COMPUTER NETWORKS LAB (U21IT6L2)

III-Year II Semester

LAB MANUAL

Prepared By

Ms. Hajira Sabuhi

Assistant Professor

Ms. Khutaija Abid

Assistant Professor

LORDS INSTITUTE OF ENGINEERING AND TECHNOLOGY
Survey No.32, Himayath Sagar, near TSPA Junction
Hyderabad-500091

Department of Information Technology

S.NO	CONTENTS	PAGE NO.
1.	Institute Vision & Mission	i
2.	Department Vision & Mission	ii
3.	PEOs	iii
4.	POs	iv
5.	PSOs	v
6.	COs	vi
7.	CO-PO Mapping	vii
	Programs	
1	Familiarization of Network Environment, Understanding and using network utilities: ipconig, ifconfig, netstat, ping, arp, telnet, ftp, finger, traceroute	1
2	Write a program to implement connection oriented and connection less client for well-known services i.e., standard ports	3
3	Implementation of concurrent server service using connection oriented socket system calls (Service: Daytime)	5
4	Implementation of concurrent server using connection less socket system calls.(Service: Echo Server)	8
5	Write a programs for TCP chat server	12
6	Write a programs for UDP chat server	15
7	Program to demonstrate the use of advanced socket sytem calls: readv(), writev(),getsockname(),setsockname(),getpeername(),gethostbyname(),gethostbyaddr(),getnetbyname(),getnetbyaddr(), getprotobyname(), getservbyname().	17

Institute Vision

Lords Institute of Engineering and Technology strives continuously for excellence in professional education through quality, innovation and teamwork and to emerge as a premier institute in the state and across the nation.

Institute Mission

- To impart quality professional education that meets the needs of present and emerging technological world.
- To strive for student achievement and success, preparing them for life, career and leadership.
- To provide a scholarly and vibrant learning environment that enables faculty, staff and students to achieve personal and professional growth.
- To contribute to advancement of knowledge, in both fundamental and applied areas of engineering and technology.
- To forge mutually beneficial relationships with government organizations, industries, society and the alumni.

Department vision

To be a frontier in Information Technology focusing on quality professional education, innovation and employability.

Department Mission

DM1: Adopt innovative teaching learning processes to achieve stated outcomes and sustained performance levels.

DM2: Provide learner centric environment to enrich knowledge, skills and innovation through industry academia collaborations.

DM3: Train the concerned stakeholders on latest technologies to meet the industry needs.

DM4: Inculcate leadership, professional, interpersonal skills and ethical values to address the contemporary societal issues.

Note: **DM**: Department Mission

Program Educational Objectives (PEOs):

PEO1: Be a competent software engineer in IT and allied industry providing viable solutions.

PEO2: Exhibit professionalism, engage in lifelong learning and adopt to changing industry and societal needs.

PEO3: Be effective as an individual and also cohesive in a group to execute multidisciplinary projects.

PEO4: Pursue higher studies in Information Technology with emphasis on competency and innovation.

Program Outcomes

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes

PSO1: Professional Skills: Identify, analyze requirements, design and implement computer programs in the area of artificial intelligence, machine learning, big data analytics and networking of varying complexity.

PSO2: Problem-Solving Skills: Deliver quality service to IT industry and provide solutions to real-life problems applying modern computer languages, software methods and tools.

Course Outcomes

1. Understand the usage of basic commands ipconig, ifconfig, netstat, ping, arp, telnet, ftp, finger, trace route, who is of LINUX platform.
2. Develop and Implement Client-Server Socket based programs using TCP, and UDP sockets.
3. To make a client server communication through TCP and UDP protocols.
4. To expose on advanced socket programming in LINUX environment.
5. Understanding of transport layer protocols : connection oriented and connection-less models, techniques to provide reliable data delivery.

CO-PO MAPPING

Course Outcomes (CO)	**Program Outcomes (PO)**												**Program Specific Outcomes (PSO's)**	
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO1 1**	**PO 12**	**PSO 1**	**PSO 2**
C367.1				3						3		3	3	3
C367.2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
C367.3		2	2	2	2				2	2	2	2	2	2
C367.4		2	2	2	2	2			2	2	2	2	2	2
C367.5		1	1	1	1				1	1	1	1	1	1
Avg. C367	**2**	**1.75**	**1.75**	**2**	**1.75**	**2**	**2**	**2**	**1.75**	**2**	**1.75**	**2**	**2**	**2**

Introduction to Socket Programming

Outline:

1.) Introduction

2.) The Client / Server Model

3.) The Socket Interface and Features of a TCP connection 4.) Byte Ordering

5.) Address Structures, Ports, Address conversion functions

6.) Outline of a TCP Server

7.) Outline of a TCP Client

8.) Client-Server communication outline

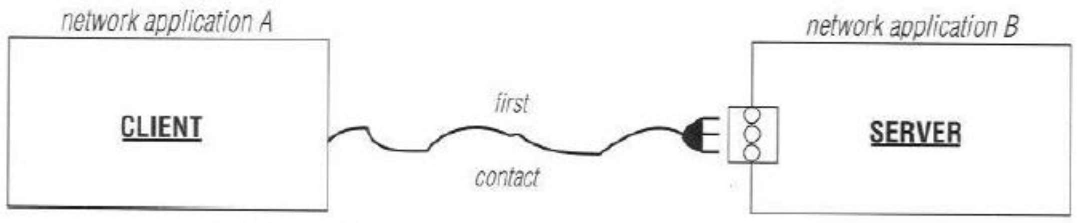
9.) Summary of Socket Functions

1.) Introduction

In this Lab you will be introduced to socket programming at a very elementary level. Specifically, we will focus on TCP socket connections which are a fundamental part of socket programming since they provide a connection oriented service with both flow and congestion control. What this means to the programmer is that a TCP connection provides a reliable connection over which data can be transferred with little effort required on the programmers part; TCP takes care of the reliability, flow control, congestion control for you. First the basic concepts will be discussed, then we will learn how to implement a simple TCP client and server.

2.) The Client / Server Model

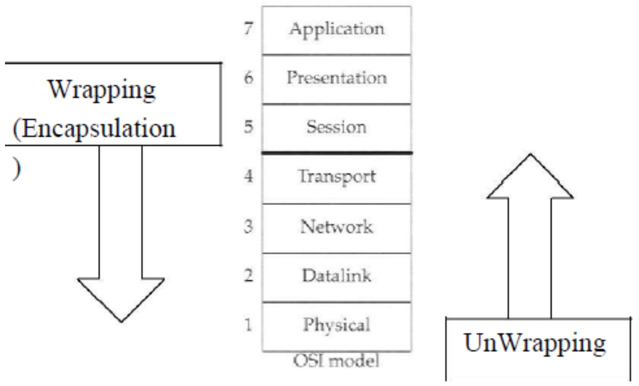
It is possible for two network applications to begin simultaneously, but it is impractical to require it. Therefore, it makes sense to design communicating network applications to perform complementary network operations in sequence, rather than simultaneously. The server executes first and waits to receive; the client executes second and sends the first network packet to the server. After initial contact, either the client or the server is capable of sending and receiving data.



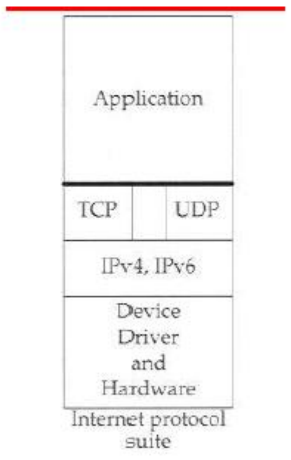
A client initiates communications to a server.

3.) The Socket Interface and Features of a TCP connection

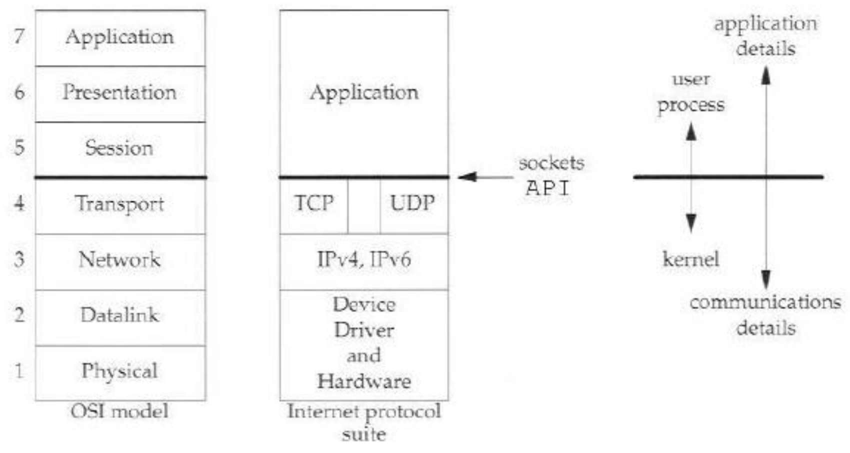
The OSI Layers:



The Internet Layers:



The Internet does not strictly obey the OSI model but rather merges several of the protocols layers together. Where is the socket programming interface in relation to the protocol stack? Features of a TCP connection:



Features of a TCP connection:

1.Connection Oriented	
2. Reliability	
i	Handles lost packets
ii.	Handles packet sequencing
iii.	Handles duplicated packets
3. Full Duplex	
4. Flow Control	
5. Congestion Control	

TCP versus UDP as a Transport Layer Protocol:

TCP	UDP
Reliable, guaranteed packets.	Unreliable. Instead, prompt delivery of packets
Connection-oriented	Connectionless
Used in applications that require safety gurantee. (eg. file applications.)	Used in media applications. (eg. video or voice transmissions.)
Flow control, sequencing of packets, error-control.	No flow or sequence control, user must handle these manually.
Uses byte stream as unit of transfer. (stream sockets)	Uses datagrams as unit of transfer. (datagram sockets)
Allows to send multiple packets with a single ACK.	

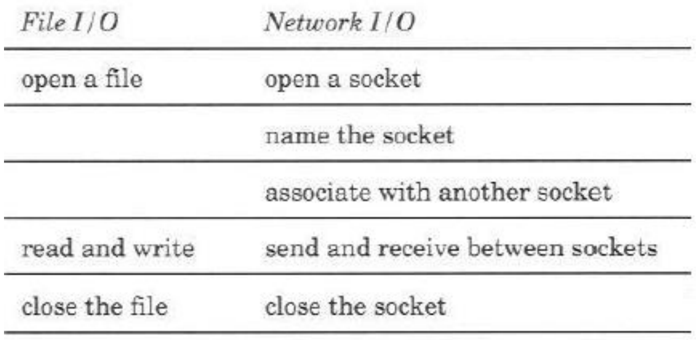
Allows two-way data exchange, once the connection is established. (full-duplex)	Allows data to be transferred in one direction at once. (half-duplex)
e.g. Telnet uses stream sockets.	e.g. TFTP (trivial file transfer protocol) uses
(everything you write on one side appears exaclt in same order on the other side)	datagram sockets.

Sockets versus File I/O

Working with sockets is very similar to working with files. The socket() and accept() functions both return handles (file descriptor) and reads and writes to the sockets requires the use of these handles (file descriptors).

In Linux, sockets and file descriptors also share the same file descriptor table. That is, if you open a file and it returns a file descriptor with value say 8, and then immediately open a socket, you will be given a file descriptor with value 9 to reference that socket. Even though sockets and files share the same file descriptor table, they are still very different.

Sockets have addresses associated with them whereas files do not, notice that this distinguishes sockets form pipes, since pipes do not have addresses with which they associate. You cannot randomly access a socket like you can a file with lseek(). Sockets must be in the correct state to perform input or output.



File I/O	*Network I/O*
open a file	open a socket
	name the socket
	associate with another socket
read and write	send and receive between sockets
close the file	close the socket

4.) Byte Ordering

Port numbers and IP Addresses (both discussed next) are represented by multi-byte data types which are placed in packets for the purpose of routing and multiplexing. Port numbers are two bytes (16 bits) and IP4 addresses are 4 bytes (32 bits), and a problem arises when transferring multi-byte data types between different architectures.

Say Host A uses a "*big-endian*" architecture and sends a packet across the network to Host B which uses a "*little-endian*" architecture. If Host B looks at the address to see if the packet is for him/her (choose a gender!), it will interpret the bytes in the opposite order and will wrongly conclude that it is not his/her packet.

