College Of Engineering Trivandrum

NETWORK PROGRAMMING LAB

CS334

FINAL REPORT

Submitted By:
Neethu S
S6 CSE Roll No:42
TVE18CS043

Department of Computer Science

September 15, 2021

Contents

1 Basics of Network Configurations Files and Networking Commands in Linux

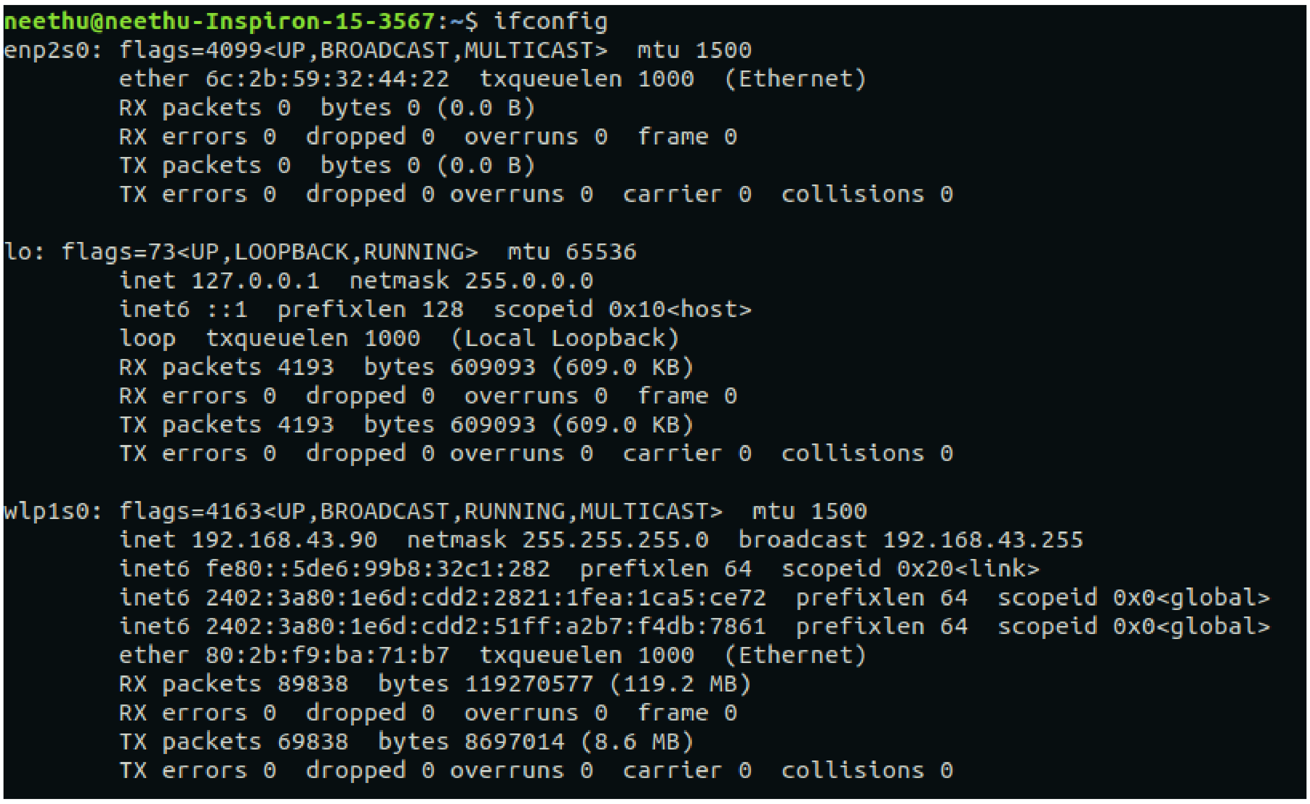
1.1 Aim

To get started with the basics of network configurations, files and networking commands in Linux.

1.2 Theory & Output

Basic Commands :

