



Introduction to the Network Programming

Dr. Rahul Banerjee

Associate Professor: CS&IS Group

Software & Development & Educational Technology Unit

Birla Institute of Technology & Science, Pilani (India)

E-mail: rahul@bits-pilani.ac.in

Home: <http://discovery.bits-pilani.ac.in/rahul/>

Network Programming Basics

Network Programming for different Software Architectures

- Mainframe / Mid-range <Mini / Super-mini ranges included> computing architectures
- PC-based <Workstations of all kinds included> File Sharing based computing architectures
- Peer-to-Peer computing architectures
- Client/Server based computing architectures <two/three-tier>

Network Programming through different methods:

- Serial Port Programming
- Parallel Port Programming
- NetBIOS Programming
- TCP/IP Programming
 - BSD Socket API based
 - System V TLI based
 - WinSock based
- RPC Programming

About Application Client / Server Processes



Definition of an Application Server Process:

- A process that provides any set of predefined services to one or more requesting clients is called a Server Process.

Types of Application Server Processes:

- Concurrent Server Process
 - » A process that simultaneously provides any set of predefined services to one or more requesting clients is called a Concurrent Server Process.
- Iterative Server Process
 - » A process that provides any set of predefined services to only one requesting client at any point of time is called an Iterative Server Process.

Definition of an Application Client Process:

- A process that solicits any specific service from any designated server is called a Client Process.

The IPv4 Class Scheme

In the IPv4, any address is 32-bit long and is represented in four parts of one byte each separated by decimal points or dots.

There exist two ways of looking at the IPv4 world:

- **Class-based view**
 - A,
 - B,
 - C,
 - D & E
- **Classless view**

The 32-bit address comprises of two parts:

- **Network address / identifier**
- **Host address / identifier**

More on IP Classes



In the class-based / classful version, the classes are designated based on the first few bits of the Network Address portion of the IP address.

For instance:

- **If the first bit in this field is 0 (zero), it is referred to as a Class-A IP address.**
- **If the first two bits in this field are 10 , it is referred to as a Class-B IP address.**
- **If the first three bits in this field are 110 , it is referred to as a Class-C IP address.**

Examples of IP Classes

Example of a Class-A address:

- **12.0.0.3**

Here, **12** is the Network Address part whereas **0.0.3** is the Host Address part.

(Technically, this means: Network Address is **12.0.0.0** and the Host Address is **0.0.0.3**.)

Example of a Class-B address:

- **180.16.0.1**

Example of a Class-C address:

- **192.12.7.8**



IPv4 Class-range

Class-A Address Range:

1.0.0.0 - 127.255.255.255

Class-B Address Range:

128.0.0.0 - 191.255.255.255

Class-C Address Range:

192.0.0.0 - 223.255.255.255

Class-D Address Range:

224.0.0.0 - 239.255.255.255

Class-E Address Range:

240.0.0.0 - 247.255.255.255

The IPv4 Header Structure

0

31

Ver.	IHL	Type of Service	Total Length
Identification		Flags	Fragment Offset
TTL	Protocol Type	Header Checksum	
Source Address (32-bit)			
Destination Address (32-bit)			
Options+Padding			

The TCP Header Structure



Source Port Number (16-bit)	Destination Port Number (16-bit)
Sequence Number (32-bit)	
Acknowledgement Number (32-bit)	
HLEN + Reserved + Code	Sliding Window
TCP Checksum (16-bit)	Urgent Pointer (16-bit)
Options (if present) + Padding (if needed)	
Payload Data	

The UDP Header Structure



Source Port Number (16-bit)	Destination Port Number (16-bit)
Message Length (16-bit)	UDP Checksum (16-bit: Optional)
Payload Data	

About the TCP & UDP Ports



TCP and UDP Ports are 16-bit numbers.

They are of three types:

- **Well-known Ports (0-1023: Controlled by the IANA),**
- **Registered Ports (1024-49159) and**
- **Ephemeral / Dynamic Ports (49152-65535). (RFC 1700 shows a list suggested initially)**

FTP over TCP uses 21 whereas TFTP over UDP uses 69 for instance.

X-Windows Server uses 6000-6063 Registered Ports.

For BSD, the scheme is as under:

- **WKP: 1-1023**
- **Reserved Ports: 1024-5000**
- **Severs: 5001-65535 (no privilege)**

For Solaris, these ranges vary again!