The Internet uses big-endian and we call it the *network-byte-order*, and it is really not important to know which method it uses since we have the following functions to convert host-byte-ordered values into network-byte-ordered values and vice versa:

5.) Address Structures, Ports, Address conversion functions

Overview of IP4 addresses:

IP4 addresses are 32 bits long. They are expressed commonly in what is known as dotted decimal notation. Each of the four bytes which makes up the 32 address are expressed as an integer value (0 – 255) and separated by a dot.

For example, 138.23.44.2 is an example of an IP4 address in dotted decimal notation. There are conversion functions which convert a 32 bit address into a dotted decimal string and vice versa which will be discussed later.

Often times though the IP address is represented by a domain name, for example, hill.ucr.edu. Several functions described later will allow you to convert from one form to another (Magic provided by DNS!).

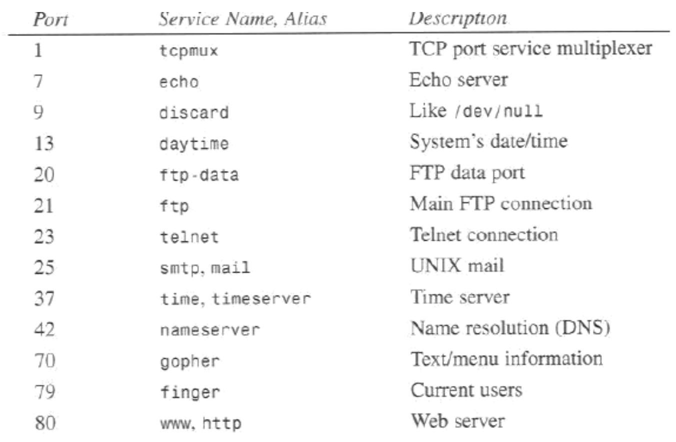
The importance of IP addresses follows from the fact that each host on the Internet has a unique IP address. Thus, although the Internet is made up of many networks of networks with many different types of architectures and transport mediums, it is the IP address which provides a cohesive structure so that at least theoretically, (there are routing issues involved as well), any two hosts on the Internet can communicate with each other.

Ports:

Sockets are UNIQUELY identified by Internet address, end-to-end protocol, and port number. That is why when a socket is first created it is vital to match it with a valid IP address and a port number. In our labs we will basically be working with TCP sockets.

Ports are software objects to multiplex data between different applications. When a host receives a packet, it travels up the protocol stack and finally reaches the application layer. Now consider a user running an ftp client, a telnet client, and a web browser concurrently. To which application should the packet be delivered? Well part of the packet contains a value holding a port number, and it is this number which determines to which application the packet should be delivered.

So when a client first tries to contact a server, which port number should the client specify? For many common services, standard port numbers are defined.



Port	Service Name, Alias	Description
1	tcpmux	TCP port service multiplexer
7	echo	Echo server
9	discard	Like /dev/null
13	daytime	System's date/time
20	ftp-data	FTP data port
21	ftp	Main FTP connection
23	telnet	Telnet connection
25	smtp, mail	UNIX mail
37	time, timeserver	Time server
42	nameserver	Name resolution (DNS)
70	gopher	Text/menu information
79	finger	Current users
80	www, http	Web server

Ports 0 – 1023, are reserved and servers or clients that you create will not be able to **bind** to these ports unless you have root privilege.

Ports 1024 – 65535 are available for use by your programs, but beware other network applications maybe running and using these port numbers as well so do not make assumptions about the availability of specific port numbers. Make sure you read Stevens for more details about the available range of port numbers!

Address Structures:

Socket functions like connect(), accept(), and bind() require the use of specifically defined address structures to hold IP address information, port number, and protocol type. This can be one of the more confusing aspects of socket programming so it is necessary to clearly understand how to use the socket address structures. The difficulty is that you can use sockets to program network applications using different protocols. For example, we can use IP4, IP6, Unix local, etc.

Here is the problem: Each different protocol uses a different address structure to hold its addressing information, yet they all use the same functions connect(), accept(), bind() etc. So how do we pass these different structures to a given socket function that requires an address structure? Well it may not be the way you would think it should be done and this is because sockets where developed a long time ago before things like a void pointer where features in C. So this is how it is done:

There is a generic address structure: struct sockaddr

This is the address structure which must be passed to all of the socket functions requiring an address structure.
This means that you must type cast your specific protocol dependent address structure to the generic address structure when passing it to these socket functions.

Protocol specific address structures usually start with *sockaddr_* and end with a ***suffix*** depending on that protocol. For example:

struct sockaddr_in (IP4, think of in as internet)

struct sockaddr_in6 (IP6)
struct sockaddr_un (Unix local)
struct sockaddr_dl (Data link)

We will be only using the IP4 address structure: struct sockaddr_in.

So once we fill in this structure with the IP address, port number, etc we will pass this to one of our socket functions and we will need to type cast it to the generic address structure. For example:
struct sockaddr_in myAddressStruct;

//Fill in the address information into myAddressStruct here, (will be explained in detail shortly)

connect(socket_file_descriptor, (struct sockaddr *) &myAddressStruct, sizeof(myAddressStruct));

```
struct sockaddr_in{

        sa_family_t  sin_family          /*Address/Protocol Family*/ (we'll use PF_INET)
        unit16_t          sin_port       /* 16-bit Port number         --Network Byte Ordered--
*/
        struct in_addr       sin_addr    /*A struct for the 32         bit IP Address  */
        unsigned char sin_zero[8]        /*Just ignore this it         is just padding*/
};

struct in_addr{
        unit32_t          s_addr   /*32 bit IP Address   --Network Byte Ordered-- */
};
```

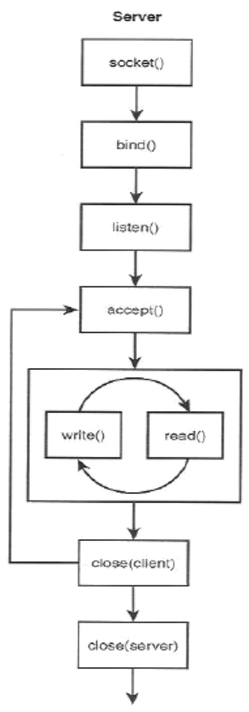
For the sa_family variable sin_family always use the constant: PF_INET or AF_INET
***Always initialize address structures with bzero() or memset() before filling them in ***
***Make sure you use the byte ordering functions when necessary for the port and IP address variables otherwise there will be strange things a happening to your packets.

To convert a string dotted decimal IP4 address to a NETWORK BYTE ORDERED 32 bit value use the functions:
• inet_addr()

• inet_aton()

To convert a 32 bit NETWORK BYTE ORDERED to a IP4 dotted decimal string use:
• inet_ntoa()

6.) Outline of a TCP Server:



Step 1: Creating a socket:

int socket(int family, int type, int protocol);

Creating a socket is in some ways similar to opening a file. This function creates a file descriptor and returns it from the function call. You later use this file descriptor for reading, writing and using with other socket functions

Parameters:

family: AF_INET or PF_INET (These are the IP4 family)
type: SOCK_STREAM (for TCP) or SOCK_DGRAM (for UDP)
protocol: IPPROTO_TCP (for TCP) or IPPROTO_UDP (for UDP) or use 0

Step 2:Binding an address and port number

int bind(int socket_file_descriptor, const struct sockaddr * LocalAddress, socklen_t AddressLength);

We need to associate an IP address and port number to our application. A client that wants to connect to our server needs both of these details in order to connect to our server. Notice the difference between this function and the connect() function of the client. The connect function specifies a remote address that the client wants to connect to, while here, the server is specifying to the bind function a local IP address of one of its Network Interfaces and a local port number.

The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *bind()* is 0 for success and –1 for failure.

**Again make sure that you cast the structure as a generic address structure in this function **
You also do not need to find information about the IP addresses associated with the host you are working on. You can specify: INNADDR_ANY to the address structure and the bind function will use on of the available (there may be more than one) IP addresses. This ensures that connections to a specified port will be directed to this socket, regardless of which Internet address they are sent to. This is useful if host has multiple IP addresses, then it enables the user to specify which IP address will be b_nded to which port number.

Step 3:Listen for incoming connections

Binding is like waiting by a specific phone in your house, and Listening is waiting for it to ring.
int listen(int socket_file_descriptor, int backlog);
The backlog parameter can be read in Stevens' book. It is important in determining how many connections the server will connect with. Typical values for backlog are 5 – 10.
The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *listen()* is 0 for success and –1 for failure.

Step 4: Accepting a connection.

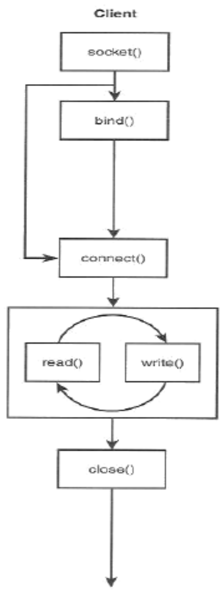
int accept (int socket_file_descriptor, struct sockaddr * ClientAddress, socklen_t *addrlen);

accept() returns a new socket file descriptor for the purpose of reading and writing to the client. The original file descriptor is used usually used for listening for new incoming connections. Servers will be discussed in much more detail in a later lab.

It dequeues the next connection request on the queue for this socket of the server. If queue is empty, this function blocks until a connection request arrives**Again, make sure you type cast to the generic socket address structure**

Note that the last parameter is a pointer. You are not specifying the length, the kernel is and returning the value to your application, the same with the ClientAddress. After a connection with a client is established the address of the client must be made available to your server, otherwise how could you communicate back with the client? Therefore, the accept() function call fills in the address structure and length of the address structure for your use. Then accept() returns a new file descriptor, and it is this file descriptor with which you will read and write to the client.

7.) Outline of a TCP Client



Step 1: Create a socket : Same as in the server.

Step 2: Binding a socket:

This is unnecessary for a client, what bind does is (and will be discussed indetail in the server section) is associate a port number to the application. If you skip this step with a TCP client, a temporary port number is automatically assigned, so it is just better to skip this step with the client.

Step 3: Connecting to a Server:

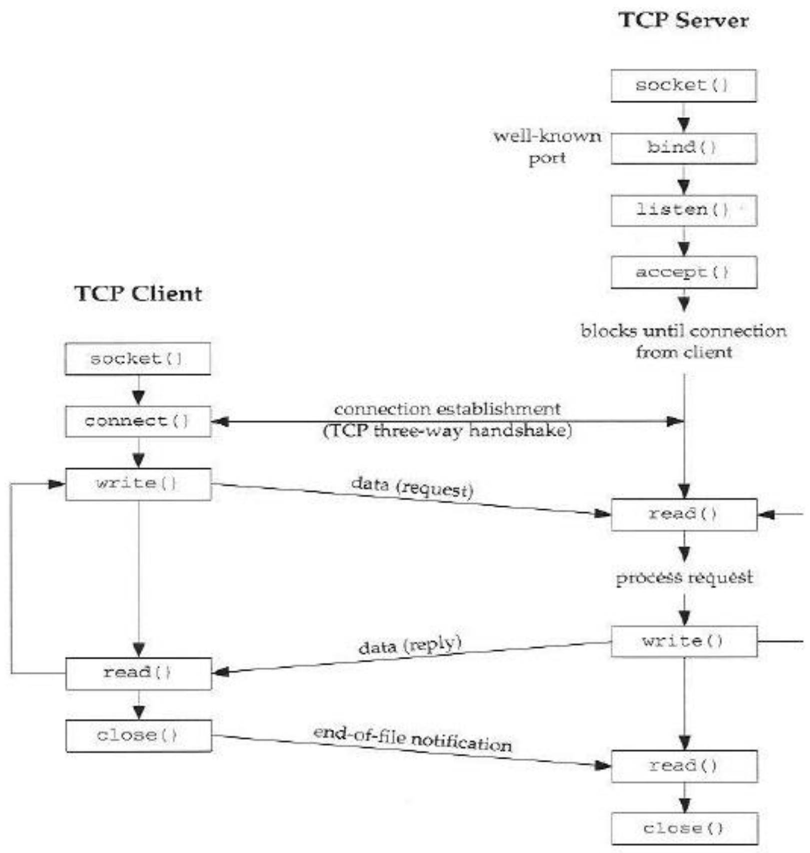
int connect(int socket_file_descriptor, const struct sockaddr *ServerAddress, socklen_t AddressLength);

Once you have created a socket and have filled in the address structure of the server you want to connect to, the next thing to do is to connect to that server. This is done with the connect function listed above.
**This is one of the socket functions which requires an address structure so remember to type cast it to the generic socket structure when passing it to the second argument **
Connect performs the three-way handshake with the server and returns when the connection is established or an error occurs.
Once the connection is established you can begin reading and writing to the socket.

