPEG, MP3, etc.). That's exactly the way HTTP (the Web protocol) does it (Section 8.4.2).

8.2 Socket Addresses

This whole section is about nothing but how to come up with a socket address (`struct sockaddr`) that you can use in calls to `bind` and `connect`. We already saw how to do this for domain `AF_UNIX`, where the address is mostly just some path name:

```
struct sockaddr_un sa;

strcpy(sa.sun_path, SOCKETNAME);
sa.sun_family = AF_UNIX;
```

But that's the only simple case. The other domains are more complicated.

8.2.1 Socket-Address Structures

Each address family has its own structure and header file. There's `sockaddr_un` for `AF_UNIX`, `sockaddr_in` for `AF_INET`, `sockaddr_x25` for `AF_X25`, and so on, although only the first two domains are standardized in [SUS2002]. One approach, which we've been using for `AF_UNIX`, is to declare a variable for the specific structure you need, fill its members somehow, and cast its address to a pointer-to-`struct sockaddr` (the generic type) when you call `bind` or `connect`.

4. It's big-endian because for years BSD systems, where the socket system calls originated, ran on big-endian VAX 11/780 computers, and network byte order was, in essence, "our byte order."

Speaking portably, the structure `sockaddr` only defines an abstract type; you can't use it to declare a structure to use because it may not have enough space. Instead, there's a standard structure named `sockaddr_storage` that is guaranteed to have enough space, but whose members aren't customized for any particular domain.

Sounds messy, but it's really not so bad:

- If you know exactly what domain you're using, declare a structure of the appropriate type (e.g., `sockaddr_un`) or allocate enough space for that structure dynamically (e.g., with `malloc`).
- If you need space enough for any domain's structure, use `sockaddr_storage` but cast a pointer to the appropriate type before using it.
- Use a cast to `struct sockaddr` in calls to `bind` and `connect`—according to their prototypes, you have no other choice.

Here's an example of the last two rules used together:

```
struct sockaddr_storage sas;
struct sockaddr_un *sa = (struct sockaddr_un *)&sas;
sa->sun_family = AF_UNIX;
...
ec_neg1( bind(fd, (struct sockaddr *)sa, sizeof(*sa)) )
```

Don't worry too much about the last rule. If you get it wrong, the compiler will tell you. Don't worry about using a `sockaddr_storage` directly either—it doesn't have any of the members you'll be wanting to use. But, since you're casting, there's no check at all at compile time that you're using the correct domain-specific structure for your domain. That you have to get right the first time.

8.2.2 `AF_UNIX` Socket Addresses

struct sockaddr_un—structure for AF_UNIX socket addresses

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

struct sockaddr_un {
    sa_family_t sun_family;    /* AF_UNIX */
    char sun_path[X];          /* socket pathname */
};
```

The actual space for the pathname, designated by *X* in the synopsis, varies from system to system. Don't assume it's more than 90 or so bytes. I already explained how to use this structure.

8.2.3 AF_INET Socket Addresses

As I mentioned earlier, the `AF_INET` domain is for communicating with sockets over the Internet. Here's the `sockaddr_in` structure:

struct sockaddr_in—structure for AF_INET socket addresses

```
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t sin_family;    /* AF_INET */
    in_port_t sin_port;        /* port number (uint16_t) */
    struct in_addr sin_addr;   /* IPv4 address */
};

struct in_addr {
    in_addr_t s_addr;          /* IPv4 address (uint32_t) */
};
```

An IPv4[5] address (usually called just an "IP address") is a 32-bit binary number assigned to a network interface on a machine. Instead of referring to these numbers the usual way (e.g., 1,182,625,240), we use dotted notation, in which each byte is expressed as a separate decimal number, as in 216.109.125.70. Even that isn't very convenient, so there's a system using descriptive names instead that allows us to say *www.yahoo.com* instead of 216.109.125.70 (Section 8.2.5), but I'll use dotted notation for now.

Each IP can be associated with lots of different services, each of which is assigned a 16-bit port number. Many of the port numbers are, by convention, assigned to well-known services. For example, HTTP (Web) servers are on port 80, FTP servers are on port 21, Telnet servers are on port 23, and so on. You can see how the ports are assigned on your machine by looking at the /etc/services file. For example, here are selected parts of the file (over 2000 lines in all) on my FreeBSD system:

```
...
ftp             21/tcp    #File Transfer [Control]
ftp             21/udp    #File Transfer [Control]
```

5. It's Version 4 of the Internet Protocol.

```
ssh             22/tcp    #Secure Shell Login
ssh             22/udp    #Secure Shell Login
telnet          23/tcp
telnet          23/udp
...
finger          79/tcp
finger          79/udp
http            80/tcp    www www-http         #World Wide Web HTTP
http            80/udp    www www-http         #World Wide Web HTTP
...
```

The types for the port and IP number, `in_port_t` and `in_addr_t`, are defined to be `uint16_t` and `uint32_t`, respectively, and must be in network byte order, which means you can use the `htons` and `htonl` functions, like this:

```
struct sockaddr_in sa;

sa.sin_family = AF_INET;
sa.sin_port = htons(80);
sa.sin_addr.s_addr =  htonl((216UL << 24) + (109UL << 16) +
  (125UL << 8) + 70UL); /* 216.109.125.70 */
```

But for dotted IP numbers, there's an even better function that goes directly from a string to a 32-bit IP number in network byte order:

inet_addr—convert dotted IPv4 address string to integer

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

in_addr_t inet_addr(
     const char *cp             /* dotted IP address */
);
/* Returns IP address or (in_addr_t)-1 on error (errno not defined) */
```

The return value, –1 cast to an unsigned 32-bit number, is the same as what we would get if we converted 255.255.255.255. But this is OK because that's an illegal IP address.

The reverse function is also sometimes handy:

inet_ntoa—convert integer IPv4 address to dotted string

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

char *inet_ntoa(
     struct in_addr in      /* integer address */
);
/* Returns string (no error return) */
```

Instead of `inet_addr` and `inet_ntoa`, you can use the more general functions `inet_ntop` and `inet_pton` (Section 8.9.5), which also work for IPv6 addresses.

Going back to our example, let's use `inet_addr` instead of `htonl` for the IP address and add some code that connects to the HTTP server on port 80, sends a request for a Web page, and displays part of what comes back:

```
#define REQUEST "GET / HTTP/1.0\r\n\r\n"

int main(void)
{
    struct sockaddr_in sa;
    int fd_skt;
    char buf[1000];
    ssize_t nread;

    sa.sin_family = AF_INET;
    sa.sin_port = htons(80);
    sa.sin_addr.s_addr =  inet_addr("216.109.125.70");
    ec_neg1( fd_skt = socket(AF_INET, SOCK_STREAM, 0) )
    ec_neg1( connect(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) )
    ec_neg1( write(fd_skt, REQUEST, strlen(REQUEST) ) )
    ec_neg1( nread = read(fd_skt, buf, sizeof(buf)) )
    (void)write(STDOUT_FILENO, buf, nread);
    ec_neg1( close(fd_skt) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The request is a GET command, which is defined by the HTTP protocol that defines Web communication (see Section 8.4.2). Here's part of the output that came back, which you may recognize as the HTML source for Yahoo's home page. It begins with a status line, followed immediately by the HTML (shown greatly abbreviated):

```
HTTP/1.1 200 OK
Date: Sat, 19 Jul 2003 18:51:56 GMT
P3P: policyref="http://p3p.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR CUR ADM ...
Cache-Control: private
Connection: close
Content-Type: text/html

<html><head>
```

```
<title>Yahoo!</title>
...
```

8.2.4 AF_INET6 Socket Addresses

If all 32 bits of an IPv4 address could be used, there would be addresses for 4.3 billion network interfaces, which is probably enough for a while. Because of the way IPv4 addresses are assigned, however, there can be only 534 million so-called Class B addresses (those that begin with 192 through 223), which are used by large organizations, and around 1992 it looked like these would run out by mid-1995. Another problem was that routing tables used by Internet backbone sites were growing too large to fit in the memory of routers available at that time.

The proposed solution was Version 6 of the Internet Protocol, called IPv6, which, among other improvements, expands the address from 4 bytes to 16, which allows 2^{128} addresses.[6]

The preferred way of writing an IPv6 address is as 8 groups of 2 bytes each, such as FEDC:BA98:7654:3210:FEDC:BA98:7654:3210.

You use a `sockaddr_in6` structure to set up an `AF_INET6` socket address; there's more to it than for `AF_INET`, as you would expect:

struct sockaddr_in6—structure for AF_INET6 socket addresses

```
#include <netinet/in.h>

struct sockaddr_in6 {
    sa_family_t sin6_family;    /* AF_INET6 */
    in_port_t sin6_port;        /* port number (uint16_t) */
    uint32_t sin6_flowinfo;     /* traffic class and flow information */
    struct in6_addr sin6_addr;  /* IPv4 address */
    uint32_t sin6_scope_id;     /* set of interfaces for a scope */
};

struct in6_addr {
    uint8_t s6_addr[16];        /* IPv6 address */
};
```

The two new members here that have no equivalent in a `sockaddr_in` structure are `sin6_flowinfo`, which is intended to hold a "flow label" and priority but whose use is currently unspecified, and `sin6_scope_id`, which identifies a set

6. This is more than the number of particles in the universe and over 665 billion trillion addresses *per square meter* of the earth's surface!

of interfaces that are used with certain addresses and is implementation dependent. If you're setting up your own `sockaddr_in6` structure, set these to zero. In fact, you're supposed to initialize the whole structure to zeros before setting the members you're interested in because some implementations have members in addition to those listed in the synopses above.

In practice, you won't be initializing a `sockaddr_in6` structure directly; instead, you'll get one from `getaddrinfo`, which is described in Section 8.2.6.

You can't use `inet_addr` and `inet_ntoa` with IPv6 addresses. Instead you use `inet_ntop` and `inet_pton`, which are in Section 8.9.5.

8.2.5 The Domain Name System (DNS)

Of course, you seldom type a numeric IP address into your Internet browser or FTP client—you type a mnemonic name like *www.yahoo.com* or *www.basepath.com*. These host names are translated to IP addresses by a world-wide distributed database called DNS. There are several standardized functions for accessing the DNS from a UNIX application. The newest and most powerful is called `getaddrinfo`, which is the one we'll use in our examples. Section 8.8.1 briefly describes some older functions (such as `gethostbyname`).

In a URL like *http://www.basepath.com/aup* (the URL for this book's site), the host name is just the middle part. That is, the overall syntax is:[7]

scheme`:`*//hostname/path*

where *scheme* specifies the method of access to the resource (e.g., HTTP, FTP), *hostname* is the name of the host as recorded in the DNS database, and *path* is a path within the host's file system.

In our examples, we use the socket system calls to connect to the host. Once we have a connection, we interact with the host according to the scheme. For instance, in the example in Section 8.2.3, the scheme was HTTP, so we sent the request

```
GET / HTTP/1.0
```

7. This is a simplified view, but it will suffice for our purposes. The complete syntax for Uniform Resource Identifiers is more complicated.

to the server, which interpreted it as a request for a Web page, which it sent back to us. How the path is treated depends on the scheme; for HTTP, it's the argument to the GET command (a single slash in our example). None of the socket system calls care about the path or the scheme.

We don't need to get into HTTP, HTML, FTP, or any other scheme-related subjects in this book beyond the minimum that we need to understand some simple examples. The best way to learn about the various schemes is to read the RFC (Request for Comments) documents [RFC].

