The BitTorrent protocol tries to provide a balance between the number of pieces each peer may have at each moment by using a strategy called the *rarest-first*. Using this strategy, a peer tries to first download the pieces with the fewest repeated copies among the neighbors. In this way, these pieces are circulated faster.

Trackerless BitTorrent

In the original BitTorrent design, if the tracker fails, new peers cannot connect to the network and updating is interrupted. There are several implementations of BitTorrent that eliminate the need for a centralized tracker. In the implementation that we describe here, the protocol still uses the tracker, but not a central one. The job of tracking is distributed among some nodes in the network. In this section, we show how Kademlia DHT can be used to achieve this goal, but we avoid becoming involved in the details of a specific protocol.

In BitTorrent with a central tracker, the job of the tracker is to provide the list of peers in a swarm when given a metadata file that defines the torrent. If we think of the hash function of metadata as the key and the hash function of the list of peers in a swarm as the value, we can let some nodes in a P2P network play the role of trackers. A new peer that joins the torrent sends the hash function of the metadata (key) to the node that it knows. The P2P network uses Kademlia protocol to find the node responsible for the key. The responsible node sends the *value*, which is actually the list of peers in the corresponding torrent, to the joining peer. Now the joining peer can use the BitTorrent protocol to share the content file with peers in the list.

2.5 SOCKET INTERFACE PROGRAMMING

In Section 2.2, we discussed the principle of the client-server paradigm. In Section 2.3, we discussed some standard applications using this paradigm. In this section, we show how to write some simple client-server programs using C, a procedural programming language. We chose the C language in this section for two reasons. First, socket programming traditionally started in the C language. Second, the low-level feature of the C language better reveals some subtleties in this type of programming. In Chapter 11, we expand this idea in Java, which provides a more compact version. However, this section can be skipped without loss of continuity in the study of the book.

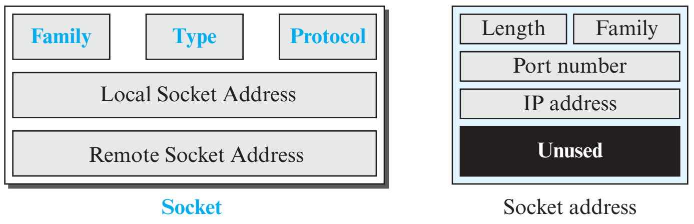
2.5.1 Socket Interface in C

We discussed socket interface in Section 2.2. In this section, we show how this interface is implemented in the C language. The important issue in socket interface is to understand the role of a socket in communication. The socket has no buffer to store data to be sent or received. It is capable of neither sending nor receiving data. The socket just acts as a reference or a label. The buffers and necessary variables are created inside the operating system.

Data Structure for Socket

The C language defines a socket as a structure (struct). The socket structure is made of five fields; each socket address itself is a structure made of five fields, as shown in Figure 2.58. Note that the programmer should not redefine this structure; it is already defined in the header files. We briefly discuss the five fields in a socket structure.

Figure 2.58 *Socket data structure*



- **Family.** This field defines the family protocol (how to interpret the addresses and port number). The common values are PF_INET (for current Internet), PF_INET6 (for next-generation Internet), and so on. We use PF_INET for this section.
- **Type.** This field defines four types of sockets: SOCK_STREAM (for TCP), SOCK_DGRAM (for UDP), SOCK_SEQPACKET (for SCTP), and SOCK_RAW (for applications that directly use the services of IP).
- **Protocol.** This field defines the specific protocol in the family. It is set to 0 for TCP/IP protocol suite because it is the only protocol in the family.
- **Local socket address.** This field defines the *local socket address*. A socket address is itself a structure made of the *length* field, the *family* field (which is set to the constant AF_INET for TCP/IP protocol suite), the port number field (which defines the process), and the IP address field (which defines the host on which the process is running). It also contains an unused field.
- **Remote socket address.** This field defines the remote socket address. Its structure is the same as the local socket address.

Header Files

To be able to use the definition of the socket and all procedures (functions) defined in the interface, we need a set of header files. We have collected all of these header files in a file named *headerFiles.h*. This file needs to be created in the same directory as the programs and its name should be included in all programs.

```
// "headerFiles.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <arpa/innet.h>
#include <sys/wait.h>
```

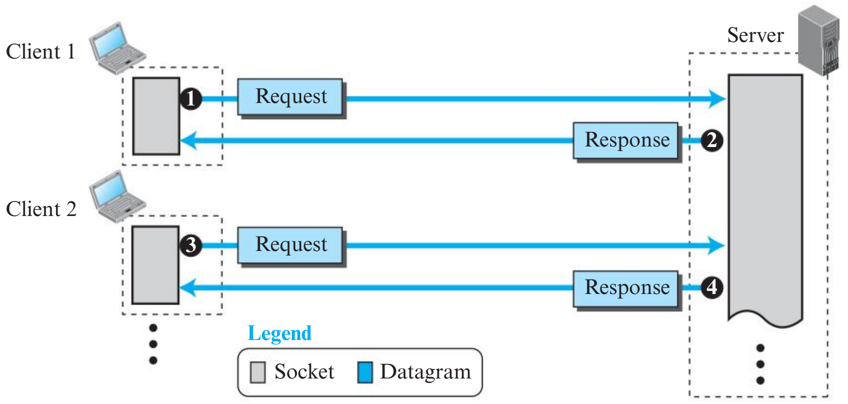
Iterative Communication Using UDP

As we discussed earlier, UDP provides a connectionless server, in which a client sends a request and the server sends back a response.

Sockets Used for UDP

In UDP communication, the client and server use only one socket each. The socket created at the server site lasts forever; the socket created at the client site is closed (destroyed) when the client process terminates. Figure 2.59 shows the lifetime of the sockets in the server and client processes. In other words, different clients use different sockets, but the server creates only one socket and changes only the remote socket address each time a new client makes a connection. This is logical, because the server does know its own socket address, but does not know the socket address of the clients who need its server; it needs to wait for the client to connect before filling this part of the socket.

