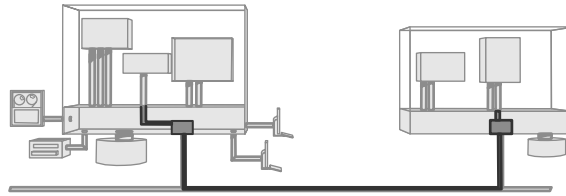


CHAPTER 11

Connecting to Processes Near and Far Servers and Sockets



OBJECTIVES

Ideas and Skills

- The client/server model
- Using pipes for two-way communication
- Coroutines
- The file/process similarity
- Sockets: Why, What, How?
- Network services
- Using sockets for client/server programs

System Calls and Functions

- `fdopen`
- `popen`
- `socket`
- `bind`
- `listen`
- `accept`
- `connect`

11.1 PRODUCTS AND SERVICES

Unix programmers use pipes to create digital assembly lines, the way manufacturers use conveyor belts to carry products from one worker to the next.

Not all businesses are factories, and some forms of communication are bidirectional. Consider dry cleaners, lawyers, and veterinarians. You drop off clothes at the cleaner, you send your pet to the vet, you mail documents to a lawyer, and, unlike a worker in the automobile factory who passes the car to the next worker, you expect to get something back. In these examples, we consider work done by the other person to be a *service*, and we consider ourselves *clients* for that service.

What does this have to do with Unix? Unix pipes carry data from one process to another. Processes and pipes can simulate not only an assembly line, producing finished goods, but also a service industry. In this chapter, we focus on interprocess data flow as the basis for *client/server* programming.

11.2 INTRODUCTORY METAPHOR: A BEVERAGE INTERFACE

Programs consume information. Some people consume soft drinks. Imagine a vending machine that dispenses cups of carbonated beverage, as shown in Figure 11.1. You insert a coin, push a button, and beverage emerges. What happens inside the dispenser? There might be a tank of carbonated water and a separate tank of drink concentrate, and pressing the button would activate a process to mix the raw materials and deliver dynamically generated beverage. On the other hand, there could be a single bottle of premixed beverage attached to a simple pump, and pressing the button simply transfers beverage to the cup.

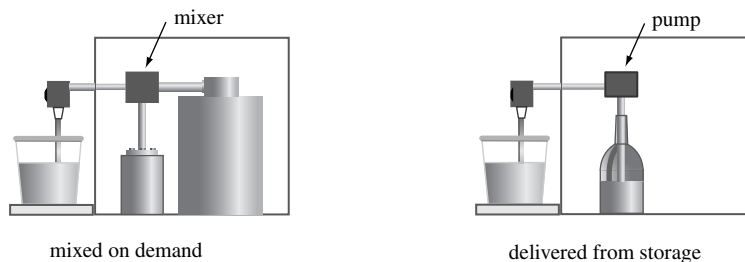


FIGURE 11.1
Dynamically generated or static beverage?

Unix, like the soda dispenser, presents one interface, even though data come from different types of sources (see Figure 11.2):

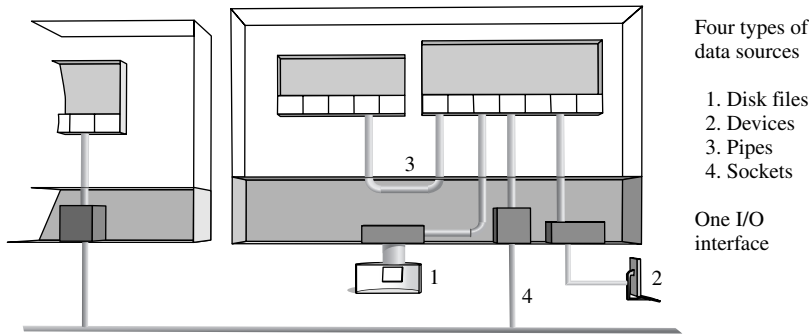


FIGURE 11.2

One interface, different sources.

(1,2) Disk/Device Files

Use `open` to connect, use `read` and `write` to transfer data.

(3) Pipes

Use `pipe` to create, use `fork` to share, and use `read` and `write` to transfer data.

(4) Sockets

Use `socket`, `listen`, and `connect` to connect, use `read` and `write` to transfer data.

Unix encapsulates in the file abstraction both the source and the means of production of data. In Chapter 2, we looked at reading data from files. In Chapter 5, we extended the idea of a file to include devices. Now we see how reading data from processes is similar to reading data from files.

11.3 bc: A UNIX CALCULATOR

Every version of Unix includes a version of the `bc` calculator. `bc` has variables, loops, and functions, and, as we saw in Chapter 1, `bc` handles very long numbers:

```
$ bc
17^123
22142024630120207359320573764236957523345603216987331732240497016947\
29282299663749675090635587202539117092799463206393818799003722068558\
0536286573569713
```

The trailing backslashes indicate continuation.

But `bc` Is Not a Calculator

A calculator program parses its input, performs the operations, and then prints the result. Most versions of `bc` parse the input but do not perform the operations.¹ Instead, `bc`

¹The GNU version of `bc` does the math, too.

runs the `dc` calculator program and communicates with it through pipes. `dc` is a stack-based calculator requiring the user to enter both values before specifying the operation. For example, the user writes `2 2 +` to add 2 and 2.

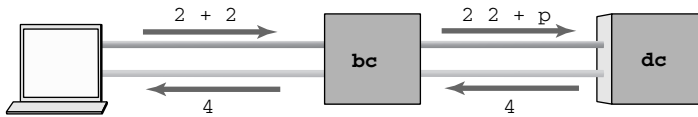


FIGURE 11.3

`bc` and `dc` as coroutines.

Figure 11.3 shows how `bc` processes `2+2`. The user types `2+2` then presses the Enter key. `bc` reads that expression from standard input, parses out the values and the operation, then sends the sequence of commands “2”, “2”, “+”, and “p” to `dc`, which stacks up the two values, applies the plus operation, and then prints to standard output the value on the top of the stack.

`bc` reads the result through the pipe it attached to the standard output of `dc` and then forwards that message to the user. `bc` does not even keep variables; if the user types `x=2+2`, then `bc` tells `dc` to do the math and store the result in register `x` in `dc`. The command `bc -c` shows what the parser sends to the calculator. Even the GNU version of `bc` converts user input into stack-based expressions.

Ideas from `bc`

1. Client/Server Model

The `bc/dc` pair of programs is an example of the client/server model of program design. `dc` provides a service: calculation. `dc` has a well-defined language, reverse Polish notation, and the two processes communicate through `stdin` and `stdout`.

`bc` provides a user interface *and* uses the services `dc` provides. `bc` is called a *client* of `dc`.

These two components are completely separate programs. You could replace the version of `dc`, and `bc` would still work. Similarly, you could create a nice graphical interface instead of `bc` and still use `dc` as the calculation engine. You could even replace `dc` with a program that parses the `dc` language and then passes the work to yet another program, perhaps on another, faster computer.

2. Bidirectional Communication

Unlike the assembly-line model of data processing, the client/server model often requires one process to communicate with both the standard input and the standard output of another process. Traditional Unix pipes carry data in one direction only.² Figure 11.3 shows two pipes from `bc` to `dc`. The top pipe carries calculator commands to the standard input of `dc`, and the bottom pipe carries the standard output of `dc` back to `bc`.

²Some pipes can carry data in two directions. (See exercise 11.11.)

3. Persistent Service

Unlike the shell we wrote, in which each user command creates a new process, the `bc` program keeps a single `dc` process running. `bc` uses that same instance of `dc` over and over again by sending it commands in response to each line of user input. This relationship differs from the standard *call-return* mechanism we use in function calls.

The `bc/dc` pair are called *coroutines* to distinguish them from *subroutines*. Both processes continue to run, but control passes from one to the other as each completes its part of the job. `bc` has the job of parsing and printing, and `dc` has the job of computing.

11.3.1 Coding `bc`: `pipe`, `fork`, `dup`, `exec`

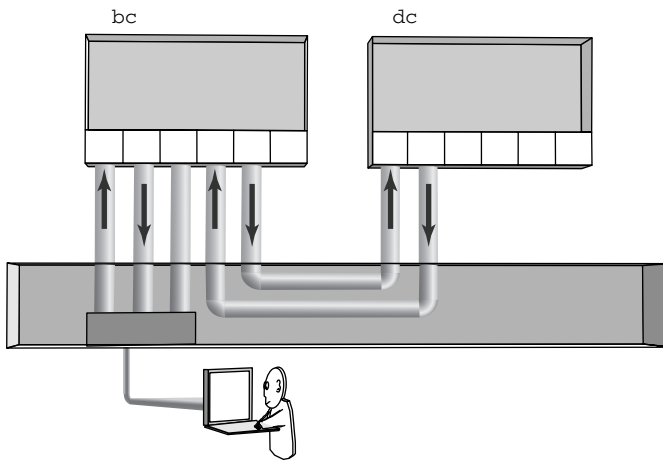


FIGURE 11.4
`bc`, `dc`, and kernel.

