Chapter 18

Connection-Oriented Communication

Most local-area networks have file servers that manage common disk space, making it easier to share files and perform backups for user clients. Standard UNIX network services such as mail and file transfer also use the client-server paradigm. This chapter discusses several common client-server models for providing services over existing network infrastructure. The models are implemented with the Universal Internet Communication Interface (UICI), a simplified API for connection-oriented communication that is freely available from the book web site. The UICI interface is then implemented in terms of stream sockets and TCP.

Objectives

- Learn about connection-oriented communication
- Experiment with sockets and TCP
- Explore different server designs
- Use the client-server model in applications
- Understand thread-safe communication

18.1 The Client-Server Model

Many network applications and services such as web browsing, mail, file transfer (`ftp`), authentication (Kerberos), remote login (`telnet`) and access to remote file systems (NFS) use the client-server paradigm. In each of these applications, a client sends a request for service to a server. A service is an action, such as changing the status of a remote file, that the server performs on behalf of the client. Often the service includes a response or returns information, for example by retrieving a remote file or web page.

The client-server model appears at many levels in computer systems. For example, an object that calls a method of another object in an object-oriented program is said to be a *client of the object*. At the system level, daemons that manage resources such as printers are servers for system user clients. On the Internet, browsers are client processes that request resources from web servers. The key elements of the client-server model are as follows.

- The client, not the service provider, initiates the action.
- The server waits passively for requests from clients.
- The client and server are connected by a communication channel that they access through communication endpoints.

Servers should robustly handle multiple simultaneous client requests in the face of unexpected client behavior. This chapter especially emphasizes the importance of catching errors and taking appropriate action during client-server interactions. You wouldn't want a web server to exit when a user mistypes a URL in the browser. Servers are long-running and must release *all* the resources allocated for individual client requests.

Although most current computer system services are based on the client-server model, other models such as event notification [4, 36] or peer-to-peer computing [90] may become more important in the future.

18.2 Communication Channels

A *communication channel* is a logical pathway for information that is accessed by participants through communication endpoints. The characteristics of the channel constrain the types of interaction allowed between sender and receiver. Channels can be shared or private, one-way or two-way. Two-way channels can be symmetric or asymmetric. Channels are distinguished from the underlying physical conduit, which may support many types of channels.

In object-orient programming, clients communicate with an object by calling a method. In this context, client and server share an address space, and the communication channel is the activation record that is created on the process stack for the call. The request consists of the parameter values that are pushed on the stack as part of the call,

and the optional reply is the method's return value. Thus, the activation record is a private, asymmetric two-way communication channel. The method call mechanism of the object-oriented programming language establishes the communication endpoints. The system infrastructure for managing the process stack furnishes the underlying conduit for communication.

Many system services in UNIX are provided by server processes running on the same machine as their clients. These processes can share memory or a file system, and clients make requests by writing to such a shared resource.

Programs 6.7 and 6.8 of Chapter 6 use a named pipe as a communication channel for client requests. The named pipe is used as a shared one-way communication channel that can handle requests from any number of clients. Named pipes have an associated pathname, and the system creates an entry in the file system directory corresponding to this pathname when `mkfifo` executes. The file system provides the underlying conduit. A process creates communication endpoints by calling `open` and accesses these endpoints through file descriptors. Figure 18.1 shows a schematic of the communication supported in this example.

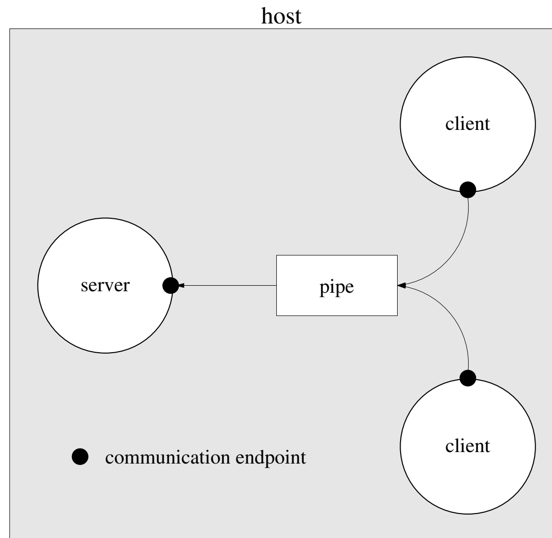

Figure 18.1: Multiple clients write requests to a shared one-way communication channel.

Named pipes can be used for short client requests, since a write of `PIPE_BUF` bytes or less is not interleaved with other writes to the same pipe. Unfortunately, named pipes present several difficulties when the requests are long or the server must respond. If the

server simply opens another named pipe for responses, individual clients have no guarantee that they will read the response meant for them. If the server opens a unique pipe for each response, the clients and server must agree in advance on a naming convention. Furthermore, named pipes are persistent. They remain in existence unless their owners explicitly unlink them. A general mechanism for communication should release its resources when the interacting parties no longer exist.

Transmission Control Protocol (TCP) is a connection-oriented protocol that provides a reliable channel for communication, using a conduit that may be unreliable. *Connection-oriented* means that the initiator (the client) first establishes a connection with the destination (the server), after which both of them can send and receive information. TCP implements the connection through an exchange of messages, called a *three-way handshake*, between initiator and destination. TCP achieves reliability by using receiver acknowledgments and retransmissions. TCP also provides flow control so that senders don't overwhelm receivers with a flood of information. Fortunately, the operating system network subsystem implements TCP, so the details of the protocol exchanges are not visible at the process level. If the network fails, the process detects an error on the communication endpoint. The process should never receive incorrect or out-of-order information when using TCP.

Figure 18.2 illustrates the setup for connection-oriented communication. The server monitors a passive communication endpoint whose address is known to clients. Unlike other endpoints, passive or listening endpoints have resources for queuing client connection requests and establishing client connections. The action of accepting a client request creates a new communication endpoint for private, two-way symmetric communication with that client. The client and server then communicate by using handles (file descriptors) and do not explicitly include addresses in their messages. When finished, the client and server close their file descriptors, and the system releases the resources associated with the connection. Connection-oriented protocols have an initial setup overhead, but they allow transparent management of errors when the underlying conduits are not error-free.

☐ Exercise 18.1

Figure 18.3 illustrates a situation in which two clients have established connections with a server. What strategies are available to the server for managing the resulting private communication channels (each with its own file descriptor)?

Answer:

The server cannot make any assumptions about the order in which information will arrive on the file descriptors associated with the clients' private communication channels. Therefore, a solution to alternately read from one descriptor and then the other is incorrect. Section 12.1 outlines the available approaches for monitoring multiple file descriptors. The server could use `select` or `poll`, but the server would not be able to accept any additional connection requests while blocking on

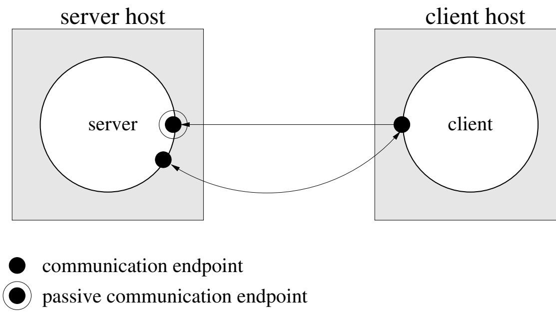

Figure 18.2: Schematic of connection-oriented client-server communication.

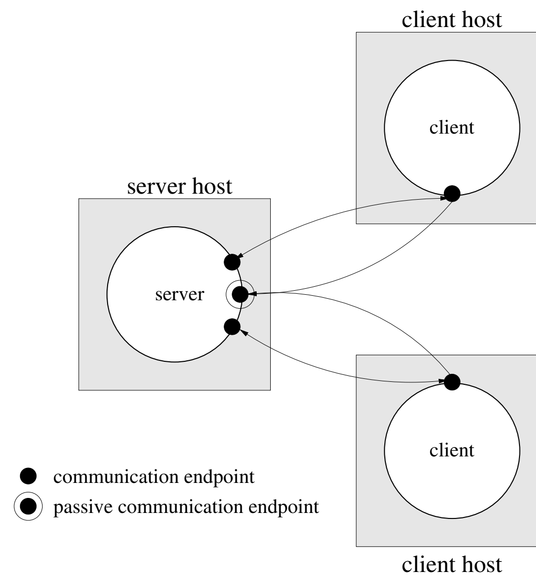

Figure 18.3: Many clients can request connections to the same communication endpoint.

these calls. Simple polling wastes CPU cycles. Asynchronous I/O is efficient, but complex to program. Alternatively, the server can fork a child process or create a separate thread to handle the client communication.

Both connectionless and connection-oriented protocols are considered to be low-level in the sense that the request for service involves visible communication. The programmer is explicitly aware of the server's location and must explicitly name the particular server to be accessed.

The naming of servers and services in a network environment is a difficult problem. An obvious method for designating a server is by its process ID and a host ID. However, the operating system assigns process IDs chronologically by process creation time, so the client cannot know in advance the process ID of a particular server process on a host.

The most commonly used method for specifying a service is by the address of the host machine (the IP address) and an integer called a port number. Under this scheme, a server monitors one or more communication channels associated with port numbers that have been designated in advance for a particular service. Web servers use port 80 by default, whereas `ftp` servers use port 21. The client explicitly specifies a host address and a port number for the communication. Section 18.8 discusses library calls for accessing IP addresses by using host names.

This chapter focuses on connection-oriented communication using TCP/IP and stream sockets with servers specified by host addresses and port numbers. More sophisticated methods of naming and locating services are available through object registries [44], directory services [129], discovery mechanisms [4] or middleware such as CORBA [104]. Implementations of these approaches are not universally available, nor are they particularly associated with UNIX.

18.3 Connection-Oriented Server Strategies

Once a server receives a request, it can use a number of different strategies for handling the request. The *serial server* depicted in Figure 18.2 completely handles one request before accepting additional requests.

