MODULE 39
NETWORK PROGRAMMING
SOCKET PART I

My Training Period: hours

Note: Program examples if any, compiled using `gcc` on Fedora 3 machine with several update, as normal user. The Fedora machine used for the testing having the "No Stack Execute" disabled and the SELinux set to default configuration.

Abilities

- Able to understand the basic of client-server model.
- Able to understand the TCP/IP suite/stack/layer.
- Able to understand important protocols such as TCP and UDP.
- Able to understand and use the Unix/Linux C language socket APIs.
- Able to understand and implement several simple Client and server basic designs.

This Tutorial introduces a network programming using sockets. Some of the information is implementation specific but all the program examples run on Fedora 3 and compiled using gcc. The following are topics that will be covered.

- Some Background Story.
- The Client-Server Model.
- Concurrent Processing.
- Programming Interface.
- The Socket Interface.
- Client Design.
- Example Clients.
- Server Design.
- Iterative, Connectionless Servers (UDP).
- Iterative, Connection-Oriented Servers (TCP).
- Concurrent, Connection-Oriented Servers (TCP).
- Single-Process, Concurrent Servers (TCP).
- Multi protocol Servers.
- Multi service Servers.
- Managing Server Concurrency.
- Client Concurrency.

Some Background Story

- This background story tries to introduce the terms used in network programming and also to give you the big picture.
- The following figure is a typical physical network devices connection.

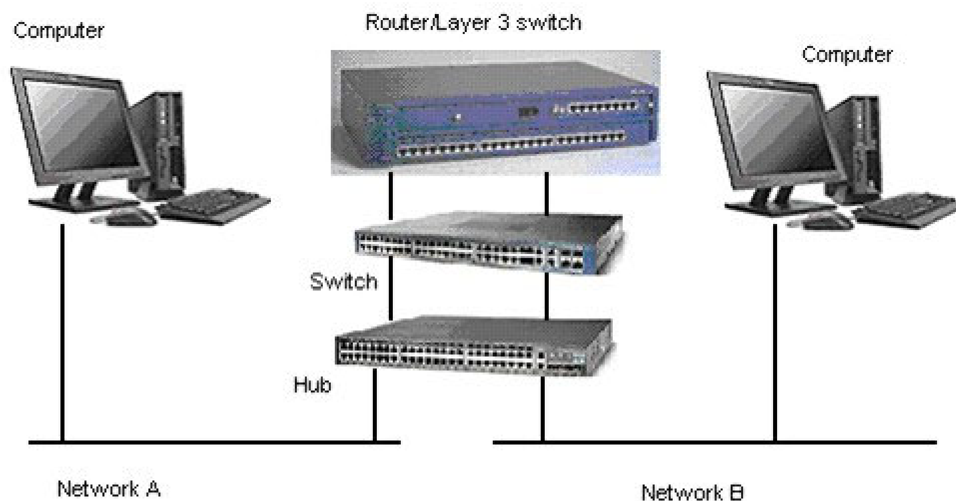

Figure 1

- Using a simple network shown in the above Figure, let trace the data stream flow from Network A to Network B, by assuming that Network A is company A's network and Network B is company B's network.
- Physically, the flow of the data stream is from a computer in Network A (source) will go through the hub, switch and router.
- Then the stream travel through the carrier such as Public Switch Telephone Network (PSTN) and leased line (copper, fiber or wireless – satellite) and finally reach Network B's router, go through the switch, hub and finally reach at the computer in company B (destination).
- And from the previous network devices layout, the OSI (Open System Interconnection) 7 layer stack mapping is shown below.

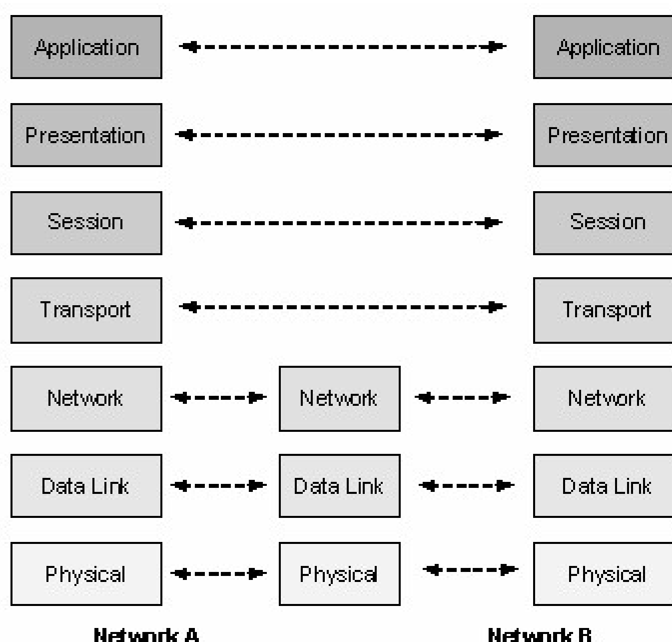

Figure 2

- From the Application layer of a computer at company A go downward the layer until the Physical (medium such as Cat 5 cable) layer, then exit Network A through the Network (router) layer in the middle of the diagram.
- After traveling through the carrier, reaches at the Network (router) layer of company B, travels through the Physical layer, goes upward until reaching at the Application layer of the computer at company B. Actually, at company B (the destination), the data flows through the network devices in the reverse manner compared to what happened at company A (the source).
- In contrast to TCP/IP, the OSI approach started from a clean slate and defined standards, adhering tightly to their own model, using a formal committee process without requiring implementations.
- Internet protocols use a less formal but more practical engineering approach, where anybody can propose and comment on Request For Comment (RFC) documents, and implementations are required to verify feasibility.
- The OSI protocols developed slowly, and because running the full protocol stack is resource intensive, they have not been widely deployed, especially in the desktop and small computer market.
- In the meantime, TCP/IP and the internet were developing rapidly, with deployment occurring at a very high rate, which is why the TCP/IP suite becomes a de facto standard.
- The OSI layer and their brief functionalities are listed in the following Table.

OSI Layer	Function provided
Application	Network application such as file transfer and terminal emulation
Presentation	Data formatting and encryption.
Session	Establishment and maintenance of sessions.
Transport	Provision for end-to-end reliable and unreliable delivery.
Network	Delivery of packets of information, which includes routing.
Data Link	Transfer of units of information, framing and error checking.
Physical	Transmission of binary data of a medium.

Table 1

- In the practical implementation, the standard used is based on TCP/IP stack. This TCP/IP stack is a de facto standard, not a pure standard but it is widely used and adopted.
- The equivalent or mapping of the OSI and TCP/IP stack is shown below. It is divided into 4 layers. The Session, Presentation and Application layers of OSI have been combined into one layer, Application layer.
- Physical and data link layers also become one layer. Different books or documentations might use different terms, but the 4 layers of TCP/IP are usually referred.

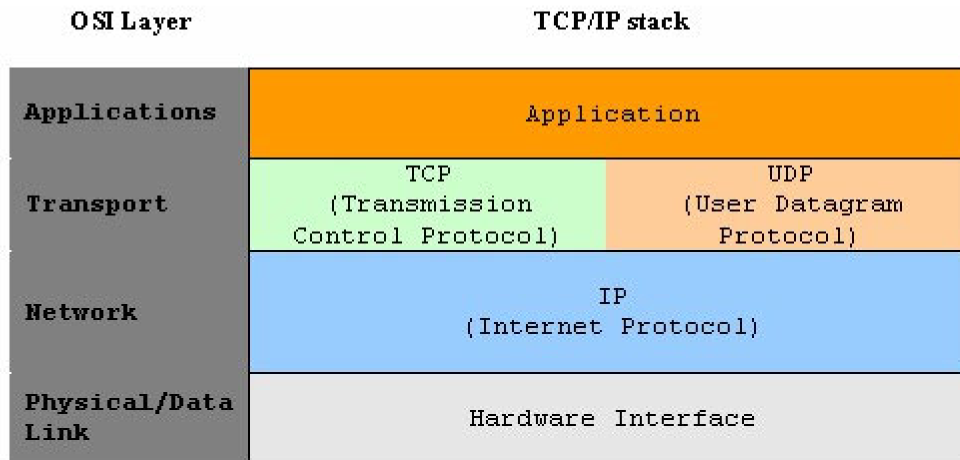

Figure 3

- In this Tutorial we will concentrate more on the Transport and Network layer of the TCP/IP stack.
- More detail TCP/IP stack with typical applications is shown below.