Figure 11.4 shows the data connections in the kernel that join the user to `bc` and `bc` to `dc`. We use the figure as a guide for building the code as follows:

- (a) Create two pipes.
- (b) Create a process to run `dc`.
- (c) In the new process, redirect `stdin` and `stdout` to the pipes, then `exec dc`.
- (d) In the parent, read and parse user input, write commands to `dc`, read response from `dc`, and send response to user.

Here is code for `tinybc.c`, a simple version of `bc` that uses `sscanf` to parse input and speaks with `dc` through two pipes:

```
/**      tinybc.c      * a tiny calculator that uses dc to do its work
**              * demonstrates bidirectional pipes
**              * input looks like number op number which
**              * tinybc converts into number \n number \n op \n p
```

```

**                                and passes result back to stdout
**
**
**                                +-----+
**    stdin  >0                    >== pipetodc  <====> |
**                                | tinybc      |      |
**                                |              |      |
**    stdout <1                    <== pipefromdc ==<  |
**                                +-----+
**
**
**                                * program outline
**                                a. get two pipes
**                                b. fork (get another process)
**                                c. in the dc-to-be process,
**                                    connect stdin and out to pipes
**                                    then execl dc
**                                d. in the tinybc-process, no plumbing to do
**                                    just talk to human via normal i/o
**                                    and send stuff via pipe
**                                e. then close pipe and dc dies
**                                * note: does not handle multiline answers
**/
#include <stdio.h>

#define oops(m,x)      { perror(m); exit(x); }

main()
{
    int      pid, todc[2], fromdc[2];          /* equipment */
    /* make two pipes */
    if ( pipe(todc) == -1 || pipe(fromdc) == -1 )
        oops("pipe failed", 1);

    /* get a process for user interface */
    if ( (pid = fork()) == -1 )
        oops("cannot fork", 2);
    if ( pid == 0 )                            /* child is dc */
        be_dc(todc, fromdc);
    else {
        be_bc(todc, fromdc);                  /* parent is ui */
        wait(NULL);                          /* wait for child */
    }
}

be_dc(int in[2], int out[2])
/*
    set up stdin and stdout, then execl dc
*/
{
    /* setup stdin from pipein */
    if ( dup2(in[0],0) == -1 )                /* copy read end to 0 */
        oops("dc: cannot redirect stdin",3);

```

```

        close(in[0]);                /* moved to fd 0          */
        close(in[1]);                /* won't write here    */

    /* setup stdout to pipeout */
    if ( dup2(out[1], 1) == -1 )     /* dupe write end to 1 */
        oops("dc: cannot redirect stdout",4);
    close(out[1]);                  /* moved to fd 1       */
    close(out[0]);                  /* won't read from here */

    /* now execl dc with the - option */
    execlp("dc", "dc", "-", NULL );
    oops("Cannot run dc", 5);
}

be_bc(int todc[2], int fromdc[2])
/*
 *   read from stdin and convert into to RPN, send down pipe
 *   then read from other pipe and print to user
 *   Uses fdopen() to convert a file descriptor to a stream
 */
{
    int      num1, num2;
    char      operation[BUFSIZ], message[BUFSIZ], *fgets();
    FILE      *fpout, *fpin, *fdopen();

    /* setup */
    close(todc[0]);                 /* won't read from pipe to dc */
    close(fromdc[1]);               /* won't write to pipe from dc */

    fpout = fdopen( todc[1], "w" ); /* convert file desc- */
    fpin  = fdopen( fromdc[0], "r" ); /* riptors to streams */
    if ( fpout == NULL || fpin == NULL )
        fatal("Error converting pipes to streams");

    /* main loop */
    while ( printf("tinybc: "), fgets(message, BUFSIZ, stdin) != NULL ){
        /* parse input */
        if ( sscanf(message, "%d%[-+*/^]%d", &num1, operation,
            &num2) != 3 ){
            printf("syntax error\n");
            continue;
        }

        if ( fprintf( fpout , "%d\n%d\n%c\np\n", num1, num2,
            *operation ) == EOF )
            fatal("Error writing");

        fflush( fpout );
        if ( fgets( message, BUFSIZ, fpin ) == NULL )
            break;
        printf("%d %c %d = %s", num1, *operation, num2, message);
    }
    fclose(fpout);                 /* close pipe          */
}

```

```

        fclose(fpin);                /* dc will see EOF      */
    }

    fatal( char mess[])
    {
        fprintf(stderr, "Error: %s\n", mess);
        exit(1);
    }

```

Here is `tinybc` in action:

```

$ cc tinybc.c -o tinybc ; ./tinybc
tinybc: 2+2
2 + 2 = 4
tinybc: 55^5
55 ^ 5 = 503284375
tinybc:

```

Look at this output carefully and identify which parts come from which programs. `tinybc` generates the prompt and the restatement of the arithmetic expression. The result of the computation is a string generated by `dc`; `tinybc` only reads that string from the pipe and includes it in the output.

11.3.2 Remarks on Coroutines

What other Unix tools can be used as coroutines? Can the `sort` utility be used as a coroutine for a program? No. `sort` reads all the data until end of file before it can generate output. The only way to send end of file through a pipe is to close the writing end. Once you close the writing end, though, you cannot send another lot of data to be sorted.

`dc`, on the other hand, processes data and commands line by line. Interaction with `dc` is simple and predictable. When you ask `dc` to print a value, you get back one line of text. When you tell `dc` to push a value, you get no response.

For a program to be part of a client-server coroutine system, the program must have a clear way of indicating the end of a message, and the program must use simple, predictable requests and replies.

11.3.3 fdopen: Making File Descriptors Look like Files

In `tinybc.c` we introduce the library function `fdopen`. `fdopen` works like `fopen`, returning a `FILE *`, but takes a file descriptor, not a filename, as an argument.

Use `fopen` to open something with a filename. `fopen` opens device files as well as regular disk files. Use `fdopen` when you have a file descriptor but no filename, as in the case of a pipe, and want to convert that connection into a `FILE *` so you can use standard, buffered I/O operations. Notice how `tinybc.c` uses `fprintf` and `fgets` to send data through the pipes to `dc`.

Using `fdopen` makes a remote process feel even more like a file. In the next section, we examine `popen`, a function that, by encapsulating calls to `pipe`, `fork`, `dup`, and `exec`, completes the illusion that programs and files are pretty much the same thing.

11.4 **popen: MAKING PROCESSES LOOK LIKE FILES**

In this section, we continue to study how a program can obtain services by connecting to another process. We examine the `popen` library function. We see what `popen` does and how `popen` works, and then we write our own version.

11.4.1 What `popen` Does

`fopen` opens a buffered connection to a file:

```
FILE *fp;                                /* a pointer to a struct */
fp = fopen( "file1", "r" );              /* args are filename, connection type */
c = getc(fp);                            /* read char by char */
fgets(buf, len, fp);                     /* line by line */
fscanf(fp, "%d%d%s", &x, &y, x);         /* token by token */
fclose(fp);                              /* close when done */
```

`fopen` takes two string arguments: the name of the file and the type of connection (e.g., “r”, “w”, “a”, ...). `popen` looks and works very much like `fopen`. `popen` opens a buffered connection to a process:

```
FILE *fp;                                /* same type of struct */
fp = popen("ls", "r");                   /* args are program name, connection type */
fgets(buf, len, fp);                     /* exactly the same functions */
pclose(fp);                              /* close when done */
```

Figure 11.5 shows similarities between `popen` and `fopen`. Both functions use the same syntax, and both functions return the same type of value. The first argument to `popen` is the name of the command to open; it can be any shell command. The second argument can be either “r” or “w”, but never “a”.

