Birla Institute of Technology &Science, Pilani
Computer Networks (CS F 303 / IS F 303)
Second Semester 2015-2016

Lab Session: 3

Aim: Introduction to Network Programming

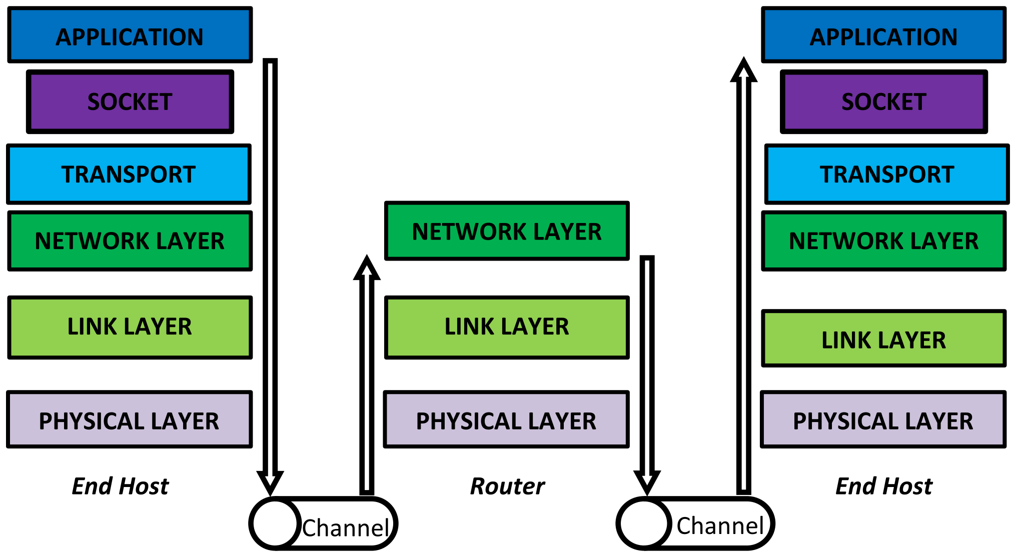
Points of Discussions:

- What is Network Programming?
- Introduction to Socket
- Socket
- Creating Socket
- Socket System Calls
- Some Important Functions

What is Network Programming?

Computer Network Programming is defined as the process of writing computer programs in order to establish the communication between processes over the Network. To write efficient Network Programming one must have a clear understanding about Socket and its Application Program Interface (API).

In a TCP/IP Layer architecture which contains five layers (Application, Transport, Network, Data Link, and Physical Layers) of communication, Application process sends messages to Transport layer via Sockets. Here, application process is controlled by the developer whereas transport layer (TCP, UDP) process is controlled by the operating system.



TCP (Transmission Control Protocol) is a set of rules (protocol) used along with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. While IP takes care of handling the actual delivery of the data, TCP takes care of keeping track of the individual units of data (called packets) that a message is divided into for efficient routing through the Internet.

TCP is known as a connection-oriented protocol, which means that a connection is established and maintained until such time as the message or messages to be exchanged by the application programs at each end have been exchanged. TCP is responsible for ensuring that a message is divided into the packets that IP manages and for reassembling the packets back into the complete message at the other end.

Examples: FTP, HTTP

UDP (User Datagram Protocol) is a connection-less communication protocol that offers a limited amount of service when messages are exchanged between computers in a network that uses the Internet Protocol (IP). Unlike TCP, however, UDP does not provide the service of dividing a message into packets (datagrams) and reassembling it at the other end. UDP doesn't provide sequencing of the packets that the data arrives in and the application program that uses UDP must be able to make sure that the entire message has arrived and is in the right order. Network applications which are time critical may prefer UDP to TCP.

Example: TFTP

Server

- passively waits for and responds to clients
- passive socket

Client

- initiates the communication
- must know the address and the port of the server
- active socket

API:

API expands as Application Programming Interface. A set of routines that an application uses to request and carry out lower-level services performed by a computer's operating system.

Introduction to Socket:

A socket is an endpoint/interface used by a process for bi-directional communication with a socket associated with another process to send/receive the messages. Sockets, introduced in Berkeley Unix (4.3 BSD, universally known as Berkeley Sockets), are a basic mechanism for Inter Process Communication (IPC) on a computer system, or on different computer systems connected by local or wide area networks. The first mainstream package - the Berkeley Socket Library is still widely in use on UNIX systems. Another very common API is the Windows Sockets (Winsock) library for Microsoft operating systems.

Sockets are uniquely identified by;

- an internet address
- an end-to-end protocol (TCP or UDP)
- a port number

To the kernel, a socket is an end point of communication and to an application a socket is a file descriptor that lets the application in a client server environment, read/write from/to the network.

Once Sockets are configured, application can;

- Pass data to socket for transmission on a network,
- Receive data from the socket (transmitted through the network by some other host)

Creating a socket

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

When you create a socket there are three main parameters that you have to specify:

- the domain
- the type
- the protocol

```
int socket(int domain, int type, int protocol);
```

The Domain parameter specifies a communications domain within which communication will take place, in our example the domain parameter was AF_INET that specify Internet Protocols.

