17.1 CLIENT-SERVER PARADIGM

The purpose of a network, or an internetwork, is to provide services to users: A user at a local site wants to receive a service from a computer at a remote site. One way to achieve this purpose is to run two programs. A local computer runs a program to request a service from a remote computer; the remote computer runs a program to give service to the requesting program. This means that two computers, connected by an internet, must each run a program, one to provide a service and one to request a service.

At first glance, it looks simple to enable communication between two application programs, one running at the local site, the other running at the remote site. But many questions arise when we want to implement the approach. Some of the questions that we may ask are:

1. Should both application programs be able to request services and provide services or should the application programs just do one or the other? One solution is to have an application program, called the *client*, running on the local machine, request a service from another application program, called the *server,* running on the remote machine. In other words, the tasks of requesting a service and providing a service are separate from each other. An application program is either a requester (a client), or a provider (a server). In other words, application programs come in pairs, client and server, both having the same name.
2. Should a server provide services only to one specific client or should the server be able to provide services to any client that requests the type of service it provides? The most common solution is a server providing a service for any client that needs that type of service, not a particular one. In other words, the server-client relationship is one-to-many.
3. Should a computer run only one program (client or server)? The solution is that any computer connected to the Internet should be able to run any client program if the appropriate software is available. The server programs need to be run on a computer that can be continuously running as we will see later.
4. When should an application program be running? All of the time or just when there is a need for the service? Generally, a client program, which requests a service, should run only when it is needed. The server program, which provides a service, should run all the time because it does not know when its service will be needed.
5. Should there be only one universal application program that can provide any type of service a user wants? Or should there be one application program for each type of service? In TCP/IP, services needed frequently and by many users have specific client-server application programs. For example, we have separate client-server application programs that allow users to access files, send e-mail, and so on. For

services that are more customized, we should have one generic application program that allows users to access the services available on a remote computer. For example, we should have a client-server application program that allows the user to log onto a remote computer and then use the services provided by that computer.

Server

A *server* is a program running on the remote machine providing service to the clients. When it starts, it opens the door for incoming requests from clients, but it never initiates a service until it is requested to do so.

A server program is an *infinite* program. When it starts, it runs infinitely unless a problem arises. It waits for incoming requests from clients. When a request arrives, it responds to the request, either iteratively or concurrently as we will see shortly.

Client

A *client* is a program running on the local machine requesting service from a server. A client program is *finite*, which means it is started by the user (or another application program) and terminates when the service is complete. Normally, a client opens the communication channel using the IP address of the remote host and the well-known port address of the specific server program running on that machine. After a channel of communication is opened, the client sends its request and receives a response. Although the request-response part may be repeated several times, the whole process is finite and eventually comes to an end.

Concurrency

Both clients and servers can run in concurrent mode.

Concurrency in Clients

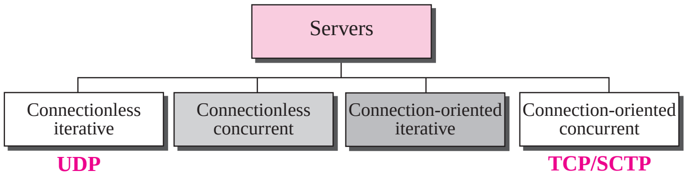
Clients can be run on a machine either iteratively or concurrently. Running clients *iteratively* means running them one by one; one client must start, run, and terminate before the machine can start another client. Most computers today, however, allow *concurrent* clients; that is, two or more clients can run at the same time.

Concurrency in Servers

An *iterative* server can process only one request at a time; it receives a request, processes it, and sends the response to the requestor before it handles another request. A concurrent server, on the other hand, can process many requests at the same time and thus can share its time between many requests.

The servers use either UDP, a connectionless transport layer protocol, or TCP/SCTP, a connection-oriented transport layer protocol. Server operation, therefore, depends on two factors: the transport layer protocol and the service method. Theoretically we can have four types of servers: connectionless iterative, connectionless concurrent, connection-oriented iterative, and connection-oriented concurrent (see Figure 17.1).

Figure 17.1 *Server types*

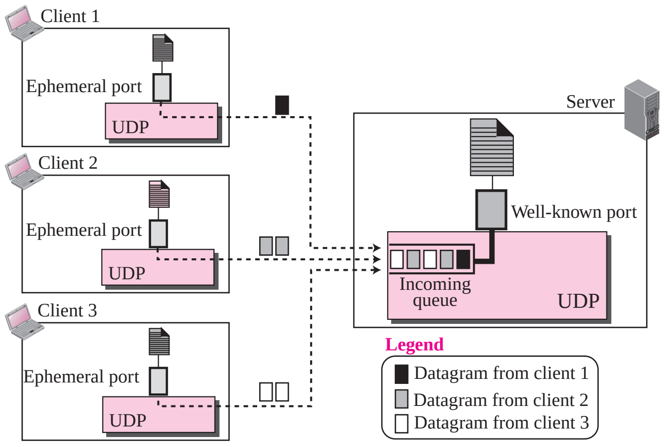

Servers
Connectionless iterative
Connectionless concurrent
Connection-oriented iterative
Connection-oriented concurrent
UDP
TCP/SCTP

Connectionless Iterative Server

The servers that use UDP are normally iterative, which, as we have said, means that the server processes one request at a time. A server gets the request received in a datagram from UDP, processes the request, and gives the response to UDP to send to the client. The server pays no attention to the other datagrams. These datagrams are stored in a queue, waiting for service. They could all be from one client or from many clients. In either case they are processed one by one in order of arrival.

The server uses one single port for this purpose, the well-known port. All the datagrams arriving at this port wait in line to be served, as is shown in Figure 17.2.