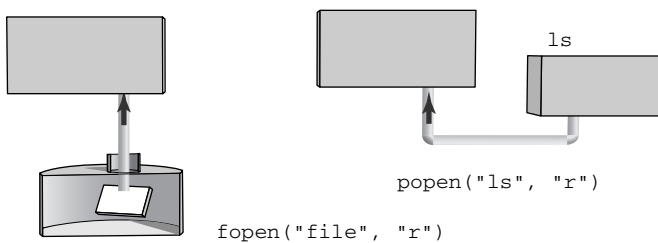


FIGURE 11.5

`fopen` and `popen`.

popen examples

The following program, in which the `who|sort` command is a source of data, uses `popen` to obtain a sorted list of current users:

```
/* popendemo.c
 * demonstrates how to open a program for standard i/o
 * important points:
 * 1. popen() returns a FILE *, just like fopen()
```

```

*           2. the FILE * it returns can be read/written
*           with all the standard functions
*           3. you need to use pclose() when done
*/
#include     <stdio.h>
#include     <stdlib.h>

int main()
{
    FILE     *fp;
    char     buf[100];
    int      i = 0;

    fp = popen( "who|sort", "r" );           /* open the command */
    while ( fgets( buf, 100, fp ) != NULL ) /* read from command */
        printf("%3d %s", i++, buf );        /* print data          */
    pclose( fp );                             /* IMPORTANT!          */
    return 0;
}

```

This second example uses `popen` to connect to the mail program and notify some users of system trouble:

```

/* popen_ex3.c
*   shows how to use popen to write to a process that
*   reads from stdin. This program writes email to
*   two users. Note how easy it is to use fprintf
*   to format the data to send.
*/
#include     <stdio.h>

main()
{
    FILE     *fp;

    fp = popen( "mail admin backup", "w" );
    fprintf( fp, "Error with backup!!\n" );
    pclose( fp );
}

```

pclose is Required

When you are done reading from or writing to the connection created by `popen`, use `pclose`, not `fclose`. A process needs to be waited for, or it becomes a zombie. `pclose` calls `wait`.

11.4.2 Writing popen: Using fdopen

How does `popen` work, and how do we write it? `popen` runs a program and returns a connection to the standard input or standard output of that program.

We need a new process to run the program, so we use `fork`. We need a connection to that process, so we use `pipe`. We need to make a file descriptor into a buffered stream, so we use `fdopen`. Finally, we need to be able to run any shell command in that process, so we use `exec`, but what do we execute?. The only program that can run any shell command is the shell itself: `/bin/sh`. Conveniently, `sh` supports a `-c` option that tells the shell to run a command and then exit. For example,

```
sh -c "who|sort"
```

tells `sh` to run the single command line `who|sort`. (See also Figure 11.6.)

We combine `pipe`, `fork`, `dup2`, and `exec` as shown in this flowchart:

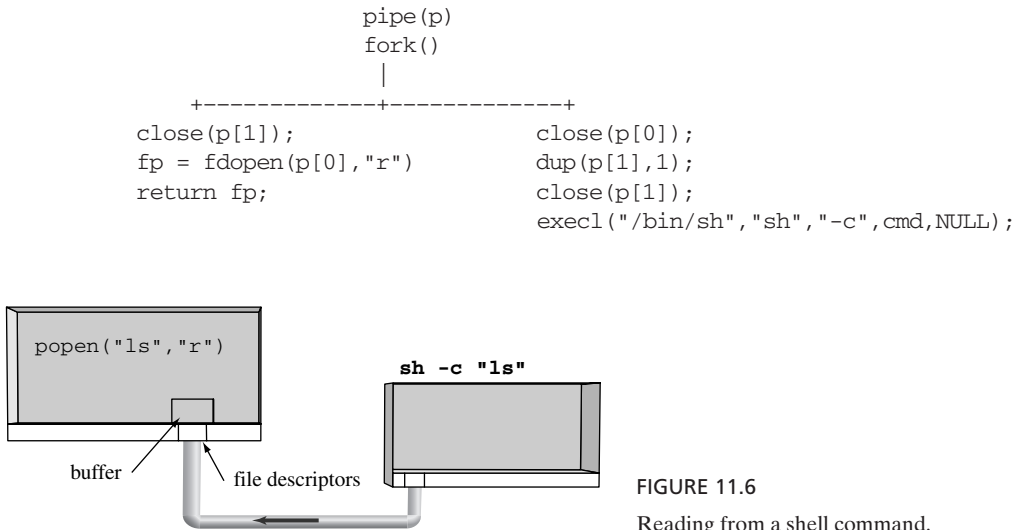


FIGURE 11.6
Reading from a shell command.

An implementation of that flowchart is `popen.c`:

```

/* popen.c - a version of the Unix popen() library function
 *
 * FILE *popen( char *command, char *mode )
 *
 *     command is a regular shell command
 *     mode is "r" or "w"
 *
 *     returns a stream attached to the command, or NULL
 *     execls "sh" "-c" command
 *
 *     todo: what about signal handling for child process?
 */
#include <stdio.h>
#include <signal.h>

#define READ 0
#define WRITE 1

FILE *popen(const char *command, const char *mode)
{
    int    pfp[2], pid;                /* the pipe and the process */

```

```

FILE    *fdopen(), *fp;           /* fdopen makes a fd a stream */
int      parent_end, child_end;   /* of pipe */

if ( *mode == 'r' ){              /* figure out direction */
    parent_end = READ;
    child_end = WRITE ;
} else if ( *mode == 'w' ){
    parent_end = WRITE;
    child_end = READ ;
} else return NULL ;

if ( pipe(pfp) == -1 )             /* get a pipe */
    return NULL;
if ( (pid = fork()) == -1 ){       /* and a process */
    close(pfp[0]);                 /* or dispose of pipe */
    close(pfp[1]);
    return NULL;
}

/* ----- parent code here ----- */
/* need to close one end and fdopen other end */

if ( pid > 0 ){
    if (close( pfp[child_end] ) == -1 )
        return NULL;
    return fdopen( pfp[parent_end] , mode ); /* same mode */
}

/* ----- child code here ----- */
/* need to redirect stdin or stdout then exec the cmd */

if ( close(pfp[parent_end]) == -1 ) /* close the other end */
    exit(1);                         /* do NOT return */

if ( dup2(pfp[child_end], child_end) == -1 )
    exit(1);

if ( close(pfp[child_end]) == -1 )   /* done with this one */
    exit(1);

/* all set to run cmd */
execl( "/bin/sh", "sh", "-c", command, NULL );
exit(1);
}

```

This version of `popen` does not do anything about signals. Is that a problem?

11.4.3 Access to Data: Files, APIs, and Servers

`fopen` gets data from a file, and `popen` gets data from a process. Let us focus on the general question of getting data and compare three methods. As an example, we compare three methods for getting the list of people logged on to a system.

Method 1: Getting Data from Files You can get data by reading from a file. In Chapter 2, we wrote a version of `who` that reads the list of current users from the `utmp` file.

File-based information services are not perfect. Client programs rely on a particular file format and specific member names in structures. The lines

```
/* Backwards compatibility hacks. */
#define ut_name      ut_user
```

in the Linux header file for the `utmp` structure show what happens.

Method 2: Getting Data from Functions You can get data by calling a function. A library function hides data formats and file locations behind a standard function interface. Unix provides a function interface to the `utmp` file. The manual page for `getutent` describes functions that read the `utmp` database. The underlying storage structure may change, but programs that use the interface still work.

Application programming interface (API)-based information services are not always the right solution, either. There are two methods for using system library functions. A program might use *static linking* and include the actual function code. Those functions may use filenames or file formats that are no longer correct. On the other hand, a program might call functions in *shared libraries*, but these libraries are not always installed on a system, or the version on a system may not match the version the program needs.

Method 3: Getting Data from Processes A third method is to get data by reading from a process. The `bc/dc` and `popen` examples showed how to create a connection to another process. A program that wants the list of users can call `popen` to connect to the `who` program. The `who` command, not your program, is responsible for knowing the correct filename and file format and for using the appropriate libraries.

Calling separate programs for data provides other benefits. Server programs can be written in any language: shell scripts, compiled C code, Java, Perl. The most dramatic benefit of implementing system services as separate programs is that the client program can run on one machine and the server program can run on a different machine. All we need is some way of connecting to a process on a different computer.

11.5 SOCKETS: CONNECTING TO REMOTE PROCESSES

Pipes allow processes to send data to other processes as easily as they send data to files, but pipes have two significant limitations. A pipe is created in one process and is shared by calling `fork`. Therefore, pipes can only connect related processes, and pipes can only connect processes on the same computer. Unix provides another method of inter-process communication—*sockets*:

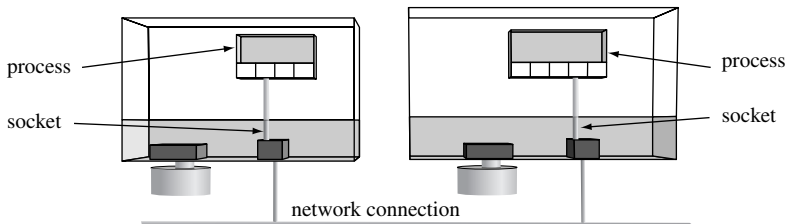


FIGURE 11.7

Connecting to a remote process.

