CHAPTER 24

Socket Interface

In a client-server model, two application programs, one running on the local system (a client for example) and the other running on the remote system (a server for example), need to communicate with one another. To standardize network programming, application programming interfaces (APIs) have been developed. An API is a set of declarations, definitions, and procedures followed by programmers to write client-server programs. Among the more common APIs are the Socket Interface, the Transport Layer Interface (TLI), the Stream Interface, the Thread Interface, and the Remote Procedure Call (RPC). The Socket Interface, which is very common today, is the implementation we will discuss in this chapter.

The Socket Interface was developed as part of UNIX BSD. It is based on UNIX and defines a set of system calls (procedures) that are an extension of system calls used in UNIX to access files.This chapter shows the fundamentals of Socket Interface programming though it by no means teaches Socket Interface programming; there are whole books devoted to this subject. Instead, we introduce the concept and idea, and, maybe, provide motivation for those readers who want to learn more.

In this chapter, we first introduce some data types and functions used in network programming. Then we define sockets and introduce socket interface calls. Finally we give two pairs of examples of client-server programs.

24.1 SOME DEFINITIONS

In this section, we introduce some data types and structures that are needed for writing client-server programs.

Data Types Defined

Figure 24.1 lists three data types used extensively in client-server programs. These are u_char, u_short, and u_long data types.

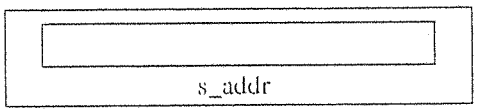
Figure 24.1 *Data types*

u_char	Unsigned 8-bit character
u_short	Unsigned 16-bit integer
u_long	Unsigned 32-bit integer

Internet Address Structure

An IPv4 address is defined as a structure (struct in C) called *in_addr*, which contains only one field called *s_addr* of type u_long. The structure holds an IP address as a 32-bit binary number. Figure 24.2 shows the structure and the corresponding declaration.

Figure 24.2 *Internet address structure*



in_addr

```
struct   in_addr
{
        u_long  s_addr ;
} ;
```

Internet Socket Address Structure

The application programs that use the TCP/IP protocol suite need a structure called a **socket address,** which mainly holds an IP address, a port number, and the protocol family. The structure is called *sockaddr_in* and has five fields: the first and the last field are normally not used, however. Figure 24.3 shows the structure and its declaration.

24.2 SOCKETS

The communication structure that we need in socket programming is a **socket.** A socket acts as an end point. Two processes need a socket at each end to communicate with each other.

A socket is defined in the operating system as a structure. Figure 24.4 shows a simplified version of a socket structure with five fields. These fields are listed below.

- **Family.** This field defines the protocol group: IPv4, IPv6, UNIX domain protocols, and so on.