Figure 17.2 *Connectionless iterative server*



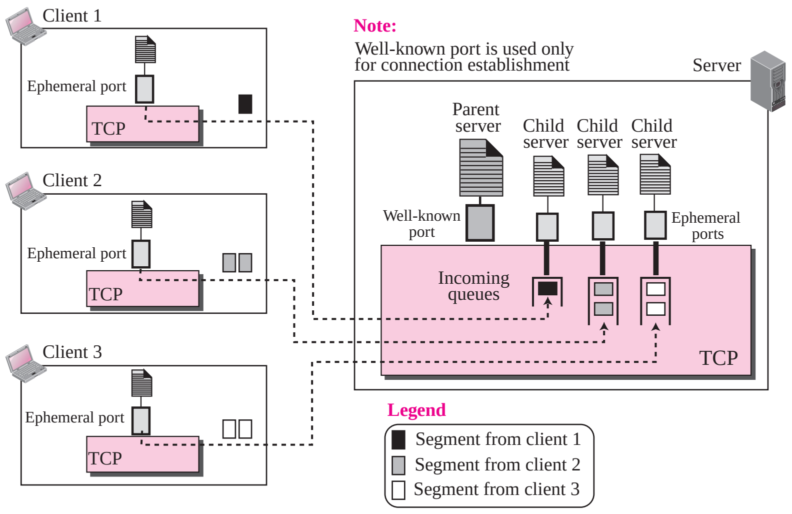
Connection-Oriented Concurrent Server

The servers that use TCP (or SCTP) are normally concurrent. This means that the server can serve many clients at the same time. Communication is connection-oriented, which means that a request is a stream of bytes that can arrive in several segments and the response can occupy several segments. A connection is established between the server and each client, and the connection remains open until the entire stream is processed and the connection is terminated.

This type of server cannot use only one port because each connection needs a port and many connections may be open at the same time. Many ports are needed, but a server can use only one well-known port. The solution is to have one well-known port and many ephemeral ports. The server accepts connection requests at the well-known port. A client can make its initial approach to this port to make the connection. After the connection is made, the server assigns a temporary port to this connection to free the well-known port. Data transfer can now take place between these two temporary ports, one at the client site and the other at the server site. The well-known port is now free for another client to make the connection. To serve several clients at the same time, a server creates child processes, which are copies of the original process (parent process).

The server must also have one queue for each connection. The segments come from the client, are stored in the appropriate queue, and will be served concurrently by the server. See Figure 17.3 for this configuration.

Figure 17.3 *Connection-oriented concurrent server*



Socket Interfaces

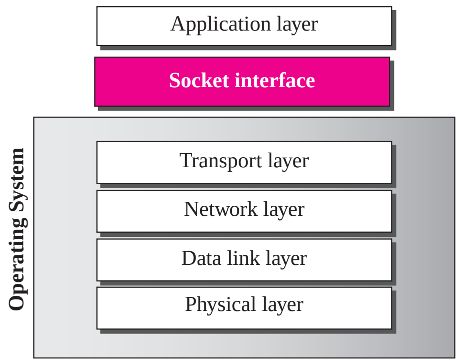
How can a client process communicate with a server process? A computer program is a set of predefined instructions that tells the computer what to do. A computer program has a set of instructions for mathematical operations, another set of instructions for string manipulation, still another set of instructions for input/output access. If we need a program to be able to communicate with another program running on another machine, we need a new set of instructions to tell the transport layer to open the connection, send data to and receive data from the other end, and close the connection. A set of instructions of this kind is normally referred to as an **interface.**

An interface is a set of instructions designed for interaction between two entities.

Several interfaces have been designed for communication. Three among them are common: **socket interface**, **transport layer interface** (**TLI**), and **STREAM.** Although a network programmer needs to be familiar with all of these interfaces, we briefly discuss only the socket interface in this chapter to give the general idea of network communication at the application layer.

Socket interface started in early 1980s at the University of Berkeley as part of a UNIX environment. To better understand the concept of socket interface, we need to consider the relationship between the underlying operating system, such as UNIX or Windows, and the TCP/IP protocol suite. The issue that we have ignored so far. Figure 17.4 shows a conceptual relation between the operating system and the suite.

Figure 17.4 *Relation between the operating system and the TCP/IP suite*



The socket interface, as a set of instructions, is located between the operating system and the application programs. To access the services provided by the TCP/IP protocol suite, an application needs to use the instructions defined in the socket interface.

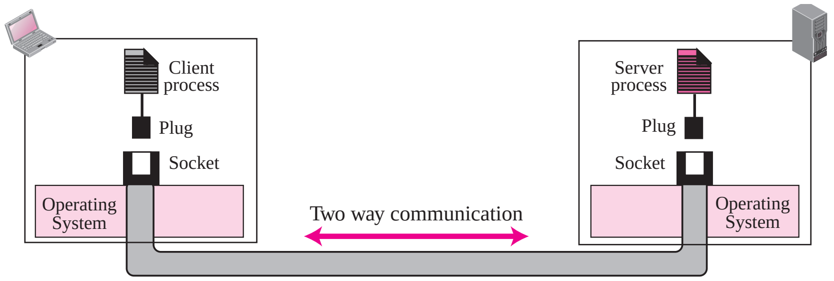
Example 17.1

Most of the programming languages have a *file interface*, a set of instructions that allow the programmer to open a file, read from the file, write to the file, perform other operations on the file, and finally close the file. When a program needs to open the file, it uses the name of the file as it is known to the operation system. When the file is opened, the operating system returns a reference to the file (an integer or pointer) that can be used for other instructions, such as read and write.

