An Introduction to Sockets in C
Through Annotated Examples

by
Peter M. Broadwell
Grinnell College, Grinnell, IA

With minor adjustments for Linux compatibility
made November 29, 2000 by Henry M. Walker

An Overview of Sockets in C:

- A socket is a type of application programming interface. As such, a socket is a software device that allows processes to communicate with one another over the Internet, within local area networks, or even on single computers. Sockets provide the programmer with a flexible communications interface that allows the reliable transfer of large amounts of data between processes. It is because sockets possess these important capabilities that they are in wide use as software communications tools today.
- The aim of this pamphlet is to provide an introduction to the use of sockets in the C programming language. Because programmers currently use sockets in a wide variety of capacities, several volumes would be necessary to catalog and explain all of the capabilities of sockets. This pamphlet seeks to introduce the basic properties and concepts behind the most commonly-used varieties of sockets. It does this by presenting a series of example programs, each highlighting an application of sockets, and then providing detailed annotations on the manner in which these programs function.
- Because the primary purpose of sockets is to promote data transfer, the introductory sample programs all deal with the computing scenario known as the readers-writers problem. In this problem, one or more writer processes write data to a central buffer in such a way that one or more reader processes may then access this buffer and retrieve the writers' data from it. As the sample programs will demonstrate, a socket functions as an ideal read-write buffer. As the sample programs will also show, the varied capabilities and communications protocols supported by sockets naturally lead to certain insightful alterations of the readers-writers problem.
- In socket terminology, one of the communicating processes is called the **server** and the other the **client**. The server has greater control over the communications process because it is the process that initially creates the socket. Because of this, numerous clients may communicate through the same socket, but only one server may be associated with a particular socket.
- Whle examining the sample problems to come, one must make a couple of distinctions regarding the properties of sockets. One such distinction is that of the socket **family**. In general, the family to which a socket belongs determines the hardware environments in which it is used. Of the two families in the upcoming examples, sockets of the family `AF_UNIX` exist as files in a directory of a local host computer, and only provide data transfer between processes on that machine. For this reason, they are referred to as UNIX domain sockets.
- Sockets of the second family presented in this text, `AF_INET`, are less efficient than UNIX domain sockets, but allow communication between processes on remote machines that are connected by a local network or by the Internet. As such, they are called Internet sockets.
- The other distinction one must make inolves the socket's **type**. The type of a socket describes the communications protocol it uses to send data. The first socket type to be used in the examples is the `SOCK_STREAM` (streaming socket) type. Processes that communicate through sockets of this type must connect explicity to a socket, and the data transfer from the client to the server (or vice versa) takes place through a reliable byte stream. This type of data transfer is called TCP (Transmission Control Protocol).
- The second type of socket is a `SOCK_DGRAM` (datagram) socket. These sockets allow data to be transferred in finite packages, called datagrams. Because of this, processes do not have to be connected explicitly to the socket in order to send and receive data through it. As later examples will demonstrate, the simplicity of this type of data transfer (called UDP, for User Datagram Protocol) carries with it a degree of unreliability.

- The following examples introduce these varied concepts in a sequential, logical progression, from basic applications to more complicated procedures. Streaming UNIX domain sockets are presented in the first examples (`streaming1.c` and `streaming2.c`) because their declaration proves insightful and because communicating through them is fairly straightforward, . The next two programs, `netserv.c` and `netclient.c`, form a single example. In this scenario, the instructions for the client and server processes have been broken up into two separate programs in order to illustrate the capability of Internet sockets to allow processes on separate machines to communicate with one another.
- The second group of examples involves datagram sockets. Unlike streaming sockets, which can belong to either the UNIX domain or Internet family, sockets of the type `SOCK_DGRAM` must always be Internet sockets. `datagram1.c` and `datagram3.c` illustrate, respectively, simple and complex applications of this protocol. `datagram2` uses a script file to execute one server process locally and several client processes remotely to demonstrate an application of sockets to parallel computing.

Program 1: streaming1.c

This first example program demonstrates the connection and data transfer protocols associated with streaming sockets.

```
/* Author: Pete Broadwell, Grinnell College
   Minor Adjustments for Linux made 11/29/00 by Henry M. Walker

   This program facilitates a simple client/server data transfer through the
   use of UNIX domain sockets. Simulating a simple scenario of the readers-
   writers problem, the writer (server) process writes 10 integers to the
   socket, which the reader (client) process then reads. */

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>     /* socket command libraries needed by some compilers */
#include <sys/socket.h>
#include <sys/un.h>

#define SOCKET_NAME "Com1" /* name of the socket file */
#define SIM_LENGTH 5 /* number of integers to be written and read */

/* procedure to remove socket file and exit */
void clean_up(int cond, int *sock)
{ printf("Exiting now.\n");
  close(*sock); /* close the socket file */
  unlink(SOCKET_NAME); /* remove the socket file */
  exit(cond);
} /* end of clean_up */

int main(void)
{ pid_t pid;
  pid = fork();     /* spawn new child process */
  if (-1 == pid)
    { perror("Error in fork");
      exit(1);
    }

  if (0 == pid)
    { /* processing for client */
      int sock; /* socket file descriptor */
      struct sockaddr_un cli_name; /* socket address structure */
      int count;
      int value; /* variable for number read from socket */

      sleep(1); /* give server process time to create the socket */
      printf("Client is alive and establishing socket connection.\n");

      /* set the socket descriptor */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0)
        { perror("Error opening channel");
          clean_up(1, &sock);
        }
```

```
    /* set the phyiscal address (cli_name) of the socket descriptor */
    bzero(&cli_name, sizeof(cli_name)); /* initializes cli_name structure */
    cli_name.sun_family = AF_UNIX; /* socket family will be AF_UNIX (UNIX) */
    strcpy(cli_name.sun_path, SOCKET_NAME); /* set the address of the
                                               socket within the file system:
                                               in this case, "Com1" */
    /* connect to the socket */
    if (connect(sock, (struct sockaddr *)&cli_name, sizeof(cli_name)) < 0)
      { perror("Error establishing communications");
        clean_up(1, &sock);
      }

    /* read data from the socket */
    for (count = 1; count <= SIM_LENGTH; count++)
      { read(sock, &value, 4);
        printf("Client has received %d from socket.\n", value);
      }
    close(sock); /* close connection to the socket */
    exit(0);

  } /* end of processing for client */

else
  { /* processing for server */
    int sock; /* variable for listening socket descriptor */
    int connect_sock; /* variable for connected socket descriptor */
    struct sockaddr_un serv_name;
    int count;
    size_t len; /* variable to store the size of the serv_name structure */

    /* set socket descriptor */
    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0)
      { perror("Error opening channel");
        clean_up(1, &sock);
      }

    /* set the physical address (serv_name) of the socket descriptor */
    bzero(&serv_name, sizeof(serv_name));
    serv_name.sun_family = AF_UNIX;
    strcpy(serv_name.sun_path, SOCKET_NAME);

    /* bind the socket address to the socket descriptor */
    if (bind(sock, (struct sockaddr *)&serv_name, sizeof(serv_name)) < 0)
      { perror("Error naming channel");
        clean_up(1, &sock);
      }

    listen(sock, 1); /* listen for connections on the socket */
    printf("Server is alive and waiting for socket connection from client.\n");

    /* accept connection request from client */
    len = sizeof(serv_name);
    connect_sock = accept(sock, (struct sockaddr *)&serv_name, &len);
```

```
    /* write data to the client's connected socket */
    for (count = 1; count <= SIM_LENGTH; count++)
      { write(connect_sock, &count, 4);
        printf("Server has written %d to socket.\n", count);
      }

    waitpid(pid, NULL, 0); /* Wait for client process to terminate */
    clean_up(0, &sock); /* Exit with no errors */
  } /* end of processing for server */

} /* end of main */
```