Figure 24.3 *Socket address structure*

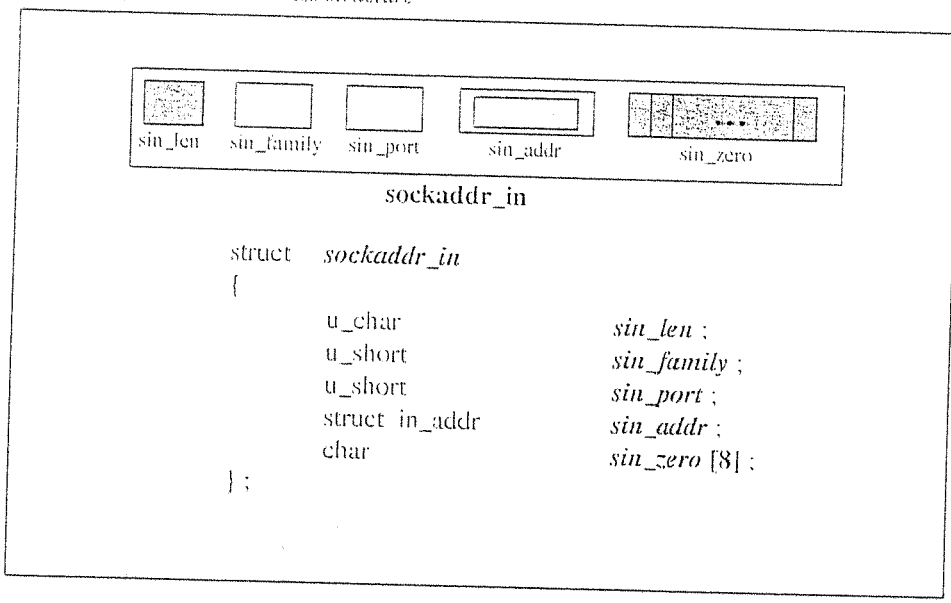
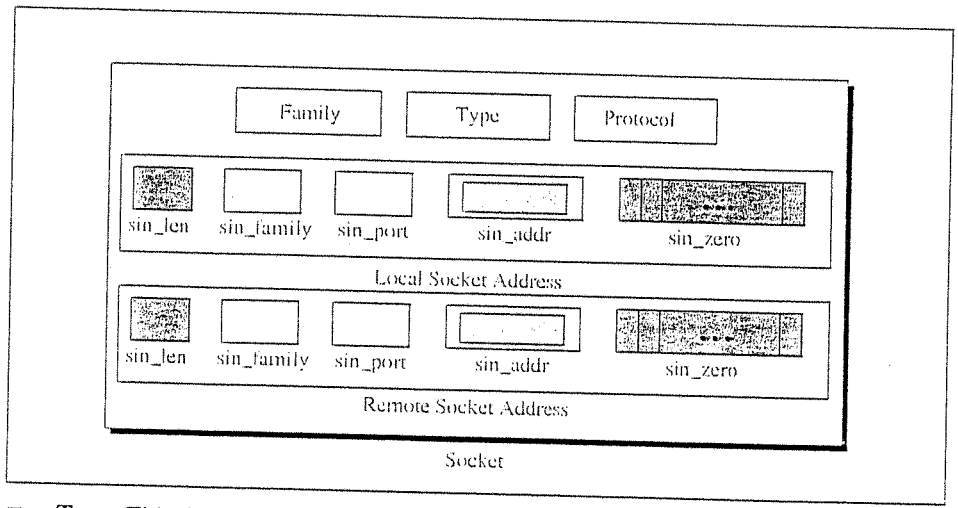


Figure 24.4 *Socket structure*

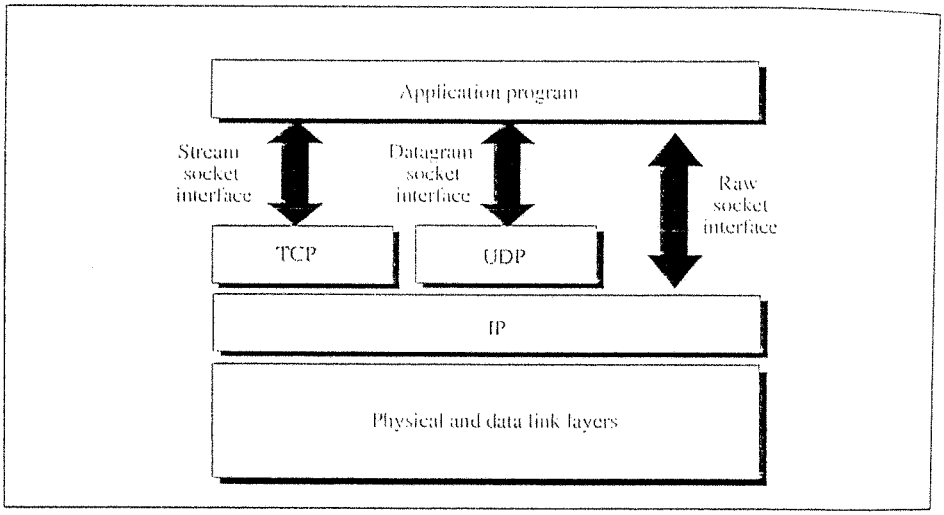


- **Type.** This field defines the type of socket: stream socket, datagram socket, or raw socket. These are discussed in the next section.
- **Protocol.** This field is usually set to zero for TCP and UDP.
- **Local socket address.** This field defines the local socket address, a structure of type sockaddr_in, as defined previously.
- **Remote socket address.** This field defines the remote socket address, a structure of type sockaddr_in, which was defined previously.

