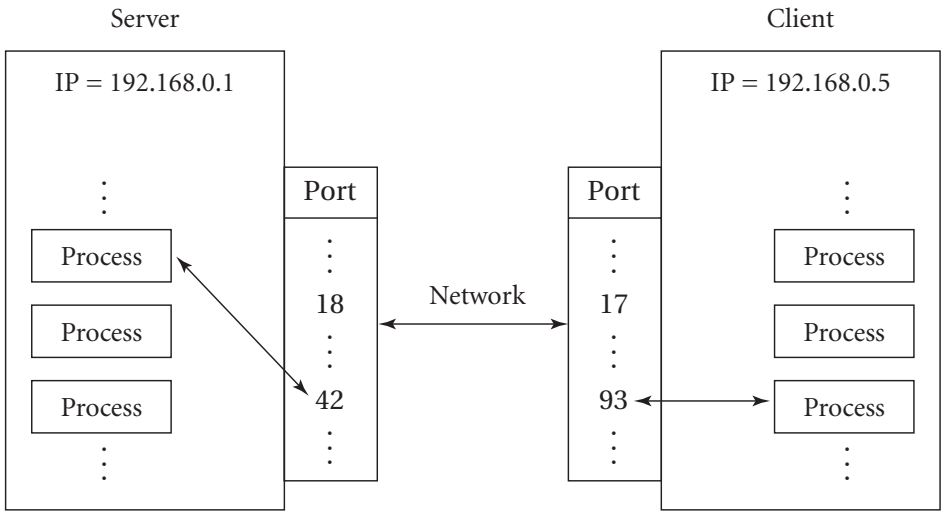

Figure 7.3 A server-client interaction.

7.5.2 Client-Server Model

Most network process communications follow a client-server model. In this model, a process on one computer acts as a server, and a process on the second computer acts as a client. The server opens up a port for listening and waits for a client to attempt to establish a connection. The client calls the server by connecting to the port on which the server is listening. When establishing a connection, a process must identify the IP and port to which it wishes to communicate. An analogy can be drawn to making a phone call to a house where multiple people live. The IP is like the phone number that identifies the house, while the port is like the name of the person to which the caller wishes to speak. Figure 7.3 shows a diagram of the idea. In this example, a server process is listening on its local port 42. The client process uses its local port 93 to attempt a connection. In the connection call, it specifies that it wishes to communicate with a process at 192.168.0.1:42, indicating both the server IP and port on which the server process is listening.

The following subsections explain the steps in creating, using, and closing a network communication from the perspectives of both a server and a client.

7.5.2.1 socket()

The first step in establishing a network communication is to create a socket. A socket provides an integer identifier through which a network communication is going to take place. The newly created socket is analogous to a telephone that has

not yet been used to place a call; the socket identifier has not yet been used to connect to anything. The following code demonstrates this first step:

```
#include <sys/types.h>  /* system type definitions */
#include <sys/socket.h> /* network system functions */
#include <netinet/in.h> /* protocol & struct definitions */

int sock;

sock=socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock < 0)
  printf("socket() failed\n");
```

The `socket()` system call has three arguments: the domain, the type of communication, and the protocol. Together these parameters describe how the socket will be used for communication. It is beyond the scope of this text to explain all possible values for these arguments; that would involve a detailed study of network protocols and layers. However, the values listed in this example provide the most common, stable connection for an IPv4 network communication.

7.5.2.2 bind()

The second step in establishing a network communication depends upon whether the process will act as a server or a client. A server will typically bind the socket, defining the IP and port on which it will listen for connections. For example:

```
int                     i,sock;
struct sockaddr_in      my_addr;
unsigned short          listen_port=60000;

      /* ... socket has been created ... */

      /* make local address structure */
memset(&my_addr, 0, sizeof (my_addr));  /* clr structure */
my_addr.sin_family = AF_INET;   /* address family */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  /* my IP */
my_addr.sin_port = htons(listen_port);

      /* bind socket to the local address */
i=bind(sock, (struct sockaddr *) &my_addr, sizeof (my_addr));
if (i < 0)
  printf("bind() failed\n");
```

The `struct sockaddr_in` holds information about the connection, including the IP and port number. It is then used in the `bind()` function to give the socket the equivalent of a telephone number. At a detailed level, the structure is first zeroed out (all bytes in the structure given a value of zero) using the `memset()` function. This is done to ensure that all unused portions of the structure have a value of zero. The structure is then filled with information about how the socket will be used. The `htonl()` function is one of a family of functions that makes sure bytes are in the correct order for network transport. In a bind() function call, the value of INADDR_ANY as an address indicates that the socket should be bound to the IP of the machine on which the process is currently executing. In this example, the server will look for communication on its own IP on port number 60000.

