Socket Programming and Its Role in Networking

Pankaj Singh, Mohd Wakil and Pankaj Singh

December 18, 2019

SOCKET PROGRAMMING AND ITS ROLE IN NETWORKING

[1]Prof. Pankaj Singh

[2]Prof.Mohd. Wakil

[3]Pankaj Singh

---

1. **[1]Prof & Dean (R D Engineering college) p.mnavy@gmail.com**
2. **[2]Prof & Head –CS (R D Engineering college) mohdvakil@gmail.com**
3. **[3]Asst.Prof.-CS (R D Engineering college) pankajsingh59@gmail.com**

Abstract

A socket represents a single connection between exactly two pieces of software. It is a communications connection point (endpoint) that you can name and address in a network. Sockets allow applications to communicate using standard mechanisms built into network hardware and operating systems. A socket also allows the exchange of information between processes on the same machine or across a network, distributes work to the most efficient machine, and allows access to centralized data easily. The processes that use a socket can reside on the same system or on different systems on different networks. Sockets are useful for both stand-alone as well as network applications. Network standards for TCP/IP Socket are provided by the application program interfaces (APIs). A wide range of operating systems support socket APIs. Socket programming shows how to use socket APIs to establish communication links between remote and local processes. OS/400 sockets support multiple transport and networking protocols. Also socket system functions and the socket network functions are thread safe. Programmers who use Integrated Language Environment (ILE) C can use the information to develop socket applications.

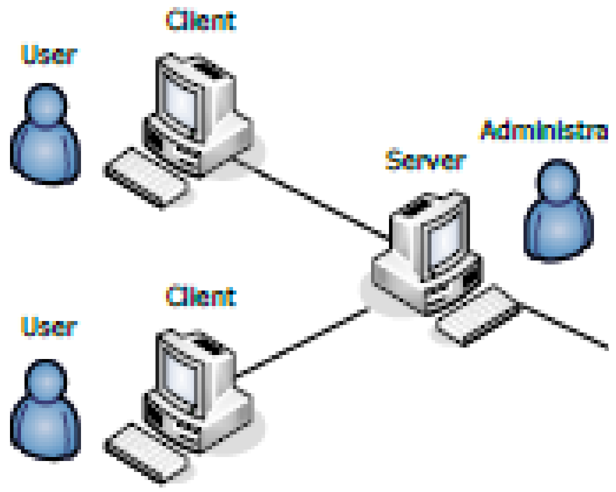
Keywords: Socket, Network Hardware, Network Applications, Network Standards, Integrated Language Environment

ARCHITECTURE

In the simplex duplex communication model the messages in a chat room are directly delivered to the remote side without the need for an intermediate node which is required for message forwarding. This kind of communication represents one to one communication.

User
Client
Server
User

While in many to many communication taking place in a chat room , it is the work of a server which act as a central message processor and it receives messages from any one of the on line clients and then “broadcasts “ them to all the users.



SOCKET PROGRAMMING

Running the server

It is the server program that allows clients to upload and download files to and from a specified directory.

The server is started or run as follows:

server directory port

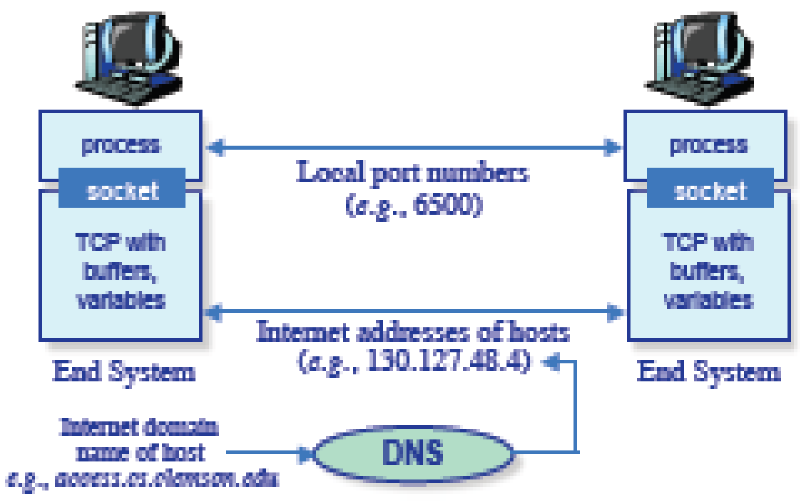
Where:

"directory" is the location of the files to be accessed.