Socket

A **socket** is a software abstract simulating a hardware socket we see in our daily life. To use the communication channel, an application program (client or server) needs to request the operating system to create a socket. The application program then can *plug* into the socket to send and receive data. For data communication to occur, a pair of sockets, each at one end of communication, is needed. Figure 17.5 simulates this abstraction using the socket and plug that we use in our daily life (for a telephone, for example); in the Internet a socket is a software data structure as we discuss shortly.

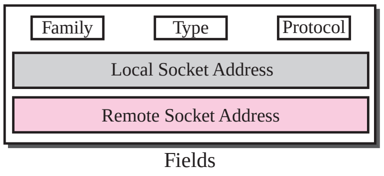
Figure 17.5 *Concept of sockets*



Data Structure

The format of data structure to define a socket depends on the underlying language used by the processes. For the rest of this chapter, we assume that the processes are written in C language. In C language, a socket is defined as a five-field structure (struct or record) as shown in Figure 17.6.

Figure 17.6 *Socket data structure*



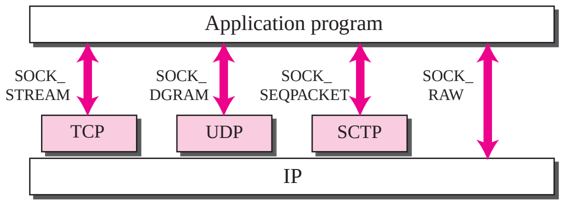
```
struct  socket
{
    int family;
    int type;
    int protocol;
    socketaddr local;
    socketaddr remote;
};
```

Generic definition

Note that the programmer should not redefine this structure; it is already defined. The programmer needs only to use the header file that includes this definition (discussed later). Let us briefly define the field uses in this structure:

- **Family.** This field defines the protocol group: IPv4, IPv6, UNIX domain protocols, and so on. The family type we use in TCP/IP is defined by the constant IF_INET for IPv4 protocol and IF_INET6 for IPv6 protocol.
- **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. They are shown in Figure 17.7.
- **Protocol.** This field defines the protocol that uses the interface. It is set to 0 for TCP/IP protocol suite.
- **Local socket address.** This field defines the local socket address. A socket address, as discussed in Chapter 13, is a combination of an IP address and a port number.
- **Remote socket address.** This field defines the remote socket address.

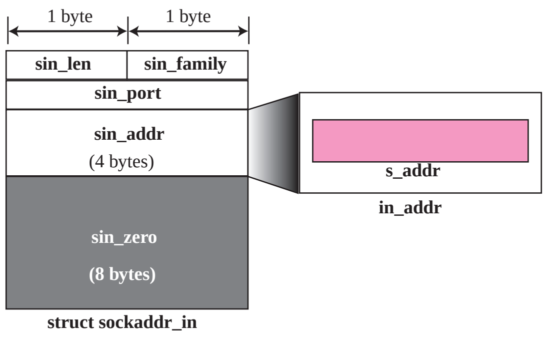
Figure 17.7 *Socket types*



Structure of a Socket Address

Before we can use a socket, we need to understand the structure of a socket address, a combination of IP address and port number. Although several types of socket addresses have been defined in network programming, we define only the one used for IPv4. In this version a socket address is a complex data structure (struct in C) as shown in Figure 17.8.

Figure 17.8 *IPv4 socket address*



Note that the struct ***sockaddr_in*** has five fields, but the ***sin_addr*** field is itself a struct of type **in_addr** with a one single field ***s_addr.*** We first define the structure ***in_addr*** as shown below:

```
struct   in_addr
{
     in_addr_t              s_addr;                  // A 32-bit IPv4 address
}
```

We now define the structure of **sockaddr_in**:

```
struct  sockaddr_in
{
    uint8_t            sin_len;        // length of structure (16 bytes)
    sa_family_t        sin_family;     // set to AF_INET
    in_port_t          sin_port;       // A 16-bit port number
    struct in_addr     sin_addr;       // A 32-bit IPv4 address
    char               sin_zero[8];    // unused
}
```

Functions

The interaction between a process and the operating system is done through a list of predefined functions. In this section, we introduce these functions; in later sections, we show how they are combined to create processes.

- **The *socket* Function**

  The operating system defines the socket structure shown in Figure 17.6. The operating system, however, does not create a socket until instructed by the process. The process needs to use the *socket* function call to create a socket. The prototype for this function is shown below:

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

  A call to this function creates a socket, but only three fields in the socket structure (family, type, and protocol) are filled. If the call is successful, the function returns a unique socket descriptor *sockfd* (a non-negative integer) that can be used to refer to the socket in other calls; if the call is not successful, the operating system returns –1.

- **The *bind* Function**

  The socket function fills the fields in the socket partially. To bind the socket to the local computer and local port, the *bind* function needs to be called. The bind function, fills the value for the local socket address (local IP address and local port number). It returns –1 if the binding fails. The prototype is shown below:

  ```
  int bind (int sockfd, const struct sockaddress* localAddress, socklen_t addrLen);
  ```

  In this prototype, *sockfd* is the value of the socket descriptor returned from the socket function call, *localAddress* is a pointer to a socket address that needs to have been defined (by the system or the programmer), and the *addrLen* is the length of the socket address. We will see later when the bind function is and is not used.

The *connect* Function

The connect function is used to add the remote socket address to the socket structure. It returns –1 if the connection fails.The prototype is given below:

```
int connect (int sockfd, const struct sockaddress* remoteAddress, socklen_t addrLen);
```

The argument is the same, except that the second and third argument defines the remote address instead of the local one.