Annotations for `streaming1.c`*:*

- In this example, the `fork` function creates the client process. Both the client (child) and server (parent) processes set up the socket file and address descriptors they will need to communicate through a streaming UNIX domain socket. While these structures could be defined globally, in practice the client and server processes are usually completely separate programs. Thus, each must set up the appropriate communications structures on its own.
- The client and the server communicate with each other through a UNIX domain socket in much the same way that two processes may communicate with each other through the use of the `pipe` function. As in all socket interactions, the server creates the communication socket. Following the streaming socket protocol, the client then connects to this socket. Once the server has accepted this connection, the two processes can send messages to each other through the socket.
- Sockets of the family used in this program (AF_UNIX), exist as files in a user-defined path of the file system. With the line
  `#define SOCKET-NAME "Com1"`
  the constant `SOCKET_NAME` is set as the name of the file `Com1`. Once the server process creates the socket, it exists in the local directory as this file. At the end of the program, this file is deleted. If one were to use the `ls` command to check the contents of the directory containing `read-write1` during its execution, a file with the name `Com1` would be visible. By typing `netstat` at the command prompt, one can also see under the section heading "Active UNIX domain sockets" all of the AF_UNIX sockets running on the current host machine (including `Com1`, if `read-write1` is currently executing.)
- Examining the `main` function, one sees that the first action of both the server and client processes is to define the socket file descriptor and address structure. The `socket` function creates a file descriptor (named `sock`), of the type `SOCK_STREAM`. Both the client and the server use this file descriptor to refer to the socket in the following functions.
- The second half of the socket initialization procedure involves setting the socket's physical address. This is stored in the `cli_name` and `serv_name` data structures for the client and server, respectively. The `bzero` function initializes this data structure (of the type `sockaddr_un`, for UNIX socket address) to all 0's. The next two functions set the family of the socket (`AF_UNIX`, in this case) and the address of the socket in the file system (`Com1` in the local directory).
- The client's next action is crucial. It invokes the `sleep` function in order to ensure that the server has created the socket before it attempts to connect to it. In most applications, the server process is running all of the time, so this is not necessary. Here, though, it's important, so that that the client process does not receive an error when it attempts to connect to the server.

- While the client sleeps, the server creates the socket. Up to this point, the socket has existed only as data structures maintained by the client and server. With the `bind` function, the server binds the file descriptor, `sock`, for the socket to the defined socket address structure, `serv_name`. Essentially, this call requires the kernel to assign an address to the socket described by `sock` and `serv_name`. Once this has happened, both the server and the client can access the socket, since they both know its file descriptor and address.
- Once the server has created the socket, the client may connect to it. With streaming sockets, this connection is accomplished through a *three-way handshake*. This handshake begins when the server invokes the `listen` function to detect when the client attempts to connect to `sock`. The integer parameter of this function indicates the number of connections the server is expecting (in this case, only 1). When it calls the `listen` function, the server then sleeps until it detects such a connection. With the `connect` function, the client attempts to connect to the socket. At this point, the server wakes up and invokes the the `accept` function. As its name implies, this function accepts the client's connection requests, completing the handshake.
- For the client, the following communication takes place through the usual `sock` file descriptor. For the server, though, the `listen` function has changed `sock` into a *listening socket*. The `accept` function creates a new socket file descriptor, `connect_sock`. For the server, this is a *connected socket*; that is, a socket through which the server may communicate with a client. Thus, the server completes the following communications through this descriptor.
- Transferring data to and from sockets of this type (SOCK_STREAM) is accomplished through the use of simple `read` and `write` functions. The server uses the `read` function to send integers to the socket, and the client uses the `read` functions to receive these integers from the socket when they arrive.
- After the processes have communicated with one another, the client closes its socket descriptor and exits. After the client terminates, the server closes its connected socket descriptor and then calls the `clean_up` function. This function closes the server's `sock` file descriptor and then removes the actual socket file `(Com1)` from the local directory before exiting.
- A sample run of `streaming1.c` Note that like `listen`, `connect`, and `accept`, `read` is a blocking system call. Unless an external timeout value is declared, the client may wait indefinitely for data to arrive at the socket if the programmer does not coordinate the data transfer between the processes correctly.

Sample Run of `streaming1.c`*:*

```
leibniz% streaming1
Server is alive and waiting for socket connection from client.
Client is alive and establishing socket connection.
Server has written 1 to socket.
Server has written 2 to socket.
Server has written 3 to socket.
Server has written 4 to socket.
Server has written 5 to socket.
Client has received 1 from socket.
Client has received 2 from socket.
Client has received 3 from socket.
Client has received 4 from socket.
Client has received 5 from socket.
Exiting now.
leibniz%
```