Inside the TCP

- **TCP stands for Transmission Control Protocol.** (RFC 793 by John Postel)
- It offers a Connection-oriented Transport Service.
- **Assumes the underlying IP-subnet as unreliable and therefore takes care of reliability, flow control and reordering of data units as per requirement.**
- Sends data to the IP Layer in MSS-sized blocks or smaller, after prefixing a TCP header to each such segment.
- **Default value of the MSS is 536, in case the peer at the other end does not specify a smaller value to be used with it.**
- MSS is normally of lesser than or equal to the size of the MTU (**for IPv4: 40, for IPv6: 60**).



Inside the TCP (Contd.)

- TCP requires that a TCP Client establishes a Full-Duplex connection with a TCP Server (before any real data could be exchanged between them). After the data exchange is over, this connection has to be explicitly Terminated.
- As the TCP provides a reliable service, it expects an ACK to be received for the data transmitted by an TCP-entity (Client or Server).
- If the ACK does not arrive within a Time-out period, it retransmits the data and waits for a longer period of time to receive an ACK.
- Even if after a certain number of such attempts the data cannot be successfully transmitted, it gives up further attempts and informs the Application Layer. (Intermediate failures are not reported to the Application, however!)

Inside the TCP (Contd.)

- The maximum period for such retransmission-attempts and associated wait-periods for a single data unit, put together, may be anywhere between 4 Minutes to 10 Minutes, depending upon the TCP implementation and Stack Configuration.
- Round-Trip Time (RTT) is automatically, dynamically, computed between a Client-Server pair by a routine internal to the TCP implementation.
- RTTs are always more for WANs than for LANs.
- TCP recognizes byte-boundaries and is thus a byte-stream-oriented protocol.
- As it provides each of its Segments a Serial Number, reordering, rejection of duplicate segments etc. becomes possible.
- It uses Sliding Window Protocol for the purpose of data transmission / reception / flow-control.

The 3-Way Handshake in TCP

- TCP requires a Three-Way Handshake for the Connection-Establishment.
- This is called so since a minimum of three data-units need to be exchanged between a TCP Client and a TCP Server for establishing a TCP connection.
- These packets may be **SYN-I-Seq-No (C-to-S)**, **SYN-Ini-Seq-No (S-to-C)** on which **ACK-I+1-Seq-No** piggybacks (S-to-C) and lastly, **ACK-Ini+1-Seq-No (C-to-S)**.
- SYN stands for Synchronize segment. It takes just 1-byte of Sequence Number Space.
- In a similar way, Connection-Termination takes four data-units. It takes place using **FIN** (Final Segment) and associated **ACK**. Both sides send one FIN and one ACK to each-other, in this case.
- SYN contains TCP options of **MSS**, **Sliding Window Scaling** (Left-Shifting by 0 to 14 bits allows window-sizes of 64K to 1 GB) [RFC 1323 by Jacobson et al], **Timestamp** (for High-speed connections) etc.

The TCP/IP Access Points



- In a TCP/IP network, services offered by any layer can only be used through parameter / data passing through the various Service Access Points (SAPs) located on Layer-boundaries.
- TSAPs and NSAPs are two major examples of SAPs located at the AL-TCP/UDP Layer-boundary and TCP/UDP-IP Layer-boundary respectively.
 - A typical example of an NSAP is an IP address.
 - A typical example of a TSAP is an IP address + a 16-bit integer called as Port Number.
- Port Numbers permit unique identification of several simultaneous processes using TCP / UDP.
- IETF's RFC 1700 lists select IANA-suggested Port Number Assignments.

Network Programming in Microsoft Windows environments

Methods of Network Programming in MS-Windows

- Windows Network Programming support exists for:
 - TCP/IP (IETF's)
 - Named Pipe <FIFO>, <server creates a pipe and awaits, allowing bi-directional data-flow>
 - Mailslots <uni-directional guaranteed or non-guaranteed data-flow, uses broadcast datagrams, can be created by any process>
 - IPX/SPX (Novell's Netware is based on this stack)
 - NetBEAU (NETBIOS Extended User Interface) <for PC-based networking, exposes interfaces for reading / writing data, can be used for C/S or P2P, connection-oriented or connection-less way>
 - AppleTalk (Apple's)