The *listen* Function

The listen function is called only by the TCP server. After TCP has created and bound a socket, it must inform the operating system that a socket is ready for receiving client requests. This is done by calling the *listen* function. The backlog is the maximum number of connection requests. The function returns –1 if it fails. The following shows the prototype:

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

The *accept* Function

The accept function is used by a server to inform TCP that it is ready to receive connections from clients. This function returns –1 if it fails. Its prototype is shown below:

```
int accept (int sockfd, const struct sockaddress* clientAddr, socklen_t* addrLen);
```

The last two arguments are pointers to address and to length. The accept function is a blocking function that, when called, blocks itself until a connection is made by a client. The accept function then gets the client socket address and the address length and passes it to the server process to be used to access the client. Note several points:

a. The call to accept function makes the process check if there is any client connection request in the waiting buffer. If not, the accept makes the process to sleep. The process wakes up when the queue has at least one request.
b. After a successful call to the accept, a new socket is created and the communication is established between the client socket and the new socket of the server.
c. The address received from the *accept* function fills the remote socket address in the new socket.
d. The address of the client is returned via a pointer. If the programmer does not need this address, it can be replaced by NULL.
e. The length of address to be returned is passed to the function and also returned via a pointer. If this length is not needed, it can be replaced by NULL.

The *fork* function

The *fork* function is used by a process to duplicate a process. The process that calls the *fork* function is referred to as the parent process; the process that is created, the duplicate, is called the child process. The prototype is shown below:

```
pid_t fork (fork);
```

It is interesting to know that the fork process is called once, but it returns twice. In the parent process, the return value is a positive integer (the process ID of the parent process that called it). In the child process, the return value is 0. If there is an error, the fork function returns –1. After the fork, two processes are running concurrently; the CPU gives running time to each process alternately.

- **The *send* and *recv* Functions**

The *send* function is used by a process to send data to another process running on a remote machine. The *recv* function is used by a process to receive data from another process running on a remote machine.These functions assume that there is already an open connection between two machines; therefore, it can only be used by TCP (or SCTP). These functions returns the number of bytes send or receive.

```
int send (int sockfd, const void* sendbuf, int nbytes, int flags);

int recv (int sockfd, void* recvbuf, int nbytes, int flags);
```

Here *sockfd* is the socket descriptor; *sendbuf* is a pointer to the buffer where data to be sent have been stored; *recvbuf* is a pointer to the buffer where the data received is to be stored. *nbytes* is the size of data to be sent or received. This function returns the number of actual bytes sent or received if successful and –1 if there is an error.

- **The *sendto* and *recvfrom* Functions**

The *sendto* function is used by a process to send data to a remote process using the services of UDP. The *recvfrom* function is used by a process to receive data from a remote process using the services of UDP. Since UDP is a connectionless protocol, one of the arguments defines the remote socket address (destination or source).

```
int sendto (int sockfd, const void* buffer, int nbytes, int flags
                  struct sockaddr* destinationAddress, socklen_t addrLen);

int recvfrom (int sockfd, void* buffer, int nbytes, int flags
                  struct sockaddr* sourceAddress, socklen_t* addrLen);
```

Here *sockfd* is the socket descriptor, *buffer* is a pointer to the buffer where data to be sent or to be received is stored, and *buflen* is the length of the buffer. Although, the value of the flag can be nonzero, we let it be 0 for our simple programs in this chapter. These functions return the number of bytes sent or received if successful and –1 if there is an error.

❑ The *close* Function

The close function is used by a process to close a socket.

```
int close (int sockfd);
```

The *sockfd* is not valid after calling this function. The socket returns an integer, 0 for success and –1 for error.

❑ Byte Ordering Functions

Information in a computer is stored in *host byte order*, which can be *little-endian*, in which the little-end byte (least significant byte) is stored in the starting address, or *big-endian*, in which the big-end byte (most significant byte) is stored in the starting address. Network programming needs data and other pieces of information to be in *network byte order*, which is big-endian. Since when we write programs, we are not sure, how the information such as IP addresses and port number are stored in the computer, we need to change them to network byte order. Two functions are designed for this purpose: ***htons*** (host to network short), which changes a short (16-bit) value to a network byte order, and ***htonl*** (host to network long), which does the same for a long (32-bit) value. There are also two functions that do exactly the opposite: ***ntohs*** and ***ntohl.*** The prototype of these functions are shown below:

```
uint16_t htons (uint16_t shortValue);

uint32_t htonl (uint32_t longValue);

uint16_t ntohs (uint16_t shortValue);

uint32_t ntohl (uint32_t longValue);
```

❑ Memory Management Functions

Finally, we need some functions to manage values stored in the memory. We introduce three common memory functions here, although we do not use all of them in this chapter.

```
void* memset (void* destination, int chr, size_t len);

void* memcpy (void* destination, const void* source, size_t nbytes);

int memcmp (const void* ptr1, const void* ptr2, size_t nbytes);
```

The first function, *memset* (memory set) is used to set (store) a specified number of bytes (value of *len*) in the memory defined by the destination pointer (starting address). The second function, *memcpy* (memory copy) is used to copy a specified number of bytes (value of *nbytes*) from part of a memory (source) to another part of memory (destination). The third function, *memcmp* (memory compare), is used to compare two sets of bytes (nbytes) starting from ptr1 and ptr2. The result is 0 if two sets are equal, less than zero if the first set is smaller than the second, and greater than zero if the first set is larger than the second. The comparison is based on comparing strings of bytes in the C language.

