BSD Sockets: A Quick And Dirty Primer
by Jim Frost
November 22, 1989

As you delve into the mysteries of UNIX, you find more and more things that are difficult to understand immediately. One of these things, at least for most people, is the BSD socket concept. This is a short tutorial that explains what they are, how they work, and gives sample code showing how to use them.

The Analogy
(or: What *IS* a socket, anyway?)

The *socket* is the BSD method for accomplishing interprocess communication (IPC). What this means is a socket is used to allow one process to speak to another, very much like the telephone is used to allow one person to speak to another.

The telephone analogy is a very good one, and will be used repeatedly to describe socket behavior.

Installing Your New Phone
(or: How to listen for socket connections)

In order for a person to receive telephone calls, he must first have a telephone installed. Likewise you must create a socket to listen for connections. This process involves several steps. First you must make a new socket, which is similar to having a telephone line installed. The *socket()* command is used to do this.

Since sockets can have several types, you must specify what type of socket you want when you create one. One option that you have is the addressing format of a socket. Just as the mail service uses a different scheme to deliver mail than the telephone company uses to complete calls, so can sockets differ. The two most common addressing schemes are *AF_UNIX* and *AF_INET*. AF_UNIX addressing uses UNIX pathnames to identify sockets; these sockets are very useful for IPC between processes on the same machine. AF_INET addressing uses Internet addresses which are four byte numbers usually written as four decimal numbers separated by periods (such as 192.9.200.10). In addition to the machine address, there is also a port number which allows more than one AF_INET socket on each machine. AF_INET addresses are what we will deal with here.

Another option which you must supply when creating a socket is the type of socket. The two most common types are *SOCK_STREAM* and *SOCK_DGRAM*. SOCK_STREAM indicates that data will come across the socket as a stream of characters, while SOCK_DGRAM indicates that data will come in bunches (called datagrams). We will be dealing with SOCK_STREAM sockets, which are very common.

After creating a socket, we must give the socket an address to listen to, just as you get a telephone number so that you can receive calls. The *bind()* function is used to do this (it binds a socket to an address, hence the name).

SOCK_STREAM type sockets have the ability to queue incoming connection requests, which is a lot like having "call waiting" for your telephone. If you are busy handling a connection, the connection request will wait until you can deal with it. The *listen()* function is used to set the maximum number of requests (up to a maximum of five, usually) that will be queued before requests start being denied. While it is not necessary to use the listen() function, it's good practice.

The following function shows how to use the socket(), bind(), and listen() functions to establish a socket which can accept calls:

```
/* code to establish a socket; originally from bzs@bu-cs.bu.edu
 */

int establish(portnum)
u_short portnum;
{ char   myname[MAXHOSTNAME+1];
  int    s;
  struct sockaddr_in sa;
  struct hostent *hp;

  bzero(&sa,sizeof(struct sockaddr_in));      /* clear our address */
  gethostname(myname,MAXHOSTNAME);            /* who are we? */
  hp= gethostbyname(myname);                  /* get our address info */
  if (hp == NULL)                             /* we don't exist !? */
    return(-1);
  sa.sin_family= hp->h_addrtype;              /* this is our host address */
  sa.sin_port= htons(portnum);                /* this is our port number */
  if ((s= socket(AF_INET,SOCK_STREAM,0)) < 0) /* create socket */
    return(-1);
  if (bind(s,&sa,sizeof sa,0) < 0) {
    close(s);
    return(-1);                               /* bind address to socket */
  }
  listen(s, 3);                               /* max # of queued connects */
  return(s);
}
```

After you create a socket to get calls, you must wait for calls to that socket. The *accept()* function is used to do this. Calling accept() is analogous to picking up the telephone if it's ringing. Accept() returns a new socket which is connected to the caller.

The following function can be used to accept a connection on a socket that has been created using the establish() function above:

```
int get_connection(s)
int s;                    /* socket created with establish() */
{ struct sockaddr_in isa; /* address of socket */
  int i;                  /* size of address */
  int t;                  /* socket of connection */

  i = sizeof(isa);                   /* find socket's address */
  getsockname(s,&isa,&i);            /* for accept() */

  if ((t = accept(s,&isa,&i)) < 0)   /* accept connection if there is one */
    return(-1);
  return(t);
}
```