Sockets allow processes to create pipeline connections to unrelated processes and even to processes on other computers. (See Figure 11.7.) In this section, we study the basic ideas of sockets and see how to use sockets to connect clients and servers on different computers. The idea is as simple as calling a telephone number to get the time of day.

11.5.1 An Analogy: “At the Tone, the Time Will Be...”

Many cities have a time telephone number. You dial that number, and the machine that picks up the call tells you the time in that city. How does it work? What if you wanted to set up your own time service? You could use the low-tech solution depicted in Figure 11.8. In the figure, that’s you on the right in the office. You are the server providing the time service. You put a clock on the wall. The steps you follow to set up your time service match exactly the steps a Unix program follows to set up a socket-based service. Therefore, we describe these steps carefully.

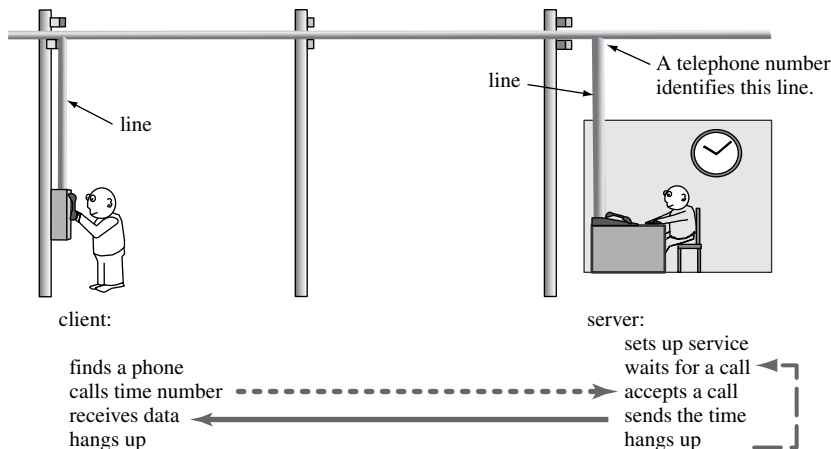


FIGURE 11.8

A time service.

Setting Up and Operating the Service

How do you set up and operate a time service once you buy and install your clock?

Setting Up the Service Setting up your service consists of three steps:

1. Get a phone line

First, you need to have a line run from the telephone network to a jack on the wall by your desk. This wire and the jack allow you to connect to the network so that calls can be routed to your desk. In fancier language, the jack is an *endpoint of communication*. The next time you have a phone line installed in your home, tell the company or electrician that you need an endpoint of communication.

2. *Get a phone number for that line*

Clients need a number to call to reach your endpoint of communication. The telephone network identifies each wall jack with a telephone number. For the purposes of this analogy, it is important to imagine that you are in a large business that offers several services in addition to your time service. Therefore, your jack is identified by a telephone number *and* an extension number.

For example, your number might be 617-999-1234, extension 8080. The telephone number identifies the building that contains your office, and the extension (8080) identifies your particular telephone in that building. Got that part? One number for the building, a second number for your service. This is important.

3. *Arrange for incoming calls*

You may have used pay telephones marked *no incoming calls*. Your service does not want that type of telephone. You tell the telephone network your line should accept incoming calls. You might set up a queue for incoming calls. You could have a message that tells callers how important their calls are to you and then plays music. The queue idea applies to sockets, the music does not.

Operating the Service Operating the time service consists of a loop with the following three steps:

4. *Wait for a call*

Sit there doing nothing until a call comes in. In technical terms, you *block* on a call. When a call arrives, you unblock and accept the call.

5. *Provide the service*

In your case, you look at the clock, then you send that number down the wire by speaking.

6. *Hang up*

Your work is done for this call, so you hang up.

Those six steps, three steps of setup and three steps per call, are the details of running a time service over a telephone network.

Using the Service

How does a client use your service? A client follows these four steps:

1. *Get a phone line*

The client also needs an endpoint of communication. The client orders a phone line from the telephone network.

2. *Connect to your number*

The client now uses the line to request a connection through the telephone network to your line. The client connects to the business number and extension that identifies your service. The combination of business number and extension is called the *network address* of your service. In technical terms, the telephone number of

the business is the *host address*, and your extension is the *port number* or just *port*. In the preceding example, the host address is 617-999-1234, and the port is 8080.

3. *Use the service*

The two endpoints of communication (the client's and the server's) are now connected. Either party may send data through this connection to the other endpoint. In the case of a time service, the server sends data through the connection, and the client receives the information. A more complicated service, such as a catalog-order line, requires a more complicated interaction between client and server. We explore more complex services later.

4. *Hang up*

The interaction is complete. The client hangs up.

Important Concepts

The time-server example includes four concepts we use in socket programming:

client and server

We discussed these ideas several times. The server is the program that provides a service. A server, in Unix terms, is a program, not a computer. Typical names for a computer are computer, host, system, machine, and box. A server process waits for a request, processes the request, then loops back to take the next request. A client process, on the other hand, does not loop. A client makes a connection, exchanges some data with the server, and then continues.

hostname and port

A server on the Internet is a process running on some computer. The computer is called the *host*. Machines often are assigned names like *sales.xyzcorp.com*; this is called the *hostname* of the machine. The server has a port number on that host. The combination of host and port identifies a server.

address family

Your time service has a telephone number. It might also have a street address and zip code. It also has a longitude and latitude, another set of numbers. Each of these sets of numbers is an address for your service. It would not work, though, to use your longitude and latitude as telephone number and extension.

Each of these addresses belongs to an *address family*. The telephone number and extension have a meaning inside the telephone-network address family, which we could symbolize as `AF_PHONE`. Similarly, the longitude and latitude make sense in the global-coordinate-system address family, which we could symbolize as `AF_GLOBAL`.

protocol

A *protocol* is the rules of interaction between the client and the server. In the time service, the protocol is simple: the client calls, the server answers, the server states the time, and the server hangs up.

What if you ran a directory-assistance service instead? The protocol would be a little more complicated. You, the server, would answer and send an initial greeting (“*Phoneco directory assistance. What city?*”). The client would respond with the name of a city. The server then asks for a name (“*What listing?*”). The client responds with the name of a person or business. The server then sends back the telephone number or a message that no such listing exists in that city. Some directory-assistance servers will offer, for a fee, to dial the number for you. That exchange of messages follows the *directory-assistance protocol*, known as DAP to readers of this paragraph. Every client/server system must define a protocol.

11.5.2 Internet Time, DAP, and Weather Servers

The dial-the-time and directory-assistance server examples are more than pedagogical metaphors; they have exact equivalents on the Internet. Try this:

```
$ telnet mit.edu 13
Trying 18.7.21.69...
Connected to mit.edu.
Escape character is '^]'.
Mon Aug 13 22:36:44 2001
Connection closed by foreign host.
$
```

Somewhere on a machine at MIT is a time server waiting to take calls on port 13. When we call that server using the `telnet` program, the server picks up the call, checks the clock on the system, sends the current time back over the wire, and then hangs up. Exactly like your dial-the-time service. It even uses the same simple protocol. Try connecting to port 13 on other hosts. You can find out what time it is on machines around the world.

The `telnet` program is like a telephone. It makes a connection to a port on a remote host, and then it transfers bytes from your keyboard to the connection and from the connection to your display.

What about directory assistance? The directory-assistance server usually listens for calls on port 79. For example, we might have

```
$ telnet princeton.edu 79
Trying 128.112.128.81...
Connected to princeton.edu.
Escape character is '^]'.
smith
-----
alias: 000012345
name: Waldo Smith
department: Special Student
email: waldos@Princeton.EDU
emailbox: waldos@mail.Princeton.EDU
netid: waldos
-----
```

```

alias: 000333333
name: Ignatz E Smith
department: Undergraduate Class of 1997
email: ismith@Princeton.EDU
emailbox: ismith@mail.Princeton.EDU
netid: ismith
.
.
.
```

When a call comes in, the server picks up the call. The protocol specifies that the client type a name and then press the return key. The server sends back all matching entries, then hangs up.

What about the weather? Try the following:

```
telnet rainmaker.wunderground.com 3000
```

The protocol for this weather server is more complicated but friendlier.