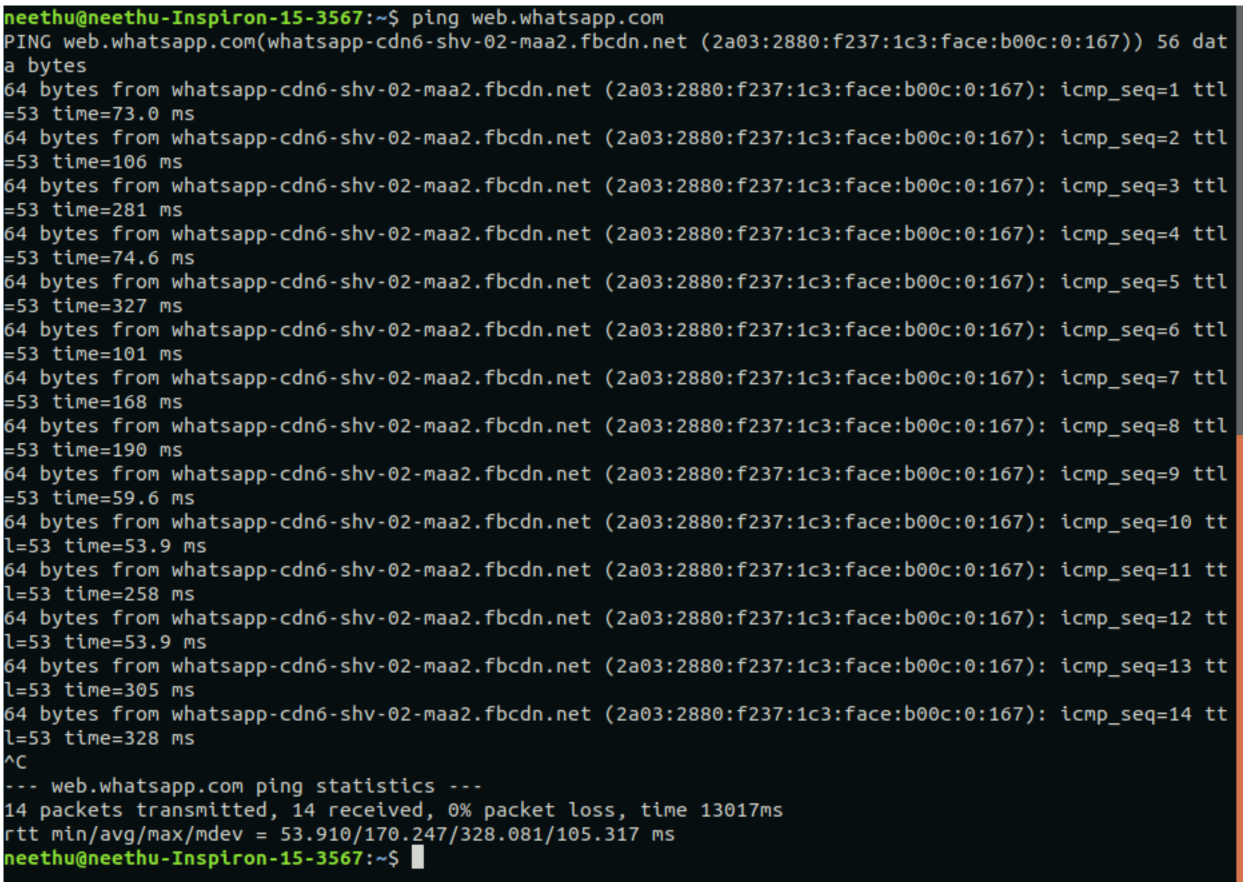
- ifconfig
  - Stands for ”interface configuation”.
  - Used to initialize an interface, assign IP address to interface and enable or disable interface on demand.
  - With this command you can view IP Address and Hardware / MAC address assign to interface and also MTU(Maximum transmission unit) size.



```
neethu@neethu-Inspiron-15-3567:~$ ifconfig
enp2s0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 6c:2b:59:32:44:22  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        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 4193  bytes 609093 (609.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4193  bytes 609093 (609.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlp1s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.43.90  netmask 255.255.255.0  broadcast 192.168.43.255
        inet6 fe80::5de6:99b8:32c1:282  prefixlen 64  scopeid 0x20<link>
        inet6 2402:3a80:1e6d:cdd2:2821:1fea:1ca5:ce72  prefixlen 64  scopeid 0x0<global>
        inet6 2402:3a80:1e6d:cdd2:51ff:a2b7:f4db:7861  prefixlen 64  scopeid 0x0<global>
        ether 80:2b:f9:ba:71:b7  txqueuelen 1000  (Ethernet)
        RX packets 89838  bytes 119270577 (119.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 69838  bytes 8697014 (8.6 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

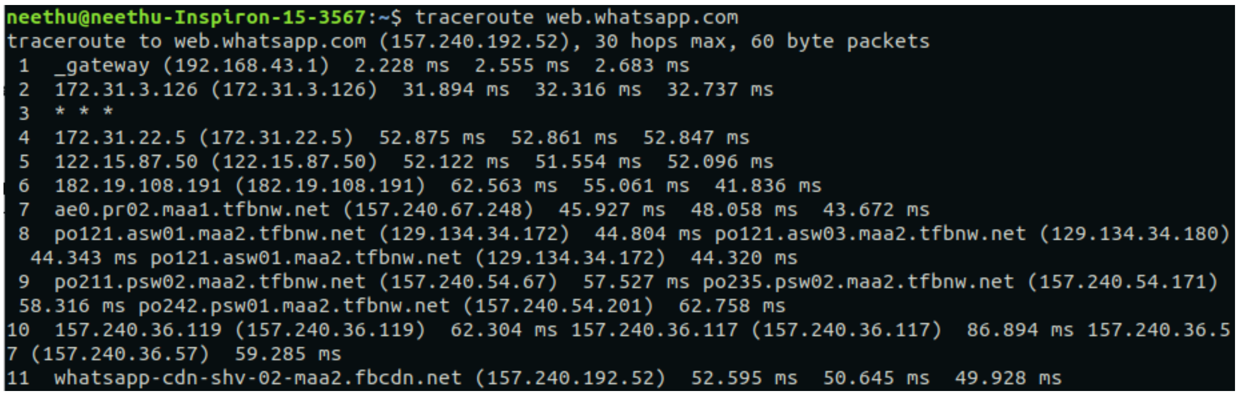
- ping

- It is used to test the connectivity of a network considered.
- It uses the ICMP (Internet Control Message Protocol) and sends a series of ICMP echo request messages to the target host and waits for an ICMP echo reply.
- It helps to calculate the package loss, round trip time of the packets, and whether the target host is reachable.



```
neethu@neethu-Inspiron-15-3567:~$ ping web.whatsapp.com
PING web.whatsapp.com(whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167)) 56 dat
a bytes
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=1 ttl
=53 time=73.0 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=2 ttl
=53 time=106 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=3 ttl
=53 time=281 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=4 ttl
=53 time=74.6 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=5 ttl
=53 time=327 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=6 ttl
=53 time=101 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=7 ttl
=53 time=168 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=8 ttl
=53 time=190 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=9 ttl
=53 time=59.6 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=10 tt
l=53 time=53.9 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=11 tt
l=53 time=258 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=12 tt
l=53 time=53.9 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=13 tt
l=53 time=305 ms
64 bytes from whatsapp-cdn6-shv-02-maa2.fbcdn.net (2a03:2880:f237:1c3:face:b00c:0:167): icmp_seq=14 tt
l=53 time=328 ms
^C
--- web.whatsapp.com ping statistics ---
14 packets transmitted, 14 received, 0% packet loss, time 13017ms
rtt min/avg/max/mdev = 53.910/170.247/328.081/105.317 ms
neethu@neethu-Inspiron-15-3567:~$
```

- traceroute
  - It is a network troubleshooting utility which shows number of hops taken to reach destination also determine packets traveling path.

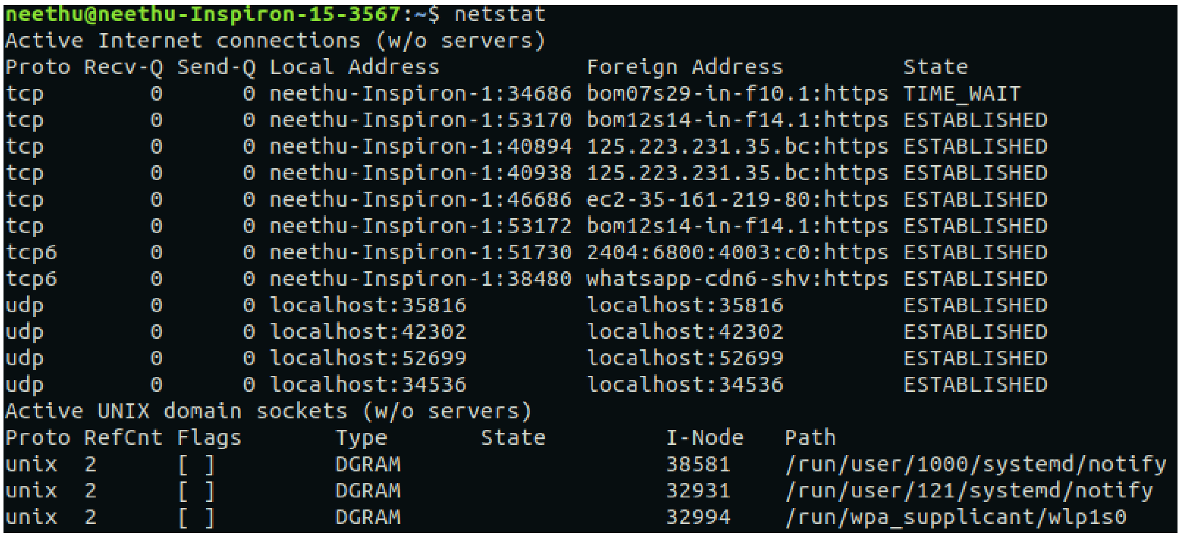