■ Example 18.2

The following pseudocode illustrates the serial-server strategy.

```
for ( ; ; ) {
   wait for a client request on the listening file descriptor
   create a private two-way communication channel to the client
   while (no error on the private communication channel)
      read from the client
      process the request
      respond to the client
   close the file descriptor for the private communication channel
}
```

A busy server handling long-lived requests such as file transfers cannot use a serial-server strategy that processes only one request at a time. A *parent server* forks a child process to handle the actual service to the client, freeing the server to listen for additional requests. Figure 18.4 depicts the parent-server strategy. The strategy is ideal for services such as file transfers, which take a relatively long time and involve a lot of blocking.

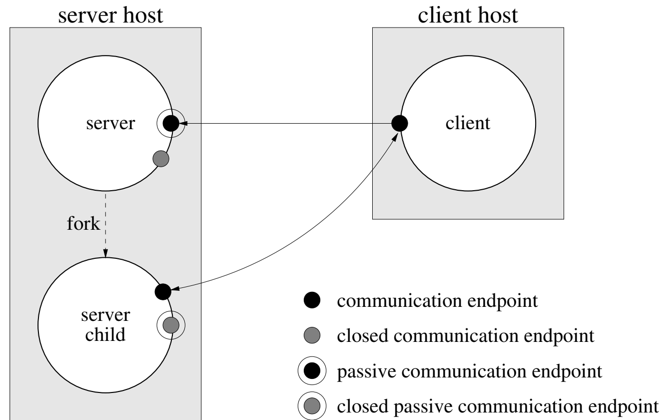

Figure 18.4: A parent server forks a child to handle the client request.

■ Example 18.3

The following pseudocode illustrates the parent-server strategy.

```
for( ; ; ) {
   wait for a client request on the listening file descriptor
   create a private two-way communication channel to the client
   fork a child to handle the client
   close file descriptor for the private communication channel
   clean up zombie children
}
```

The child process does the following.

```
close the listening file descriptor
handle the client
close the communication for the private channel
exit
```

Since the server's child handles the actual service in the parent-server strategy, the server can accept multiple client requests in rapid succession. The strategy is analogous to the old-fashioned switchboard at some hotels. A client calls the main number at the hotel (the connection request). The switchboard operator (server) answers the call, patches the

connection to the appropriate room (the server child), steps out of the conversation, and resumes listening for additional calls.

☐ Exercise 18.4

What happens in Example 18.3 if the parent does not close the file descriptor corresponding to the private communication channel?

Answer:

In this case, both the server parent and the server child have open file descriptors to the private communication channel. When the server child closes the communication channel, the client will not be able to detect end-of-file because a remote process (the server parent) still has it open. Also, if the server runs for a long time with many client requests, it will eventually run out of file descriptors.

☐ Exercise 18.5

What is a zombie child? What happens in Example 18.3 if the server parent does not periodically wait for its zombie children?

Answer:

A *zombie* is a process that has completed execution but has not been waited for by its parent. Zombie processes do not release all their resources, so eventually the system may run out of some critical resource such as memory or process IDs.

The *threaded server* depicted in Figure 18.5 is a low-overhead alternative to the parent server. Instead of forking a child to handle the request, the server creates a thread in its own process space. Threaded servers can be very efficient, particularly for small or I/O intensive requests. A drawback of the threaded-server strategy is possible interference among multiple requests due to the shared address space. For computationally intensive services, the additional threads may reduce the efficiency of or block the main server thread. Per-process limits on the number of open file descriptors may also restrict the number of simultaneous client requests that can be handled by the server.

■ Example 18.6

The following pseudocode illustrates the threaded-server strategy.

```
for ( ; ; ) {
   wait for a client request on the listening file descriptor
   create a private two-way communication channel to the client
   create a detached thread to handle the client
}
```

☐ Exercise 18.7

What is the purpose of creating a detached (as opposed to attached) thread in Example 18.6?

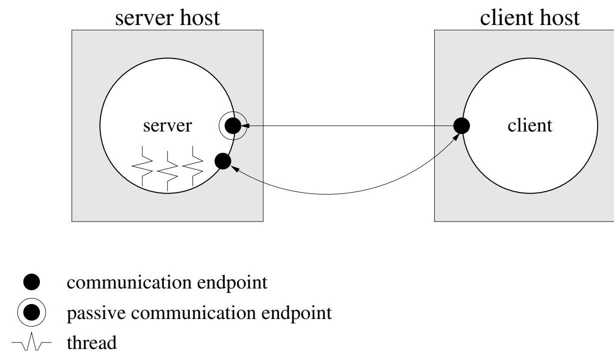

Figure 18.5: A threaded server creates threads to handle client requests.

Answer:
Detached threads release all their resources when they exit, hence the main thread doesn't have to wait for them. The `waitpid` function with the `NOHANG` option allows a process to wait for completed children without blocking. There is no similar option for the `pthread_join` function.

◻ Exercise 18.8

What would happen if the main thread closed the communication file descriptor after creating the thread to handle the communication?

Answer:
The main thread and child threads execute in the same process environment and share the same file descriptors. If the main thread closes the communication file descriptor, the newly created thread cannot access it. Compare this situation to that encountered in the parent server of Example 18.3, in which the child process receives a copy of the file descriptor table and executes in a different address space.

Other strategies are possible. For example, the server could create a fixed number of child processes when it starts and each child could wait for a connection request. This approach allows a fixed number of simultaneous parallel connections and saves the overhead of creating a new process each time a connection request arrives. Similarly, another threading strategy has a main thread that creates a pool of worker threads that each wait for connection requests. Alternatively, the main thread can wait for connection requests and distribute communication file descriptors to free worker threads. Chapter 22 outlines a project to compare the performance of different server strategies.

18.4 Universal Internet Communication Interface (UICI)

The Universal Internet Communication Interface (UICI) library, summarized in Table 18.1, provides a simplified interface to connection-oriented communication in UNIX. UICI is not part of any UNIX standard. The interface was designed by the authors to abstract the essentials of network communication while hiding the details of the underlying network protocols. UICI has been placed in the public domain and is available on the book web site. Programs that use UICI should include the `uici.h` header file.

This section introduces the UICI library. The next two sections implement several client-server strategies in terms of UICI. Section 18.7 discusses the implementation of UICI using sockets, and Appendix C provides a complete UICI implementation.

When using sockets, a server creates a communication endpoint (a socket) and associates it with a well-known port (binds the socket to the port). Before waiting for client requests, the server sets the socket to be passive so that it can accept client requests (sets the socket to listen). Upon detection of a client connection request on this endpoint, the server generates a new communication endpoint for private two-way communication with the client. The client and server access their communication endpoints by using file descriptors to read and write. When finished, both parties close the file descriptors, releasing the resources associated with the communication channel.

UICI prototype	description (assuming no errors)
`int u_open(u_port_t port)`	creates a TCP socket bound to `port` and sets the socket to be passive returns a file descriptor for the socket
`int u_accept(int fd, char *hostn, int hostnsize)`	waits for connection request on `fd`; on return, `hostn` has first `hostname-1` characters of the client's host name returns a communication file descriptor
`int u_connect(u_port_t port, char *hostn)`	initiates a connection to server on port `port` and host `hostn`. returns a communication file descriptor

Table 18.1: The UICI API. If unsuccessful, UICI functions return –1 and set `errno`.

Figure 18.6 depicts a typical sequence of UICI calls used in client-server communication. The server creates a communication endpoint (`u_open`) and waits for a client to send a request (`u_accept`). The `u_accept` function returns a private communication file descriptor. The client creates a communication endpoint for communicating with the server (`u_connect`).

Once they have established a connection, a client and server can communicate over the network by using the ordinary `read` and `write` functions. Alternatively, they can use the

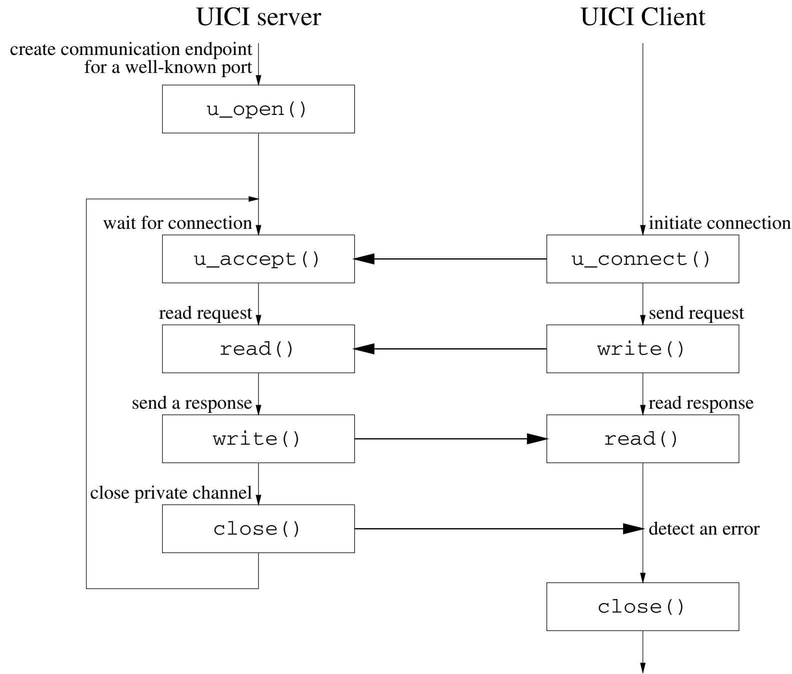

Figure 18.6: A typical interaction of a UICI client and server.

more robust r_read and r_write from the restart library of Appendix B. Either side can terminate communication by calling close or r_close. After close, the remote end detects end-of-file when reading or an error when writing. The diagram in Figure 18.6 shows a single request followed by a response, but more complicated interactions might involve several exchanges followed by close.

In summary, UICI servers follow these steps.

1. Open a well-known listening port (u_open). The u_open functions returns a *listening file descriptor*.
2. Wait for a connection request on the listening file descriptor (u_accept). The u_accept function blocks until a client requests a connection and then returns a *communication file descriptor* to use as a handle for private, two-way client-server communication.
3. Communicate with the client by using the communication file descriptor (read and write).
4. Close the communication file descriptor (close).

UICI clients follow these steps.

1. Connect to a specified host and port (u_connect). The connection request returns the communication file descriptor used for two-way communication with the server.
2. Communicate with the server by using the communication file descriptor (read and write).
3. Close the communication file descriptor (close).