11.5.3 Lists of Services: Well-Known Ports

How did I know to use port 13 for the time and port 79 for directory assistance? The same way people in the United States know to dial 911 for emergencies and 411 for directory assistance—these are *well-known ports*. The file `/etc/services` is a list of well-known services and their port numbers:

```

$ more /etc/services
#      $NetBSD: services,v 1.18 1996/03/26 00:07:58 mrg Exp $
#
# Network services, Internet style
#
# Note that it is presently the policy of IANA to assign a single well-known
# port number for both TCP and UDP; hence, most entries here have two entries
# even if the protocol doesn't support UDP operations.
# Updated from RFC 1340, "Assigned Numbers" (July 1992).  Not all ports
# are included, only the more common ones.
#
#      from: @(#)services      5.8 (Berkeley) 5/9/91
#
tcpmux      1/tcp          # TCP port service multiplexer
echo        7/tcp
echo        7/udp
discard     9/tcp          sink null
discard     9/udp          sink null
systat      11/tcp         users
daytime     13/tcp
daytime     13/udp
--More-- (13%)
```

You can see in this listing that the *daytime* service uses port 13. Explore this file to see the standard services on Internet machines. Look for the *ftp*, *telnet*, *finger*, and *http* entries.

All these servers running on Internet hosts are based on the ideas and steps we saw in the telephone-based time service. We now translate these ideas into Unix system calls to write our own version of the time server and time client.

11.5.4 Writing **timeserv.c**: A Time Server

Our telephone-based time server involved six steps. Each step corresponds to a system call. This table shows the translation:

action	syscall
1. Get a phone line	socket
2. Assign a number	bind
3. Allow incoming calls	listen
4. Wait for a call	accept
5. Transfer data	read/write
6. Hang up	close

Here is the code:

```
/* timeserv.c - a socket-based time of day server
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <strings.h>
#define PORTNUM 13000 /* our time service phone number */
#define HOSTLEN 256
#define oops(msg)      { perror(msg) ; exit(1) ; }

int main(int ac, char *av[])
{
    struct sockaddr_in  saddr; /* build our address here */
    struct hostent      *hp;   /* this is part of our   */
    char                hostname[HOSTLEN]; /* address             */
    int                 sock_id,sock_fd; /* line id, file desc   */
    FILE                *sock_fp; /* use socket as stream */
    char                *ctime(); /* convert secs to string */
    time_t              thetime; /* the time we report   */

    /*
     * Step 1: ask kernel for a socket
     */
}
```

```

sock_id = socket( PF_INET, SOCK_STREAM, 0 );    /* get a socket */
if ( sock_id == -1 )
    oops( "socket" );

/*
 * Step 2: bind address to socket.  Address is host,port
 */

bzero( (void *)&saddr, sizeof(saddr) ); /* clear out struct */

gethostname( hostname, HOSTLEN );           /* where am I ? */
hp = gethostbyname( hostname );              /* get info about host */
                                           /* fill in host part */
bcopy( (void *)hp->h_addr, (void *)&saddr.sin_addr, hp->h_length);
saddr.sin_port = htons(PORTNUM);            /* fill in socket port */
saddr.sin_family = AF_INET ;                /* fill in addr family */

if ( bind(sock_id, (struct sockaddr *)&saddr, sizeof(saddr)) != 0 )
    oops( "bind" );

/*
 * Step 3: allow incoming calls with Qsize=1 on socket
 */

if ( listen(sock_id, 1) != 0 )
    oops( "listen" );

/*
 * main loop: accept(), write(), close()
 */

while ( 1 ){
    sock_fd = accept(sock_id, NULL, NULL); /* wait for call */
    printf("Wow! got a call!\n");
    if ( sock_fd == -1 )
        oops( "accept" );                /* error getting calls */

    sock_fp = fdopen(sock_fd,"w"); /* we'll write to the */
    if ( sock_fp == NULL )              /* socket as a stream */
        oops( "fdopen" );              /* unless we can't */

    thetime = time(NULL);                /* get time */
                                           /* and convert to string */
    fprintf( sock_fp, "The time here is .." );
    fprintf( sock_fp, "%s", ctime(&thetime) );
    fclose( sock_fp );                  /* release connection */
}
}

```

And here is an explanation of how the program works:

Step 1: Ask kernel for a socket

A *socket* is an endpoint of communication. Like the telephone jack at the wall, a socket is a place from which calls can be made and a place to which calls can be directed. The `socket` system call creates a socket:

socket		
PURPOSE	Create a socket	
INCLUDE	<pre>#include <sys/types.h> #include <sys/socket.h></pre>	
USAGE	<code>sockid = socket(int domain, int type, int protocol)</code>	
ARGS	domain	communication domain. PF_INET is for Internet sockets
	type	type of socket. SOCK_STREAM looks like a pipe
	protocol	protocol used within the socket. 0 is default.
RETURNS	-1	if error
	sockid	a socket id if successful

`socket` creates an endpoint for communication and returns an identifier for that socket. There are various sorts of communication systems, each of which is called a *domain* of communication. The Internet is one domain. We shall see later that the Unix kernel is another domain. Linux supports communication in several other domains.

The type of a socket specifies the type of data flow the program plans to use. The `SOCK_STREAM` type works like a bidirectional pipe. Data written in one end can be read from the other end as a continuous sequence of bytes. We examine `SOCK_DGRAM` in a later chapter.

The last argument, *protocol*, refers to the protocol used within the network code in the kernel, not the protocol between the client and server. A value of 0 selects the standard protocol.

Step 2: Bind address to socket. Address is host, port

The next step is to assign a network address to our socket. In the Internet domain, an address consists of a host and a port number. We cannot use port 13; that is reserved for the *real* time server. Instead, we use port 13000. You can select any port number you like for a server, as long as it is not too low and not already in use. Low-numbered ports may be used only by system services, not by regular users. Check your system for the restricted range. Port numbers are 16-bit quantities, so there are a lot of them. `bind` is summarized as follows:

bind		
PURPOSE	Bind an address to a socket	
INCLUDE	#include <sys/types.h> #include <sys/socket.h>	
USAGE	result = bind(int sockid, struct sockaddr *addrp, socklen_t addrlen)	
ARGS	sockid	the id of the socket
	addrp	a pointer to a struct containing the address
	addrlen	the length of the struct
RETURNS	-1	if error
	0	if success

`bind` assigns an address to a socket. This address works like the telephone number assigned to the jack on the wall in your office; other processes use that address when they want to connect to your server. Each address family has its own format. The Internet address family (`AF_INET`) uses host and port. An address is a struct with the host and port number as members. Our program first zeros the struct, then fills in the host address, then fills in the port number, and, finally, fills in the address family. Consult the manual pages for the functions used to construct each of these numbers.

When all those parts are filled in, the address is attached to the socket. Other types of sockets use addresses with different members.

Step 3: Allow incoming calls with queue size=1 on socket

A server accepts incoming calls, so our program must call `listen`:

listen		
PURPOSE	Listen for connections on a socket	
INCLUDE	#include <sys/socket.h>	
USAGE	result = listen(int sockid, int qsize)	
ARGS	sockid	socket that will accept calls
	qsize	backlog of incoming connections
RETURNS	-1	if error
	0	if success

`listen` asks the kernel to allow the specified socket to receive incoming calls. Not all types of sockets can receive incoming calls. `SOCK_STREAM` can. The second argument specifies the size of the queue for incoming calls. In our code, we request a queue of one call. The maximum queue size depends on the socket implementation.

Step 4: Wait For/Accept a Call

Once the socket is created, assigned an address, and set up to receive incoming calls, the program is ready to begin its work. The server now waits until a call comes in. It uses `accept`:

accept		
PURPOSE	Accept a connection on a socket	
INCLUDE	#include <sys/types.h> #include <sys/socket.h>	
USAGE	fd = accept(int sockid, struct sockaddr *callerid, socklen_t *addrlenp)	
ARGS	sockid	accept a call on this socket
	callerid	pointer to struct for address of caller
	addrlenp	pointer to storage for length of address of caller
RETURNS	-1	if error
	fd	a file descriptor open for reading and writing

`accept` suspends the current process until an incoming connection on the specified socket is established. `accept` returns a file descriptor opened for reading and writing. That file descriptor is a connection to a file descriptor in the calling process.

The `accept` call supports a form of *caller ID*. The socket in the caller has an address. For Internet connections, the address is a host and port number. If the *callerid* and *addrlenp* pointers are not null, the kernel puts the address of the caller into the struct pointed to by *callerid* and puts the length of that struct into the value pointed to by *addrlenp*.

Just as a human can use caller-ID information to decide how to handle an incoming call, a network program can use the address of the calling process to decide how to handle an incoming connection.

Step 5: Transfer Data

The file descriptor returned by `accept` is a regular file descriptor, the sort of thing we have been using since we learned about `open` back in Chapter 2. In `timeserv.c`, we use `fopen` to make this file descriptor into a buffered data stream so we can use `fprintf`. We could have used plain old `write`.