```
neethu@neethu-Inspiron-15-3567:~$ traceroute web.whatsapp.com
traceroute to web.whatsapp.com (157.240.192.52), 30 hops max, 60 byte packets
 1  _gateway (192.168.43.1)  2.228 ms  2.555 ms  2.683 ms
 2  172.31.3.126 (172.31.3.126)  31.894 ms  32.316 ms  32.737 ms
 3  * * *
 4  172.31.22.5 (172.31.22.5)  52.875 ms  52.861 ms  52.847 ms
 5  122.15.87.50 (122.15.87.50)  52.122 ms  51.554 ms  52.096 ms
 6  182.19.108.191 (182.19.108.191)  62.563 ms  55.061 ms  41.836 ms
 7  ae0.pr02.maa1.tfbnw.net (157.240.67.248)  45.927 ms  48.058 ms  43.672 ms
 8  po121.asw01.maa2.tfbnw.net (129.134.34.172)  44.804 ms po121.asw03.maa2.tfbnw.net (129.134.34.180)
  44.343 ms po121.asw01.maa2.tfbnw.net (129.134.34.172)  44.320 ms
 9  po211.psw02.maa2.tfbnw.net (157.240.54.67)  57.527 ms po235.psw02.maa2.tfbnw.net (157.240.54.171)
 58.316 ms po242.psw01.maa2.tfbnw.net (157.240.54.201)  62.758 ms
10  157.240.36.119 (157.240.36.119)  62.304 ms 157.240.36.117 (157.240.36.117)  86.894 ms 157.240.36.5
7 (157.240.36.57)  59.285 ms
11  whatsapp-cdn-shv-02-maa2.fbcdn.net (157.240.192.52)  52.595 ms  50.645 ms  49.928 ms
```

- netstat

– Netstat(Network Statistic) command displays connection info, routing table information etc.

– It displays various network related information such as network connections, routing tables, masquerade connections, multicast memberships, interface statistics etc.

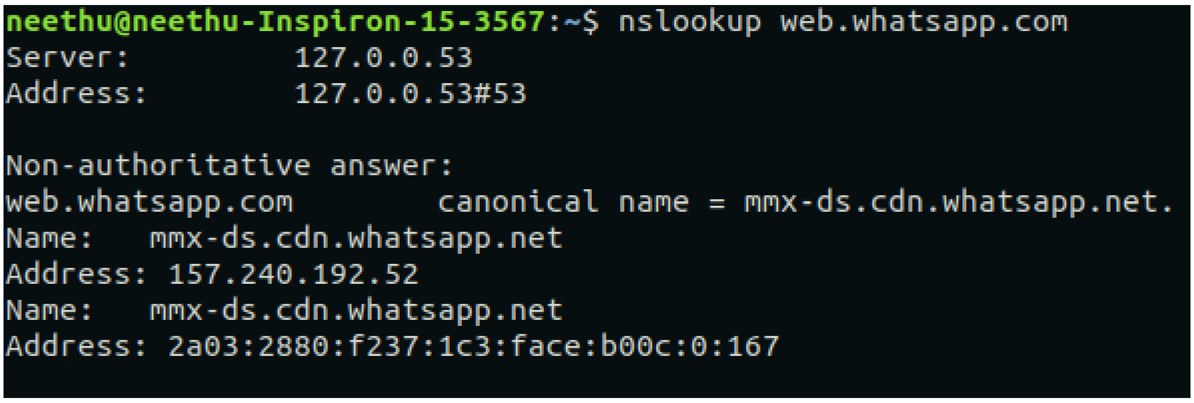


```
neethu@neethu-Inspiron-15-3567:~$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 neethu-Inspiron-1:34686 bom07s29-in-f10.1:https TIME_WAIT
tcp        0      0 neethu-Inspiron-1:53170 bom12s14-in-f14.1:https ESTABLISHED
tcp        0      0 neethu-Inspiron-1:40894 125.223.231.35.bc:https ESTABLISHED
tcp        0      0 neethu-Inspiron-1:40938 125.223.231.35.bc:https ESTABLISHED
tcp        0      0 neethu-Inspiron-1:46686 ec2-35-161-219-80:https ESTABLISHED
tcp        0      0 neethu-Inspiron-1:53172 bom12s14-in-f14.1:https ESTABLISHED
tcp6       0      0 neethu-Inspiron-1:51730 2404:6800:4003:c0:https ESTABLISHED
tcp6       0      0 neethu-Inspiron-1:38480 whatsapp-cdn6-shv:https ESTABLISHED
udp        0      0 localhost:35816         localhost:35816         ESTABLISHED
udp        0      0 localhost:42302         localhost:42302         ESTABLISHED
udp        0      0 localhost:52699         localhost:52699         ESTABLISHED
udp        0      0 localhost:34536         localhost:34536         ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   Path
unix  2      [ ]         DGRAM                    38581    /run/user/1000/systemd/notify
unix  2      [ ]         DGRAM                    32931    /run/user/121/systemd/notify
unix  2      [ ]         DGRAM                    32994    /run/wpa_supplicant/wlp1s0
```

• nslookup

– nslookup is a command-line administrative tool for testing and troubleshooting DNS servers (Domain Name Server).

– Most operating systems comes with built-in nslookup feature.

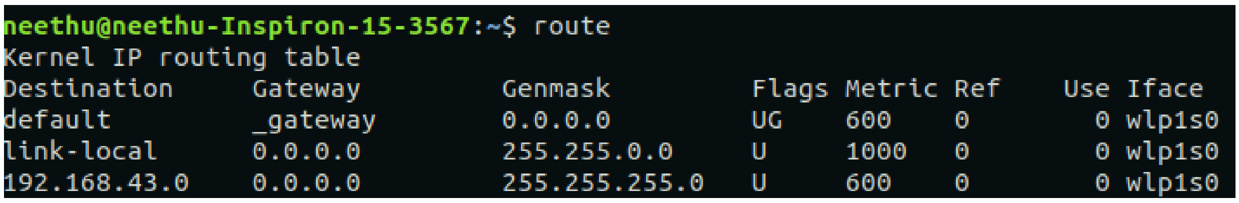