- **Address Conversion Functions**

  We normally like to work with 32-bit IP address in dotted decimal format. When we want to store the address in a socket, however, we need to change it to a number. Two functions are used to convert an address from a presentation to a number and vice versa: ***inet_pton*** (presentation to number) and ***inet_ntop*** (number to presentation). The constant use for family value is AF_INET for our purpose. Their prototypes are shown below:

```
int inet_pton (int family, const char* stringAddr, void* numericAddr);

char* inet_ntop (int family, const void* numericAddr, char* stringAddr, int len);
```

Header Files

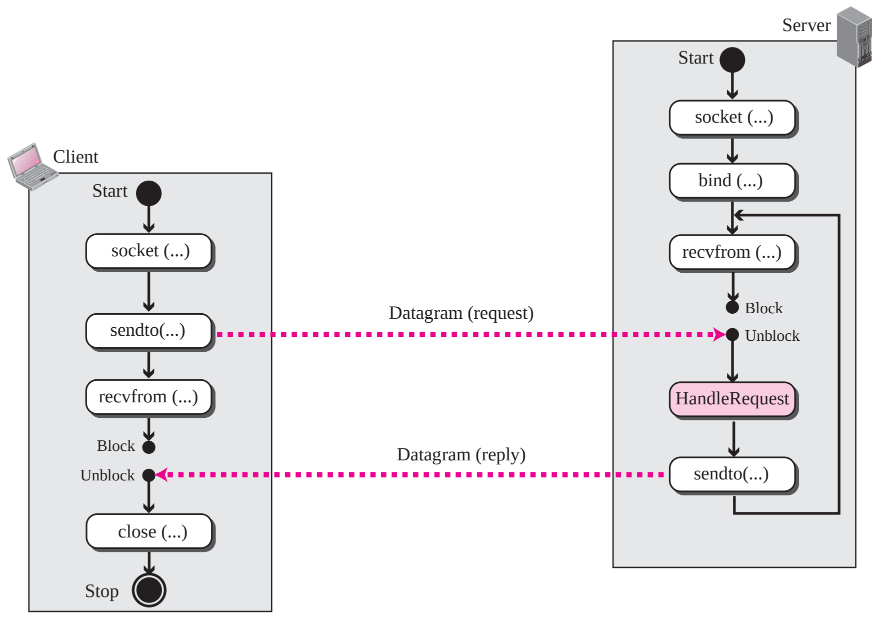
To use previously described functions, we need a set of header files. We define this header file in a separate file, which we name "headerFiles.h". We include this file in our programs to avoid including long lists of header files. Not all of these header files may be needed in all programs, but it is recommended to include all of them in case.

```
//  "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>
```

Communication Using UDP

Figure 17.9 shows a simplified flow diagram for this type of communication.

Figure 17.9 *Connectionless iterative communication using UDP*



Server Process

The server process starts first. The server process calls the *socket* function to create a socket. It then calls the *bind* function to bind the socket to its well-known port and the IP address of the computer on which the server process is running. The server then calls the *recvfrom* function, which blocks until a datagram arrives. When the datagram arrives, the *recvfrom* function unblocks, extracts the client socket address and address length from the received datagram, and returns them to the process. The process saves these two pieces of information and calls a procedure (function) to handle the request. When the result is ready, the server process calls the *sendto* function and uses the saved information to send the result to the client that requested it. The server uses an infinite loop to respond to the requests coming from the same client or different clients.

Client Process

The client process is simpler. The client calls the *socket* function to create a socket. It then calls the *sendto* function and pass the socket address of the server and the location of the buffer from which UDP can get the data to make the datagram. The client then calls a *recvfrom* function call that blocks until the reply arrives from the server. When the reply arrives, UDP delivers the data to the client process, which make the recv function to unblock and deliver the data received to the client process. Note that we assume that the client message is so small that it fits into one single datagram. If this is not the case, the two function calls, *sendto* and *recvfrom*, need to be repeated. However, the server is not aware of multidatagram communication; it handles each request separately.

Example 17.2

As an example, let us see how we can design and write two programs: an *echo* server and an *echo* server. The client sends a line of text to the server; the server sends the same line back to the client. Although this client/server pair looks useless, it has some applications. It can be used, for example, when a computer wants to test if another computer in the network is alive. To better understand the code in a program, we first give the layout of variables used in both programs as shown in Figure 17.10.

Figure 17.10 *Variables used in echo server and echo client using UDP service*

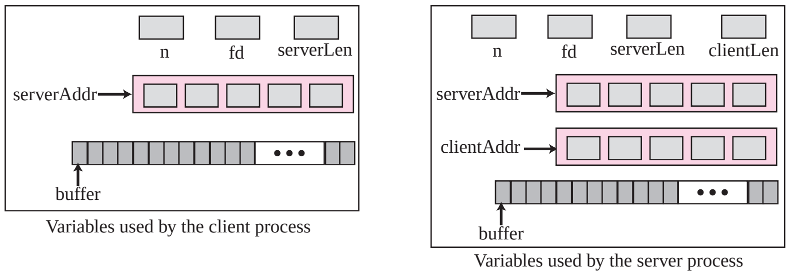


Table 17.1 shows the program for the echo server. We have eliminated many details and error-handling to simplify the program. We ask the reader to provide more details in exercises.