8.2.6 `getaddrinfo`

Rather than construct a `sockaddr_in` or `sockaddr_in6` address from scratch, it's easier to call `getaddrinfo`, which builds a socket address for you:

`getaddrinfo`—get socket-address information

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

int getaddrinfo(
     const char *nodename,         /* node name */
     const char *servname,         /* service name */
     const struct addrinfo *hint, /* hint */
     struct addrinfo **infop       /* returned info as linked list */
);
/* Returns 0 on success or error number on error (errno not set) */
```

`struct addrinfo`—structure for getaddrinfo

```
struct addrinfo {
     int ai_flags;                /* input flags */
     int ai_family;               /* address family */
     int ai_socktype;             /* socket type */
     int ai_protocol;             /* protocol */
     socklen_t ai_addrlen;        /* length of socket address */
     struct sockaddr *ai_addr;    /* socket address */
     char *ai_canonname;          /* canonical name of service location */
     struct addrinfo *ai_next;    /* pointer to next structure in list */
};
```

Typically, you call `getaddrinfo` with `nodename` set to the host name you want, `servname` set to the port number (as a string), and `hint` set to the address family and socket type you want. You get back, through the `infop` argument, a linked list of `addrinfo` structures that contain socket addresses and other information that match your hint. You pick a suitable socket address and use it directly in a call to `connect` or `bind`, casting the `ai_addr` member to `struct sockaddr *`.

The "host name" can be a name to be looked up using a DNS server, a name defined in the local machine's /etc/hosts file, an IPv4 address in dotted notation (as a string), or an IPv6 address in colon notation (as a string).

The one critical flag is `AI_PASSIVE`, which means that the socket addresses returned are for use with `accept`; that is, the call to `getaddrinfo` is from a server. Otherwise, with the flag clear, the call is from a client and the socket address will be used with `connect`. For a connectionless protocol (e.g., `SOCK_DGRAM`), it can also be used with `sendto` or `sendmsg`.

`getaddrinfo` returns its own kind of error code that you can't treat as an `errno` value, which means that you can't use standard functions like `perror` or `strerror` in case of an error. Instead, there's a special function just for the error code returned by `getaddrinfo` and another function, `getnameinfo` (Section 8.8.1):

gai_strerror—get error-code description

```
#include <netdb.h>

const char *gai_strerror(
    int code              /* error code */
);
/* Returns string (no error return) */
```

In our examples, we'll use the error-checking macro `ec_ai`, which knows how to deal with the return from `getaddrinfo` and `getnameinfo`, so we won't call `gai_strerror` directly.

Here's a simple example using `getaddrinfo`:

```
int main(void)
{
    struct addrinfo *infop = NULL, hint;

    memset(&hint, 0, sizeof(hint));
    hint.ai_family = AF_INET;
    hint.ai_socktype = SOCK_STREAM;
    ec_ai( getaddrinfo("www.yahoo.com", "80", &hint, &infop) )
    for ( ; infop != NULL; infop = infop->ai_next) {
        struct sockaddr_in *sa = (struct sockaddr_in *)infop->ai_addr;

        printf("%s port: %d protocol: %d\n", inet_ntoa(sa->sin_addr),
          ntohs(sa->sin_port), infop->ai_protocol);
    }
    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The output:

```
66.218.70.48 port: 80 protocol: 0
66.218.70.49 port: 80 protocol: 0
66.218.71.88 port: 80 protocol: 0
66.218.71.86 port: 80 protocol: 0
66.218.70.50 port: 80 protocol: 0
66.218.71.91 port: 80 protocol: 0
66.218.71.80 port: 80 protocol: 0
66.218.71.84 port: 80 protocol: 0
66.218.71.90 port: 80 protocol: 0
66.218.71.93 port: 80 protocol: 0
66.218.71.94 port: 80 protocol: 0
66.218.71.92 port: 80 protocol: 0
66.218.71.89 port: 80 protocol: 0
```

So it seems that *www.yahoo.com* is associated with several different IP numbers. Because of the hint, we know that all of them are `AF_INET` addresses of type `SOCK_STREAM`, so any of them are suitable for access to Web pages. To see this, let's pair up the retrieval of the socket address via `getaddrinfo` with the connection and display code from the example in Section 8.2.3:

```
#define REQUEST "GET / HTTP/1.0\r\n\r\n"

int main(void)
{
    struct addrinfo *infop = NULL, hint;
    int fd_skt;
    char buf[1000];
    ssize_t nread;

    memset(&hint, 0, sizeof(hint));
    hint.ai_family = AF_INET;
    hint.ai_socktype = SOCK_STREAM;
    ec_ai( getaddrinfo("www.yahoo.com", "80", &hint, &infop) )
    ec_neg1( fd_skt = socket(infop->ai_family, infop->ai_socktype,
      infop->ai_protocol) )
    ec_neg1( connect(fd_skt, (struct sockaddr *)infop->ai_addr,
      infop->ai_addrlen) )
    ec_neg1( write(fd_skt, REQUEST, strlen(REQUEST) ) )
    ec_neg1( nread = read(fd_skt, buf, sizeof(buf)) )
    (void)write(STDOUT_FILENO, buf, nread);
    ec_neg1( close(fd_skt) )
    exit(EXIT_SUCCESS);
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Running this produced the same output that we showed before in Section 8.2.3.

If your system, your DNS server, and the host you're accessing support it, you can use `getaddrinfo` to access more than `SOCK_STREAM AF_INET` addresses on port 80. You can use `NULL` for either the `nodename` or `servname` arguments (but not both) and/or use an address "family" of `AF_UNSPEC` in the hint to find out what's available. Then you search through the returned addresses to find what you want to work with. For example, you might want to use an `AF_INET6` (IPv6) address if one is there. For the details on this advanced use of `getaddrinfo`, see [SUS2002], your system's documentation, or Chapter 11 of [Ste2003].

One more thing before we leave this section: You should free the list returned by `getaddrinfo` when you no longer need it with a call to `freeaddrinfo`:

freeaddrinfo—free socket-address information

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

void freeaddrinfo(
    struct addrinfo *infop    /* list to free */
);
```

We didn't call `freeaddrinfo` in our examples because they were `main` functions that were going to exit anyway.

8.2.7 `gethostname`

Sometimes, such as in the example Web server in Section 8.4.4, a program needs the name of its own host. Usually the generic name "localhost" will work, but if other machines need to connect to the name, what's needed is the public name assigned by the system administrator. That's what `gethostname` is for:

gethostname—get name of host

```
#include <unistd.h>

int gethostname(
    char *name,           /* returned name */
    size_t namelen        /* size of name buffer */
);
/* Returns 0 on success or -1 on error (errno not defined) */
```

See Section 8.4.4 for an example of `gethostname` in use.

8.3 Socket Options

One reason why the `socket` system call is so simple, even though sockets are complicated, is that all of the options are handled by a separate system call:

setsockopt—set socket options

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

int setsockopt(
    int socket_fd,          /* socket file descriptor */
    int level,              /* level to be accessed */
    int option,             /* option to set */
    const void *value,      /* value to set */
    socklen_t value_len     /* length of value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

Four of the five arguments are pretty easy to explain: `socket_fd` is the socket file descriptor, `option` is the name of the option, `value` points to the value to set it to, and `value_len` is the length of the value that `value` points to. The value is usually an integer, but sometimes it's a structure specific to the option, as we'll see.

The second argument, `level`, identifies the protocol level to which the option belongs. One level, `SOL_SOCKET`, applies to the socket level itself, and it's the one you'll probably use the most. [SUS2002] defines six additional protocol levels in <netinet/in.h>:

`IPPROTO_IP`	Internet protocol
`IPPROTO_IPV6`	Internet Protocol Version 6
`IPPROTO_ICMP`	Control message protocol
`IPPROTO_RAW`	Raw IP Packets Protocol
`IPPROTO_TCP`	Transmission control protocol
`IPPROTO_UDP`	User datagram protocol

[SUS2002] defines some options for these six protocols, but implementations typically define additional ones. Unfortunately, it's not easy to get a list of all the options—you'll have to read through the documentation for the protocol you're using. For example, on Solaris, typing `man ip` brings up a manual page that describes options for `IPPROTO_IP`.

We'll just describe the standardized `SOL_SOCKET` options here; for the rest, consult your system's documentation or [Ste2003].

There's a matching call to get options, with almost the same arguments:

getsockopt—get socket options

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

int getsockopt(
     int socket_fd,         /* socket file descriptor */
     int level,             /* level to be accessed */
     int option,            /* option to get */
     void *value,           /* returned value */
     socklen_t *value_len   /* length of value */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The one thing different between `setsockopt` and `getsockopt` is that with the latter the `value_len` argument is a pointer to the length, and it must be set before the call to the size of the buffer pointed to by the `value` argument. On return it's set to the size of the returned value.

Here's a quick example that sets a socket's `SO_REUSEADDR` option (I'll get to what that actually means below):

```
int socket_option_value = 1;

ec_neg1( setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
  &socket_option_value, sizeof(socket_option_value)) )
```

I'll briefly describe all of the portable `SOL_SOCKET` options. Most of the option values are integers, in which case `value` must point to an `int`, and `value_len` will be `sizeof(int)`. Some integers are Boolean values, for which 1 is true and 0 is false. Don't use the C constant `true` in place of a 1 because `true` may have the value –1, which is fine for C but not acceptable to `setsockopt`. In the following list, the notation (B) indicates a Boolean value, (I) indicates an integer, and (X) indicates a type that's explained further. The letter is followed by an "s" and/or a "g" to indicate whether it's used with `setsockopt`, `getsockopt`, or both.

Be aware that whether some options actually do anything (e.g., SO_KEEPALIVE) depends on the actual protocol in use and its implementation.

SO_ACCEPTCONN Socket is accepting connections; that is, listen has been called. (Bg)

SO_BROADCAST Sending of broadcast messages is allowed if the protocol supports it. (Bsg)

SO_DEBUG Debugging information is recorded by the underlying protocol implementation. (Bsg)

SO_DONTROUTE Sent messages bypass the standard routing facilities and go directly to the network interface according to the destination address. (Bsg)

SO_ERROR Socket error status, which is cleared after it's retrieved. (Ig)

SO_KEEPALIVE Keep the connection active by periodically sending messages. If there's no response, the socket is disconnected. (Writing to a disconnected socket, like writing to a pipe with no reader, generates a SIGPIPE signal.) (Bsg)

SO_LINGER If on, causes a close to block if there are any unsent messages until they're sent or the linger-time expires. (Xsg) The value is a linger structure:

```
struct linger {
    int l_onoff;    /* on (1) or off (0) */
    int l_linger;   /* linger-time in seconds */
};
```

SO_OOBINLINE Received out-of-band data stays inline (see Section 8.7). (Bsg)

SO_RCVBUF Receive buffer size. (Isg)

SO_RCVLOWAT Receive low-water-mark. Blocking receive operations (e.g., read) block until the lesser of this amount and the requested amount are received. The default is 1. (Isg)

SO_RCVTIMEO Uses a timeval structure (Section 1.7.1) for the maximum time to wait for a blocking receive operation to complete. Zero time (the default) means infinite. If the time expires, the operation returns with a partial count or with –1 and errno set to EAGAIN or EWOULDBLOCK. (Xsg)

`SO_REUSEADDR` `bind` allows reuse of local addresses. Otherwise, `bind` returns –1 with `errno` set to `EADDRINUSE` if a previous `bind` has occurred within a system-defined period (for example, a few minutes). Very convenient for debugging and testing. (Bsg)

`SO_SNDBUF` Send buffer size. This option takes an int value. (Isg)

`SO_SNDLOWAT` Send low-water-mark. Nonblocking send operations (e.g., `write`) send no data unless the lesser of this amount and the requested amount can be sent immediately. (Isg)

`SO_SNDTIMEO` Uses a `timeval` structure (Section 1.7.1) for the maximum time to wait for a blocking send operation to complete. Zero time (the default) means infinite. If the time expires, the operation returns with a partial count or with –1 and `errno` set to `EAGAIN` or `EWOULDBLOCK`. (Xsg)

`SO_TYPE` Socket type (e.g., `SOCK_STREAM`). (Isg)

To show how these options are used with `getsockopt`, here's a program that displays the values for several different kinds of sockets:

```
typedef enum {OT_INT, OT_LINGER, OT_TIMEVAL} OPT_TYPE;

static void show(int skt, int level, int option, const char *name,
  OPT_TYPE type)
{
    socklen_t len;
    int n;
    struct linger lng;
    struct timeval tv;

    switch (type) {
    case OT_INT:
        len = sizeof(n);
        if (getsockopt(skt, level, option, &n, &len) == -1)
            printf("%s FAILED (%s)\n", name, strerror(errno));
        else
            printf("%s = %d\n", name, n);
        break;
    case OT_LINGER:
        len = sizeof(lng);
        if (getsockopt(skt, level, option, &lng, &len) == -1)
            printf("%s FAILED (%s)\n", name, strerror(errno));
```

```
        else
            printf("%s = l_onoff: %d; l_linger: %d secs.\n", name,
              lng.l_onoff, lng.l_linger);
        break;
    case OT_TIMEVAL:
        len = sizeof(tv);
        if (getsockopt(skt, level, option, &tv, &len) == -1)
            printf("%s FAILED (%s)\n", name, strerror(errno));
        else
            printf("%s = %ld secs.; %ld usecs.\n", name,
              (long)tv.tv_sec, (long)tv.tv_usec);
    }
}

static void showall(int skt, const char *caption)
{
    printf("\n%s\n", caption);
    show(skt, SOL_SOCKET, SO_ACCEPTCONN, "SO_ACCEPTCONN", OT_INT);
    show(skt, SOL_SOCKET, SO_BROADCAST, "SO_BROADCAST", OT_INT);
    show(skt, SOL_SOCKET, SO_DEBUG, "SO_DEBUG", OT_INT);
    show(skt, SOL_SOCKET, SO_DONTROUTE, "SO_DONTROUTE", OT_INT);
    show(skt, SOL_SOCKET, SO_ERROR, "SO_ERROR", OT_INT);
    show(skt, SOL_SOCKET, SO_KEEPALIVE, "SO_KEEPALIVE", OT_INT);
    show(skt, SOL_SOCKET, SO_LINGER, "SO_LINGER", OT_LINGER);
    show(skt, SOL_SOCKET, SO_OOBINLINE, "SO_OOBINLINE", OT_INT);
    show(skt, SOL_SOCKET, SO_RCVBUF, "SO_RCVBUF", OT_INT);
    show(skt, SOL_SOCKET, SO_RCVLOWAT, "SO_RCVLOWAT", OT_INT);
    show(skt, SOL_SOCKET, SO_RCVTIMEO, "SO_RCVTIMEO", OT_TIMEVAL);
    show(skt, SOL_SOCKET, SO_REUSEADDR, "SO_REUSEADDR", OT_INT);
    show(skt, SOL_SOCKET, SO_SNDBUF, "SO_SNDBUF", OT_INT);
    show(skt, SOL_SOCKET, SO_SNDLOWAT, "SO_SNDLOWAT", OT_INT);
    show(skt, SOL_SOCKET, SO_SNDTIMEO, "SO_SNDTIMEO", OT_TIMEVAL);
    show(skt, SOL_SOCKET, SO_TYPE, "SO_TYPE", OT_INT);
}

int main(void)
{
    int skt;

    ec_neg1( skt = socket(AF_INET, SOCK_STREAM, 0) )
    showall(skt, "AF_INET SOCK_STREAM");
    ec_neg1( close(skt) )
    ec_neg1( skt = socket(AF_INET, SOCK_DGRAM, 0) )
    showall(skt, "AF_INET SOCK_DGRAM");
    ec_neg1( close(skt) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The output on Solaris was:

```
AF_INET SOCK_STREAM
SO_ACCEPTCONN = 0
SO_BROADCAST = 0
SO_DEBUG = 0
SO_DONTROUTE = 0
SO_ERROR = 0
SO_KEEPALIVE = 0
SO_LINGER = l_onoff: 0; l_linger: 0 secs.
SO_OOBINLINE = 0
SO_RCVBUF = 65536
SO_RCVLOWAT FAILED (Option not supported by protocol)
SO_RCVTIMEO FAILED (Option not supported by protocol)
SO_REUSEADDR = 0
SO_SNDBUF = 65536
SO_SNDLOWAT FAILED (Option not supported by protocol)
SO_SNDTIMEO FAILED (Option not supported by protocol)
SO_TYPE = 2

AF_INET SOCK_DGRAM
SO_ACCEPTCONN = 0
SO_BROADCAST = 0
SO_DEBUG = 0
SO_DONTROUTE = 0
SO_ERROR = 0
SO_KEEPALIVE = 0
SO_LINGER = l_onoff: 0; l_linger: 0 secs.
SO_OOBINLINE = 0
SO_RCVBUF = 65536
SO_RCVLOWAT FAILED (Option not supported by protocol)
SO_RCVTIMEO FAILED (Option not supported by protocol)
SO_REUSEADDR = 0
SO_SNDBUF = 65536
SO_SNDLOWAT FAILED (Option not supported by protocol)
SO_SNDTIMEO FAILED (Option not supported by protocol)
SO_TYPE = 1
```

8.4 Simple Socket Interface (SSI)

Instead of calling `getaddrinfo`, `socket`, `bind`, or `connect` and the other related functions every time we want to use sockets, let's code some higher-level functions to hide some of the tedious details. We'll call the interface to these functions SSI, for Simple Socket Interface.

8.4.1 SSI Function Calls

Internally, the SSI functions keep the state of an open connection in a structure of type `SSI`, which we'll look into shortly. You get a pointer to an `SSI` when you call `ssi_open`, and you close the SSI when you're done with `ssi_close`:

ssi_open—open SSI connection

```
SSI *ssi_open(
    const char *name,      /* server name */
    bool server            /* called from server? */
);
/* Returns pointer to SSI or NULL on error (sets errno) */
```

ssi_close—close SSI connection

```
bool ssi_close(
    SSI *ssip              /* pointer to SSI */
);
/* Returns true on success or false on error (sets errno) */
```

As we'll see, `ssi_open` encapsulates the building of the socket address (e.g., with a call to `getaddrinfo`), and the calls to `socket`, `bind`, `listen`, and `connect`.

The name passed to `ssi_open` is taken as an `AF_INET` host name if it begins with two slashes (which aren't part of the name). The name must be followed by a colon and a port number. If it doesn't begin with two slashes, it's a local `AF_UNIX` name like those in Section 8.1.1. Some examples:

`//www.basepath.com:80`	`AF_INET` connection to port 80 on host *www.basepath.com*
`//firecracker:31000`	`AF_INET` connection to port 31000 on firecracker
`//216.109.125.43:21`	`AF_INET` connection to port 21 on 216.109.125.43
`MyServer`	`AF_UNIX` connection

A server calls `ssi_wait_server` to wait for a client's file descriptor to be ready; it encapsulates the calls to `select` and `accept` that we saw in the example in Section 8.1.3:

ssi_wait_server—wait for ready file descriptor

```
int ssi_wait_server(
    SSI *ssip          /* pointer to SSI */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

A client calls `ssi_get_server_fd` to get the file descriptor for its connection to the server:

ssi_get_server_fd—get server's file descriptor

```
int ssi_get_server_fd(
    SSI *ssip          /* pointer to SSI */
);
/* Returns file descriptor or -1 on error (sets errno) */
```

Finally, a server calls `ssi_close_fd` when it gets an EOF from a client's file descriptor or otherwise knows it will no longer be needed:

ssi_close_fd—close client file descriptor

```
bool ssi_close_fd(
    SSI *ssip,         /* pointer to SSI */
    int fd             /* file descriptor */
);
/* Returns true on success or false on error (sets errno) */
```

These functions are all we need to implement simple servers and clients using connected (`SOCK_STREAM`) clients over `AF_UNIX` or `AF_INET`. We'll show how they're used by showing a simple Web browser and a simple Web server. Then we'll show the SSI implementation.

8.4.2 A Brief Introduction to HTTP

HTTP, the Hypertext Transfer Protocol, is defined by the Internet Engineering Task Force (IETF) in a document called RFC 2616. You can read the whole RFC, and many others, at their Web site [RFC]. What follows is a highly simplified explanation of HTTP—just enough to understand the examples in this section.

For HTTP, the client is normally a Web browser, and the server is a Web server. Once a connection has been made, the client initiates an exchange by sending a string to the server of the form

`GET` *path* `HTTP/`*version*`\r\n\r\n`

where *path* is the desired document (e.g., /index.html) and *version* is the desired version of HTTP to be used (e.g., 1.0).

The server determines whether the document exists and if the client can have access to it. If not, it responds with a status line like

```
HTTP/1.1 404 Not Found\r\n
```

which is followed by an HTML document that explains the error. (We'll get to how documents are sent in a moment.)

If the file can be sent, the status line is like

```
HTTP/1.1 200 OK\r\n
```

Each document, whether HTML text, a JPEG, or whatever, is preceded by a header that describes it, which has this general form (in our examples, anyway):

```
Server: servername\r\n
Content-Length: length\r\n
Content-Type: type\r\n\r\n
```

The *servername* can be anything; we'll use "AUP-ws" for our servers. The *length* is the length of the document in bytes so the client knows how much to read. That way the connection can be left open, since the client doesn't need an EOF to tell it to stop reading. The *type* is a so-called MIME (Multipurpose Internet Mail Extensions) type, only two of which we care about: "text/html" and "image/jpeg." (There are dozens more.) After the header comes the document, exactly as it exists on the server's file system.

8.4.3 SSI Web Browser

Here's a simple Web browser called `minibr`. As with the earlier examples, it can retrieve HTML; however, it doesn't know how to interpret the tags for a nice screen display, so it just dumps everything to the standard output:

```
int main(void)
{
    char url[100], s[500], *path = "", *p;
    SSI *ssip;
    int fd;
    ssize_t nread;

    while (true) {
        printf("URL: ");
        if (fgets(url, sizeof(url), stdin) == NULL)
            break;
        if ((p = strrchr(url, '\n')) != NULL)
            *p = '\0';
```

```
        if ((p = strchr(url, '/')) != NULL) {
            path = p + 1;
            *p = '\0';
        }
        snprintf(s, sizeof(s), "//%s:80", url);
        ec_null( ssip = ssi_open(s, false) )
        ec_neg1( fd = ssi_get_server_fd(ssip) )
        snprintf(s, sizeof(s), "GET /%s HTTP/1.0\r\n\r\n", path);
        ec_neg1( writeall(fd, s, strlen(s)) )
        while (true) {
            switch (nread = read(fd, s, sizeof(s))) {
            case 0:
                printf("EOF\n");
                break;
            case -1:
                EC_FAIL
            default:
                ec_neg1( writeall(STDOUT_FILENO, s, nread) )
                continue;
            }
            break;
        }
        ec_false( ssi_close(ssip) )
    }
    ec_false( !ferror(stdin) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

The code just after the call to `fgets` splits the entered URL into its host name and path (explained in Section 8.2.5). The host name, prepended with slashes and appended with port 80, is passed to `ssi_open`. The path is formatted into the HTTP GET request that's sent to the Web server after the connection is made.

Here's a sample `minibr` session with each retrieved page greatly abbreviated:

```
URL: www.basepath.com
HTTP/1.1 200 OK
Date: Sat, 19 Jul 2003 19:01:41 GMT
Server: Apache/1.3.27 (Unix) FrontPage/5.0.2.2510 mod_jk/1.1.0
Last-Modified: Thu, 15 May 2003 19:56:49 GMT
ETag: "61744-191-3ec3f101"
Accept-Ranges: bytes
Content-Length: 401
Connection: close
Content-Type: text/html
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
...
URL: 216.109.125.70
HTTP/1.1 200 OK
Date: Sat, 19 Jul 2003 19:02:49 GMT
P3P: policyref="http://p3p.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR CUR ADM
...
Cache-Control: private
Connection: close
Content-Type: text/html

<html><head>

<title>Yahoo!</title>
...
```

8.4.4 SSI Web Server

So far we've been showing clients—now it's time for a server. Our simple Web server looks for a GET request from a client, extracts the path from it, and writes a response followed by the requested file to the client's file descriptor, as described in Section 8.4.2.

It starts with some macros for the HTTP responses that it can provide and some HTML for "not found" errors:

```
#define HEADER\
    "HTTP/1.0 %s\r\n"\
    "Server: AUP-ws\r\n"\
    "Content-Length: %ld\r\n"

#define CONTENT_TEXT\
    "Content-Type: text/html\r\n\r\n"

#define CONTENT_JPEG\
    "Content-Type: image/jpeg\r\n\r\n"

#define HTML_NOTFOUND\
    "<!DOCTYPE html PUBLIC \"-//IETF//DTD HTML 2.0//EN\">\n"\
    "<html><head><title>Error 404</title>\n"\
    "</head><body>\n"\
    "<h2>AUP-ws server can't find document</h2>"\
    "</body></html>\r\n"
```

Note that `HEADER` contains formatting codes that are replaced by the status code and content length when it's actually used by the function `send_header`:

```
static void send_header(SSI *ssip, const char *msg, off_t len,
  const char *path, int fd)
{
    char buf[1000], *dot;

    snprintf(buf, sizeof(buf), HEADER, msg, (long)len);
    ec_neg1( writeall(fd, buf, strlen(buf)) )
    dot = strrchr(path, '.');
    if (dot != NULL && (strcasecmp(dot, ".jpg") == 0 ||
      strcasecmp(dot, ".jpeg") == 0))
        ec_neg1( writeall(fd, CONTENT_JPEG, strlen(CONTENT_JPEG)) )
    else
        ec_neg1( writeall(fd, CONTENT_TEXT, strlen(CONTENT_TEXT)) )
    return;

EC_CLEANUP_BGN
    EC_FLUSH("Error sending response (nonfatal)")
EC_CLEANUP_END
}
```

This function is called with a `msg` of either "404 Not Found" or "200 OK," as I explained in Section 8.4.2. The `len` argument is the length of the document, and `path` is its file name, used only to determine whether the file is a JPEG or HTML. The caller of `send_header` is responsible for sending the file.

`send_header` is called by the function `handle_request` that handles GET requests:

```
#define DEFAULT_DOC "index.html"
#define WEB_ROOT "/aup/webroot/"

static bool handle_request(SSI *ssip, char *s, int fd)
{
    char *tkn, buf[1000], path[500];
    int ntkn;
    FILE *in;
    struct stat statbuf;
    ssize_t nread;

    for (ntkn = 1; (tkn = strtok(s, " ")) != NULL; ntkn++) {
        s = NULL;
        switch (ntkn) {
        case 1:
            if (strcasecmp(tkn, "get") != 0) {
                printf("Unknown request\n");
```

```
                return false;
            }
            continue;
        case 2:
            break;
        }
        break;
    }
    snprintf(path, sizeof(path) - 1 - strlen(DEFAULT_DOC),
      "%s%s", WEB_ROOT, tkn);
    if (stat(path, &statbuf) == 0 && S_ISDIR(statbuf.st_mode)) {
        if (path[strlen(path) - 1] != '/')
            strcat(path, "/");
        strcat(path, DEFAULT_DOC);
    }
    if (stat(path, &statbuf) == -1 ||
      (in = fopen(path, "rb")) == NULL) {
          send_header(ssip, "404 Not Found", strlen(HTML_NOTFOUND), "",
            fd);
        ec_neg1( writeall(fd, HTML_NOTFOUND, strlen(HTML_NOTFOUND)) )
        return false;
    }
    send_header(ssip, "200 OK", statbuf.st_size, path, fd);
    while ((nread = fread(buf, 1, sizeof(buf), in)) > 0)
        ec_neg1( writeall(fd, buf, nread) )
    ec_eof( fclose(in) )
    return true;