```
neethu@neethu-Inspiron-15-3567:~$ nslookup web.whatsapp.com
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
web.whatsapp.com        canonical name = mmx-ds.cdn.whatsapp.net.
Name:   mmx-ds.cdn.whatsapp.net
Address: 157.240.192.52
Name:   mmx-ds.cdn.whatsapp.net
Address: 2a03:2880:f237:1c3:face:b00c:0:167
```

• route

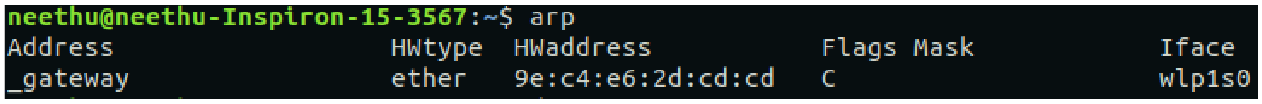
– Its basic function is to show or manipulate IP routing tables.

– It can add and delete routes and default Gateway.



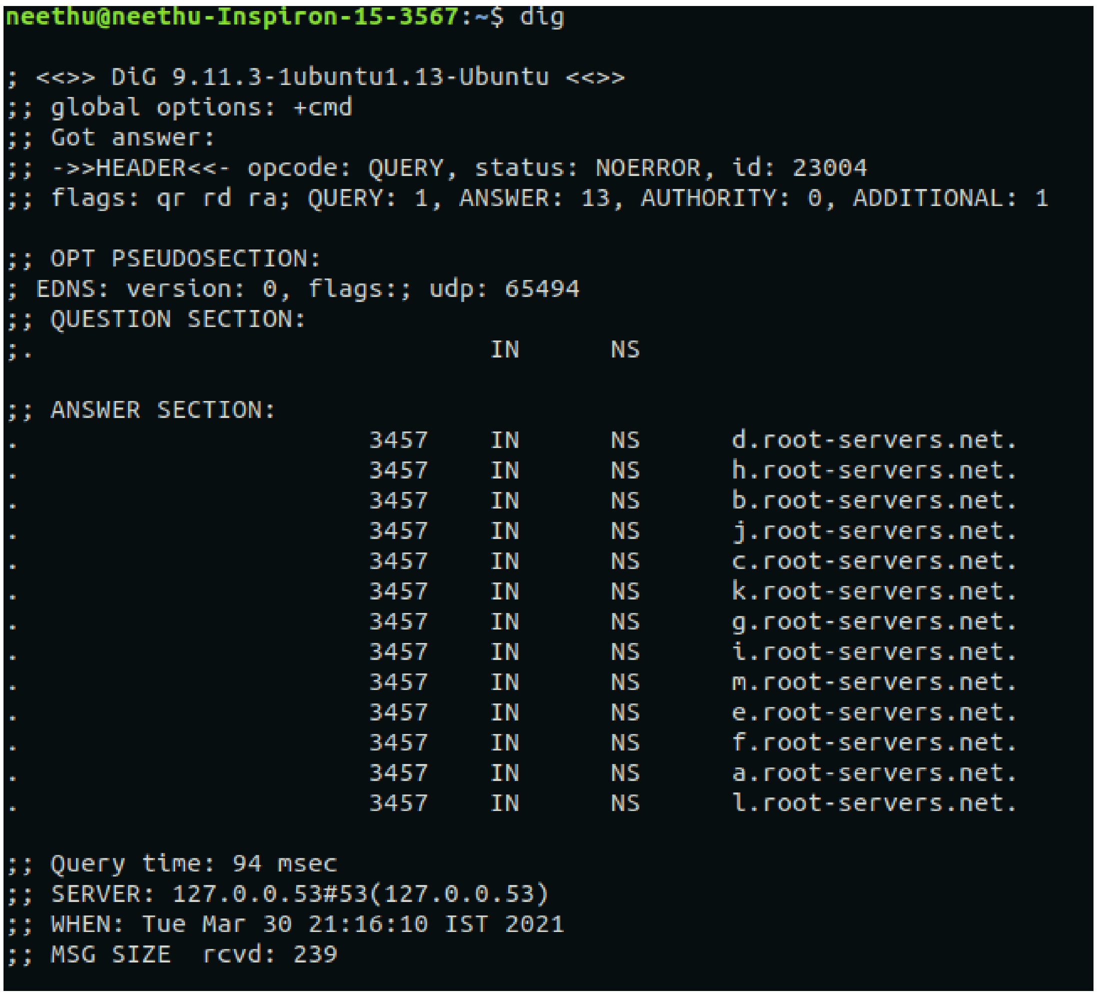
```
neethu@neethu-Inspiron-15-3567:~$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    600    0        0 wlp1s0
link-local      0.0.0.0         255.255.0.0     U     1000   0        0 wlp1s0
192.168.43.0    0.0.0.0         255.255.255.0   U     600    0        0 wlp1s0
```

- arp
  - It is a protocol used by network nodes to resolve IP addresses to their corresponding MAC addresses.
  - It can also be used to add an address for which to proxy arp and to forcefully add permanent entries to the ARP table.



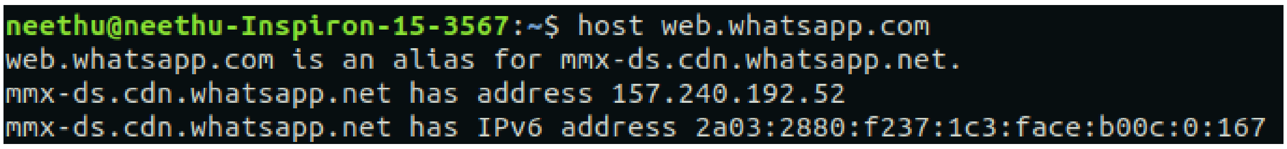
```
neethu@neethu-Inspiron-15-3567:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
_gateway                 ether   9e:c4:e6:2d:cd:cd   C                     wlp1s0
```

- dig
  - Dig (Domain Information Groper) query DNS related information like a record, CNAME, MX Record etc. This command mainly use to troubleshoot DNS related query.



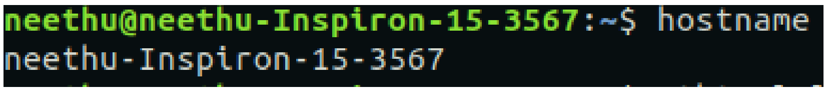
```
neethu@neethu-Inspiron-15-3567:~$ dig