Table 17.1 *Echo Server Program using the Service of UDP*

```
// UDP echo server program
#include "headerFiles.h"

int main (void)
{
      // Declaration and definition
      int sd;                                      // Socket descriptor
      int nr;                                      // Number of bytes received
      char buffer [256];                           // Data buffer
      struct sockaddr_in serverAddr;               // Server address
      struct sockaddr_in clientAddr;               // Client address
      int clAddrLen;                               // Length of client Address
      // Create socket
      sd = socket (PF_INET, SOCK_DGRAM, 0);
      // Bind socket to local address and port
      memset (&serverAddr, 0, sizeof (serverAddr));
      serverAddr.sin_family = AF_INET;
```

Table 17.1 *Echo Server Program using the Service of UDP (continued)*

```
18        serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);        // Default address
19        serverAddr.sin_port = htons (7)     // We assume port 7
20        bind (sd, (struct sockaddr*) &serverAddr, sizeof (serverAddr));
21        // Receiving and echoing datagrams
22        for ( ; ; )         // Run forever
23        {
24            nr = recvfrom (sd, buffer, 256, 0, (struct sockaddr*)&clientAddr, &clAddrLen);
25            sendto (sd, buffer, nr, 0, (struct sockaddr*)&clientAddr, sizeof(clientAddr));
26        }
27    } // End of echo server program
```

Line 14 creates a socket. Lines 16 to 19 create the local socket address; the remote socket address will be created in line 24 as explained later. Line 20 binds the socket to the local socket address. Lines 22 to 26 receive and send datagrams, possibly from many clients. When the server receives a datagram from a client, the client socket address and the length of the socket address are returned to the server. These two pieces of information are used in line 34 to send the echo message to the corresponding client. Note that we have eliminated many error-checking lines of codes; they have left as exercises.

Table 17.2 shows the client program for an echo process. We have assumed that the client sends only one datagram to be echoed by the server. If we need to send more than one datagram, the data transfer sections should be repeated using a loop.

Table 17.2 *Echo Client Program using the Service of UDP*

```
01    // UDP echo client program
02    #include "headerFiles.h"
04
04    int main (void)
05    {
06        // Declaration and definition
07        int sd;                              // Socket descriptor
08        int ns;                              // Number of bytes send
09        int nr;                              // Number of bytes received
10        char buffer [256];                   // Data buffer
11        struct sockaddr_in serverAddr;       // Socket address
11        // Create socket
12        sd = socket (PF_INET, SOCK_DGRAM, 0);
13        // Create server socket address
14        memset (&servAddr, 0, sizeof(serverAddr));
15        servAddr.sin_family = AF_INET;
16        inet_pton (AF_INET, “server address”, &serverAddr.sin_addr);
17        serverAddr.sin_port = htons (7);
18        // Send and receive datagrams
19        fgets (buffer, 256, stdin);
```

Table 17.2 *Echo Client Program using the Service of UDP (continued)*

```
20          ns = sendto (sd, buffer, strlen (buffer), 0,
21                  (struct sockaddr)&serverAddr, sizeof(serveraddr));
22          recvfrom (sd, buffer, strlen (buffer), 0, NULL, NULL);
23          buffer [nr] = 0;
24          printf ("Received from server:");
25          fputs (buffer, stdout);
26          // Close and exit
27          close (sd);
28          exit (0);
29  } // End of echo client program
```

Line 12 creates a socket. Lines 14 to 17 show how we create the server socket address; there is no need to create the client socket address. Lines 19 to 25 read a string from the keyboard, send it to the server, and receive it back. Line 23 adds a null character to the received string to make it displayable in line 25. In line 27, we close the socket. We have eliminated all error checking; we leave them as exercises.

To be complete, error-checking code needs to be added to both server and client programs.