Step 4: Read and Writing to the socket will be discussed shortly

Step 5: Closing the socket will be discussed shortly

8.) Outline of a client-server network interaction:



Communication of 2 pairs via sockets necessitates existence of this 4-tuple:
- Local IP address
- Local Port#

- Foreign IP address
- Foreign Port#

!!!! When a server receives (accepts) the client's connection request => it *fork*s a copy of itself and lets the child handle the client. (make sure you remember these Operating Systems concepts)

Therefore on the server machine, listening socket is distinct from the connected socket.
read/write:These are the same functions you use with files but you can use them with sockets as well. However, it is extremely important you understand how they work so please read Stevens carefully to get a full understanding.

Writing to a socket:

int write(int file_descriptor, const void * buf, size_t message_length);

The return value is the number of bytes written, and –1 for failure. The number of bytes written may be less than the message_length. What this function does is transfer the data from you application to a buffer in the kernel on your machine, it does not directly transmit the data over the network. This is extremely important to understand otherwise you will end up with many headaches trying to debug your programs. TCP is in complete control of sending the data and this is implemented inside the kernel. Due to network congestion or errors, TCP may not decide to send your data right away, even when the function call returns. TCP has an elaborate sliding window mechanism which you will learn about in class to control the rate a which data is sent.

Reading from a socket:

int read(int file_descriptor, char *buffer, size_t buffer_length);

The value returned is the number of bytes read which may not be buffer_length! It returns –1 for failure. As with write(), read() only transfers data from a buffer in the kernel to your application , you are not Directly reading the byte stream from the remote host, but rather TCP is in control and buffers the data for your application.

Shutting down sockets:

After you are finished reading and writing to your socket you most call the close system call on the socket file descriptor just as you do on a normal file descriptor otherwise you waste system resources.

The close() function:

int close(int filedescriptor);

The shutdown() function: You can also shutdown a socket in a partial way which is often used when Forking off processes. You can shutdown the socket so that it won't send anymore or you could also shutdown the socket so that it won't read anymore as well.
You can look at the man pages for a full description of this function.

12.) Summary of Functions
For specific and up-to-date information about each of the following functions, please use the online man pages and Steven's Unix Network Programming Vol. I.

Socket creation and destruction:
• socket()

• close()

• shutdown()

Client:
• connect()

• bind()

Server:

• accept()

• bind()

• listen()

Data Transfer:
• send()
• recv()
• write()
• read()

Miscellaneous:
• bzero()

• memset()

Host Information:
• uname()

• gethostbyname()

• gethostbyaddr()

Address Conversion:
• inet_aton()

• inet_addr()

• inet_ntoa()

PROGRAM 1:
Familiarization of Network Environment, Understanding and using network utilities: ipconig, ifconfig,netstat, ping,arp,telnet, ftp,finger, traceroute,whois.

IFCONFIG

`ifconfig`, short for "interface configuration", is a command-line utility used in Unix-like operating systems to configure, manage, and query network interface parameters. Here's a brief explanation of its usage:

- Viewing Interface Information: Running `ifconfig` without any arguments displays information about all active network interfaces on the system, including their IP addresses, netmasks, MAC addresses, transmission and reception statistics, and more.

- Configuring Interfaces: `ifconfig` can be used to configure network interfaces by specifying options like IP address, netmask, broadcast address, and other parameters. For example, you can set up a static IP address for an interface.

- Bringing Interfaces Up or Down: `ifconfig` can bring network interfaces up or down. For instance, you can enable or disable network connectivity for a specific interface using `ifconfig`.

- Diagnostic Tool: It's also used as a diagnostic tool to troubleshoot network connectivity issues by examining interface configurations and statistics.

Parameters

Parameters used with this command must be prefixed with a hyphen (-) rather than a slash (/).

-a : Displays **a**ll active TCP connections and the TCP and UDP ports on which the computer islistening.
-e : Displays **e**thernet statistics, such as the number of bytes and packets sent and received. Thisparameter can be combined with **-s**.

-f : Displays **f**ully qualified domain names <FQDN> for foreign addresses.

-i : Displays network **i**nterfaces and their statistics (not available under Windows)

-n : Displays active TCP connections, however, addresses and port numbers are expressed numericallyand no attempt is made to determine names.
-o : Displays active TCP connections and includes the process ID (PID) for each connection.

-p Linux: **P**rocess : Show which processes are using which sockets

```
ab@lab-V520-15IKL:~/Desktop$ ifconfig
```

OUTPUT

```
enp1s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
```

```
        inet 172.23.111.206  netmask 255.255.0.0  broadcast 172.23.255.255
        inet6 fe80::6210:a84d:878:7702  prefixlen 64  scopeid 0x20<link>
        ether 6c:4b:90:49:68:3d  txqueuelen 1000  (Ethernet)
        RX packets 81034  bytes 88983388 (88.9 MB)
        RX errors 0  dropped 69  overruns 0  frame 0
        TX packets 40659  bytes 6950281 (6.9 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 1806  bytes 208731 (208.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1806  bytes 208731 (208.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

EXPLANATION

Sure, let's break down each line from the network interface configuration:

1. `flags=4163<UP,BROADCAST,RUNNING,MULTICAST>`:
   - `UP`: Indicates that the interface is up and operational.
   - `BROADCAST`: Indicates that the interface supports broadcast transmissions.
   - `RUNNING`: Indicates that the interface is operational.
   - `MULTICAST`: Indicates that the interface supports multicast transmissions.

2. `mtu 1500`:
   - `mtu`: Stands for Maximum Transmission Unit, which is the maximum size of a data packet that can be transmitted over the network. In this case, it's set to 1500 bytes.

3. `inet 172.23.111.206  netmask 255.255.0.0  broadcast 172.23.255.255`:
   - `inet`: Stands for Internet Protocol version 4 (IPv4) address. In this case, the interface has been assigned the IPv4 address `172.23.111.206`.
   - `netmask`: Specifies the subnet mask associated with the IPv4 address. Here, it's `255.255.0.0`, indicating a subnet of size 16 bits.
   - `broadcast`: Specifies the broadcast address for the network. In this case, it's `172.23.255.255`.

4. `inet6 fe80::6210:a84d:878:7702  prefixlen 64  scopeid 0x20<link>`:
   - `inet6`: Stands for Internet Protocol version 6 (IPv6) address. Here, the interface has been assigned the IPv6 address `fe80::6210:a84d:878:7702`.
   - `prefixlen`: Indicates the length of the network prefix in bits. In this case, it's `64`.
   - `scopeid`: Specifies the scope of the address. In this case, it's `0x20<link>`, indicating a link-local address.

5. `ether 6c:4b:90:49:68:3d  txqueuelen 1000  (Ethernet)`:
  - `ether`: Indicates the hardware (MAC) address of the interface. Here, it's `6c:4b:90:49:68:3d`.
  - `txqueuelen`: Stands for Transmit Queue Length. It represents the length of the transmit queue for this interface. In this case, it's `1000`.
  - `(Ethernet)`: Indicates the type of interface, which is Ethernet.

6. `RX packets 81034  bytes 88983388 (88.9 MB)`:
  - `RX`: Stands for Receive. Indicates the number of packets received (`81034`) and the total number of bytes received (`88983388` bytes or `88.9 MB`).

7. `RX errors 0  dropped 69  overruns 0  frame 0`:
  - Indicates statistics related to received packets: `RX errors` (number of receive errors), `dropped` (number of packets dropped), `overruns` (number of packet overruns), and `frame` (number of frame errors).

8. `TX packets 40659  bytes 6950281 (6.9 MB)`:
  - `TX`: Stands for Transmit. Indicates the number of packets transmitted (`40659`) and the total number of bytes transmitted (`6950281` bytes or `6.9 MB`).

9. `TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0`:
- Indicates statistics related to transmitted packets: `TX errors` (number of transmit errors), `dropped` (number of packets dropped), `overruns` (number of packet overruns), `carrier` (number of carrier errors), and `collisions` (number of collisions).

The **netmask,** short for network mask, is a 32-bit number that is used to divide an IP address into network and host portions. It is used in conjunction with an IP address to determine the network to which the address belongs.

NETSTAT

`netstat` is a command-line utility used to display network-related information on Unix-like operating systems. Here's a brief overview of its functionality:

- Viewing Network Connections: `netstat` can display a list of all network connections, both incoming and outgoing, along with their associated protocol, local and remote addresses, and state.

- Displaying Routing Table: It can also show the routing table of the system, including information about the network interfaces and their associated routes.

- Monitoring Network Statistics: `netstat` can be used to monitor various network statistics, such as the number of packets and bytes transmitted and received, error counts, and more.

- Displaying Interface Information: It provides detailed information about network interfaces, including their configuration, status, and statistics.

- Displaying Multicast Memberships: `netstat` can display information about multicast group memberships for IPv4 and IPv6.

- Displaying Network Services: With appropriate options, `netstat` can show information about network services and their associated ports.

lab@lab-V520-15IKL:~/Desktop$ netstat

Each line in the provided output represents a Unix domain socket or communication endpoint. Here's an explanation of each field and what it represents:

1. `unix  3      [ ]         STREAM     CONNECTED     30607    /run/user/998/bus`
   - `unix`: Indicates that it's a Unix domain socket.
   - `3`: Represents the socket type, which is a stream socket.
   - `[ ]`: Indicates the socket's status. In this case, it's not specified.
   - `STREAM`: Indicates the type of socket, which is a stream socket used for connection-oriented communication.
   - `CONNECTED`: Indicates that the socket is currently connected.
   - `30607`: Represents the process identifier (PID) or identifier of the connected endpoint.
   - `/run/user/998/bus`: Specifies the endpoint to which this socket is connected.

2. `unix  3      [ ]         STREAM     CONNECTED     31532    /run/user/998/bus`
   - Similar to the first line, but with a different PID (`31532`) for the connected endpoint.

3. `unix  3      [ ]         STREAM     CONNECTED     28798    /run/user/999/bus`
   - Another connected Unix domain socket, but this time associated with a different user (`999`).

4. `unix  3      [ ]         STREAM     CONNECTED     27052`
   - Indicates a Unix domain socket of type stream that is connected but does not specify the endpoint it is connected to.

5. `unix  3      [ ]         STREAM     CONNECTED     61118`
   - Similar to the previous line, this socket is connected but doesn't specify the endpoint.

6. `unix  3      [ ]         STREAM     CONNECTED     37595    /run/dbus/system_bus_socket`
   - A connected stream Unix domain socket associated with the D-Bus system bus socket.

7. `unix  3      [ ]         STREAM     CONNECTED     36306    @/tmp/.X11-unix/X1`
   - Indicates a connected stream Unix domain socket associated with an X11 display (`X1`).

8. `unix  3      [ ]         STREAM     CONNECTED     33347`
   - Another connected Unix domain socket without specifying the endpoint.

9. `unix  3      [ ]         STREAM     CONNECTED     25567    /run/user/999/at-spi/bus_0`

- Indicates a connected stream Unix domain socket associated with the AT-SPI accessibility bus for a specific user (`999`).

10. `unix  3      [ ]         STREAM     CONNECTED     34888    /run/user/998/at-spi/bus_1`
    - Similar to the previous line but associated with a different user (`998`) and bus number (`1`).

11. `unix  3      [ ]         STREAM     CONNECTED     28087`
    - Another connected Unix domain socket without specifying the endpoint.

12. `unix  3      [ ]         STREAM     CONNECTED     26075    /run/user/999/bus`
    - Indicates a connected stream Unix domain socket associated with the user's bus for user `999`.

13. `unix  3      [ ]         STREAM     CONNECTED     26460`
    - Another connected Unix domain socket without specifying the endpoint.

14. `unix  3      [ ]         STREAM     CONNECTED     28172    /run/user/999/bus`
    - Another connected stream Unix domain socket associated with the user's bus for user `999`.

15. `unix  3      [ ]         STREAM     CONNECTED     23088    /run/systemd/journal/stdout`
    - Indicates a connected stream Unix domain socket associated with the systemd journal's standard output.

16. `unix  3      [ ]         STREAM     CONNECTED     65404    /run/user/1000/bus`
    - Indicates a connected stream Unix domain socket associated with the user's bus for user `1000`.

17. `unix  3      [ ]         STREAM     CONNECTED     60195`
    - Another connected Unix domain socket without specifying the endpoint.

18. `unix  3      [ ]         STREAM     CONNECTED     41009    /run/user/1`
    - Indicates a connected stream Unix domain socket associated with the user's bus for user `1`.

19. `unix  2      [ ]         DGRAM      CONNECTED     2943`
    - Indicates a connected datagram Unix domain socket.

20. `unix  3      [ ]         SEQPACKET  CONNECTED     39731`

- Indicates a connected sequential packet Unix domain socket.

A **SEQPACKET** socket type in Unix domain sockets represents a connection-oriented socket that provides a reliable, sequenced, and unduplicated flow of data.

PING

`ping` is a command-line utility used to test the reachability of a host on an Internet Protocol (IP) network and to measure the round-trip time for messages sent from the originating host to a destination computer. Here's a brief overview of its functionality:

- Basic Functionality: `ping` sends ICMP (Internet Control Message Protocol) Echo Request messages to the target host and waits for ICMP Echo Reply messages in response.

- Reachability Test: It verifies whether the target host is reachable over the network. If the host responds to the ICMP Echo Request, it means the host is reachable.

- Round-Trip Time (RTT) Measurement: `ping` measures the round-trip time taken for an ICMP Echo Request message to travel from the source to the destination and back. This RTT provides an indication of the network latency between the source and destination.

- Continuous Testing: By default, `ping` sends a single ICMP Echo Request and waits for a reply. However, it can also be configured to send a continuous stream of ICMP Echo Request messages, allowing users to monitor the connectivity status over time.

- Packet Loss Detection: `ping` detects packet loss by analyzing the percentage of ICMP Echo Request messages that do not receive a corresponding ICMP Echo Reply. Packet loss can indicate network congestion or connectivity issues.

- IPv4 and IPv6 Support: `ping` supports both IPv4 and IPv6 addresses, enabling testing across different network protocols.

`ping` is a simple yet powerful tool for network troubleshooting, diagnosing connectivity issues, and measuring network performance.