; <<>> DiG 9.11.3-1ubuntu1.13-Ubuntu <<>>
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 23004
;; flags: qr rd ra; QUERY: 1, ANSWER: 13, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;.                              IN      NS

;; ANSWER SECTION:
.                       3457    IN      NS      d.root-servers.net.
.                       3457    IN      NS      h.root-servers.net.
.                       3457    IN      NS      b.root-servers.net.
.                       3457    IN      NS      j.root-servers.net.
.                       3457    IN      NS      c.root-servers.net.
.                       3457    IN      NS      k.root-servers.net.
.                       3457    IN      NS      g.root-servers.net.
.                       3457    IN      NS      i.root-servers.net.
.                       3457    IN      NS      m.root-servers.net.
.                       3457    IN      NS      e.root-servers.net.
.                       3457    IN      NS      f.root-servers.net.
.                       3457    IN      NS      a.root-servers.net.
.                       3457    IN      NS      l.root-servers.net.

;; Query time: 94 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
;; WHEN: Tue Mar 30 21:16:10 IST 2021
;; MSG SIZE  rcvd: 239
```

- host
  - Host command to find name to IP or IP to name in IPv4 or IPv6 and also query DNS records.



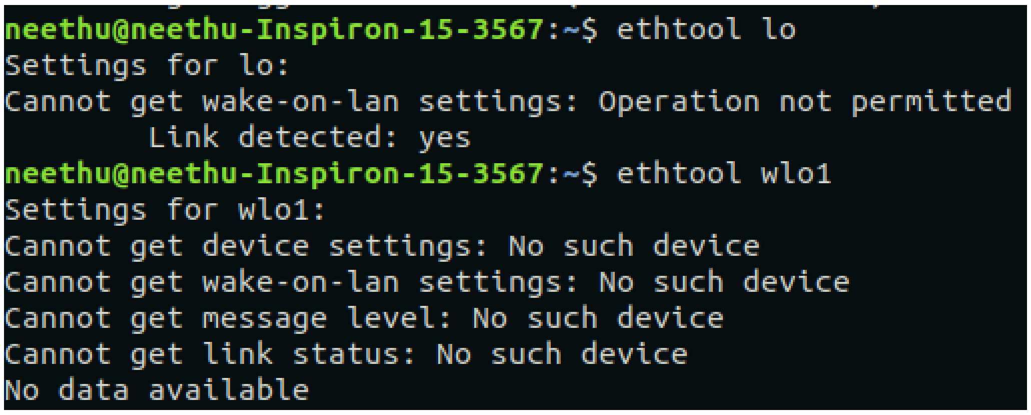
```
neethu@neethu-Inspiron-15-3567:~$ host web.whatsapp.com
web.whatsapp.com is an alias for mmx-ds.cdn.whatsapp.net.
mmx-ds.cdn.whatsapp.net has address 157.240.192.52
mmx-ds.cdn.whatsapp.net has IPv6 address 2a03:2880:f237:1c3:face:b00c:0:167
```

- hostname
  - hostname is to identify the host in a network.



```
neethu@neethu-Inspiron-15-3567:~$ hostname
neethu-Inspiron-15-3567
```

- ethtool
  - ethtool is used to view, setting speed and duplex of your Network Interface Card (NIC).



```
neethu@neethu-Inspiron-15-3567:~$ ethtool lo
Settings for lo:
Cannot get wake-on-lan settings: Operation not permitted
        Link detected: yes
neethu@neethu-Inspiron-15-3567:~$ ethtool wlo1
Settings for wlo1:
Cannot get device settings: No such device
Cannot get wake-on-lan settings: No such device
Cannot get message level: No such device
Cannot get link status: No such device
No data available
```

1.3 Result

Implemented the basic networking commands in Ubuntu 18.04 and the above outputs were obtained.

2 System Calls

2.1 Aim

To familiarize and understand the use and functioning of System Calls used for Operating System and Network Programming in Linux.

2.2 Theory

System Calls

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via the so called system calls. Whenever a program makes a system call, the mode will be switched from the user mode to kernel mode. Then the kernel satisfies the program request. After that, another switch takes place which results in change of mode from kernel mode back to user mode.

Types of System Calls

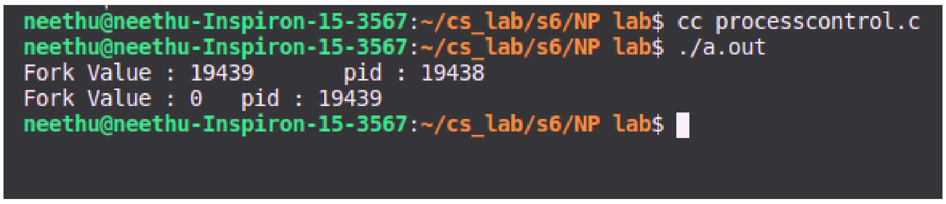
- Process Control
  - fork():
    - It is used to create processes, which becomes the child process of the caller. It takes no arguments and returns a process ID, which helps to distinguish the parent and child processes.
  - exit():
    - It is used to end or terminate a process.
    - In the case of a multithreading system, it is used to terminate a single thread.
  - wait():
    - It blocks the calling process until one of its child processes exits.
    - It takes the address of some integer variable and returns the process ID of the completed process.
- File Manipulation
  - open():
    - A call to it creates a new open file description, and is used to open a file.
    - The open() function returns a new file descriptor. Unsuccessful attempt returns -1.
  - close():
    - It is used to close the corresponding file descriptor.
    - Successful operation returns 0 and error returns -1.
  - read():
    - Used to retrieve data from a file stored in a file system.
    - The value returned is the number of bytes read (or -1 for error) and the file position is moved forward by this number.
  - write():
    - It writes data from a buffer to a file/device.
    - The value returned is the number of bytes written (or -1 for error) successfully.
- Device Manipulation
  - ioctl():

    * input/output control(ioctl) is a device-specific system call used for terminal I/O, hardware device control, and kernel extension operations.
    * Successful operation returns 0 and error returns -1.
  - read():
    * It is the same function used for file manipulation.
  - write():
    * It is the same function used for file manipulation.

- Information Maintenance
  - getpid():
    * It returns the process ID of the calling process.
    * It never throws any errors and hence is always successful.
  - alarm():
    * It sets an alarm clock for delivery of a signal in seconds.
    * Setting a new alarm cancels the previously defined alarms. It returns the number of seconds remaining until any previously scheduled alarm was due, or zero if none.
  - sleep():
    * It places a process to a suspended state for a specified interval of time in seconds.
    * Expiration of the time or an interrupt causes program to resume execution.
- Communication
  - pipe():
    * It facilitates inter-process communication where one process can write to and another related process can read from a so called 'virtual file'.
    * It returns -1 on error.
  - shmget():
    * It requests for a shared memory segment.
    * On a successful operation, its return value is the shared memory ID, otherwise, the return value is negative.

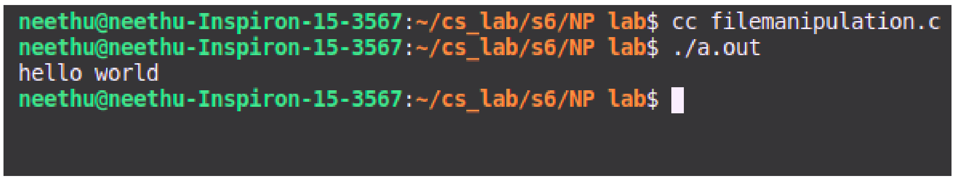
2.3 Code & Output