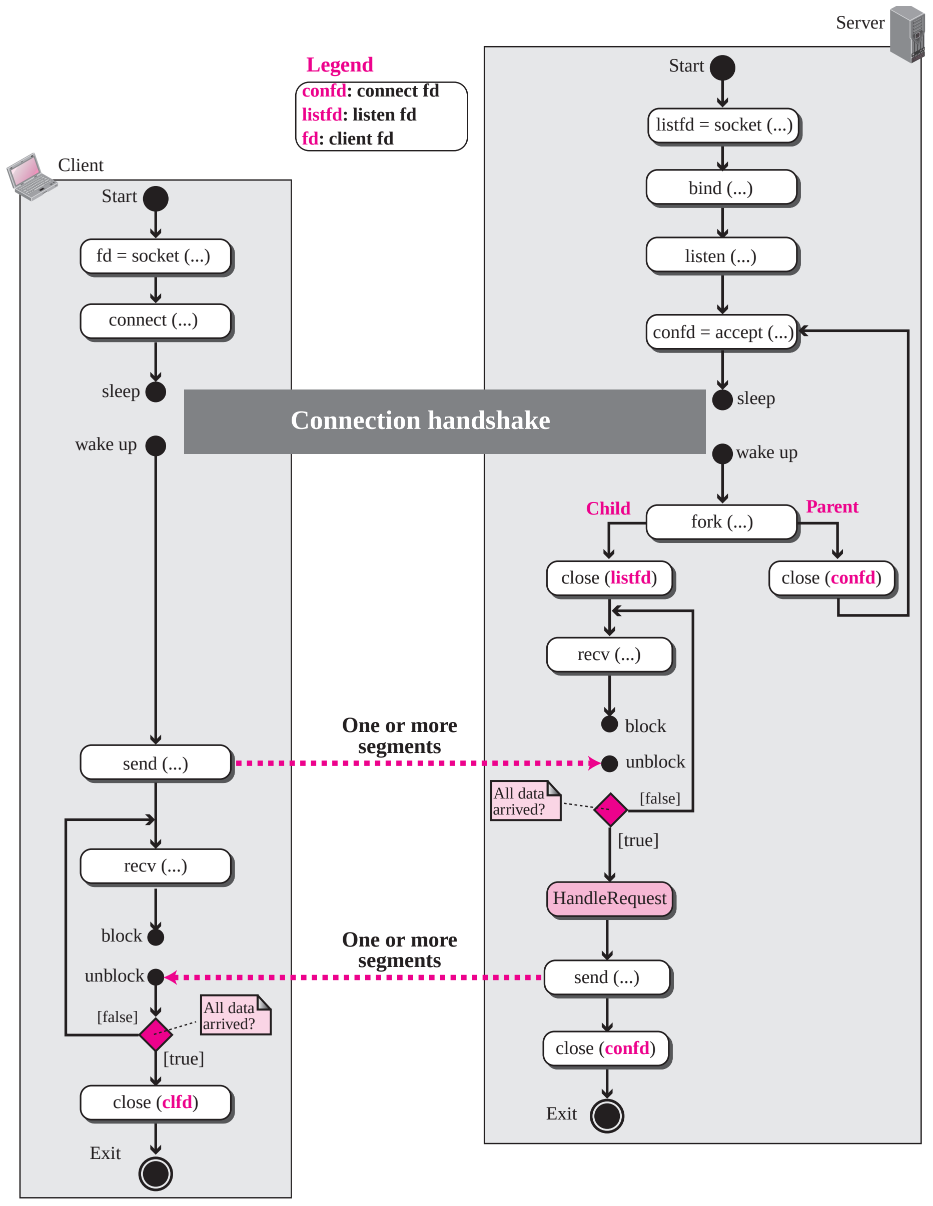
Communication Using TCP

Now we discuss connection-oriented, concurrent communication using the service of TCP (the case of SCTP would be similar). Figure 17.11 shows the general flow diagram for this type of communication.

Server Process

The server process starts first. It calls the *socket* function to create a socket, which we call the *listen* socket. This socket is only used during connection establishment. The server process then calls the *bind* function to bind this connection to the socket address of the server computer. The server program then calls the *accept* function. This function is a blocking function; when it is called, it is blocked until the TCP receives a connection request (SYN segment) from a client. The *accept* function then is unblocked and creates a new socket called the connect socket that includes the socket address of the client that sent the SYN segment. After the *accept* function is unblocked, the server knows that a client needs its service. To provide concurrency, the server process (parent process) calls the *fork* function. This function creates a new process (child process), which is exactly the same as the parent process. After calling the *fork* function, the two processes are running concurrently, but each can do different things. Each process now has two sockets: listen and connect sockets. The parent process entrusts the duty of serving the client to the hand of the child process and calls the accept function again to wait for another client to request connection. The child process is now ready to serve the client. It first closes the listen socket and calls the *recv* function to receive data from the client. The *recv* function, like the *recvfrom* function, is a blocking

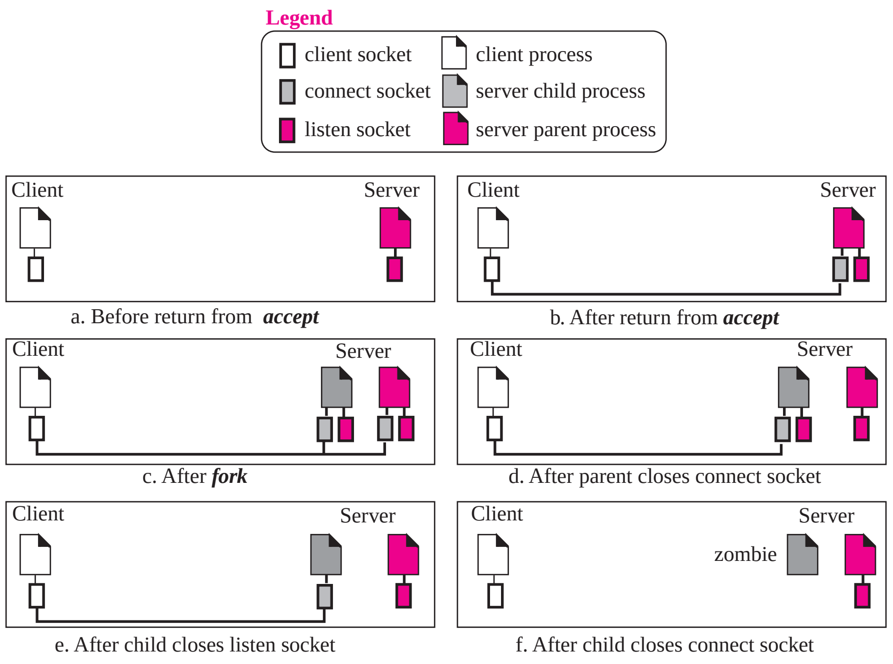
Figure 17.11 *Flow diagram for connection-oriented, concurrent communication*



function; it is blocked until a segment arrives. The child process uses a loop and calls the *recv* function repeatedly until it receives all segments sent by the client. The child process then gives the whole data to a function (we call it *handleRequest*), to handle the request and return the result. The result is then sent to the client in one single call to the *send* function. We need to emphasize several points here. First, the flow diagram we are using is the simplest possible one. The server may use many other functions to receive and send data, choosing the one which is appropriate for a particular application. Second, we assume that size of data to be sent to the client is so small that can be sent in one single call to the *send* function; otherwise, we need a loop to repeatedly call the *send* function. Third, although the server may send data using one single call to the *send* function, TCP may use several segments to send the data. The client process, therefore, may not receive data in one single segment, as we will see when we explain the client process.