```
lab@lab-V520-15IKL:~/Desktop$ ping www.google.com

PING www.google.com (142.250.192.132) 56(84) bytes of data.
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=1 ttl=55 time=18.5 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=2 ttl=55 time=18.4 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=3 ttl=55 time=18.4 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=4 ttl=55 time=18.4 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=5 ttl=55 time=18.4 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=6 ttl=55 time=19.0 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=7 ttl=55 time=18.4 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=8 ttl=55 time=18.3 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=9 ttl=55 time=18.3 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=10 ttl=55 time=18.2 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=11 ttl=55 time=18.3 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=12 ttl=55 time=18.4 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=13 ttl=55 time=18.4 ms
64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=14 ttl=55 time=18.2 ms
```

```
--- www.google.com ping statistics ---
124 packets transmitted, 124 received, 0% packet loss, time 123174ms
rtt min/avg/max/mdev = 18.124/18.472/23.772/0.842 ms
```

EXPLANATION

Each line in the provided output of the ping command represents a response from the destination host (www.google.com), along with some additional information.

1. `PING www.google.com (142.250.192.132) 56(84) bytes of data.`:
   - Indicates that the `ping` command is sending ICMP Echo Request messages to the host `www.google.com` with the IP address `142.250.192.132`. It specifies that each packet being sent is 56 bytes in size, with 84 bytes of additional data.

2. `64 bytes from bom12s18-in-f4.1e100.net (142.250.192.132): icmp_seq=1 ttl=55 time=18.5 ms`:
   - Represents the response received from the destination host.
   - `64 bytes`: Indicates the size of the ICMP Echo Reply packet received.
   - `bom12s18-in-f4.1e100.net`: Hostname of the responding host.
   - `(142.250.192.132)`: IP address of the responding host.
   - `icmp_seq=1`: Sequence number of the ICMP Echo Request message sent.
   - `ttl=55`: Time to Live value of the packet received.
   - `time=18.5 ms`: Round-trip time (RTT) in milliseconds for the ICMP Echo Request and Echo Reply messages.

3. The subsequent lines follow the same format as line 2, each representing a response to an ICMP Echo Request message sent.

4. `--- www.google.com ping statistics ---`:
   - Indicates the beginning of the statistics section, summarizing the `ping` results.

5. `124 packets transmitted, 124 received, 0% packet loss, time 123174ms`:
   - Indicates the number of ICMP Echo Request packets transmitted and received, along with the percentage of packet loss. In this case, all 124 packets were received successfully (`0% packet loss`).

6. `rtt min/avg/max/mdev = 18.124/18.472/23.772/0.842 ms`:
   - Provides statistics regarding the round-trip time (RTT) for the received ICMP Echo Reply packets.
   - `min`: Minimum RTT observed.
   - `avg`: Average RTT observed.
   - `max`: Maximum RTT observed.
- `mdev`: Mean deviation of RTT.

ICMP Echo Request messages are used in network diagnostics to determine whether a remote host is reachable and responsive

ARP

ARP (Address Resolution Protocol) is a protocol used for mapping an IP address to a physical machine address, such as an Ethernet MAC address. Here's a brief overview:

- Mapping IP to MAC: ARP resolves IP addresses to MAC addresses, enabling communication between devices within the same network segment.

- Protocol Operation: When a device needs to send data to another device within its network, it checks its ARP cache (a table of recent IP-to-MAC mappings). If the mapping isn't found, it sends out an ARP request broadcast asking for the MAC address associated with the target IP.

- Response: The device with the corresponding IP address responds with its MAC address, allowing the requesting device to update its ARP cache and establish communication.

- Address Resolution: ARP is crucial for Ethernet networks, where MAC addresses are used for low-level communication. It helps devices locate each other on the same network segment.

-Dynamic Nature: ARP mappings are dynamic and can change over time due to network changes or device configurations.

- ARP Cache: Devices maintain an ARP cache to store recently resolved mappings, improving efficiency by reducing the need for repeated ARP requests.

```
lab@lab-V520-15IKL:~/Desktop$ arp

Address              HWtype  HWaddress          Flags Mask        Iface
_gateway              ether   c4:ad:34:5e:00:b8   C                enp1s0
172.23.118.239          ether   44:37:e6:e4:65:26   C                enp1s0
```

EXPLANATION

Each line of the output represents an entry in the ARP cache table. Here's an explanation of each field:

1. `_gateway              ether   c4:ad:34:5e:00:b8   C                enp1s0`:
   - `_gateway`: The IP address corresponding to the default gateway.
   - `ether`: The hardware (MAC) address type.
   - `c4:ad:34:5e:00:b8`: The MAC address of the default gateway.
   - `C`: Indicates that this is a complete (resolved) entry in the ARP cache.
   - `enp1s0`: The network interface associated with this ARP entry.

2. `172.23.118.239          ether   44:37:e6:e4:65:26   C                enp1s0`:
   - `172.23.118.239`: The IP address.
   - `ether`: The hardware (MAC) address type.

- `44:37:e6:e4:65:26`: The MAC address corresponding to the IP address.
- `C`: Indicates that this is a complete (resolved) entry in the ARP cache.
- `enp1s0`: The network interface associated with this ARP entry.

- Gateway: A network node or device serving as an entry point into another network, facilitating communication between different networks.

- Default Gateway: Specifically in TCP/IP networking, the router or network device used to forward traffic from a device to destinations outside its local network or subnet. It's the default route for traffic not matching any other specific route in the device's routing table.

TELNET

Telnet is a network protocol used on the Internet or local area networks to provide bidirectional interactive text-oriented communication between two devices. It allows a user to remotely access and manage another computer or device over a network, typically using a command-line interface.

Here's a brief overview of how Telnet works:

1. Connection Establishment: The Telnet client initiates a connection to the Telnet server by specifying the server's IP address or domain name and the port number (usually port 23).

2. Session Initiation: Once the connection is established, a Telnet session begins, allowing the user to interact with the remote system. This interaction typically takes place through a text-based terminal interface.

3. Text Transmission: Commands and responses are exchanged between the client and server as plain text. The user sends commands to the remote system, which executes them and sends back the results.

4. End of Session: When the user finishes the session, they can gracefully terminate the connection, closing the Telnet session.

While Telnet is a simple and straightforward protocol, it's important to note that it lacks security features. Communications through Telnet are not encrypted, meaning that sensitive information, such as login credentials, can be intercepted by malicious actors if transmitted over insecure networks. As a result, Telnet has largely been replaced by more secure protocols like SSH (Secure Shell) for remote access and management tasks.

Protocol details:

Telnet is a client-server protocol, based on a reliable connection-oriented transport. Typically thisprotocol is used to establish a connection to TCP port 23.

sudo apt install telnetd-y(install in both systems)

```
telnet <ip address>
ex: telnet 172.23.101.141...
connected to 172.23.101.141
login :student
pwd: lords
welcome to Ubuntu
ls:
Desktop Documents Downloads....
cd Desktop
All files will be displayed
```

FTP

FTP, or File Transfer Protocol, is a standard network protocol used for transferring files from one host to another over a TCP-based network, such as the Internet or a local network. Here's a brief overview:

1. Client-Server Architecture: FTP operates on a client-server model, where one device (the client) initiates a connection to another device (the server) to transfer files.

2. Two Connection Channels: FTP uses two separate channels for communication: the control channel and the data channel. The control channel handles commands and responses between the client and server, while the data channel is used for transferring actual file data.

3. Commands and Responses: FTP clients send commands to the FTP server to perform various operations such as listing directories, uploading files, downloading files, renaming files, and deleting files. The server responds to these commands with status codes to indicate the success or failure of the operation.

4. Modes of Operation: FTP supports two modes of operation: active mode and passive mode. In active mode, the client initiates the data connection to the server, while in passive mode, the server initiates the data connection to the client. Passive mode is commonly used when the client is behind a firewall or NAT device.

5. Security Considerations: Traditional FTP transmits data, including usernames and passwords, in plain text, making it susceptible to eavesdropping attacks. To address this issue, secure alternatives like FTPS (FTP Secure) and SFTP (SSH File Transfer Protocol) have been developed, which add encryption and authentication mechanisms to FTP.

Overall, FTP is a widely used protocol for transferring files between systems, although its use has declined in favor of more secure alternatives like FTPS and SFTP, as well as web-based file transfer solutions.

INSTALL: sudo apt-get update && sudo apt-get install vsftpd

```
TYPE  FTP 172.23.101.141
CONNECTED TO 172.23.101.141
NAME: STUDENT
PWD: LORDS
LOGIN SUCCESSFUL
PWD
CD/HOME/STUDENT/Desktop
DIRECTORY SUCCESSFULLY CHANGED
FTP> GET FILE.TXT
EX: GET PLOTTING3E077.M

.
.
.
TRANSFER COMPLETE
FTP> LS
SHOWS ALL FILES
FTP>?
```

FINGER

The `finger` command is a simple utility used in Unix-like operating systems to retrieve information about users on a remote system. It allows users to gather details such as login name, real name, terminal name, idle time, login time, and more. Here's a brief overview:

1. Usage: To use the `finger` command, simply type `finger` followed by the username you want to query. For example:

   ```
   ```

```
finger username
```

2. Information Provided: When you execute the `finger` command with a username, it typically returns information about that user, including their login name, real name, terminal name, login time, idle time, and possibly other details depending on the system configuration.

3. System Configuration: The information displayed by the `finger` command can vary based on the system's configuration. System administrators can control the type and amount of information that the `finger` command displays.

4. Security Concerns: `finger` can reveal sensitive information about users on a system, so its use may be restricted or disabled on some systems for security reasons.

5. Alternative Usage: In addition to querying specific usernames, you can also use `finger` without specifying a username to display a list of users currently logged into the system.

Overall, the `finger` command provides a convenient way to retrieve basic information about users on a Unix-like system, but its use may be limited due to security considerations.

