Computer Networks (CS425)

Instructor: Dr. Dheeraj Sanghi

Prev | Next | Index

Unix Socket Programming (Contd..)

Client-Server Communication Overview

The analogy given below is often very useful in understanding many such networking concepts. The analogy is of a number of people in a room communicating with each other by way of talking. In a typical scenario, if A has to talk to B, then he would call out the name of B and only if B was listening would he respond. In case B responds, then one can say that a connection has been established. Henceforth until both of them desire to communicate, they can carry out their conversation.

A Client-Server architecture generally employed in networks is also very similar in concept. Each machine can act as a client or a server.

Server: It is normally defined which provides some sevices to the client programs. However, we will have a deeper look at the concept of a "service" in this respect later. The most important feature of a server is that it is a passive entiry, one that listens for request from the clients.

Client: It is the active entity of the architecture, one that generated this request to connect to a particular port number on a particular server

Communication takes the form of the client process sending a message over the network to the server process. The client process then waits for a reply message. When the server process gets the request, it performs the requested work and sends back a reply.The server that the client will try to connect to should be up and running before the client can be executed. In most of the cases, the servers runs continuously as a daemon.

There is a general misconception that servers necessarily provide some service and is therefore called a server. For example an e-mail client provides as much service as an mail server does. Actually the term service is not very well defined. So it would be better not to refer to the term at all. In fact servers can be programmed to do practically anything that a normal application can do. In brief, a server is just an entity that listens/waits for requests.

To send a request, the client needs to know the address of the server as well as the port number which has to be supplied to establish a connection. One option is to make the server choose a random number as a port number,

which will be somehow conveyed to the client. Subsequently the client will use this port number to send requests. This method has severe limitations as such information has to be communicated offline, the network connection not yet being established. A better option would be to ensure that the server runs on the same port number always and the client already has knowledge as to which port provides which service. Such a standardization already exists. The port numbers 0-1023 are reserved for the use of the superuser only. The list of the services and the ports can be found in the file /etc/services.

Connection Oriented vs Connectionless Communication

Connection Oriented Communication

Analogous to the telephone network.The sender requests for a communication (dial the number), the receiver gets an indication (the phone ring) the receiver accepts the connection (picks up the phone) and the sender receives the acknowledgment (the ring stops). The connection is established through a dedicated link provided for the communication. This type of communication is characterized by a high level of reliability in terms of the number and the sequence of bytes.

Connectionless Communication

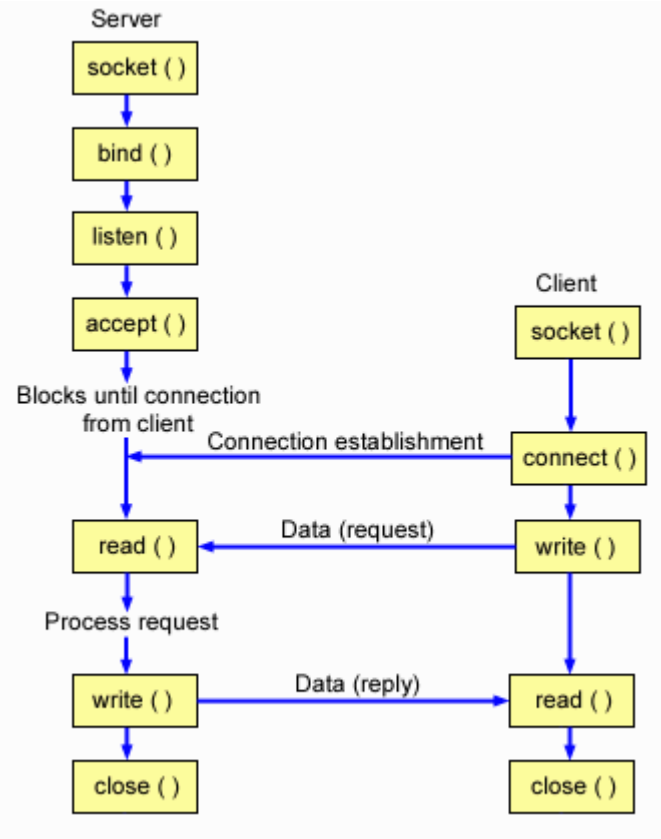
Analogous to the postal service. Packets(letters) are sent at a time to a particular destination. For greater reliability, the receiver may send an acknowledgement (a receipt for the registered letters).