Figure 2.59 *Sockets for UDP communication*

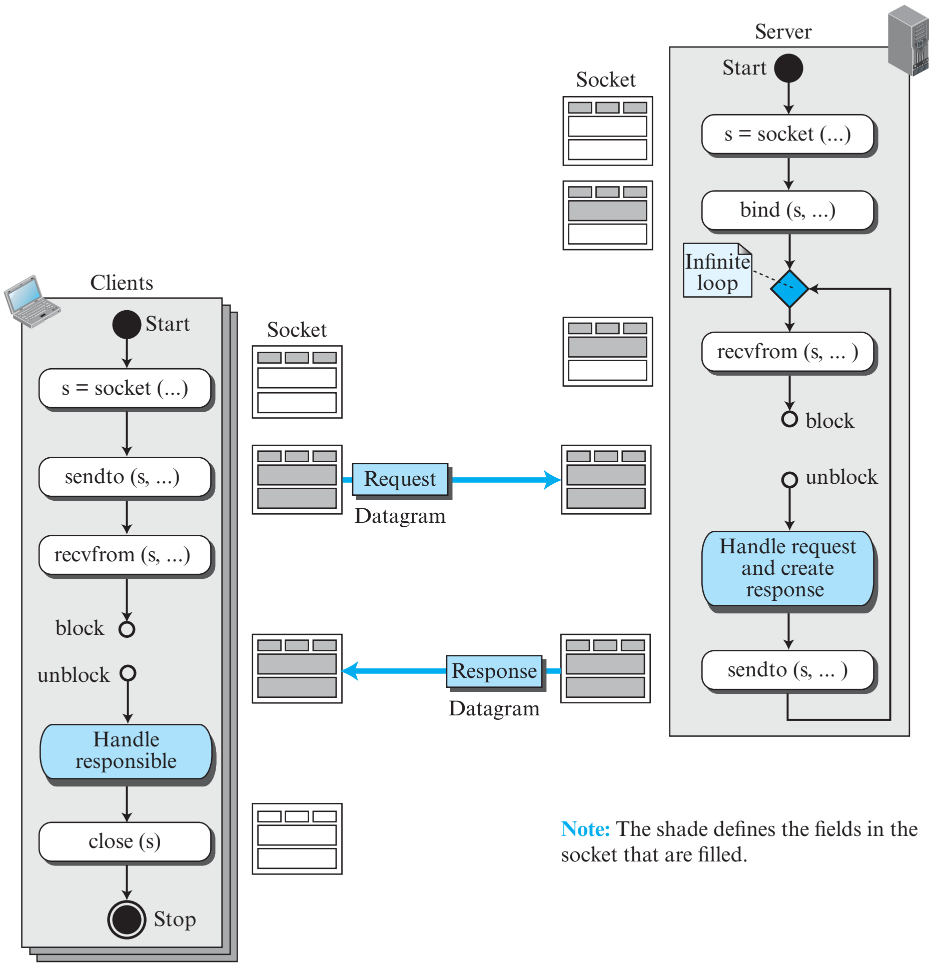


Communication Flow Diagram

Figure 2.60 shows a simplified flow diagram for iterative communication. There are multiple clients, but only one server. Each client is served in each iteration of the loop in the server. Note that there are no connection establishment and connection termination. Each client sends one single datagram and receives one single datagram. In other words, if a client wants to send two datagrams, it is considered as two clients for the server. The second datagram needs to wait for its turn.

Server Process The server makes a *passive open*, in which it becomes ready for the communication, but it waits until a client process makes the connection. It calls the *socket* procedure to create a socket. The arguments in this procedure call fill the first three fields, but the local and remote socket address fields are still undefined. The server process then calls the *bind* procedure to fill the local socket address field (information comes from the operating system). It then calls another procedure, called *recvfrom*. This procedure, however, blocks the server process until a client datagram arrives. When a datagram arrives, the server process unblocks and extracts the request from the datagram. It also

Figure 2.60 *Flow diagram for iterative UDP communication*



Note: The shade defines the fields in the socket that are filled.

extracts the sender socket address to be used in the next step. After the request is processed and the response is ready, the server process completes the socket structure by filling the remote socket address field with the sender socket address in the received datagram. Now the datagram is ready to be sent. This is done by calling another procedure, called *sendto*. Note that all the fields in the socket should be filled before the server process can send the response. After sending the response, the server process starts a new iteration and waits for another client to connect. The remote socket address field will be refilled again with the address of a new client (or the same client considered as a new one). The server process is an infinite process; it runs forever. The server socket is never closed unless there is a problem and the process needs to be aborted.

Client Process The client process makes an *active open*. In other words, it starts a connection. It calls the *socket* procedure to create a socket and fill the first three fields.

Although some implementations require that the client process also calls the *bind* procedure to fill the local socket address, normally this is done automatically by the operating system, which selects a temporary port number for the client. The client process then calls the *sendto* procedure and provides the information about the remote socket address. This socket address must be provided by the user of the client process. The socket is complete at this moment and the datagram is sent. The client process now calls the *recvfrom* procedure, which blocks the client process until the response comes back from the server process. There is no need to extract the remote socket address from this procedure because there is no call to the *sendto* procedure. In other words, the *recvfrom* procedure at the server site and the client site behave differently. In the server process, the *recvfrom* procedure is called first and the *sendto* next, so the remote socket address for *sendto* can be obtained from *recvfrom*. In the server process, the *sendto* is called before the *recvfrom*, so the remote address should be provided by the program user who knows to which server she wants to connect. Finally the *close* procedure is called to destroy the socket. Note that the client process is finite; after the response has been returned, the client process stops. Although we can design clients to send multiple datagrams, using a loop, each iteration of the loop looks like a new client to the server.

Programming Examples

In this section, we show how to write client and server programs to simulate the standard *echo* application using UDP. The client program sends a short string of characters to the server; the server echoes back the same string to the server. The standard application is used by a computer, the client, to test the liveliness of another computer, the server. Our programs are simpler than the ones used in the standard; we have eliminated some error-checking and debugging details for simplicity.

Echo Server Program Table 2.22 shows the echo server program using UDP. The program follows the flow diagram in Figure 2.60.

Table 2.22 *Echo server program using UDP*

```
// UDP echo server program
#include "headerFiles.h"
int main (void)
{
    // Declare and define variables
    int s;                                  // Socket descriptor (reference)
    int len;                                // Length of string to be echoed
    char   buffer [256];                    // Data buffer
    struct sockaddr_in servAddr;            // Server (local) socket address
    struct sockaddr_in clntAddr;            // Client (remote) socket address
    int  clntAddrLen;                       // Length of client socket address
    // Build local (server) socket address
    memset (&servAddr, 0, sizeof (servAddr));          // Allocate memory
    servAddr.sin_family =   AF_INET;                   // Family field
    servAddr.sin_port = htons (SERVER_PORT)            // Default port number
```

Table 2.22 *Echo server program using UDP (continued)*

```
16      servAddr.sin_addr.s_addr =  htonl (INADDR_ANY);          // Default IP address
17      // Create socket
18      if ((s = socket (PF_INET, SOCK_DGRAM, 0) < 0);
19      {
20          perror ("Error: socket failed!");
21          exit (1);
22      }
23      // Bind socket to local address and port
24      if ((bind (s, (struct sockaddr*) &servAddr, sizeof (servAddr)) < 0);
25      {
26          perror ("Error: bind failed!");
27          exit (1);
28      }
29      for ( ; ; )         // Run forever
30      {
31          // Receive String
32          len = recvfrom (s, buffer, sizeof (buffer), 0,
33                  (struct sockaddr*)&clntAddr, &clntAddrLen);
34          // Send String
35          sendto (s, buffer, len, 0, (struct sockaddr*)&clntAddr, sizeof(clntAddr));
36      } // End of for loop
37  } // End of echo server program
```

Lines 6 to 11 declare and define variables used in the program. Lines 13 to 16 allocate memory for the server socket address (using the *memset* function) and fill the field of the socket address with default values provided by the transport layer. To insert the port number, we use the *htons* (host to network short) function, which transforms a value in host byte-ordering format to a short value in network byte-ordering format. To insert the IP address, we use the *htonl* (host to network long) function to do the same thing.