18.4.1 Handling errors

A major design issue for UICI was how to handle errors. UNIX library functions generally report errors by returning –1 and setting errno. To keep the UICI interface simple and familiar, UICI functions also return –1 and set errno. None of the UICI functions display error messages. Applications using UICI should test for errors and display error messages as appropriate. Since UICI functions always set errno when a UICI function returns an error, applications can use perror to display the error message. POSIX does not specify an error code corresponding to the inability to resolve a host name. The u_connect function returns –1 and sets errno to EINVAL, indicating an invalid parameter when it cannot resolve the host name.

18.4.2 Reading and writing

Once they have obtained an open file descriptor from u_connect or u_accept, UICI clients and servers can use the ordinary read and write functions to communicate. We use the functions from the restart library since they are more robust and simplify the code.

Recall that r_read and r_write both restart themselves after being interrupted by a signal. Like read, r_read returns the number of bytes read or 0 if it encounters end-of-file. If unsuccessful, r_read returns –1 and sets errno. If successful, r_write returns the number of bytes requested to write. The r_write function returns –1 and sets errno if an error occurred or if it could not write all the requested bytes without error. The r_write function restarts itself if not all the requested bytes have been written. This chapter also uses the copyfile function from the restart library, introduced in Program 4.6 on page 100 and copy2files introduced in Program 4.13 on page 111.

The restart library supports only blocking I/O. That is, r_read or r_write may cause the caller to block. An r_read call blocks until some information is available to be read. The meaning of blocking for r_write is less obvious. In the present context, blocking means that r_write returns when the output has been transferred to a buffer used by the transport mechanism. Returning does not imply that the message has actually been delivered to the destination. Writes may also block if message delivery problems arise in the lower protocol layers or if all the buffers for the network protocols are full. Fortunately, the issues of blocking and buffering are transparent for most applications.

18.5 UICI Implementations of Different Server Strategies

Program 18.1 shows a serial-server program that copies information from a client to standard output, using the UICI library. The server takes a single command-line argument specifying the number of the well-known port on which it listens. The server obtains a listening file descriptor for the port with `u_open` and then displays its process ID. It calls `u_accept` to block while waiting for a client request. The `u_accept` function returns a communication file descriptor for the client communication. The server displays the name of the client and uses `copyfile` of Program 4.6 on page 100 to perform the actual copying. Once it has finished the copying, the server closes the communication file descriptor, displays the number of bytes copied, and resumes listening.

Program 18.1 — **server.c**

A serial server implemented using UICI.

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "restart.h"
#include "uici.h"

int main(int argc, char *argv[]) {
   int bytescopied;
   char client[MAX_CANON];
   int communfd;
   int listenfd;
   u_port_t portnumber;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s port\n", argv[0]);
      return 1;
   }
   portnumber = (u_port_t) atoi(argv[1]);
   if ((listenfd = u_open(portnumber)) == -1) {
      perror("Failed to create listening endpoint");
      return 1;
   }
   fprintf(stderr, "[%ld]:waiting for the first connection on port %d\n",
                    (long)getpid(), (int)portnumber);
   for ( ; ; ) {
      if ((communfd = u_accept(listenfd, client, MAX_CANON)) == -1) {
         perror("Failed to accept connection");
         continue;
      }
      fprintf(stderr, "[%ld]:connected to %s\n", (long)getpid(), client);
      bytescopied = copyfile(communfd, STDOUT_FILENO);
      fprintf(stderr, "[%ld]:received %d bytes\n", (long)getpid(), bytescopied);
      if (r_close(communfd) == -1)
         perror("Failed to close communfd\n");
   }
}
```

Program 18.1 — **server.c**

☐ Exercise 18.9

Under what circumstances does a client cause the server in Program 18.1 to terminate?

Answer:

The server executes the first `return` statement if it is not started with a single command-line argument. The `u_open` function creates a communication endpoint associated with a port number. The `u_open` function fails if the port is invalid, if the port is in use, or if system resources are not available to support the request. At this point, no clients are involved. Once the server has reached `u_accept`, it does not terminate unless it receives a signal. A client on a remote machine cannot cause the server to terminate. A failure of `u_accept` causes the server to loop and try again. Notice that I/O errors cause `copyfile` to return, but these errors do not cause server termination.

Program 18.2 implements the parent-server strategy. The parent accepts client connections and forks a child to call `copyfile` so that the parent can resume waiting for connections. Because the child receives a copy of the parent's environment at the time of the fork, it has access to the private communication channel represented by `communfd`.

☐ Exercise 18.10

What happens if the client name does not fit in the buffer passed to `u_accept`?

Answer:

The implementation of `u_accept` does not permit the name to overflow the buffer. Instead, `u_accept` truncates the client name. (See Section 18.7.6.)

☐ Exercise 18.11

What happens if after the connection is made, you enter text at standard input of the server?

Answer:

The server program never reads from standard input, and what you type at standard input is not sent to the remote machine.

☐ Exercise 18.12

Program 18.2 uses `r_close` and `r_waitpid` from the restart library. How does this affect the behavior of the program?

Answer:

Functions in the restart library restart the corresponding function when the return value is –1 and `errno` is `EINTR`. This return condition occurs when the signal handler of a caught signal returns. Program 18.2 does not catch any signals, so using the restarted versions is not necessary. We use the functions from the restart library to make it easier to add signal handling capability to the programs.

Program 18.2 **serverp.c**

A server program that forks a child to handle communication.

```
#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "restart.h"
#include "uici.h"

int main(int argc, char *argv[]) {
   int bytescopied;
   pid_t child;
   char client[MAX_CANON];
   int communfd;
   int listenfd;
   u_port_t portnumber;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s port\n", argv[0]);
      return 1;
   }
   portnumber = (u_port_t) atoi(argv[1]);
   if ((listenfd = u_open(portnumber)) == -1) {
      perror("Failed to create listening endpoint");
      return 1;
   }
   fprintf(stderr, "[%ld]: Waiting for connection on port %d\n",
                    (long)getpid(), (int)portnumber);
   for ( ; ; ) {
      if ((communfd = u_accept(listenfd, client, MAX_CANON)) == -1) {
         perror("Failed to accept connection");
         continue;
      }
      fprintf(stderr, "[%ld]:connected to %s\n", (long)getpid(), client);
      if ((child = fork()) == -1) {
         perror("Failed to fork a child");
         continue;
      }
      if (child == 0) {                                        /* child code */
         if (r_close(listenfd) == -1) {
            fprintf(stderr, "[%ld]:failed to close listenfd: %s\n",
                             (long)getpid(), strerror(errno));
            return 1;
         }
         bytescopied = copyfile(communfd, STDOUT_FILENO);
         fprintf(stderr, "[%ld]:received %d bytes\n",
                          (long)getpid(), bytescopied);
         return 0;
      }
      if (r_close(communfd) == -1)                            /* parent code */
         fprintf(stderr, "[%ld]:failed to close communfd: %s\n",
                          (long)getpid(), strerror(errno));
      while (r_waitpid(-1, NULL, WNOHANG) > 0)  ;         /* clean up zombies */
   }
}
```

Program 18.2 **serverp.c**

18.6 UICI Clients

Program 18.3 shows the client side of the file copy. The client connects to the desired port on a specified host by calling `u_connect`. The `u_connect` function returns the communication file descriptor. The client reads the information from standard input and copies it to the server. The client exits when it receives end-of-file from standard input or if it encounters an error while writing to the server.

Program 18.3 — **client.c**

A client that uses UICI for communication.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "restart.h"
#include "uici.h"

int main(int argc, char *argv[]) {
   int bytescopied;
   int communfd;
   u_port_t portnumber;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s host port\n", argv[0]);
      return 1;
   }
   portnumber = (u_port_t)atoi(argv[2]);
   if ((communfd = u_connect(portnumber, argv[1])) == -1) {
      perror("Failed to make connection");
      return 1;
   }
   fprintf(stderr, "[%ld]:connected %s\n", (long)getpid(), argv[1]);
   bytescopied = copyfile(STDIN_FILENO, communfd);
   fprintf(stderr, "[%ld]:sent %d bytes\n", (long)getpid(), bytescopied);
   return 0;
}
```

Program 18.3 — **client.c**

◻ Exercise 18.13

How would you use Programs 18.1 and 18.3 to transfer information from one machine to another?

Answer:

Compile the server of Program 18.1 as `server`. First, start the server listening on a port (say 8652) by executing the following command.

```
server 8652
```

Compile Program 18.3 as `client`. If the server is running on `usp.cs.utsa.edu`, start the client on another machine with the following command.

```
client usp.cs.utsa.edu 8652
```

Once the client and server have established a connection, enter text on the standard input of the client and observe the server output. Enter the end-of-file character (usually Ctrl-D). The client terminates, and both client and server print the number of bytes transferred. Be sure to replace `usp.cs.utsa.edu` with the host name of your server.

◻ Exercise 18.14

How would you use Programs 18.1 and 18.3 to transfer the file `t.in` on one machine to the file `t.out` on another? Will `t.out` be identical to `t.in`? What happens to the messages displayed by the client and server?

Answer:

Use I/O redirection. Start the server of Program 18.1 on the destination machine (say, `usp.cs.utsa.edu`) by executing the following command.

```
server 8652 > t.out
```

Start the client of Program 18.3 on the source machine by executing the following command.

```
client usp.cs.utsa.edu 8652 < t.in
```

Be sure to substitute your server's host name for `usp.cs.utsa.edu`. The source and destination files should have identical content. Since the messages are sent to standard error, which is not redirected, these messages still appear in the usual place on the two machines.

The client and server programs presented so far support communication only from the client to the server. In many client-server applications, the client sends a request to the server and then waits for a response.

◻ Exercise 18.15

How would you modify the server of Program 18.1 to produce a server called `reflectserver` that echoes its response back to the client, rather than to standard output?

Answer:

The only modification needed would be to replace the reference to `STDOUT_FILENO` with `communfd`.

Program 18.4 is a client program that can be used with the server of Exercise 18.15. The `reflectclient.c` sends a fixed-length message to a server and expects that message to be echoed back. Program 18.4 checks to see that it receives exactly the same message that it sends.