```
lab@lab-V520-15IKL:~/Desktop$ sudo apt install finger
lab@lab-V520-15IKL:~/Desktop$ finger
Login        Name       Tty      Idle  Login Time   Office     Office Phone
guest-brnkej  Guest      tty7     3:03  Mar 11 10:31 (:0)
guest-ca6uns  Guest      tty8     3:03  Mar 11 10:36 (:1)
lab          415-lab    tty9     3:03  Mar 11 10:39 (:2)
lab@lab-V520-15IKL:~/Desktop$ finger lab
Login: lab                          Name: 415-lab
Directory: /home/lab                Shell: /bin/bash
On since Mon Mar 11 10:39 (IST) on tty9 from :2
   3 hours 4 minutes idle
No mail.
No Plan.
```

EXPLANATION

1. Login: This column shows the username or login name of the user currently logged in.

2. Name: This column typically displays the real name associated with the user account. However, in many cases, it may display a generic label like "Guest" or "Lab" instead of an actual name.

3. Tty: This column indicates the terminal (or TTY - TeleTYpewriter) to which the user is logged in. Each terminal session on a Unix-like system is assigned a unique TTY number.

4. Idle: This column shows the idle time of the user, indicating how long the user's terminal session has been inactive. In the provided example, "3:03" suggests that the user's terminal has been idle for three hours and three minutes.

5. Login Time: This column displays the time when the user logged into the system. In the provided example, it shows the date and time of the login, such as "Mar 11 10:31" for March 11th at 10:31 AM.

6. Office: This column typically provides additional information about the user, such as their job title, department, or location. In the example, it seems to display the name of the office associated with the user's session, like "415-lab" for laboratory 415.

7. Office Phone: This column, if available, would display the phone number associated with the user's office or workplace. However, in the provided example, it appears to be empty.

TRACEROUTE

Traceroute is a network diagnostic tool used to trace the route that packets take from one host to another across an IP network. It works by sending packets with gradually increasing time-to-live (TTL) values to the destination host and analyzing the ICMP (Internet Control Message Protocol) error messages returned by intermediate routers.

Here's a brief overview of how traceroute works:

1. Packet Transmission: Traceroute sends packets to the destination host with an initial TTL value of 1. When the packet reaches the first router, the TTL expires, and the router sends back an ICMP Time Exceeded message to the sender.

2. Hop-by-Hop Analysis: Traceroute then sends another packet with a TTL value of 2, causing it to reach the second router along the path before expiring. This process repeats, gradually increasing the TTL value with each packet sent, until the packet reaches the destination host.

3. Recording Route Information: As the packets traverse the network, each router along the path forwards the packet to the next hop and responds with an ICMP Time Exceeded message. Traceroute records the IP address and round-trip time (RTT) of each router in the path.

4. Displaying Results: Traceroute displays the recorded information, showing the sequence of routers (often referred to as hops) that packets took to reach the destination host. It also provides the RTT for each hop, indicating the time taken for the packet to travel to that router and back.

5. Identifying Network Issues: Traceroute is commonly used to diagnose network connectivity issues, such as routing misconfigurations, packet loss, or network congestion. By analyzing the traceroute results, network administrators can identify problematic routers or network segments causing delays or disruptions.

Overall, traceroute is a valuable tool for troubleshooting network problems and understanding the path that packets take across the Internet or local network. It provides insights into the network topology and helps pinpoint the source of connectivity issues.

INSTALL

```
lab@lab-V520-15IKL:~/Desktop$ sudo apt install inetutils-traceroute
traceroute google.com
traceroute to google.com (142.250.76.174), 64 hops max
  1   172.23.100.1  0.413ms  0.310ms  0.240ms
  2   103.102.86.145  8.458ms  2.159ms  2.155ms
  3   103.88.102.1  1.160ms  1.046ms  0.934ms
  4   103.41.98.33  1.439ms  1.469ms  1.575ms
  5   *  *  *
  6   *  *  *
  7   198.18.5.19  20.890ms  20.785ms  20.777ms
  8   72.14.198.36  17.894ms  17.862ms  17.921ms
  9   192.178.110.123  26.066ms  25.811ms  25.861ms
 10   74.125.253.165  18.513ms  18.446ms  18.446ms
 11   142.250.209.70  16.313ms  16.652ms  16.116ms
 12   192.178.110.107  21.126ms  21.303ms  21.315ms
 13   142.250.76.174  20.851ms  27.201ms  21.287ms
```

EXPLANATION

`1   172.23.100.1  0.413ms  0.310ms  0.240ms`: This line indicates the first hop in the traceroute path.

- `1`: Represents the hop count, indicating that this is the first router encountered.

- `172.23.100.1`: IP address of the first router.

- `0.413ms`, `0.310ms`, `0.240ms`: Round-trip time (RTT) in milliseconds for three packets sent to the first router.

`2   103.102.86.145   8.458ms   2.159ms   2.155ms`: This line represents the second hop in the traceroute path.

- `2`: Indicates the second hop.

- `103.102.86.145`: IP address of the second router.

- `8.458ms`, `2.159ms`, `2.155ms`: RTT for three packets sent to the second router.

`3   103.88.102.1  1.160ms  1.046ms  0.934ms`: This line represents the third hop.

- `3`: Third hop in the path.

- `103.88.102.1`: IP address of the third router.

- `1.160ms`, `1.046ms`, `0.934ms`: RTT for three packets sent to the third router.

`4   103.41.98.33  1.439ms  1.469ms  1.575ms`: This line represents the fourth hop.

- `4`: Fourth hop in the path.

- `103.41.98.33`: IP address of the fourth router.

- `1.439ms`, `1.469ms`, `1.575ms`: RTT for three packets sent to the fourth router.

`5   *  *  *`: This line indicates that the traceroute tool didn't receive a response from the fifth hop. The asterisks (*) represent lost packets or no response within the timeout period.

`6   *  *  *`: Similar to the previous line, traceroute didn't receive a response from the sixth hop.

`7   198.18.5.19  20.890ms  20.785ms  20.777ms`: This line represents the seventh hop.

- `7`: Seventh hop in the path.

- `198.18.5.19`: IP address of the seventh router.

- `20.890ms`, `20.785ms`, `20.777ms`: RTT for three packets sent to the seventh router.

`8   72.14.198.36  17.894ms  17.862ms  17.921ms`: This line represents the eighth hop.

- `8`: Eighth hop in the path.
- `72.14.198.36`: IP address of the eighth router.
- `17.894ms`, `17.862ms`, `17.921ms`: RTT for three packets sent to the eighth router.

`9 192.178.110.123 26.066ms 25.811ms 25.861ms`: This line represents the ninth hop.

- `9`: Ninth hop in the path.
- `192.178.110.123`: IP address of the ninth router.
- `26.066ms`, `25.811ms`, `25.861ms`: RTT for three packets sent to the ninth router.

`10 74.125.253.165 18.513ms 18.446ms 18.446ms`: This line represents the tenth hop.

- `10`: Tenth hop in the path.
- `74.125.253.165`: IP address of the tenth router.
- `18.513ms`, `18.446ms`, `18.446ms`: RTT for three packets sent to the tenth router.

`11 142.250.209.70 16.313ms 16.652ms 16.116ms`: This line represents the eleventh hop.

- `11`: Eleventh hop in the path.
- `142.250.209.70`: IP address of the eleventh router.
- `16.313ms`, `16.652ms`, `16.116ms`: RTT for three packets sent to the eleventh router.

`12 192.178.110.107 21.126ms 21.303ms 21.315ms`: This line represents the twelfth hop.

- `12`: Twelfth hop in the path.
- `192.178.110.107`: IP address of the twelfth router.
- `21.126ms`, `21.303ms`, `21.315ms`: RTT for three packets sent to the twelfth router.

`13 142.250.76.174 20.851ms 27.201ms 21.287ms`: This line represents the final destination.

- `13`: Indicates the final destination.
- `142.250.76.174`: IP address of the destination host (in this case, google.com).
- `20.851ms`, `27.201ms`, `21.287ms`: RTT for three packets sent to the destination host.

PROGRAM-2:
Write a program to implement connection oriented and connection less client for well-known services i.e., standard ports.

connection oriented

//Server

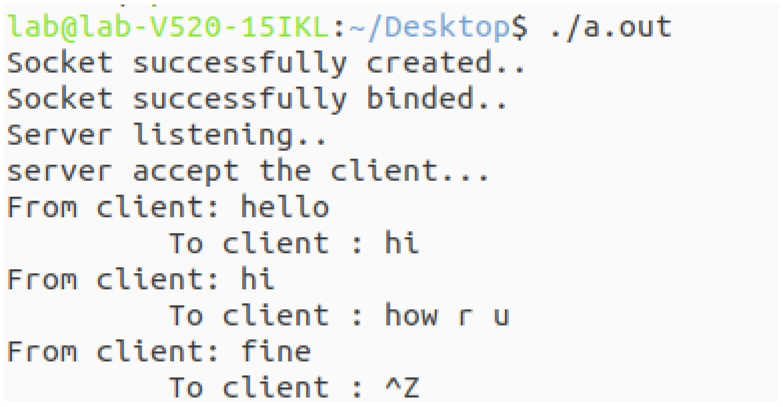
```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
// Function designed for chat between client and server.
void func(int sockfd)
{
char buff[MAX];
int n;
// infinite loop for chat
for (;;) {
bzero(buff, MAX);
// read the message from client and copy it in buffer
read(sockfd, buff, sizeof(buff));
// print buffer which contains the client contents
printf("From client: %s\t To client : ", buff);
bzero(buff, MAX);
n = 0;
// copy server message in the buffer
while ((buff[n++] = getchar()) != '\n')
;
// and send that buffer to client
write(sockfd, buff, sizeof(buff));
// if msg contains "Exit" then server exit and chat ended.
if (strncmp("exit", buff, 4) == 0) {
printf("Server Exit...\n");
break;
}
}
}
// Driver function
int main()
```

```
{
int sockfd, connfd, len;
struct sockaddr_in servaddr, cli;
// socket create and verification
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
printf("socket creation failed...\n");
exit(0);
}
else
printf("Socket successfully created..\n");
bzero(&servaddr, sizeof(servaddr));
// assign IP, PORT
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);
// Binding newly created socket to given IP and verification
if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
printf("socket bind failed...\n");
exit(0);
}
else
printf("Socket successfully binded..\n");
// Now server is ready to listen and verification
if ((listen(sockfd, 5)) != 0) {
printf("Listen failed...\n");
exit(0);
}
else
printf("Server listening..\n");
len = sizeof(cli);
// Accept the data packet from client and verification
connfd = accept(sockfd, (SA*)&cli, &len);
if (connfd < 0) {
printf("server acccept failed...\n");
exit(0);
25
}
else
printf("server acccept the client...\n");
// Function for chatting between client and server
func(connfd);
// After chatting close the socket
close(sockfd);
}
```

SAVE: conn-oriented-server.c

COMPILE: gcc conn-oriented-server.c

RUN:./a.out



```
lab@lab-V520-15IKL:~/Desktop$ ./a.out
Socket successfully created..
Socket successfully binded..
Server listening..
server accept the client...
From client: hello
         To client : hi
From client: hi
         To client : how r u
From client: fine
         To client : ^Z
```

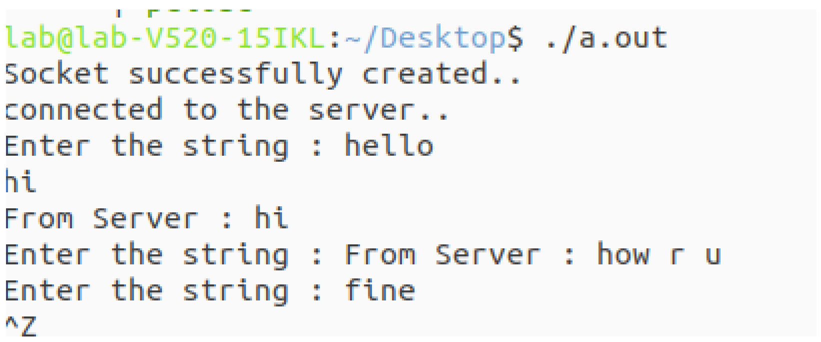
CLIENT.C

```
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
void func(int sockfd)
{
char buff[MAX];
int n;
for (;;) {
bzero(buff, sizeof(buff));
printf("Enter the string : ");
n = 0;
while ((buff[n++] = getchar()) != '\n')
;
write(sockfd, buff, sizeof(buff));
bzero(buff, sizeof(buff));
read(sockfd, buff, sizeof(buff));
printf("From Server : %s", buff);
if ((strncmp(buff, "exit", 4)) == 0) {
printf("Client Exit...\n");
break;
}
}
}
int main()
{
int sockfd, connfd;
struct sockaddr_in servaddr, cli;
// socket create and varification
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
printf("socket creation failed...\n");
exit(0);
}
else
printf("Socket successfully created..\n");
bzero(&servaddr, sizeof(servaddr));
// assign IP, PORT
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
servaddr.sin_port = htons(PORT);
// connect the client socket to server socket
if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
printf("connection with the server failed...\n");
exit(0);
}
else
printf("connected to the server..\n");
// function for chat
func(sockfd);
// close the socket
close(sockfd);
}
```

SAVE: conn-oriented-server.c

COMPILE: gcc conn-oriented-server.c

RUN:./a.out