Lines 18 to 22 call the socket function in an if-statement to check for error. Since this function returns –1 if the call fails, the programs prints the error message and exits. The *perror* function is a standard error function in C. Similarly, lines 24 to 28 call the bind function to bind the socket to the server socket address. Again, the function is called in an if-statement for error checking.

Lines 29 to 36 use an infinite loop to be able to serve clients in each iteration. Lines 32 and 33 call the *recvfrom* function to read the request sent by the client. Note that this function is a blocking one; when it unblocks, it receives the request message and, at the same time, provides the client socket address to complete the last part of the socket. Line 35 calls the *sendto* function to send back (echo) the same message to the client, using the client socket address obtained in the *recvfrom* message. Note that there is no processing done on the request message; the server just echoes what has been received.

Echo Client Program Table 2.23 shows the echo client program using UDP. The program follows the flow diagram in Figure 2.60.

Table 2.23 *Echo client program using UDP*

```
// UDP echo client program
#include "headerFiles.h"
int main (int argc, char* argv[ ])                  // Three arguments to be checked later
{
        // Declare and define variables
        int   s;                                    // Socket descriptor
        int   len;                                  // Length of string to be echoed
        char*  servName;                            // Server name
        int  servPort;                              // Server port
        char*  string;                              // String to be echoed
        char  buffer[256 + 1];                      // Data buffer
        struct sockaddr_in servAddr;                // Server socket address
        // Check and set program arguments
        if (argc != 3)
        {
                printf ("Error: three arguments are needed!");
                exit(1);
        }
        servName = argv[1];
        servPort = atoi (argv[2]);
        string =  argv[3];
        // Build server socket address
        memset (&servAddr, 0, sizeof (servAddr));
        servAddr.sin_family = AF_INET;
        inet_pton (AF_INET, servName, &servAddr.sin_addr);
        servAddr.sin_port = htons (servPort);
        // Create socket
        if ((s = socket (PF_INET, SOCK_DGRAM, 0) < 0);
        {
                perror ("Error: Socket failed!");
                exit (1);
        }
        // Send echo string
        len =  sendto (s, string, strlen (string), 0, (struct sockaddr)&servAddr, sizeof (servAddr));
        // Receive echo string
        recvfrom (s, buffer, len, 0, NULL, NULL);
        // Print and verify echoed string
```

Table 2.23 *Echo client program using UDP (continued)*

```
38          buffer [len] = '\0';
39          printf ("Echo string received: ";
40          fputs (buffer, stdout);
41          // Close the socket
42          close (s);
43          // Stop the program
44          exit (0);
45   } // End of echo client program
```

Lines 6 to 12 declare and define variables used in the program. Lines 14 to 21 test and set arguments that are provided when the program is run. The first two arguments provide the server name and server port number; the third argument is the string to be echoed. Lines 23 to 26 allocate memory, convert the server name to the server IP address using the function *inet_pton*, which is a function that calls DNS (discussed earlier in the chapter), and convert the port number to the appropriate byte-order. These three pieces of information, which are need for the *sendto* function, are stored in appropriate variables.

Line 34 calls the *sendto* function to send the request. Line 36 calls the *recvfrom* function to receive the echoed message. Note that the two arguments in this message are NULL because we do not need to extract the socket address of the remote site; the message already has been sent.

Lines 38 to 40 are used to display the echoed message on the screen for debugging purposes. Note that in line 38 we add a null character at the end of the echoed message to make it displayable by the next line. Finally, line 42 closes the socket and line 44 exits the program.

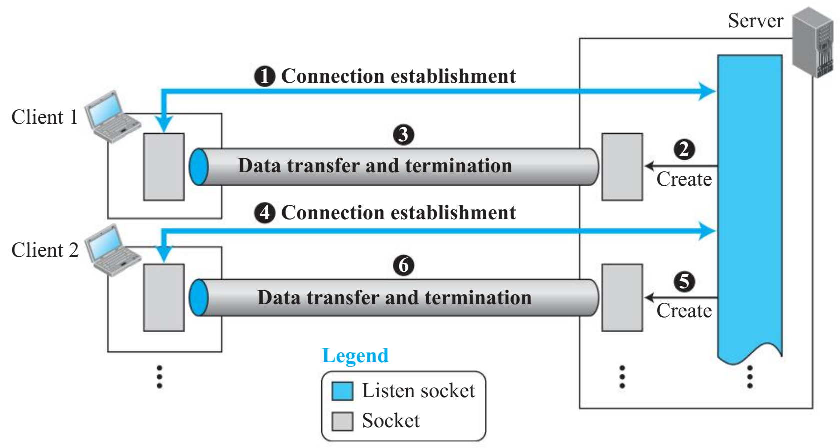
Communication Using TCP

As we described before, TCP is a connection-oriented protocol. Before sending or receiving data, a connection needs to be established between the client and the server. After the connection is established, the two parties can send and receive chunks of data to each other as long as they have data to do so. TCP communication can be iterative (serving a client at a time) or concurrent (serving several clients at a time). In this section, we discuss only the iterative approach. See concurrent approach in Java (Chapter 11).

Sockets Used in TCP

The TCP server uses two different sockets, one for connection establishment and the other for data transfer. We call the first one the *listen socket* and the second the *socket*. The reason for having two types of sockets is to separate the connection phase from the data exchange phase. A server uses a listen socket to listen for a new client trying to establish connection. After the connection is established, the server creates a socket to exchange data with the client and finally to terminate the connection. The client uses only one socket for both connection establishment and data exchange (see Figure 2.61).