Socket Types

The socket interface defines three types of sockets: the stream socket, the datagram socket, and the raw socket. All three types can be used in a TCP/IP environment (see Figure 24.5).

Figure 24.5 *Socket types*



Stream Socket

A stream socket is designed to be used with a connection-oriented protocol such as TCP. TCP uses a pair of stream sockets to connect one application program to another across the Internet.

Datagram Socket

A datagram socket is designed to be used with a connectionless protocol such as UDP. UDP uses a pair of datagram sockets to send a message from one application program to another across the Internet.

Raw Socket

Some protocols such as ICMP or OSPF that directly use the services of IP use neither stream sockets nor datagram sockets. Raw sockets are designed for these types of applications.

24.3 BYTE ORDERING

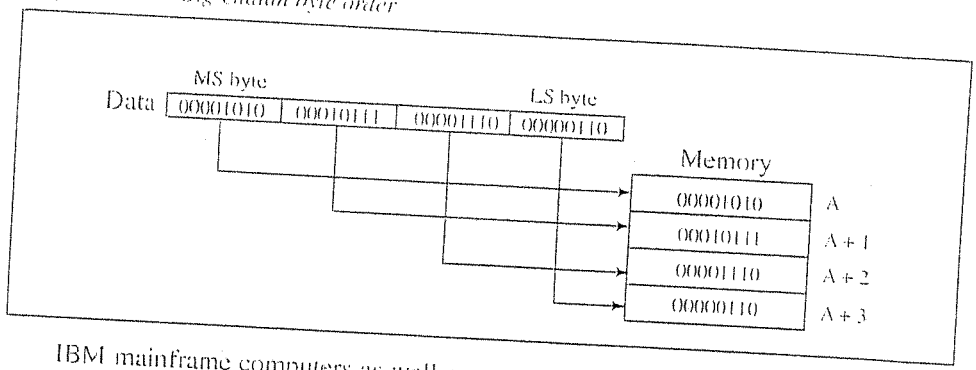
Computers can be classified by the way they store data in their internal memories. Memories are addressed byte by byte. A data unit can span more than one byte, however. For example, in most computers, a short integer is two bytes (16 bits) and a long

integer is four bytes (32 bits). How a two-byte short integer or a four-byte long integer is stored in bytes of memory defines the category of computer: big-endian or little-endian.

Big-Endian Byte Order

A computer that uses the **big-endian** system stores the most significant byte (the big end) of data in the starting address of the data unit. For example, an IP address such as 10.23.14.6, when expressed as a 32-bit binary number (long integer), can be stored in a big-endian computer as shown in Figure 24.6.