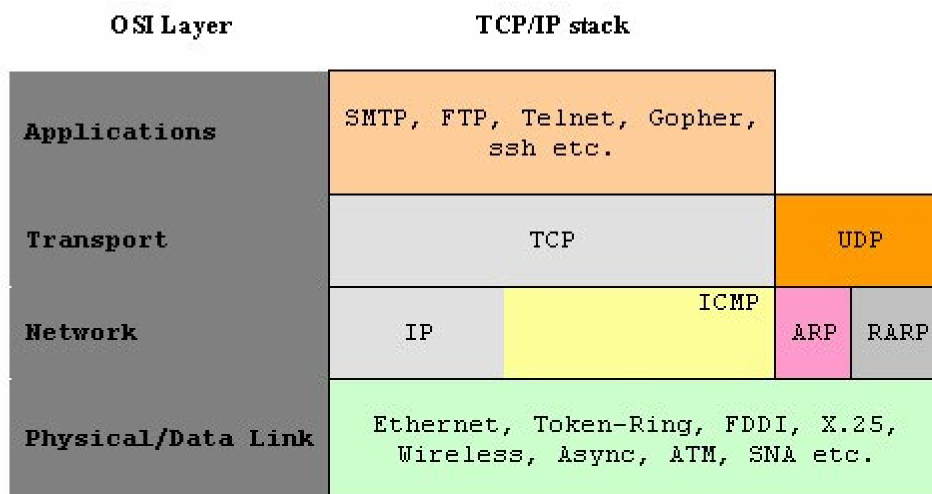

Figure 4

- The following figure is a TCP/IP architectural model. Frame, packet and message are same entity but called differently at the different layer because there are data encapsulations at every layer.

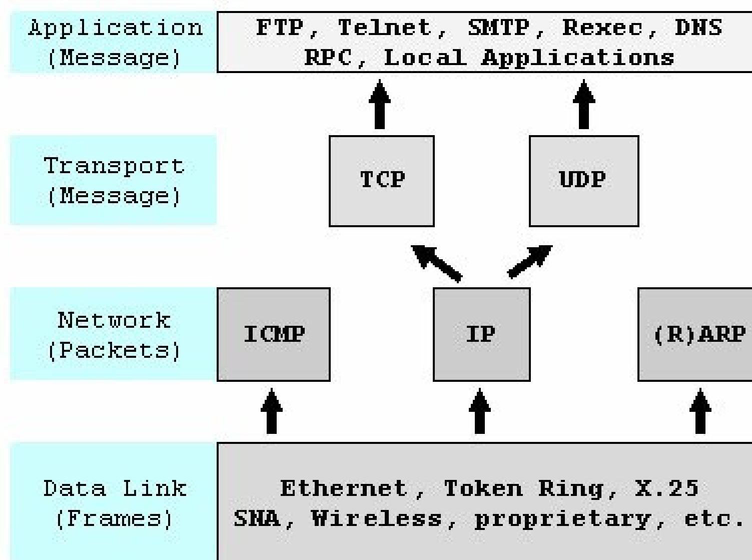

Figure 5

- The common applications that you encounter in your everyday use are:

  - FTP (file transfer protocol).
  - SMTP (simple mail transfer protocol).
  - telnet (remote logins).
  - rlogin (simple remote login between UNIX machines).
  - World Wide Web (built on http) and https (secure http).
  - NFS (network filing system – originally for SUNs).
  - TFTP (trivial file transfer protocol – used for booting).
  - SNMP (simple network management protocol).

- The user interfaces developed (programs) for the communication should depend on the platform.

Protocol

- In computing field, a **protocol** is a convention or standard rules that enables and controls the connection, communication and data transfer between two computing endpoints.
- Protocols may be implemented by hardware, software, or a combination of the two. At the lowest level, a protocol defines the behavior of a hardware connection.
- In term of controls, protocol may provide data transfer reliability, resiliency and integrity.
- An actual communication is defined by **various communication protocols**. In the context of data communication, a network protocol is a formal set of rules, conventions and data structure that governs how computers and other network devices exchange information over a network.
- In other words, protocol is a standard procedure and format that two data communication devices must understand, accept and use to be able to talk to each other.
- A wide variety of network protocols exist, which are defined by many standard organizations worldwide and technology vendors over years of technology evolution and developments.
- One of the most popular network protocol suites is TCP/IP, which is the heart of internetworking communications.

TCP and UDP

- TCP and UDP protocols were built on top of the IP protocol.
- Basic for the TCP:

  - Transmission Control Protocol is defined by **RFC-793**.
  - TCP provides connection-oriented transport service and reliable.
  - End-to-end transparent byte-stream.
  - E.g.: FTP, telnet, http, SMTP.

- While the UDP:

- User Datagram Protocol is defined by **RFC-768**.
- UDP provides datagram service that is a packet based.
- Connectionless.
- Unreliable.
- E.g.: NFS, TFTP.

Port numbers

- It is 16 bit integers. So we have 2^{16} = 65536 ports maximum.
- It is unique within a machine/IP address..
- To make a connection we need an IP address and port number of the protocol.
- The connection defined by:

```
IP address & port of server + IP address & port of client
```

- Normally, server port numbers are low numbers in the range 1 – 1023 and normally assign for root (Administrator) only.
- It is used for authentication e.g. rlogin.
- And normally, client port numbers are higher numbers starting at 1024.
- A server running on a well-known port lets the OS know what port it wants to listen on.
- Whereas a client normally simply lets the operating system picks a new port that isn't already in use.

Numeric IP Addresses

- Ipv4 Internet addresses are 32 bit integers.
- For convenience they are displayed in "dotted decimal" format.
- Each byte is presented as a decimal number.
- Dots separate the bytes for example:

```
131.95.115.204
```

IP Address Classes

- To simplify packet routing, internet addresses are divided into classes.
- An IP address has two parts: The network portion and the host portion.
- The network portion is unique to each company/organization/domain/group/network, and the host portion is unique to each system (host) in the network.
- Where the network portion ends and the host portion begin is different for each class of IP address.
- You can determine this by looking at the two high-order bits in the IP address.

192.168.1.100			
xxxxxxxx.xxxxxxxx.xxxxxxxx.xxxxxxxx Byte 1.Byte 2.Byte 3.Byte 4			
Class and Network size	**Range (decimal)**	**Network ID**	**Host ID**
Class A (Large)	1 -127	Byte 1	Bytes 2, 3, 4
Class B (Medium)	128 – 191	Bytes 1, 2	Bytes 3, 4
Class C (Small)	192 – 223	Bytes 1, 2, 3	Bytes 4

Table 2

- The first four bits (bits 0-3) of an address determine its class:

0xxx = class A bits 1-7 define a network. bits 8-31 define a host on that network. 128 networks with 16 million hosts.
10xx = class B bits 2-15 define a network. bits 16-31 define a host on that network. 16384 networks with 65536 hosts.
110x = class C bits 3-23 define a network. bits 24-31 define a host on that network.

2 million networks with 256 hosts.

Table 3

- The IP network portion can represent a very large network that may spans multiple geographic sites.
- To make this situation easier to manage, you can use subnetworks. Subnetworks use the two parts of the address to define a set of IP addresses that are treated as group. The subnetting divides the address into smaller networks.
- You configure a subnetwork by defining a mask, which is a series of bits. Then, the system performs a logical AND operation on these bits and the IP address.
- The 1 bit defines the subnetwork portion of the IP address (which must include at least the network portion). The 0 bits define the host portion.
- **Class D** is a multicast address and **class E** is reserved.
- As a summary:

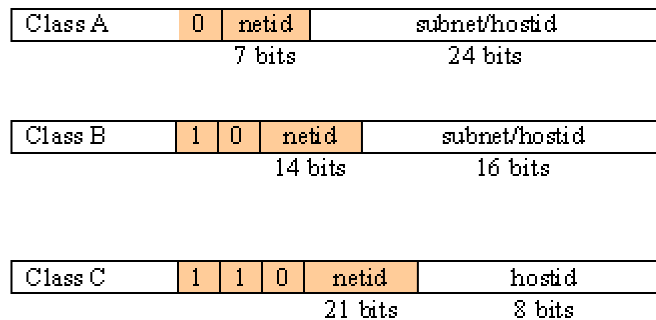

Figure 6

- Nowadays we use classless IP address. That means we subnet the class type IP into smaller subnet or smaller group of IP addresses creating smaller networks.
- Before the IPV4 run out of the IP addresses, now we have IPV6 with 128 bits.

Host Names and DNS

- People need names to make it simpler to use the Internet.
- The Domain Name System (DNS) can translate from name to number or from number to name. This is called name resolution.
- Name resolution done by Domain Name Service (DNS – although the term is same as the Domain Name System and same acronym, this is Microsoft implementation of the Domain Name System **:o)**) in Windows and in Unices/Linux it is done by BIND.

Ethernet Addresses

MAC and ARP protocol

- In Local Area Network (LAN) we have several network types such as Ethernet and Token Ring.
- The most widely used is Ethernet.
- Each Ethernet interface (Network Interface Card -NIC) has a unique Ethernet address provided by the manufacturer, hard coded into the NIC, normally called Media Access Control (MAC) or physical address.
- Ethernet addresses are 6 bytes shown as 6 hexadecimal values separated by colons.
- For example: `00:C0:F0:1F:3C:27`.
- You can see this MAC address by issuing the `arp` command as shown below:

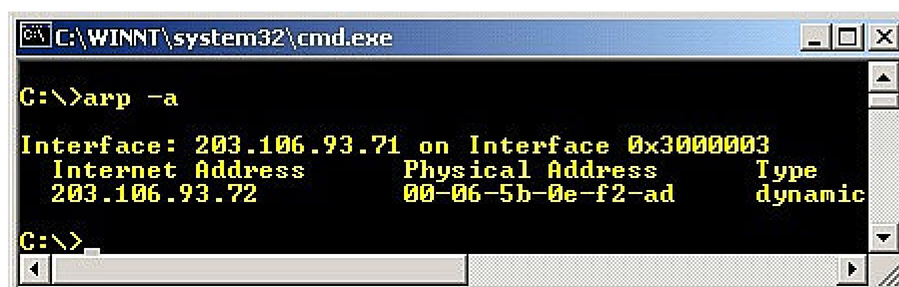

Figure 7

- Ethernet packets have header and data sections.
- The header contains the source and destination of the Ethernet addresses (MAC) and a 2 byte packet type.
- For IP packets the data area contains the IP fields which hold the IP source and destination addresses that are readable and more suitable for human.
- To send to an IP address, a computer uses the Address Resolution Protocol (`arp`) to determine a MAC address.