Program 18.4 **reflectclient.c**

A client that sends a fixed-length test message to a server and checks that the reply is identical to the message sent.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "restart.h"
#include "uici.h"
#define BUFSIZE 1000

int main(int argc, char *argv[]) {
   char bufrecv[BUFSIZE];
   char bufsend[BUFSIZE];
   int bytesrecvd;
   int communfd;
   int i;
   u_port_t portnumber;
   int totalrecvd;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s host port\n", argv[0]);
      return 1;
   }
   for (i = 0; i < BUFSIZE; i++)                    /* set up a test message */
      bufsend[i] = (char)(i%26 + 'A');
   portnumber = (u_port_t)atoi(argv[2]);
   if ((communfd = u_connect(portnumber, argv[1])) == -1) {
      perror("Failed to establish connection");
      return 1;
   }
   if (r_write(communfd, bufsend, BUFSIZE) != BUFSIZE) {
      perror("Failed to write test message");
      return 1;
   }
   totalrecvd = 0;
   while (totalrecvd < BUFSIZE) {
      bytesrecvd = r_read(communfd, bufrecv + totalrecvd, BUFSIZE - totalrecvd);
      if (bytesrecvd <= 0) {
         perror("Failed to read response message");
         return 1;
      }
      totalrecvd += bytesrecvd;
   }
   for (i = 0; i < BUFSIZE; i++)
      if (bufsend[i] != bufrecv[i])
         fprintf(stderr, "Byte %d read does not agree with byte written\n", i);
   return 0;
}
```

Program 18.4 **reflectclient.c**

Many client-server applications require symmetric bidirectional communication between client and server. The simplest way to incorporate bidirectionality is for the client and the server to each fork a child to handle the communication in the opposite direction.

■ Example 18.16

To make the client in Program 18.3 bidirectional, declare an integer variable, `child`, and replace the line

```
bytescopied = copyfile(STDIN_FILENO, communfd);
```

with the following code segment.

```
if ((child = fork()) == -1) {
   perror("Failed to fork a child");
   return 1;
}
if (child == 0)                                          /* child code */
   bytescopied = copyfile(STDIN_FILENO, communfd);
else                                                    /* parent code */
   bytescopied = copyfile(communfd, STDOUT_FILENO);
```

□ Exercise 18.17

Suppose we try to make a bidirectional serial server from Program 18.1 by declaring an integer variable called `child` and replacing the following line with the replacement code of Example 18.16.

```
bytescopied = copyfile(communfd, STDOUT_FILENO);
```

What happens?

Answer:

This approach has several flaws. Both the parent and child return to the `u_accept` loop after completing the transfer. While copying still works correctly, the number of processes grows each time a connection is made. After the first connection completes, two server processes accept client connections. If two server connections are active, characters entered at standard input of the server go to one of the two connections. The code also causes the process to exit if `fork` fails. Normally, the server should not exit on account of a possibly temporary problem.

■ Example 18.18

To produce a bidirectional serial server, replace the `copyfile` line in Program 18.1 with the following code.

```
int child;

child = fork();
if ((child = fork()) == -1)
   perror("Failed to fork second child");
else if (child == 0) {                                   /* child code */
   bytescopied = copyfile(STDIN_FILENO, communfd);
   fprintf(stderr, "[%ld]:sent %d bytes\n", (long)getpid(), bytes_copied);
   return 0;
}
bytescopied = copyfile(communfd, STDOUT_FILENO);        /* parent code */
fprintf(stderr, "[%ld]:received %d bytes\n", (long)getpid(), bytescopied);
r_wait(NULL);
```

The child process exits after printing its message. The original process waits for the child to complete before continuing and does not accept a new connection until both ends of the transmission complete. If the fork fails, only the parent communicates.

☐ Exercise 18.19

The modified server suggested in Example 18.18 prints out the number of bytes transferred in each direction. How would you modify the code to print a single number giving the total number of bytes transferred in both directions?

Answer:

This modification would not be simple because the values for transfer in each direction are stored in different processes. You can establish communication by inserting code to create a pipe before forking the child. After it completes, the child could write to the pipe the total number of bytes transferred to the parent.

☐ Exercise 18.20

Suppose that the child of Example 18.18 returns the number of bytes transferred and the parent uses the return value from the status code to accumulate the total number of bytes transferred. Does this approach solve the problem posed in Exercise 18.19?

Answer:

No. Only 8 bits are typically available for the child's return value, which is not large enough to hold the number of bytes transferred.

Another way to do bidirectional transfer is to use `select` or `poll` as shown in Program 4.13 on page 111. The `copy2files` program copies bytes from `fromfd1` to `tofd1` and from `fromfd2` to `tofd2`, respectively, without making any assumptions about the order in which the bytes become available in the two directions. You can use `copy2files` by replacing the `copyfile` line in both server and client with the following code.

```
bytescopied = copy2files(communfd, STDOUT_FILENO, STDIN_FILENO, communfd);
```

Program 18.5 shows the bidirectional client.

☐ Exercise 18.21

How does using `copy2files` differ from forking a child to handle communication in the opposite direction?

Answer:

The `copy2files` function of Program 4.13 terminates both directions of communication if either receives an end-of-file from standard input or if there is an error in the network communication. The child method allows communication to continue in the other direction after one side is closed. You can modify `copy2files`

to keep a flag for each file descriptor indicating whether the descriptor has encountered an error or end-of-file. Only active descriptors would be included in each iteration of `select`.

Program 18.5 — **client2.c**

A bidirectional client.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "uici.h"
#include "restart.h"

int main(int argc, char *argv[]) {
   int bytescopied;
   int communfd;
   u_port_t portnumber;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s host port\n", argv[0]);
      return 1;
   }
   portnumber = (u_port_t)atoi(argv[2]);
   if ((communfd = u_connect(portnumber, argv[1])) == -1) {
      perror("Failed to establish connection");
      return 1;
   }
   fprintf(stderr, "[%ld]:connection made to %s\n", (long)getpid(), argv[1]);
   bytescopied = copy2files(communfd, STDOUT_FILENO, STDIN_FILENO, communfd);
   fprintf(stderr, "[%ld]:transferred %d bytes\n", (long)getpid(), bytescopied);
   return 0;
}
```

Program 18.5 — **client2.c**

18.7 Socket Implementation of UICI

The first socket interface originated with 4.1cBSD UNIX in the early 1980s. In 2001, POSIX incorporated 4.3BSD sockets and an alternative, XTI. XTI (X/Open Transport Interface) also provides a connection-oriented interface that uses TCP. XTI's lineage can be traced back to AT&T UNIX System V TLI (Transport Layer Interface). This book focuses on socket implementations. (See Stevens [115] for an in-depth discussion of XTI.)

This section introduces the main socket library functions and then implements the UICI functions in terms of sockets. Section 18.9 discusses a thread-safe version of UICI. Appendix C gives a complete unthreaded socket implementation of UICI as well as four alternative thread-safe versions. The implementations of this chapter use IPv4 (Internet

UICI	socket functions	action
u_open	socket bind listen	create communication endpoint associate endpoint with specific port make endpoint passive listener
u_accept	accept	accept connection request from client
u_connect	socket connect	create communication endpoint request connection from server

Table 18.2: Overview of UICI API implementation using sockets with TCP.

Protocol version 4). The names of the libraries needed to compile the socket functions are not yet standard. Sun Solaris requires the library options -lsocket and -lnsl. Linux just needs -lnsl, and Mac OS X does not require that any extra libraries be specified. The man page for the socket functions should indicate the names of the required libraries on a particular system. If unsuccessful, the socket functions return –1 and set errno.

Table 18.2 shows the socket functions used to implement each of the UICI functions. The server creates a handle (socket), associates it with a physical location on the network (bind), and sets up the queue size for pending requests (listen). The UICI u_open function, which encapsulates these three functions, returns a file descriptor corresponding to a passive or listening socket. The server then listens for client requests (accept).

The client also creates a handle (socket) and associates this handle with the network location of the server (connect). The UICI u_connect function encapsulates these two functions. The server and client handles, sometimes called *communication* or *transmission endpoints*, are file descriptors. Once the client and server have established a connection, they can communicate by ordinary read and write calls.

18.7.1 The socket function

The socket function creates a communication endpoint and returns a file descriptor. The domain parameter selects the protocol family to be used. We use AF_INET, indicating IPv4. A type value of SOCK_STREAM specifies sequenced, reliable, two-way, connection-oriented byte streams and is typically implemented with TCP. A type value of SOCK_DGRAM provides connectionless communication by using unreliable messages of a fixed length and is typically implemented with UDP. (See Chapter 20.) The protocol parameter specifies the protocol to be used for a particular communication type. In most implementations, each type parameter has only one protocol available (e.g., TCP for SOCK_STREAM and UDP for SOCK_DGRAM), so protocol is usually 0.

SYNOPSIS

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

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

POSIX

If successful, `socket` returns a nonnegative integer corresponding to a socket file descriptor. If unsuccessful, `socket` returns –1 and sets `errno`. The following table lists the mandatory errors for `socket`.

errno	cause
EAFNOSUPPORT	implementation does not support specified address family
EMFILE	no more file descriptors available for process
ENFILE	no more file descriptors available for system
EPROTONOSUPPORT	protocol not supported by address family or by implementation
EPROTOTYPE	socket type not supported by protocol

■ Example 18.22

The following code segment sets up a socket communication endpoint for Internet communication, using a connection-oriented protocol.

```
int sock;

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
   perror("Failed to create socket");
```

18.7.2 The `bind` function

The `bind` function associates the handle for a socket communication endpoint with a specific logical network connection. Internet domain protocols specify the logical connection by a port number. The first parameter to `bind`, `socket`, is the file descriptor returned by a previous call to the `socket` function. The `*address` structure contains a family name and protocol-specific information. The `address_len` parameter is the number of bytes in the `*address` structure.