Step 6: Close Connection

The file descriptor returned by `accept` may be closed with the standard `close` system call. When one process closes its end of the socket, the process on the other end will see an end-of-file result if it tries to read data. Pipes work the same way.

11.5.5 Testing `timeserv.c`

We can now compile and run our time server:

```
$ cc timeserv.c -o timeserv
$ timeserv&
29362
$
```

We started our server with a trailing ampersand, so the shell runs it but does not call wait. The server is blocked at the `accept` system call. We can connect to it with `telnet`:

```
$ telnet `hostname` 13000
Trying 123.123.123.123
Connected to somesite.net
Escape character is '^]'.
Wow! got a call!
The time here is ..Tue Aug 14 11:36:30 2001
Connection closed by foreign host.
$
$ telnet `hostname` 13000
Trying 123.123.123.123
Connected to somesite.net
Escape character is '^]'.
Wow! got a call!
The time here is ..Tue Aug 14 11:36:53 2001
Connection closed by foreign host.
$
```

We have made two connections, and the server responded both times. The server will continue to run until we kill it:

```
$ kill 29362
```

`telnet` works as a client for this server, but it is not always a suitable way to connect to a server. We now write a special client for this server.

11.5.6 Writing `timeclnt.c`: A Time Client

Our telephone-based time client uses four steps, each corresponding to a system call:

action	syscall
1. Get a phone line	<code>socket</code>
2. Call the server	<code>connect</code>
3. Transfer data	<code>read/write</code>
4. Hang up	<code>close</code>

Here is the code:

```

/* timeclnt.c - a client for timeserv.c
 *          usage: timeclnt hostname portnumber
 */
#include      <stdio.h>
#include      <sys/types.h>
#include      <sys/socket.h>
#include      <netinet/in.h>
#include      <netdb.h>

#define      oops(msg)      { perror(msg); exit(1); }

main(int ac, char *av[])
{
    struct sockaddr_in  servadd;          /* the number to call */
    struct hostent      *hp;              /* used to get number */
    int      sock_id, sock_fd;            /* the socket and fd */
    char      message[BUFSIZ];            /* to receive message */
    int      messlen;                     /* for message length */

    /*
     * Step 1: Get a socket
     */

    sock_id = socket( AF_INET, SOCK_STREAM, 0 );    /* get a line */
    if ( sock_id == -1 )
        oops( "socket" );                          /* or fail */

    /*
     * Step 2: connect to server
     *          need to build address (host,port) of server first
     */

    bzero( &servadd, sizeof( servadd ) );    /* zero the address */
    hp = gethostbyname( av[1] );              /* lookup host's ip # */
    if (hp == NULL)
        oops(av[1]);                          /* or die */
    bcopy(hp->h_addr, (struct sockaddr *)&servadd.sin_addr, hp->h_length);
    servadd.sin_port = htons(atoi(av[2]));    /* fill in port number */
    servadd.sin_family = AF_INET ;             /* fill in socket type */

                                           /* now dial */
    if ( connect(sock_id,(struct sockaddr *)&servadd, sizeof(servadd)) !=0)
        oops( "connect" );

    /*
     * Step 3: transfer data from server, then hangup
     */

    messlen = read(sock_id, message, BUFSIZ);    /* read stuff */
    if ( messlen == - 1 )
        oops("read") ;

```

```

        if ( write( 1, message, messlen ) != messlen ) /* and write to */
            oops( "write" );                          /* stdout      */
        close( sock_id );
    }

```

And here is an explanation of how the program works:

Step 1: Ask Kernel for a Socket

The client needs a socket to connect to the network, just as a client for your telephone time service needs a phone line to connect to the phone network. The socket for the client also has to be an Internet socket (`AF_INET`) and has to be a stream socket (`SOCK_STREAM`).

Step 2: Connect to Server

The client connects to the time server. The `connect` system call is the network equivalent of making a telephone call.

connect		
PURPOSE	Connect to a socket	
INCLUDE	#include <sys/types.h> #include <sys/socket.h>	
USAGE	result = connect(int sockfd, struct sockaddr *serv_addrp, socklen_t addrlen);	
ARGS	sockfd	socket to use for connection
	serv_addrp	pointer to struct containing server address
	addrlen	length of that struct
RETURNS	-1	if error
	0	if success

`connect` attempts to connect the socket specified by *sockid* to the socket identified by the socket address pointed to by *serv_addrp*. If the attempt succeeds, `connect` returns 0. In that case, the *sockid* is now a valid file descriptor open for reading and writing. Data written into this file descriptor are sent to the socket at the other end of the connection, and data written into the other end may be read from this file descriptor.

Steps 3 and 4: Transfer Data and Then Hang Up

After a successful `connect`, a process may read and write data from this file descriptor as though it were connected to a regular file or pipe. In the time client/server system, `timeclnt` simply reads one line from the server.

After reading the time, the client `closes` the file descriptor and exits. The client could have just exited, and the kernel would have closed this open file descriptor.

11.5.7 Testing `timeclnt.c`

We have not seen any pictures for several pages, so you may have forgotten what all this code is supposed to do. A look at Figure 11.9 will remind you. The server process runs on one computer. A client process on another computer connects to the server over the network. The server sends data to the client by calling `write`. The client receives that message by calling `read`.

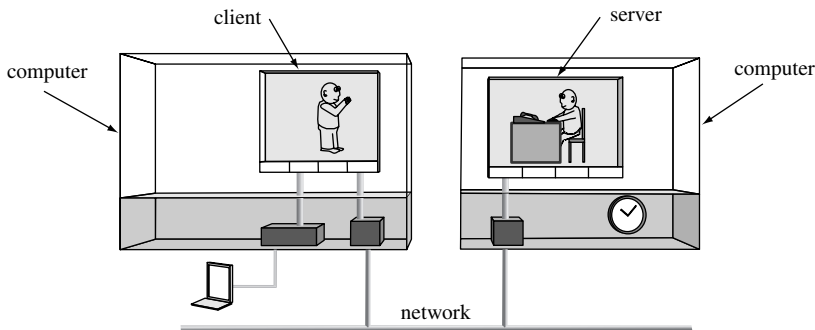


FIGURE 11.9

Processes on different computers.

A real test of our software involves running the two programs on different computers. I am not sure how clear this looks in a book, but here goes:

```
$ hostname                                # check current machine
computer1.mysite.net                     # the first machine
$ cc timeserv.c -o timeserv               # build server
$ ./timeserv &                           # and run it
[1] 10739
$
$ scp timeclnt.c bruce@computer2: # send client code elsewhere
bruce@computers2's password:
timeclnt.c                               |      1 KB |   1.8 kB/s | ETA: 00:00:00 | 100%
$ ssh bruce@computer2
bruce@computer2's password:
No mail.
computer2:bruce$ cc timeclnt.c -o timeclnt
computer2:bruce$ ./timeclnt computer1 13000Wow! got a call!
The time here is ..Tue Aug 14 02:44:31 2001
computer2:bruce$
```

The server is compiled and set running on `computer1`. I then copy the client code to `computer2` and log in to `computer2`. On `computer2`, I compile the client and ask it to connect to the server running on `computer1` listening at port 13000. The message I see

is sent over the network from the server on `computer1` to the client on `computer2`. That client sends the message to standard output.

Am I really seeing the output on `computer2`? I am connected to `computer2` from `computer1`, so the terminal on which the message appears is actually attached to `computer1`. See the exercise that asks you to think about what is *really* going on.

These `timeserv`/`timeclnt` programs let us see what time it is on another computer. Checking the time on another computer also allows computers to keep their clocks synchronized. One machine on a network can serve as the authority on time. Other machines can use this sort of client/server system to reset their clocks periodically.

11.5.8 Another Server: `remote ls`

Our next project is to write a program that lists files on a remote computer. You may have accounts on two systems. What if you wanted to list files you have on the other machine? You could log in to the other machine and run `ls`. A quicker, more convenient method would be a *remote ls* program, we can call it `rls`. You would specify a hostname and a directory:

```
$ rls computer2.site.net /home/me/code
```

Of course, `rls` needs a server process running on the other machine to receive the request, do the work, and return the result. The system looks like that shown in Figure 11.10. A server runs on one computer. A client on another computer connects to the server and sends the name of a directory. The server sends back to the client a list of the files in that directory. The client displays the list by writing to standard output. This two-process system provides access to directories on a different computer.