Program 2: streaming2.c

This is a simulation of the next level of complexity of the readers-writers problem. In this program, multiple readers (clients) communicate with the writer (server). This program illustrates an important property of server using streaming sockets: the explicit connection that is required between the server and the client makes it possible for the server to fork child processes to handle each of the connection requests recursively.

```
/* Author: Pete Broadwell, Grinnell College
   Minor Adjustments for Linux made 11/29/00 by Henry M. Walker

   This program spawns multiple server child processes to handle socket
   connection requests from multiple clients. In this way, the parent
   server process handles the client connection requests recursively.
   This program represents an alteration of the normal readers-writers
   problem, in that the parent server effectively assigns one writer
   to each reader, and the two communicate through a proprietary connected
   streaming socket.

*/

#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>   /* socket command libraries needed by some compilers */
#include <sys/socket.h>
#include <sys/un.h>

#define SOCKET_NAME "Com2" /* name of the socket file */
#define NUM_READERS 3 /* number of reader (client) processes to be spawned */
#define MAX_READ 10.0 /* maximum number of integers that clients will read */

/* procedure to remove socket file and exit */
void clean_up(int cond, int *sock)
{ printf("Exiting now.\n");
  close(*sock); /* close listening socket descriptor */
  unlink(SOCKET_NAME); /* remove the socket file */
  exit(cond);
} /* end of clean_up */

int main(void)
{ pid_t pid;
  int p_count;
  pid_t proc[2*NUM_READERS];

  /* spawn reader (client) processes */
  for (p_count = 1; p_count <= NUM_READERS; p_count++)
    { pid = fork();
      if (-1 == pid)
        { perror("Error in fork");
          exit (1);
        }
```

```
    if (0 == pid)
      { /* processing for client == reader */
        int sock; /* variable for listening socket descriptor */
        struct sockaddr_un cli_name; /* socket address structure */
        int count;
        int ints_read; /* variable to store the number of integers each
                          client will read */
        int value; /* variable for number read from socket */
        time_t seed; /* variable to initialize random number generator */

        time(&seed);           /* initialize the random number generator */
        srand(seed + p_count); /* with the current system time */

        sleep(1); /* give server time to create the socket */
        printf("Client/reader process %d active.\n", p_count);

        /* set the socket descriptor */
        sock = socket(AF_UNIX, SOCK_STREAM, 0);
        if (sock < 0)
          { perror("Error opening channel");
            clean_up(1, &sock);
          }

        /* set the physical address (cli_name) of the socket descriptor */
        bzero(&cli_name, sizeof(cli_name));
        cli_name.sun_family = AF_UNIX;
        strcpy(cli_name.sun_path, SOCKET_NAME);

        /* connect to the socket */
        if (connect(sock, (struct sockaddr *)&cli_name, sizeof(cli_name)) < 0)
          { perror("Error establishing communications");
            clean_up(1, &sock);
          }

        /* randomly generate the number of integers the client will read */
        ints_read = 1 + (int) (MAX_READ * rand() / (RAND_MAX + 1.0));

        printf("Client %d wants to read %d integers.\n", p_count, ints_read);
        write(sock, &ints_read, 4);

        /* read values from socket */
        for (count = 1; count <= ints_read; count++)
          { read(sock, &value, 4);
            printf("Client %d has received %d from socket.\n", p_count, value);
          }
        printf("Client %d done.\n", p_count);
        close(sock); /* close connection to the socket */
        exit(0);
      }
    else proc[p_count - 1] = pid;
  } /* end of processing for client */

{ /* processing for server */
  int sock; /* listening socket descriptor */
  int connect_sock;  /* connected socket descriptor */
  struct sockaddr_un serv_name;
  int count, i, data, write_length;
```

```
size_t len; /* variable to store the length of the serv__name structure */

/* set socket descriptor */
sock = socket(AF_UNIX, SOCK_STREAM, 0);
if (sock < 0)
  { perror("Error opening channel");
    clean_up(1, &sock);
  }

/* set the physical address (serv_name) of the socket descriptor */
bzero(&serv_name, sizeof(serv_name));
serv_name.sun_family = AF_UNIX;
strcpy(serv_name.sun_path, SOCKET_NAME);

/* bind the socket address to the socket descriptor */
if (bind(sock, (struct sockaddr *)&serv_name, sizeof(serv_name)) < 0)
  { perror("Error naming channel");
    clean_up(1, &sock);
  }

listen(sock, NUM_READERS); /* listen for connections on the socket */
printf("Server/writer is alive and waiting for socket connections.\n");

/* accept connection requests from clients */
for (i = 1; i <= NUM_READERS; i++)
  { connect_sock = accept(sock, (struct sockaddr *)&serv_name, &len);

    /* spawn a child of the server process to handle each separate
       connection request from the clients */
    pid = fork();

    if (-1 == pid)
    { perror("Error in fork");
      clean_up(1, &sock);
    }

    if (0 == pid)
      { /* processing for child of server */
        printf("Server child %d active.\n", i);
        close(sock); /* close the listening socket for this process */
        /* read the number of integers the client wants to read */
        read(connect_sock, &write_length, 4);
        printf("Server child %d now writing %d integers to socket.\n",
               i, write_length);

        /* write the requested number of integers in ascending order to the
           reader's connected socket descriptor. */
        for (count = 1; count <= write_length; count++)
          { data = count + ((i - 1) * 10);
            write(connect_sock, &data, 4);
            printf("Server child %d has written %d to socket.\n", i, data);
          }
        close(connect_sock); /* close the connected socket for the reader */
        exit(0);
      } /* end of processing for child of server */
    else proc[NUM_READERS + p_count - 1] = pid;
  }
```

```
    for (i = 0; i < (2 * NUM_READERS); i++) {
      waitpid(proc[i], NULL, 0); /* Wait for all child processes to terminate */
    }
    printf ("server cleaning up\n");
    clean_up(0, &sock); /* Exit with no errors */
  } /* end of processing for server */

} /* end of main */
```

Annotations for `streaming2.c`*:*

- This program introduces an important concept of streaming socket servers, but little new syntax. Each process (the server and all of the clients) once again sets up a socket file descriptor `sock` and its corresponding address structure, `serv_name` or `cli_name`. The server then creates the actual socket file, `"Com2"` this time, with the `bind` function.
- Each client is a child process created with the `fork` function. The extra code at the beginning of the client section allows the client determine randomly the number of integers it wants to read from the server. This is accomlished by seeding the random number generator with the current system time, and then generating a random integer between 1 and 10. The communications setup for each client is the same as in the previous example. In terms of actual data transfer, the only difference is that the client first sends to the server the number of integers it expects to read (not an uncommon situation in client-server interactions) and then reads this number of integers from the socket.
- The major conceptual difference in this program involves the server operation. If a server utilizes streaming sockets, it is necessary for the server to connect explicitly (through the `listen` and `accept` commands) to each of its clients. The most common and most efficient way of doing this is for the server to fork a child process to handle each connection request. In this manner, each server child `accept`s a connection request from a client and writes the requested data to it.
- Note that each server child process, after creating a connected socket descriptor with `accept`, immediately closes `sock`. This is because the `listen` command has changed `sock` into a listening socket for the server, and only the parent server process needs to listen for more connections. Other than this, the data transfer for the server child process is relatively straightforward.
- One may also note that if in some other situation it is important for the server to deal with the clients on an individual basis, it may communicate with them iteratively (one at a time). One may accomplish this by removing the `fork` functions from the server's `for` loop and establishing an array of connected socket descriptors, one for each client (for example, `connect_sock[0]`, `connect_sock[1]`, etc.).
- This scenario is something of a departure from the standard readers-writers problem. In this case, what first appears to be a single writer quickly becomes multiple writers, and each writer communicates explicitly with one reader. This makes the simulation a somewhat more conversation-oriented scenario of the readers-writers problem than usual. The upcoming examples of datagram socket communications will produce a multiple readers-multiple writers situation that is closer to the traditional setup.
- A sample run of `streaming2.c`. Note that due to the nondeterministic nature of process execution order in UNIX, a different output order will likely result from each run of the program. Also, because of similar issues involving the display buffer, it is common for the output to report that a client has read data before the server has sent it. This is not actually what has happened, of course, but it does illustrate the unpredictable nature of output order when multiple process interact with one another.

Sample Run of `streaming2.c`*:*

```
leibniz% streaming2
Server/writer is alive and waiting for socket connections.
Client/reader process 2 active.
Client/reader process 1 active.
Server child 1 active.
Client 1 wants to read 7 integers.
Server child 1 now writing 7 integers to socket.
Server child 1 has written 1 to socket.
Server child 1 has written 2 to socket.
Server child 1 has written 3 to socket.
Server child 1 has written 4 to socket.
Server child 1 has written 5 to socket.
Server child 1 has written 6 to socket.
Server child 1 has written 7 to socket.
Client 1 has received 1 from socket.
Client 1 has received 2 from socket.
Client 1 has received 3 from socket.
Client 1 has received 4 from socket.
Client 1 has received 5 from socket.
Client 1 has received 6 from socket.
Client 1 has received 7 from socket.
Client 1 done.
Client/reader process 3 active.
Client 3 wants to read 7 integers.
Client 2 wants to read 2 integers.
Server child 2 active.
Server child 2 now writing 7 integers to socket.
Client 3 has received 11 from socket.
Server child 2 has written 11 to socket.
Client 3 has received 12 from socket.
Server child 2 has written 12 to socket.
Client 3 has received 13 from socket.
Server child 2 has written 13 to socket.
Client 3 has received 14 from socket.
Server child 2 has written 14 to socket.
Client 3 has received 15 from socket.
Server child 2 has written 15 to socket.
Client 3 has received 16 from socket.
Server child 2 has written 16 to socket.
Client 3 has received 17 from socket.
Client 3 done.
Server child 2 has written 17 to socket.
Server child 3 active.
Server child 3 now writing 2 integers to socket.
Client 2 has received 21 from socket.
Server child 3 has written 21 to socket.
Client 2 has received 22 from socket.
Client 2 done.
Server child 3 has written 22 to socket.
Exiting now.
leibniz%
```