"port" is the TCP port number that clients will use to locate the server

File server listening on port 10000

After the server has started, it simply spins in an infinite loop waiting for incoming connections until you decide to kill it. Needless to say, the server isn't too terribly exciting to watch (despite the fact that it is the most interesting part of the project to implement).


process
socket
TCP with buffers, variables
End System
Local port numbers (e.g., 6500)
Internet addresses of hosts (e.g., 130.127.48.4)
process
socket
TCP with buffers, variables
End System
Internet domain name of host e.g., access.cs.clemson.edu
DNS

Sockets are addressed using an IP address and port Number

When the server starts , it attempts to open up a socket on the given port .if it prints a message , it means that socket opens up.The message has the following format :

% server /home/beazley/myfiles 10000
File server listening on classes.cs.uchicago.edu port 10000

If however the given port is in use and the server is unable to open it It prints the following message , which represents an error and exits :

```
% server /home/beazley/myfiles 10000
   Unable to bind to port 10000.  Sorry!
%
```

If we get the above message it shows that the port which we requested is already in use .In this case we should try to choose another port number and try again .But while choosing the port number we should be careful that the port number is greater than 1024 and should be less than 65535.

Also if we try to kill the server and immediately start it using the same port number, it might result in error message. Sometimes the operating system takes some time in reclaiming the port numbers that are no longer in use. If this happens to use, a different port number should br picked up or wait a little while.

ACCEPTING A CONNECTION

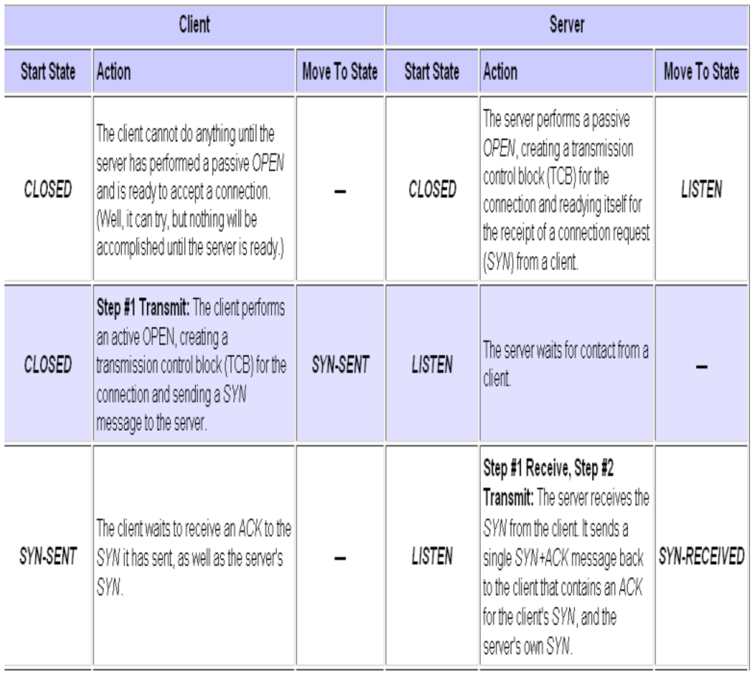
When the server and client are connected for the first time , the server should use fork() to make the client handler run as a different process. And meanwhile the server goes back to listen to more client connections .As an example the main loop of the server will look roughly as the following:

```
while(1)
{
  int clientfd = accept( ... )        // Wait for a connection
  if (!fork())
    {
       // Child process
       handle_client(clientfd);       // Go listen to the client
       close(clientfd);               // Close the client
       exit(0);
  } else
  {
       // Parent process.  Do nothing.
  }
  close(clientfd);
}
```