Figure 2.61 *Sockets used in TCP communication*


Server
1 Connection establishment
Client 1
3
Data transfer and termination
2
Create
4 Connection establishment
Client 2
6
Data transfer and termination
5
Create
Legend
Listen socket
Socket

Communication Flow Diagram

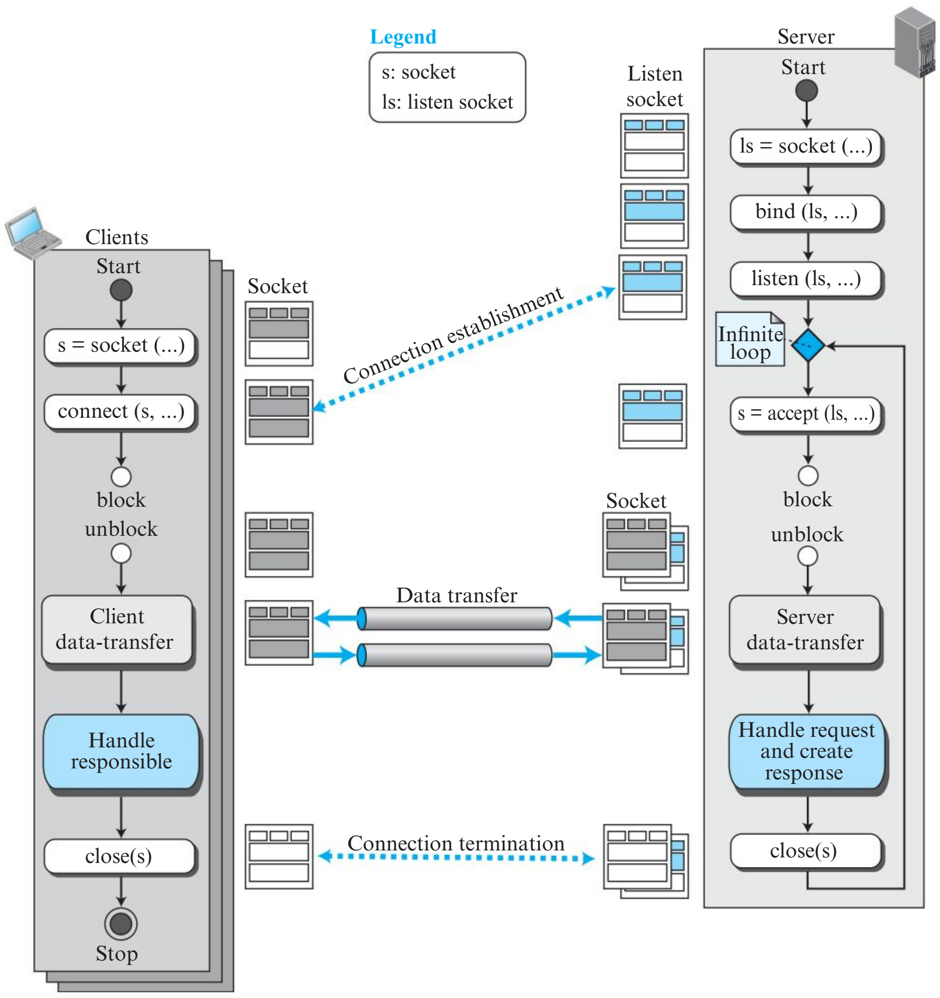
Figure 2.62 shows a simplified flow diagram for iterative communication. There are multiple clients, but only one server. Each client is served in each iteration of the loop. The flow diagram is almost similar to the one for UDP, but there are differences that we explain for each site.

Server Process In Figure 2.62, the TCP server process, like the UDP server process, calls the *socket* and *bind* procedures, but these two procedures create the listen socket to be used only for the connection establishment phase. The server process then calls the *listen* procedure, to allow the operating system to start accepting the clients, completing the connection phase, and putting them in the waiting list to be served. This procedure also defines the size of the connected client waiting list, which depends on the complexity of the server process, but it is normally 5.

The server process now starts a loop and serves the clients one by one. In each iteration, the server process calls the *accept* procedure that removes one client from the waiting list of the connected clients for serving. If the list is empty, the *accept* procedure blocks until there is a client to be served. When the accept procedure returns, it creates a new socket that is the same as the listen socket. The listen socket now moves to the background and the new socket becomes the active one. The server process now uses the client socket address obtained during the connection establishment to fill the remote socket address field in the newly created socket.

At this time the client and server can exchange data. We have not shown the specific way in which the data transfer takes place because it depends on the specific client/server pair. TCP uses the *send* and *recv* procedures to transfer bytes of data between them. These two procedures are simpler than the *sendto* and *recvfrom* procedures used in UDP because they do not provide the remote socket address; a connection has already been established between the client and server. However, since TCP is used to transfer messages with no boundaries, each application needs to carefully design the data transfer section. The *send* and *recv* procedures may be called several times to

Figure 2.62 *Flow diagram for iterative TCP communication*



handle a large amount of data transfer. The flow diagram in Figure 2.62 can be considered as a generic one; for a specific purpose the diagram for the *server data-transfer* box needs to be defined. We will this do for a simple example when we discuss the echo client/server program.

Client Process The client flow diagram is almost similar to the UDP version except that the *client data-transfer* box needs to be defined for each specific case. We do so when we write a specific program later.

Programming Examples

In this section, we show how to write client and server programs to simulate the standard *echo* application using TCP. The client program sends a short string of characters to the server; the server echoes back the same string to the client. However, before we do so, we need to provide the flow diagram for the client and server data-transfer boxes, which is shown in Figure 2.63.

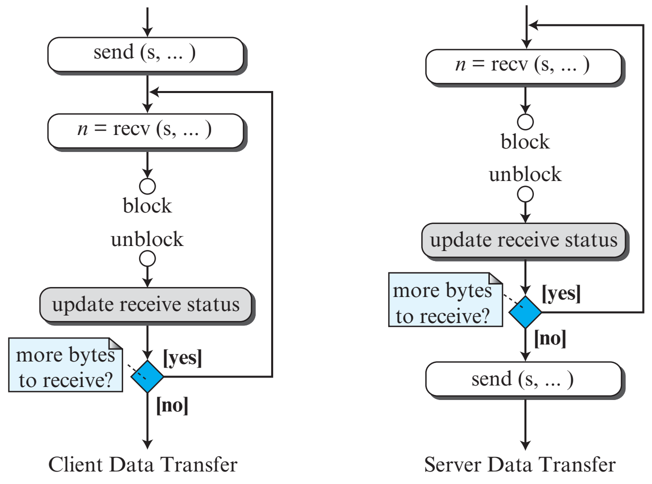


Figure 2.63 *Flow diagram for the client and server data-transfer boxes*

For this special case, since the size of the string to be sent is small (less than a few words), we can do it in one call to the *send* procedure in the client. However, it is not guaranteed that the TCP will send the whole message in one segment. Therefore, we need to use a set of *recv* calls in the server site (in a loop), to receive the whole message and collect them in the buffer to be sent back in one shot. When the server is sending back the echo message, it may also use several segments to do so, which means the *recv* procedure in the client needs to be called as many times as needed.

Another issue to be solved is setting the buffers that hold data at each site. We need to control how many bytes of data we have received and where the next chunk of data is stored. The program sets some variables to control the situation, as shown in Figure 2.64. In each iteration, the pointer (ptr) moves ahead to point to the next bytes to receive, the length of received bytes (len) is increased, and the maximum number of bytes to be received (maxLen) is decreased.