IPv6

- Current IP is IPv4 (Internet Protocol version 4).
- IPv4 has 32 bit addresses.
- Due to splitting addresses, 32 bits is not enough.
- IPv6 will have 128 bit addresses.
- Addresses will be shown in a colon hexadecimal format with internal strings of 0s omitted. For example:

```
69DC:88F4:FFFF:0:ABCD:DBAC:1234:FBCD:A12B::F6
```

- New service types exist to accommodate IPv6 such as multimedia and wireless.
- For Windows Xp and above, you can try the **`ipv6`** command at prompt to view and/or set the IPv6 configuration. For example: **`ipv6 if`**.

Distributed Applications

- The goal is to hide the fact that the application is distributed other than to provide the redundancy for reliability.
- User interfaces can look identical.
- Typically data resides on remote systems.
- In many instances, remote users interact with each other.

Application Protocols

- Protocol: a set of rules defining how to communicate.
- Application protocol: communication rules for an application.
- Standard protocols: documented in RFCs such as ftp, telnet, http.
- Non-standard protocols: programmers write a distributed application = new protocol.
- Programmers choose standard protocols where they apply.

E.g. telnet:

```
telnet computer.some.where [port]
telnet www.yahoo.com 23
```

- Port is a number defining which service to connect to.
- For example, port 23 is the default for telnet services.
- Ports, protocols and service names are specified in `/etc/services`.

Providing Concurrent Access to Services

- Users expect almost immediate response.
- Network servers must handle multiple clients "apparently simultaneously".
- The CPU and network must be shared.
- Normally in the form of multiple server **processes** or multiple server **threads** in one process.

The Client-Server Model

- TCP/IP enables peer-to-peer communication.
- Computers can cooperate as equals or in any desired way.
- Most distributed applications have special roles. For example:

- Server waits for a client request.
- Client requests a service from server.

Some Security Definitions

- Authentication: verifying a computer's identity.
- Authorization: determining whether permission is allowed.
- Data security: preserving data integrity.
- Privacy: preventing unauthorized access.
- Protection: preventing abuse.
- These security matters have experienced quite a pretty good evolution. The standards also have been produced such as the obsolete C2, formally known as **Trusted Computer System Evaluation Criteria** (TCSEC) then superseded by **Common Criteria** and the ISO version: **ISO 15408 Common Criteria for Information Technology Security Evaluation**.

Connectionless (UDP) vs Connection-Oriented (TCP) Servers

- Programmer can choose a connection-oriented server or a connectionless server based on their applications.
- In Internet Protocol terminology, the basic unit of data transfer is a **datagram**. This is basically a header followed by some data. The datagram socket is connectionless.
- User Datagram Protocol (UDP):

    - Is a connectionless.
    - A single socket can send and receive packets from many different computers.
    - Best effort delivery.
    - Some packets may be lost some packets may arrive out of order.

- Transmission Control Protocol (TCP):

    - Is a connection-oriented.
    - A client must connect a socket to a server.
    - TCP socket provides bidirectional channel between client and server.
    - Lost data is re-transmitted.
    - Data is delivered in-order.
    - Data is delivered as a stream of bytes.
    - TCP uses flow control.

- It is simple for a single UDP server to accept data from multiple clients and reply.
- It is easier to cope with network problems using TCP.

Stateless vs Stateful Servers

- A stateful server remembers client data (state) from one request to the next.
- A stateless server keeps no state information.
- Using a stateless file server, the client must:

    - Specify complete file names in each request.
    - Specify location for reading or writing.
    - Re-authenticate for each request.

- Using a stateful file server, the client can send less data with each request.
- A stateful server is simpler.
- On the other hand a stateless server is:

    - More robust.
    - Lost connections can't leave a file in an invalid state.
    - Rebooting the server does not lose state information because there is no state information hold.
    - Rebooting the client does not confuse a stateless server.

Concurrent Processing

Concurrency

- Real or apparent simultaneous processing.
- Time-sharing: a single CPU switches from 1 process to the next.
- Multiprocessing: multiple CPUs handle processes.

Network Concurrency

- Multiple distributed application share a network.
- With a single Ethernet segment or hub, one packet at a time can use the network. Network sharing is like time-sharing.
- With a switch, multiple packets can be in transit and transfer simultaneously, like multiprocessing.
- With several networks, multiple packets can be in transit and transfer like having multiple computers.

Server Concurrency

- An iterative server finishes one client request before accepting another.
- An iterative telnet daemon is almost useless.
- Concurrent servers are difficult to write.
- We will consider several designs for concurrency.

Programs vs Processes

- Program: executable instructions.
- Process: program being executed.
- Each process has its own private data.

e.g.: Multiple `pico` processes have different text on the screen.

Concurrency using `fork()` in UNIX/Linux

```
/*testpid.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (int argc, char **argv)
{
int i, pid;
pid = fork();

printf("Forking...the pid: %d\n", pid);
for (i = 0; i < 5; i++)
printf(" %d    %d\n", i, getpid());
if (pid)
wait(NULL);
return 0;
}
```

```
[bodo@bakawali testsocket]$ gcc -g testpid.c -o testpid
[bodo@bakawali testsocket]$ ./testpid
Forking...the pid: 0
 0     27166
 1     27166
 2     27166
 3     27166
 4     27166
Forking...the pid: 27166
 0     27165
 1     27165
 2     27165
 3     27165
 4     27165
```

- New process starts execution by returning from fork.
- Child and parent are nearly identical.
- Parent gets the child process id from fork.
- Child gets 0 back from fork.

- Parent should wait for child to exit.

Time slicing

- Round robin method.
- Operating system is interrupted regularly by a clock.
- In the clock interrupt handler the kernel checks to see if the current process has exceeded its time quantum.
- Processes are forced to take turns using the CPU but every process will get their time slice.