Based on this two types of communication, two kinds of sockets are used:

- **stream sockets:** used for connection-oriented communication, when reliability in connection is desired.
- **datagram sockets:** used for connectionless communication, when reliability is not as much as an issue compared to the cost of providing that reliability. For eg. streaming audio/video is always send over such sockets so as to diminish network traffic.

Sequence of System Calls for Connection Oriented communication

The typical set of system calls on both the machines in a connection-oriented setup is shown in Figure below.



The sequence of system calls that have to be made in order to setup a connection is given below.

1. The *socket* system call is used to obtain a socket descriptor on both the client and the server. Both these calls need not be synchronous or related in the time at which they are called.The synopsis is given below:

   #include<sys/types.h>
   #include<sys/socket.h>
   int socket(int domain, int type, int protocol);

2. Both the client and the server 'bind' to a particular port on their machines using the *bind* system call. This function has to be called only after a socket has been created and has to be passed the socket descriptor returned by the *socket* call. Again this binding on both the machines need not be in any particular order. Moreover the binding procedure on the client is entirely optional. The *bind* system call requires the address family, the port number and the IP address. The address family is known to be AF_INET, the IP address of the client is already known to the operating system. All that remains is the port number. Of course the programmer can specify which port to bind to, but this is not necessary. The binding can be done on a random port as well and still everything would work fine. The way to make this happen is not to call *bind* at all. Alternatively *bind* can be called with the port number set to 0. This tells the operating system to assign a random port number to this socket. This way whenever the program tries to connect

to a remote machine through this socket, the operating system binds this socket to a random local port. This procedure as mentioned above is not applicable to a server, which has to listen at a standard predetermined port.

3. The next call has to be *listen* to be made on the server. The synopsis of the *listen* call is given below.

   #include<sys/socket.h>
   int listen(int skfd, int backlog);

   skfd is the socket descriptor of the socket on which the machine should start listening.
   backlog is the maximum length of the queue for accepting requests.

   The *connect* system call signifies that the server is willing to accept connections and thereby start communicating.

   Actually what happens is that in the TCP suite, there are certain messages that are sent to and fro and certain initializations have to be performed. Some finite amount of time is required to setup the resources and allocate memory for whatever data structures that will be needed. In this time if another request arrives at the same port, it has to wait in a queue. Now this queue cannot be arbitrarily large. After the queue reaches a particular size limit no more requests are accepted by the operating system. This size limit is precisely the *backlog* argument in the *listen* call and is something that the programmer can set. Today's processors are pretty speedy in their computations and memory allocations. So under normal circumstances the length of the queue never exceeds 2 or 3. Thus a *backlog* value of 2-3 would be fine, though the value typically used is around 5.Note that this call is different from the concept of "parallel" connections.The established connections are not counted in n. So, we may have 100 parallel connection running at a time when n=5.

4. The *connect* function is then called on the client with three arguments, namely the socket descriptor, the remote server address and the length of the address data structure. The synopsis of the function is as follows:

   #include<sys/socket.h>
   #include<netinet/in.h> /* only for AF_INET , or the INET Domain */

   int connect(int skfd, struct sockaddr* addr, int addrlen);

   This function initiates a connection on a socket.
   skfd is the same old socket descriptor.
   addr is again the same kind of structure as used in the *bind* system call. More often than not, we will be creating a structure of the type

sockaddr_in instead of *sockaddr* and filling it with appropriate data. Just while sending the pointer to that structure to the *connect* or even the *bind* system call, we cast it into a pointer to a *sockaddr* structure. The reason for doing all this is that the *sockaddr_in* is more convenient to use in case of INET domain applications. *addr* basically contains the port number and IP address of the server which the local machine wants to connect to. This call normally **blocks** until either the connection is established or is rejected.
addrlen is the length of the socket address structure, the pointer to which is the second argument.