The Type parameter specifies the semantics of communication, for TCP communication use Stream socket type (SOCK_STREAM) and for UDP use Datagram socket type (SOCK_DGRAM). The third type of socket, the so called *raw* socket, bypasses the library's built-in support for standard protocols like TCP and UDP. Raw sockets are used for custom low-level protocol development which may be used in routers.

Finally the protocol type, to communicate over the Internet, IP socket libraries uses the IP address to identify specific computers. Many parts of the Internet work with *naming services*, so that the users and socket programmers can work with computers by name (*e.g.*, "bits-pilani.ac.in") instead of by address (*e.g.*, 202.78.175.200). Stream and datagram sockets also use IP port numbers to distinguish multiple applications from each other. For example, Web browsers on the Internet know to use port 80 as the default for socket communications with Web servers.

We can see in /etc/protocols the number of ip, 0, which means default.

So our function now is:

```
s = socket (AF_INET, SOCK_STREAM, 0)
```

Where ‘s’ is the file descriptor returned by the socket function.

Socket System Calls:

bind System call

The bind system call assigns a name to an unnamed socket.

```
#include<sys/types.h>
#include<sys/socket.h>
int bind (int sockfd,struct sockaddr *myaddr, int addrlen);
```

The second argument is a protocol specific address & the third argument is the size of this address structure.

Uses of bind system call:

- Servers register their well-known address with the system. It tells the system “this is my address & any message received for this address is to be given to me. Both connection-oriented & connectionless servers need to this before accepting client request.
- A client can register a specific address for itself.
- A connectionless client needs to assure that the system assigns it some unique address, so that the other end has a valid return address to send its responses to.

Connect system call

A client process connects a socket descriptor following the socket system call to establish a connection with a server.

```
#include<sys/types.h>
#include<sys/socket.h>
int connect( int sockfd, struct sockaddr *servaddr, int
addrlen);
```

The connection typically causes these four elements of the association 5-tuple to be assigned: local_addr, local_process, foreign-addr & foreign-process.

The connect & bind system calls require only the pointer to the address structure & its size as arguments, not the protocol;

Listen system call

This system call is used by a connection-oriented server to indicate that it is willing to receive connections.

```
int listen(int sockfd , int backlog);
```

It is executed after both the socket & bind system calls & immediately before the accept system call. The backlog argument specifies how many connection requests can be queued by the system while it waits for the server to execute the system call. (Maximum is 5)

Accept system call

After a connection-oriented server executes the listen system call, an actual connection from some client process is waited for by having the server execute the accept system call.

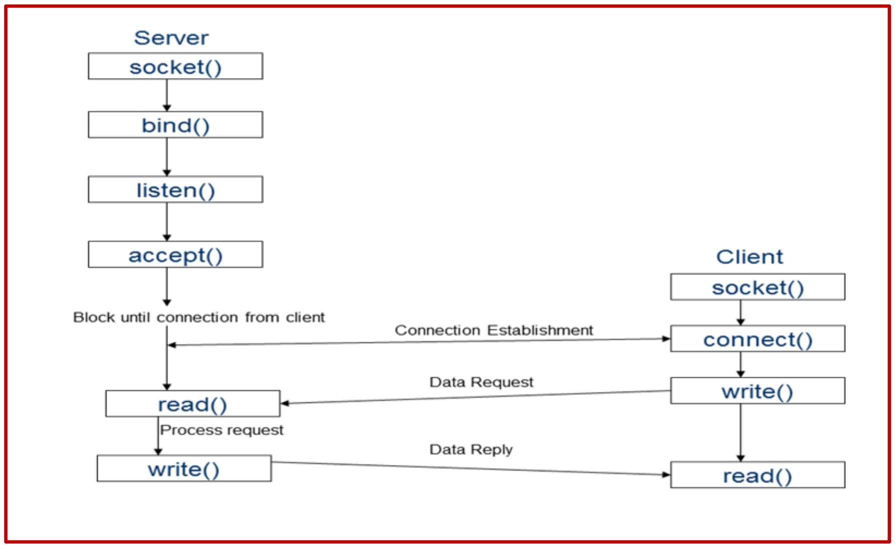
```
#include<sys/types.h>
#include<sys/socket.h>
int accept(int sockfd, struct sockaddr *peer, int *addrlen);
```

accept takes the first connection request on the queue & creates another socket with the same properties as sockfd. If there are no connection requests pending, this call blocks the caller until one arrives.
The peer & addrlen arguments are used to return the address of the connected peer process (the client). addrlen is called a value-result argument.