Figure 24.6 *Big-endian byte order*

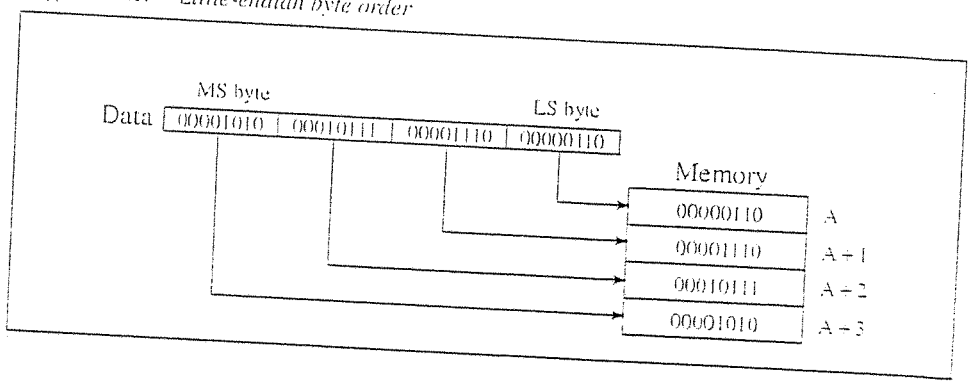


IBM mainframe computers as well as computers based on Motorola microprocessors are based on the big-endian system.

Little-Endian Byte Order

If a computer uses the **little-endian** system, it stores the least significant byte (the little end) of data in the starting address of the data unit. For example, an IP address such as 10.23.14.6, when expressed as a 32-bit binary number (long integer), can be stored in a little-endian computer as shown in Figure 24.7.

Figure 24.7 *Little-endian byte order*



DEC VAX computers and computers using Intel microprocessors are based on the little-endian system.

Network Byte Order

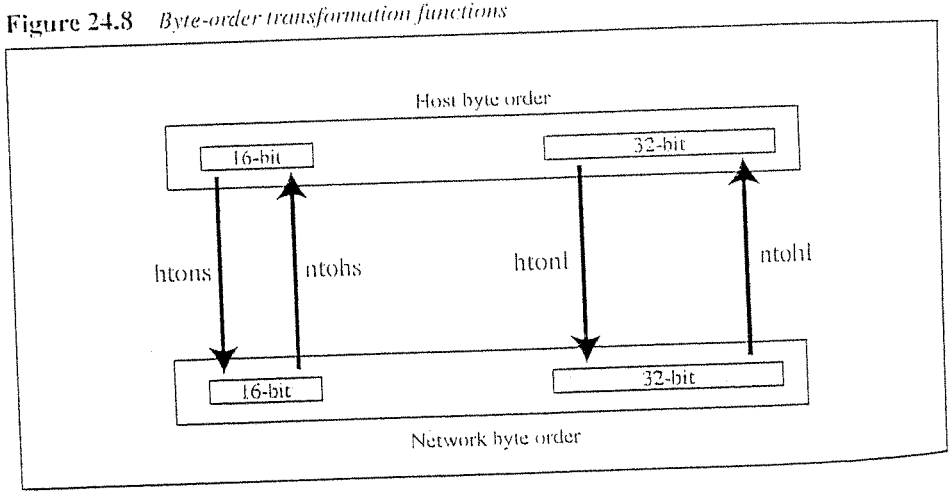
Networking protocols can also choose their own byte order. The TCP/IP protocol suite has chosen the big-endian byte order.

The byte order for the TCP/IP protocol suite is big endian.

Byte-Order Transformation

To create portability in application programs, TCP/IP software provides a set of functions that transforms integers from a host byte order (big endian or little endian) to network byte order (big endian). Four functions are designed for this purpose: htons, htonl, ntohs, and ntohl (see Figure 24.8).

Figure 24.8 *Byte-order transformation functions*



The prototypes are shown in Figure 24.9.

Figure 24.9 *Declarations for byte-order transformation*

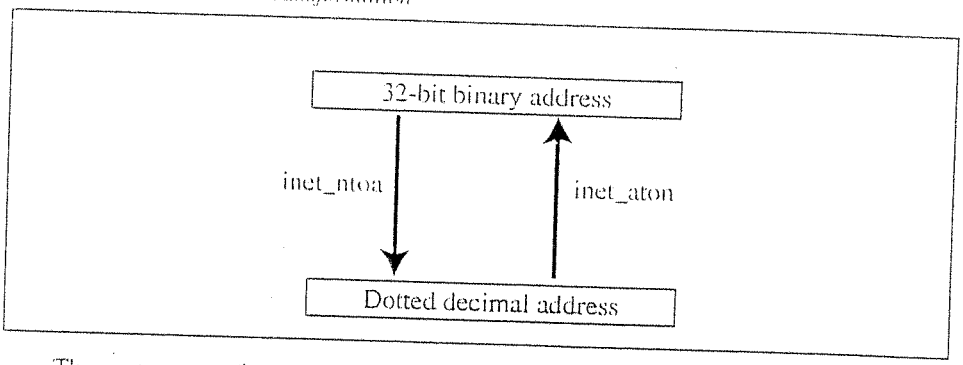
```
u_short  htons ( u_short  host_short ) ;

u_short  ntohs ( u_short  network_short ) ;

u_long   htonl ( u_long  host_long ) ;

u_long   ntohl ( u_long  network_long ) ;
```

