Table 17.3 *Echo Server Program using the Services of TCP*

```
// Echo server program
#include "headerFiles.h"

int main (void)
{
    // Declaration and definition
    int listensd;                           // Listen socket descriptor
    int connectsd;                          // Connecting socket descriptor
    int n;                                  // Number of bytes in each reception
    int  bytesToRecv;                       // Total bytes to receive
    int   processID;                        // ID of the child process
    char buffer [256];                      // Data buffer
    char* movePtr;                          // Pointer to the buffer
    struct sockaddr_in serverAddr;          // Server address
    struct sockaddr_in clientAddr;          // Client address
     int clAddrLen;                         // Length of client address
    // Create listen socket
    listensd = socket (PF_INET, SOCK_STREAM, 0);
    // Bind listen socket to the local address and port
    memset (&serverAddr, 0, sizeof (serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);
    serverAddr.sin_port = htons (7);  // We assume port 7
    bind (listensd, &serverAddr, sizeof (serverAddr));
    // Listen to connection requests
     listen (listensd, 5);
    // Handle the connection
     for ( ; ; )                  // Run forever
    {
        connectsd = accept (listensd, &clientAddr, &clAddrLen);
        processID = fork ();
        if (processID == 0)                      // Child process
        {
            close (listensd);
            bytesToRecv = 256;
            movePtr = buffer;
            while ( (n = recv (connectfd, movePtr, bytesToRecv, 0)) > 0)
            {
                movePtr = movePtr + n;
                bytesToRecv = movePtr – n;
            }   // End of while
```

Table 17.3 *Echo Server Program using the Services of TCP (continued)*

```
42                  send (connectsd, buffer, 256, 0);
43                  exit (0);
44              } // End of if
45              close (connectsd):                      // Back to parent process
46          }  // End of for loop
47  } // End of echo server program
```

The program follows the flow diagram of Figure 17.11. Every time the *recv* function is unblocked it gets some data and stores it at the end of the buffer. The movePtr is then moved to point to the location where the next data chunk should be stored (line 39). The number of bytes to read is also decremented from the original value (26) to prevent overflow in the buffer (line 40). After all data have been received, the server calls the *send* function to send them to the client. As we mentioned before, there is only one *send* call, but TCP may send data in several segments. The child process then calls the *close* function to destroy the connect socket.

Table 17.4 shows the client program for the echo process that uses the services of TCP. It then uses the same strategy described in Table 17.2 to create the server socket address. The program then gets the string to be echoed from the keyboard, stores it in *sendBuffer*, and sends it. The result may come in different segments. The program uses a loop and repeat, calling the *recv* function until all data arrive. As usual, we have ignored code for error checking to make the program simpler. It needs to be added if the code is actually used to send data to a server.

Table 17.4 *Echo Client Program using the services of TCP*

```
01  // TCP echo client program
02  #include "headerFiles.h"
03
04  int main (void)
05  {
06          // Declaration and definition
07          int  sd;                                 // Socket descriptor
08          int  n;                                  // Number of bytes received
09          int  bytesToRecv;                        // Number of bytes to receive
10          char  sendBuffer [256];                  // Send buffer
11          char  recvBuffer [256];                  // Receive buffer
12          char* movePtr;                           // A pointer the received buffer
13          struct sockaddr_in  serverAddr;          // Server address
14
15          // Create socket
16          sd = socket (PF_INET, SOCK_STREAM, 0);
17          // Create server socket address
18          memset (&serverAddr, 0, sizeof(serverAddr);
19          serverAddr.sin_family = AF_INET;
20          inet_pton (AF_INET, “server address”, &serverAddr.sin_addr);
21          serverAddr.sin_port = htons (7);     // We assume port 7
22          // Connect
```

Table 17.4 *Echo Client Program using the services of TCP (continued)*

```
    connect (sd, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
    // Send and receive data
    fgets (sendBuffer, 256, stdin);
    send (fd, sendBuffer, strlen (sendbuffer), 0);
    bytesToRecv = strlen (sendbuffer);
    movePtr = recvBuffer;
    while ( (n = recv (sd, movePtr, bytesToRecv, 0) ) > 0)
    {
        movePtr = movePtr + n;
        bytesToRecv = bytesToRecv – n;
    } // End of while loop
    recvBuffer[bytesToRecv] = 0;
    printf (“Received from server:”);
    fputs (recvBuffer, stdout);
    // Close and exit
    close (sd);
    exit (0);
} // End of echo client program
```

Predefined Client-Server Applications

The Internet has defined a set of applications using **client-server paradigms.** They are established programs that can be installed and be used. Some of these applications are designed to give some specific service (such as FTP), some are designed to allow users to log into the server and perform the desired task (such TELNET), and some are designed to help other application programs (such as DNS). We discuss these application programs in detail in Chapters 18 to 24.

In Appendix F we give some simple Java versions of programs in Table 17.1 to 17.4

17.2 PEER-TO-PEER PARADIGM

Although most of the applications available in the Internet today use the client-server paradigm, the idea of using the so called **peer-to-peer (P2P) paradigm** recently has attracted some attention. In this paradigm, two peer computers (laptops, desktops, or main frames) can communicate with each other to exchange services. This paradigm is interesting in some areas such file as transfer in which the client-server paradigm may put a lot of the load on the server machine if a client wants to transfer a large file such as an audio or video file. The idea is also interesting if two peers need to exchange some files or information to each other without going to a server. However, we need to mention that the P2P paradigm does not ignore the client-server paradigm. What it does