After the above two considerations, we can now write the server and the client program.

Echo Server Program Table 2.24 shows the echo server program using TCP. The program follows the flow diagram in Figure 2.62. Each shaded section corresponds to one instruction in the layout. The colored sections show the data transfer section of the diagram.

Lines 6 to 16 declare and define variables. Lines 18 to 21 allocate memory and construct the local (server) socket address as described in the UDP case. Lines 23 to 27 create the listen socket. Lines 29 to 33 bind the listen socket to the server socket address constructed in lines 18 to 21. Lines 35 to 39 are new in TCP communication.

Figure 2.64 *Buffer used for receiving*

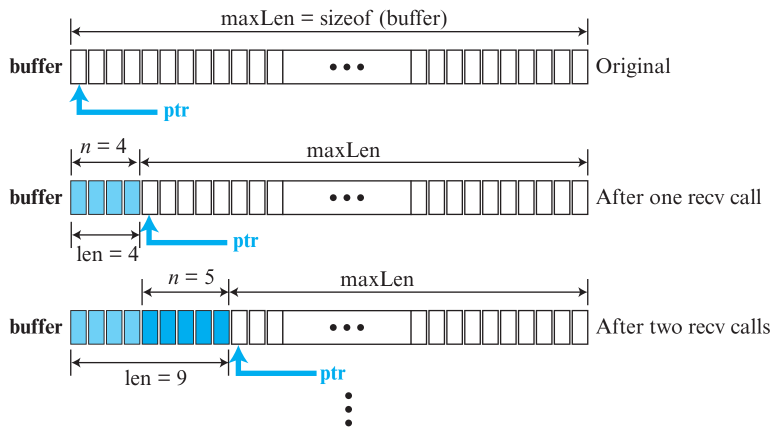


Table 2.24 *Echo server program using the services of TCP*

```
// Echo server program
#include "headerFiles.h"
int main (void)
{
        // Declare and define
        int ls;                                 // Listen socket descriptor (reference)
        int s;                                  // socket descriptor (reference)
        char buffer [256];                      // Data buffer
        char* ptr = buffer;                     // Data buffer
        int len = 0;                            // Number of bytes to send or receive
        int maxLen = sizeof (buffer);           // Maximum number of bytes to receive
        int n = 0;                              // Number of bytes for each recv call
        int waitSize = 16;                      // Size of waiting clients
        struct sockaddr_in serverAddr;          // Server address
        struct sockaddr_in clientAddr;          // Client address
        int clntAddrLen;                        // Length of client address
        // Create local (server) socket address
        memset (&servAddr, 0, sizeof (servAddr));
        servAddr.sin_family = AF_INET;
        servAddr.sin_addr.s_addr = htonl (INADDR_ANY);      // Default IP address
        servAddr.sin_port = htons (SERV_PORT);              // Default port
        // Create listen socket
        if (ls = socket (PF_INET, SOCK_STREAM, 0) < 0);
        {
                perror ("Error: Listen socket failed!");
```

Table 2.24 *Echo server program using the services of TCP (continued)*

```
26            exit (1);
27        }
28        // Bind listen socket to the local socket address
29        if (bind (ls, &servAddr, sizeof (servAddr)) < 0);
30        {
31            perror ("Error: binding failed!");
32            exit (1);
33        }
34        // Listen to connection requests
35        if (listen (ls, waitSize) < 0);
36        {
37            perror ("Error: listening failed!");
38            exit (1);
39        }
40        // Handle the connection
41        for ( ; ; )                             // Run forever
42        {
43            // Accept connections from client
44            if (s = accept (ls, &clntAddr, &clntAddrLen) < 0);
45            {
46                perror ("Error: accepting failed!);
47                exit (1);
48            }
49            // Data transfer section
50            while ((n = recv (s, ptr, maxLen, 0)) > 0)
51            {
52                ptr + = n;              // Move pointer along the buffer
53                maxLen - = n;           // Adjust maximum number of bytes to receive
54                len + = n;              // Update number of bytes received
55            }
56            send (s, buffer, len, 0);   // Send back (echo) all bytes received
57            // Close the socket
58            close (s);
59        }  // End of for loop
60    } // End of echo server program
```

The *listen* function is called to let the operating system complete the connection establishment phase and put the clients in the waiting list. Lines 44 to 48 call the *accept* function to remove the next client in the waiting list and start serving it. This function blocks if there is no client in the waiting list. Lines 50 to 56 code the data transfer section depicted in Figure 2.63. The maximum buffer size, the length of the string echoed, is the same as shown in Figure 2.64.

Echo Client Program Table 2.25 shows the echo client program using TCP. The program follows the outline in Figure 2.62. Each shaded section corresponds to one instruction in the flow diagram. The colored section corresponds to the data transfer section.

Table 2.25 *Echo client program using TCP*

```
// TCP echo client program
#include "headerFiles.h"
int main (int argc, char* argv[ ])              // Three arguments to be checked later
{
     // Declare and define
     int  s;                                    // Socket descriptor
     int  n;                                    // Number of bytes in each recv call
     char* servName;                            // Server name
     int servPort;                              // Server port number
     char* string;                              // String to be echoed
     int  len;                                  // Length of string to be echoed
     char  buffer [256 + 1];                    // Buffer
     char* ptr = buffer;                        // Pointer to move along the buffer
     struct sockaddr_in serverAddr;             // Server socket address
     // Check and set arguments
     if (argc != 3)
     {
          printf ("Error: three arguments are needed!");
          exit (1);
     }
     servName = arg [1];
     servPort = atoi (arg [2]);
     string = arg [3];
     // Create remote (server) socket address
     memset (&servAddr, 0, sizeof(servAddr);
     serverAddr.sin_family = AF_INET;
     inet_pton (AF_INET, servName, &serverAddr.sin_addr); // Server IP address
     serverAddr.sin_port = htons (servPort);              // Server port number
     // Create socket
     if ((s = socket (PF_INET, SOCK_STREAM, 0) < 0);
     {
          perror ("Error: socket creation failed!");
          exit (1);
     }
     // Connect  to the server
     if (connect (sd, (struct sockaddr*)&servAddr, sizeof(servAddr)) < 0);
     {
          perror ("Error: connection failed!");
```

Table 2.25 *Echo client program using TCP (continued)*

```
39              exit (1);
40          }
41          // Data transfer section
42          send (s, string, strlen(string), 0);
43          while ((n = recv (s, ptr, maxLen, 0)) > 0)
44          {
45              ptr + = n;                      // Move pointer along the buffer
46              maxLen - = n;                   // Adjust the maximum number of bytes
47              len += n;                       // Update the length of string received
48          } // End of while loop
49          // Print and verify the echoed string
50          buffer [len] = '\0';
51          printf ("Echoed string received: ");
52          fputs (buffer, stdout);
53          // Close socket
54          close (s);
55          // Stop program
56          exit (0);
57      } // End of echo client program
```

The client program for TCP is very similar to the client program for UDP, with a few differences. Since TCP is a connection-oriented protocol, the *connect* function is called in lines 36 to 40 to make connection to the server. Data transfer is done in lines 42 to 48 using the idea depicted in Figure 2.63. The length of data received and the pointer movement is done as shown in Figure 2.64.