- **htons.** The function htons (host to network short) converts a 16-bit integer from host byte order to network byte order.
- **htonl.** The function htonl (host to network long) converts a 32-bit integer from host byte order to network byte order.
- **ntohs.** The function ntohs (network to host short) converts a 16-bit integer from network byte order to host byte order.
- **ntohl.** The function ntohl (network to host long) converts a 32-bit integer from network byte order to host byte order.

24.4 ADDRESS TRANSFORMATION

Network software provides functions to transform an IP address from ASCII dotted decimal format to 32-bit binary format and vice versa. Two of these functions are discussed here: inet_aton and inet_ntoa (see Figure 24.10).

Figure 24.10 *Address transformation*



The prototypes of transformation functions are shown in Figure 24.11.

Figure 24.11 *Declarations for transformation functions*

```
int     inet_aton ( const char  *strptr , struct in_addr  *addrptr ) ;
char    *inet_ntoa (struct in_addr  inaddr ) ;
```

- **inet_aton.** This function transforms an ASCII string that contains up to four segments separated by dots to a 32-bit binary address in network byte order.
- **inet_ntoa.** This function transforms a 32-bit binary address in network byte order to an ASCII string with four segments separated by dots.

24.5 BYTE MANIPULATION FUNCTIONS

In network programming, we often need to initialize a field, copy the contents of one field to another, or compare the contents of two fields. We cannot use string functions such as *strcpy* or *strcmp* because these functions assume that a field is terminated with a null character, which is not true in network programming. As a matter of fact, we may need to copy a sequence of bytes from one field to another that may contain a zero byte. The string functions interpret this zero as a terminator and stop at that point.

Several functions have been defined in the <string.h> header file for these byte manipulation purposes. We introduce the three most common: *memset*, *memcpy*, and *memcmp*. Their prototypes are shown in Figure 24.12.

Figure 24.12 *Declaration for byte-manipulation functions*

```
void *memset ( void *dest , int chr , int len ) ;

void *memcpy ( void *dest , const void *src , int len ) ;

int  memcmp ( const void *first , const void *second , int len ) ;
```

- **Memset.** This function sets a specified number of bytes to a value. The first argument is a pointer to the destination, the field to be set. The second argument is the value, and the third argument is the number of bytes. One can use the sizeof operator to fill the entire field. For example, the following stores zeros in a field called *x*:

```
memset (&x , 0 , sizeof(x) ) ;
```

- **Memcpy.** This function copies the value of one field to another. The first argument is a pointer to the destination. The second argument is a pointer to the source. The third argument is the number of bytes to be copied. For example, the following copies the value of the *y* field to the *x* field:

```
memcpy (&x , &y , sizeof(x) ) ;
```

- **Memcmp.** This function compares two fields. The first argument is a pointer to the first field. The second argument is a pointer to the second field. The third argument is the number of bytes to be compared. This function returns zero if the two fields are the same. It returns a number less than zero if the first field is smaller than the second. It returns a number greater than zero if the first field is greater than the second. For example, the following compares the first 10 bytes of fields *x* and *y*:

```
memcmp ( &x , &y , 10 ) ;
```