SYNOPSIS

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

int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);
```

POSIX

If successful, `bind` returns 0. If unsuccessful, `bind` returns –1 and sets `errno`. The following table lists the mandatory errors for `bind` that are applicable to all address families.

errno	cause
EADDRINUSE	specified address is in use
EADDRNOTAVAIL	specified address not available from local machine
EAFNOSUPPORT	invalid address for address family of specified socket
EBADF	socket parameter is not a valid file descriptor
EINVAL	socket already bound to an address, protocol does not support binding to new address, or socket has been shut down
ENOTSOCK	socket parameter does not refer to a socket
EOPNOTSUPP	socket type does not support binding to address

The Internet domain uses `struct sockaddr_in` for `struct sockaddr`. POSIX states that applications should cast `struct sockaddr_in` to `struct sockaddr` for use with socket functions. The `struct sockaddr_in` structure, which is defined in `netinet/in.h`, has at least the following members expressed in network byte order.

```
sa_family_t     sin_family;  /* AF_NET */
in_port_t       sin_port;    /* port number */
struct in_addr  sin_addr;    /* IP address */
```

For Internet communication, `sin_family` is `AF_INET` and `sin_port` is the port number. The `struct in_addr` structure has a member, called `s_addr`, of type `in_addr_t` that holds the numeric value of an Internet address. A server can set the `sin_addr.s_addr` field to `INADDR_ANY`, meaning that the socket should accept connection requests on any of the host's network interfaces. Clients set the `sin_addr.s_addr` field to the IP address of the server host.

■ Example 18.23

The following code segment associates the port 8652 with a socket corresponding to the open file descriptor `sock`.

```
struct sockaddr_in server;
int sock;

server.sin_family = AF_INET;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons((short)8652);
if (bind(sock, (struct sockaddr *)&server, sizeof(server)) == -1)
   perror("Failed to bind the socket to port");
```

Example 18.23 uses `htonl` and `htons` to reorder the bytes of `INADDR_ANY` and `8652` to be in network byte order. Big-endian computers store the most significant byte first; little-endian computers store the least significant byte first. Byte ordering of integers presents a problem when machines with different endian architectures communicate, since they may misinterpret protocol information such as port numbers. Unfortunately, both architectures are common—the SPARC architecture (developed by Sun Microsystems) uses big-endian, whereas Intel architectures use little-endian. The Internet protocols specify

that big-endian should be used for *network byte order*, and POSIX requires that certain socket address fields be given in network byte order. The `htonl` function reorders a `long` from the host's internal order to network byte order. Similarly, `htons` reorders a `short` to network byte order. The mirror functions `ntohl` and `ntohs` reorder integers from network byte order to host order.

18.7.3 The `listen` function

The `socket` function creates a communication endpoint, and `bind` associates this endpoint with a particular network address. At this point, a client can use the socket to connect to a server. To use the socket to accept incoming requests, an application must put the socket into the passive state by calling the `listen` function.

The `listen` function causes the underlying system network infrastructure to allocate queues to hold pending requests. When a client makes a connection request, the client and server network subsystems exchange messages (the TCP *three-way handshake*) to establish the connection. Since the server process may be busy, the host network subsystem queues the client connection requests until the server is ready to accept them. The client receives an `ECONNREFUSED` error if the server host refuses its connection request. The `socket` value is the descriptor returned by a previous call to `socket`, and the `backlog` parameter suggests a value for the maximum allowed number of pending client requests.

SYNOPSIS

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

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

POSIX

If successful, `listen` returns 0. If unsuccessful, `listen` returns –1 and sets `errno`. The following table lists the mandatory errors for `listen`.

`errno`	cause
`EBADF`	`socket` is not a valid file descriptor
`EDESTADDRREQ`	socket is not bound to a local address and protocol does not allow listening on an unbound socket
`EINVAL`	socket is already connected
`ENOTSOCK`	`socket` parameter does not refer to a socket
`EOPNOTSUPP`	socket protocol does not support `listen`

Traditionally, the `backlog` parameter has been given as 5. However, studies have shown [115] that the `backlog` parameter should be larger. Some systems incorporate a fudge factor in allocating queue sizes so that the actual queue size is larger than `backlog`. Exercise 22.14 explores the effect of `backlog` size on server performance.

18.7.4 Implementation of `u_open`

The combination of `socket`, `bind` and `listen` establishes a handle for the server to monitor communication requests from a well-known port. Program 18.6 shows the implementation of `u_open` in terms of these socket functions.

Program 18.6 **u_open.c**

A socket implementation of the UICI `u_open`.

```
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include "uici.h"

#define MAXBACKLOG 50

int u_ignore_sigpipe(void);

int u_open(u_port_t port) {
   int error;
   struct sockaddr_in server;
   int sock;
   int true = 1;

   if ((u_ignore_sigpipe() == -1) ||
        ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1))
      return -1;

   if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *)&true,
                  sizeof(true)) == -1) {
      error = errno;
      while ((close(sock) == -1) && (errno == EINTR));
      errno = error;
      return -1;
   }

   server.sin_family = AF_INET;
   server.sin_addr.s_addr = htonl(INADDR_ANY);
   server.sin_port = htons((short)port);
   if ((bind(sock, (struct sockaddr *)&server, sizeof(server)) == -1) ||
        (listen(sock, MAXBACKLOG) == -1)) {
      error = errno;
      while ((close(sock) == -1) && (errno == EINTR));
      errno = error;
      return -1;
   }
   return sock;
}
```

Program 18.6 **u_open.c**

If an attempt is made to write to a pipe or socket that no process has open for reading, write generates a SIGPIPE signal in addition to returning an error and setting errno to EPIPE. As with most signals, the default action of SIGPIPE terminates the process. Under no circumstances should the action of a client cause a server to terminate. Even if the server creates a child to handle the communication, the signal can prevent a graceful termination of the child when the remote host closes the connection. The socket implementation of UICI handles this problem by calling u_ignore_sigpipe to ignore the SIGPIPE signal if the default action of this signal is in effect.

The htonl and htons functions convert the address and port number fields to network byte order. The setsockopt call with SO_REUSEADDR permits the server to be restarted immediately, using the same port. This call should be made before bind.

If setsockopt, bind or listen produces an error, u_open saves the value of errno, closes the socket file descriptor, and restores the value of errno. Even if close changes errno, we still want to return with errno reporting the error that originally caused the return.

18.7.5 The accept function

After setting up a passive listening socket (socket, bind and listen), the server handles incoming client connections by calling accept. The parameters of accept are similar to those of bind. However, bind expects *address to be filled in before the call, so that it knows the port and interface on which the server will accept connection requests. In contrast, accept uses *address to return information about the client making the connection. In particular, the sin_addr member of the struct sockaddr_in structure contains a member, s_addr, that holds the Internet address of the client. The value of the *address_len parameter of accept specifies the size of the buffer pointed to by address. Before the call, fill this with the size of the *address structure. After the call, *address_len contains the number of bytes of the buffer actually filled in by the accept call.

SYNOPSIS

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

int accept(int socket, struct sockaddr *restrict address,
           socklen_t *restrict address_len);
```

POSIX

If successful, accept returns the nonnegative file descriptor corresponding to the accepted socket. If unsuccessful, accept returns –1 and sets errno. The following table lists the mandatory errors for accept.

errno	cause
EAGAIN or EWOULDBLOCK	O_NONBLOCK is set for socket file descriptor and no connections are present to be accepted
EBADF	socket parameter is not a valid file descriptor
ECONNABORTED	connection has been aborted
EINTR	accept interrupted by a signal that was caught before a valid connection arrived
EINVAL	socket is not accepting connections
EMFILE	OPEN_MAX file descriptors are currently open in calling process
ENFILE	maximum number of file descriptors in system are already open
ENOTSOCK	socket does not refer to a socket
EOPNOTSUPP	socket type of specified socket does not support the accepting of connections

■ Example 18.24

The following code segment illustrates how to restart accept if it is interrupted by a signal.

```
int len = sizeof(struct sockaddr);
int listenfd;
struct sockaddr_in netclient;
int retval;

while (((retval =
       accept(listenfd, (struct sockaddr *)(&netclient), &len)) == -1) &&
      (errno == EINTR))
   ;
if (retval == -1)
   perror("Failed to accept connection");
```

18.7.6 Implementation of u_accept

The u_accept function waits for a connection request from a client and returns a file descriptor that can be used to communicate with that client. It also fills in the name of the client host in a user-supplied buffer. The socket accept function returns information about the client in a struct sockaddr_in structure. The client's address is contained in this structure. The socket library does not have a facility to convert this binary address to a host name. UICI calls the addr2name function to do this conversion. This function takes as parameters a struct in_addr from a struct sockaddr_in, a buffer and the size of the buffer. It fills this buffer with the name of the host corresponding to the address given. The implementation of this function is discussed in Section 18.8.

Program 18.7 implements the UICI u_accept function. The socket accept call waits for a connection request and returns a communication file descriptor. If accept is interrupted by a signal, it returns –1 with errno set to EINTR. The UICI u_accept function reinitiates accept in this case. If accept is successful and the caller has furnished a hostn buffer, then u_accept calls addr2name to convert the address returned by accept to an ASCII host name.

Program 18.7 — **u_accept.c**

A socket implementation of the UICI u_accept *function.*

```
#include <errno.h>
#include <netdb.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include "uiciname.h"

int u_accept(int fd, char *hostn, int hostnsize) {
   int len = sizeof(struct sockaddr);
   struct sockaddr_in netclient;
   int retval;

   while (((retval =
           accept(fd, (struct sockaddr *)(&netclient), &len)) == -1) &&
          (errno == EINTR))
      ;
   if ((retval == -1) || (hostn == NULL) || (hostnsize <= 0))
      return retval;
   addr2name(netclient.sin_addr, hostn, hostnsize);
   return retval;
}
```

Program 18.7 — **u_accept.c**

☐ Exercise 18.25

Under what circumstances does u_accept return an error caused by client behavior?

Answer:

The conditions for u_accept to return an error are the same as for accept to return an error except for interruption by a signal. The u_accept function restarts accept when it is interrupted by a signal (e.g., errno is EINTR). The accept function may return an error for various system-dependent reasons related to insufficient resources. The accept function may also return an error if the client disconnects after the completion of the three-way handshake. A server that uses accept or u_accept should be careful not to simply exit on such an error. Even an error due to insufficient resources should not necessarily cause the server to exit, since the problem might be temporary.

18.7.7 The connect function

The client calls socket to set up a transmission endpoint and then uses connect to establish a link to the well-known port of the remote server. Fill the struct sockaddr structure as with bind.