2.6 END-CHAPTER MATERIALS

2.6.1 Further Reading

For more details about subjects discussed in this chapter, we recommend the following books and RFCs. The items enclosed in brackets refer to the reference list at the end of the book.

Books

Several books give thorough coverage of materials discussed in this chapter including [Com 06], [Mir 07], [Ste 94], [Tan 03], [Bar *et al.* 05].

RFCs

HTTP is discussed in RFC's 2068 and 2109. FTP is discussed in RFC's 959, 2577, and 2585. TELNET is discussed in RFC's 854, 855, 856, 1041, 1091, 1372, and 1572. SSH is discussed in RFC's 4250, 4251, 4252, 4253, 4254, and 4344. DNS is discussed in RFC's 1034, 1035, 1996, 2535, 3008, 3658, 3755, 3757, 3845, 3396, and 3342. SMTP is discussed in RFC's 2821and 2822. POP3 is explained in RFC 1939. MIME is discussed in RFC's 2046, 2047, 2048, and 2049.

2.6.2 Key Terms

active document
application programming interface (API)
browser
Chord
client-server paradigm
cookie
country domain
Distributed Hash Table (DHT)
domain name
domain name space
Domain Name System (DNS)
dynamic document
Dynamic Domain Name System (DDNS)
File Transfer Protocol (FTP)
fully qualified domain name (FQDN)
generic domain
hypermedia
hypertext
HyperText Transfer Protocol (HTTP)
Internet Mail Access Protocol (IMAP)
iterative resolution
Kademlia
label
local login
message access agent (MAA)
message transfer agent (MTA)
Multipurpose Internet Mail Extensions (MIME)
name space
Network Virtual Terminal (NVT)
nonpersistent connection
partially qualified domain name (PQDN)
Pastry
peer-to-peer (P2P) paradigm
persistent connection
port forwarding
Post Office Protocol, version 3 (POP3)
proxy server
remote logging
resolver
root server
Secure Shell (SHH)
Simple Mail Transfer Protocol (SMTP)
socket address
socket interface
static document
STREAM
terminal network (TELNET)
Transport Layer Interface (TLI)
uniform resource locator (URL)
user agent (UA)
web page
World Wide Web (WWW)
zone

2.6.3 Summary

Applications in the Internet are designed using either a client-server paradigm or a peer-to-peer paradigm. In a client-server paradigm, an application program, called a server, provides services and another application program, called a client, receives services. A server program is an infinite program; a client program is finite. In a peer-to-peer paradigm, a peer can be both a client and a server.

The World Wide Web (WWW) is a repository of information linked together from points all over the world. Hypertext and hypermedia documents are linked to one another through pointers. The HyperText Transfer Protocol (HTTP) is the main protocol used to access data on the World Wide Web (WWW).

File Transfer Protocol (FTP) is a TCP/IP client-server application for copying files from one host to another. FTP requires two connections for data transfer: a control connection and a data connection. FTP employs NVT ASCII for communication between dissimilar systems.

Electronic mail is one of the most common applications on the Internet. The e-mail architecture consists of several components such as user agent (UA), main transfer agent (MTA), and main access agent (MAA). The protocol that implements MTA is called Simple Main Transfer Protocol (SMTP). Two protocols are used to implement MAA: Post Office Protocol, version 3 (POP3) and Internet Mail Access Protocol, version 4 (IMAP4).

TELNET is a client-server application that allows a user to log into a remote machine, giving the user access to the remote system. When a user accesses a remote system via the TELNET process, this is comparable to a time-sharing environment.

The Domain Name System (DNS) is a client-server application that identifies each host on the Internet with a unique name. DNS organizes the name space in a hierarchical structure to decentralize the responsibilities involved in naming.

In a peer-to-peer network, Internet users that are ready to share their resources become peers and form a network. Peer-to-peer networks are divided into centralized and decentralized. In a centralized P2P network, the directory system uses the client-server paradigm, but the storing and downloading of the files are done using the peer-to-peer paradigm. In a decentralized network, both the directory system and storing and downloading of flies are done using the peer-to-peer paradigm.

2.7 PRACTICE SET

2.7.1 Quizzes

A set of interactive quizzes for this chapter can be found on the book website. It is strongly recommended that students take the quizzes to check his/her understanding of the materials before continuing with the practice set.

2.7.2 Questions

Q2-1. In the client-server paradigm, explain why a server should be run all the time, but a client can be run whenever it is needed.

Q2-2. Can a program written to use the services of UDP be run on a computer that has installed TCP as the only transport-layer protocol? Explain.

Q2-3. During the weekend, Alice often needs to access files stored on her office desktop, from her home laptop. Last week, she installed a copy of the FTP server process on her desktop at her office and a copy of the FTP client process on her laptop at home. She was disappointed when she could not access her files during the weekend. What could have gone wrong?

Q2-4. Most of the operating systems installed on personal computers come with several client processes, but normally no server processes. Explain the reason.

Q2-5. Assume that we add a new protocol to the application layer. What changes do we need to make to other layers?

Q2-6. Explain which entity provides service and which one receives service in the client-server paradigm.

Q2-7. A source socket address is a combination of an IP address and a port number. Explain what each section identifies.

Q2-8. Explain how a client process finds the IP address and the port number to be inserted in a remote socket address.

Q2-9. A new application is to be designed using the client-server paradigm. If only small messages need to be exchanged between the client and the server without the concern for message loss or corruption, what transport-layer protocol do you recommend?

Q2-10. Which of the following can be a source of data?

a. a keyboard **b.** a monitor **c.** a socket

Q2-11. Alice has a video clip that Bob is interested in getting; Bob has another video clip that Alice is interested in getting. Bob creates a web page and runs an HTTP server. How can Alice get Bob's clip? How can Bob get Alice's clip?

Q2-12. When an HTTP server receives a request message from an HTTP client, how does the server know when all headers have arrived and the body of the message is to follow?

Q2-13. If an HTTP request needs to run a program at the server site and downloads the result to the client server, the program is an example of

a. a static document **b.** a dynamic document **c.** an active document

Q2-14. Assume that we design a new client-server application program that requires persistent connection. Can we use UDP as the underlying transport-layer protocol for this new application?

Q2-15. In FTP, which entity (client or server) starts (actively opens) the control connection? Which entity starts (actively opens) the data transfer connection?

Q2-16. What do you think would happen if the control connection were severed before the end of an FTP session? Would it affect the data connection?