24.6 INFORMATION ABOUT REMOTE HOST

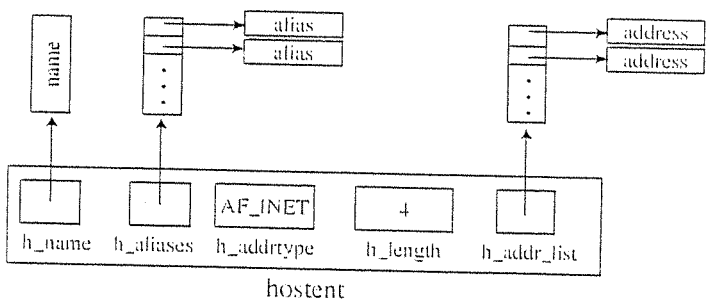
A process often needs information about a remote host. Several functions have been designed to provide this information. We discuss one such function, called *gethostbyname*. This function is actually a call to the DNS. The function accepts the domain name of the host and returns structured information called *hostent* that is actually the content of a resource record. The prototype of the function is given in Figure 24.13. The hostname is the domain name of the host in the form xxx.yyy.zzz. The function returns a pointer to the hostent structure.

Figure 24.13 *Declaration for gethostbyname*

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

The struct hostent provides several pieces of information. The first field is a pointer to the name of the host. The second field is a pointer to an array of pointers with each pointer pointing to an alias by which the host can be called. The third field is the type of address (AF_INET in the Internet). The next field is the length of the address (4 bytes for IPv4). The last field is a pointer to an array of pointers with each pointer pointing to one of the host addresses (the host can be a multihomed host). See Figure 24.14.

Figure 24.14 *The hostent structure*




```
struct    hostent
{
          char          *h_name ;
          char          **h_aliases ;
          int           h_addrtype ;
          int           h_length ;
          char          **h_addr_list ;
} ;
```

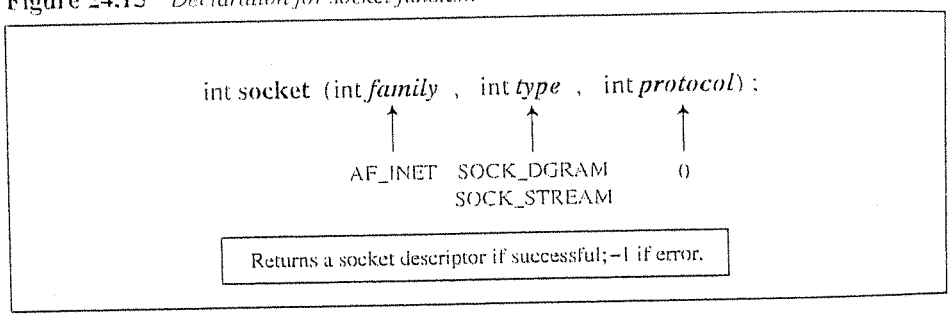
24.7 SOCKET SYSTEM CALLS

Several functions have been defined that can be called by an application program to communicate with another application program. We introduce some of these in this section for later use in our client-server programs.

Socket

The socket function is used by a process to create a socket. The prototype for this function is given in Figure 24.15. The family, type, and protocol fields were defined in Figure 24.4. For TCP/IP, the value of family is the constant AF_INET. The value of type as used in this chapter is either the constant SOCK_STREAM (used by stream sockets) or SOCK_DGRAM (used by datagram sockets).

Figure 24.15 *Declaration for socket function*



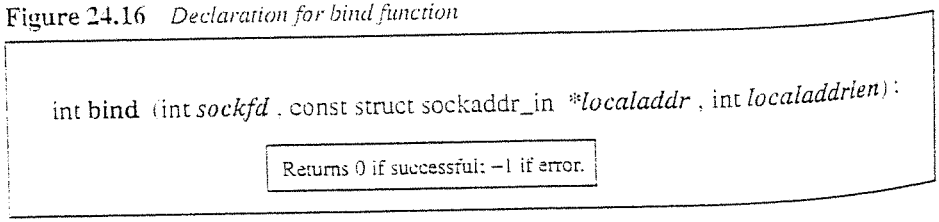
Although this function creates a socket, it sets values for only the first three fields (family, type, and protocol) of the socket structure. The other fields are set by the other functions or by the operating system, as we will discuss later.