Program 3: netserv.c

This is the first (server) program in a set of two separate programs that can be run on different machines to demonstrate the capabilities of streaming Internet (AF_INET) sockets.

```
/* Author: Pete Broadwell, Grinnell College

   This program uses streaming Internet sockets to simulate the readers-writers
   problem. This is the server/writer portion of the simulation. */

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>     /* socket command libraries needed by some compilers */
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>

#define SIM_LENGTH 10 /* number of integers to be written and read */
#define PORT 1227 /* local port on which connection will be established */

/* procedure to remove socket and exit */
void clean_up(int cond, int *sock)
{ printf("Exiting now.\n");
  close(*sock); /* close socket connection */
  exit(cond);
} /* end of clean_up */

int main(void)
{ /* processing for server */
  int sock; /* listening socket descriptor */
  int connect_sock; /* connected socket descriptor */
  struct sockaddr_in serv_name; /* socket address structure */
  size_t len; /* variable for the size of the socket address structure */
  int count;

  /* set socket descriptor */
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    { perror ("Error opening channel");
      clean_up(1, &sock);
    }

  /* set name as the physical address of the socket descriptor (sock) */
  bzero(&serv_name, sizeof(serv_name)); /* initialize the address structure */
  serv_name.sin_family = AF_INET; /* socket family will be AF_INET
                                     (Internet sockets) */
  serv_name.sin_port = htons(PORT); /* sets the port on which the server will
       establish the socket connection */

  /* bind the socket address to the socket descriptor */
  if (bind(sock, (struct sockaddr *)&serv_name, sizeof(serv_name)) < 0)
    { perror ("Error naming channel");
      clean_up(1, &sock);
    }
```

```
  printf("Server is alive and waiting for socket connection from client.\n");
  listen(sock, 1); /* listen for connections on the socket */

  /* accept connection request from client */
  len = sizeof(serv_name);
  connect_sock = accept(sock, (struct sockaddr *)&serv_name, &len);

  /* send integers to the client's connected socket descriptor */
  for (count = 1; count <= SIM_LENGTH; count++)
    { write(connect_sock, &count, 4);
      printf("Server has written %d to socket.\n", count);
    }

  close(connect_sock); /* close connected socket */
  clean_up(0, &sock); /* exit with no errors */

} /* end of main */
```

Annotations for `netserv.c`*:*

- The family of sockets that we have studied until now, AF_UNIX, is used to provide high-speed full-duplex communication between processes on a single host computer. However, these sockets are not designed to function as API's when the client and server are on separate hosts. Internet sockets (sockets in the family AF_INET) are used to facilitate this type of communication. While slower than UNIX domain sockets, AF_INET sockets are used extensively as the communications protocol of the Internet.
- As can be observed from this program, much of the communications syntax for Internet sockets is similar to that of UNIX domain sockets. A few differences exist, though. Note that `serv_addr` is declared as a `sockaddr_in` instead of a `sockaddr_un` data structure. Also note the change in the socket family setting (`AF_INET` instead of `AF_UNIX`) in the definition of the `sock` descriptor.
- The most important difference to note is within the setup of the `serv_name` data structure. The first two lines are fairly self-explanatory. Understanding the final line, however, requires some background knowledge of Internet communications protocols.
- In order for a process to communicate with a process on another machine, it must know the Internet Protocol (IP) address of the host machine (such as 132.161.33.155). In addition, both processes must reocrd the numeric port number on which the process that will act as the server expects connections. The final line of the definition of `serv_name` establishes this. Port addresses below 1024 are reserved for superuser activites, so any value from 1024 to 49151 is acceptible. In most applications, an Internet server will have a *well-known port*, a default port value to which client processes try to connect first.
- Also note that `netserv.c` does not deal at all with the server's IP address, since it is the server and must only wait for the remote client to contact it.
- Once `serv_name` is declared, the rest of the program runs identically to the earlier examples using `AF_UNIX` sockets.

Sample Run of `netserv.c`*:*

```
markov% netserv
Server is alive and waiting for socket connection from client.
Server has written 1 to socket.
Server has written 2 to socket.
Server has written 3 to socket.
Server has written 4 to socket.
Server has written 5 to socket.
Server has written 6 to socket.
Server has written 7 to socket.
Server has written 8 to socket.
Server has written 9 to socket.
Server has written 10 to socket.
Exiting now.
markov%
```

Program 4: netclient.c

This is the client portion of the internet socket demonstration.

```
/* Author: Pete Broadwell
           Grinnell College

   This program uses streaming Internet sockets to simulate the readers-writers
   problem. This is the client/reader portion of the simulation.
*/

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>     /* socket command libraries needed by some compilers */
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>

#define SIM_LENGTH 10 /* number of integers to be written and read */
#define IP_ADDRESS "132.161.33.175" /* Internet address of server */
#define PORT 1227 /* local port on which the server establishes connection */

int main(void)
{ /* processing for client */
  int sock; /* socket file descriptor */
  struct sockaddr_in cli_name; /* socket address structure */
  int count;
  int value; /* variable for number read from socket */

  printf("Client is alive and establishing socket connection.\n");

  /* set the socket descriptor */
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    { perror ("Error opening channel");
      close(sock);
      exit(1);
    }

  /* set the physical address (cli_name) of the socket descriptor */
  bzero(&cli_name, sizeof(cli_name)); /* initializes the address structure */
  cli_name.sin_family = AF_INET; /* socket family will be AF_INET (Internet) */
  cli_name.sin_addr.s_addr = inet_addr(IP_ADDRESS); /* sets the Internet
       Protocol (IP) address
       of the server */
  cli_name.sin_port = htons(PORT); /* sets the port on which the server will
      establish the socket connection */

  /* connect to the socket */
  if (connect(sock, (struct sockaddr *)&cli_name, sizeof(cli_name)) < 0)
    { perror ("Error establishing communications");
      close(sock);
      exit(1);
    }
```

```
  /* read integers from the socket */
  for (count = 1; count <= SIM_LENGTH; count++)
    { read(sock, &value, 4);
      printf("Client has received %d from socket.\n", value);
    }

  printf("Exiting now.\n");

  close(sock); /* close connection to socket */
  exit(0); /* exit with no errors */

} /* end of main */
```

Annotations for `netclient.c`*:*

- This program also functions quite similarly to its UNIX domain socket counterpart. Once again, the only major difference is within the definition of the socket address structure (which is `cli_name`, this time).
- The first two lines of the definition of `cli_name` are again self-explanatory. The third line is the line at which the client records the IP address of the intended server (the dotted decimal string). The client MUST know this address ahead of time. On the HP network, the easiest method of finding the address of the intended host machine is to use the `traceroute` command. Simply type `traceroute (name of host computer)` at a command prompt. This displays the network name of the host with its IP address in parentheses. The final line of `cli_name`'s declaration sets the port address of the remote host at which it will connect to the server process. This must also be idential to the setting for the server.
- Note that `netserv.c` is not being run on the same host machine as `netclient.c`. Due to the nature of Internet sockets, the two processes could be run on the same machine (in which case it would not be necessary for the client to define the server's IP address), two different machines on the same local area network (as in this example) or even on two machines on opposite sides of the world, assuming they are both connected to the Internet.
- By now, one has undoubtedly noticed the large amount of syntax that some socket functions require. In particular, nearly every socket function requires that it be given the size of the socket address structure. `accept`, for example, requires that it be passed a pointer to the size of the socket address structure, and thus the size variable `len` must be declared ahead of time. In many cases, it is sufficient to give the function a null pointer to the socket address structure, which takes the form `(struct sockaddr *)&serv_name`.
- Following a run of `netserv.c` and `netclient.c`, typing `netstat` at the comand prompt of either machine (the client or server) will reveal one or two socket entries under "Active Internet connections" with the server's port number at the end of the "Local address" or "Foreign Address" field. The state of these sockets will be listed at `TIME_WAIT`. The socket exists in this state for approximately 30 seconds after it has been closed by the server in case a data packet that encountered network difficulty is late in arriving at the client. For this reason, it is also not possible to run `netserv.c` again until this socket has expired, because the port is technically still in use.