Using exec family to execute a new program in UNIX/Linux

- The following are exec family prototypes.

```
int execve(const char *file, char *const argv [], char *const envp[]);
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- `exec` family executes a new program (same process id).
- The first argument is the new program if it says path, it requires a full pathname otherwise, and it searches in the current path.
- Program arguments follow as a list or a vector.
- `execve()` and `execle()` allow specifying the environment.

Context Switching

- Having multiple server processes means context switches.
- A context switch consumes CPU time.
- Other processes are blocked during that time.
- We must consider the benefits of multiple servers versus the overhead.

Asynchronous I/O

- Asynchronous I/O means allowing a process to start an I/O operation and proceeding with other work while the I/O occurs.
- UNIX I/O occurs asynchronously if you use `select()`.
- A process asks the `select()` system call to tell which of a collection of file descriptors is ready to finish I/O.
- After calling `select()` the process can call `read()` or `write()` to perform I/O which is at that time no more than a copying of data to/from kernel space with real I/O either already done or scheduled for later.
- A server process can use `select()` to determine which of a collection of sockets it can read without blocking.

Programming Interface

TCP/IP Application Programming Interface (API)

- API: routines supplied by the OS defining the interface between an application and the protocol software.
- Better to avoid vendor-specific data format and features, use standard APIs for portability.
- The API only suggests required functionality and it depend on the implementation.
- For UNIX - **socket** (original Berkeley system calls) and **TLI** (Transport Layer Interface - AT&T UNIX System V).
- For Apple Mac – MacTCP.
- For MS Windows – Winsock (quite similar to socket).
- There is also other TCP/IP APIs that implementation dependent.
- Unices TCP/IP APIs are kernel system calls.
- Mac and Windows using extension/drivers and dynamic link library (dll).
- In this Tutorial we will use socket APIs and in general socket refer to socket APIs that includes the `socket()`.

- In fact the APIs just routines/functions in C language.

Required Functionality

- Allocate resources for communication such as memory.
- Specify local and remote endpoints.
- Initiate a client connection.
- Wait for a client connection.
- Send or receive data.
- Determine when data arrives.
- Generate urgent data.
- Handle received urgent data.
- Terminate a connection.
- Abort.
- Handle errors.
- Release resources.

System Calls

- An operating system should run user programs in a restricted mode i.e. user program should not do I/O directly.
- User programs should make a system call to allow trusted code to perform I/O.
- In UNIX, functions like `open()`, `read()`, `write()`, `close()` are actually system calls.
- A UNIX system call is a transition from user mode to kernel mode.
- TCP/IP code is called through the system calls.

UNIX I/O with TCP/IP

- To a certain degree, I/O with sockets is like file I/O.
- TCP/IP sockets are identified using file descriptors.
- `read()` and `write()` work with TCP/IP sockets.
- `open()` is not adequate for making a connection.
- Calls are needed to allow servers to wait for connections.
- UDP data is always a datagram and not a stream of bytes.

The Socket Interface

- Socket is an Application Programming Interface (API) used for Interprocess Communications (IPC).
- It is a well defined method of connecting two processes, locally or across a network.
- It is a Protocol and Language Independent.
- Often referred to as Berkeley Sockets or BSD Sockets.
- The following figure illustrates the example of client/server relationship of the socket APIs for connection-oriented protocol (TCP).

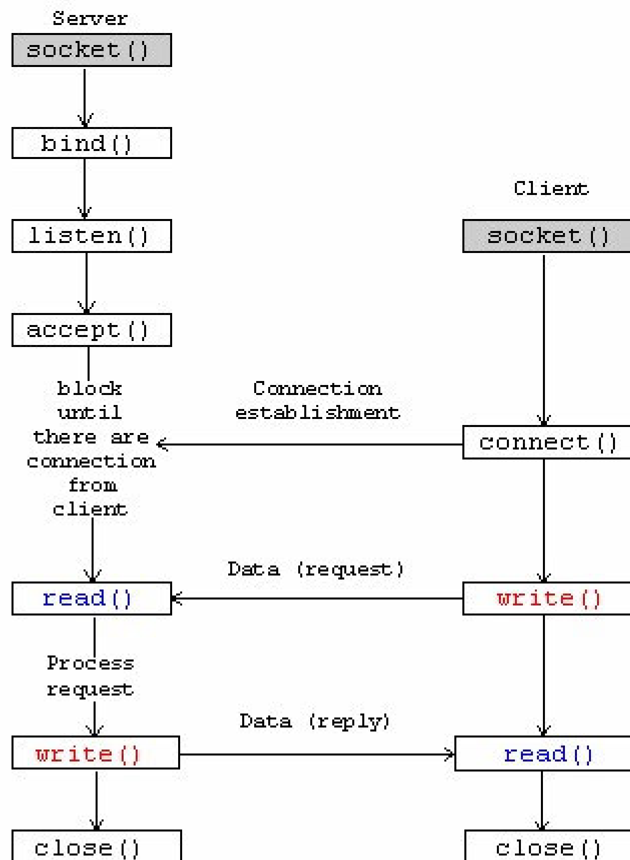

Figure 8

- The following figure illustrates the example of client/server relationship of the socket APIs for a connectionless protocol (UDP).

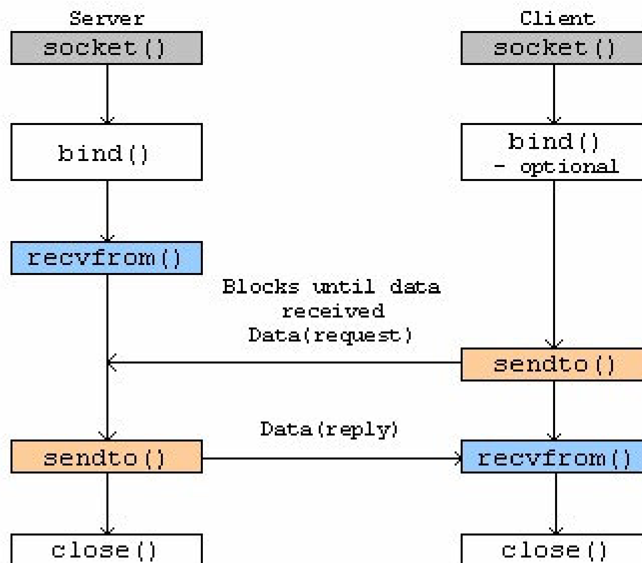

Figure 9

Berkeley Sockets: API for TCP/IP Communication

- Uses existing I/O features where possible.
- Allows TCP/IP connections, internal connections, and possibly more.
- Started in BSD UNIX in the 1980s.
- BSD UNIX adopted by Sun, Tektronix and Digital.

- Now the socket interface is a de facto standard.
- Sockets make the network look much like a file system.

Socket Descriptors

- UNIX `open()` yields a file descriptor: a small integer used to read/write a file.
- UNIX keeps a file descriptor table for each process an array of pointers to the data about the open files.
- A file descriptor is used to index the array.
- Sockets are added to this abstraction.
- The `socket()` system call returns a socket descriptor.
- Actually, files and sockets are accessed using the same table.
- The structure pointed to by a table entry has a field which tells whether it is a file or socket.

System Data Structures for Sockets

- In order to use a socket, the kernel needs to keep track of several pieces of data as the following:

    1. Protocol Family: a parameter to the socket call.
    2. Service Type (Stream, Datagram): parameter to socket.
    3. Local IP Address: can be set with `bind()`.
    4. Local Port: can be set with `bind()`.
    5. Remote IP Address: can be set with `connect()`.
    6. Remote Port: can be set with `connect()`.

- Ultimately all 6 values must be known to do the communication.

Active vs Passive Sockets

- A server uses a passive socket to wait the client connections.
- A client uses an active socket to initiate a connection.
- Both start using the `socket()` call.
- Later on, servers and clients will use other calls.

Socket Endpoints

- TCP/IP communication occurs between 2 endpoints.
- An endpoint is defined as an IP address and a port number.
- To allow other protocols to merge into the socket abstraction, address families are used.
- We will use `PF_INET` for internet protocol family.
- We will also use `AF_INET` for internet address family. Normally `PF_INET` = `AF_INET` = 2.
- Socket types for `AF_INET` are listed in the following Table.

Socket type	Protocol	TYPE
STREAM	TCP, Systems Network Architecture (SNA-IBM), Sequenced Packet eXchange (SPX-Novell).	SOCK_STREAM
SEQPACKET	SPX.	SOCK_SEQPACKET
DGRAM	UDP, SNA, Internetwork Packet eXchange (IPX-Novell).	SOCK_DGRAM
RAW	IP.	SOCK_RAW

Table 4: `AF_INET` socket combinations

- There are other address families that you will find somewhere and sometime such as:

    - AF_UNIX
    - AF_NS
    - AF_TELEPHONY

AF_UNIX address family:

- The system uses this address family for communicating between two programs that are on the same physical machine. The address is a path name to an entry that is in a hierarchical file system.
- Sockets with address family `AF_UNIX` use the `sockaddr_un` address structure:

```
struct sockaddr_un {
short sun_family;
char sun_path[126];
};
```

- The sun_family field is the address family. The sun_path field is the pathname. The <sys/un.h> header file contains the sockaddr_un address structure definition.
- For the AF_UNIX address family, protocol specifications do not apply because protocol standards are not involved.
- The communications mechanism between the two processes on the same machine is specific to that machine.

AF_NS address family:

- This address family uses addresses that follow Novell or Xerox NS protocol definitions. It consists of a 4-byte network, a 6-byte host (node), and a 2-byte port number.
- Sockets with address family AF_NS use the sockaddr_ns address structure:

```
struct sockaddr_ns {
unsigned short sns_family;
struct ns_addr sns_addr;
char sns_zero[2];
};
```

AF_TELEPHONY address family:

- Telephony domain sockets (sockets that use the AF_TELEPHONY address family) permit the user to initiate (dial) and complete (answer) telephone calls through an attached ISDN telephone network using standard socket APIs.
- The sockets forming the endpoints of a connection in this domain are really the called (passive endpoint) and calling (active endpoint) parties of a telephone call.
- The AF_TELEPHONY addresses are telephone numbers that consist of up to 40 digits (0 - 9), which are contained in sockaddr_tel address structures.
- The system supports AF_TELEPHONY sockets only as connection-oriented (type SOCK_STREAM) sockets.
- Keep in mind that a connection in the telephony domain provides no more reliability than that of the underlying telephone connection. If guaranteed delivery is desired, you must accomplish this at the application level, such as in fax applications that use this family.
- Sockets with address family AF_TELEPHONY use the sockaddr_tel address structure.

```
struct sockaddr_tel {
short stel_family;
struct tel_addr stel_addr;
char stel_zero[4];
};
```

- The telephony address consists of a 2-byte length followed by a telephone number of up to 40 digits (0 - 9).