The function returns an integer called the *socket descriptor*, which uniquely defines the created socket if the creation is successful. It returns −1 if there is an error. The socket descriptor is used by other functions to refer to the socket.

Bind

The bind function binds a socket to a local socket address by adding the local socket address to an already created socket. The prototype is given in Figure 24.16.

Figure 24.16 *Declaration for bind function*



Sockfd is the socket descriptor returned by the socket function; *localaddr* is a pointer to the socket address of the local machine; and *localaddrlen* is the length of the local socket address.

To use this function, the client needs first to call the socket function in order to use the returned value as the socket descriptor. The function sets values for the local socket address. Note that this function is not usually called by the client. In a client, the information about the local socket address is usually provided by the operating system. This function returns an integer, 0 for success and –1 for any error.

Connect

The connect function is used by a process (usually a client) to establish an active connection to a remote process (normally a server). The prototype is given in Figure 24.17.

Figure 24.17 *Declaration for connect function*

```
int connect (int sockfd , const struct sockaddr_in *serveraddr , int serveraddrlen) ;
```

Returns 0 if successful; –1 if error.

Sockfd is the socket descriptor returned by the socket function; *serveraddr* is a pointer to the remote socket address; and *serveraddrlen* is the length of that address.

To use this function, the client needs first to call the socket function in order to use the returned value as the socket descriptor. This function sets values for the remote socket address. The local socket address is either provided by the bind function or set by the operating system. The function returns an integer, 0 for success and –1 for any error.

Listen

The listen function is called only by the TCP server. It creates a passive socket from an unconnected socket. Before calling the listen function, the socket must already be created and the local socket address fields set. This function informs the operating system that the server is ready to accept connection through this socket. Figure 24.18 shows the prototype.

Figure 24.18 *Declaration for listen function*

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

Returns 0 if successful; –1 if error.

Sockfd is the socket descriptor returned by the socket function; *backlog* is the number of requests that can be queued for this connection. This function returns an integer, 0 for success and −1 for any error.

Accept

The accept function is called by a TCP server to remove the first connection request from the corresponding queue. If there are no requests (the queue is empty), the accept function is put to sleep. The prototype is given in Figure 24.19.

Figure 24.19 *Declaration for accept function*

```
int accept (int sockfd , const struct sockaddr_in *clientaddr , int *clientaddrlen) ;
```

Returns a socket descriptor if successful; −1 if error.

Sockfd is the socket descriptor; *clientaddr* is the pointer to the address of the client that has requested the connection; and *clientaddrlen* is a pointer to the client address length. Note that the socket address is not passed to the function but is returned from it. The length, however, is passed to the function as a value; it is also returned as a result.

This function actually creates a new socket (child socket) that can be used by a child server to connect to the client. All of the information needed for a new socket is provided by the operating system. The return value is the new socket descriptor.

Sendto

The sendto function is used by a process using UDP to send a message to another process usually running on a remote machine. The prototype is given in Figure 24.20.

Figure 24.20 *Declaration for sendto function*

```
int sendto (int sockfd , const void *buf , int buflen , int flags ,
            const struct sockaddr_in *toaddr , int toaddrlen) ;
```

Returns number of bytes sent if successful; −1 if error.

Sockfd is the socket descriptor; *buf* is a pointer to the buffer holding the message to be sent; *buflen* defines the length of the buffer; and the *flags* field specifies out-of-band data or lookahead messages. Normally it is set to zero. *Toaddr* is a pointer to the socket address of the receiver and *toaddrlen* is the length of the socket address. The function returns the number of characters sent if there is no error and −1 otherwise.