- Methods available:
 - **Windows NT Remote Procedure Call (RPC)** <uses a collection of libraries / tools with a focus on the Application Procedure alone, targeting execution on a remote machine, independent of transport protocols>
 - **Windows Socket (WinSock)** <uses Windows Socket interface, extension of BSD Sockets, allows bi-directional data-flow, upon completion of asynchronous I/O, exports functions which transmit messages to the relevant applications: feature fits within the message-driven interface model of MS-Windows>
 - **WinNet API** <for over-the-MS-Windows-network sharing of file-server / printer etc.>

Network Programming in Linux / UNIX and similar environments



The Sockets and Socket Pairs

- Socket is another name for the TSAP discussed above. Thus, it comprises of IP Address+Port Number. Sockets are often created by the `socket()` system call.
- A Socket Pair refers to a set of socket endpoints at the two communicating ends of a TCP/IP network. Typically, a Socket-Pair, for TCP as well as UDP, comprises of four components:
 - IP Address-1 & Port Address/Number-1
 - IP Address-2 & Port Address/Number-2
- In case of TCP, a Socket-pair typically uniquely represents a TCP-connection (logical).

Creating a Socket

- Every network protocol has its own definition of *Network Address*.
- In C, a protocol implementation provides a *struct sockaddr* as the elementary form of a *Network Address*.
- A sample definition of *struct sockaddr*
 - » `#include <sys/socket.h>`
 - » `struct sockaddr {`
 - » `unsigned short sa_family;`
 - » `char sa_data [MAXSOCKADDRDATA]`
 - » `}`

Creating a Socket ...

- In UNIX and Linux, Sockets are created by the `socket()` system call.
- This call returns a *file descriptor* for the Socket that is yet to be initialized.
- Then the socket is initialized by binding it to a particular protocol and address using the `bind()` system call.
 - `#include <sys/socket.h>`
 - `int socket (int domain, int type, int protocol);`
*Here,
the parameter domain specifies the PF;
parameter type usually specifies either of SOCK_STREAM or SOCK_DGRAM;
parameter protocol specifies the protocol to be used (0=> default protocol associated).*
 - `int bind (int sock, struct sockaddr * my_addr, int addrlen);`
- Here, the parameter `sock` is the socket-in-question; parameter `sockaddr` is the address of protocol; parameter `addrlen` is the length of the address for the local end-point.

Creating a Socket ...



Next, *listen()* system call is executed for informing the system that the process is now ready to allow other processes establish a connection to this socket at the specified end-point.

This step does not really establish a connection by itself, however!

Now, the *accept ()* system call is made for accepting the connection requests.

accept () is a blocking call as it blocks until a process requests a connection. In case, the socket has been marked as 'non-blocking' by the *fcntl()* call, *accept()* would return an error if no process is requesting it for a connection.

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

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

Here,

the parameter **sock** is the socket-in-question;

parameter **backlog** is the number of connection requests that may be pending on the socket before any further connection requests are explicitly refused.

```
int accept (int sock, struct sockaddr * addr, int * addrlen);
```

The *select ()* system call can also be made for determining if any connection request is currently pending to a socket.

Similarly, a Client attempts to connect to a Server by creating a socket, binding it to an address (optionally) and making the *connect()* call to the Server at the known address.

A Glimpse of Address and Protocol Families for Various Stacks



Address Families:

Unix / Linux Domain:	AF_UNIX
TCP/IPv4 Domain:	AF_INET
TCP/IPv6 Domain:	AF_INET6
Novell NetWare Domain:	AF_IPX
AppleTalk Domain:	AF_APPLETALK

Protocol Families:

Unix / Linux Domain:	PF_UNIX
TCP/IPv4 Domain:	PF_INET
TCP/IPv6 Domain:	PF_INET6
Novell NetWare Domain:	PF_IPX
AppleTalk Domain:	PF_APPLETALK

The IP Socket Address Structure

```
struct in_addr {  
    in_addr_t  s_addr;          /* Big Endian 32-bit IP address */  
};  
  
struct sockaddr_in {  
    uint8_t  sin_len;          /* 16-byte structure length */  
    sa_family_t  sin_family;  /* AF_INET */  
    in_port_t  sin_port;      /* 16-bit Big Endian TL Port */  
    struct in_addr sin_addr;  /* 32-bit Big Endian IP address*/  
    char  sin_zero [8];  
    }  
  
/* This is defined in netinet/in.h file. */
```



On the Byte-ordering and Byte-manipulation Functions

The term **Byte-ordering** refers to the manner in which a multi-byte / multi-octet number / field is stored in **Lower-order** (i.e. starting address) and **Higher-order Memory Addresses**.