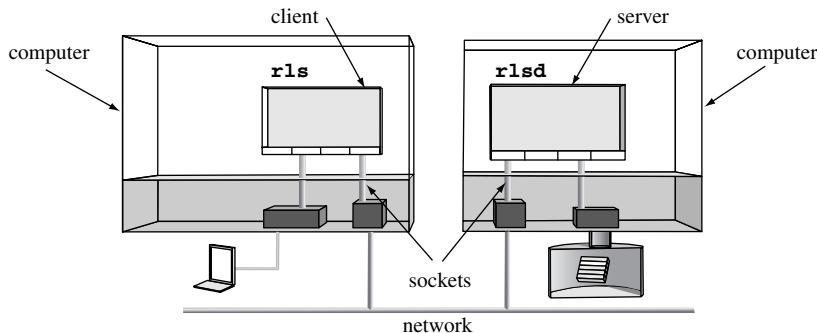


FIGURE 11.10

A *remote ls* system.

Planning the `remote ls` System

We need three things to implement the `rls` system:

- (a) a protocol
- (b) a client program
- (c) a server program

The Protocol

The protocol consists of a request and a reply. First, the client sends a single line containing the name of a directory. The server reads that line. Next, the server opens and reads that directory and sends back to the client the list of files. The client reads the list of files, line by line, until the server closes the connection, which generates an end-of-file condition.

The Client: `rls`

```
/* rls.c - a client for a remote directory listing service
 *          usage: rls hostname directory
 */
#include      <stdio.h>
#include      <sys/types.h>
#include      <sys/socket.h>
#include      <netinet/in.h>
#include      <netdb.h>

#define      oops(msg)          { perror(msg); exit(1); }
#define      PORTNUM           15000

main(int ac, char *av[])
{
    struct sockaddr_in  servadd;          /* the number to call */
    struct hostent      *hp;              /* used to get number */
    int                 sock_id, sock_fd; /* the socket and fd */
    char                buffer[BUFSIZ];   /* to receive message */
    int                 n_read;           /* for message length */

    if ( ac != 3 ) exit(1);

    /** Step 1: Get a socket **/

    sock_id = socket( AF_INET, SOCK_STREAM, 0 ); /* get a line */
    if ( sock_id == -1 )
        oops( "socket" ); /* or fail */

    /** Step 2: connect to server **/
    bzero( &servadd, sizeof(servadd) ); /* zero the address */
    hp = gethostbyname( av[1] ); /* lookup host's ip # */
    if ( hp == NULL )
        oops(av[1]); /* or die */
    bcopy(hp->h_addr, (struct sockaddr *)&servadd.sin_addr, hp->h_length);

    servadd.sin_port = htons(PORTNUM); /* fill in port number */
    servadd.sin_family = AF_INET ; /* fill in socket type */

    if ( connect(sock_id, (struct sockaddr *)&servadd, sizeof(servadd)) !=0 )
        oops( "connect" );

    /** Step 3: send directory name, then read back results **/
```

```

if ( write(sock_id, av[2], strlen(av[2])) == -1)
    oops("write");
if ( write(sock_id, "\n", 1) == -1 )
    oops("write");

while( (n_read = read(sock_id, buffer, BUFSIZ)) > 0 )
    if ( write(1, buffer, n_read) == -1 )
        oops("write");
close( sock_id );
}

```

Note the differences between this client and the `time` client. The `rls` client first writes the directory name into the socket. Our protocol states that the client sends a line, so we append a newline character. Next, the client enters a loop, copying data from the socket to standard output until end of file. `rls.c` uses low-level `write` and `read` to transfer data to and from the server. The loop uses a standard buffer size to be efficient. Next, we write the server.

The Server: `rlsd`

The server has to get a socket, bind, listen, and then accept a call. After accepting a call, the server reads the name of a directory from the socket and then lists the contents of that directory. How does the server list a directory? We could copy our version of `ls` from Chapter 3, but we can use a simpler method: Just use `popen` to read the output from the regular version of `ls`. (See Figure 11.11.)

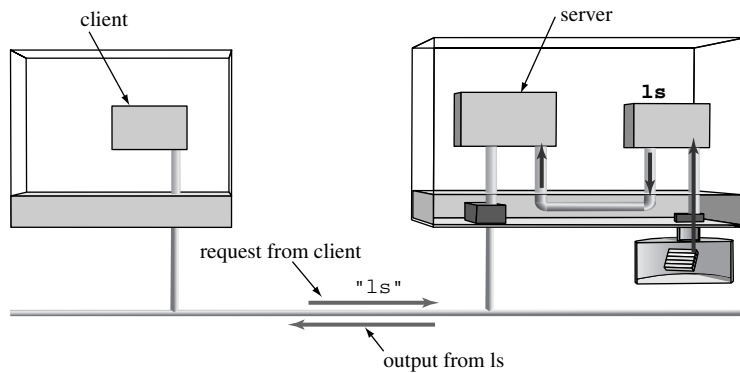


FIGURE 11.11

Using `popen("ls")` to list remote directories.

The following code uses `popen` toward that end:

```

/* rlsd.c - a remote ls server - with paranoia
 */

#include <stdio.h>

```

```

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
#include <strings.h>

#define PORTNUM 15000 /* our remote ls server port */
#define HOSTLEN 256
#define oops(msg)      { perror(msg) ; exit(1) ; }

int main(int ac, char *av[])
{
    struct sockaddr_in  saddr; /* build our address here */
    struct hostent      *hp;   /* this is part of our */
    char hostname[HOSTLEN]; /* address */
    int sock_id, sock_fd; /* line id, file desc */
    FILE *sock_fpi, *sock_fpo; /* streams for in and out */
    FILE *pipe_fp; /* use popen to run ls */
    char dirname[BUFSIZ]; /* from client */
    char command[BUFSIZ]; /* for popen() */
    int dirlen, c;

    /** Step 1: ask kernel for a socket */

    sock_id = socket( PF_INET, SOCK_STREAM, 0 ); /* get a socket */
    if ( sock_id == -1 )
        oops( "socket" );

    /** Step 2: bind address to socket. Address is host,port */

    bzero( (void *)&saddr, sizeof(saddr) ); /* clear out struct */
    gethostname( hostname, HOSTLEN ); /* where am I ? */
    hp = gethostbyname( hostname ); /* get info about host */
    bcopy( (void *)hp->h_addr, (void *)&saddr.sin_addr, hp->h_length);
    saddr.sin_port = htons(PORTNUM); /* fill in socket port */
    saddr.sin_family = AF_INET ; /* fill in addr family */
    if ( bind(sock_id, (struct sockaddr *)&saddr, sizeof(saddr)) != 0 )
        oops( "bind" );

    /** Step 3: allow incoming calls with Qsize=1 on socket */

    if ( listen(sock_id, 1) != 0 )
        oops( "listen" );

    /*
    * main loop: accept(), write(), close()
    */

    while ( 1 ){
        sock_fd = accept(sock_id, NULL, NULL); /* wait for call */
        if ( sock_fd == -1 )
            oops("accept");
    }
}

```

```

/* open reading direction as buffered stream */
if ( (sock_fpi = fdopen(sock_fd, "r")) == NULL )
    oops("fdopen reading");

if ( fgets(dirname, BUFSIZ-5, sock_fpi) == NULL )
    oops("reading dirname");
sanitize(dirname);

/* open writing direction as buffered stream */
if ( (sock_fpo = fdopen(sock_fd, "w")) == NULL )
    oops("fdopen writing");

sprintf(command, "ls %s", dirname);
if ( (pipe_fp = popen(command, "r")) == NULL )
    oops("popen");

/* transfer data from ls to socket */
while ( (c = getc(pipe_fp)) != EOF )
    putc(c, sock_fpo);
pclose(pipe_fp);
fclose(sock_fpo);
fclose(sock_fpi);
    }
}

sanitize(char *str)
/*
 * it would be very bad if someone passed us an dirname like
 * "; rm *" and we naively created a command "ls ; rm *"
 *
 * so..we remove everything but slashes and alphanumerics
 * There are nicer solutions, see exercises
 */
{
    char *src, *dest;

    for ( src = dest = str ; *src ; src++ )
        if ( *src == '/' || isalnum(*src) )
            *dest++ = *src;

    *dest = '\0';
}

```

Notice that our server uses standard buffered streams for reading and for writing. The server uses `fgets` to read the directory name from the client. After calling `popen`, the server transfers data using `getc` and `putc` just as though it were copying a file. Of course, the server is actually copying data from one process to a process on another computer.

Note the `sanitize` function. Any server that runs commands based on arguments and data it receives over the Internet *must* be written extremely carefully. Our server expects to receive the name of a directory from the client. The server appends that

string to the command `ls`. For example, if the client sends the string `"/bin"`, our server creates and executes the string `"ls /bin"`, which is fine. If, though, someone sends the string `" ; rm *"` to our server, our server would create and execute the string `"ls ; rm *"`.