EC_CLEANUP_BGN
    EC_FLUSH("Error sending response (nonfatal)")
    return false;
EC_CLEANUP_END
}
```

The `for` loop is just a way to check that it's indeed a GET request and to strip out the path name, which is then taken as a subdirectory of `WEB_ROOT`, to ensure that not just any path on the local system will be served up. If it's a directory, the default HTML file index.html is appended; otherwise, it's assumed to name an HTML or JPEG file.

If the path doesn't exist, the 404 header is sent, followed by some HTML. Otherwise, the 200 header is sent followed by the file. Note that it might be text or a binary JPEG.

Finally, here's the `main` program that contains calls to SSI functions to handle the actual work of connecting to clients:

```
#define PORT ":8000"

int main(void)
{
    SSI *ssip = NULL;
    char msg[1600];
    ssize_t nrcv;
    int fd;
    char host[100] = "//";

    ec_neg1( gethostname(&host[2], sizeof(host) - 2 - strlen(PORT)) )
    strcat(host, PORT);
    printf("Connecting to host \"%s\"\n", host);
    ec_null( ssip = ssi_open(host, true) )
    printf("\t...connected\n");
    while (true) {
        ec_neg1( fd = ssi_wait_server(ssip) )
        switch (nrcv = read(fd, msg, sizeof(msg) - 1)) {
        case -1:
            printf("Read error (nonfatal)\n");
            /* fall through */
        case 0:
            ec_false( ssi_close_fd(ssip, fd) )
            continue;
        default:
            msg[nrcv] = '\0';
            (void)handle_request(ssip, msg, fd);
        }
    }
    ec_false( ssi_close(ssip) )
    printf("Done.\n");
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    return EXIT_FAILURE;
EC_CLEANUP_END
}
```

The server name for `ssi_open` is formed like this:

//host`:8000`

We use port 8000 because port 80 was already in use by a real Web server (Apache) on our system and because ports under 1024 are restricted to the superuser.

The function connects and then goes into a loop in which it waits for a ready file descriptor, reads a command, and gives it to `handle_request`. It can get an EOF if a client terminates or simply closes the connection, which some clients do from

time to time. In that case it has to call `ssi_close_fd` to tell `ssi_wait_server` not to consider that file descriptor any more, as I explained in Section 8.1.3.

This Web server really works, and you can access it from any browser, not just the simple one in the previous section. For example, Figure 8.3 shows it being accessed via the URL *http://suse2:8000:*

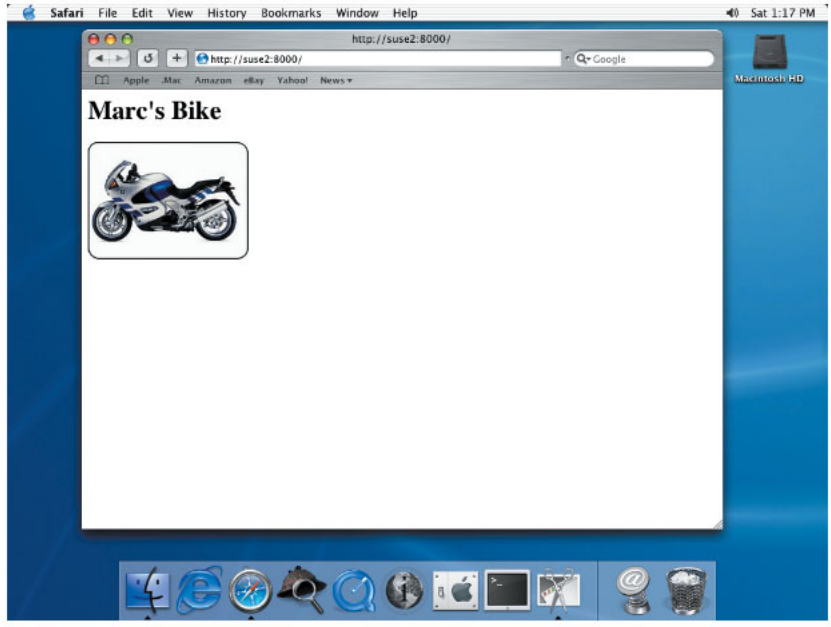

Figure 8.3 SSI Web server accessed via Safari browser on a Macintosh.

The HTML in /aup/webroot/index.html on the server (called suse2 because it runs SuSE Linux) was:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
<head>
</head>
<body>
<h1>Marc's Bike</h1>
<p><img src="picture.jpg">
</body>
</html>
```

8.4.5 SSI Implementation

I've discussed all the issues involved in implementing the SSI functions and even shown all the code in one example or another. You may want to review Sections 8.1.3, 8.2.2, and 8.2.6.

This is the structure for the type `SSI`:

```
#define SSI_NAME_SIZE 200

typedef struct {
    bool ssi_server;                    /* server? (vs. client) */
    int ssi_domain;                     /* AF_INET or AF_UNIX */
    int ssi_fd;                         /* socket fd */
    char ssi_name_server[SSI_NAME_SIZE]; /* server name */
    fd_set ssi_fd_set;                  /* set for server's select */
    int ssi_fd_hwm;                     /* high-water-mark for fds seen */
} SSI;
```

We'll see how the members are used as we go though the code.

First comes a handy function that can make socket addresses for `AF_UNIX` or `AF_INET`. The name argument is of the form host:port for `AF_INET`, and just a plain name for `AF_UNIX`. The `will_bind` argument indicates whether the socket address is for use with `bind` or `connect`, as explained in Section 8.2.6.

```
bool make_sockaddr(struct sockaddr *sa, socklen_t *len, const char *name,
  int domain, bool will_bind)
{
    struct addrinfo *infop = NULL;

    if (domain == AF_UNIX) {
        struct sockaddr_un *sunp = (struct sockaddr_un *)sa;

        if (strlen(name) >= sizeof(sunp->sun_path)) {
            errno = ENAMETOOLONG;
            EC_FAIL
        }
        strcpy(sunp->sun_path, name);
        sunp->sun_family = AF_UNIX;
        *len = sizeof(*sunp);
    }
    else {
        struct addrinfo hint;
        char nodename[SSI_NAME_SIZE], *servicename;

        memset(&hint, 0, sizeof(hint));
        hint.ai_family = domain;
```

```
        hint.ai_socktype = SOCK_STREAM;
        if (will_bind)
            hint.ai_flags = AI_PASSIVE;
        strcpy(nodename, name);
        servicename = strchr(nodename, ':');
        if (servicename == NULL) {
            errno = EINVAL;
            EC_FAIL
        }
        *servicename = '\0';
        servicename++;
        ec_ai( getaddrinfo(nodename, servicename, &hint, &infop) )
        memcpy(sa, infop->ai_addr, infop->ai_addrlen);
        *len = infop->ai_addrlen;
        freeaddrinfo(infop);
    }
    return true;

EC_CLEANUP_BGN
    if (infop != NULL)
        freeaddrinfo(infop);
    return false;
EC_CLEANUP_END
}
```

I deliberately kept `make_sockaddr` independent of the `SSI` structure in case you want to pull it out and use it in your own applications.

Next come two functions used by `ssi_open` and `ssi_close` to maintain the high-water-mark of file descriptors, just as we did in Section 8.1.3:

```
static void set_fd_hwm(SSI *ssip, int fd)
{
    if (fd > ssip->ssi_fd_hwm)
        ssip->ssi_fd_hwm = fd;
}

static void reset_fd_hwm(SSI *ssip, int fd)
{
    if (fd == ssip->ssi_fd_hwm)
        ssip->ssi_fd_hwm--;
}
```

Now we're ready for `ssi_open`, which doesn't do anything we haven't already seen:

```
SSI *ssi_open(const char *name_server, bool server)
{
    SSI *ssip = NULL;
```

```
    struct sockaddr_storage sa;
    socklen_t sa_len;

    ec_null(  ssip = calloc(1, sizeof(SSI)) )
    ssip->ssi_server = server;
    if (strncmp(name_server, "//", 2) == 0) {
        ssip->ssi_domain = AF_INET;
        name_server += 2;
    }
    else {
        ssip->ssi_domain = AF_UNIX;
        if (ssip->ssi_server)
            (void)unlink(name_server);
    }
    if (strlen(name_server) >= sizeof(ssip->ssi_name_server)) {
        errno = ENAMETOOLONG;
        EC_FAIL
    }
    strcpy(ssip->ssi_name_server, name_server);
    ec_false( make_sockaddr((struct sockaddr *)&sa, &sa_len,
      ssip->ssi_name_server, ssip->ssi_domain, ssip->ssi_server) )
    ec_neg1(  ssip->ssi_fd = socket(ssip->ssi_domain, SOCK_STREAM, 0) )
    set_fd_hwm(ssip, ssip->ssi_fd);
    if (ssip->ssi_domain == AF_INET) {
        int socket_option_value = 1;

        ec_neg1( setsockopt(ssip->ssi_fd, SOL_SOCKET, SO_REUSEADDR,
          &socket_option_value, sizeof(socket_option_value)) )
    }
    if (ssip->ssi_server) {
        FD_ZERO(&ssip->ssi_fd_set);
        ec_neg1( bind(ssip->ssi_fd, (struct sockaddr *)&sa, sa_len) )
        ec_neg1( listen(ssip->ssi_fd, SOMAXCONN) )
        FD_SET(ssip->ssi_fd, &ssip->ssi_fd_set);
    }
    else
        ec_neg1( connect(ssip->ssi_fd, (struct sockaddr *)&sa, sa_len) )
    return ssip;

EC_CLEANUP_BGN
    free(ssip);
    return NULL;
EC_CLEANUP_END
}
```

The socket option `SO_REUSEADDR` was explained in Section 8.3.

Closing an `SSI` is straightforward:

```
bool ssi_close(SSI *ssip)
{
    if (ssip->ssi_server) {
        int fd;

        for (fd = 0; fd <= ssip->ssi_fd_hwm; fd++)
            if (FD_ISSET(fd, &ssip->ssi_fd_set))
                (void)close(fd);
        if (ssip->ssi_domain == AF_UNIX)
            (void)unlink(ssip->ssi_name_server);
    }
    else
        (void)close(ssip->ssi_fd);
    free(ssip);
    return true;
}
```

Next comes `ssi_wait_server`, which is just a reworking of the code from Section 8.1.3, only it returns a ready file descriptor (other than the server's socket, that is) instead of using it itself:

```
int ssi_wait_server(SSI *ssip)
{
    if (ssip->ssi_server) {
        fd_set fd_set_read;
        int fd, clientfd;
        struct sockaddr_un from;
        socklen_t from_len = sizeof(from);

        while (true) {
            fd_set_read = ssip->ssi_fd_set;
            ec_neg1( select(ssip->ssi_fd_hwm + 1, &fd_set_read, NULL, NULL,
              NULL) )
            for (fd = 0; fd <= ssip->ssi_fd_hwm; fd++)
                if (FD_ISSET(fd, &fd_set_read)) {
                    if (fd == ssip->ssi_fd) {
                        ec_neg1( clientfd = accept(ssip->ssi_fd,
                          (struct sockaddr *)&from, &from_len) );
                        FD_SET(clientfd, &ssip->ssi_fd_set);
                        set_fd_hwm(ssip, clientfd);
                        continue;
                    }
                    else
                        return fd;
                }
        }
    }
```

```
    else {
        errno = ENOTSUP;
        EC_FAIL
    }

EC_CLEANUP_BGN
    return -1;
EC_CLEANUP_END
}
```

The other two functions are trivial:

```
int ssi_get_server_fd(SSI *ssip)
{
    return ssip->ssi_fd;
}

bool ssi_close_fd(SSI *ssip, int fd)
{
    ec_neg1( close(fd) );
    FD_CLR(fd, &ssip->ssi_fd_set);
    reset_fd_hwm(ssip, fd);
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

That's the whole implementation. For connected sockets (`SOCK_STREAM`), we won't have to use any of the raw system calls in any further examples.

8.5 Socket Implementation of SMI

It's interesting to implement the Simple Messaging Interface (SMI) from Chapter 7 to use sockets so we can compare it to the five other implementations shown there. We'll use the SSI functions from the previous section rather than call the various socket-related system calls directly, so this job is pretty easy.

This section assumes you're very familiar with Chapter 7 and the SMI implementations there; if you're not, skip this section until you've had time to get through Chapter 7—otherwise it won't make much sense.

For sockets, the internal `SMIQ_SKT` structure is simple, mostly because the work of keeping track of a client that the other implementations were burdened with is

in this case done by the `accept` system call and because we already have SSI to handle a lot of the details:

```
typedef struct {
    SMIENTITY sq_entity;            /* entity */
    SSI *sq_ssip;                   /* structure for SSI */
    struct client_id sq_client;     /* client ID */
    size_t sq_msgsize;              /* msg size */
    struct smi_msg *sq_msg;         /* msg buffer */
} SMIQ_SKT;
```

There's not much to do to open the `SMIQ_SKT` since `ssi_open` does most of the work:

```
SMIQ *smi_open_skt(const char *name, SMIENTITY entity, size_t msgsize)
{
    SMIQ_SKT *p = NULL;

    ec_null(  p = calloc(1, sizeof(SMIQ_SKT)) )
    p->sq_msgsize = msgsize + offsetof(struct smi_msg, smi_data);
    ec_null( p->sq_msg = calloc(1, p->sq_msgsize) )
    p->sq_entity = entity;
    ec_null( p->sq_ssip = ssi_open(name, entity == SMI_SERVER) )
    return (SMIQ *)p;

EC_CLEANUP_BGN
    (void)smi_close_skt((SMIQ *)p);
    return NULL;
EC_CLEANUP_END
}
```

Ditto for closing:

```
bool smi_close_skt(SMIQ *sqp)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;
    SSI *ssip;

    if (p != NULL) {
        ssip = p->sq_ssip;
        free(p->sq_msg);
        free(p);
        if (ssip != NULL)
            ec_false( ssi_close(ssip) )
    }
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

As we've seen for some of the other implementations in Chapter 7, all `smi_send_getaddr_skt` has to do is save the `client_id`, if it's called by a server, and return the message address:

```
bool smi_send_getaddr_skt(SMIQ *sqp, struct client_id *client,
  void **addr)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;

    if (p->sq_entity == SMI_SERVER)
        p->sq_client = *client;
    *addr = p->sq_msg;
    return true;
}
```

`smi_send_release_skt` writes the message (using `writeall` from Section 2.9) either to the client's file descriptor, if it's from the server, or to the server's socket file descriptor, if it's from a client:

```
bool smi_send_release_skt(SMIQ *sqp)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;
    int fd;

    if (p->sq_entity == SMI_SERVER)
        ec_neg1( fd = p->sq_client.c_id1 )
    else
        ec_neg1( fd = ssi_get_server_fd(p->sq_ssip) )
    ec_neg1( writeall(fd, p->sq_msg, p->sq_msgsize) )
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

For a server, receiving a message by the server means having to wait for a ready file descriptor, which SSI handles for us via `ssi_wait_server`. A server also has to save the file descriptor that was ready in the `client_id` part of the message so the caller (the server) will know who to respond to. All a client has to do is call `ssi_get_server_fd` to get the server's socket file descriptor:

```
bool smi_receive_getaddr_skt(SMIQ *sqp, void **addr)
{
    SMIQ_SKT *p = (SMIQ_SKT *)sqp;
    ssize_t /*nremain,*/ nread;
    int fd;
```

```
    *addr = p->sq_msg;
    while (true) {
        if (p->sq_entity == SMI_SERVER)
            ec_neg1( fd = ssi_wait_server(p->sq_ssip) )
        else
            ec_neg1( fd = ssi_get_server_fd(p->sq_ssip) )
        ec_neg1( nread = readall(fd, p->sq_msg, p->sq_msgsize) )
        if (nread == 0) {
            if (p->sq_entity == SMI_SERVER) {
                ec_false( ssi_close_fd(p->sq_ssip, fd) )
                continue;
            }
            else {
                errno = ENOMSG;
                EC_FAIL
            }
        }
        else
            break;
    }
    if (p->sq_entity == SMI_SERVER)
        p->sq_msg->smi_client.c_id1 = fd;
    return true;

EC_CLEANUP_BGN
    return false;
EC_CLEANUP_END
}
```

And the final function has no work to do at all:

```
bool smi_receive_release_skt(SMIQ *sqp)
{
    return true;
}
```

With this SMI implementation, all of the message-sending examples from Chapter 7 work on sockets. We already compared the performance relative to the other IPC methods in Section 7.15.

Note that our SMI examples, which were designed for IPC within a machine, use plain server names. This means that when the name gets into `ssi_open` it's treated as being in the `AF_UNIX` domain. You could just as easily use the SMI interface for `AF_INET` message-passing, between machines, although if your names are written that way—beginning with two slashes—only the socket implementation of SMI will work.

8.6 Connectionless Sockets

So far all of our examples and the implementation of SSI used connected sockets of type `SOCK_STREAM`. There was a server that called `listen` and `accept`, and one or more clients that connected to it. The connection stayed open until one side or the other closed it.

By contrast, connectionless sockets don't use a connection so `listen` and `accept` aren't called. A sender specifies the address it wants to send to. A receiver specifies the address it wants to receive from, or it receives from anyone and is told the address of the sender. Receivers always have to `bind` to an external name since that's the only way a sender can reach them. Thus, connectionless sockets are more symmetrical than connected ones; there could be a client/server relationship, but it's more useful sometimes to think of the participants as peers.

8.6.1 About Datagrams

Connectionless communication uses *datagrams*, which are independent chunks of data containing a destination address. There's no attempt to keep datagrams in order, and no guarantee that they'll even arrive. By contrast, a `SOCK_STREAM` guarantees both ordering and arrival. Think of a datagram like a postal letter or an email, and a `SOCK_STREAM` like a telephone call.

In many applications, the lack of guarantees of order and arrival doesn't really matter. For example, suppose the application is for reserving library books. There's a form on the screen which, when submitted, sends a datagram to the library's server, which responds with a confirmation. Reordering the datagrams is perhaps a bit unfair, if one reservation preceded another by a second or so, but since the patrons are unaware of each other's activities, there won't be any complaints. In the rare case that a datagram doesn't arrive at all, the patron won't get the confirmation and will just send the form again. So, in this application, datagrams are much more efficient than setting up a connection just to reserve a book and then immediately disconnecting. On the other hand, if the patron is going to browse the catalog of books for a while interactively, then setting up a connection is probably the best choice.

Figure 8.4 shows two peers sending datagrams to each other's sockets. Compare this to Figure 8.1, which showed connected sockets. The symbols for listening and accepting are gone, and both sockets are bound to names, not just the server's socket.

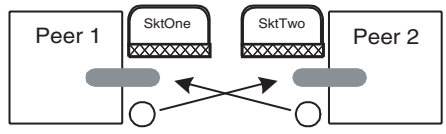

Figure 8.4 Peers sending datagrams.

8.6.2 `sendto` and `recvfrom` System Calls

In all of our previous socket examples, we used `write` (or `writeall`) to send data to a socket, which worked because we had a file descriptor representing the writing end of the connection. But, if the sockets aren't connected, we only have one file descriptor, to our own socket; we need to specify the target as a socket address instead. So, `write` won't do. We need another system call named `sendto` that takes a socket address:

sendto—send message to socket

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

ssize_t sendto(
     int socket_fd,                 /* socket file descriptor */
     const void *message,           /* message to send */
     size_t length,                 /* length of message */
     int flags,                     /* flags */
     const struct sockaddr *sa,     /* target address */
     socklen_t sa_len               /* length of target address */
);
/* Returns number of bytes sent or -1 on error (sets errno) */
```

The argument `socket_fd` is the sender's socket; the receiver is specified with arguments `sa` and `sa_len`. A successful return from `sendto` doesn't mean the message arrived, only that there weren't any locally detected errors.

Your implementation may define some flags, but the only important portable one is `MSG_OOB`, which sends out-of-band (urgent) data; see Section 8.7.

For connectionless sockets, the message of `length` bytes is considered as an indivisible datagram. If the socket channel is full, `sendto` normally blocks until it all

will fit; if the `O_NONBLOCK` flag is set and it won't fit, none of it is sent, –1 is returned, and `errno` is set to `EAGAIN` or `EWOULDBLOCK`.

You can use `sendto` with a connected socket, but in that case the destination socket address is ignored and the output goes to the `socket_fd` socket, just as for `write`. The only advantage over `write` is that you can specify flags, but, if that's all you want, it would be more straightforward to use `send` (Section 8.9.1).

Now we can show an example that does what Figure 8.4 shows, with the addition that each peer reads incoming data. Peer 2 gets a message, makes a little change to it, and sends it to Peer 1. Peer 1 puts together four messages to send to Peer 2 and displays the responses. Note that, unlike the connected-socket examples, here both peers do a `bind`, and neither calls `listen`, `accept`, or `connect`.

```
#define SOCKETNAME1 "SktOne"
#define SOCKETNAME2 "SktTwo"

#define MSG_SIZE 100

int main(void)
{
    struct sockaddr_un sa1, sa2;

    strcpy(sa1.sun_path, SOCKETNAME1);
    sa1.sun_family = AF_UNIX;
    strcpy(sa2.sun_path, SOCKETNAME2);
    sa2.sun_family = AF_UNIX;
    (void)unlink(SOCKETNAME1);
    (void)unlink(SOCKETNAME2);
    if (fork() == 0) { /* Peer 1 */
        int fd_skt;
        ssize_t nread;
        char msg[MSG_SIZE];
        int i;

        sleep(1); /* let peer 2 startup first */
        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
        ec_neg1( bind(fd_skt, (struct sockaddr *)&sa1, sizeof(sa1)) )
        for (i = 1; i <= 4; i++) {
            snprintf(msg, sizeof(msg), "Message #%d", i);
            ec_neg1( sendto(fd_skt, msg, sizeof(msg), 0,
              (struct sockaddr *)&sa2, sizeof(sa2)) )
            ec_neg1( nread = read(fd_skt, msg, sizeof(msg)) )
            if (nread != sizeof(msg)) {
                printf("Peer 1 got short message\n");
                break;
            }
```

```
            printf("Got \"%s\" back\n", msg);
        }
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    else { /* Peer 2 */
        int fd_skt;
        ssize_t nread;
        char msg[MSG_SIZE];

        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
        ec_neg1( bind(fd_skt, (struct sockaddr *)&sa2, sizeof(sa2)) )
            ec_neg1( nread = read(fd_skt, msg, sizeof(msg)) )
        while (true) {
            if (nread != sizeof(msg)) {
                printf("Peer 2 got short message\n");
                break;
            }
            msg[0] = 'm';
            ec_neg1( sendto(fd_skt, msg, sizeof(msg), 0,
              (struct sockaddr *)&sa1, sizeof(sa1)) )
        }
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Putting both processes in the same program isn't normal; we did it just to make the example simpler. However, we had to stick in the call to `sleep` at the start of the child process (Peer 1) to let Peer 2 go first. Not the best way to sequence things but OK for our present purposes. In a real application getting the server started before the clients isn't usually a problem. If it is, a correct way to synchronize the two is in Section 9.2.

This is the output:

```
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
```

For datagrams `read` isn't really a good choice because it doesn't tell us who sent the data. This wasn't a problem for connected sockets because each return from `accept` gave us a separate file descriptor for each client that we could both `read`

and `write`. In the example just given, Peer 2 merely assumed it was supposed to `sendto` Peer 1, but, in general, applications are more complicated than that.

Sure enough, there's the opposite of `sendto` called `recvfrom`:

recvfrom—receive message from socket

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

ssize_t recvfrom(
     int socket_fd,         /* socket file descriptor */
     void *buffer,          /* buffer for received message */
     size_t length,         /* length of buffer */
     int flags,             /* flags */
     struct sockaddr *sa,   /* address of sender */
     socklen_t *sa_len      /* address length */
);
/* Returns number of bytes received, 0, or -1 on error (sets errno) */
```

`recvfrom`'s first three arguments are like those of `read`. In addition, it has a `flags` argument and two arguments that receive the sender's socket address. Prior to the call, you must set `sa` to something big enough to hold the sender's socket address, and then `sa_len` to the size of that storage. On return `sa_len` is set to the actual size of the address. Then, if you want, you can pass `sa` and `sa_len` directly to `sendto` for a response.

Like `sendto`, `recvfrom` always returns a complete message for a connectionless socket. It waits for one if `O_NONBLOCK` is clear, or, if set, returns –1 with `errno` to `EAGAIN` or `EWOULDBLOCK`.

You could use `recvfrom` for a connected socket if you want to know the source address, but there's not a whole lot you can do with that address since it would be ignored in a call to `sendto` anyway, as previously explained. Another way to get the address of a socket whose file descriptor you have would be to call `getsockname` (Section 8.9.2).

There are three portable flags that you can use with `recvfrom`:

`MSG_OOB` Receives out-of-band data, if any; see Section 8.7.

`MSG_PEEK` Returns the message, but leaves it unread for the next `read`, `recvfrom`, or other input operation.

`MSG_WAITALL` For connected sockets only, causes `recvfrom` to block until the entire requested length is available, unless `MSG_PEEK` is also set

or the call is interrupted by a signal, termination of the connection, or an error. Has no effect on connectionless sockets because the message is always indivisible.