7.5.2.3 listen()

After binding, a server will typically call the `listen()` function to await communication:

```
int i,sock;

      /* ... socket has been created and bound ... */

      /* listen */
i=listen(sock,5);
if (i < 0)
  printf("listen() failed\n");
```

The second parameter of the listen() function describes how many connections can be queued while the server is handling another communication. In this example, the operating system will queue up to five connection attempts before returning an error value to additional clients trying to connect.

7.5.2.4 connect()

A client performs a step similar to binding, but instead of listening, it actively makes a call, establishing a connection. For example:

```
int                     i,sock;
struct sockaddr_in      addr_send;
char                    *server_ip="130.127.24.92";
unsigned short          server_port=60000;

      /* ... socket has been created ... */
```

```
      /* create socket address structure to connect to */
memset(&addr_send, 0, sizeof (addr_send)); /* clr structure */
addr_send.sin_family = AF_INET; /* address family */
addr_send.sin_addr.s_addr = inet_addr(server_ip);
addr_send.sin_port = htons(server_port);

      /* connect to the server */
i=connect(sock, (struct sockaddr *) &addr_send,
        sizeof (addr_send));
if (i < 0)
  printf("connect() failed\n");
```

In this example, the `struct sockaddr_in` is filled with information about the server to which the client wishes to connect. The `connect()` function is used to call the server in an attempt to establish a connection.

7.5.2.5 accept()

Once a server has received an incoming connect attempt, it can accept the connection. For example:

```
int                     i,sock,sock_recv,addr_size;
struct sockaddr_in      recv_addr;

      /* ... socket created, bound and listening ... */

      /* incoming xion -- get new socket to receive data on */
addr_size=sizeof(recv_addr);
sock_recv=accept(sock, (struct sockaddr *) &recv_addr,
        &addr_size);
```

The `accept()` function returns a second socket (in this example, `sock_recv`) on which data will be transmitted. This allows the original socket (in this example, `sock`) to continue to listen for additional connections.

7.5.2.6 send() and recv()

After a connection has been established between the client and server, data can be transmitted and received. For example, the client could execute the following code:

```
int sock,bytes_sent;
char text[80];

    /* ... socket has been created and connected ... */
```

```
    /* send some data */
printf("Send? ");
scanf("%s",text);
bytes_sent=send(sock,text,strlen(text),0);
```

The `send()` function call takes four arguments: the socket identifier, an address pointing to data, the number of bytes to send, and a flags setting. Normally the flags value is set to zero. The server executes similar code, receiving the sent data. For example:

```
int sock_recv,bytes_received;
char text[80];

    /* ... socket created, bound and accepted ... */

    /* receive some data */
bytes_received=recv(sock_recv,text,80,0);
text[bytes_received]=0;
printf("Received: %s\n",text);
```

The parameters for the `recv()` function call are similar to those for `send()`, except that the third argument (in this example, 80) indicates the maximum number of bytes that can be received. Both the server and client can execute send() and recv(); the communication can go both ways. Also note that the data does not have to be text. The send() and recv() functions are similar to the fread() and fwrite() functions in that the arguments define an address and a number of bytes, rather than the type of data at the given address.

7.5.2.7 close()

Once communication is finished, both the server and client should close their respective sockets. For example:

```
int i,sock_recv;

      /* ... socket communication is finished ... */
i=close(sock_recv);
if (i < 0)
  printf("close() failed\n");
```

A `close()` system call initiates a series of operations within the O/S to terminate the connection. Thus, the socket may still appear in a `netstat` listing for several seconds, until the O/S has finished the close.

7.5.3 Examples

In the previous section, an analogy was made between initiating a network communication and placing a telephone call. Unlike telephony, however, in which communications are mostly one person to one person, in the case of network programming, there are a variety of situations besides one-to-one network process communications. The caller may want to broadcast an announcement, calling a whole subnetwork of computers at the same time. The caller may open a socket and use it to communicate with several remote processes at the same time. The server may take only one call at a time, or open itself up to multiple simultaneous calls. The calls may be managed all through the same port, or the server might take multiple incoming calls on one port and then manage the communication for each on a different transient port. These are only some of the situations for which socket system calls can be used. The following two examples demonstrate two specific situations and detail a client and server program for each.

7.5.3.1 Single server-client connection

The following programs put together all the ideas from Section 7.5.2 into a server program and a client program. The server listens on port 60000 for a client. When a client connects, the server reads any incoming data as text and prints it out one line at a time. If the server receives the string "shutdown," then it closes the socket and exits.

The following code is for the server:

```
#include <stdio.h>
#include <sys/types.h>  /* system type definitions */
#include <sys/socket.h> /* network system functions */
#include <netinet/in.h> /* protocol & struct definitions */

#define BUF_SIZE        1024
#define LISTEN_PORT     60000

int main(int argc, char *argv[])

{
int                     sock_listen,sock_recv;
struct sockaddr_in      my_addr,recv_addr;
int                     i,addr_size,bytes_received;
fd_set                  readfds;
struct timeval          timeout={0,0};
```