To reduce the risk of damage, our program makes sure the string it receives does not overflow the input buffer, does not overflow the buffer for the command, and does not allow any special characters in the directory name, although limiting the directory name to alphanumeric is too restrictive. The `popen` function is too risky for network services because it passes a string to a shell. It is a poor idea to write any network servers that pass strings to a shell. I included this example for two reasons: first, to show another use of `popen`, and second, to alert you to this danger. It is a serious one.

11.6 SOFTWARE DAEMONS

Unix server programs, like most Unix programs, have short, concise names. Many server programs have names that end in the letter *d*, such as `httpd`, `inetd`, `syslogd`, `atd`. The *d* in these names stands for *daemon*, so the name `syslogd` identifies the program as the *system log daemon*. A daemon is a term for an attendant spirit, sort of a supernatural helper that floats around waiting to help out. On your system, type `ps -el` or `ps -ax` and look for processes running programs with names that end in the letter *d*. You can then read the manual pages about those commands. By doing so, you can learn more about the way Unix uses client/server programming to manage many basic operations.

Most daemon processes are started when the system comes up. Shell scripts in a directory with a name like `/etc/rc.d`³ start these servers in the background, where they run detached from any terminals, ready to provide data or services.

SUMMARY

MAIN IDEAS

- Some programs are constructed as separate processes that send data back and forth. In a client/server system, a server process provides processing or data to client processes.
- A client/server system consists of a communication system and a protocol. Clients and servers can communicate through pipes or sockets. A protocol is a set of rules for the structure of a conversation.
- The `popen` library function can make any shell command into a server program and makes access to the server look like access to buffered files.
- A pipe is a connected pair of file descriptors. A socket is an unconnected communication endpoint, a potential file descriptor. A client process creates a communication link by connecting its socket to a server socket.
- Connections between sockets may extend from one computer to another. Each socket is identified by a machine number and a port number.

³The exact directory name depends on the version of Unix.

- Connections to pipes and sockets use file descriptors. File descriptors provide programs with a single interface for communication with files, devices, and other processes.

WHAT NEXT?

In this chapter, we looked at the client/server model of program design. We saw two methods for connecting processes: pipes and sockets. In the next chapter, we focus on the design principles of client/server programming, and we write more complex applications. In particular, we combine socket programming with our knowledge of file systems and process control to write a Web server.

EXPLORATIONS

- 11.1** *The pizza-delivery-order protocol* What if you ran a pizza-delivery service instead of a time or directory-assistance service? The protocol would be more complicated. Describe the sequence of messages passed between server and client for a pizza-delivery service. Note that this protocol contains a loop that allows the client to add several items to the order.
- 11.2** *popen and signals* The version of `popen` provided in the text does not do anything about signals. Is that correct? A child inherits the signal-handling settings from its parent. Answer this question by considering the three cases of signal handling for the parent: terminate, ignore, and call a function.
- 11.3** *Data flow in testing timeserv* The sample run of the time server and time client showed that I used `ssh` to log in to `computer2` from `computer1`. I was still logged into `computer1`, but I was running a shell on `computer2`. From that shell, I compiled and ran the time client.
- My terminal is really connected to `computer1`. Redraw figure 11.11 so it includes my shell on `computer1`, my shell on `computer2`, my terminal, and the correct flow of data from `timeclnt` to my terminal. Pretty complex data flow, isn't it?
- 11.4** *Sockets are not files* We saw earlier that disk files and device files both support the standard file interface, but connections to disk files have one set of properties, and connections to device files have a very different set of properties. What special properties do sockets have? Look at the manual page for `setsockopt` for details.
- 11.5** *stderr and servers* The remote directory lister runs the `ls` command. What happens when `ls` encounters an error? For example, the directory specified may not exist or may not be readable by the server. What happens to the error messages `ls` generates? Consider two ways to handle error messages from `ls`. First, how would you send error messages back to the client? Second, how would you record error messages in a log and tell the user about the problem?

PROGRAMMING EXERCISES

- 11.6** Add the `-c` option to `tinybc`. Once you have added this option, the following command should work:

```
printf "2 + 2\n4 * 4\n" | tinybc -c | dc
```

- 11.7** Add the `-c` option to your shell. What changes do you need to make?

- 11.8** Write `pclose`. The function takes as an argument the `FILE *` returned from `popen`. The `fdopen` function has allocated memory for the buffer and for bookkeeping details. The `fclose` function deallocates that memory and closes the underlying file descriptor. What else does `pclose` have to do? What if another child process dies between the call to `popen` and `pclose`?
- 11.9** *caller ID* Our time server does not use the caller-ID feature the `accept` system call provides. Modify `timeserv.c` so that it prints a message such as, *Got a call from 123.123.123.123 (computer2.mysite.net)* when it receives a request.
Read the manual and header files to learn about the functions and structures you need for this project.
- 11.10** Write a program that uses `sort` as a subroutine. Your program should read lines of data into an array of strings. The program should then create two pipes and then create a process to run `sort`. Send the sequence of lines of input to `sort` through one pipe, then close that pipe. Read the results from `sort` through the other pipe, and put the results back into the array. Print the array.
- 11.11** *Bidirectional pipes* Versions of Unix based on System V provide bidirectional pipes. You can test if a version of Unix supports these by running this program:

```
/*
 * testbpd.c - test bidirectional pipes
 */

main()
{
    int    p[2];
    if ( pipe(p) == -1 ) exit(1);
    if ( write(p[0], "hello", 5) == -1 )
        perror("write into pipe[0] failed");
    else
        printf("write into pipe[0] worked\n");
}
```

Internally, a bidirectional pipe contains two queues, one from `pipe[0]` to `pipe[1]` and one going the other way. Writing data into one end of a pipe adds it to the queue that goes to the other end, and reading data from one end of a pipe pulls bytes from the queue leading from the other end.

If your system does not support bidirectional pipes, you can create a pair using this call:

```
#include <sys/types.h>
#include <sys/socket.h>
int apipe[2]; /* a pipe */
socketpair(AF_UNIX, SOCK_STREAM, PF_UNSPEC, apipe);
```

Recode `tinybc.c` so it uses a single bidirectional pipe rather than using two unidirectional pipes.

- 11.12** *IP blocking* Modify `timeserv.c` so it only responds to clients calling from a specific host IP number. The server accepts the call and checks the address of the client. If the client is not at the specified IP number, the server hangs up, otherwise, the server sends back the time message.

Enhance this blocking feature so the server reads a list of acceptable IP numbers from a file. Describe some practical applications of this technique.

- 11.13** *More secure* Using `popen` in a server is extremely risky. There are two ways to address the risks. The first is to write a more flexible, but still secure version of the `sanitize` function. For example, there is no problem with directory names that contain periods, dashes, spaces, and many other characters. A directory name can contain asterisks and semicolons. It is just that the shell assigns special meanings to those characters. Write a more useful, but still safe string-cleaning function.
- The other method is to drop `popen` and use `fork`, `exec`, `dup`, etc. Rewrite `rlsd.c` using this method. Do you need to use `wait`? Why or why not?
- 11.14** *A finger server* Write a version of the directory-assistance server we saw running at port 79. The server should accept a username as a single line of input and then send back to the client a list of matching records from the local user list.
- 11.15** *Time-server proxy* A proxy is a program that accepts your request, forwards it to another server, and then sends the response from that server back to you. Like a dry cleaner storefront, the shop does not do the cleaning; it just transfers the clothes to and from the cleaning plant.
- Write a time-server proxy. Your program should accept connections on the standard port. To process the connection, your program should then open a connection to a “real” time server, get the time from that server, and send the response back to your client.
- 11.16** *Caches and proxies* Read about proxy servers in the previous problem. The time only changes once per second, so if your proxy server gets lots of connections milliseconds apart, there is no reason for your proxy to call a server for the time. Write a *caching proxy* time server that stores the time it read from the server and only makes a new call when that string is more than a second old. (See `gettimeofday`.)
- 11.17** *More caches and proxies* Admittedly, the caching time server in the previous exercise is a silly idea. Explain why using a cache for a finger server makes sense. Write a finger server that keeps a cache of user information.
- The time server has a natural lifetime for items in the cache, but how do you decide how long to keep user information in the finger-server cache?
- 11.18** *A bakery number server* Some bakery shops have a machine that dispenses numbers to customers. A sign over the counter that says “Now Serving” displays the number of the next customer to be served. Devise a client-server pair to implement the bakery number system. The server issues sequential numbers. A user runs the client program to obtain a number from the server.
- 11.19** *Using popen* Every C programmer knows that `argv[0]` usually contains the name of the program being run. There is another, slightly roundabout, way a process can obtain its name. A program can use `popen` and search the output of the `ps` command for its own process ID number. Write a program that uses this method.