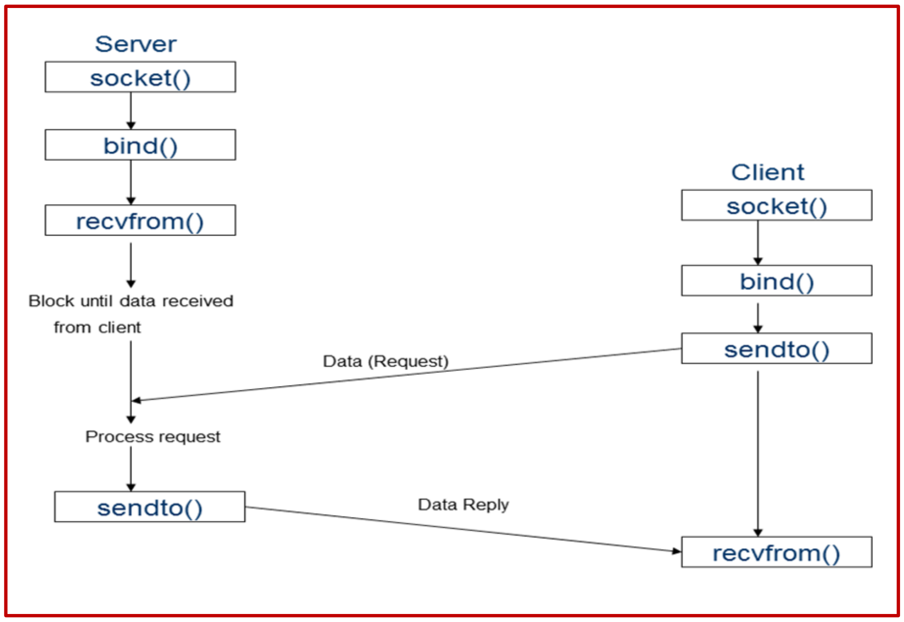
This system call returns up to three values:

- an integer return code that is either an error indication or a new socket descriptor
- the address of the client process(peer) &
- the size of this address.(addrlen)

Socket System Call for Connection-oriented Protocol



Socket System Call for Connection-less Protocol



<u>Some Important Functions:</u>

<u>gethostbyname linux</u>

This is the most important function to learn. This function accepts the name of the host that you want to resolve, and it returns a structure identifying it in various ways. The function synopsis is as follows:

```
#include <netdb.h>
extern int h_errno;
struct hostent *gethostbyname(const char *name);
```

The function gethostbyname() accepts one input argument that is a C string representing the hostname that you want to resolve into an address. The value returned is a pointer to the hostent structure if the call is successful. If the function fails, then a NULL pointer is returned, and the value of h_errno contains the reason for the failure.

ntohl function c

```
/*
 * ntohl linux
 * ntohl socket
 * network to host long long
 * network to host byte order
 */

#include <netinet/in.h>

unsigned long ntohl(unsigned long netlong);
```

Return value: The ntohl function returns the value supplied in the netlong parameter with the byte order reversed. If netlong is already in host byte order, then this function will reverse it. It is up to the application to determine if the byte order must be reversed.
Remarks: The ntohl function takes a 32-bit number in TCP/IP network byte order (the AF_INET or AF_INET6 address family) and returns a 32-bit number in host byte order.
The ntohl function can be used to convert an IPv4 address in network byte order to the IPv4 address in host byte order.
This function does not do any checking to determine if the netlong parameter is a valid IPv4 address.

struct sockaddr_in

Forming Internet (IPv4) Socket Addresses

The most commonly used address family under Linux is the AF_INET family. This gives a socket an IPv4 socket address to allow it to communicate with other hosts over a TCP/IP network. Include file that defines the structure sockaddr_in is defined by the C language statement:

```
#include <netinet/in.h>
Example
struct sockaddr_in {
     sa_family_t sin_family;    /* Address Family */
     uint16_t sin_port;         /* Port number */
     struct in_addr sin_addr;   /* Internet address */
     unsigned char sin_zero[8]; /* Pad bytes */
};
struct in_addr {
     uint32_ t s_ addr;         /* Internet address */
};
```

- The sin_family member occupies the same storage area that sa_family does in the generic socket definition. The value of sin_family is initialized to the value of AF_INET.
- The sin_port member defines the TCP/IP port number for the socket address. This value must be in network byte order (this will be elaborated upon later).
- The sin_addr member is defined as the structure in_addr, which holds the IP number in network byte order. If you examine the structure in_addr, you will see that it consists of one 32- bit unsigned integer.
- Finally, the remainder of the structure is padded to 16 bytes by the member sin_zero[8] for 8 bytes. This member does not require any initialization and is not used.

struct sockaddr

Because the BSD socket interface was developed before the ANSI C standard was adopted, there was no (void *) data pointer type to accept any structure address. Consequently, the BSD solution chosen was to define a generic address structure. The generic structure is defined by the C language statement

Example:

```
#include <sys/socket.h>
struct sockaddr {
     sa_family_t sa_family; /* Address Family */
     char sa_data [14];     /* Address data. */
};
```

Presently the data type sa_family_t is an unsigned short integer, which is two bytes in length under Linux. The total structure size is 16 bytes. The structure element sa_data[14] represents 14 remaining bytes of address information.

Note: Hard copies of the workbook will be provided separately.