```
lab@lab-V520-15IKL:~/Desktop$ ./a.out
Socket successfully created..
connected to the server..
Enter the string : hello
hi
From Server : hi
Enter the string : From Server : how r u
Enter the string : fine
^Z
```

Connection Less

//SERVER

```
#include <stdio.h>

#include <errno.h>

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

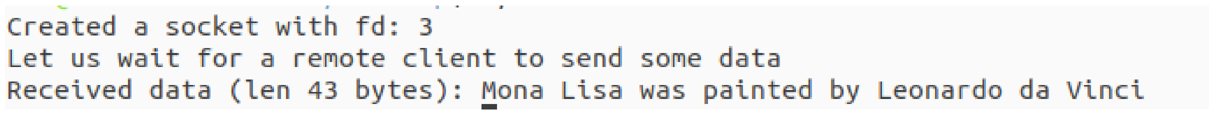
```
#define DATA_BUFFER 5000
int main ()
{
struct sockaddr_in saddr, new_addr;
int fd, ret_val;
char buf[DATA_BUFFER];
socklen_t addrlen;
/* Step1: open a UDP socket */
fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (fd == -1) {
fprintf(stderr, "socket failed [%s]\n", strerror(errno));
return -1;
}
printf("Created a socket with fd: %d\n", fd);
/* Initialize the socket address structure */
saddr.sin_family = AF_INET;
saddr.sin_port = htons(7000);
saddr.sin_addr.s_addr = INADDR_ANY;
/* Step2: bind the socket */
ret_val = bind(fd, (struct sockaddr *)&saddr, sizeof(struct sockaddr_in));
if (ret_val != 0) {
fprintf(stderr, "bind failed [%s]\n", strerror(errno));
close(fd);
return -1;
}
/* Step3: Start receiving data. */
printf("Let us wait for a remote client to send some data\n");
ret_val = recvfrom(fd, buf, DATA_BUFFER, 0,
(struct sockaddr *)&new_addr, &addrlen);
if (ret_val != -1) {
printf("Received data (len %d bytes): %s\n", ret_val, buf);
} else {
printf("recvfrom() failed [%s]\n", strerror(errno));
}
/* Last step: close the socket */
close(fd);
return 0;
}
```

SAVE: conn-less-server.c

COMPILE: gcc conn-less-server.c

RUN:./a.out

OUTPUT:



```
Created a socket with fd: 3
Let us wait for a remote client to send some data
Received data (len 43 bytes): Mona Lisa was painted by Leonardo da Vinci
```

//Client

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <netinet/in.h>
#include <netdb.h>
#define DATA_BUFFER "Mona Lisa was painted by Leonardo da Vinci"
int main () {
```

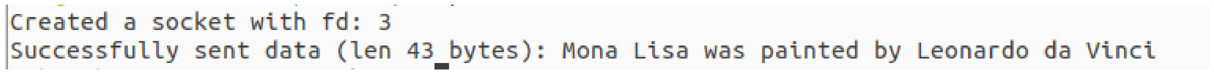
```
struct sockaddr_in saddr;
int fd, ret_val;
struct hostent *host; /* need netdb.h for this */
/* Step1: open a UDP socket */
fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (fd == -1) {
fprintf(stderr, "socket failed [%s]\n", strerror(errno));
return -1;
}
printf("Created a socket with fd: %d\n", fd);
/* Next, initialize the server address */
saddr.sin_family = AF_INET;
saddr.sin_port = htons(7000);
host = gethostbyname("127.0.0.1");
saddr.sin_addr = *((struct in_addr *)host->h_addr);
/* Step2: send some data */
ret_val = sendto(fd,DATA_BUFFER, strlen(DATA_BUFFER) + 1, 0,
(struct sockaddr *)&saddr, sizeof(struct sockaddr_in));
if (ret_val != -1) {
printf("Successfully sent data (len %d bytes): %s\n", ret_val, DATA_BUFFER);
} else {
printf("sendto() failed [%s]\n", strerror(errno));
}
/* Last step: close the socket */
close(fd);
return 0;
}
```

SAVE: conn-less-client.c

COMPILE: gcc conn-less-client.c

RUN:./a.out

OUTPUT:



```
Created a socket with fd: 3
Successfully sent data (len 43 bytes): Mona Lisa was painted by Leonardo da Vinci
```

PROGRAM 3:
Implementation of concurrent server service using connection-oriented socket system calls(Service: Daytime)

A concurrent server service using connection-oriented socket system calls, specifically for the Daytime service, involves creating a server that can handle multiple client connections simultaneously, providing each client with the current date and time upon request.

1. Socket Initialization: The server initializes a socket using the `socket()` system call. This socket will be used to listen for incoming connections from clients.

2. Binding: The server binds the socket to a specific port using the `bind()` system call. This allows clients to connect to the server using the specified port number.

3. Listening: The server sets the socket to listen for incoming connections using the `listen()` system call. This prepares the socket to accept client connections.

4. Accepting Connections: The server enters a loop where it continuously accepts incoming connections using the `accept()` system call. Each time a client connects, the server creates a new socket dedicated to handling communication with that client.

5. Handling Clients Concurrently: For each accepted connection, the server forks a new process or spawns a new thread to handle communication with the client concurrently. This allows the server to handle multiple clients simultaneously without blocking.

6. Daytime Service: Upon receiving a connection from a client, the server retrieves the current date and time using system-specific functions (e.g., `gettimeofday()` or `localtime()`) and sends this information back to the client over the established connection.

7. Closing Connections: After sending the date and time to the client, the server closes the connection for that client using the `close()` system call. The server then continues to listen for new connections.

8. Shutdown: When the server needs to stop, it closes the listening socket and terminates any remaining client connections gracefully.

Here are the methods involved in implementing a concurrent server service for the Daytime protocol using connection-oriented socket system calls:

1. initialize_socket(): This method initializes the socket for communication. It calls the `socket()` system call to create a new socket and sets it up for TCP communication.

2. bind_socket(): This method binds the socket to a specific IP address and port number using the `bind()` system call. It specifies the address and port on which the server will listen for incoming connections.

3. listen_for_connections(): This method sets the socket to the listening state using the `listen()` system call. It specifies the maximum number of pending connections that can be queued up by the operating system.

4. accept_connections(): This method continuously accepts incoming connections using the `accept()` system call within a loop. When a new connection is accepted, it spawns a new process or thread to handle communication with the client.

5. handle_client(): This method is called for each accepted connection to handle communication with the client. It retrieves the current date and time using system-specific functions and sends this information back to the client over the established connection.

6. close_connection(): This method closes the connection with the client using the `close()` system call after sending the date and time information. It ensures that the resources associated with the connection are released properly.

7. shutdown_server(): This method shuts down the server gracefully. It closes the listening socket and terminates any remaining client connections before exiting the server application.

//Client Program

```
#include"netinet/in.h"

#include"sys/socket.h"

#include"stdio.h"
```

```
main()
{
struct sockaddr_in sa,cli;
int n,sockfd;
int len;char buff[100];
sockfd=socket(AF_INET,SOCK_STREAM,0);
if(sockfd<0){ printf("\nError in Socket");
exit(0);
}
else printf("\nSocket is Opened");
bzero(&sa,sizeof(sa));
sa.sin_family=AF_INET;
sa.sin_port=htons(5600);
if(connect(sockfd,(struct sockaddr*)&sa,sizeof(sa))<0)
{
printf("\nError in connection failed");
exit(0);
}
else
printf("\nconnected successfully");
if(n=read(sockfd,buff,sizeof(buff))<0)
{
printf("\nError in Reading");
exit(0);
}
else
{printf("\nMessage Read %s",buff);
}}
```

SAVE: concurrent-client.c

COMPILE: gcc concurrent-client.c

RUN:./a.out

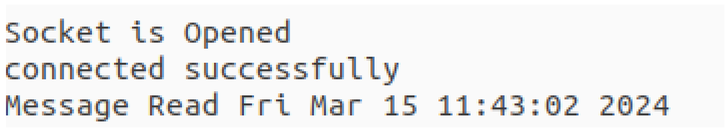
OUTPUT:

SAVE: concurrent-client.c

COMPILE: gcc concurrent-client.c

RUN:./a.out

OUTPUT:



```
Socket is Opened
connected successfully
Message Read Fri Mar 15 11:43:02 2024
```

//Server Program

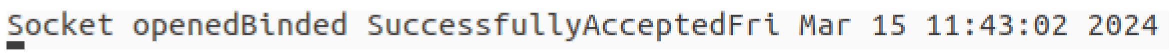
```
 #include"netinet/in.h"
 #include"sys/socket.h"
 #include"stdio.h"
#include"string.h"
#include"time.h"
main( )
{
struct sockaddr_in sa;
struct sockaddr_in cli;int sockfd,conntfd;int len,ch;char str[100];
time_t tick;
sockfd=socket(AF_INET,SOCK_STREAM,0);
if(sockfd<0)
{
printf("error in socket\n");
exit(0);
}
else printf("Socket opened");
bzero(&sa,sizeof(sa));
sa.sin_port=htons(5600);
sa.sin_addr.s_addr=htonl(0);
if(bind(sockfd,(struct sockaddr*)&sa,sizeof(sa))<0)
{
printf("Error in binding\n");
}
else
printf("Binded Successfully");
listen(sockfd,50);
for(;;)
{
len=sizeof(ch);
conntfd=accept(sockfd,(struct sockaddr*)&cli,&len);
printf("Accepted");
tick=time(NULL);
snprintf(str,sizeof(str),"%s",ctime(&tick));
printf("%s",str);write(conntfd,str,100);
}
}
```

SAVE: concurrent-server.c

COMPILE: gcc concurrent-server.c

RUN: ./a.out

OUTPUT:



```
Socket openedBinded SuccessfullyAcceptedFri Mar 15 11:43:02 2024
```

PROGRAM 4:
Implementation of concurrent server using connection less socket system calls.(Service: Echo server)

SERVER PROGRAM

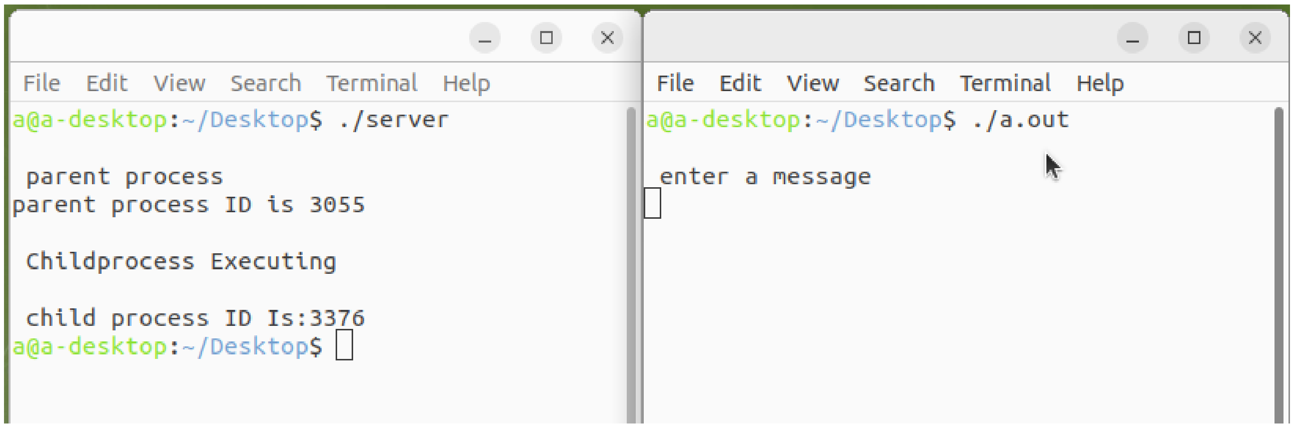
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
main(int argc, char *argv[])
{
    int sockfd, rval, pid;
    char buff1[20], buff2[20];
    struct sockaddr_in server, client;
    int len;
    sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sockfd == -1)
    {
    }
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_port = htons(3221);
    rval = bind(sockfd, (struct sockaddr *)&server, sizeof(server));
    if (rval != -1)
    {
        pid = fork();
        if (pid == 0)
        {
            printf("\n Childprocess Executing \n");
            printf("\n child process ID Is:%d\n",
                  getpid());
            len = sizeof(client);
            rval = recvfrom(sockfd, buff1, sizeof(buff1), 0, (struct sockaddr *)&client, &len);
            if (rval == -1)
            {
              perror("\n RECV_ERR\n");
              exit(1);
            }
            else
            {
              printf("\n Received Message Is:%s\n", buff1);
            }
            rval = sendto(sockfd, buff1, sizeof(buff1), 0, (struct sockaddr *)&client, sizeof(client));
            if (rval != -1)
            {
              printf("\n Message sent successfully \n");
            }
            else
            {
              perror("\n SEND_ERR\n");
              exit(1);
            }
        }
        else
            printf("\n parent process\n");
        printf("parent process ID is %d\n", getppid());
    }
    else
    {
            perror("\n BIND_ERR\n");
            exit(1);
    }
}
```