Sample Program 4: netclient.c, continued

Sample Run of `netclient.c`*:*

```
neyman% netclient
Client is alive and establishing socket connection.
Client has received 1 from socket.
Client has received 2 from socket.
Client has received 3 from socket.
Client has received 4 from socket.
Client has received 5 from socket.
Client has received 6 from socket.
Client has received 7 from socket.
Client has received 8 from socket.
Client has received 9 from socket.
Client has received 10 from socket.
Exiting now.
neyman%
```

Program 5: datagram1.c

This program is the first program in a series that demonstrates the capabilities of datagram (SOCK_DGRAM) sockets and the protocol that accompanies their use. In this initial example, one notes that a client does not have to connect explicitly to the server through a socket in order to read from the socket, but that it must first have a greeting read from the datagram socket before it may commence reading.

```
/* Author: Pete Broadwell, Grinnell College
   Minor Adjustments for Linux made 11/29/00 by Henry M. Walker

   This program uses datagram sockets to simulate a simple scenario of the
   readers-writers program. The writer (server) process reads the reader
   (client)'s greeting from the socket and then writes 10 integers to the
   socket. The client process, in an unconnected state, reads these integers.

*/

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>   /* socket command libraries needed by some compilers */
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>

#define SIM_LENGTH 9 /* number of integers to be written and read */
#define PORT 1228 /* local port on which connection will be established */

/* procedure to remove socket file and exit */
void clean_up(int cond, int *sock)
{ printf("Exiting now.\n");
  close(*sock);
  exit(cond);
} /* end of clean_up */

int main(void)
{ pid_t pid;

  pid = fork();     /* spawn new child process */
  if (-1 == pid)
    { perror ("Error in fork");
      exit(1);
    }

  if (0 == pid)
    { /* processing for client */
      int sock;
      struct sockaddr_in cli_name;
      int value; /* variable for number read from socket */
      size_t len; /* variable for size of cli_name structure */
      int count;
      char msg[20]; /* array for message to be written to socket */
```

```
    /* set the socket descriptor */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
      { perror("Error opening channel");
        clean_up(1, &sock);
      }

    /* set the physical address (cli_name) of the socket descriptor */
    bzero(&cli_name, sizeof(cli_name));
    cli_name.sin_family = AF_INET;
    cli_name.sin_port = htons(PORT);

    sleep(1); /* give server time to create socket */
    printf("Client is alive.\n");

    /* initialize the msg array */
    for (count = 0; count++; count < 20)
      msg[count] = 0;

    /* write greeting to the socket */
    len = sizeof(cli_name);
    strcpy(msg, "Hello, server");
    sendto(sock, &msg, strlen(msg), 0, (struct sockaddr *)&cli_name, len);
    printf("Client has sent  %s  to socket.\n", msg);

    /* read greeting from the server */
    recvfrom(sock, &msg, 20, 0, (struct sockaddr *)&cli_name, &len);
    printf("Client has received  %s  from socket.\n", msg);

    /* read integers from the socket */
    for (count = 0; count < SIM_LENGTH; count++)
      { recvfrom(sock, &value, 4, 0, (struct sockaddr *)&cli_name, &len);
        printf("Client has received %d from socket.\n", value);
      }

    close(sock); /* close connection to socket */
    exit(0);

  } /* end of processing for client */

else
  { /* processing for server */
    int sock;
    struct sockaddr_in serv_name;
    int count;
    char msg[20]; /* array for greeting to be read from socket */
    size_t len; /* variable for size of name structure */

    /* set socket descriptor */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
      { perror("Error opening channel");
        clean_up(1, &sock);
      }
```

```
    /* set the physical address (serv_name) of the socket descriptor */
    bzero(&serv_name, sizeof(serv_name));
    serv_name.sin_family = AF_INET;
    serv_name.sin_port = htons(PORT);

    /* bind the socket address to the socket descriptor */
    if (bind(sock, (struct sockaddr *)&serv_name, sizeof(serv_name)) < 0)
      { perror("Error naming channel");
        clean_up(1, &sock);
      }

    printf("Server is on line.\n");

    for (count = 0; count++; count < 20)
      msg[count] = 0;

    /* read greeting from client */
    len = sizeof(serv_name);
    recvfrom(sock, &msg, 20, 0, (struct sockaddr *)&serv_name, &len);
    printf("Server has read  %s  from socket.\n", msg);

    strcpy(msg, "Hello, client");

    /* write reply to client */
    sendto(sock, &msg, strlen(msg), 0, (struct sockaddr *)&serv_name, len);

    printf("Server has written  %s  to socket.\n", msg);

    /* write integers to socket */
    for (count = 1; count <= SIM_LENGTH; count++)
      { sendto(sock, &count, 4, 0, (struct sockaddr *)&serv_name, len);
        printf("Server has written %d to socket.\n", count);
      }

    waitpid(pid, NULL, 0); /* Wait for client process to terminate */
    clean_up(0, &sock); /* Exit with no errors */
  } /* end of processing for server */

} /* end of main */
```

Annotations for `datagram1.c`*:*

- The declaration of datagram sockets is almost identical to that of streaming Internet sockets, with the only difference existing in the setup of `sock`. The transfer protocol for datagram sockets, however, is quite different from that of streaming sockets.
- As before, the client must sleep while the server binds the socket. Once this is done, though, the client may immediately send data to the socket, without conducting any sort of three-way handshake. Because the socket is in an unconnected state, though, the client must use the `sendto` function instead of the `write` function to do this. In addition to the socket file descriptor, a pointer to the message to be sent and the message length, `sendto` also requires that it be sent any applicable flags (there are none in this case, so the value is 0), a null pointer to the socket address structure, and the size of the socket address structure.

- In this example, the client sends a greeting string to the server. Before the client may read any data from the socket, the server must read this greeting from the socket. The next example will show that this requirement becomes very important when a server must deal with multiple client (reader) processes. For now, it suffices to say that until the server reads this initial greeting string, the client has no way of knowing that the socket to which it has just written actually exists.
- When reading and writing to the socket, the server must also use different functions than before. To read the client's greeting string, it uses the `recvfrom` function. This function is similar to the `sendto` function, except that it requires a pointer to the size of the socket address structure instead of the actual size. Also, one may find the `MSG_PEEK` flag useful in many applications, because it allows the calling process to read the next item in the socket without removing it from the queue.
- Once the server has read the client's greeting, the integer transfer occurs as it has in earlier applications, with `sendto` replacing `write` and `recvfrom` replacing `read`.