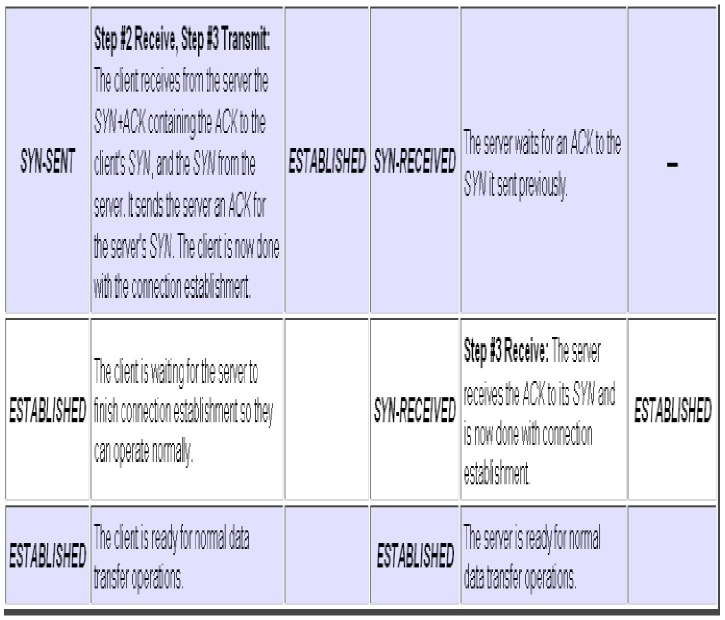
We should keep one thing in mind that the fork() makes a copy of the running server program and returns two distinct values . The process that called fork(), returns a non –zero process id of the child .In the child ,fork() returns zero.

HANDSHAKING

TCP "Three-Way Handshake" Connection Establishment Procedure



Client			Server		
Start State	Action	Move To State	Start State	Action	Move To State
CLOSED	The client cannot do anything until the server has performed a passive *OPEN* and is ready to accept a connection. (Well, it can try, but nothing will be accomplished until the server is ready.)	–	*CLOSED*	The server performs a passive *OPEN*, creating a transmission control block (TCB) for the connection and readying itself for the receipt of a connection request (*SYN*) from a client.	*LISTEN*
CLOSED	**Step #1 Transmit:** The client performs an active OPEN, creating a transmission control block (TCB) for the connection and sending a *SYN* message to the server.	*SYN-SENT*	*LISTEN*	The server waits for contact from a client.	–
SYN-SENT	The client waits to receive an *ACK* to the *SYN* it has sent, as well as the server's *SYN*.	–	*LISTEN*	**Step #1 Receive, Step #2 Transmit:** The server receives the *SYN* from the client. It sends a single *SYN+ACK* message back to the client that contains an *ACK* for the client's *SYN*, and the server's own *SYN*.	*SYN-RECEIVED*



SYN-SENT	**Step #2 Receive, Step #3 Transmit:** The client receives from the server the SYN+ACK containing the ACK to the client's SYN, and the SYN from the server. It sends the server an ACK for the server's SYN. The client is now done with the connection establishment.	ESTABLISHED	SYN-RECEIVED	The server waits for an ACK to the SYN it sent previously.	-
ESTABLISHED	The client is waiting for the server to finish connection establishment so they can operate normally.		SYN-RECEIVED	**Step #3 Receive:** The server receives the ACK to its SYN and is now done with connection establishment.	ESTABLISHED
ESTABLISHED	The client is ready for normal data transfer operations.		ESTABLISHED	The server is ready for normal data transfer operations.	

The server should send a message to the client identifying itself after receiving a connection and forking the client handler.And all this process should be completed using a packet called DAT packet to the client that contains a greeting message.

The client should reply by sending an ACK. Following lines illustrate the working of the process:

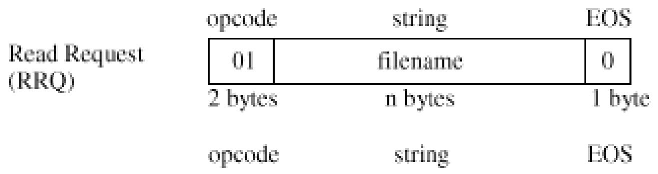
CLIENT CONNECTING TO THE SERVER

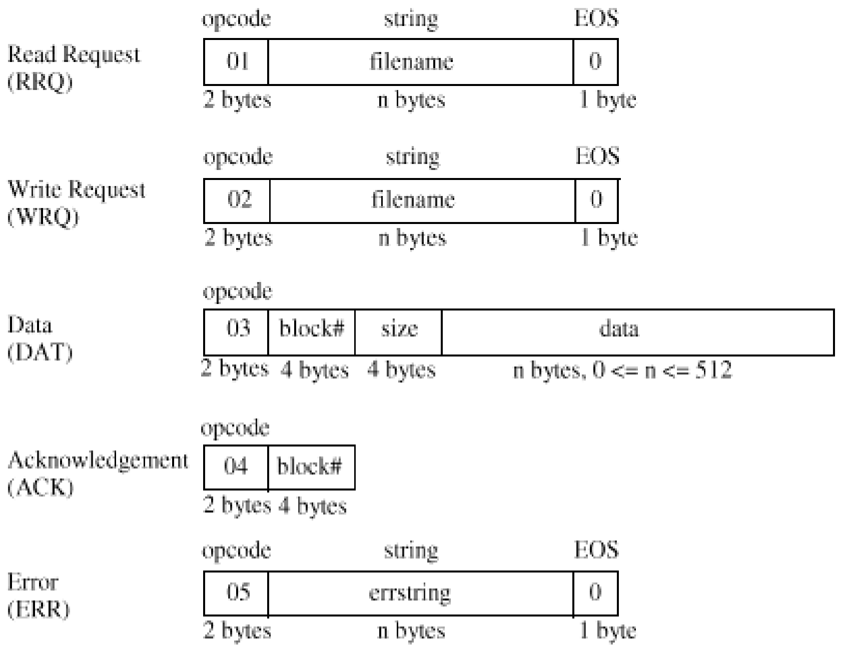
```
Client (receiver)                Server (sender)
------------------               -----------------
  connect              --->
                                   fork()
                       <---    DAT,block 1, "Hello
                           from Your name's fileserver"
  ACK,block 1          --->
  print message
  wait for user                         wait for packets.
```

If however the client does not reply with an ACK , the server should immediately drop the connection .The server should start waiting for the incoming packets once the initial packet is received and acknowledged by the client ,

PACKETS

Data packets are sent back and forth between the client and the server for them to communicate effectively. Following lines represent the format of the packets.





starts with a 2 byte opcode. Filenames and error messages are represented in ascii and terminated with a zero byte (EOS). data is sent in blocks of 512 bytes or less.

Some kind of check needs to be performed with opcode , block number , and block size parameters. If 16 bit integer value are to be written on a paper, it would be probably written with the most significant bit on the left and the least significant bit on the right:

(most significant) 1101111100110101 (least significant)

The ordering of the bits may be different for different machines. It depends on the machine being used.

For example, the most significant part may appear first like this:
byte 1 byte 2
(most significant) [11011111] [00110101] (least significant)
or, it may appear last like this:
byte 1 byte 2
(least significant) [00110101] [11011111] (most significant)
As we can see in the above lines , the first case has the most significant bit in the left most position this kind of arrangement is known as “big endian” .And if we look at the second case where the least significant bit appears in the left most position is known as “little endian”.Generally we never worry about such ordering but while sending data across the network we should be specific about constant ordering.And if however we ignore such practice the data between the machines are sent with different kind of ordering schemes.

SENDING AND RECEIVING FILES

Now when the connection is successfully established, the waits for the packets to arrive from the client .After the packets are received there opcodes are checked to acknowledge what to do next.The server must be able to recognize all five types of opcodes(RRQ, WRQ, ACK, DAT, and ERR).

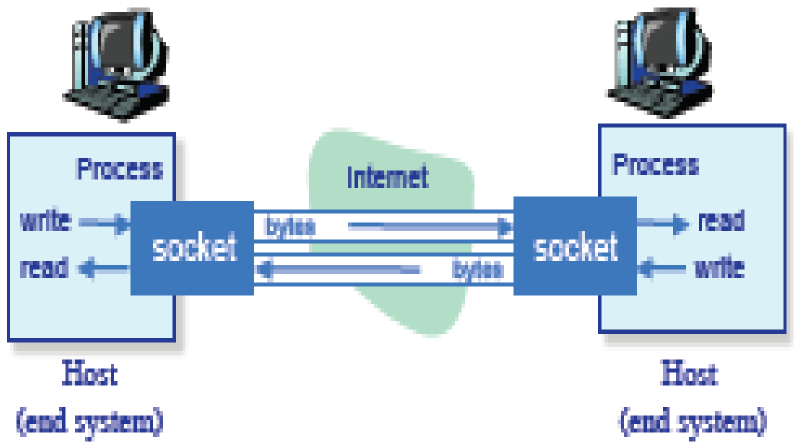
A RRQ packet is sent to the server in order to retrieve a file. And when the client needs to upload a file it sends a WRQ packet .The receive and send operations work as follows:

The client asks to receive a file from the server (download)

```
Client (receiver)                  Server (sender)
--------------------               -------------------
  RRQ                --->
                     <---          DAT, block #1
  ACK,block #1       --->
                     <---          DAT, block #2
  ACK,block #2       --->
                     <---          DAT, block #3
  ACK,block #3       --->
```

The client asks to send a file to the server (upload)

```
Client (sender)                  Server (receiver)
------------------               -----------------
 WRQ               --->
                   <---          ACK, block #0
 DAT, block #1     --->
                   <---          ACK, block #1
 DAT, block #2     --->
                   <---          ACK, block #2
 DAT, block #3     --->
                   <---          ACK, block #3
 .
```



If the packet size and the data being sent in each packet are equal it represents that more data will follow.And if it is less it means that packet is final one.If however during transmission any different type of packet is received it simply means that either there is a protocol error or any other error occurred.

The connection should be immediately terminated if the following conditions hold true:

- The number of bytes in a block exceeds the maximum block size (512 bytes).
- The expected block number of a data packet or ACK is incorrect.
- There is a protocol error (an unexpected packet type is received).

It is not necessary to assume that all the packets are valid one at any given time. As an example suppose server receives a WRQ packet instead of receiving a DAT packet then it should report a protocol error and immediately close the connection. Thus, the client handling loop might look roughly like this:

```
handle_client() {
   ...
   while (1) {
      get_packet()
      switch(opcode) {
        case OP_RRQ:
               if (strlen(filename) == 0)
send_directory();
          else send_file();
          break;
        case OP_WRQ:
          receive_file();
break;
        case OP_DRQ:
          send_directory();
          break;
        case OP_ERR:
```

```
        case OP_DAT:
        case OP_ACK:
          protocol_error()
          exit(0);
        default:
          unknown_opcode_error()
          exit(0)
      }
  } }
```

In the same way functions to send, receive and tranmit files directories would be programmed to recognize certain packets and report protocol errors in similar manner

SERVER SECURITY

Servers in this real world can be a lot misused. Precautions should be taken to prevent a server from crashing .Following are some measures which should be taken in order to prevent a system server from crashing:

- Clients should be authorized to access only the directory they want to
  access and no other directory outside that directory. For example a client should not be able to download the system password file or upload a file to a weird location.
- While uploading a file , the client should not be able to overwrite any existing file , in such situation the client should be sent a message that "file already exist".
- File names should not contain any kind of non – printable character.
- During any violation of protocol errors, the connection should be terminated.
- And we should never expect that client will behave properly while working with the server.

SOCKET CONNECTIONS

OPENING A SOCKET ON THE SERVER

```
 struct sockaddr_in servaddr;
 /* Create a socket */
 int fd = socket(AF_INET, SOCK_STREAM, 0);
 if (fd < 0) {
   printf("Unable to open socket!\n");
   exit(1);
 }
 /* Bind it to a specific port number (contained in port) */
 memset(&servaddr, 0, sizeof(servaddr));
```

```
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(port);
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

  if (bind(fd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
    printf("unable to bind to port %d\n", port);
    exit(1);
  }  /* Tell the OS to allow no more than 5 pending connections.
     Note : This is different from allowing 5 active client connections */
    listen(fd,5);
```