```
//C program to illustrate I/O System Calls - Process Control

#include<stdio.h>
#include<sys/types.h>
#include<stdlib.h>
#include<unistd.h>

int main(){
    int x = fork();
    wait();
    printf("Fork Value : %d\t pid : %d\n",x,getpid());
    exit(0);

    printf("Hello World!");
    return 0;
}
```



```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ cc processcontrol.c
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
Fork Value : 19439        pid : 19438
Fork Value : 0   pid : 19439
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ 
```

```
//C program to illustrate I/O System Calls - File Manipulation

#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<fcntl.h>

int main (void){
    int fd[2];
    char buf1[12] = "hello world\n";
    char buf2[12];

    fd[0] = open("exp2.txt", O_RDWR);
    fd[1] = open("exp2.txt", O_RDWR);

    write(fd[0], buf1, strlen(buf1));
    write(1, buf2, read(fd[1], buf2, 12));

    close(fd[0]);
    close(fd[1]);

    return 0;
}
/*The program prints "hello world" using system call
Content of exp2.txt:
    hello world
*/
```



```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ cc filemanipulation.c
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
hello world
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ 
```

2.4 Result

Implemented the program for system calls using C language in Ubuntu 18.04 with kernel and the above outputs were obtained.

3 Process and Thread

3.1 Aim

To familiarize and implement programs related to process and thread.

3.2 Theory

Thread: A thread is a path of execution within a process. It has its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history. It shares with its peer threads few information like code segment, data segment and open files.

Process: A process can be defined as an entity which represents the basic unit of work to be implemented in the system. Basically speaking, a process is a program in execution. For example, the original code we write and binary code which we process are both programs. When we actually run the binary code, it becomes a process.

Differences: A thread can do anything a process can do. But a process can consist of multiple threads, and hence thread could be considered a 'lightweight' process. Threads are used for small tasks, whereas processes are used for more 'heavyweight' tasks – basically the execution of applications. Another difference between a thread and a process is that threads within the same process share the same address space, whereas different processes do not. This allows threads to read from and write to the same data structures and variables, and also facilitates communication between threads. Communication between processes – also known as IPC, or inter-process communication is quite difficult and resource-intensive.

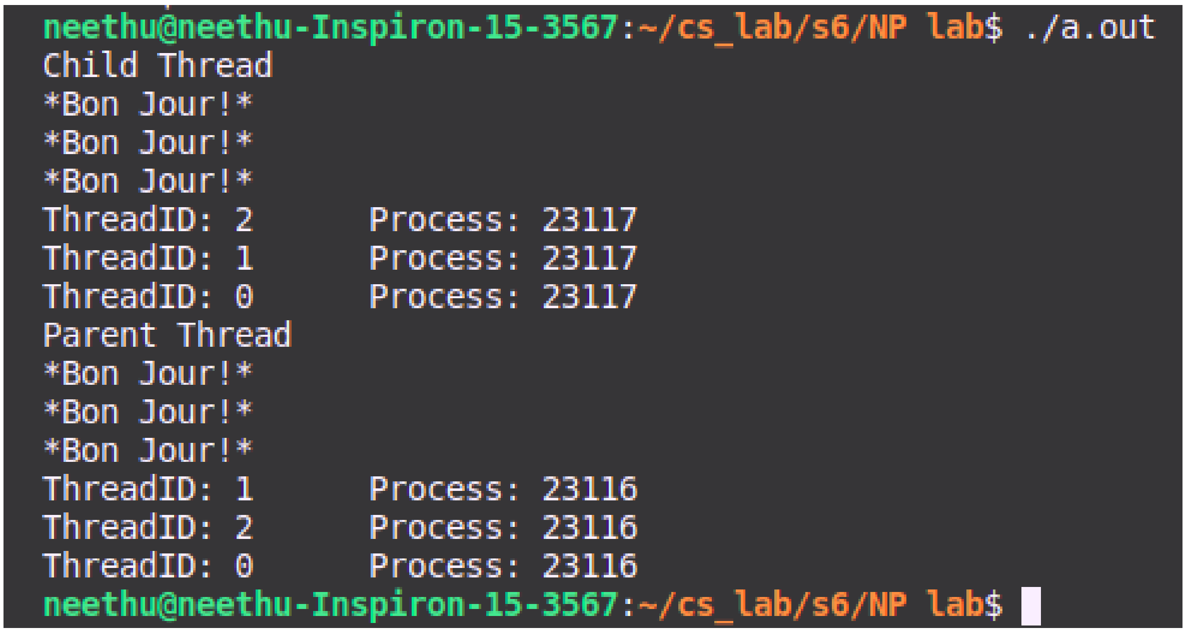
3.3 Algorithm