Recvfrom

The recvfrom function extracts the next message that arrives at a socket. It also extracts the sender's socket address. In this way, the process that uses this function can record the socket address and use it to send a reply back to the sender. It is used mostly by a UDP process. The prototype is given in Figure 24.21.

Figure 24.21 *Declaration for recvfrom function*

```
int recvfrom (int sockfd, const void *buf, int buflen, int flags
              const struct sockaddr_in *fromaddr, int *fromaddrlen);
```

Returns number of bytes received if successful; −1 if error.

Sockfd is the socket descriptor; *buf* is a pointer to the buffer where the message will be stored; *buflen* defines the length of the buffer; and the *flags* field specifies out-of-band data or lookahead messages. Normally it is set to zero. *Fromaddr* is a pointer to the socket address of the sender, and *fromaddrlen* is a pointer to the length of the socket address. The function returns the number of characters received if there is no error and −1 otherwise. Note that the socket address is not passed to the function, it is returned from it. It can be used by the process to respond to the remote process. The length, however, is passed as a value and is also returned as a result.

Read

The read function is used by a process to receive data from another process running on a remote machine. This function assumes that there is already an open connection between two machines; therefore, it can only be used by TCP processes. The prototype is given in Figure 24.22.

Figure 24.22 *Declaration for read function*

```
int read (int sockfd, const void *buf, int buflen);
```

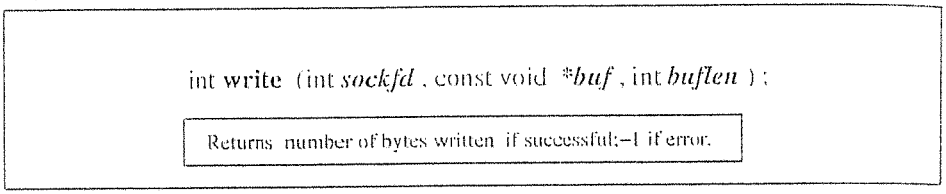
Returns number of bytes read if successful; 0 for end of file; −1 if error.

Sockfd is the socket descriptor; *buf* is a pointer to the buffer where data will be stored; and *buflen* is the length of the buffer. This function returns the number of bytes received (read) if successful, 0 if an end-of-file condition is detected, and −1 if there is an error.

Write

The write function is used by a process to send data to another process running on a remote machine. This function assumes that there is already an open connection between two machines. Therefore, it can only be used by TCP processes. The prototype is given in Figure 24.23.

Figure 24.23 *Declaration for write function*

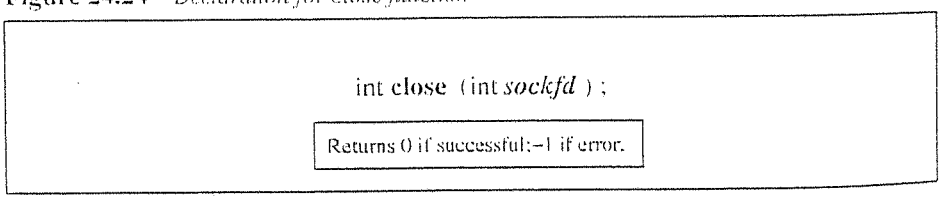


Sockfd is the socket descriptor; *buf* is a pointer to the buffer where data to be sent is stored; and *buflen* is the length the buffer. This function returns the number of bytes sent (written) if successful and –1 if there is an error.

Close

The close function is used by a process to close a socket and terminate a TCP connection. The prototype is given in Figure 24.24.

Figure 24.24 *Declaration for close function*



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

24.8 CONNECTIONLESS ITERATIVE SERVER

In this section, we discuss connectionless, iterative client-server communication using UDP and datagram sockets. As we discussed in Chapter 14, a server that uses UDP is usually connectionless iterative. This means that the server serves 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, which could all be from one client or from many clients, are stored in a queue, waiting for service. They are processed one by one in order of arrival.