CLIENT PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(int argc, char *argv[])
{
    int sockfd, rval;
    char buff1[20], buff2[20];
    struct sockaddr_in server, client;
    int len;
    sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sockfd == -1)
    {
        perror("\n SOCK_ERR\n");
        exit(1);
    }
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_port = htons(3221);
    printf("\n enter a message \n");
    scanf("%s", buff1);
    rval = sendto(sockfd, buff1, sizeof(buff1), 0, (struct sockaddr *)&server, sizeof(server));
    if (rval != -1)
    {
        printf("\nmessage sent successfully\n");
    }
    else
    {
        perror("\n SEND_ERR\n");
        exit(1);
    }
    len = sizeof(server);
    rval = recvfrom(sockfd, buff1, sizeof(buff1), 0, (struct sockaddr *)&server, &len);
    if (rval == -1)
    {
        perror("\nRECV_ERR\n");
        exit(1);
    }
    else
    {
        printf("\n Received Message is %s\n", buff1);
    }
}
```

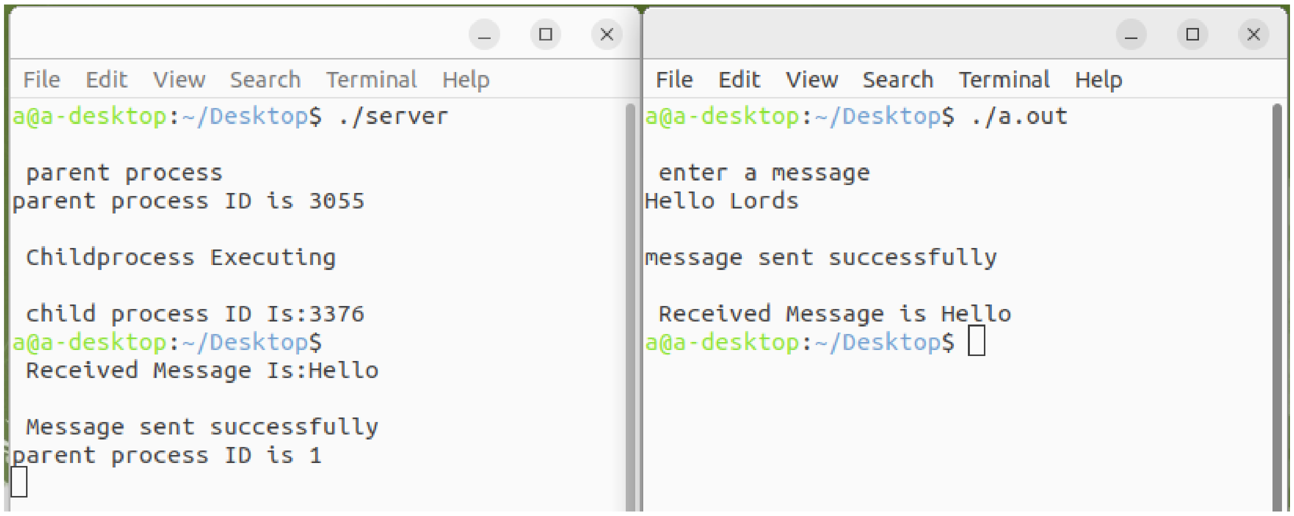
OUTPUT:

screenshot 1:



```
a@a-desktop:~/Desktop$ ./server

 parent process
parent process ID is 3055

 Childprocess Executing

 child process ID Is:3376
a@a-desktop:~/Desktop$
```

```
a@a-desktop:~/Desktop$ ./a.out

 enter a message
```



```
a@a-desktop:~/Desktop$ ./server

 parent process
parent process ID is 3055

 Childprocess Executing

 child process ID Is:3376
a@a-desktop:~/Desktop$
 Received Message Is:Hello

 Message sent successfully
parent process ID is 1
```

```
a@a-desktop:~/Desktop$ ./a.out

 enter a message
Hello Lords

message sent successfully

 Received Message is Hello
a@a-desktop:~/Desktop$
```

screenshot 2:

PROGRAM 5:
Write a programs for TCP chat server

// Program for chatappserver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include<errno.h>

#define PORT 8080
#define MAX_CLIENTS 5
#define BUFFER_SIZE 1024

int main() {
    int server_fd, client_fds[MAX_CLIENTS], max_sd, activity, sd, new_socket;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];
    fd_set readfds;

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 5) == -1) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d...\n", PORT);

    for (int i = 0; i < MAX_CLIENTS; ++i) {
        client_fds[i] = 0;
    }

    while (1) {
        FD_ZERO(&readfds);
        FD_SET(server_fd, &readfds);
        max_sd = server_fd;

        for (int i = 0; i < MAX_CLIENTS; ++i) {
            sd = client_fds[i];
            if (sd > 0) {
                FD_SET(sd, &readfds);
            }
            if (sd > max_sd) {
                max_sd = sd;
            }
        }

        activity = select(max_sd + 1, &readfds, NULL, NULL, NULL);
        if ((activity < 0) && (errno != EINTR)) {
            perror("select error");
        }

        if (FD_ISSET(server_fd, &readfds)) {
            if ((new_socket = accept(server_fd, NULL, NULL)) < 0) {
                perror("accept");
                exit(EXIT_FAILURE);
            }

            printf("New connection, socket fd is %d\n", new_socket);

            for (int i = 0; i < MAX_CLIENTS; ++i) {
                if (client_fds[i] == 0) {
                    client_fds[i] = new_socket;
                    break;
                }
            }
        }

        for (int i = 0; i < MAX_CLIENTS; ++i) {
            sd = client_fds[i];
            if (FD_ISSET(sd, &readfds)) {
                int valread;
                if ((valread = read(sd, buffer, BUFFER_SIZE)) == 0) {
                    getpeername(sd, (struct sockaddr *)&server_addr, (socklen_t *)&server_addr);
                    printf("Host disconnected, ip %s, port %d\n", inet_ntoa(server_addr.sin_addr),
ntohs(server_addr.sin_port));
                    close(sd);
                    client_fds[i] = 0;
                } else {
                    buffer[valread] = '\0';
                    for (int j = 0; j < MAX_CLIENTS; ++j) {
                        if (client_fds[j] != 0 && client_fds[j] != sd) {
                            send(client_fds[j], buffer, strlen(buffer), 0);
                        }
                    }
                }
            }
        }
    }

    return 0;
}
```

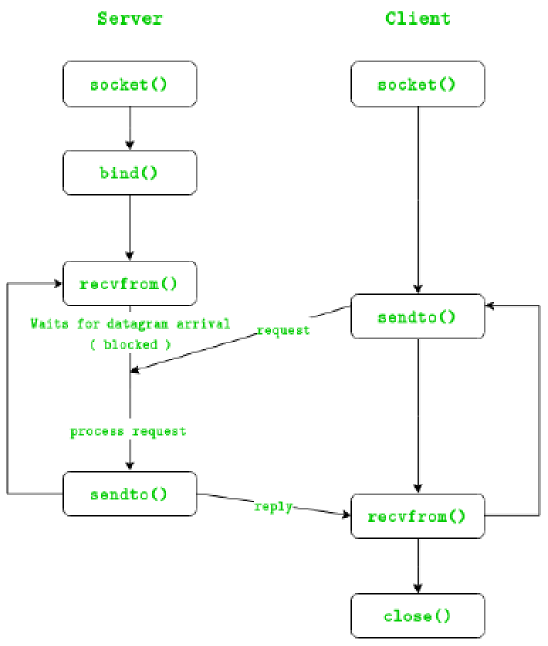
// **Program for chatappclient.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

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

#define PORT 8080
#define SERVER_IP "127.0.0.1"
#define BUFFER_SIZE 1024

int main() {
    int sockfd;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) exit(EXIT_FAILURE);
    server_addr = (struct sockaddr_in){.sin_family = AF_INET, .sin_port = htons(PORT),
.sin_addr.s_addr = inet_addr(SERVER_IP)};
    if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1)
exit(EXIT_FAILURE);

    printf("Connected to server on port %d\n", PORT);

    while (1) {
        printf("Enter message: ");
        fgets(buffer, BUFFER_SIZE, stdin);
        send(sockfd, buffer, strlen(buffer), 0);
    }

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

PROGRAM 6:
Write a programs for UDP chat server.

Theory In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of sender which the server uses to send data to the correct client.



The entire process can be broken down into following steps : **UDP Server :**

1. Create UDP socket.
2. Bind the socket to server address.
3. Wait until datagram packet arrives from client.
4. Process the datagram packet and send a reply to client.
5. Go back to Step 3.

UDP Client :

1. Create UDP socket.
2. Send message to server.
3. Wait until response from server is recieved.
4. Process reply and go back to step 2, if necessary.
5. Close socket descriptor and exit.

Necessary Functions :

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

Creates an unbound socket in the specified domain.

Returns socket file descriptor.

Arguments : domain – Specifies the communication domain (AF_INET for IPv4/ AF_INET6 for IPv6) **type –** Type of socket to be created (SOCK_STREAM for TCP / SOCK_DGRAM for UDP) **protocol –** Protocol to be used by socket. 0 means use default protocol for the address family.

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

Assigns address to the unbound socket.

Arguments : sockfd – File descriptor of socket to be binded **addr –** Structure in which address to be binded to is specified **addrlen –** Size of *addr* structure

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,

const struct sockaddr *dest_addr, socklen_t addrlen)

Send a message on the socket

Arguments : sockfd – File descriptor of socket **buf –** Application buffer containing the data to be sent **len –** Size of *buf* application buffer **flags –** Bitwise OR of flags to modify socket behaviour **dest_addr –** Structure containing address of destination **addrlen –** Size of *dest_addr* structure.
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,

struct sockaddr *src_addr, socklen_t *addrlen)

Receive a message from the socket.