OPENING A SOCKET ON THE CLIENT

```
  int     fd;
  struct  sockaddr_in servaddr;
  struct  hostent *host;

  /* Create the socket */
  fd = socket(AF_INET, SOCK_STREAM, 0);
  if (fd < < 0) {
     printf("Unable to create socket!\n");
     exit(1);
  }
  /* Set its address :
        port      =  Server port number (e.g., 10000)
        hostname  =  Hostname (e.g., "classes.cs.uchicago.edu")
  */
  memset(&servaddr,0,sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(port);

  /* Look up the hostname try to turn it into an IP address */
  if ((host = gethostbyname(hostname)) == (NULL)) {
    printf("Unknown hostname : %s\n", hostname);
    exit(1);
  }
  memcpy(&servaddr.sin_addr,host->h_addr,host->h_length);

  /* Connect with the server */
  if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
    printf("Unable to connect with the server.\n");
    exit(1);
  }
```

ACCEPTING A CONNECTION ON THE SERVER

```
while(1) {    /* Server runs forever */
  /* Accept a client connection */
  struct sockaddr_in clientaddr;
  int len = sizeof(clientaddr);
  int clientfd = accept(fd, (struct sockaddr *) &clientaddr, &len);
  if (clientfd < 0) {
      if (errno != EINTR) {
         printf("Accept error!\n");
         printf("%s\n",strerror(errno));
```

```
        exit(1);
      }
  } else {
      /* Fork off the client handling process */
      if (!fork()) {
          /* I am the child */
          close(fd);
          /* Say where the connection came
from */
              printf("[%d] Received a
connection from %s, port %d\n", getpid(),

inet_ntoa(clientaddr.sin_addr),
ntohs(clientaddr.sin_port));

          /* Go off and talk to the client for
awhile */
              handle_client(clientfd);

          /* Done. Close the client connection
and exit. */
          close(clientfd);
          exit(0);



          /* Child process terminated */
      } else {
          /* I am the parent */
          /* Do nothing interesting.  Just go
back to the top of this
            loop and keeping listening for more
connections */
      }
      close(clientfd);
  }
}
```

SIGNALS AND TIMEOUTS

If the server remains idle and is not working for more than 60 seconds, in this case the server should terminate its connection.

For such implementation following function should be written:

```
void sig_alarm(int signo) {
  /* A timeout occurred. Deal with it */
  ...

}
```

Next, in server, the operating system needs be told about signal handler by putting the following statement somewhere in the code.
signal(SIGALRM,sig_alarm);

And now whenever we want to start the timer the statement as follows needs to be executed.

Alarm(60);

If you re-issue the alarm() function, the timer is reset to whatever new value is provided

In case if the alarm expires the system will interrupt the program to immediately sig alarm function. In this case, the alarm should be used to terminate idle client connections.

USEFULNESS

- Socket programming is based on client/server programming model. It is the socket on the server that enables the client to send request to the server and receive responses from the server. It is possible due to the socket that the client sends messages to the server and receive responses from the server to the client.
- The connection between client and server is due to the socket programming. Accepting connections on a server, Reading and writing data on a socket, Marshalling and unmarshalling data.
- Socket act as a user level interface on the network and it also acts as a basis for all internet applications.
- Socket programming is very important topic as port scanning, host and network fingerprinting, worms etc will be using sockets to do most of their job. All the above mentioned security topics make use of sockets
- Socket programming in some form or the other is also used in network security software such as vulnerability assessment toolkits, networking monitoring software etc.

CONCLUSION

In computer networking, one of the most fundamental technologies is a socket. By the help of sockets and using standard mechanisms which are built into network hardware and operating systems, application communicate with each other. Socket technologies are in existence since long as compared to network software which is relatively new to web phenomenon. Also, many software packages, like web browsers, instant messaging application and peer to peer file sharing system all rely on sockets.

REFERENCES

A.S. Tanenbaum, “Computer Networks”, PHI,4th Edition
W.Stallings, “Data and Computer Communication”, Macmillan Press.
Richard Stevens, UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI, Prentice Hall, 1998.
Gilligan, R. E., Thomson, S., Bound, J., and Stevens, W. R. 1999. "Basic Socket Interface Extensions for IPv6," RFC 2553.
Stevens, W. R., and Thomas, M. 1998. "Advanced Sockets API for IPv6," RFC 2292.
http://compnetworking.about.com/od/itinformationtechnology/l/aa083100a.htm
http://en.wikipedia.org/wiki/Internet_socket
http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/rzab6/rzab6soxoverview.htm
http://world.std.com/~swmcd/steven/perl/pm/socket/socket.html