Here's my example reworked to use `recvfrom` and to have several peers sending to the same socket, so it's a client/server arrangement:

```
#define SOCKETNAME_SERVER "SktOne"
#define SOCKETNAME_CLIENT "SktTwo"

static struct sockaddr_un sa_server;

#define MSG_SIZE 100

static void run_client(int nclient)
{
    struct sockaddr_un sa_client;
    int fd_skt;
    ssize_t nrecv;
    char msg[MSG_SIZE];
    int i;

    if (fork() == 0) { /* client */
        sleep(1); /* let server startup first */
        ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
        snprintf(sa_client.sun_path, sizeof(sa_client.sun_path),
          "%s-%d", SOCKETNAME_CLIENT, nclient);
        (void)unlink(sa_client.sun_path);
        sa_client.sun_family = AF_UNIX;
        ec_neg1( bind(fd_skt, (struct sockaddr *)&sa_client,
          sizeof(sa_client)) )
        for (i = 1; i <= 4; i++) {
            snprintf(msg, sizeof(msg), "Message #%d", i);
            ec_neg1( sendto(fd_skt, msg, sizeof(msg), 0,
              (struct sockaddr *)&sa_server, sizeof(sa_server)) )
            ec_neg1( nrecv = read(fd_skt, msg, sizeof(msg)) )
            if (nrecv != sizeof(msg)) {
                printf("client got short message\n");
                break;
            }
            printf("Got \"%s\" back\n", msg);
        }
        ec_neg1( close(fd_skt) )
        exit(EXIT_SUCCESS);
    }
    return;
```

```
EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

static void run_server(void)
{
    int fd_skt;
    ssize_t nrecv;
    char msg[MSG_SIZE];
    struct sockaddr_storage sa;
    socklen_t sa_len;

    ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
    ec_neg1( bind(fd_skt, (struct sockaddr *)&sa_server, sizeof(sa_server)) )
    while (true) {
        sa_len = sizeof(sa);
        ec_neg1( nrecv = recvfrom(fd_skt, msg, sizeof(msg), 0,
          (struct sockaddr *)&sa, &sa_len) )
        if (nrecv != sizeof(msg)) {
            printf("server got short message\n");
            break;
        }
        msg[0] = 'm';
        ec_neg1( sendto(fd_skt, msg, sizeof(msg), 0,
          (struct sockaddr *)&sa, sa_len) )
    }
    ec_neg1( close(fd_skt) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}

int main(void)
{
    int nclient;

    strcpy(sa_server.sun_path, SOCKETNAME_SERVER);
    sa_server.sun_family = AF_UNIX;
    (void)unlink(SOCKETNAME_SERVER);
    for (nclient = 1; nclient <= 3; nclient++)
        run_client(nclient);
    run_server();
    exit(EXIT_SUCCESS);
}
```

In this new example, only the server's socket address is known to the server and the clients. A client socket address is known only to that client; the server gets it from its call to `recvfrom`. I left the `read` calls in the clients alone, but those could have been calls to `recvfrom` as well. There's more output since there are now three clients:

```
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
Got "message #1" back
Got "message #2" back
Got "message #3" back
Got "message #4" back
```

8.6.3 `sendmsg` and `recvmsg`

There are variants of `sendto` and `recvfrom` that use the same flags and have the same return value but which use a single argument that points to a `msghdr` structure that contains both the socket address and the message. They're capable of scatter reading and gather writing, just like `readv` and `writev`, which we saw way back in Section 2.15.

sendmsg—send message to socket using msghdr structure

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

ssize_t sendmsg(
     int socket_fd,                /* socket file descriptor */
     const struct msghdr *message, /* message */
     int flags                     /* flags */
);
/* Returns number of bytes sent or -1 on error (sets errno) */
```

recvmsg—receive message from socket using msghdr structure

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

ssize_t recvmsg(
     int socket_fd,                /* socket file descriptor */
     struct msghdr *message,       /* message */
     int flags                     /* flags */
);
/* Returns number of bytes received, 0, or -1 on error (sets errno) */
```

struct msghdr—structure for sendmsg and recvmsg

```
struct msghdr {
    void *msg_name;           /* optional address */
    socklen_t msg_namelen;    /* size of address */
    struct iovec *msg_iov;    /* scatter/gather array */
    int msg_iovlen;           /* number of elements in msg_iov */
    void *msg_control;        /* ancillary data */
    socklen_t msg_controllen; /* ancillary data buffer len */
    int msg_flags;            /* flags on received message */
};
```

For `sendmsg`, the socket address is pointed to by the misnamed `msg_name` member, and the socket-address length goes in `msg_namelen`. The data to be sent is pointed to via the `msg_iov` member, which points to a `iovec` structure and is used just as with `writev`. Member `msg_iovlen` is the number of elements in the `msg_iov` array. I explained setting up these members in Section 2.15. Member `msg_flags` is ignored.

For `recvmsg`, you can set `msg_name` to `NULL` or to a buffer of length `msg_namelen` to receive the sender's socket address, similar to the way `recvfrom` works. When `recvmsg` returns, `msg_namelen` is changed to the actual socket-address length. Members `msg_iov` and `msg_iovlen` are used as with `readv`. There's one portable flag that can be set in the returned `msg_flags` member: `MSG_TRUNC`, which means that the message was too long to fit in the supplied buffers.

The socket-address members, `msg_name` and `msg_namelen`, are used as with connectionless sockets; for connected sockets they're ignored.

The two other members I didn't explain, `msg_control` and `msg_controllen`, are used for ancillary data that has to do with access rights. I won't go into it here, but you can read about it in [SUS2002] or in your system's documentation.

Here's just the `run_server` function from the previous section redone to use `recvmsg` and `sendmsg`:

```
static void run_server(void)
{
    int fd_skt;
    ssize_t nrecv;
    char msg[MSG_SIZE];
    struct sockaddr_storage sa;
```

```
    struct msghdr m;
    struct iovec v;

    ec_neg1( fd_skt = socket(AF_UNIX, SOCK_DGRAM, 0) )
    ec_neg1( bind(fd_skt, (struct sockaddr *)&sa_server, sizeof(sa_server)) )
    while (true) {
        memset(&m, 0, sizeof(m));
        m.msg_name = &sa;
        m.msg_namelen = sizeof(sa);
        v.iov_base = msg;
        v.iov_len = sizeof(msg);
        m.msg_iov = &v;
        m.msg_iovlen = 1;
        ec_neg1( nrecv = recvmsg(fd_skt, &m, 0) )
        if (nrecv != sizeof(msg)) {
            printf("server got short message\n");
            break;
        }
        ((char *)m.msg_iov->iov_base)[0] = 'm';
        ec_neg1( sendmsg(fd_skt, &m, 0) )
    }
    ec_neg1( close(fd_skt) )
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

There's additional work to set up the `msghdr` structure, but the `recvmsg` and `sendmsg` calls are then very simple. Imagine an application where there are a lot of different messages being sent to different addresses, and you can begin to see the benefits of keeping the addresses in the same structure as the message data.

8.6.4 Using connect with Connectionless Sockets

Normally a client uses `connect` (Section 8.1.2) to connect with a server's socket, but it also works with connectionless sockets to establish a default address for subsequent `send`s and `recv`s, which don't take a socket-address argument. The specifics of those two system calls are in Section 8.9.1.

Calling `connect` with a `NULL` socket address removes the default address.

8.7 Out-of-Band Data

Occasionally it's necessary to send an urgent message on a socket connection. If it's sent inline, it won't be read until everything ahead of it is read first, so there's a way to send it *out-of-band*. The receiver also must be receiving out-of-band messages.

To send an out-of-band message, set the `MSG_OOB` flag with `sendto`, `sendmsg`, or `send`.

A receiver using `select` (Section 8.1.3) finds out about an out-of-band message through the so-called "error" set of file descriptors passed as the fourth argument to `select`. Then it gets the message by setting the `MSG_OOB` flag in a call to `recvfrom`, `recvmsg`, or `recv`. Or, it can just check for an out-of-band message every time it's about to receive ordinary data by calling one of the receiving functions with the `O_NONBLOCK` flag set on the socket file descriptor (via `fcntl`, Section 3.8.3) and `MSG_OOB` set for the receiving call.

If the `SO_OOBINLINE` option is set (Section 8.3), out-of-band data may be placed inline, along with the other data. The protocol may precede the out-of-band data with an out-of-band mark, ignored when data is read, but whose presence can be detected with a call just for that purpose:

sockatmark—test presence of out-of-band mark

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

int sockatmark(
    int socket_fd      /* socket file descriptor */
);
/* Returns 1 if at mark, 0 if not, or -1 on error (sets errno) */
```

The problem with `sockatmark` is that, if it returns 0 because the socket is empty, out-of-band data may arrive after the call to `sockatmark` but before the next receive operation. Since the out-of-band mark is ignored when data is received, the mark will be lost. The way to avoid this is ensure that data is ready to be received before calling `sockatmark`, with, for example, a call to `select`. That way a 0 return from `sockatmark` means unambiguously that there is data, but it is not preceded by the out-of-band mark.

You can also arrange to get a `SIGURG` signal when out-of-band data has arrived. To do this, call `fcntl` with the `F_SETOWN` operation to set the process or process-group that should get the signal. For more details, see [SUS2002] or your system's documentation.

8.8 Network Database Functions

These *network database* functions provide information about hosts, networks, protocols, and services. I already introduced the most important of these functions, `getaddrinfo`, in Section 8.2.6. Another handy function, `gethostname`, was in Section 8.2.7. Here I'll describe the rest, grouped according to the kind of information they deal with.

8.8.1 Host Functions

The three functions for scanning the host database are:

sethostent—start host-database scan

```
#include <netdb.h>

void sethostent(
     int stayopen           /* leave connection open? */
);
```

gethostent—get next host-database entry

```
#include <netdb.h>

struct hostent *gethostent(void);
/* Returns next entry or NULL on end of database (errno not defined) */
```

endhostent—end host-database scan

```
#include <netdb.h>

void endhostent(void);
```

struct hostent—structure for host-database functions

```
struct hostent {
     char *h_name;          /* official host name */
     char **h_aliases;      /* array of alternative host names */
     int h_addrtype;        /* address family (not type) */
     int h_length;          /* length of each address */
     char **h_addr_list;    /* array of pointers to network addresses */
};
```

You can call `gethostent` in a loop to scan through all the known host names. You start the scan with `sethostent`, using the argument to indicate whether the connection to the host database should stay open, or whether `gethostent` should open and close it each time it's called. You call `endhostent` at the end. Thus, it sounds like a lot could be happening, but in practice all these functions are likely to do is scan the /etc/hosts file on the local machine.

There are two arrays in the `hostent` structure, each terminated with a `NULL` pointer. One, `h_aliases`, is an array of alternate host names. The other, `h_addr_list`, is an array of pointers to network addresses. For `AF_INET`, these are IP addresses stored as 32-bit binary numbers in network byte order; therefore, use `ntohl` (Section 8.1.4) to convert them to local byte order, or use `inet_ntoa` (Section 8.2.3) or `inet_ntop` (Section 8.9.5) to get them as a dotted string

Here's an example:

```
static void hostdb(void)
{
    struct hostent *h;

    sethostent(true);
    while ((h = gethostent()) != NULL)
        display_hostent(h);
    endhostent();
}

static void display_hostent(struct hostent *h)
{
    int i;

    printf("name: %s; type: %d; len: %d\n", h->h_name, h->h_addrtype,
      h->h_length);
    for (i = 0; h->h_aliases[i] != NULL; i++)
        printf("\t%s\n", h->h_aliases[i]);
    if (h->h_addrtype == AF_INET) {
        for (i = 0; h->h_addr_list[i] != NULL; i++)
            printf("\t%s\n",
              inet_ntoa(*((struct in_addr *)h->h_addr_list[i])));
    }
}
```

The output from a little main program (not shown) that called `hostdb` was:

```
name: localhost; type: 2; len: 4
        127.0.0.1
name: sol; type: 2; len: 4
        loghost
        192.168.0.10
name: bsd; type: 2; len: 4
        192.168.0.15
name: suse2; type: 2; len: 4
        suse2.MSHOME
        192.168.0.19
```

For comparison, here's what the /etc/hosts file looked like on the same system:

```
127.0.0.1       localhost
192.168.0.10    sol     loghost
192.168.0.15    bsd
192.168.0.19    suse2   suse2.MSHOME
```

To look up a host by its name, you can call `gethostbyname`. It uses DNS if such access is available, not only the /etc/hosts file. In fact, it was the primary way you got network addresses until `getaddrinfo` came along, and it's still used a lot more than `getaddrinfo` is because `gethostbyname` is very old, and that's how almost everyone learned to do things. But, unlike with `getaddrinfo`, with `gethostbyname` all you get is the IP address—you still have to build the `sockaddr_in` structure yourself, as we did in Section 8.2.3. In addition, `gethostbyname` can't handle IPv6 addresses. Here's the synopsis:

gethostbyname—look up host by name

```
#include <netdb.h>

struct hostent *gethostbyname(
    const char *nodename,      /* node name */
);
/* Returns pointer to hostent or NULL on error (sets h_errno) */
```

The parenthetical remark at the end of the synopsis is no misprint—this function really does set `h_errno`, not `errno`, and the codes it uses aren't `errno` codes, either. (I didn't code an "ec" macro for `h_errno` because I don't use the function.)