If the Lower-order Byte is stored in the Starting / Lower-order Memory Location and Higher-order Memory Location, then it is called the Little Endian scheme.

If the storage is carried out in exactly opposite manner, it is called the Big Endian scheme.

Different manufacturers may choose any one of these Byte-ordering Schemes. For instance, traditionally, Motorola processors have followed the Big Endian scheme whereas the Intel processors have used the Little Endian scheme.

The Network Byte-order is always Big Endian, by convention whereas the Host Byte-order may be of either type.

None of these two schemes is superior or inferior to the other since they merely represent two possible ways in which Lower and Higher Order Bytes are stored into and retrieved from Memory or transferred over a network link.



On the Byte-ordering and Byte-manipulation Functions (Contd.)

Over a network, between any two nodes, data transfer takes place in Big Endian manner.

There are standard function prototypes for 'Host-Byte-order to Network-Byte-order' conversion and 'Network-Byte-order to Host-Byte-order' conversion. These functions are defined in the `netinet/in.h` header file in the systems supporting Socket Programming in C and its variations. Their usage has been shown below:

<code>#include <netinet/in.h></code>	
<code>uint16_t htons (uint16_t</code>	<i>16-bit-host-address</i>)
<code>uint32_t htonl (uint32_t</code>	<i>32-bit-host-address</i>)
<code>uint16_t ntohs (uint16_t</code>	<i>16-bit-network-address</i>)
<code>uint32_t ntohl (uint32_t</code>	<i>32-bit-network-address</i>)

There exist two categories of functions those are capable of operating on multi-byte / multi-octet numbers / fields.

The first category has BSD functions like `bcopy`, `bzero` etc. Functions belonging to this category begin with 'b' (b=> byte).

The second category has ANSI-C functions like `memset`, `memcpy` etc. Functions belonging to this category begin with 'mem' (mem=> memory). Both types are defined in the header files `strings.h` and `string.h` respectively.

On the Functions used for Address Conversions



A few functions used for IPv4 Address conversion from / to ASCII string and Binary strings expressed in the Network Byte-order have been defined in `arpa/inet.h` header file. These include: `inet_pton`, `inet_ntop`, `inet_aton`, `inet_ntoa`, `inet_addr` etc.

Here, `inet`=> internet, `a`=> ASCII, `n`=> network, `addr`=> address, `ntoa` and `aton` refer to network to ASCII and ASCII to network respectively.

The first two of these functions can handle IPv4 as well as IPv6 addresses and are therefore commonly used these days since majority of implementations attempt to offer dual-stack compatibility.

It is advisable to avoid use of `inet_addr`, `inet_aton` and `inet_ntoa` as these have known problems as well as are considered deprecated. In near future, they may not be supported.

Using the preferred functions:

```
#include <arpa/inet.h>
```

```
int inet_pton ( int <addr-family>, const char *<strptr>, void  
                *<addrptr> );
```

```
const char *inet_ntop ( <addr-family>, const void *<addrptr>, char  
                        *<strptr>, size_t <len> );
```

```
/* Here, addr-family may be AF_INET or AF_INET6 whichever needed. */
```

On the Functions used for Address Conversions (Contd.)



As may be noticed in the syntax, `inet_ntop` requires *char *`<strptr>`*, this in turn requires to get the binary address that is stored as a part of a Socket Address Structure. This would mean some degree of protocol dependence of the function to be written since the programmer must know the said structure beforehand.

```
const char *inet_ntop ( <addr-family>, const void *<addrptr>,  
char *<strptr>, size_t <len> );
```

This problem may be easily taken care of if the programmer writes his own function that extracts this data and its presentation format and supplies it to the standard function.

There are numerous other similar situations in which such custom-built functions, if written by the programmer, increase the level of protocol independence of the resulting code. This, however, comes with its own overheads and may not be a preferable approach where speed and efficiency are the primary requirements.

On the Functions used for I/O Operations on Stream Sockets and POSIX-compliant Status Information



Two points are important whenever a programmer wishes to perform various read and write operations on Stream Sockets. (*Stream here refers to Byte Stream as in case of TCP Sockets.*)

- A read operation or a write operation, even without an error, may read or write lesser number of bytes than explicitly mentioned particularly when a Buffer gets full. This does not mean that inaccurate result needs to be accepted; instead, it simply requires that the function is called again in order to read or write the rest of the bytes subsequently.
- This may happen during all read as well as all non-blocking write operations.