Sample Run of `datagram1.c`*:*

```
leibniz% datagram1
Server is on line.
Client is alive.
Server has read  Hello, server  from socket.
Server has written  Hello, client  to socket.
Server has written 1 to socket.
Server has written 2 to socket.
Server has written 3 to socket.
Server has written 4 to socket.
Server has written 5 to socket.
Server has written 6 to socket.
Server has written 7 to socket.
Server has written 8 to socket.
Server has written 9 to socket.
Client has sent  Hello, server  to socket.
Client has received  Hello, client  from socket.
Client has received 1 from socket.
Client has received 2 from socket.
Client has received 3 from socket.
Client has received 4 from socket.
Client has received 5 from socket.
Client has received 6 from socket.
Client has received 7 from socket.
Client has received 8 from socket.
Client has received 9 from socket.
Exiting now.
leibniz%
```

Program 6: dgserv.c

This is the server portion of a demonstration that uses a shell script to execute an iterative datagram server process locally and multiple client processes remotely.

```
/* Author: Pete Broadwell, Grinnell College

   This program simulates a scenario of the readers-writers program in which
   multiple readers (clients) and a single writer (server) communicate through
   a datagram socket. This server process handles the clients iteratively. */

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>   /* socket command libraries needed by some compilers */
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>

#define SIM_LENGTH 6 /* number of integers to be written and read */
#define NUM_READERS 3
#define PORT 1229 /* local port on which the server establishes connection */

/* procedure to remove socket file and exit */
void clean_up(int cond, int *sock)
{ printf("Exiting now.\n");
  close(*sock);
  exit(cond);
} /* end of clean_up */

int main(void)
{ /* processing for server */
  int sock; /* socket file descriptor */
  struct sockaddr_in serv_name; /* socket address structure */
  int count, i, data;
  char msg[20]; /* array for greetings to be read from socket */
  size_t len; /* variable for the size of the cli_name structure */

  /* set socket descriptor */
  sock = socket(AF_INET, SOCK_DGRAM, 0);
  if (sock < 0)
    { perror("Opening channel");
      clean_up(1, &sock);
    }

  /* set the physical address (serv_name) of the socket descriptor */
  bzero(&serv_name, sizeof(serv_name));
  serv_name.sin_family = AF_INET;
  serv_name.sin_port = htons(PORT);

  /* bind the socket address to the socket descriptor */
  if (bind(sock, (struct sockaddr *)&serv_name, sizeof(serv_name)) < 0)
    { perror("Naming channel");
      clean_up(1, &sock);
    }
```

```
  printf("Server is on line.\n");

  /* initialize the msg array */
  for (count = 0; count < 20; count++)
    msg[count] = 0;

  /* iteratively read greetings from readers and write integers to them */
  len = sizeof(serv_name);

  for (i = 1; i <= NUM_READERS; i++)
    { recvfrom(sock, &msg, 20, 0, (struct sockaddr *)&serv_name, &len);
      printf("Server has read  %s  from socket.\n", msg);
      sendto(sock, &i, 4, 0, (struct sockaddr *)&serv_name, len);
      printf("Server has sent ID number to client %d.\n", i);

      for (count = 1; count <= SIM_LENGTH; count++)
        { data = ( (i - 1) * 10) + count;
          sendto(sock, &data, 4, 0, (struct sockaddr *)&serv_name, len);
          printf("Server has written %d to socket.\n", data);
        }
    }

  clean_up(0, &sock); /* Exit with no errors */

} /* end of main */
```

Annotations for `dgserv.c`*:*

- The difference to note in the processing of this datagram server is the manner in which it deals with multiple clients. Unlike a streaming socket server, this datagram server does not fork itself and deal recursively with each client process. Because clients do no explicitly connect to servers through datagram sockets, the server instead deals iteratively with each client.
- In the previous example, one noted that a client must have a greeting mesage read by the server in order to receive data from the socket. In this scenario, the server uses this requirement to coordinate its interactions with the client processes. It deals with each client one at a time by reading a single client's greeting and then writing the requisite number of integers to the socket. Because only one client is able to read from the socket (the client whose greeting the server has just read), it will read these integers and then exit, allowing the server to deal with the next client in line (that is, the next client with a greeting in the socket.)

Program 7: dgclient.c

This is the client portion of a demonstration that uses a shell script to execute an iterative datagram server process locally and multiple client processes remotely.

```
/* Author: Pete Broadwell
           Grinnell College

   This program simulates a scenario of the readers-writers program in which
   multiple readers (clients) and a single writer (server) communicate through
   a datagram socket. This is the processing for each client/reader.

*/

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>   /* socket command libraries needed by some compilers */
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>

#define SIM_LENGTH 6 /* number of integers to be written and read */
#define PORT 1229 /* local port on which the server establishes connection */

/* procedure to remove socket file and exit */
void clean_up(int cond, int *sock)
{ printf("Exiting now.\n");
  close(*sock);
  exit(cond);
} /* end of clean_up */

int main(int argc, char *argv[])
{ /* processing for client */
  int sock; /* socket file descriptor */
  struct sockaddr_in cli_name; /* socket address structure */
  int value; /* variable for number read from socket */
  size_t len; /* variable for the size of the cli_name structure */
  int client_id; /* variable for client's number as assigned by server */
  int count;
  char msg[20]; /* array for message to be written to socket */

  /* set the socket descriptor */
  sock = socket(AF_INET, SOCK_DGRAM, 0);
  if (sock < 0)
    { perror("Opening channel");
      clean_up(1, &sock);
    }

  /* set the physical address (cli_name) of the socket descriptor */
  bzero(&cli_name, sizeof(cli_name));
  cli_name.sin_family = AF_INET;
  cli_name.sin_addr.s_addr = inet_addr(argv[1]);
  cli_name.sin_port = htons(PORT);
```

Sample Program 7: dgclient.c, continued

```
  sleep(1); /* give server time to create the socket */

  printf("Client is alive.\n");

  /* initialize the msg array */
  for (count = 0; count < 20; count++)
    msg[count] = 0;

  /* write greeting to the socket */
  len = sizeof(cli_name);
  strcpy(msg, "Hello, server");

  sendto(sock, &msg, 20, 0, (struct sockaddr *)&cli_name, len);

  printf("Client has sent  %s  to socket.\n", msg);

  /* read identification number from socket */
  recvfrom(sock, &client_id, 4, 0, (struct sockaddr *)&cli_name, &len);
  printf("Client %d has received ID from server.\n", client_id);

  /* read integers from socket */
  for (count = 0; count < SIM_LENGTH; count++)
    { recvfrom(sock, &value, 4, 0, (struct sockaddr *)&cli_name, &len);
      printf("Client %d has received %d from socket.\n", client_id, value);
    }
  printf("Client %d done.\n", client_id);
  close(sock); /* close connection to socket */
  exit(0); /* exit with no errors */

} /* end of main */
```

Annotations for `dgclient.c`*:*

- The communications processing for this client is nearly identical to the actions of the client process in `datagram1.c`. That segment of code code has simply been isolated into its own program so that it may be run on several remote machines by a shell script. The only notable differnce between this client and the previous one is that this program takes the IP address of the intended server as a command-line argument.
- One should note that although the unreliability of the datagram protocol has not shown itself in this or the previous examples, it nonetheless exists. In streaming socket communications, a continuous byte stream runs back and forth between the client and the server, confirming the success or failure of each `read` or `write` operation. The extremely reliable nature of this communication has made TCP the standard communications protocol of many Internet applications, including SMTP (e-mail), FTP, and HTTP. The User Datagram Protocol layer, on the other hand, sends data in fixed-size datagrams, without any full-duplex byte stream to confirm whether or not the datagram reached its destination. If a datagram fails to reach its destination (due to network blockage or interference), the receiving application will often fail to notice this if it is in the midst of receiveing a large amount of data. This lack of *flow control* makes datagram sockets unsuitable for some communications applications.
- The inherent flexibility of datagram sockets that we have seen and will see even more in the next example makes them ideal for other applications, however. UDP is the standard transport layer for SNMP (network management) and other applications that require communication through sockets that are not explicitly connected.