SYNOPSIS

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

int connect(int socket, const struct sockaddr *address,
            socklen_t address_len);
```

POSIX

If successful, connect returns 0. If unsuccessful, connect returns –1 and sets errno. The following table lists the mandatory errors for connect that are applicable to all address families.

errno	cause
EADDRNOTAVAIL	specified address is not available from local machine
EAFNOSUPPORT	specified address is not a valid address for address family of specified socket
EALREADY	connection request already in progress on socket
EBADF	socket parameter not a valid file descriptor
ECONNREFUSED	target was not listening for connections or refused connection
EINPROGRSS	O_NONBLOCK set for file descriptor of the socket and connection cannot be immediately established, so connection shall be established asynchronously
EINTR	attempt to establish connection was interrupted by delivery of a signal that was caught, so connection shall be established asynchronously
EISCONN	specified socket is connection mode and already connected
ENETUNREACH	no route to network is present
ENOTSOCK	socket parameter does not refer to a socket
EPROTOTYPE	specified address has different type than socket bound to specified peer address
ETIMEDOUT	attempt to connect timed out before connection made

18.7.8 Implementation of u_connect

Program 18.8 shows u_connect, a function that initiates a connection request to a server. The u_connect function has two parameters, a port number (port) and a host name (hostn), which together specify the server to connect to.

Program 18.8 **u_connect.c**

A socket implementation of the UICI `u_connect` *function.*

```
#include <ctype.h>
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/types.h>
#include "uiciname.h"
#include "uici.h"

int u_ignore_sigpipe(void);

int u_connect(u_port_t port, char *hostn) {
   int error;
   int retval;
   struct sockaddr_in server;
   int sock;
   fd_set sockset;

   if (name2addr(hostn,&(server.sin_addr.s_addr)) == -1) {
      errno = EINVAL;
      return -1;
   }
   server.sin_port = htons((short)port);
   server.sin_family = AF_INET;

   if ((u_ignore_sigpipe() == -1) ||
        ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1))
      return -1;

   if (((retval =
       connect(sock, (struct sockaddr *)&server, sizeof(server))) == -1) &&
       ((errno == EINTR) || (errno == EALREADY))) {           /* asynchronous */
       FD_ZERO(&sockset);
       FD_SET(sock, &sockset);
       while (((retval = select(sock+1, NULL, &sockset, NULL, NULL)) == -1)
           && (errno == EINTR)) {
          FD_ZERO(&sockset);
          FD_SET(sock, &sockset);
       }
   }
   if (retval == -1) {
        error = errno;
        while ((close(sock) == -1) && (errno == EINTR));
        errno = error;
        return -1;
   }
   return sock;
}
```

Program 18.8 **u_connect.c**

The first step is to verify that hostn is a valid host name and to find the corresponding IP address using name2addr. The u_connect function stores this address in a struct sockaddr_in structure. The name2addr function, which takes a string and a pointer to in_addr_t as parameters, converts the host name stored in the string parameter into a binary address and stores this address in the location corresponding to its second parameter. Section 18.8 discusses the implementation of name2addr.

If the SIGPIPE signal has the default signal handler, u_ignore_sigpipe sets SIGPIPE to be ignored. (Otherwise, the client terminates when it tries to write after the remote end has been closed.) The u_connect function then creates a SOCK_STREAM socket. If any of these steps fails, u_connect returns an error.

The connect call can be interrupted by a signal. However, unlike other library functions that set errno to EINTR, connect should not be restarted, because the network subsystem has already initiated the TCP 3-way handshake. In this case, the connection request completes asynchronously to program execution. The application must call select or poll to detect that the descriptor is ready for writing. The UICI implementation of u_connect uses select and restarts it if interrupted by a signal.

☐ Exercise 18.26

How would the behavior of u_connect change if

```
if ((u_ignore_sigpipe() != 0) ||
     ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1))
    return -1;
```

were replaced by the following?

```
if (((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) ||
    (u_ignore_sigpipe() != 0) )
    return -1;
```

Answer:

If u_ignore_sigpipe() fails, u_connect returns with an open file descriptor in sock. Since the calling program does not have the value of sock, this file descriptor could not be closed.

☐ Exercise 18.27

Does u_connect ever return an error if interrupted by a signal?

Answer:

To determine the overall behavior of u_connect, we must analyze the response of each call within u_connect to a signal. The u_ignore_sigpipe code of Appendix C only contains a sigaction call, which does not return an error when interrupted by a signal. The socket call does not return an EINTR error, implying that it either restarts itself or blocks signals. Also, name2addr does not return EINTR. An arriving signal is handled, ignored or blocked and the program continues (unless of course a handler terminates the program). The connect call can

return if interrupted by a signal, but the implementation then calls `select` to wait for asynchronous completion. The `u_connect` function also restarts `select` if it is interrupted by a signal. Thus, `u_connect` should never return because of interruption by a signal.

18.8 Host Names and IP Addresses

Throughout this book we refer to hosts by name (e.g., `usp.cs.utsa.edu`) rather than by a numeric identifier. Host names must be mapped into numeric network addresses for most of the network library calls. As part of system setup, system administrators define the mechanism by which names are translated into network addresses. The mechanism might include local table lookup, followed by inquiry to domain name servers if necessary. The Domain Name Service (DNS) is the glue that integrates naming on the Internet [81, 82].

In general, a host machine can be specified either by its name or by its address. Host names in programs are usually represented by ASCII strings. IPv4 addresses are specified either in binary (in network byte order as in the `s_addr` field of `struct in_addr`) or in a human readable form, called the *dotted-decimal notation* or *Internet address dot notation*. The dotted form of an address is a string with the values of the four bytes in decimal, separated by decimal points. For example, 129.115.30.129 might be the address of the host with name `usp.cs.utsa.edu`. The binary form of an IPv4 address is 4 bytes long. Since 4-byte addresses do not provide enough room for future Internet expansion, a newer version of the protocol, IPv6, uses 16-byte addresses.

The `inet_addr` and `inet_ntoa` functions convert between dotted-decimal notation and the binary network byte order form used in the `struct in_addr` field of a `struct sockaddr_in`.

The `inet_addr` function converts a dotted-decimal notation address to binary in network byte order. The value can be stored directly in the `sin_addr.s_addr` field of a `struct sockaddr_in`.

SYNOPSIS

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

in_addr_t inet_addr(const char *cp);
```

POSIX

If successful, `inet_addr` returns the Internet address. If unsuccessful, `inet_addr` returns `(in_addr_t)`–1. No errors are defined for `inet_addr`.

The `inet_ntoa` function takes a `struct in_addr` structure containing a binary address in network byte order and returns the corresponding string in dotted-decimal notation. The binary address can come from the `sin_addr` field of a `struct sockaddr_in` structure. The returned string is statically allocated, so `inet_ntoa` may not be safe to use

in threaded applications. Copy the returned string to a different location before calling inet_ntoa again. Check the man page for inet_ntoa on your system to see if it is thread-safe.

SYNOPSIS

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

char *inet_ntoa(const struct in_addr in);
```

POSIX

The inet_ntoa function returns a pointer to the network address in Internet standard dot notation. No errors are defined for inet_ntoa.

The different data types used for the binary form of an address often cause confusion. The inet_ntoa function, takes a struct in_addr structure as a parameter; the inet_addr returns data of type in_addr_t, a field of a struct in_addr structure. POSIX states that a struct in_addr structure must contain a field called s_addr of type in_addr_t. It is implied that the binary address is stored in s_addr and that a struct in_addr structure may contain other fields, although none are specified. It seems that in most current implementations, the struct in_addr structure contains only the s_addr field, so pointers to sin_addr and sin_addr.s_addr are identical. To maintain future code portability, however, be sure to preserve the distinction between these two structures.

At least three collections of library functions convert between ASCII host names and binary addresses. None of these collections report errors in the way UNIX functions do by returning –1 and setting errno. Each collection has advantages and disadvantages, and at the current time none of them stands out as the best method.

UICI introduces the addr2name and name2addr functions to abstract the conversion between strings and binary addresses and allow for easy porting between implementations. The uiciname.h header file shown in Program C.3 contains the following prototypes for addr2name and name2addr.

```
int name2addr(const char *name, in_addr_t *addrp);
void addr2name(struct in_addr addr, char *name, int namelen);
```

Link uiciname.c with any program that uses UICI.

The name2addr function behaves like inet_addr except that its parameter can be either a host name or an address in dotted-decimal format. Instead of returning the address, name2addr stores the address in the location pointed to by addrp to allow the return value to report an error. If successful, name2addr returns 0. If unsuccessful, name2addr returns –1. An error occurs if the system cannot determine the address corresponding to the given name. The name2addr function does not set errno. We suggest that when name2addr is called by a function that must return with errno set, the value EINVAL be used to indicate failure.

The addr2name function takes a struct in_addr structure as its first parameter and writes the corresponding name to the supplied buffer, name. The namelen value specifies the size of the name buffer. If the host name does not fit in name, addr2name copies the first namelen - 1 characters of the host name followed by a string terminator. This function never produces an error. If the host name cannot be found, addr2name converts the host address to dotted-decimal notation.

We next discuss two possible strategies for implementing name2addr and addr2name. Section 18.9 discusses two additional implementations. Appendix C presents complete implementations using all four approaches. Setting the constant REENTRANCY in uiciname.c picks out a particular implementation. We first describe the default implementation that uses gethostbyname and gethostbyaddr.

A traditional way of converting a host name to a binary address is with the gethostbyname function. The gethostbyname function takes a host name string as a parameter and returns a pointer to a struct hostent structure containing information about the names and addresses of the corresponding host.

SYNOPSIS

```
#include <netdb.h>

struct hostent {
   char    *h_name;          /* canonical name of host */
   char    **h_aliases;      /* alias list */
   int     h_addrtype;       /* host address type */
   int     h_length;         /* length of address */
   char    **h_addr_list;    /* list of addresses */
};

struct hostent *gethostbyname(const char *name);
```

POSIX:OB

If successful, gethostbyname returns a pointer to a struct hostent. If unsuccessful, gethostbyname returns a NULL pointer and sets h_errno. Macros are available to produce an error message from an h_errno value. The following table lists the mandatory errors for gethostbyname.

h_errno	cause
HOST_NOT_FOUND	no such host
NO_DATA	server recognized request and name but has no address
NO_RECOVERY	unexpected server failure that cannot be recovered
TRY_AGAIN	temporary or transient error

The struct hostent structure includes two members of interest that are filled in by gethostbyname. The h_addr_list field is an array of pointers to network addresses used by this host. These addresses are in network byte order, so they can be used directly

in the address structures required by the socket calls. Usually, we use only the first entry, h_addr_list[0]. The integer member h_length is filled with the number of bytes in the address. For IPv4, h_length should always be 4.

■ Example 18.28

The following code segment translates a host name into an IP address for the s_addr member of a struct sockaddr_in.