Q2-17. In FTP, if the client needs to retrieve one file from the server site and store one file on the server site, how many control connections and how many data-transfer connections are needed?

Q2-18. In FTP, can a server retrieve a file from the client site?

Q2-19. In FTP, can a server get the list of the files or directories from the client?

Q2-20. FTP can transfer files between two hosts using different operating systems with different file formats. What is the reason?

Q2-21. In a nonpersistent HTTP connection, how can HTTP inform the TCP protocol that the end of the message has been reached?

Q2-22. Can you find an analogy in our daily life as to when we use two separate connections in communication similar to the control and data connections in FTP?

Q2-23. FTP uses two separate well-known port numbers for control and data connection. Does this mean that two separate TCP connections are created for exchanging control information and data?

Q2-24. FTP uses the services of TCP for exchanging control information and data transfer. Could FTP have used the services of UDP for either of these two connections? Explain.

Q2-25. Can we have a control connection without a data-transfer connection in FTP? Explain.

Q2-26. Can we have a data-transfer connection without a control connection in FTP? Explain.

Q2-27. Assume that we need to download an audio using FTP. What file type should we specify in our command?

Q2-28. Both HTTP and FTP can retrieve a file from a server. Which protocol should we use to download a file?

Q2-29. Does FTP have a message format for exchanging commands and responses during control connection?

Q2-30. Does FTP have a message format for exchanging files or a list of directories/files during the file-transfer connection?

Q2-31. Alice has been on a long trip without checking her e-mail. She then finds out that she has lost some e-mails or attachments her friends claim they have sent to her. What can be the problem?

Q2-32. Assume that a TELNET client uses ASCII to represent characters, but the TELNET server uses EBCDIC to represent characters. How can the client log into the server when character representations are different?

Q2-33. The TELNET application has no commands such as those found in FTP or HTTP to allow the user to do something such as transfer a file or access a web page. In what way can this application be useful?

Q2-34. Can a host use a TELNET client to get services provided by other client-server applications such as FTP or HTTP?

Q2-35. Are the HELO and MAIL FROM commands both necessary in SMTP? Why or why not?

Q2-36. In Figure 2.20 in the text, what is the difference between the MAIL FROM in the envelope and the FROM in the header?

Q2-37. In a DHT-based network, assume node 4 has a file with key 18. The closest next node to key 18 is node 20. Where is the file stored?

a. in the direct method **b.** in the indirect method

Q2-38. In a Chord network, we have node N5 and key k5. Is N5 the predecessor of k5? Is N5 the successor of k5?

Q2-39. In a Kademlia network, the size of the identifier space is 1024. What is the height of the binary tree (the distance between the root and each leaf)? What is the number of leaves? What is the number of subtrees for each node? What is the number of rows in each routing table?

Q2-40. In Kademlia, assume $m = 4$ and active nodes are N4, N7, and N12. Where is the key k3 stored in this system?

Q2-41. In DNS, which of the following are FQDNs and which are PQDNs?

a. xxx **b.** xxx.yyy.net **c.** zzz.yyy.xxx.edu.

Q2-42. In a DHT-based network, assume $m = 4$. If the hash of a node identifier is 18, where is the location of the node in the DHT space?

2.7.3 Problems

P2-1. Draw a diagram to show the use of a proxy server that is part of the client network:

a. Show the transactions between the client, proxy server, and the target server when the response is stored in the proxy server.

b. Show the transactions between the client, proxy server, and the target server when the response is not stored in the proxy server.

P2-2. In Chapter 1, we mentioned that the TCP/IP suite, unlike the OSI model, has no presentation layer. But an application-layer protocol can include some of the features defined in this layer if needed. Does HTTP have any presentation-layer features?

P2-3. HTTP version 1.1 defines the persistent connection as the default connection. Using RFC 2616, find out how a client or server can change this default situation to nonpersistent.

P2-4. In Chapter 1, we mentioned that the TCP/IP suite, unlike the OSI model, has no session layer. But an application-layer protocol can include some of the features defined in this layer, if needed. Does HTTP have any session-layer features?

P2-5. Assume that there is a server with the domain name *www.common.com.*

a. Show an HTTP request that needs to retrieve the document **/usr/users/doc**. The client accepts MIME version 1, GIF or JPEG images, but the document should not be more than four days old.

b. Show the HTTP response to part *a* for a successful request.

P2-6. In HTTP, draw a figure to show the application of cookies in a scenario in which the server allows only the registered customer to access the server.

P2-7. In HTTP, draw a figure to show the application of cookies in a web portal using two sites.

P2-8. In HTTP, draw a figure to show the application of cookies in a scenario in which the server uses cookies for advertisement. Use only three sites.

P2-9. Encode the following message in base64:

01010111 00001111 11110000

P2-10. Encode the following message in quoted-printable:

01001111 10101111 01110001

P2-11. In SMTP, a sender sends unformatted text. Show the MIME header.

P2-12. In SMTP,

a. A non-ASCII message of 1000 bytes is encoded using base64. How many bytes are in the encoded message? How many bytes are redundant? What is the ratio of redundant bytes to the total message?

b. A message of 1000 bytes is encoded using quoted-printable. The message consists of 90 percent ASCII and 10 percent non-ASCII characters. How many bytes are in the encoded message? How many bytes are redundant? What is the ratio of redundant bytes to the total message?

c. Compare the results of the two previous cases. How much has the efficiency improved if the message is a combination of ASCII and non-ASCII characters?

P2-13. POP3 protocol has some optional commands (that a client/server can implement). Using the information in RFC 1939, find the meaning and the use of the following optional commands:

a. UIDL b. TOP 1 15 c. USER d. PASS

P2-14. Using RFC 1939, assume a POP3 client is in the download-and-keep mode. Show the transaction between the client and the server if the client has only two messages of 192 and 300 bytes to download from the server.

P2-15. According to RFC 1939, a POP3 session is in one of the following four states: closed, authorization, transaction, or update. Draw a diagram to show these four states and how POP3 moves between them.

P2-16. POP3 protocol has some basic commands (that each client/server needs to implement). Using the information in RFC 1939, find the meaning and the use of the following basic commands:

a. STAT b. LIST c. DELE 4

P2-17. In FTP, a user (Jane) wants to retrieve an EBCDIC file named *huge* from */usr/users/report* directory using the ephemeral port 61017. The file is so large that the user wants to compress it before it is transferred. Show all the commands and responses.

P2-18. In FTP, a user (Jan) wants to make a new directory called *Jan* under the directory */usr/usrs/letters*. Show all of the commands and responses.

P2-19. In FTP, a user (Maria) wants to move a file named *file1* from */usr/users/report* directory to the directory */usr/top/letters*. Note that this is a case of renaming a file. We first need to give the name of the old file and then define the new name. Show all of the commands and responses.