Here's an example (`display_hostent` is from the previous example):

```
static void gethostbyname_ex(void)
{
    struct hostent *h;

    if ((h = gethostbyname("www.yahoo.com")) == NULL) {
        if (h_errno == HOST_NOT_FOUND)
            printf("host not found\n");
        else
            printf("h_errno = %d\n", h_errno);
    }
    else
        display_hostent(h);
}
```

Here's the output we got from a call to this function:

```
name: www.yahoo.akadns.net; type: 2; len: 4
        www.yahoo.com
        66.218.71.95
        66.218.70.49
        66.218.71.88
        66.218.71.81
        66.218.71.86
        66.218.71.92
        66.218.71.94
        66.218.71.89
        66.218.70.48
        66.218.71.93
        66.218.70.50
        66.218.71.87
        66.218.71.84
```

You may want to compare this to the output shown in Section 8.2.6, where we did pretty much the same thing with a call to `getaddrinfo`.

The opposite of `gethostbyname` is `gethostbyaddr`, which uses the host database (perhaps DNS) to translate an IP address to a name:

gethostbyaddr—look up host by address

```
#include <netdb.h>

struct hostent *gethostbyaddr(
     const void *addr,      /* IP address */
     socklen_t len,         /* length of address */
     int family             /* family (called "type" in SUS) */
);
/* Returns pointer to hostent or NULL on error (sets h_errno) */
```

Here's an example program (`inet_addr` is in Section 8.2.3):

```
static void gethostbyaddr_ex(void)
{
    struct hostent *h;
    in_addr_t a;

    ec_neg1( a = inet_addr("66.218.71.94") )
    if ((h = gethostbyaddr(&a, sizeof(a), AF_INET)) == NULL) {
        if (h_errno == HOST_NOT_FOUND)
            printf("address not found\n");
        else
            printf("h_errno = %d\n", h_errno);
    }
    else
        display_hostent(h);
    return;
```

```
EC_CLEANUP_BGN
    EC_FLUSH("gethostbyaddr_ex")
EC_CLEANUP_END
}
```

The output:

```
name: w15.www.scd.yahoo.com; type: 2; len: 4
        66.218.71.94
```

What happened is that when we got the IP addresses for *www.yahoo.com,* 66.218.71.94 was on the list, but when we asked for that IP's name we got *w15.www.scd.yahoo.com*. That's the way DNS works—apparently Yahoo uses a lot of servers, which, of course, it must.

`gethostbyaddr` is just as obsolete as `gethostbyname`. The modern function is the counterpart to `getaddrinfo` (Section 8.2.6), called `getnameinfo`:

getnameinfo—get name information

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

int getnameinfo(
     const struct sockaddr *sa,     /* socket address */
     socklen_t sa_len,              /* socket-address length */
     char *nodename,                /* node name */
     socklen_t nodelen,             /* node-name buffer length */
     char *servname,                /* service name */
     socklen_t servlen,             /* service-name buffer length */
     unsigned flags                 /* flags */
);
/* Returns 0 on success or error number on error */
```

`getnameinfo` takes a whole socket address, not just a bare IP address, because it potentially works with lots of different families. Most importantly, it works with IPv6 addresses, which `gethostbyname` doesn't. The answers come back into two buffers: `nodename`, of length `nodelen`, for the node (or host) name, and `servname`, of length `servlen`, for the service name. If a buffer pointer is `NULL` or its length is 0, you don't get that information back.

Like `getaddrinfo`, the error numbers returned by `getnameinfo` aren't `errno` codes, but special "EAI" codes, and you can use the function `gai_strerror` (Section 8.2.6) on them if you want. Or, use the `ec_ai` macro that we've provided just for these two functions.

Various flags that you can OR together control what `getnameinfo` returns:

NI_NOFQDN	Return just the node name part of local hosts, not the fully qualified domain name.
NI_NUMERICHOST	Return the numeric form (dotted notation for IPv4, colon notation for IPv6) of the address instead of the name.
NI_NAMEREQD	Return an error if the host's name can't be found. Normally the numeric form is returned in this case.
NI_NUMERICSERV	Return the port number (as a string) instead of the service name.
NI_DGRAM	Look for a SOCK_DGRAM (UDP) service. Normally it looks for a SOCK_STREAM (TCP) service.

Here's an example:

```
static void getnameinfo_ex(void)
{
    struct sockaddr_in sa;
    char nodename[200], servname[200];

    sa.sin_family = AF_INET;
    sa.sin_port = htons(80);
    sa.sin_addr.s_addr =  inet_addr("216.109.125.70");
    ec_ai( getnameinfo((struct sockaddr *)&sa, sizeof(sa), nodename,
      sizeof(nodename), servname, sizeof(servname), 0) )
    printf("node: %s; service: %s\n", nodename, servname);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("getnameinfo_ex")
EC_CLEANUP_END
}
```

The output:

```
node: w17.www.dcn.yahoo.com; service: http
```

Another host-oriented function to talk about is:

gethostid—get identifier for local host

```
#include <unistd.h>

long gethostid(void);
/* Returns identifier (no error return) */
```

`gethostid` looks like it might return the local machine's IP, but it doesn't necessarily do that. All it's required to do is return a unique identifier, and whether that even works probably depends on whether such an identifier has been set at boot time. I've documented it here just so you don't accidentally confuse it with a useful function.

There's another function, not directly associated with networking, that provides information to identify the system:

uname—get info about current system

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

int uname(
    struct utsname *info      /* returned info */
);
/* Returns non-negative value on success or -1 on error (sets errno) */
```

struct utsname—structure for uname

```
struct utsname {
    char sysname[];           /* OS name */
    char nodename[];          /* node name within network */
    char release[];           /* release number (as string) */
    char version[];           /* version number (as string) */
    char machine[];           /* hardware type or computer model */
};
```

Unfortunately, none of the members of the `utsname` structure[8] are standardized, so you can't use them to control your application's processing. It would be nice, say, to test the machine member to see if you're on an Intel CPU; however, each system that runs on that CPU formats the string the way it wants, and the word "Intel" or "x86" might not even be there. What you can do is use the strings for display purposes—maybe to label performance-testing output. And, naturally, the `uname` system call is the guts of the `uname` command, our version (lacking options) of which is:

```
int main(void)
{
    struct utsname info;

    ec_neg1( uname(&info) )
    printf("sysname = %s\n", info.sysname);
    printf("nodename = %s\n", info.nodename);
```

8. Empty brackets as shown in the synopsis aren't legal C, but you get the idea. Each implementation allocates whatever space it needs.

```
    printf("release = %s\n", info.release);
    printf("version = %s\n", info.version);
    printf("machine = %s\n", info.machine);
    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
}
```

Here's the output[9] from running this program on our four test systems; see if you can spot any patterns:

```
sysname = SunOS
nodename = sol
release = 5.8
version = Generic_108529-13
machine = i86pc

sysname = Darwin
nodename = Marc-Rochkinds-Computer.local.
release = 6.6
version = Darwin Kernel Version 6.6: Thu May  1 21:48:54 PDT 2003;
  root:xnu/xnu-344.34.obj~1/RELEASE_PPC
machine = Power Macintosh

sysname = FreeBSD
nodename = bsd.MSHOME
release = 4.6-RELEASE
version = FreeBSD 4.6-RELEASE #0: Tue Jun
machine = i386

sysname = Linux
nodename = suse2
release = 2.4.18-4GB
version = #1 Wed Mar 27 13:57:05 UTC 2002
machine = i686
```

8.8.2 Network Functions

Functions in this category get information about networks that the local machine might be connected to. First come three functions like the ones for hosts (e.g., `gethostent`) for scanning for networks:

9. Output was folded to fit within the page margins.

setnetent—start network-database scan

```
#include <netdb.h>

void setnetent(
     int stayopen           /* leave connection open? */
);
```

getnetent—get network-database entry

```
#include <netdb.h>

struct netent *getnetent(void);
/* Returns pointer to netent or NULL on end (errno not defined) */
```

endnetent—end network-database scan

```
#include <netdb.h>

void endnetent(void);
```

struct netent—structure for network-database functions

```
struct netent {
     char *n_name;          /* official network name */
     char **n_aliases;      /* array of alternative network names */
     int n_addrtype;        /* address family (not type) */
     uint32_t n_net;        /* network number */
};
```

Here's an example:

```
static void netdb(void)
{
    struct netent *n;

    setnetent(true);
    while ((n = getnetent()) != NULL)
        display_netent(n);
    endnetent();
}

static void display_netent(struct netent *n)
{
    int i;

    printf("name: %s; type: %d; number: %lu\n", n->n_name, n->n_addrtype,
      (unsigned long)n->n_net);
    for (i = 0; n->n_aliases[i] != NULL; i++)
        printf("\t%s\n", n->n_aliases[i]);
}
```

With this output:

```
name: loopback; type: 2; number: 127
name: arpanet; type: 2; number: 10
        arpa
```

Most, if not all, machines have a loopback network for testing. This machine (running Solaris) also has a connection to an Internet network that it calls "arpanet" in honor of its history. Remember that the "type" displayed is really the family; it turns out that the macro `AF_INET` is defined as 2.

Analogously to the host database, there are functions for finding an entry by network number or by name:

getnetbyname—look up network by name

```
#include <netdb.h>

struct netent *getnetbyname(
     const char *name       /* network name (to match n_name member) */
);
/* Returns pointer to netent or NULL if not found (errno not defined) */
```

getnetbyaddr—look up network by number

```
#include <netdb.h>

struct netent *getnetbyaddr(
     uint32_t net,          /* network number (to match n_net member) */
     int type               /* family (to match n_addrtype member) */
);
/* Returns pointer to netent or NULL if not found (errno not defined) */
```

8.8.3 Protocol Functions

Following the pattern, the functions for scanning the protocol database are:

setprotoent—start protocol-database scan

```
#include <netdb.h>

void setprotoent(
     int stayopen           /* leave connection open? */
);
```

getprotoent—get protocol-database entry

```
#include <netdb.h>

struct protoent *getprotoent(void);
/* Returns pointer to protoent or NULL on end (errno not defined) */
```

endprotoent—end protocol-database scan

```
#include <netdb.h>

void endprotoent(void);
```

struct protoent—structure for protocol-database functions

```
struct protoent {
     char *p_name;          /* official protocol name */
     char **p_aliases;      /* array of alternative protocol names */
     int p_proto;           /* protocol number */
};
```

On some systems, you can use a protocol number for a protocol level when you call `setsockopt` or `getsockopt`, but that's nonstandard.

The example code is pretty obvious if you've been reading along:

```
static void protodb(void)
{
    struct protoent *p;

    setprotoent(true);
    while ((p = getprotoent()) != NULL)
        display_protoent(p);
    endprotoent();
}

static void display_protoent(struct protoent *p)
{
    int i;

    printf("name: %s; number: %d\n", p->p_name, p->p_proto);
    for (i = 0; p->p_aliases[i] != NULL; i++)
        printf("\t%s\n", p->p_aliases[i]);
}
```

But the output is pretty interesting. On SuSE Linux I got 135 protocols. Here's just part of the output:

```
...
name: mobile; number: 55
        MOBILE
name: tlsp; number: 56
        TLSP
name: skip; number: 57
        SKIP
name: ipv6-icmp; number: 58
        IPv6-ICMP
```

```
        ICMPV6
        icmpv6
        icmp6
name: ipv6-nonxt; number: 59
        IPv6-NoNxt
name: ipv6-opts; number: 60
        IPv6-Opts
name: cftp; number: 62
        CFTP
name: sat-expak; number: 64
        SAT-EXPAK
name: kryptolan; number: 65
        KRYPTOLAN
...
```

There are the two more predictable functions to round out the set:

getprotobyname—look up protocol by name

```
#include <netdb.h>

struct protoent *getprotobyname(
    const char *name          /* protocol name */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

getprotobynumber—look up protocol by number

```
#include <netdb.h>

struct protoent *getprotobynumber(
    int proto                 /* protocol number */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

8.8.4 Service Functions

There's one more set of scanning and get-by-name and number functions, this time for services. These scan the local system's /etc/services file.

setservent—start service-database scan

```
#include <netdb.h>

void setservent(
    int stayopen              /* leave connection open? */
);
```

getservent—get service-database entry

```
#include <netdb.h>

struct servent *getservent(void);
/* Returns pointer to servent or NULL on end (errno not defined) */
```

endservent—end service-database scan

```
#include <netdb.h>

void endservent(void);
```

struct servent—structure for service-database functions

```
struct servent {
     char *s_name;          /* official service name */
     char **s_aliases;      /* array of alternative service names */
     int s_port;            /* port number */
     char *s_proto;         /* name of protocol for this service */
};
```

getservbyname—look up service by name

```
#include <netdb.h>

struct servent *getservbyname(
     const char *name,      /* service name */
     const char *proto      /* protocol name */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

getservbyport—look up service by port

```
#include <netdb.h>

struct servent *getservbyport(
     int port,              /* port */
     const char *proto      /* protocol name */
);
/* Returns pointer to protoent or NULL if not found (errno not defined) */
```

And our last example of this ilk is:

```
static void servdb(void)
{
    struct servent *s;

    setservent(true);
    while ((s = getservent()) != NULL)
        display_servent(s);
    endservent();
}

static void display_servent(struct servent *s)
{
    int i;
```

```
    printf("name: %s; port: %d; protocol: %s\n", s->s_name,
       ntohs(s->s_port), s->s_proto);
    for (i = 0; s->s_aliases[i] != NULL; i++)
        printf("\t%s\n", s->s_aliases[i]);
}
```

The output is huge because /etc/services files are typically huge. Here's a fragment:

```
...
name: ftp-data; port: 20; protocol: tcp
name: ftp-data; port: 20; protocol: udp
name: ftp; port: 21; protocol: tcp
name: fsp; port: 21; protocol: udp
name: ssh; port: 22; protocol: tcp
name: ssh; port: 22; protocol: udp
name: telnet; port: 23; protocol: tcp
name: telnet; port: 23; protocol: udp
name: smtp; port: 25; protocol: tcp
        mail
name: smtp; port: 25; protocol: udp
        mail
name: nsw-fe; port: 27; protocol: tcp
name: nsw-fe; port: 27; protocol: udp
name: msg-icp; port: 29; protocol: tcp
name: msg-icp; port: 29; protocol: udp
name: msg-auth; port: 31; protocol: tcp
name: msg-auth; port: 31; protocol: udp
...
```

8.8.5 Network Interface Functions

There's a set of functions to retrieve the names of the network interfaces and their index numbers. First, here's a function that gets them all as an array of `if_nameindex` structures and a corresponding function to free the array:

if_nameindex—get all network interface names and indexes

```
#include <net/if.h>

struct if_nameindex *if_nameindex(void);
/* Returns array or NULL on error (sets errno) */
```

if_freenameindex—free array allocated by if_nameindex

```
#include <net/if.h>

void if_freenameindex(
     struct if_nameindex *ptr   /* pointer to array */
);
```

struct if_nameindex—structure for network-interface functions

```
struct if_nameindex {
    unsigned if_index;     /* interface index */
    char *if_name;         /* interface name */
};
```

Here's a function that displays the indexes and interfaces:

```
static void ifdb(void)
{
    struct if_nameindex *ni;
    int i;

    ec_null( ni = if_nameindex() )
    for (i = 0; ni[i].if_index != 0 || ni[i].if_name != NULL; i++)
        printf("index: %d; name: %s\n", ni[i].if_index, ni[i].if_name);
    if_freenameindex(ni);
    return;

EC_CLEANUP_BGN
    EC_FLUSH("ifdb")
EC_CLEANUP_END
}
```

The output on Solaris was:

```
index: 1; name: lo0
index: 2; name: iprb0
```

and on SuSE Linux:

```
index: 1; name: lo
index: 2; name: eth0
```

These sort of match up with the network names displayed by the functions in Section 8.8.2: a loopback interface, and an Internet interface using Ethernet.

There's another function to map a name to its index:

if_nametoindex—map network interface name to index

```
#include <net/if.h>

unsigned if_nametoindex(
    const char *ifname     /* interface name */
);
/* Returns index or 0 on error (errno not defined) */
```

Be careful—this function returns 0 if the name isn't found, not –1.

To map an index to a name, you call:

if_indextoname—map network interface index to name

```
#include <net/if.h>

char *if_indextoname(
     unsigned ifindex,      /* interface index */
     char *ifname           /* interface name */
);
/* Returns name or NULL on error (sets errno) */
```

The `ifname` argument must be a buffer of at least `IF_NAMESIZE` bytes, which includes space for the terminating NUL byte. It also returns a pointer to that buffer.

8.9 Miscellaneous System Calls

This section describes some networking system calls that don't fit into the previous sections.

8.9.1 `send` and `recv`

These two functions behave exactly like `write` and `read`, except they allow you to specify flags. Because they don't have a socket address argument, they're normally used with connected sockets. For connectionless sockets, `sendto`, `sendmsg`, `recvfrom`, and `recvmsg` are more convenient.

send—send data to socket

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

ssize_t send(
     int socket_fd,         /* socket file descriptor */
     const void *data,      /* data to send */
     size_t length,         /* length of data */
     int flags              /* flags */
);
/* Returns number of bytes sent or -1 on error (sets errno) */
```

recv—receive data from socket

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

ssize_t recv(
     int socket_fd,         /* socket file descriptor */
     void *buffer,          /* buffer to receive data */
     size_t length,         /* length of buffer */
     int flags              /* flags */
);
/* Returns number of bytes received, 0, or -1 on error (sets errno) */
```

You can use the `MSG_OOB` flag with both `send` and `recv`; it was explained in Section 8.7. In addition, for `recv`, you can specify `MSG_PEEK` and/or `MSG_WAITALL`, which were explained in Section 8.6.2. We could have used `MSG_WAITALL` with `recv` instead of calling `readall` in the SMI implementation in Section 8.5.

8.9.2 `getsockname` and `getpeername`

`getsockname` gets the socket address that a socket has been bound to with a previous call to `bind`:

getsockname—get socket address

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

int getsockname(
     int socket_fd,         /* socket file descriptor */
     struct sockaddr *sa,   /* socket address */
     socklen_t *sa_len      /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

As usual for socket-address-returning functions, on input `sa` should point to a buffer big enough for the socket address and `sa_len` should be the size of that buffer. On output `sa_len` is the actual size of the address.

A similar function gets the socket address of the socket connected to a socket:

getpeername—get socket address of connected socket

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

int getpeername(
     int socket_fd,         /* socket file descriptor */
     struct sockaddr *sa,   /* socket address */
     socklen_t *sa_len      /* address length */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

If the connected socket is unbound, as is often the case, then what `getpeername` returns is unspecified. You don't get an error return in this case.

8.9.3 socketpair

AF_UNIX sockets act something like bidirectional FIFOs, and, as with FIFOs, each process that wants to communicate sets up its own socket. By contrast, the pipe system call returns two file descriptors that a child process can inherit. The socketpair system call sort of blends the two approaches—you get two socket file descriptors with one call:

socketpair—create pair of sockets

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

int socketpair(
     int domain,            /* domain (AF_UNIX, AF_INET, etc.) */
     int type,              /* SOCK_STREAM, SOCK_DGRAM, etc. */
     int protocol           /* specific to type; usually zero */
     int socket_vector[2]   /* returned socket file descriptors */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

The first three arguments are just like the ones for socket. What domains and types are handled is up to the implementation, and typically all that's supported is AF_UNIX and SOCK_STREAM.

8.9.4 shutdown

When you close a connected socket that still has data to be sent or received, the system keeps trying for a while until it gives up and discards the data. You can indicate before you close the socket that you don't want the data by calling shutdown:

shutdown—shut down socket send and/or receive operations

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

int shutdown(
     int socket_fd,         /* socket file descriptor */
     int how                /* SHUT_RD, SHUT_WR, or SHUT_RDWR */
);
/* Returns 0 on success or -1 on error (sets errno) */
```

SHUT_RD disables further receives, SHUT_WR disables sends, and SHUT_RDWR disables both.

8.9.5 `inet_ntop` and `inet_pton`

The `inet_ntoa` and `inet_addr` functions discussed in Section 8.2.3 convert between binary IPv4 addresses and dotted strings. The following two functions, `inet_ntop` and `inet_pton` are smarter: They work with both IPv4 and IPv6 addresses.

inet_ntop—convert IPv4 or IPv6 binary address to string

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

const char *inet_ntop(
    int domain,            /* AF_INET or AF_INET6 */
    const void *src,       /* pointer to binary address (input) */
    char *dst,             /* string (output) */
    socklen_t dst_len      /* length of dst buffer */
);
/* Returns string or NULL on error (sets errno) */
```

inet_pton—convert IPv4 or IPv6 string address to binary

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

int inet_pton(
    int domain,            /* AF_INET or AF_INET6 */
    const char *src,       /* string (input) */
    void *dst              /* buffer for binary address (output) */
);
/* Returns 1 on success, 0 on bad string, or -1 on error (sets errno) */
```

You call `inet_ntop` with `src` pointing to the binary address, either a 32-bit number for IPv4 or a 16-byte array for IPv6. The result is placed in the buffer pointed to by `dst`. Two macros are available for sizing `dst`: `INET_ADDRSTRLEN` for IPv4, and `INET6_ADDRSTRLEN` for IPv6.

For `inet_pton`, the input is a string in dotted or colon notation, and the binary output goes into whatever `dst` points to, which better be big enough, because there's no length argument. You need 32 bits for IPv4 addresses and 128 bits (16 bytes) for IPv6.

Here's an example:

```
static void cvt(void)
{
    char ipv6[16], ipv6str[INET6_ADDRSTRLEN], ipv4str[INET_ADDRSTRLEN];
    uint32_t ipv4;
    int r;
```

```
    ec_neg1( r = inet_pton(AF_INET, "66.218.71.94", &ipv4) )
    if (r == 0)
        printf("Can't convert\n");
    else {
        ec_null( inet_ntop(AF_INET, &ipv4, ipv4str, sizeof(ipv4str)) )
        printf("%s\n", ipv4str);
    }
    ec_neg1( r = inet_pton(AF_INET6,
      "FEDC:BA98:7654:3210:FEDC:BA98:7654:3210", &ipv6) )
    if (r == 0)
        printf("Can't convert\n");
    else {
        ec_null( inet_ntop(AF_INET6, &ipv6, ipv6str, sizeof(ipv6str)) )
        printf("%s\n", ipv6str);
    }
    return;

EC_CLEANUP_BGN
    EC_FLUSH("cvt")
EC_CLEANUP_END
}
```

The output from calling this function:

```
66.218.71.94
fedc:ba98:7654:3210:fedc:ba98:7654:3210
```

8.10 High-Performance Considerations

I showed an example of a Web server in Section 8.4.4, using the multiple-client approach from Section 8.1.3. Imagine now that our Web server is trying to handle 1000 clients at once. Or 10,000. Will it work?

Let's make a list of what the so-called "C10K" ("clients 10,000") problems might be with our simple example:

- `select` has to look at up to 10,001 file descriptors. That's probably bigger than an `fd_set` can hold. Even if it can hold that number, it would take `select` a long time to process them all.
- There's a limit on how many file descriptors a process can have open, and it's usually much less than 10,001.
- To provide decent response, we'd want to engage multiple processes or multiple threads to handle all the work, but there's a limit on those resources, too.

- All the copying of data from files to sockets (in and out of the process's memory) is very time consuming. It would be a killer for 10,000 clients.
- Lots of other internal tables and other resources that our server might use start running out of space, or slowing way down, or both.

So, as you can see, the problems get pretty serious. This is one reason why a real Web server (or any other large-capacity server) is a hugely complex piece of software.

If you're trying to handle 10,000 clients, or even 500, there's a terrific article titled "The C10K Problem" that explains the problem and surveys most of the possible solutions [Keg2003].

Exercises

8.1. Modify the multiple-client example in Section 8.1.3 to work between computers (`AF_INET`). Run the server on one computer and run clients on at least one other computer.

8.2. Based on the information in [RFC1288], implement a simple version of the `finger` command, without options other than the `user@host` argument. Use connected sockets (`SOCK_STREAM`). For the next Exercise, you might need an option to set the port number.

8.3. Also based on [RFC1288], implement a simple `finger` server, but use an unused high-numbered port (e.g., 3079) instead of the standard port 79. Use connected sockets. Test your server with the `finger` command you wrote in the previous Exercise. If you have a machine you can play with, install your server on port 79 and test it with a standard `finger` command running on another machine.

8.4. Same as Exercise 8.2, but with connectionless sockets.

8.5. Same as Exercise 8.3, but with connectionless sockets.

8.6. Design and try to implement a simple communication example based on RFC 2549, which you can find at [RFC]. You may restrict yourself to the Concorde and First quality-of-service levels.

8.7. Implement a graphical Web browser using a GUI toolkit such as Qt. Warning: This is really hard! You may want to do this as a semester-long group project. Hint: Start your design with a way to handle nested tables. If you get that part right, everything else is relatively easy.

8.8. Implement the SMI with connectionless sockets, instead of with connected sockets as we did in Section 8.5. Run some timing tests like those in Section 7.15.

8.9. Implement a web crawler that starts with some URL and then scans what it finds there to discover additional URLs (e.g., by looking for strings that start with "href"). It adds the discovered URLs to a list, and then scans them, discovering even more URLs. It stops when it's found some specified number of URLs or when it's interrupted. (Obviously, it needs to continue when it encounters an error.) There should be a way to display the results—at least all the URLs discovered and whether they were scanned successfully (HTTP status of 200) or not and, if not, what the reason was. You *must* implement the Robot Exclusion Protocol (see *www.robotstxt.org* for details) so you don't crawl onto pages that you've been requested to stay away from. Provide a "unique host" option that crawls each host (the part of the URL up to the first slash) only once, using whatever URL was tried first for that host (e.g., *www.basepath.com/index.htm*). Using that option, try to find a starting URL that yields the most unique hosts that are successfully scanned. (I thought starting with *www.yahoo.com* would be a good idea, but I got exactly one hit because all the links there were relative to *www.yahoo.com*.) Be nice to others when you run your crawler by making sure that you're not hogging network resources.

8.10. Extend the program you wrote in Exercise 5.14 to include process attributes from Appendix A that are explained in this chapter.

This page intentionally left blank