```
struct tel_addr {
unsigned short t_len;
char t_addr[40];
};
```

- The stel_family field is the address family. The stel_addr field is the telephony address, and stel_zero is a reserved field. The <nettel/tel.h> header file contains the tel_addr and sockaddr_tel structure definitions.

Generic Socket Address Structure

Host IP Addresses

- Each computer on the Internet has one or more Internet addresses, numbers which identify that computer among all those on the Internet.

- Users typically write numeric host addresses as sequences of four numbers, separated by periods, as in `128.54.46.100`.
- Each computer also has one or more **host names**, which are strings of words separated by periods, as in `www.google.com`.
- Programs that let the user specify a host typically accept both numeric addresses and host names.
- But the program needs a numeric address to open a connection; to use a host name; you must convert it to the numeric address it stands for.

Internet Host Addresses – Abstract Host Address

- Each computer on the Internet has one or more Internet addresses, numbers which identify that computer among all those on the Internet.
- An Internet host address is a number containing four bytes of data. These are divided into two parts, a **network number** and a **local network address number** within that network.
- The network number consists of the first one, two or three bytes; the rest of the bytes are the local address.
- Network numbers are registered with the Network Information Center (NIC), and are divided into three classes as discussed before: class A, B, and C for the IPv4. The local network address numbers of individual machines are registered with the administrator of the particular network.
- Since a single machine can be a member of multiple networks, it can have multiple Internet host addresses.
- However, there is never supposed to be more than one machine with the same host address.
- There are four forms of the **standard numbers-and-dots notation** for Internet addresses as discussed before:

  a.b.c.d
  This specifies all four bytes of the address individually.

  a.b.c
  The last part of the address, **c**, is interpreted as a 2-byte quantity. This is useful for specifying host addresses in a Class B network with network address number **a.b**.

  a.b
  The last part of the address, **c**, is interpreted as a 3-byte quantity. This is useful for specifying host addresses in a Class A network with network address number **a**.

  a
  If only one part is given, this corresponds directly to the host address number.

- Within each part of the address, the usual C conventions for specifying the radix apply. In other words, a leading '`0x`' or '`0X`' implies hexadecimal radix; a leading '`0`' implies octal; and otherwise decimal radix is assumed.

Host Address Data Type - Data type for a host number.

- Internet host addresses are represented in some contexts as integers (type `unsigned long int`).
- In other contexts, the integer is packaged inside a structure of type `struct in_addr`. It would be better if the usages were made consistent, but it is not hard to extract the integer from the structure or put the integer into a structure.
- The following basic definitions for Internet addresses appear in the header file '`netinet/in.h`':

Data Type `struct in_addr`

- This data type is used in certain contexts to contain an Internet host address. It has just one field, named `s_addr`, which records the host address number as an `unsigned long int`.

Macro `unsigned long int INADDR_LOOPBACK`

- You can use this constant to stand for the "address of this machine" instead of finding its actual address.
- It is the Internet address '`127.0.0.1`', which is usually called '`localhost`'. This special constant saves you the trouble of looking up the address of your own machine.

- Also, the system usually implements `INADDR_LOOPBACK` specially, avoiding any network traffic for the case of one machine talking to itself.

Macro `unsigned long int INADDR_ANY`

- You can use this constant to stand for "any incoming address" when binding to an address.
- This is the usual address to give in the `sin_addr` member of `struct sockaddr_in` when you want your server to accept Internet connections.

Macro `unsigned long int INADDR_BROADCAST`

- This constant is the address you use to send a broadcast message.

Macro unsigned long int `INADDR_NONE`

- This constant is returned by some functions to indicate an error.

Host Address Functions - Functions to operate on them.

- These additional functions for manipulating Internet addresses are declared in '`arpa/inet.h`'.
- They represent Internet addresses in **network byte order**; they represent network numbers and local-address-within-network numbers in **host byte order**.

Function `int inet_aton(const char *name, struct in_addr *addr)`

- This function converts the Internet host address **`name`** from the standard numbers-and-dots notation into binary data and stores it in the `struct in_addr` that **`addr`** points to.
- `inet_aton` returns nonzero if the address is valid, zero if not.

Function `unsigned long int inet_addr(const char *name)`

- This function converts the Internet host address **`name`** from the standard numbers-and-dots notation into binary data. If the input is not valid, `inet_addr` returns `INADDR_NONE`.
- This is an **obsolete** interface to `inet_aton`, described above; it is obsolete because `INADDR_NONE` is a valid address (`255.255.255.255`), and `inet_aton` provides a cleaner way to indicate error return.

Function `unsigned long int inet_network(const char *name)`

- This function extracts the network number from the address **`name`**, given in the standard numbers-and-dots notation. If the input is not valid, `inet_network` returns `-1`.

Function `char * inet_ntoa(struct in_addr addr)`

- This function converts the Internet host address **`addr`** to a string in the standard numbers-and-dots notation. The return value is a pointer into a statically-allocated buffer.
- Subsequent calls will overwrite the same buffer, so you should copy the string if you need to save it.

Function `struct in_addr inet_makeaddr(int net, int local)`

- This function makes an Internet host address by combining the network number **`net`** with the local-address-within-network number **`local`**.

Function `int inet_lnaof(struct in_addr addr)`

- This function returns the local-address-within-network part of the Internet host address **`addr`**.

Function `int inet_netof(struct in_addr addr)`

- This function returns the network number part of the Internet host address **`addr`**.

Host Names - Translating host names to host IP numbers

- Besides the standard numbers-and-dots notation for Internet addresses, you can also refer to a host by a symbolic name.
- The advantage of a symbolic name is that it is usually easier to remember. For example, the machine with Internet address '128.52.46.32' is also known as 'testo.google.com'; and other machines in the 'google.com' domain can refer to it simply as 'testo'.
- Internally, the system uses a database to keep track of the mapping between host names and host numbers.
- This database is usually either the file '/etc/hosts' or an equivalent provided by a name/DNS server. The functions and other symbols for accessing this database are declared in 'netdb.h'. They are BSD features, defined unconditionally if you include 'netdb.h'.
- The IP address to name and vice versa is called name resolution. It is done by Domain Name Service. Other than the hosts file, in Windows platform it is called DNS (Domain Name **Service**) and other Microsoft specifics may use WINS or lmhost file.
- Keep in mind that the general term actually Domain Name **System** also has DNS acronym. In UNIX it is done by BIND.
- The complete process or steps taken for name resolution quite complex but Windows normally use DNS service and UNIX/Linux normally use BIND.

Data Type `struct hostent`

- This data type is used to represent an entry in the hosts database. It has the following members:

Data Type	Description
char *h_name	This is the "official" name of the host.
char **h_aliases	These are alternative names for the host, represented as a null-terminated vector of strings.
int h_addrtype	This is the host address type; in practice, its value is always AF_INET. In principle other kinds of addresses could be represented in the data base as well as Internet addresses; if this were done, you might find a value in this field other than AF_INET.
int h_length	This is the length, in bytes, of each address.
char **h_addr_list	This is the vector of addresses for the host. Recall that the host might be connected to multiple networks and have different addresses on each one. The vector is terminated by a null pointer.
char *h_addr	This is a synonym for h_addr_list[0]; in other words, it is the first host address.

Table 5

- As far as the host database is concerned, each address is just a block of memory h_length bytes long.
- But in other contexts there is an implicit assumption that you can convert this to a struct in_addr or an unsigned long int. Host addresses in a struct hostent structure are always given in network byte order.
- You can use gethostbyname() or gethostbyaddr() to search the hosts database for information about a particular host. The information is returned in a statically-allocated structure.
- You must copy the information if you need to save it across calls.

Function `struct hostent * gethostbyname(const char *name)`

- The gethostbyname() function returns information about the host named **name**. If the lookup fails, it returns a null pointer.

Function `struct hostent * gethostbyaddr(const char *addr, int length, int format)`

- The gethostbyaddr() function returns information about the host with Internet address **addr**. The **length** argument is the size (in bytes) of the address at **addr**.
- **format** specifies the address format; for an Internet address, specify a value of AF_INET.
- If the lookup fails, gethostbyaddr() returns a null pointer.

- If the name lookup by `gethostbyname()` or `gethostbyaddr()` fails, you can find out the reason by looking at the value of the variable `h_errno`.
- Before using `h_errno`, you must declare it like this:

```
extern int h_errno;
```

- Here are the error codes that you may find in `h_errno`:

h_errno	Description
HOST_NOT_FOUND	No such host is known in the data base.
TRY_AGAIN	This condition happens when the name server could not be contacted. If you try again later, you may succeed then.
NO_RECOVERY	A non-recoverable error occurred.
NO_ADDRESS	The host database contains an entry for the name, but it doesn't have an associated Internet address.

Table 6

- You can also scan the entire hosts database one entry at a time using `sethostent()`, `gethostent()`, and `endhostent()`.
- Be careful in using these functions, because they are not re-entrant.

Function `void sethostent(int stayopen)`

- This function opens the `hosts` database to begin scanning it. You can then call `gethostent()` to read the entries.
- If the `stayopen` argument is nonzero, this sets a flag so that subsequent calls to `gethostbyname()` or `gethostbyaddr()` will not close the database (as they usually would).
- This makes for more efficiency if you call those functions several times, by avoiding reopening the database for each call.

Function `struct hostent * gethostent()`

- This function returns the next entry in the `hosts` database. It returns a null pointer if there are no more entries.

Function `void endhostent()`

- This function closes the `hosts` database.

The API Details

- In this section and that follows we will discuss the socket APIs details: the structures, functions, macros and types.

`struct sockaddr`

```
struct sockaddr {
    u_char  sa_len;
    u_short sa_family;     // address family, AF_xxx
    char    sa_data[14];   // 14 bytes of protocol address
};
```

- **`sockaddr`** consists of the following parts:

  1. The short integer that defines the address family (the value that is specified for address family on the **`socket()`** call).
  2. Fourteen bytes that are reserved to hold the address itself.

- Originally `sa_len` was not there.
- Depending on the address family, `sa_data` could be a file name or a socket endpoint...
- `sa_family` can be a variety of things, but it'll be `AF_INET` for everything we do in this Tutorial.

- sa_data contains a destination address and port number for the socket. This is rather unwieldy since you don't want to tediously pack the address in the sa_data by hand.
- To deal with struct sockaddr, programmers created a parallel structure: struct sockaddr_in ("in" for "Internet".)

struct sockaddr_in