P2-20. In Chapter 1, we mentioned that the TCP/IP suite, unlike the OSI model, has no presentation layer. But an application-layer protocol can include some of the features defined in this layer if needed. Does FTP have any presentation-layer features?

P2-21. Using RFC 1939, assume that a POP3 client is in the download-and-delete mode. Show the transaction between the client and the server if the client has only two messages of 230 and 400 bytes to download from the server.

P2-22. In Chapter 1, we mentioned that the TCP/IP suite, unlike the OSI model, has no presentation layer. But an application-layer protocol can include some of the features defined in this layer, if needed. Does SMTP have any presentation-layer features?

P2-23. In Chapter 1, we mentioned that the TCP/IP suite, unlike the OSI model, has no session layer. But an application-layer protocol can include some of the features defined in this layer, if needed. Does SMTP or POP3 have any session-layer features?

P2-24. In FTP, assume a client with user name John needs to store a video clip called **video2** on the directory **/top/videos/general** on the server. Show the commands and responses exchanged between the client and the server if the client chooses ephemeral port number 56002.

P2-25. In Chord, assume that the successor of node N12 is N17. Find whether node N12 is the predecessor of any of the following keys:

a. k12 **b.** k15 **c.** k17 **d.** k22

P2-26. In a Chord network using DHT with $m = 4$, draw the identifier space and place four peers with node ID addresses N3, N8, N11, and N13 and three keys with addresses k5, k9, and k14. Determine which node is responsible for each key. Create a finger table for each node.

P2-27. In Chapter 1, we mentioned that the TCP/IP suite, unlike the OSI model, has no session layer. But an application-layer protocol can include some of the features defined in this layer if needed. Does FTP have any session-layer features?

P2-28. In Chord, assume that the size of the identifier space is 16. The active nodes are N3, N6, N8, and N12. Show the finger table (only the target-key and the successor column) for node N6.

P2-29. Repeat Example 2.16 in the text, but assume that node N5 needs to find the responsible node for key k16. *Hint:* Remember that interval checking needs to be done in modulo 32 arithmetic.

P2-30. In Pastry, assume the address space is 16 and that $b = 2$. How many digits are in an address space? List some of the identifiers.

P2-31. In a Pastry network with $m = 32$ and $b = 4$, what is the size of the routing table and the leaf set?

P2-32. Show the outline of a routing table for Pastry with address space of 16 and $b = 2$. Give some possible entries for each cell in the routing table of Node N21.

P2-33. In a Pastry network using DHT, in which $m = 4$ and $b = 2$, draw the identifier space with four nodes, N02, N11, N20, and N23, and three keys, k00, k12, and k24. Determine which node is responsible for each key. Also, show the leaf set and routing table for each node. Although it is unrealistic, assume that the proximity metric between each two nodes is based on numerical closeness.

P2-34. In the previous problem, answer the following questions:

a. Show how node N02 responds to a query to find the responsible node for k24.

b. Show how node N20 responds to a query to find the responsible node for k12.

P2-35. In a Chord network with $m = 4$, node N2 has the following finger-table values: N4, N7, N10, and N12. For each of the following keys, first find if N2 is the predecessor of the key. If the answer is no, find which node (the closest predecessor) should be contacted to help N2 find the predecessor.

a. k1 **b.** k6 **c.** k9 **d.** k13

P2-36. Repeat Example 2.16 in the text, but assume that node N12 needs to find the responsible node for key k7. *Hint:* Remember that interval checking needs to be done in modulo 32 arithmetic.

P2-37. In a Kademlia network with $m = 4$, we have five active nodes: N2, N3, N7, N10, and N12. Find the routing table for each active node (with only one column).

P2-38. In Figure 2.60 in the text, how does the server know that a client has requested a service?

P2-39. In Figure 2.62 in the text, how is the socket created for data transfer at the server site?

P2-40. Assume that we want to make the TCP client program in Table 2.25 more generic to be able to send a string and to handle the response received from the server. Show how this can be done.

P2-41. Using the binary tree in Figure 2.55 in the text, show the subtree for node N11.

P2-42. Using the routing tables in Figure 2.56 in the text, explain and show the route if node N0 receives a lookup message for the node responsible for K12.

2.8 SIMULATION EXPERIMENTS

2.8.1 Applets

We have created some Java applets to show some of the main concepts discussed in this chapter. It is strongly recommended that students activate these applets on the book website and carefully examine the protocols in action.

2.8.2 Lab Assignments

In Chapter one, we downloaded and installed Wireshark and learned about its basic features. In this chapter, we use Wireshark to capture and investigate some application-layer protocols. We use Wireshark to simulate six protocols: HTTP, FTP, TELNET, SMTP, POP3, and DNS.

Lab2-1. In the first lab, we retrieve web pages using HTTP. We use Wireshark to capture packets for analysis. We learn about the most common HTTP messages. We also capture response messages and analyze them. During the lab session, some HTTP headers are also examined and analyzed.

Lab2-2. In the second lab, we use FTP to transfer some files. We use Wireshark to capture some packets. We show that FTP uses two separate connections: a control connection and a data-transfer connection. The data connection is opened and then closed for each file transfer activity. We also show that FTP is an insecure file transfer protocol because the transaction is done in plaintext.

Lab2-3. In the third lab, we use Wireshark to capture packets exchanged by the TELNET protocol. As in FTP, we are able to observe commands and responses during the session in the captured packets. Like FTP, TELNET is vulnerable to hacking because it sends all data including the password in plaintext.

Lab2-4. In the fourth lab, we investigate SMTP protocol in action. We send an e-mail and, using Wireshark, we investigate the contents and the format of the SMTP packet exchanged between the client and the server. We check that the three phases we discussed in the text exist in this SMTP session.

Lab2-5. In the fifth lab, we investigate the state and behavior of the POP3 protocol. We retrieve the mails stored in our mailbox at the POP3 server and observe and analyze the states of the POP3 and the type and the contents of the messages exchanged, by analyzing the packets through Wireshark.

Lab2-6. In the sixth lab, we analyze the behavior of the DNS protocol. In addition to Wireshark, several network utilities are available for finding some information stored in the DNS servers. In this lab, we use the *dig* utilities (which has replaced *nslookup*). We also use *ipconfig* to manage the cached DNS records in the host computer. When we use these utilities, we set Wireshark to capture the packets sent by these utilities.

2.9 PROGRAMMING ASSIGNMENT

Write, compile, and test the following programs using the computer language of your choice:

Prg2-1. Write a program to make the UDP server program in Table 2.22 more generic: to receive a request, to process the request, and to send back the response.

Prg2-2. Write a program to make the UDP client program in Table 2.23 more generic to be able to send any request created by the client program.

Prg2-3. Write a program to make the TCP server program in Table 2.24 more generic: to receive a request, to process the request, and to send back the response.