```
char *hostn = "usp.cs.utsa.edu";
struct hostent *hp;
struct sockaddr_in server;

if ((hp = gethostbyname(hostn)) == NULL)
   fprintf(stderr, "Failed to resolve host name\n");
else
   memcpy((char *)&server.sin_addr.s_addr, hp->h_addr_list[0], hp->h_length);
```

Often, a host has multiple names associated with it. For example, because usp.cs.utsa.edu is a web server for this book, the system also responds to the alias www.usp.cs.utsa.edu.

□ Exercise 18.29

Use the struct hostent structure returned in Example 18.28 to output a list of aliases for usp.cs.utsa.edu.

Answer:

```
char **q;
struct hostent *hp;

for (q = hp->h_aliases; *q != NULL; q++)
   (void) printf("%s\n", *q);
```

□ Exercise 18.30

Use the struct hostent structure returned in Example 18.28 to find out how many IP addresses are associated with usp.cs.utsa.edu.

Answer:

```
int addresscount = 0;
struct hostent *hp;
char **q;

for (q = hp->h_addr_list; *q != NULL; q++)
   addresscount++;
printf("Host %s has %d IP addresses\n", hp->h_name, addresscount);
```

Program 18.9 is one implementation of name2addr. The name2addr function first checks to see if name begins with a digit. If so, name2addr assumes that name is a dotted-decimal address and uses inet_addr to convert it to in_addr_t. Otherwise, name2addr uses gethostbyname.

Program 18.9 — **name2addr_gethostbyname.c**

An implementation of `name2addr` *using* `gethostbyname`.

```
#include <ctype.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>

int name2addr(char *name, in_addr_t *addrp) {
   struct hostent *hp;

   if (isdigit((int)(*name)))
      *addrp = inet_addr(name);
   else {
      hp = gethostbyname(name);
      if (hp == NULL)
         return -1;
      memcpy((char *)addrp, hp->h_addr_list[0], hp->h_length);
   }
   return 0;
}
```

Program 18.9 — **name2addr_gethostbyname.c**

The conversion from address to name can be done with `gethostbyaddr`. For IPv4, the `type` should be `AF_INET` and the `len` value should be 4 bytes. The `addr` parameter should point to a `struct in_addr` structure.

SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostbyaddr(const void *addr,
                              socklen_t len, int type);
```

POSIX:OB

If successful, `gethostbyaddr` returns a pointer to a `struct hostent` structure. If unsuccessful, `gethostbyaddr` returns a `NULL` pointer and sets `h_error`. The mandatory errors for `gethostbyaddr` are the same as those for `gethostbyname`.

■ Example 18.31

The following code segment prints the host name from a previously set `struct sockaddr_in` structure.

```
struct hostent *hp;
struct sockaddr_in net;
int sock;

if (( hp = gethostbyaddr(&net.sin_addr, 4, AF_INET))
   printf("Host name is %s\n", hp->h_name);
```

Program 18.10 is an implementation of the addr2name function that uses the gethostbyaddr function. If gethostbyaddr returns an error, then addr2name uses inet_ntoa to convert the address to dotted-decimal notation. The addr2name function copies at most namelen-1 bytes, allowing space for the string terminator.

Program 18.10 — **addr2name_gethostbyaddr.c**

An implementation of addr2name *using* gethostbyaddr.

```
#include <ctype.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>

void addr2name(struct in_addr addr, char *name, int namelen) {
   struct hostent *hostptr;
   hostptr = gethostbyaddr((char *)&addr, 4, AF_INET);
   if (hostptr == NULL)
      strncpy(name, inet_ntoa(addr), namelen-1);
   else
      strncpy(name, hostptr->h_name, namelen-1);
   name[namelen-1] = 0;
}
```

Program 18.10 — **addr2name_gethostbyaddr.c**

When an error occurs, gethostbyname and gethostbyaddr return NULL and set h_errno to indicate an error. Thus, errno and perror cannot be used to display the correct error message. Also, gethostbyname and gethostbyaddr are not thread-safe because they use static data for storing the returned struct hostent. They should not be used in threaded programs without appropriate precautions being taken. (See Section 18.9.) A given implementation might use the same static data for both of these, so be careful to copy the result before it is modified.

A second method for converting between host names and addresses, getnameinfo and getaddrinfo, first entered an approved POSIX standard in 2001. These general functions, which can be used with both IPv4 and IPv6, are preferable to gethostbyname and gethostbyaddr because they do not use static data. Instead, getnameinfo stores the name in a user-supplied buffer, and getaddrinfo dynamically allocates a buffer to return with the address information. The user can free this buffer with freeaddrinfo. These functions are safe to use in a threaded environment. The only drawback in using these functions, other than the complication of the new structures used, is that they are not yet available on many systems.

SYNOPSIS

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

void freeaddrinfo(struct addrinfo *ai);
int getaddrinfo(const char *restrict nodename,
                const char *restrict servname,
                const struct addrinfo *restrict hints,
                struct addrinfo **restrict res);
int getnameinfo(const struct sockaddr *restrict sa,
                socklen_t  salen, char *restrict node,
                socklen_t nodelen, char *restrict service,
                socklen_t servicelen, unsigned flags);
```

POSIX

If successful, `getaddrinfo` and `getnameinfo` return 0. If unsuccessful, these functions return an error code. The following table lists the mandatory error codes for `getaddrinfo` and `getnameinfo`.

error	cause
EAI_AGAIN	name cannot be resolved at this time
EAI_BADFLAGS	`flags` had an invalid value
EAI_FAIL	unrecoverable error
EAI_FAMILY	address family was not recognized or address length invalid for specified family
EAI_MEMORY	memory allocation failure
EAI_NONAME	name does not resolve for supplied parameters
EAI_SERVICE	service passed not recognized for socket (`getaddrinfo`)
EAI_SOCKTYPE	intended socket type not recognized (`getaddrinfo`)
EAI_SYSTEM	a system error occurred and error code can be found in `errno`
EAI_OVERFLOW	argument buffer overflow (`getaddrinfo`)

The `struct addrinfo` structure contains at least the following members.

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

The user passes the name of the host in the `nodename` parameter of `getaddrinfo`. The `servname` parameter can contain a service name (in IPv6) or a port number. For our purposes, the `nodename` determines the address, and the `servname` parameter can be a `NULL` pointer. The `hints` parameter tells `getaddrinfo` what type of addresses the caller is interested in. For IPv4, we set `ai_flags` to 0. In this case, `ai_family`, `ai_socktype`

and ai_protocol are the same as in socket. The ai_addrlen parameter can be set to 0, and the remaining pointers can be set to NULL. The getaddrinfo function, using the res parameter, returns a linked list of struct addrinfo nodes that it dynamically allocates to contain the address information. When finished using this linked list, call freeaddrinfo to free the nodes.

Program 18.11 shows an implementation of name2addr that uses getaddrinfo. After calling getaddrinfo, the function copies the address and frees the memory that was allocated.

Program 18.11 **name2addr_getaddrinfo.c**

An implementation of name2addr *using* getaddrinfo.

```
#include <ctype.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>

int name2addr(char *name, in_addr_t *addrp) {
   struct addrinfo hints;
   struct addrinfo *res;
   struct sockaddr_in *saddrp;

   hints.ai_flags = 0;
   hints.ai_family = PF_INET;
   hints.ai_socktype = SOCK_STREAM;
   hints.ai_protocol = 0;
   hints.ai_addrlen = 0;
   hints.ai_canonname = NULL;
   hints.ai_addr = NULL;
   hints.ai_next = NULL;

   if (getaddrinfo(name,NULL,&hints,&res) != 0)
      return -1;

   saddrp = (struct sockaddr_in *)(res->ai_addr);
   memcpy(addrp, &saddrp->sin_addr.s_addr, 4);
   freeaddrinfo(res);
   return 0;
}
```

Program 18.11 **name2addr_getaddrinfo.c**

To use getnameinfo to convert an address to a name, pass a pointer to a sockaddr_in structure in the first parameter and its length in the second parameter. Supply a buffer to hold the name of the host as the third parameter and the size of that buffer as the fourth parameter. Since we are not interested in the service name, the fifth parameter can be NULL and the sixth parameter can be 0. The last parameter is for flags, and it can be 0, causing the fully qualified domain name to be returned. The

sin_family field of the sockaddr_in should be AF_INET, and the sin_addr field contains the addresses. If the name cannot be determined, the numeric form of the host name is returned, that is, the dotted-decimal form of the address.

Program 18.12 shows an implementation of addr2name. The addr2name function never returns an error. Instead, it calls inet_ntoa if getnameinfo produces an error.

Program 18.12 **addr2name_getnameinfo.c**

An implementation of addr2name *using* getnameinfo.