```
Algorithm for creating threads

1 START
2 Let n = 3
3 Procedure ThreadFunction (void arg)
4 BEGIN
5 Print some random text
6 sleep(1)
7 print Thread ID, Process ID
8 END ThreadFunction
9 Create pthread t
10 Let j = fork( )
11 IF j = 0 THEN //child process
12 BEGIN  for i < n //create thread for child process
13 CALL pthreadcreate(&t, NULL, ThreadFunction, i ) and increment i by 1
14 END  for
15 ELSE then // parent process
16 BEGIN  for i < n //create parent thread for process
17 CALL pthreadcreate(&t, NULL, ThreadFunction, i ) and increment i by 1
18 END  for
19 ENDIF
20 STOP
```

3.4 Code

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<sys/types.h>

const int n=3;

void *ThreadFunction(void *arg){
    printf("*Bon Jour!*\n");  //some random text
    sleep(1);
    printf("ThreadID: %d\t Process: %d\n",(void *)arg, getpid());
}

int main(){
    int i;
    pthread_t t;
    int j = fork();
    wait();
    if(j == 0){
        printf("Child Thread\n");
        for(i = 0; i < n; i++)
            pthread_create(&t, NULL, ThreadFunction, (void *)i);
    }
    else{
        printf("Parent Thread\n");
        for(i = 0; i < n; i++)
            pthread_create(&t, NULL, ThreadFunction, (void *)i);
    }

    pthread_exit(NULL);
    return 0;
}
```

3.5 Output



```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
Child Thread
*Bon Jour!*
*Bon Jour!*
*Bon Jour!*
ThreadID: 2      Process: 23117
ThreadID: 1      Process: 23117
ThreadID: 0      Process: 23117
Parent Thread
*Bon Jour!*
*Bon Jour!*
*Bon Jour!*
ThreadID: 1      Process: 23116
ThreadID: 2      Process: 23116
ThreadID: 0      Process: 23116
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$
```

3.6 Result

Implemented the program for creating threads using C language in Ubuntu 18.04 with kernel and the above outputs were obtained.

4 Pipes, Message Queues and Shared Memory

4.1 Aim

To implement programs for Inter Process Communication using PIPE, Message Queue and Shared Memory.

- Write a program for communication between a child process and its parent process with the use of Ordinary Pipes.
- Program to show IPC using Named pipes.
- Program to show IPC with the help of message queue. Here there are two processes- writer and reader.
- Program to show IPC with the help of shared memory. Here there are two processes- writer and reader.

4.2 Theory

Pipes

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes, or even be multi-level such as communication between the parent, child, grand-child, etc. Two common types of pipes used on both UNIX and Windows systems: ordinary pipes and named pipes.

Ordinary Pipes: Ordinary pipes allow two processes to communicate in a producer consumer fashion where the producer writes to one end of the pipe and the consumer reads from the other end. They are unidirectional. On UNIX systems, ordinary pipes are constructed using the function

pipe(int fd[ ]), where
fd[0] is the read-end of the pipe,
and fd[1] is the write-end.

The major limitation of an ordinary pipe is that they are always local, ie. they cannot be used for communication over a network.

Named Pipes: A named pipe is a more powerful communication tool as it can be bidirectional and does not require any parent-child relationship. Once a named pipe is established, several processes can use it for communication. A named pipe exists in the file system. After input-output has been performed by the sharing processes, the pipe still exists in the file system independently of the process, and can be used for communication between some other processes. In order to create a namped pipe we make use of the mkfifo() function. A FIFO special file is entered into the filesystem by calling it. Once we have created it, any process can open it for reading or writing from both ends.

Message Queues

It is a component used for IPC, or for inter-thread communication within the same process. It makes use of a queue for messaging – the passing of control or of content. Message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.

Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. A new queue is created or an existing queue opened by msgget(). New messages are added to the end of a queue by msgsnd() and are fetched from the queue by msgrcv(). We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Shared Memory

Shared memory is a component used in IPC, where two or more process can access the common memory. Here, communication is done via this shared memory where changes made by one process can be viewed by anther process. The function shmget() is used to create and return an identifier for the shared memory segment. Before one can use a shared memory segment, we should attach to it using the function, shmat(). Ususally the memory address to be used is automatically chosen by the OS. After this, we can use the memory segment for communication. When we are done with the shared memory segment, the program should detach itself from it using shmdt(). In order to destroy the shared memory segment, shmctl() is used.

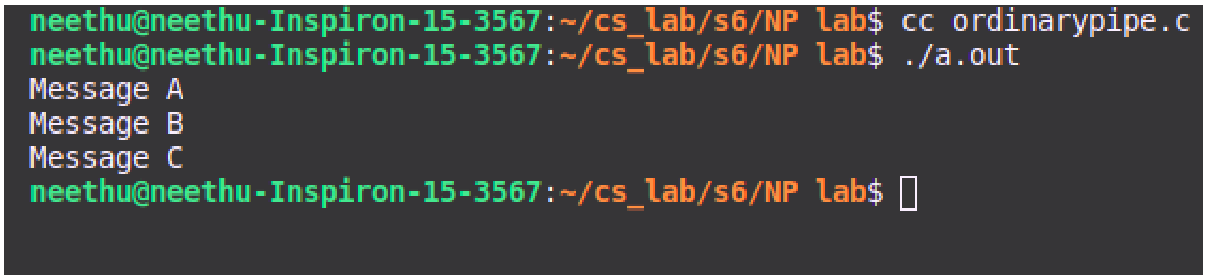
4.3 Program & Output

4.3.1 Ordinary Pipes

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#define size 16

char* msgA = "Message A";
char* msgB = "Message B";
char* msgC = "Message C";

int main(){
    char inbuf[size];
    int p[2], i;