Arguments : sockfd – File descriptor of socket **buf –** Application buffer in which to receive data **len –** Size of *buf* application buffer **flags –** Bitwise OR of flags to modify socket behaviour **src_addr –** Structure containing source address is returned **addrlen –** Variable in which size of *src_addr* structure is returned

int close(int fd)

Close a file descriptor

Arguments : fd – File descriptor

In the below code, exchange of one hello message between server and client is shown to demonstrate the model.

```
// Client side implementation of UDP client-server model

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

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

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#define PORT 8080
#define MAXLINE 1024
// Driver code
int main() {
int sockfd;
char buffer[MAXLINE];
char *hello = "Hello from client";
struct sockaddr_in servaddr;
// Creating socket file descriptor
if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
perror("socket creation failed");
exit(EXIT_FAILURE);
}
memset(&servaddr, 0, sizeof(servaddr));
// Filling server information
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(PORT);
servaddr.sin_addr.s_addr = INADDR_ANY;
int n, len;
sendto(sockfd, (const char *)hello, strlen(hello),
MSG_CONFIRM, (const struct sockaddr *) &servaddr,
sizeof(servaddr));
printf("Hello message sent.\n");
n = recvfrom(sockfd, (char *)buffer, MAXLINE,
MSG_WAITALL, (struct sockaddr *) &servaddr,
&len);
buffer[n] = '\0';
printf("Server : %s\n", buffer);
close(sockfd);
return 0;
}
```

Output :

```
$ ./server
Client : Hello from client
Hello message sent.
$ ./client
Hello message sent.
Server : Hello from server
```

PROGRAM 7:
Program to demonstrate the use of advanced socket system calls:readv(),writev(),getsockname(),setsockname(),getpeername(),gethostbyname(),gethostbyaddr(),getnetbyname(),getnetbyaddr(),getprotobyname(),getserv

1. **WRITE()**

```
#include<unistd.h>

int main()

{

write(1,"hello\n",6); //1 is the file descriptor, "hello\n" is the data, 6 is the count of characters in data

}
```

COMPILE: gcc write.c

RUN:./a.out

OUTPUT

```
hello
```

2. **READV()**

```
#include<unistd.h>

int main()

{

char buff[20];

read(0,buff,10);//read 10 bytes from standard input device(keyboard), store in buffer (buff)

write(1,buff,10);//print 10 bytes from the buffer on the screen

}
```

COMPILE: gcc read.c

RUN:./a.out

OUTPUT

```
1234567
```

3. getsockname()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERVER_ADDR "172.217.160.99"
#define SERVER_PORT 80

int main()
{
    char myIP[16];
    unsigned int myPort;
    struct sockaddr_in server_addr, my_addr;
    int sockfd;

    // Connect to server
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Can't open stream socket.");
        exit(-1);
    }

    // Set server_addr
    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(SERVER_ADDR);
    server_addr.sin_port = htons(SERVER_PORT);

    // Connect to server
    if (connect(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
        perror("Connect server error");
        close(sockfd);
        exit(-1);
    }

    // Get my ip address and port
    bzero(&my_addr, sizeof(my_addr));
    socklen_t len = sizeof(my_addr);
    getsockname(sockfd, (struct sockaddr *) &my_addr, &len);
    inet_ntop(AF_INET, &my_addr.sin_addr, myIP, sizeof(myIP));
    myPort = ntohs(my_addr.sin_port);

    printf("Local ip address: %s\n", myIP);
    printf("Local port : %u\n", myPort);

    return 0;
}
```

COMPILE: gcc getsockname.c

RUN:./a.out

OUTPUT

```
Local ip address: 178.77.101.235
Local port : 38576
```

4. Setsockname()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>

#define PORT 8080

int main() {
    int sockfd;
    struct sockaddr_in addr;

    // Create socket
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Set the local address and port
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(PORT);
    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }

    // Display the bound address and port
    socklen_t len = sizeof(addr);
    if (getsockname(sockfd, (struct sockaddr *)&addr, &len) == -1) {
        perror("Getsockname failed");
        exit(EXIT_FAILURE);
    }

    printf("Local address: %s\n", inet_ntoa(addr.sin_addr));
    printf("Local port: %d\n", ntohs(addr.sin_port));

    // Close the socket
    close(sockfd);

    return 0;
}
```

COMPILE: gcc setsockname.c

RUN:./a.out

OUTPUT

```
Local address: 0.0.0.0
Local port: 8080
```

5. gethostbyname()

```
// C program to display hostname
// and IP address
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// Returns hostname for the local computer
void checkHostName(int hostname)
{
        if (hostname == -1)
        {
                perror("gethostname");
                exit(1);
        }
}

// Returns host information corresponding to host name
void checkHostEntry(struct hostent * hostentry)
{
        if (hostentry == NULL)
        {
                perror("gethostbyname");
                exit(1);
        }
}

// Converts space-delimited IPv4 addresses
// to dotted-decimal format
void checkIPbuffer(char *IPbuffer)
{
        if (NULL == IPbuffer)
        {
                perror("inet_ntoa");
                exit(1);
        }
}

// Driver code
int main()
{
        char hostbuffer[256];
        char *IPbuffer;
        struct hostent *host_entry;
        int hostname;

        // To retrieve hostname
        hostname = gethostname(hostbuffer, sizeof(hostbuffer));
        checkHostName(hostname);

        // To retrieve host information
        host_entry = gethostbyname(hostbuffer);
        checkHostEntry(host_entry);
        // To convert an Internet network
        // address into ASCII string
        IPbuffer = inet_ntoa(*((struct in_addr*)
                                        host_entry->h_addr_list[0]));

        printf("Hostname: %s\n", hostbuffer);
        printf("Host IP: %s", IPbuffer);

        return 0;
}
```

COMPILE: gcc gethostname.c

RUN:./a.out

OUTPUT

```
Hostname: lvps178-77-101-235.dedicated.hosteurope.de
Host IP: 178.77.101.235
```

6. getpeername()

server program

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#define PORT 8080

int main() {
    int sockfd, new_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);

    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Assigning IP and port
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(PORT);

    // Binding newly created socket to given IP and verification
    if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    // Listening for clients
    if (listen(sockfd, 5) == -1) {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    printf("Server listening on port %d...\n", PORT);

    // Accepting incoming connections
    if ((new_socket = accept(sockfd, (struct sockaddr *)&client_addr, &addr_len)) == -1) {
        perror("accept");
        exit(EXIT_FAILURE);
    }

    // Getting peer name of the connected client
    if (getpeername(new_socket, (struct sockaddr *)&client_addr, &addr_len) == -1) {
        perror("getpeername");
        exit(EXIT_FAILURE);
    }

    printf("Client connected from %s:%d\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

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

Client program

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#define PORT 8080

int main() {
    int sockfd, new_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len = sizeof(client_addr);

    // Creating socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Assigning IP and port
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(PORT);

    // Binding newly created socket to given IP and verification
    if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    // Listening for clients
    if (listen(sockfd, 5) == -1) {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    printf("Server listening on port %d...\n", PORT);

    // Accepting incoming connections
    if ((new_socket = accept(sockfd, (struct sockaddr *)&client_addr, &addr_len)) == -1) {
        perror("accept");
        exit(EXIT_FAILURE);
    }

    // Getting peer name of the connected client
    if (getpeername(new_socket, (struct sockaddr *)&client_addr, &addr_len) == -1) {
        perror("getpeername");
        exit(EXIT_FAILURE);
    }

    printf("Client connected from %s:%d\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

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

COMPILE:

gcc server.c -o server

gcc client.c -o client

RUN:

./server

```
Server listening on port 8080...
Client connected from 127.0.0.1:59178
```

./client

```
Connected to server on port 8080
```

7. gethostbyaddress()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <netdb.h>

int main() {
  struct hostent *host_entry;
  struct in_addr addr;
  char ip_address[INET_ADDRSTRLEN];

  // IP address to resolve
  const char *ip = "8.8.8.8";

  // Convert IP address from string to binary form
  if (inet_pton(AF_INET, ip, &addr) != 1) {
    perror("inet_pton");
    exit(EXIT_FAILURE);
  }

  // Get host information using IP address
  host_entry = gethostbyaddr(&addr, sizeof(addr), AF_INET);
  if (host_entry == NULL) {
    herror("gethostbyaddr");
    exit(EXIT_FAILURE);
  }

  // Convert IP address to printable string format
  if (inet_ntop(AF_INET, &addr, ip_address, INET_ADDRSTRLEN) == NULL) {
    perror("inet_ntop");
    exit(EXIT_FAILURE);
  }

  printf("Official name: %s\n", host_entry->h_name);
  printf("IP address: %s\n", ip_address);

  return 0;
}
```

COMPILE: gcc gethostbyaddr.c

RUN:./a.out

OUTPUT:

8. gethostbyname()

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <arpa/inet.h>

int main() {
    struct hostent *host_entry;

    // Domain name to resolve
    const char *domain = "example.com";

    // Get host information using domain name
    host_entry = gethostbyname(domain);
    if (host_entry == NULL) {
        herror("gethostbyname");
        exit(EXIT_FAILURE);
    }

    // Display host information
    printf("Official name: %s\n", host_entry->h_name);

    // Display list of IP addresses
    printf("IP addresses:\n");
    for (int i = 0; host_entry->h_addr_list[i] != NULL; i++) {
        struct in_addr addr;
        memcpy(&addr, host_entry->h_addr_list[i], sizeof(struct in_addr));
        printf("%s\n", inet_ntoa(addr));
    }

    return 0;
}
```

COMPILE: gcc gethostbyname.c

RUN:./a.out

OUTPUT

9. getnetbyname()

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <netdb.h>

int main() {
    struct netent *net_entry;

    // Network name to retrieve information for
    const char *network_name = "loopback";

    // Get network information using network name
    net_entry = getnetbyname(network_name);
    if (net_entry == NULL) {
        perror("getnetbyname");
        exit(EXIT_FAILURE);
    }

    // Display network information
    printf("Network name: %s\n", net_entry->n_name);
    printf("Network number (host byte order): %u\n", net_entry->n_net);
    printf("Network address type: %d\n", net_entry->n_addrtype);

    return 0;
}
```

COMPILE: gcc getnetbyname.c

RUN:./a.out

OUTPUT:

```
getnetbyname: Success
```

10. getnetbyaddr()

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <arpa/inet.h>

int main() {
    struct netent *net_entry;
    unsigned long net_number;

    // Network number (in host byte order) to retrieve information for
    const char *net_address = "127.0.0.1";

    // Convert network address from string to binary form
    if (inet_pton(AF_INET, net_address, &net_number) != 1) {
        perror("inet_pton");
        exit(EXIT_FAILURE);
    }

    // Get network information using network number
    net_entry = getnetbyaddr(net_number, AF_INET);
    if (net_entry == NULL) {
        perror("getnetbyaddr");
        exit(EXIT_FAILURE);
    }

    // Display network information
    printf("Network name: %s\n", net_entry->n_name);
    printf("Network number (host byte order): %u\n", net_entry->n_net);
    printf("Network address type: %d\n", net_entry->n_addrtype);
    return 0;
}
```

COMPILE: gcc getnetbyaddr.c

RUN:./a.out

OUTPUT:

```
getnetbyaddr: Success
```

11. getprotobyname()

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>

int main() {
    struct protoent *proto_entry;

    // Protocol name to retrieve information for
    const char *protocol_name = "tcp";

    // Get protocol information using protocol name
    proto_entry = getprotobyname(protocol_name);
    if (proto_entry == NULL) {
        perror("getprotobyname");
        exit(EXIT_FAILURE);
    }
    // Display protocol information
    printf("Protocol name: %s\n", proto_entry->p_name);
    printf("Protocol number: %d\n", proto_entry->p_proto);

    return 0;
}
```

COMPILE: gcc getprotobyname.c

RUN:./a.out

OUTPUT:

```
Protocol name: tcp
Protocol number: 6
```

12. gerserverbyname()

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>

int main() {
    struct servent *serv_entry;

    // Service name and protocol to retrieve information for
    const char *service_name = "http";
    const char *protocol_name = "tcp";

    // Get service information using service name and protocol
    serv_entry = getservbyname(service_name, protocol_name);
    if (serv_entry == NULL) {
        perror("getservbyname");
        exit(EXIT_FAILURE);
    }

    // Display service information
    printf("Service name: %s\n", serv_entry->s_name);
    printf("Port number: %d\n", ntohs(serv_entry->s_port));
    printf("Protocol name: %s\n", serv_entry->s_proto);

    return 0;
}
```

COMPILE: gcc getserverbyname.c

RUN:./a.out

OUTPUT:

```
Service name: http
Port number: 80
Protocol name: tcp
```

Viva Questions

Q1. What is socket?
Q2. How does the race condition occur?
Q3. What is multiprogramming?
Q4. Name the seven layers of the OSI model?
Q5. What is the difference between TCP and UDP?
Q6. What does socket consists of?
Q7. What is firewall?
Q8. How do I monitor the activity of sockets?
Q9. What is the role of TCP protocol and IP PROTOCOL?
Q10. How should I choose a port number for my server?
Q11. What is DHCP?
Q12. What does routing work?
Q13.What is VPN?
Q14. How do I open a socket?
Q15. How do I create an input stream?
Q16. How do I close a socket?
Q17. What is SMTP?
Q18. What is echo server?
Q19. What this function bind() does?
Q20. What this function socket() does?
Q21. What is IP address?
Q22. What are network host names?
Q23. How to find a machine address?
Q24. Difference between ARP and RARP.
Q25. What is MAC address?
Q26. What is multicasting?
Q27. What is DNS?
Q28. What is RMI?
Q29. How does TCP handshaking works?
Q30. What is sliding window protocol?
Q31. What is the difference between a Null and a void pointer?
Q32. Can I connect two computers to internet using same line?