```
#include <ctype.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>

void addr2name(struct in_addr addr, char *name, int namelen) {
   struct sockaddr_in saddr;
   saddr.sin_family = AF_INET;
   saddr.sin_port = 0;
   saddr.sin_addr = addr;
   if (getnameinfo((struct sockaddr *)&saddr, sizeof(saddr), name, namelen,
         NULL, 0, 0) != 0) {
      strncpy(name, inet_ntoa(addr), namelen-1);
      name[namelen-1] = 0;
   }
}
```

Program 18.12 **addr2name_getnameinfo.c**

18.9 Thread-Safe UICI

The UNIX functions that use errno were originally unsafe for threads. When errno was an external integer shared by all threads, one thread could set errno and have another thread change it before the first thread used the value. Multithreaded systems solve this problem by using thread-specific data for errno, thus preserving the syntax for the standard UNIX library functions. This same problem exists with any function that returns values in variables with static storage class.

The TCP socket implementation of UICI in Section 18.7 is thread-safe provided that the underlying implementations of socket, bind, listen, accept, connect, read, write and close are thread-safe and that the name resolution is thread-safe. The POSIX standard states that all functions defined by POSIX and the C standard are thread-safe, except the ones shown in Table 12.2 on page 432. The list is short and mainly includes functions, such as strtok and ctime, that require the use of static data.

The gethostbyname, gethostbyaddr and inet_ntoa functions, which are used in some versions of UICI name resolution, appear on the POSIX list of functions that might

not be thread-safe. Some implementations of inet_ntoa (such as that of Sun Solaris) are thread-safe because they use thread-specific data. These possibly unsafe functions are used only in name2addr and addr2name, so the issue of thread safety of UICI is reduced to whether these functions are thread-safe.

Since getnameinfo and getaddrinfo are thread-safe, then if inet_ntoa is thread-safe, the implementations of name2addr and addr2name that use these are also thread-safe. Unfortunately, as stated earlier, getnameinfo and getaddrinfo are not yet available on many systems.

On some systems, thread-safe versions of gethostbyname and gethostbyaddr, called gethostbyname_r and gethostbyaddr_r, are available.

SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostbyname_r(const char *name,
     struct hostent *result, char *buffer, int buflen,
     int *h_errnop);
struct hostent *gethostbyaddr_r(const char *addr,
     int length, int type, struct hostent *result,
     char *buffer,  int buflen, int *h_errnop);
```

These functions perform the same tasks as their unsafe counterparts but do not use static storage. The user supplies a pointer to a struct hostent in the result parameter. Pointers in this structure point into the user-supplied buffer, which has length buflen. The supplied buffer array must be large enough for the generated data. When the gethostbyname_r and gethostbyaddr_r functions return NULL, they supply an error code in the integer pointed to by *h_errnop. Program 18.13 shows a thread-safe implementation of addr2name, assuming that inet_ntoa is thread-safe. Section C.2.2 contains a complete implementation of UICI, using gethostbyname_r and gethostbyaddress_r.

Unfortunately, gethostbyname_r and gethostbyaddress_r were part of the X/OPEN standard, but when this standard was merged with POSIX, these functions were omitted. Another problem associated with Program 18.13 is that it does not specify how large the user-supplied buffer should be. Stevens [115] suggests 8192 for this value, since that is what is commonly used in the implementations of the traditional forms.

An alternative for enforcing thread safety is to protect the sections that use static storage with mutual exclusion. POSIX:THR mutex locks provide a simple method of doing this. Program 18.14 is an implementation of addr2name that uses mutex locks. Section C.2.3 contains a complete implementation of UICI using mutex locks. This implementation does not require inet_ntoa to be thread-safe, since its static storage is protected also.

Program 18.13 — **addr2name_gethostbyaddr_r.c**

A version of `addr2name` *using* `gethostbyaddr_r`.

```
#include <ctype.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#define GETHOST_BUFSIZE 8192

void addr2name(struct in_addr addr, char *name, int namelen) {
   char buf[GETHOST_BUFSIZE];
   int h_error;
   struct hostent *hp;
   struct hostent result;

   hp = gethostbyaddr_r((char *)&addr, 4, AF_INET, &result, buf,
                        GETHOST_BUFSIZE, &h_error);
   if (hp == NULL)
      strncpy(name, inet_ntoa(addr), namelen-1);
   else
      strncpy(name, hp->h_name, namelen-1);
   name[namelen-1] = 0;
}
```

Program 18.13 — **addr2name_gethostbyaddr_r.c**

Program 18.14 — **addr2name_mutex.c**

A thread-safe version of `addr2name` *using POSIX mutex locks.*

```
#include <ctype.h>
#include <netdb.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void addr2name(struct in_addr addr, char *name, int namelen) {
   struct hostent *hostptr;

   pthread_mutex_lock(&mutex);
   hostptr = gethostbyaddr((char *)&addr, 4, AF_INET);
   if (hostptr == NULL)
      strncpy(name, inet_ntoa(addr), namelen-1);
   else
      strncpy(name, hostptr->h_name, namelen-1);
   pthread_mutex_unlock(&mutex);
   name[namelen-1] = 0;
}
```

Program 18.14 — **addr2name_mutex.c**

18.10 Exercise: Ping Server

The ping command can be used to elicit a response from a remote host. The default for some systems is to just display a message signifying that the host responded. On other systems the default is to indicate how long it took for a reply to be received.

■ Example 18.32

The following command queries the usp.cs.utsa.edu host.

```
ping usp.cs.utsa.edu
```

The command might output the following message to mean that the host usp.cs.utsa.edu is responding to network communication.

```
usp.cs.utsa.edu is alive
```

This section describes an exercise that uses UICI to implement myping, a slightly fancier version of the ping service. The myping function responds with a message such as the following.

```
usp.cs.utsa.edu: 5:45am up 12:11, 2 users, load average: 0.14, 0.08, 0.07
```

The myping program is a client-server application. A myping server running on the host listens at a well-known port for client requests. The server forks a child to respond to the request. The original server process continues listening. Assume that the myping well-known port number is defined by the constant MYPINGPORT.

Write the code for the myping client. The client takes the host name as a command-line argument, makes a connection to the port specified by MYPINGPORT, reads what comes in on the connection and echoes it to standard output until end-of-file, closes the connection, and exits. Assume that if the connection attempt to the host fails, the client sleeps for SLEEPTIME seconds and then retries. After the number of failed connection attempts exceeds RETRIES, the client outputs the message that the host is not available and exits. Test the program by using the bidirectional server discussed in Example 18.18.

Implement the myping server. The server listens for connections on MYPINGPORT. If a client makes a connection, the server forks a child to handle the request and the original process resumes listening at MYPINGPORT. The child closes the listening file descriptor, calls the process_ping function, closes the communication file descriptor, and exits.

Write a process_ping function with the following prototype.

```
int process_ping(int communfd);
```

For initial testing, process_ping can just output an error message to the communication file descriptor. For the final implementation, process_ping should construct a message consisting of the host name and the output of the uptime command. An example message is as follows.

```
usp.cs.utsa.edu: 5:45am up 13:11, 2 users, load average: 0.14, 0.08, 0.07
```

Use `uname` to get the host name.

SYNOPSIS

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

int uname(struct utsname *name);
```

POSIX

If successful, `uname` returns a nonnegative value. If unsuccessful, `uname` returns –1 and sets `errno`. No mandatory errors are defined for `uname`.

The `struct utsname` structure, which is defined in `sys/utsname.h`, has at least the following members.

```
char sysname[];   /* name of this OS implementation */
char nodenamep[]; /* name of this node within communication network */
char release[];   /* current release level of this implementation */
char version[];   /* current version level of this release */
char machine[];   /* name of hardware type on which system is running */
```

18.11 Exercise: Transmission of Audio

This section extends the UICI server and client of Program 18.1 and Program 18.3 to send audio information from the client to the server. These programs can be used to implement a network intercom, network telephone service, or network radio broadcasts, as described in Chapter 21.

Start by incorporating audio into the UICI server and client as follows.

- Run Programs 18.1 and 18.3 with redirected input and output to transfer files from client to server, and vice versa. Use `diff` to verify that each transfer completes correctly.
- Redirect the input to the client to come from the audio device (microphone) and redirect the output on the server to go to the audio device (speakers). You should be able to send audio across the network. (See Section 6.6 for information on how to do this.)
- Modify the bidirectional server and client to call the audio functions developed in Section 6.6 and Section 6.7 to transmit audio from the microphone of the client to the speaker of the server. Test your program for two-way communication.

The program sends even if no one is talking because once the program opens the audio device, the underlying device driver and interface card sample the audio input at a fixed rate until the program closes the file. The continuous sampling produces a prohibitive

amount of data for transmission across the network. Use a filter to detect whether a packet contains voice, and throw away audio packets that contain no voice. A simple method of filtering is to convert the u-law (μ-law) data to a linear scale and reject packets that fall below a threshold. Program 18.15 shows an implementation of this filter for Solaris. The `hasvoice` function returns 1 if the packet contains voice and 0 if it should be thrown away. Incorporate `hasvoice` or another filter so that the client does not transmit silence.

Program 18.15 — **hasvoice.c**

A simple threshold function for filtering data with no voice.

```
#include <stdio.h>
#include <stdlib.h>
#include "/usr/demo/SOUND/include/multimedia/audio_encode.h"
#define THRESHOLD 20   /* amplitude of ambient room noise, linear PCM */

               /* return 1 if anything in audiobuf is above THRESHOLD */
int hasvoice(char *audiobuf, int length) {
   int i;

   for (i = 0; i < length; i++)
      if (abs(audio_u2c(audiobuf[i])) > THRESHOLD)
         return 1;
   return 0;
}
```

Program 18.15 — **hasvoice.c**

Write the following enhancements to the basic audio transmission service.

1. Develop a calibration function that allows the threshold for voice detection to be adjusted according to the current value of the ambient room noise.
2. Use more sophisticated filtering algorithms in place of simple thresholds.
3. Keep track of the total number of packets and the actual number of those that contain voice data. Display the information on standard error when the client receives a `SIGUSR1` signal.
4. Add volume control options on both client and server sides.
5. Design an interface for accepting or rejecting connections in accordance with sender information.
6. Devise protocols analogous to caller ID and call-waiting.
7. Add an option on the server side to record the incoming audio to a file for later playback. Recording is easy if the client is sending all the packets. However, since the client is sending only packets with voice, straight recording does not sound right on playback because all silences are compressed. Keep timing information as well as the audio information in the recorded data.

18.12 Additional Reading

Computer Networks, 4th ed. by Tanenbaum [123] is a standard reference on computer networks. The three-volume set *TCP/IP Illustrated* by Stevens and Wright [113, 134, 114] provides details of the TCP/IP protocol and its implementation. The two volumes of *UNIX Network Programming* by Stevens [115, 116] are the most comprehensive references on UNIX network programming. *UNIX System V Network Programming* by Rago [92] is an excellent reference book on network programming under System V. The standard for network services was incorporated into POSIX in 2001 [49].