5. The request generated by this *connect* call is processed by the remote server and is placed in an operating system buffer, waiting to be handed over to the application which will be calling the *accept* function. The *accept* call is the mechanism by which the networking program on the server receives that requests that have been accepted by the operating system. This synopsis of the *accept* system call is given below.

   #include<sys/socket.h>

   int accept(int skfd, struct sockaddr* addr, int addrlen);

   skfd is the socket descriptor of the socket on which the machine had performed a *listen* call and now desires to accept a request on that socket.
   addr is the address structure that will be filled in by the operating system by the port number and IP address of the client which has made this request. This *sockaddr* pointer can be type-casted to a *sockaddr_in* pointer for subsequent operations on it.
   addrlen is again the length of the socket address structure, the pointer to which is the second argument.

   This function *accept* extracts aconnection on the buffer of pending connections in the system, creates a new socket with the same properties as skfd, and returns a new file descriptor for the socket.

   In fact, such an architecture has been criticized to the extent that the applications do not have a say on what connections the operating system should accept. The system accepts all requests irrespective of which IP, port number they are coming from and which application they are for. All such packets are processed and sent to the respective applications, and it is then that the application can decide what to do with that request.
   The *accept* call is a blocking system call. In case there are requests present in the system buffer, they will be returned and in case there aren't any, the call simply blocks until one arrives.
   This new socket is made on the same port that is listening to new connections. It might sound a bit weird, but it is perfectly valid and the

new connection made is indeed a unique connection. Formally the definition of a *connection* is

connection: defined as a 4-tuple : (Local IP, Local port, Foreign IP, Foreign port)

For each connection at least one of these has to be unique. Therefore multiple connections on one port of the server, actually are different.

6. Finally when both *connect* and *accept* return the connection has been established.

7. The socket descriptors that are with the server and the client can now be used identically as a normal I/O descriptor. Both the *read* and the *write* calls can be performed on this socket descriptor. The *close* call can be performed on this descriptor to close the connection. Man pages on any UNIX type system will furnish further details about these generic I/O calls.

8. Variants of *read* and *write* also exist, which were specifically designed for networking applications. These are *recv* and *send*.

```
#include<sys/socket.h>

int recv(int skfd, void *buf, int buflen, int flags);
int send(int skfd, void *buf, int buflen, int flags);
```

Except for the *flags* argument the rest is identical to the arguments of the *read* and *write* calls. Possible values for the *flags* are:

used for	*macro for the flag*	*comment*
recv	**MSG_PEEK**	look at the message in the buffer but do not consider it read
send	**MSG_DONT_ROUTE**	send message only if the destination is on the same network, i.e. directly connected to the local machine.
recv & *send*	**MSG_OOB**	used for transferring data out of sequence, when some bytes in a stream might be more important than others.

9. To close a particular connection the *shutdown* call can also be used to

achieve greater flexibility.

#include<sys/socket.h>

int shutdown(int skfd, int how);

skfd is the socket descriptor of the socket which needs to be closed. *how* can be one of the following:

SHUT_RD	or 0	stop all read operations on this socket, but continue writing
SHUT_WR	or 1	stop all write operations on this socket, but keep receiving data
SHUT_RDWR	or 2	same as close

A port can be reused only if it has been closed completely.

Image References

- http://publib.boulder.ibm.com/iseries/v5r1/ic2924/info/rzab6/rxab6502.gif

---

back to top
Prev| Next | Index

.