Unlike with the telephone, you may still accept calls while processing previous connections. For this reason you usually fork off jobs to handle each connection. The following code shows how to use establish() and get_connection() to allow multiple connections to be dealt with:

```
#include <errno.h>       /* obligatory includes */
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORTNUM 50000 /* random port number, we need something */

void fireman(), do_something();

main()
{ int s, t;

  if ((s= establish(PORTNUM)) < 0) {  /* plug in the phone */
    perror("establish");
    exit(1);
  }

  signal(SIGCHLD, fireman);           /* this eliminates zombies */

  for (;;) {                          /* loop for phone calls */
    if ((t= get_connection(s)) < 0) { /* get a connection */
      if (errno == EINTR)             /* EINTR might happen on accept(), */
        continue;                     /* try again */
      perror("accept");               /* bad */
      exit(1);
    }
    switch(fork()) {                  /* try to handle connection */
    case -1 :                         /* bad news.  scream and die */
      perror("fork");
      close(s);
      close(t);
      exit(1);
    case 0 :                          /* we’re the child, do something */
      do_something(t);
      exit(0);
    default :                         /* we’re the parent so look for */
      close(t);                       /* another connection */
      continue;
    }
  }
}

/* as children die we should get catch their returns or else we get
 * zombies, A Bad Thing.  fireman() catches falling children.
 */

void fireman()
{ union wait wstatus;

  while(wait3(&wstatus,WNOHANG,NULL) >= 0);
```

```
}

/* this is the function that plays with the socket.  it will be called
 * after getting a connection.
 */

void do_something(s)
int s;
{
  /* do your thing with the socket here
      :
      :
   */
}
```

Dialing
(or: How to call a socket)

You now know how to create a socket that will accept incoming calls. So how do you call it? As with the telephone, you must first have the phone before using it to call. You use the socket() function to do this, exactly as you establish a socket to listen to.

After getting a socket to make the call with, and giving it an address, you use the *connect()* function to try to connect to a listening socket. The following function calls a particular port number on a particular host:

```
int call_socket(hostname, portnum)
char *hostname;
{ struct sockaddr_in sa;
  struct hostent     *hp;
  int a, s;

  if ((hp= gethostbyname(hostname)) == NULL) { /* do we know the host's */
    errno= ECONNREFUSED;                       /* address? */
    return(-1);                                /* no */
  }

  bzero(&sa,sizeof(sa));
  bcopy(hp->h_addr,(char *)&sa.sin_addr,hp->h_length); /* set address */
  sa.sin_family= hp->h_addrtype;
  sa.sin_port= htons((u_short)portnum);

  if ((s= socket(hp->h_addrtype,SOCK_STREAM,0)) < 0)   /* get socket */
    return(-1);
  if (connect(s,&sa,sizeof sa) < 0) {                  /* connect */
    close(s);
    return(-1);
  }
  return(s);
}
```

This function returns a connected socket through which data can flow.

Conversation
(or: How to talk between sockets)

Now that you have a connection between sockets you want to send data between them. The *read()* and *write()* functions are used to do this, just as they are for normal files. There is only one major difference between socket reading and writing and file reading and writing: you don't usually get back the same number of characters that you asked for, so you usually loop until you have read the number of characters that you want. A simple function to read a given number of characters into a buffer is:

```
int read_data(s,buf,n)
int  s;                /* connected socket */
char *buf;             /* pointer to the buffer */
int  n;                /* number of characters (bytes) we want */
{ int bcount,          /* counts bytes read */
      br;              /* bytes read this pass */

  bcount= 0;
  br= 0;
  while (bcount < n) {             /* loop until full buffer */
    if ((br= read(s,buf,n-bcount)) > 0) {
      bcount += br;                /* increment byte counter */
      buf += br;                   /* move buffer ptr for next read */
    }
    if (br < 0)                    /* signal an error to the caller */
      return(-1);
  }
  return(bcount);
}
```

A very similar function should be used to write data; we leave that function as an exercise to the reader.

Hanging Up
(or: What to do when you're done with a socket)

Just as you hang up when you're through speaking to someone over the telephone, so must you close a connection between sockets. The normal *close()* function is used to close each end of a socket connection. If one end of a socket is closed and the other tries to write to its end, the write will return an error.

Speaking The Language
(or: Byte order is important)

Now that you can talk between machines, you have to be careful what you say. Many machines use differing dialects, such as ASCII versus (yech) EBCDIC. More commonly there are byte-order problems. Unless you always pass text, you'll run up against the byte-order problem. Luckily people have already figured out what to do about it.

Once upon a time in the dark ages someone decided which byte order was "right". Now there exist functions that convert one to the other if necessary. Some of these functions are *htons()* (host to network short integer), *ntohs()* (network to host short integer), *htoni()* (host to network integer), *ntohi()* (network to host integer), *htonl()* (host to network long integer), and *ntohl()* (network to host long integer). Before sending an integer through a socket, you should first massage it with the htoni() function:

```
i= htoni(i);
write_data(s, &i, sizeof(i));
```

and after reading data you should convert it back with ntohi():

```
read_data(s, &i, sizeof(i));
i= ntohi(i);
```

If you keep in the habit of using these functions you'll be less likely to goof it up in those circumstances where it is necessary.

The Future Is In Your Hands
(or: What to do now)

Using just what's been discussed here you should be able to build your own programs that communicate with sockets. As with all new things, however, it would be a good idea to look at what's already been done. Many public domain programs exist which make use of the socket concept, and many books exist which go into much more depth than I have here. In addition I've deliberately left out a lot of details such as what kinds of things can go wrong; the manual pages for each of the functions should be consulted for this information.

If you have further questions about sockets or this primer, please feel free to ask me at email address madd@bu-it.bu.edu.

Jim Frost
Saber Software
(617) 876-7636
madd@saber.com