The `fstat` function has been used traditionally for getting the information about any specified descriptor. This function, along with others has a prototype in the `sys/stat.h` header file.

Another function `isfdtype` has been more common in use of late. This is used as follows:

```
#include <sys/stat.h>  
int isfdtype (int <sock-fd>, int <fd-type>);  
Here, fd=> file descriptor.
```

The TCP/IP Tips

Syntax of the **socket** function that is required to create an end-point has been discussed earlier as:

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

```
int socket (int <add-family or proto-family>, int <socket-type>, int <protocol>);
```

Unlike the **Server**, described earlier, the **Client** need not invoke the **bind** function call. In case of TCP over IPv4, this function, as discussed before, assigns a **32-bit+16-bit address** to a Socket created by the **socket** call. **Server** processes normally prefer to be assigned a standard-convention-based port called **Well Known Port**. Whenever used, syntax of the function call is is:

```
int bind (int <sock-fd>, const struct sockaddr * <assigned-addr>, socklen_t <addr-len>);
```

Syntax of the **connect** function that is needed by a Client to initiate a connection-request to a Server is:

```
int connect (int <sock-fd>, const struct sockaddr * <server-addr>, socklen_t <addr-len>);
```

The TCP/IP Tips (Contd.)



Just like bind, another function that is never invoked by the Client is the listen function. Syntax of the listen function that is required to make an unconnected Active Socket of a Server process behave as a Passive Socket entity has been discussed earlier as:

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

```
int listen (int <sock-fd>, int <composite-connect-queue-len>);
```

Where, composite-connect-queue-len=> sum of complete+incomplete connection queues.

Like bind and listen, another function that is called only by the Server is the function accept. As shown earlier, its syntax is:

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

```
int accept (int <sock-fd>, struct sockaddr *<client-addr>,  
socklen_t *<addr-len>);
```

Tips on Programming a Concurrent Server



Under UNIX and Linux environments, typically, the only available way to create a new process is by invoking the fork function that creates a Child Process. This function returns twice per invocation – once in the Parent Process that invoked it and thereafter in the created Child process. The first time it returns a value to the Parent that is actually the PID of the Child and second time it returns a value of zero to the Child.

Syntax of its usage is:

```
#include <unistd.h>  
pid_t fork(void);
```

Interestingly, this function can be used for a different reason altogether, i.e. it can be used in association with the exec function whenever it is desired that a given process must invoke another process. *In this case, first a replica / child process is created by the fork call which then replaces itself with the process named as an argument to the exec function call.* The exec has six variations (?).

Syntax of use of exec is:

```
int exec?(const char * <file-or-path-name>, <argument-list>, <...>);
```

Tips on Programming a Concurrent Server (Contd.)



Steps involved in writing a simple Concurrent Server are:

- Create a routine that creates a Socket, binds it to a well known address, changes it to listen mode and waits for a connection request at this address from a Client. (socket+bind+listen thus form Step One!)
- Once a connection request is received, based on the queuing status, this routine has to ensure that following events occur:
 - The accept call returns,
 - The fork call is used to spawn a Child,
 - Let the Child close the listening Socket, service the Client etc. While in the mean time, the Parent returns to its passive ('listening') status and awaits the next request.
 - Once the Child has serviced the Client, it closes all open descriptors and exits,
- Finally, the routine must ensure that the Parent closes the connected Socket. This completes the cycle.
- Syntax for a UNIX / Linux close is:

```
#include <unistd.h>  
int close (int <sock-fd>);
```



A Few More Functions of Importance

Certain other functions of relevance include:
ioctl, fcntl, recvfrom, sendto, signal,
select, poll, shutdown, pselect,
getsockname, getpeername, getsockopt,
setsockopt,
gethostbyname, gethostbyname2, gethostbyaddr,
uname,
gethostname, getservbyname, getservbyport,
getaddrinfo, getnameinfo
gai_strerror, freeaddrinfo

Summary



In TCP/IP setup, services specific to any layer can only be accessed through Service Access Points located at the layer-boundaries.

A TSAP is an IP address+ a 16-bit Port Number.

Typically, a Socket-Pair, for TCP as well as UDP, comprises of Server IP Address, Port Address / Number of the Server, Client IP Address, Port Address / Number of the Client.