```
struct sockaddr_in {
    u_char  sin_len;
    u_short sin_family;         //Address family
    u_short sin_port;           //Port number
    struct  in_addr sin_addr;   //Internet or IP address
    char    sin_zero[8];        //Same size as struct sockaddr
};
```

- The sin_family field is the address family (always AF_INET for TCP and UDP).
- The sin_port field is the port number, and the sin_addr field is the Internet address. The sin_zero field is reserved, and you must set it to hexadecimal zeroes.
- Data type struct in_addr - this data type is used in certain contexts to contain an Internet host address. It has just one field, named s_addr, which records the host address number as an unsigned long int.
- sockaddr_in is a "specialized" sockaddr.
- sin_addr could be u_long.
- sin_addr is 4 bytes and 8 bytes are unused.
- sockaddr_in is used to specify an endpoint.
- The sin_port and sin_addr must be in **Network Byte Order**.

Socket System Calls

socket()

```
NAME
       socket() - create an endpoint for communication

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

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

- domain should be set to "AF_INET", just like in the struct sockaddr_in.
- The type argument tells the kernel what kind of socket this is. For example SOCK_STREAM or SOCK_DGRAM.
- Just set protocol to "0" to have socket() choose the correct protocol based on the type.
- protocol is frequently 0 if only one protocol in the family supports the specified type. You can look at /etc/protocols.
- There are many more domains and types that you will find later on.
- Also, there's a "better" way to get the protocol. See the getprotobyname() man page.
- socket() simply returns to you an integer of the socket descriptor that you can use in later system calls, or -1 on error. The global variable errno is set to the error's value (see the perror() man page).
- In some documentation, you'll see the mentioning of a mystical "PF_INET.
- Once a long time ago, it was thought that maybe an address family (what the "AF" in "AF_INET" stands for) might support several protocols that were referenced by their protocol family (what the "PF" in "PF_INET" stands for). That didn't happen.
- So the correct thing to do is to use AF_INET in your struct sockaddr_in and PF_INET in your call to socket(). But practically speaking, you can use AF_INET everywhere.

bind()

```
NAME
       bind() - bind a name to a socket

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

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

- `sockfd` is the socket file descriptor returned by `socket()`.
- `my_addr` is a pointer to a `struct sockaddr` that contains information about your address, namely, port and IP address.
- `addrlen` can be set to `sizeof(struct sockaddr)`.
- Bind attaches a local endpoint to a socket.
- Once you have a socket, you might have to associate that socket with a port on your local machine.
- Typically servers use `bind()` to attach to well-known ports so clients can connect.
- This is commonly done if you're going to `listen()` for incoming connections on a specific port.
- The port number is used by the kernel to match an incoming packet to a certain process's socket descriptor. If you're going to only be doing a `connect()` that is just a client, this may be unnecessary.
- Let's have an example:

```
[bodo@bakawali testsocket]$ cat test1.c
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define MYPORT 3334

int main()
{
int sockfd;  /*socket file descriptor*/
struct sockaddr_in my_addr;

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
  perror("Server-socket() error lol!");
  exit(1);
}
else
  printf("Server-socket() sockfd is OK...\n");

/* host byte order */
my_addr.sin_family = AF_INET;
/* short, network byte order */
my_addr.sin_port = htons(MYPORT);
my_addr.sin_addr.s_addr = INADDR_ANY;
/* zero the rest of the struct */
memset(&(my_addr.sin_zero), 0, 8);

if(bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
{
  perror("Server-bind() error lol!");
  exit(1);
}
else
  printf("Server-bind() is OK...\n");

/*....other codes....*/

return 0;
}

[bodo@bakawali testsocket]$ gcc test1.c -o test1
[bodo@bakawali testsocket]$ ./test1
Server-socket() sockfd is OK...
Server-bind() is OK...
```

- `my_addr.sin_port` and `my_addr.sin_addr.s_addr` are in Network Byte Order.
- For `bind()`, some of the process of getting your own IP address and/or port can be automated:

```
/* choose an unused port at random */
my_addr.sin_port = 0;
/* use my IP address */
my_addr.sin_addr.s_addr = INADDR_ANY;
```

- By setting `my_addr.sin_port` to zero, you are telling `bind()` to choose the port for you.

- Likewise, by setting `my_addr.sin_addr.s_addr` to `INADDR_ANY`, you are telling it to automatically fill in the IP address of the machine the process is running on.
- `INADDR_ANY` is actually zero. For `0.0.0.0`, it means any IP.

```
/* choose an unused port at random */
my_addr.sin_port = htons(0);
/* use my IP address */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

- Now our code quite portable.
- `bind()` also returns `-1` on error and sets `errno` to the error's value.
- When calling `bind()`, don't go below 1024 for your port numbers. All ports below 1024 are reserved (unless you're the superuser/root or suid type access).
- You can have any port number above that, right up to 65535 (2^{16}) provided they aren't already being used by another program.
- Sometimes, you might notice, you try to rerun a server and `bind()` fails, claiming "Address already in use". This means a socket that was connected is still hanging around in the kernel, and it's hogging the port.
- You can either wait for it to clear (a minute or so), or add code to your program allowing it to reuse the port, like this:

```
int yes = 1;
/* "Address already in use" error message */
if(setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
{
  perror("setsockopt() error");
  exit(1);
}
else
  printf("setsockopt() is OK.\n");
```

- There are times when you won't absolutely have to call `bind()`. If you are `connect()`ing to a remote machine and you don't care what your local port is (as is the case with **telnet** where you only care about the remote port), you can simply call `connect()`, it'll check to see if the socket is unbound, and will `bind()` it to an unused local port if necessary.
- For the ports that has been blocked or closed by the firewall for security reason, **you have to open it for communication**.
- And if the access to the port denied, you have to allow it.
- Standard ports with their respective services have been defined in `/etc/protocol`. You can define ports for specific services by editing the `/etc/protocol`.
- Then with the newly defined ports, the new service is defined and created in `/etc/xinetd.d`. Please check the man pages for `xinetd` service (`inetd` is the older one).
- The following section summarizes the sockets related function prototypes.

connect()

```
NAME
       connect() - initiate a connection on a socket.

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

       int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

- `sockfd` is our friendly neighborhood socket file descriptor, as returned by the `socket()` call, `serv_addr` is a `struct sockaddr` containing the destination port and IP address
- `addrlen` can be set to `sizeof(struct sockaddr)`.
- As an example, for telnet client application, firstly, get a socket file descriptor. Then if no error, we are ready to connect to remote host, let say "`127.0.0.1`" on port "`23`", the standard telnet port. For this we need `connect()`.
- Let's have an example:

```
[bodo@bakawali testsocket]$ cat test2.c
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>

#define DEST_IP "127.0.0.1"
#define DEST_PORT 80

int main(int argc, char *argv[])
{
int sockfd;
/* will hold the destination addr */
struct sockaddr_in dest_addr;

sockfd = socket(AF_INET, SOCK_STREAM, 0);

if(sockfd == -1)
{
  perror("Client-socket() error lol!");
  exit(1);
}
else
  printf("Client-socket() sockfd is OK...\n");

/* host byte order */
dest_addr.sin_family = AF_INET;
/* short, network byte order */
dest_addr.sin_port = htons(DEST_PORT);
dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);

/* zero the rest of the struct */
memset(&(dest_addr.sin_zero), 0, 8);

if(connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr)) == -1)
{
  perror("Client-connect() error lol");
  exit(1);
}
else
  printf("Client-connect() is OK...\n");

/*...other codes...*/

return 0;
}
```

```
[bodo@bakawali testsocket]$ gcc test2.c -o test2
[bodo@bakawali testsocket]$ ./test2
Client-socket() sockfd is OK...
Client-connect() error lol: Connection refused
```

- Again, be sure to check the return value from `connect()`. It will return `-1` on error and set the variable `errno`.
- Also, notice that we didn't call `bind()`. Basically, we don't care about our local port number; we only care where we're going to connect to that is the remote port.
- The kernel will choose a local port for us, and the site we connect to will automatically get this information from us.

listen()

```
NAME
       listen() - listen for connections on a socket

SYNOPSIS
       #include <sys/socket.h>

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

- `sockfd` is the usual socket file descriptor from the `socket()` system call.
- `backlog` is the number of connections allowed on the incoming queue.
- As an example, for the server, if you want to wait for incoming connections and handle them in some way, the steps are: first you `listen()`, then you `accept()`.
- The incoming connections are going to wait in this queue until you `accept()` (explained later) them and this is the limit on how many can queue up.
- Again, as per usual, `listen()` returns `-1` and sets `errno` on error.

