Code samples (UNIX domain)

Examples of client and server socket programs make up the rest of this section. These programs illustrate how to use the sockets interface to establish a connection and transmit data between hosts using TCP.

To send data between stream sockets (type SOCK_STREAM), the sockets must be connected. ``Sample UNIX server'' and ``Sample UNIX client'' show two programs that create such a connection. The client program is relatively simple. To initiate a connection, this program simply creates a stream socket, then calls **connect**, specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter). If a process persists in sending messages after the connection is closed, a **SIGPIPE** signal is sent to the process by the operating system. Unless explicit action is taken to handle the signal (see **signal**(S)) the process terminates, and the shell prints the message `broken pipe`.

Forming a connection is asymmetrical; one process, such as the program in ``Sample UNIX client'', requests a connection with a particular socket, the other process accepts connection requests. Before a connection can be accepted a socket must be created and an address bound to it.

If there are several possible communicants, one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection may be destroyed by closing the corresponding socket.

The program in ``Sample UNIX server'' is a simple example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case, it prints out the socket pathname.) The program then calls **listen** for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. **listen** marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of **listen**; the maximum length is limited by the system. Once the listen call has been completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and thus a new socket for the connection is created.

Server code (UNIX domain)

The following code creates a server that waits for client connection requests. The manual page for each routine defines the necessary *#include* files when calling that routine.

This program creates a socket in the UNIX domain and binds a name to it. After printing the socket's name it begins a loop. Each time through the loop it accepts a connection and prints out messages from it. When the connection breaks, or a termination message comes through, the program accepts a new connection.

Sample UNIX server

```
  1 #include <sys/types.h>
  2 #include <sys/socket.h>
  3 #include <sys/un.h>
  4 #include <stdio.h>


  5 #define NAME "socket"


  6 main()
  7 {
  8     int sock, msgsock, rval;
  9     struct sockaddr_un server;
 10     char buf[1024];


 11     sock = socket(AF_UNIX, SOCK_STREAM, 0);
 12     if (sock < 0) {
 13         perror("opening stream socket");
 14         exit(1);
 15     }
 16     server.sun_family = AF_UNIX;
 17     strcpy(server.sun_path, NAME);
 18     if (bind(sock, (struct sockaddr *) &server, sizeof(struct sockaddr_un))) {
 19         perror("binding stream socket");
 20         exit(1);
 21     }
 22     printf("Socket has name %s\n", server.sun_path);
 23     listen(sock, 5);
 24     for (;;) {
 25         msgsock = accept(sock, 0, 0);
 26         if (msgsock == -1)
 27             perror("accept");
 28         else do {
 29             bzero(buf, sizeof(buf));
 30             if ((rval = read(msgsock, buf, 1024)) < 0)
 31                 perror("reading stream message");
 32             else if (rval == 0)
 33                 printf("Ending connection\n");
 34             else
 35                 printf("-->%s\n", buf);
 36         } while (rval > 0);
 37         close(msgsock);
 38     }
 39     close(sock);
 40     unlink(NAME);
 41 }
```

lines 11-14
    Create the socket

lines 16-20
: Name and bind the socket using file system name

lines 23-37
: Start accepting connections, and start reading on the accepted connection

lines 39-40
: These statements are not executed, because they follow an infinite loop. However, most ordinary programs will not run forever. In the UNIX domain it is necessary to tell the file system that one is through using NAME. In most programs one uses the call **unlink** as in line 40. Because the user has to kill this program, it is necessary to remove the name by a command from the shell.

Client code (UNIX domain)

This program connects to the socket named in the command line and sends a one line message to that socket. The form of the command line is:

programname pathname

Sample UNIX client

```
  1 #include <sys/types.h>
  2 #include <sys/socket.h>
  3 #include <sys/un.h>
  4 #include <stdio.h>


  5 #define DATA "Half a league, half a league . . ."


  6 main(argc, argv)
  7     int argc;
  8     char *argv[];
  9 {
 10     int sock;
 11     struct sockaddr_un server;
 12     char buf[1024];


 13     if (argc < 2) {
 14         printf("usage:%s <pathname>", argv[0]);
 15         exit(1);
 16     }


 17     sock = socket(AF_UNIX, SOCK_STREAM, 0);
 18     if (sock < 0) {
 19         perror("opening stream socket");
 20         exit(1);
 21     }
```

```
22     server.sun_family = AF_UNIX;
23     strcpy(server.sun_path, argv[1]);


24     if (connect(sock, (struct sockaddr *) &server, sizeof(struct sockaddr_un)) < 0) {
25         close(sock);
26         perror("connecting stream socket");
27         exit(1);
28     }
29     if (write(sock, DATA, sizeof(DATA)) < 0)
30         perror("writing on stream socket");
31     close(sock);
32 }
```

lines 13-16
: Check command-line arguments for validity

lines 17-21
: Create socket

lines 22-28
: Connect socket using name specified by command line

lines 29-31
: Write data on the connected socket and close the socket before exiting the program

---

Next topic: Datagrams in the UNIX domain
Previous topic: Data transfer (UNIX domain)