In case of TCP, a Socket-pair typically uniquely represents a TCP-connection.

Each network protocol may have its own definition of *Network Address*.

The Network Byte-order is always Big Endian, by convention whereas the Host Byte-order may be of either type.

functions used for IPv4 Address conversion from / to ASCII string and Binary strings expressed in the Network Byte-order have been defined in *arpa/inet.h* header file. These include: `inet_pton`, `inet_ntop`, `inet_aton`, `inet_ntoa`, `inet_addr` etc.

Summary (Contd.)



Under UNIX and Linux environments, typically, the only available way to create a new process is by invoking the fork function that creates a Child Process. This function returns twice per invocation – once in the Parent Process that invoked it and thereafter in the created Child process.

The function fork can be used in association with the exec function whenever it is desired that a given process must invoke another process.

Commonly used functions include socket, bind, listen, connect, accept, close, getsockname, getpeername, select, poll, shutdown, pselect, getsockopt, setsockopt, ioctl, fcntl, recvfrom, sendto, signal, gethostbyname, gethostbyname2, gethostbyaddr, uname, gethostname, getservbyname, getservbyport, getaddrinfo, gai_strerror, freeaddrinfo, getnameinfo.

If you know how to write a Client-Server pair for TCP-based application, we may easily identify the distinctly simple alterations that we may need to make in this code if we ever need to write a code for UDP-based application.

Summary of a Quick tour to basics



Every network protocol has its own definition of *Network Address*.

In C, a protocol implementation provides a

struct sockaddr
as the elementary form of a *Network Address*.

A sample definition of *struct sockaddr*

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

```
struct sockaddr {  
    unsigned short  
    sa_family;  
    char sa_data  
    [MAXSOCKADDRD  
    ATA]  
}
```

Summary of a Quick tour to basics



In Linux, sockets are created by the `socket()` system call.

This call returns a *file descriptor* for the socket that is yet to be initialized.

Then the socket is initialized by binding it to a particular protocol and address using the `bind()` system call.

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

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

Here,

the parameter `domain` specifies the PF; parameter `type` usually specifies either of `SOCK_STREAM` or `SOCK_DGRAM`; parameter `protocol` specifies the protocol to be used (0=> default protocol associated).

```
int bind (int sock, struct sockaddr * my_addr, int addrlen);
```

Here,

the parameter `sock` is the socket-in-question;

parameter `sockaddr` is the address of protocol;

parameter `addrlen` is the length of the address for the local end-point.

Summary of a Quick tour to basics



Next, *listen()* system call is executed for informing the system that the process is now ready to allow other processes establish a connection to this socket at the specified end-point.

This step does not really establish a connection by itself, however!

Now, the *accept()* system call is made for accepting the connection requests.

accept() is a blocking call as it blocks until a process requests a connection. In case, the socket has been marked as 'non-blocking' by the *fcntl()* call, *accept()* would return an error if no process is requesting it for a connection.

A Client attempts to connect to a Server by creating a socket, binding it to an address (optionally) and making the *connect()* call to the Server at the known address.

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

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

Here,

the parameter **sock** is the socket-in-question;

parameter **backlog** is the number of connection requests that may be pending on the socket before any further connection requests are explicitly refused.

```
int accept (int sock, struct  
sockaddr * addr, int * addrlen);
```

The *select()* system call can also be made for determining if any connection request is currently pending to a socket.

References



1. **W. R. Stevens: UNIX Network Programming, Vols. 1-2, ISE, Addison-Wesley, Mass.**
2. **Alok K. Sinha: Network Programming in Windows NT, Addison-Wesley, Mass.**
3. **W. R. Stevens: TCP/IP, Vol. 1, Addison-Wesley, Mass.**
4. **D. Comer: Internetworking with TCP / IP , Vol.-1, PHI.**
5. **D. Comer & D. L. Stevens: Internetworking with TCP /IP, Vol.. 2-3, PHI.**
6. **M. K. Johnson and E. W. Troan: Linux Application Development, ISE, Addison-Wesley.**
7. **Rahul Banerjee: Lecture Notes in Computer Networking, BITS-Pilani.**

Some interesting commands

- Linux / Unix:
 - ps
 - Ipconfig
 - Netstat
 - ping
- Windows:
 - tlist /t
 - Qslice
 - Ifconfig
 - Netstat
 - ping