- We need to call `bind()` before we call `listen()` or the kernel will have us listening on a random port.
- So if you're going to be listening for incoming connections, the sequence of system calls you'll make is something like this:

```
socket();
bind();
listen();
/*accept() goes here*/
```

accept()

```
NAME
       accept() - accept a connection on a socket

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

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

- `sockfd` is the `listen()`ing socket descriptor.
- `addr` will usually be a pointer to a local `struct sockaddr_in`. This is where the information about the incoming connection will go (and with it you can determine which host is calling you from which port).
- `addrlen` is a local integer variable that should be set to `sizeof(struct sockaddr_in)` before its address is passed to `accept()`.
- `accept()` will not put more than that many bytes into `addr`. If it puts fewer in, it'll change the value of `addrlen` to reflect that.
- `accept()` returns `-1` and sets `errno` if an error occurs.
- Basically, after `listen()`, a server calls `accept()` to wait for the next client to connect. `accept()` will create a new socket to be used for I/O with the new client. The server then will continue to do further accepts with the original `sockfd`.
- When someone try to `connect()` to your machine on a port that you are `listen()`ing on, their connection will be queued up waiting to be `accept()`ed. You call `accept()` and you tell it to get the pending connection.
- It'll return to you a **new socket file descriptor** to use for this single connection.
- Then, you will have two socket file descriptors where the original one is still listening on your port and the newly created one is finally ready to `send()` and `recv()`.
- A program example:

```
[bodo@bakawali testsocket]$ cat test3.c
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

/*the port users will be connecting to*/
#define MYPORT 3440
/* how many pending connections queue will hold */
#define BACKLOG 10

int main()
{
/* listen on sock_fd, new connection on new_fd */
int sockfd, new_fd;
/* my address information, address where I run this program */
struct sockaddr_in my_addr;
/* remote address information */
struct sockaddr_in their_addr;
int sin_size;

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if(sockfd == -1)
{
  perror("socket() error lol!");
  exit(1);
}
else
  printf("socket() is OK...\n");
```

```
/* host byte order */
my_addr.sin_family = AF_INET;
/* short, network byte order */
my_addr.sin_port = htons(MYPORT);
/* auto-fill with my IP */
my_addr.sin_addr.s_addr = INADDR_ANY;

/* zero the rest of the struct */
memset(&(my_addr.sin_zero), 0, 8);

if(bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
{
  perror("bind() error lol!");
  exit(1);
}
else
  printf("bind() is OK...\n");

if(listen(sockfd, BACKLOG) == -1)
{
  perror("listen() error lol!");
  exit(1);
}
else
  printf("listen() is OK...\n");

/* ...other codes to read the received data... */

sin_size = sizeof(struct sockaddr_in);
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);

if(new_fd == -1)
  perror("accept() error lol!");
else
  printf("accept() is OK...\n");

/*.....other codes.......*/

close(new_fd);
close(sockfd);
return 0;
}
```

```
[bodo@bakawali testsocket]$ gcc test3.c -o test3
[bodo@bakawali testsocket]$ ./test3
socket() is OK...
bind() is OK...
listen() is OK...
```

- Note that we will use the socket descriptor `new_fd` for all `send()` and `recv()` calls.
- If you're only getting one single connection ever, you can `close()` the listening `sockfd` in order to prevent more incoming connections on the same port, if you so desire.

send()

```
int send(int sockfd, const void *msg, int len, int flags);
```

- `sockfd` is the socket descriptor you want to send data to (whether it's the one returned by `socket()` or the new one you got with `accept()`).
- `msg` is a pointer to the data you want to send.
- `len` is the length of that data in bytes.
- Just set `flags` to 0. (See the `send()` man page for more information concerning flags).
- Some sample code might be:

```
char *msg = "I was here!";
int len, bytes_sent;
...
...
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
...
```

- `send()` returns the number of bytes actually sent out and this might be less than the number you told it to send.

- Sometimes you tell it to send a whole gob of data and it just can't handle it. It'll fire off as much of the data as it can, and trust you to send the rest later.
- Remember, if the value returned by `send()` doesn't match the value in `len`, it's up to you to send the rest of the string.
- If the packet is small (less than 1K or so) it will probably manage to send the whole thing all in one go.
- Again, `-1` is returned on error, and `errno` is set to the error number.

recv()

- The `recv()` call is similar in many respects:

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

- `sockfd` is the socket descriptor to read from, `buf` is the buffer to read the information into and `len` is the maximum length of the buffer.
- `flags` can again be set to `0`. See the `recv()` man page for flag information.
- `recv()` returns the number of bytes actually read into the buffer, or `-1` on error (with `errno` set, accordingly).
- If `recv()` return `0`, this can mean only one thing that is the remote side has closed the connection on you. A return value of `0` is `recv()`'s way of letting you know this has occurred.
- At this stage you can now pass data back and forth on stream sockets.
- These two functions `send()` and `recv()` are for communicating over stream sockets or connected datagram sockets.
- If you want to use regular unconnected datagram sockets (UDP), you need to use the `sendto()` and `recvfrom()`.
- Or you can use more general, the normal file system functions, `write()` and `read()`.

write()

```
NAME
       write() - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);
```

- Writes to files, devices, sockets etc.
- Normally data is copied to a system buffer and write occurs asynchronously.
- If buffers are full, write can block.

read()

```
NAME
       read() - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);
```

- Reads from files, devices, sockets etc.
- If a socket has data available up to `count` bytes are read.
- If no data is available, the read blocks.
- If less than `count` bytes are available, read returns what it can without blocking.
- For UDP, data is read in whole or partial datagrams. If you read part of a datagram, the rest is discarded.

close() and shutdown()

```
NAME
       close() - close a file descriptor

SYNOPSIS
       #include <unistd.h>
```

```
int close(int sockfd);
```

- You can just use the regular UNIX file descriptor `close()` function:

```
close(sockfd);
```

- This will prevent any more reads and writes to the socket. Anyone attempting to read or write the socket on the remote end will receive an error.
- UNIX keeps a count of the number of uses for an open file or device.
- Close decrements the use count. If the use count reaches 0, it is closed.
- Just in case you want a little more control over how the socket closes, you can use the `shutdown()` function.
- It allows you to cut off communication in a certain direction, or both ways just like `close()` does.
- The prototype:

```
int shutdown(int sockfd, int how);
```

- `sockfd` is the socket file descriptor you want to shutdown, and `how` is one of the following:

    1. 0 – Further receives are disallowed.
    2. 1 – Further sends are disallowed.
    3. 2 – Further sends and receives are disallowed (like `close()`).

- `shutdown()` returns `0` on success, and `-1` on error (with `errno` set accordingly).
- If you deign to use `shutdown()` on unconnected datagram sockets, it will simply make the socket unavailable for further `send()` and `recv()` calls (remember that you can use these if you `connect()` your datagram socket).
- It's important to note that `shutdown()` doesn't actually close the file descriptor, it just change its usability.
- To free a socket descriptor, you need to use `close()`.

`sendto()` and `recvfrom()` for DATAGRAM (UDP)

- Since datagram sockets aren't connected to a remote host, we need to give the destination address before we send a packet. The prototype is:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const
struct sockaddr *to, int tolen);
```

- This call is basically the same as the call to `send()` with the addition of two other pieces of information.
- `to` is a pointer to a `struct sockaddr` (which you'll probably have as a `struct sockaddr_in` and cast it at the last minute) which contains the destination IP address and port.
- `tolen` can simply be set to `sizeof(struct sockaddr)`.
- Just like with `send()`, `sendto()` returns the number of bytes actually sent (which, again, might be less than the number of bytes you told it to send!), or `-1` on error.
- Equally similar are `recv()` and `recvfrom()`. The prototype of `recvfrom()` is:

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct
sockaddr *from, int *fromlen);
```

- Again, this is just like `recv()` with the addition of a couple fields.
- `from` is a pointer to a local `struct sockaddr` that will be filled with the IP address and port of the originating machine.
- `fromlen` is a pointer to a local `int` that should be initialized to `sizeof(struct sockaddr)`. When the function returns, `fromlen` will contain the length of the address actually stored in `from`. `recvfrom()` returns the number of bytes received, or `-1` on error (with `errno` set accordingly).
- Remember, if you `connect()` a datagram socket, you can then simply use `send()` and `recv()` for all your transactions.
- The socket itself is still a datagram socket and the packets still use UDP, but the socket interface will automatically add the destination and source information for you.

Sample of the client socket call flow

```
socket()
connect()
while (x)
{
   write()
   read()
}
close()
```

Sample of the server socket call flow

```
socket()
bind()
listen()
while (1)
{
   accept()
   while (x)
{
      read()
      write()
}
   close()
}
close()
```

Network Integers versus Host Integers

- Little Endian and big Endian issue regarding the use of the different processors.
- Usually integers are either most-significant byte first or least-significant byte first.
- On Intel based machines the hex value 0x01020304 would be stored in 4 successive bytes as: 04, 03, 02, 01. This is little endian.
- On an Most Significant Bit (MSB)-first (big endian) machine (IBM RS6000), this would be: 01, 02, 03, 04.
- It is important to use network byte order (MSB-first) and the conversion functions are listed below:

`htons()`	Host to network short.
`ntohs()`	Network to host short.
`htonl()`	Host to network long.
`ntohl()`	Network to host long.

Table 7

- Use these functions to write portable network code.
- Fortunately for you, there are a bunch of functions that allow you to manipulate IP addresses. No need to figure them out by hand and stuff them in a `long` with the « operator.
- First, let's say you have a:

```
struct sockaddr_in ina
```

- And you have an IP address "`10.12.110.57`" that you want to store into it.
- The function you want to use, `inet_addr()`, converts an IP address in numbers-and-dots notation into an unsigned long. The assignment can be made as follows:

```
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

- Notice that `inet_addr()` returns the address in Network Byte Order already so you don't have to call `htonl()`.
- Now, the above code snippet isn't very robust because there is no error checking. `inet_addr()` returns `-1` on error.
- For binary numbers `(unsigned)-1` just happens to correspond to the IP address `255.255.255.255`! That's the broadcast address! Remember to do your error checking properly.
- Actually, there's a cleaner interface you can use instead of `inet_addr()`: it's called `inet_aton()` ("`aton`" means "ascii to network"):

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
```

- And here's a sample usage, while packing a `struct sockaddr_in` is shown below:

```
struct sockaddr_in my_addr;
/* host byte order */
my_addr.sin_family = AF_INET;
/* short, network byte order */
my_addr.sin_port = htons(MYPORT);
inet_aton("10.12.110.57", &(my_addr.sin_addr));
/* zero the rest of the struct */
memset(&(my_addr.sin_zero), 0, 8);
```

- `inet_aton()`, unlike practically every other socket-related function, returns non-zero on success, and zero on failure. And the address is passed back in `inp`.
- Unfortunately, not all platforms implement `inet_aton()` so, although its use is preferred, normally the older more common `inet_addr()` is used.
- All right, now you can convert string IP addresses to their binary representations. What about the other way around?
- What if you have a `struct in_addr` and you want to print it in numbers-and-dots notation? In this case, you'll want to use the function `inet_ntoa()` ("`ntoa`" means "network to ascii") something like this:

```
printf("%s", inet_ntoa(ina.sin_addr));
```

- That will print the IP address. Note that `inet_ntoa()` takes a `struct in_addr` as an argument, not a `long`. Also notice that it returns a pointer to a char.
- This points to a statically stored char array within `inet_ntoa()` so that each time you call `inet_ntoa()` it will overwrite the last IP address you asked for. For example:

```
char *a1, *a2;
.
.
a1 = inet_ntoa(ina1.sin_addr); /* this is 192.168.4.1 */
a2 = inet_ntoa(ina2.sin_addr); /* this is 10.11.110.55 */
printf("address 1: %s\n", a1);
printf("address 2: %s\n", a2);
```

- Will print:

```
address 1: 10.11.110.55
address 2: 10.11.110.55
```

- If you need to save the address, `strcpy()` it to your own character array.

---**SOME SUMMARY**---

- Let see, what we have covered till now.

Socket Library Functions

- System calls:
  - Startup / close.
  - Data transfer.
  - Options control.
  - Other.
- Network configuration lookup:
  - Host address.
  - Ports for services.
  - Other.
- Utility functions:
  - Data conversion.
  - Address manipulation.
  - Error handling.

Primary Socket Calls

`socket()`	Create a new socket and return its descriptor.

bind()	Associate a socket with a port and address.
listen()	Establish queue for connection requests.
accept()	Accept a connection request.
connect()	Initiate a connection to a remote host.
recv()	Receive data from a socket descriptor.
send()	Send data to a socket descriptor.
read()	Reads from files, devices, sockets etc.
write()	Writes to files, devices, sockets etc.
close()	"One-way" close of a socket descriptor.
shutdown()	Allows you to cut off communication in a certain direction, or both ways just like close() does.

Table 8

Network Database Administration functions

- gethostbyname() - given a hostname, returns a structure which specifies its DNS name(s) and IP address (es).
- getservbyname() - given service name and protocol, returns a structure which specifies its name(s) and its port address.
- gethostname() - returns hostname of local host.
- getservbyname(), getservbyport(), getservent().
- getprotobyname(), getprotobynumber(), getprotobyent(), getnetbyname(), getnetbyaddr(), getnetent().

Socket Utility Functions

ntohs()/ntohl()	Convert short/long from network byte order (big endian) to host byte order.
htons()/htonl()	Convert short/long from host byte order to network byte order.
inet_ntoa()/inet_addr()	Convert 32-bit IP address (network byte order to/from a dotted decimal string).
perror()	Print error message (based on "errno") to stderr.
herror()	Print error message for gethostbyname() to stderr (used with DNS).

Table 9

Primary Header Files

- Include file sequence may affect processing (order is important!). Other header files that define macro, data type, structure and functions are given in the summary Table at the end of this Tutorial.

<sys/types.h>	Prerequisite typedefs.
<errno.h>	Names for "errno" values (error numbers).
<sys/socket.h>	struct sockaddr; system prototypes and constants.
<netdb.h.h>	Network info lookup prototypes and structures.
<netinet/in.h>	struct sockaddr_in; byte ordering macros.
<arpa/inet.h>	Utility function prototypes.

Table 10

Ancillary Socket Topics

- UDP versus TCP.
- Controlling/managing socket characteristics.
    - get/setsockopt() - keepalive, reuse, nodelay.
    - fcntl() - async signals, blocking.
    - ioctl() - file, socket, routing, interface options.
- Blocking versus Non-blocking socket.
- Signal based socket programming (SIGIO).
- Implementation specific functions.

--End socket Part I---

Socket header files

- Programs that use the socket functions must include one or more header files that contain information that is needed by the functions, such as:
    - Macro definitions.
    - Data type definitions.
    - Structure definitions.
    - Function prototypes.
- The following Table is a summary of the header files used in conjunction with the socket APIs.

Header file name	Description
`<arpa/inet.h>`	Defines prototypes for those network library routines that convert Internet address and dotted-decimal notation, for example, `inet_makeaddr()`.
`<arpa/nameser.h>`	Defines Internet name server macros and structures that are needed when the system uses the resolver routines.
`<error.h>`	Defines macros and variables for error reporting.
`<fcntl.h>`	Defines prototypes, macros, variables, and structures for control-type functions, for example, `fcntl()`.
`<net/if.h>`	Defines prototypes, macros, variables, and the `ifreq` and `ifconf` structures that are associated with `ioctl()` requests that affect interfaces.
`<net/route.h>`	Defines prototypes, macros, variables, and the `rtentry` and `rtconf` structures that are associated with `ioctl()` requests that affect routing entries.
`<netdb.h>`	Contains data definitions for the network library routines. Defines the following structures: ▪ `hostent` and `hostent_data`. ▪ `netent` and `netent_data`. ▪ `servent` and `servent_data`. ▪ `protoent` and `protoent_data`.
`<netinet/in.h>`	Defines prototypes, macros, variables, and the `sockaddr_in` structure to use with Internet domain sockets.
`<netinet/ip.h>`	Defines macros, variables, and structures that are associated with setting IP options.
`<netinet/ip_icmp.h>`	Defines macros, variables, and structures that are associated with the Internet Control Message Protocol (ICMP).
`<netinet/tcp.h>`	Defines macros, variables, and structures that are associated with setting TCP options.
`<netns/idp.h>`	Defines IPX packet header. May be needed in `AF_NS` socket applications.
`<netns/ipx.h>`	Defines `ioctl` structures for IPX `ioctl()` requests. May be needed in `AF_NS` socket applications.
`<netns/ns.h>`	Defines `AF_NS` socket structures and options. You must include this file in `AF_NS` socket applications.
`<netns/sp.h>`	Defines SPX packet header. May be needed in `AF_NS` socket applications.
`<nettel/tel.h>`	Defines `sockaddr_tel` structure and related structures and macros. You must include this file in `AF_TELEPHONY` socket applications.
`<resolv.h>`	Contains macros and structures that are used by the resolver routines.
`<ssl.h>`	Defines Secure Sockets Layer (SSL) prototypes, macros, variables, and the following structures: ▪ SSLInit ▪ SSLHandle
`<sys/ioctl.h>`	Defines prototypes, macros, variables, and structures for I/O control-type functions, for example, `ioctl()`.
`<sys/param.h>`	Defines some limits to system fields, in addition to miscellaneous macros and prototypes.
`<sys/signal.h>`	Defines additional macros, types, structures, and functions that are used by signal routines.
`<sys/socket.h>`	Defines socket prototypes, macros, variables, and the following structures:

	▪ sockaddr ▪ msghdr ▪ linger You must include this file in all socket applications.
<sys/time.h>	Defines prototypes, macros, variables, and structures that are associated with time functions.
<sys/types.h>	Defines various data types. Also includes prototypes, macros, variables, and structures that are associated with the select() function. You must include this file in all socket applications.
<sys/uio.h>	Defines prototypes, macros, variables, and structures that are associated with I/O functions.
<sys/un.h>	Defines prototypes, macros, variables, and the sockaddr_un structure to use with UNIX domain sockets.
<unistd.h>	Contains macros and structures that are defined by the integrated file system. Needed when the system uses the read() and write() system functions.

Table 11: Header files for the sockets APIs

*...Continue on next Module...More program examples...*More in-depth discussion about TCP/IP suite is given in Module 42.

---------------------------------------End Part I-------------------------------------

----www.tenouk.com---

Further reading and digging:

1. Check the best selling C/C++, Networking, Linux and Open Source books at Amazon.com.
2. Protocol sequence diagram examples.
3. Another site for protocols information.
4. RFCs.
5. External Data Representation (XDR).
6. Remote Procedure Call (RPC).