Figure 17.12 shows the status of the parent and child process with respect to the sockets. Part a in the figure shows the status before the *accept* function returns. The parent process uses the *listen* socket to wait for request from the clients. When the accept function is blocked and returned (part b), the parent process has two sockets: the *listen* and the *connect* sockets. The client is connected to the *connect* socket. After calling the *fork* function (part c), we have two processes, each with two sockets. The client is connected to both processes.The parent needs to close its *connect* socket to free itself from the client and be free to listen to requests from other clients (part d). Before the child can start serving the connected client, it needs to close its *listen* socket so that a future request does not affect it (part e). Finally, when the child finishes serving the connected client, it needs to close its connect socket to disassociate itself from the client that has been served (part f).

Figure 17.12 *Status of parent and child processes with respect to the sockets*


Legend
client socket
connect socket
listen socket
client process
server child process
server parent process
Client
Server
a. Before return from ***accept***
b. After return from ***accept***
c. After ***fork***
d. After parent closes connect socket
e. After child closes listen socket
zombie
f. After child closes connect socket

Although we do not include the operation in our program (for simplicity), the child process, after serving the corresponding process, needs to be destroyed. The child process that has done its duty and is dormant is normally called a zombie in the UNIX environment system. A child can be destroyed as soon as it is not needed. Alternatively, the system can run a special program once in a while to destroy all zombies in the system. The zombies occupy space in the system and can affect the performance of the system.

Client Process

The client process is simpler. The client calls the *socket* function to create a socket. It then calls the *connect* function to request a connection to the server. The *connect* function is a blocking function; it is blocked until the connection is established between two TCPs. When the *connect* function returns, the client calls the *send* function to send data to the server. We use only one call to the *send* function, assuming that data can be sent with one call. Based on the type of the application, we may need to call this function repeatedly (in a loop). The client then calls the *recv* function, which is blocked until a segment arrives and data are delivered to the process by TCP. Note that, although the data are sent by the server in one single call to the *send* function, the TCP at the server site may have used several segments to send data. This means we may need to call the *recv* function repeatedly to receive all data. The loop can be controlled by the return value of the *recv* function.

Example 17.3

We want to write two programs to show how we can have an echo client and echo server using the services of TCP. Figure 17.13 shows the variables we use in these two programs. Since data may arrive in different chunks, we need pointers to point to the buffer. The first buffer is fixed and always points to the beginning of the buffer; the second pointer is moving to let the arrived bytes be appended to the end of the previous section.

Figure 17.13 *Variable used Example 17.3.*

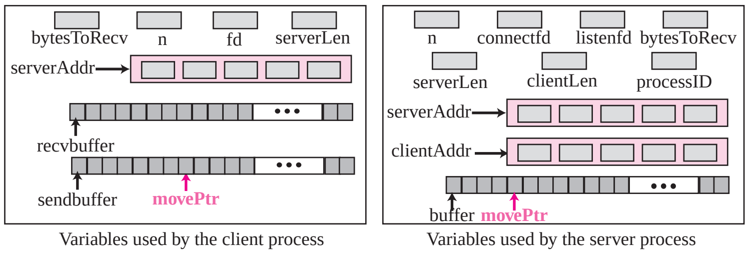


Table 17.3 shows the server program for an echo server that uses the services of TCP. We have eliminated many details and left them for the books on the network programming. We want just to show a global picture of the program.

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*

```
// TCP echo client program
#include "headerFiles.h"

int main (void)
{
        // Declaration and definition
        int  sd;                                // Socket descriptor
        int  n;                                 // Number of bytes received
        int  bytesToRecv;                       // Number of bytes to receive
        char  sendBuffer [256];                 // Send buffer
        char  recvBuffer [256];                 // Receive buffer
        char* movePtr;                          // A pointer the received buffer
        struct sockaddr_in  serverAddr;         // Server address

        // Create socket
        sd = socket (PF_INET, SOCK_STREAM, 0);
        // Create server socket address
        memset (&serverAddr, 0, sizeof(serverAddr);
        serverAddr.sin_family = AF_INET;
        inet_pton (AF_INET, “server address”, &serverAddr.sin_addr);
        serverAddr.sin_port = htons (7);     // We assume port 7
        // 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 actually is to let the duty of a server be shared by some users that want to participate in the process. For example, instead of allowing several clients to make a connection and each download a large file, a server can let each client download a part of a file and then share it with each other. In the process of downloading part of the file or sharing the downloaded file, however, a computer needs to play the role of a client and the other the role of a server. In other words, a computer can be a client for a specific application at one moment and the server at another moment. These applications are now controlled commercially and not formally part of the Internet. We leave the exploration of these applications to the books designed for each specific application.

17.3 FURTHER READING

Several books give thorough coverage of network programming. In particular, we recommend [Ste et al. 04], [Com 96], [Rob & Rob 96], and [Don & Cal 01].

17.4 KEY TERMS

client-server paradigm	socket interface
interface	STREAM
peer-to-peer (P2P) paradigm	transport-layer interface (TLI)
socket	

17.5 SUMMARY

- Most applications in the Internet are designed using a client-server paradigm in which an application program, called a server, provides services and another application program, called a client, receives services. A server program is an infinite program. When it starts it runs infinitely unless a problem arises. It waits for incoming requests from clients. A client program is finite, which means it is started by the user and terminates when the service is complete. Both clients and servers can run in concurrent mode.
- Clients can be run on a machine either iteratively or concurrently. An iterative server can process only one request at a time. A concurrent server, on the other hand, can process many requests at the same time.
- Client-server paradigm is based on a set of predefined functions called an interface. We discussed the most common interface, called socket interface, in this chapter. The socket interface, as a set of instructions, is located between the operating stem and the application programs. A socket is a software abstract simulating the hardware socket we see in our daily life. To use the communication channel, an application program (client or server) needs to request the operating system to create a socket.