Program 8: datagram2

This is the shell script that executes a datagram server locally and then executes multiple client processes on remote machines.

```
# datagram2
# Author:  Pete Broadwell, Grinnell College
# Minor Adjustments for Linux made 11/29/00 by Henry M. Walker

# This program coordinates the second datagraph sockets readers-writers
# simulation.  In this simulation, the writer (server) and reader (client)
# processes are separate programs.  This script first runs the server
# process and then remotely executes the client processes.  Also, the client
# processes take the IP address of the server as a command-line argument.
# Note:  IP address 132.161.33.175 is for MathLAN machine dijkstra, and this
# script must be run on dijkstra to establish the required server process.

# For security reasons, rsh operations on MathLAN require Kerberos
# authentication and must communicate using encryption.
# To use kerberos, some initial work is required.  See MathLAN software
# documentation.
# After Kerberos is set up for a user, the user must use the
# /usr/bin/kinit
# command to establish tickets before executing this script.

dgserv&
rsh -x frege    /home/walker/c/sockets/dgclient "132.161.33.175" &
rsh -x galton   /home/walker/c/sockets/dgclient "132.161.33.175" &
rsh -x hamilton /home/walker/c/sockets/dgclient "132.161.33.175" &
wait
```

Annotations for `datagram2`*:*

- The command line syntax for executing processes on other machines is simple. One needs only to include the name of the remote machine before the name of the program to be run in order to do this. Note that the clients are all given the IP address of the server computer as a command-line argument. As mentioned before, most clients by default try the well-known port of an Internet server first, so it is not necessary that the port number also be included as a command-line parameter. This value still must be specified within the code for the client, though.
- The ampersand after each procedure call signifies that the procedure in question will be run as an *attached job* on the remote host. This means that the remote computer will take over all execution of the process, allowing it to be run in true parallel with the jobs called on other machines.
- Since the output for the remotely-run clients must return to the server computer to be expressed at the terminal window, large disrepancies may occasionally arise in the order of the output.

Sample Run of `datagram2`*:*

```
pappus% datagram2
[1] 21578
Server is on line.
[2] 21579
[3] 21580
[4] 21581
Server has read  Hello, server  from socket.
Server has sent ID number to client 1.
Server has written 1 to socket.
Server has written 2 to socket.
Server has written 3 to socket.
Server has written 4 to socket.
Server has written 5 to socket.
Server has written 6 to socket.
Client is alive.
Client has sent  Hello, server  to socket.
Client 1 has received ID from server.
Client 1 has received 1 from socket.
Client 1 has received 2 from socket.
Client 1 has received 3 from socket.
Client 1 has received 4 from socket.
Client 1 has received 5 from socket.
Client 1 has received 6 from socket.
Client 1 done.
Server has written 7 to socket.
Server has read  Hello, server  from socket.
Server has sent ID number to client 2.
Server has written 11 to socket.
Server has written 12 to socket.
Server has written 13 to socket.
Server has written 14 to socket.
Server has written 15 to socket.
Server has written 16 to socket.
Server has written 17 to socket.
Client is alive.
Client has sent  Hello, server  to socket.
Client 2 has received ID from server.
Client 2 has received 11 from socket.
Client 2 has received 12 from socket.
Client 2 has received 13 from socket.
Client 2 has received 14 from socket.
Client 2 has received 15 from socket.
Client 2 has received 16 from socket.
Client 2 done.
Server has read  Hello, server  from socket.
Server has sent ID number to client 3.
Server has written 21 to socket.
Server has written 22 to socket.
Server has written 23 to socket.
Server has written 24 to socket.
Server has written 25 to socket.
Server has written 26 to socket.
Server has written 27 to socket.
Client is alive.
```

Sample Program 8: datagram2, continued

```
Client has sent  Hello, server  to socket.
Client 3 has received ID from server.
Client 3 has received 21 from socket.
Client 3 has received 22 from socket.
Client 3 has received 23 from socket.
Client 3 has received 24 from socket.
Client 3 has received 25 from socket.
Client 3 has received 26 from socket.
Client 3 done.
Exiting now.
[4]  +Done                  mathieu dgclient 132.161.33.93
[3]  -Done                  banach dgclient 132.161.33.93
[2]  +Done                  goedel dgclient 132.161.33.93
[1]  +Done                  dgserv
pappus%
```

Program 9: datagram3.c

This third datagram sockets example program simulates a true multiple readers-multiple writers problem. The overall server process creates two sockets, one into which the writer clients write and one from which the reader clients read. Due to this setup, the server acts as the manager of the central buffer, transfering data from the write socket to the read socket.

```
/* Author: Pete Broadwell, Grinnell College
        Minor Adjustments for Linux made 11/29/00 by Henry M. Walker
   This program simulates a scenario of the readers-writers program in which
   multiple reader and multiple writer processes (which are all clients, in
   this case) communicate through a pair of datagram sockets.
   A single server process manages this data transfer by reading integers from
   the writers' socket, processing them, and writing them to the readers'
   socket. The server deals with the reader processes iteratively. */

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>   /* socket command libraries needed by some compilers */
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>

#define WRITE_LENGTH 5
#define STRING_LENGTH 30
#define READ_LENGTH 5
#define NUM_READERS 3
#define NUM_WRITERS 3
#define READ_PORT 1229 /* local port on which read read is established */
#define WRITE_PORT 1230 /* local port on which write socket is established */

int read_sock, write_sock; /* global variable for the communication sockets */

/* procedure to remove socket file and exit */
void clean_up(int cond, int *read_sock, int *write_sock)
{ printf("Exiting now.\n");
  /* remove both socket files */
  close(*read_sock);
  close(*write_sock);
  exit(cond);
} /* end of clean_up */

int main(void)
{ pid_t pid;
  int p_count;
  pid_t proc[NUM_READERS+NUM_WRITERS];

  /* spawn client processes */
  for (p_count = 1; p_count <= NUM_READERS; p_count++)
    { pid = fork();
      if (-1 == pid)
        { perror ("Error in fork");
          exit (1);
        }
```

```
    if (0 == pid)
      { /* processing for reader */
        int read_sock; /* file descriptor for read socket */
        struct sockaddr_in read_name; /* read socket address structure */
        int value; /* variable for number read from socket */
        size_t len; /* variable for the size of the read_name structure */
        int count;
        char msg[STRING_LENGTH];

        /* set the read socket descriptor */
        read_sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (read_sock < 0)
          { perror ("Error opening channel");
            clean_up(1, &read_sock, &write_sock);
          }

        /* set the physical address (read_name) of the read_sock descriptor */
        bzero(&read_name, sizeof(read_name));
        read_name.sin_family = AF_INET;
        read_name.sin_port = htons(READ_PORT);

        sleep(1); /* wait for server to create the read socket */
        printf("Reader %d is alive.\n", p_count);

        /* clean out msg array */
        for (count = 0; count < STRING_LENGTH; count++)
          msg[count] = 0;

        /* write greeting to the read socket */
        len = sizeof(read_name);
        sprintf(msg, "Hello from reader %d", p_count);
        sendto(read_sock, &msg, STRING_LENGTH, 0,
               (struct sockaddr *)&read_name, len);
        printf("Reader %d has sent  %s  to read socket.\n", p_count, msg);

        /* read integers from the read socket */
        for (count = 0; count < READ_LENGTH; count++)
          { recvfrom(read_sock, &value, 4, 0,
                     (struct sockaddr *)&read_name, &len);
            printf("Reader %d has received %d from read socket.\n", p_count,
                 value);
          }
        printf("Reader %d done.\n", p_count);
        close(read_sock); /* close connection to read socket */
        exit(0);
      }
    else proc[p_count - 1] = pid;
  } /* end of processing for reader */

/* spawn writer processes */
for (p_count = 1; p_count <= NUM_WRITERS; p_count++)
  { pid = fork();
    if (-1 == pid)
      { perror ("Error in fork");
        exit (1);
      }
```

```
    if (0 == pid)
      { /* processing for writer */
        int write_sock;
        struct sockaddr_in write_name;
        int count, data;
        size_t len;

        /* set write socket descriptor */
        write_sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (write_sock < 0)
          { perror ("Error opening channel");
            clean_up(1, &read_sock, &write_sock);
          }

     /* set the physical address (write_name) of the write_sock descriptor */
        bzero(&write_name, sizeof(write_name));
        write_name.sin_family = AF_INET;
        write_name.sin_port = htons(WRITE_PORT);

        sleep(1); /* wait for server to create write socket */\
        printf("Writer %d is alive.\n", p_count);

        /* write integers to write socket */
        len = sizeof(write_name);
        for (count = 1; count <= WRITE_LENGTH; count++)
          { data = ( (p_count - 1) * 10) + count;
            sendto(write_sock, &data, 4, 0, (struct sockaddr *)&write_name, len);
            printf("Writer %d has written %d to write socket.\n", p_count, data);
          }

        printf("Writer %d done.\n", p_count);
        close(write_sock); /* close connection to write socket */
        exit(0);
      }
    else proc[NUM_READERS + p_count - 1] = pid;
  } /* end of processing for writer */

{ /* processing for main server */
  int read_sock, write_sock;
  struct sockaddr_in read_name, write_name;
  size_t read_len, write_len;
  char msg[STRING_LENGTH];
  int count, place, i, value;

  /* set socket descriptors */
  read_sock = socket(AF_INET, SOCK_DGRAM, 0);
  if (read_sock < 0)
    { perror ("Error opening channel");
      clean_up(1, &read_sock, &write_sock);
    }

  write_sock = socket(AF_INET, SOCK_DGRAM, 0);
  if (write_sock < 0)
    { perror ("Error opening channel");
      clean_up(1, &read_sock, &write_sock);
    }
```

```
    /* set the physical addresses of the socket descriptors */
    bzero(&read_name, sizeof(read_name));
    read_name.sin_family = AF_INET;
    read_name.sin_port = htons(READ_PORT);

    bzero(&write_name, sizeof(write_name));
    write_name.sin_family = AF_INET;
    write_name.sin_port = htons(WRITE_PORT);

    /* bind the socket addresses to the socket descriptors */
    if (bind(read_sock, (struct sockaddr *)&read_name, sizeof(read_name)) < 0)
      { perror ("Error naming channel");
        clean_up(1, &read_sock, &write_sock);
      }

    if (bind(write_sock, (struct sockaddr *)&write_name, sizeof(write_name)) < 0)
      { perror ("Error naming channel");
        clean_up(1, &read_sock, &write_sock);
      }

    printf("Main server is on line.\n");

    /* initialize the msg array */
    for (count = 0; count < STRING_LENGTH; count++)
      msg[count] = 0;

    write_len = sizeof(write_name);
    read_len = sizeof(read_name);

    /* manage data transfer */
    for (count = 1; count <= NUM_READERS; count++)
      { /* read readers' greetings one at a time */
        recvfrom(read_sock, &msg, STRING_LENGTH, 0,
                 (struct sockaddr *)&read_name, &read_len);
        printf("Server has read  %s  from read socket.\n", msg);

        /* read a number of integers from the write socket, alter them, and
           write them to the read socket */
        for (i = 0; i < READ_LENGTH; i++)
          { place = ( (count - 1) * 10) + i;
            recvfrom(write_sock, &value, 4, 0,
                     (struct sockaddr *)&write_name, &write_len);
            printf("Server has read %d from write socket.\n", value);
            value = value * 100;
            sendto(read_sock, &value, 4, 0,
                   (struct sockaddr *)&read_name, read_len);
            printf("Server has written %d to read socket.\n", value);
          }
       }

    for (i = 0; i < NUM_READERS + NUM_WRITERS; i++)
      waitpid(proc[i], NULL, 0); /* Wait for child processes to terminate */
    clean_up(0, &read_sock, &write_sock); /* Exit with no errors */

  } /* end of processing for main server */
} /* end of main */
```

Annotations for `datagram3.c`*:*

- From the structure of this program, one can observe that while it is possible for several processes to write to a socket at the same time, it is not possible for several proesses to read from a socket at once. The overall server process in this simulation thus creates two sockets, one to which all of the writer client processes will write, and one from which the reader processes will read.
- It is possible for the server to read from the write socket while the writers are writing to it, and that is exactly what it does. The server reads the written integers from the write socket one at a time, alters them slightly (as is often the case in client-client data transfer) and then deposits them in the read socket.
- The server deals with the reader processes iteratively, as in the previous example. It reads a reader client's greeting from the read socket, and then writes the required number of integers to this socket. The reader in question immediately reads these integers and then exits, at which point the server repeats the procedure.
- Note that this sort of flexibility, especially in allowing the writers to write to their socket at once while the server reads from it, would be difficult to accomplish with streaming sockets. This problem thus illustrates an advantage of datagram sockets.

Sample Run of `datagram3.c`*:*

```
leibniz% datagram3
Main server is on line.
Reader 2 is alive.
Reader 1 is alive.
Server has read  Hello from reader 1  from read socket.
Reader 1 has sent  Hello from reader 1  to read socket.
Writer 1 is alive.
Server has read 1 from write socket.
Reader 1 has received 100 from read socket.
Writer 2 is alive.
Writer 2 has written 11 to write socket.
Writer 2 has written 12 to write socket.
Writer 2 has written 13 to write socket.
Writer 2 has written 14 to write socket.
Writer 2 has written 15 to write socket.
Writer 2 done.
Reader 3 is alive.
Reader 3 has sent  Hello from reader 3  to read socket.
Server has written 100 to read socket.
Server has read 11 from write socket.
Reader 1 has received 1100 from read socket.
Writer 1 has written 1 to write socket.
Writer 1 has written 2 to write socket.
Writer 1 has written 3 to write socket.
Writer 1 has written 4 to write socket.
Writer 1 has written 5 to write socket.
Writer 1 done.
Writer 3 is alive.
Writer 3 has written 21 to write socket.
Writer 3 has written 22 to write socket.
Writer 3 has written 23 to write socket.
Writer 3 has written 24 to write socket.
```

```
Writer 3 has written 25 to write socket.
Writer 3 done.
Server has written 1100 to read socket.
Server has read 12 from write socket.
Reader 1 has received 1200 from read socket.
Server has written 1200 to read socket.
Server has read 13 from write socket.
Reader 1 has received 1300 from read socket.
Server has written 1300 to read socket.
Server has read 14 from write socket.
Reader 1 has received 1400 from read socket.
Reader 1 done.
Server has written 1400 to read socket.
Server has read  Hello from reader 3  from read socket.
Server has read 15 from write socket.
Reader 3 has received 1500 from read socket.
Server has written 1500 to read socket.
Server has read 2 from write socket.
Reader 3 has received 200 from read socket.
Server has written 200 to read socket.
Server has read 3 from write socket.
Reader 3 has received 300 from read socket.
Server has written 300 to read socket.
Server has read 4 from write socket.
Reader 3 has received 400 from read socket.
Server has written 400 to read socket.
Server has read 5 from write socket.
Reader 3 has received 500 from read socket.
Reader 3 done.
Server has written 500 to read socket.
Server has read  Hello from reader 2  from read socket.
Server has read 21 from write socket.
Server has written 2100 to read socket.
Server has read 22 from write socket.
Server has written 2200 to read socket.
Server has read 23 from write socket.
Server has written 2300 to read socket.
Server has read 24 from write socket.
Server has written 2400 to read socket.
Server has read 25 from write socket.
Server has written 2500 to read socket.
Reader 2 has sent  Hello from reader 2  to read socket.
Reader 2 has received 2100 from read socket.
Reader 2 has received 2200 from read socket.
Reader 2 has received 2300 from read socket.
Reader 2 has received 2400 from read socket.
Reader 2 has received 2500 from read socket.
Reader 2 done.
Exiting now.
leibniz%
```