    if(pipe(p) < 0)
        exit(1);
    write(p[1], msgA, size);
    write(p[1], msgB, size);
    write(p[1], msgC, size);
    for(i = 0; i < 3; i++){
        read(p[0], inbuf, size);
        printf("%s\n", inbuf);
    }
    return 0;
}
```

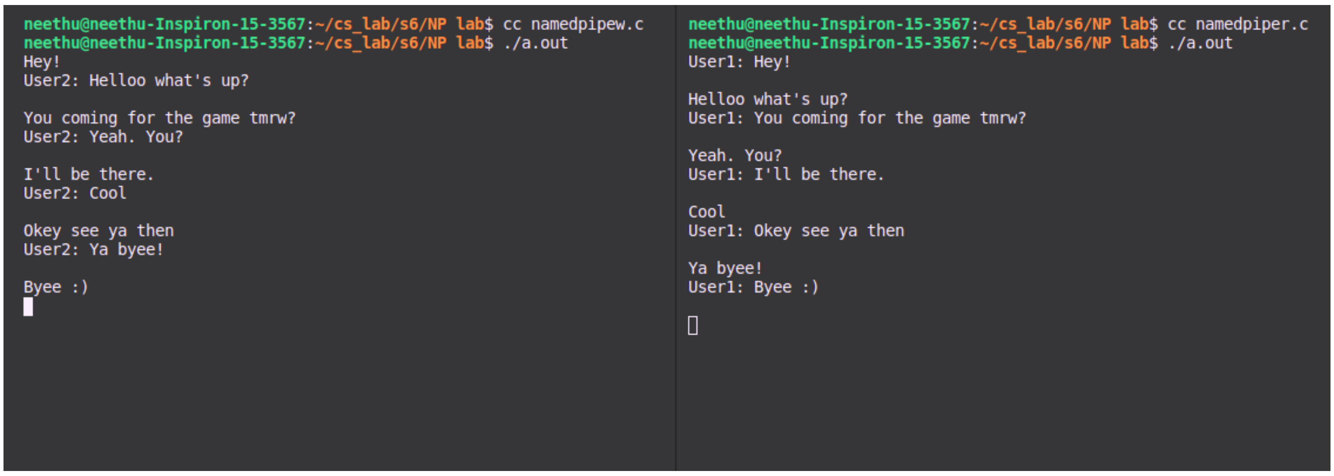
4.3.2 Named Pipes

```
// This side writes first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
```



```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ cc ordinarypipe.c
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
Message A
Message B
Message C
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$
```

```c
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int fd;

    char * myfifo = "myfifo";

    mkfifo(myfifo, 0666);

    char arr1[80], arr2[80];
    while (1){
         fd = open(myfifo, O_WRONLY);
         fgets(arr2, 80, stdin);

         write(fd, arr2, strlen(arr2)+1);
         close(fd);

         fd = open(myfifo, O_RDONLY);
         read(fd, arr1, sizeof(arr1));

         printf("User2: %s\n", arr1);
         close(fd);
    }
    return 0;
}

// This side reads first, then writes
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
    int fd1;

    char * myfifo = "myfifo";

    mkfifo(myfifo, 0666);
    char str1[80], str2[80];
    while (1){
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);

        printf("User1: %s\n", str1);
        close(fd1);

        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
    }
    return 0;
}
```

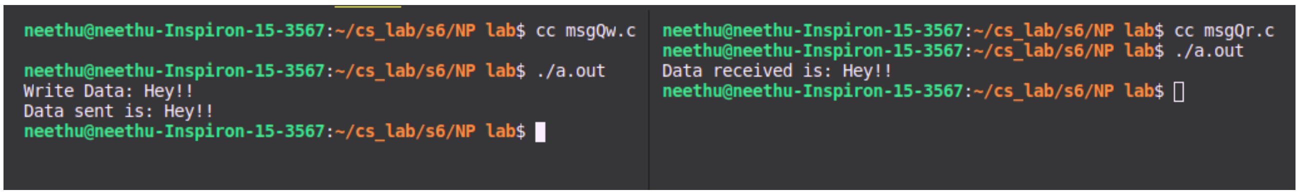


```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ cc namedpipew.c
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
Hey!
User2: Helloo what's up?

You coming for the game tmrw?
User2: Yeah. You?

I'll be there.
User2: Cool

Okey see ya then
User2: Ya byee!

Byee :)
```

```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ cc namedpiper.c
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
User1: Hey!

Helloo what's up?
User1: You coming for the game tmrw?

Yeah. You?
User1: I'll be there.

Cool
User1: Okey see ya then

Ya byee!
User1: Byee :)
```

4.3.3 Message Queue

```
//Writer program
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>

struct msg_buffer{
    long msgType;
    char msgText[100];
}msg;

int main(){
    key_t key;
    int msgID;

    key = ftok("pgmfile", 65);
    msgID = msgget(key, 0666 | IPC_CREAT);
    msg.msgType = 1;

    printf("Write Data: ");
    scanf("%s",msg.msgText);
    msgsnd(msgID, &msg, sizeof(msg), 0);

    printf("Data sent is: %s\n", msg.msgText);
    return 0;
}

//Reader Program
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>

struct msg_buffer{
    long msgType;
    char msgText[100];
}msg;

int main(){
    key_t key;
    int msgID;

    key = ftok("pgmfile", 65);
    msgID = msgget(key, 0666 | IPC_CREAT);

    msgrcv(msgID, &msg, sizeof(msg), 1, 0);

    printf("Data received is: %s\n", msg.msgText);
    msgctl(msgID, IPC_RMID, NULL);

    return 0;
}
```

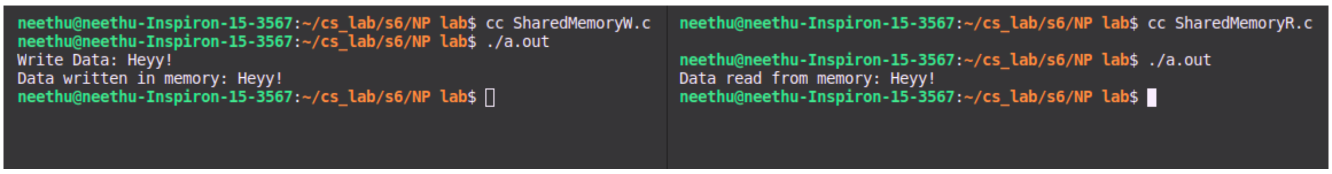


```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ cc msgQw.c

neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
Write Data: Hey!!
Data sent is: Hey!!
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$
```

```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ cc msgQr.c
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
Data received is: Hey!!
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$
```

4.3.4 Shared Memory

```
//Writer Program
#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>

int main(){
    key_t key = ftok("shmfile", 65);
    int shmID = shmget(key, 1024, 0666 | IPC_CREAT);
    char * str = (char *)shmat(shmID, (void *)0, 0);

    printf("Write Data: ");
    scanf("%s", str);
    printf("Data written in memory: %s\n", str);
    shmdt(str);

    return 0;
}

//Reader Program
#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>

int main(){
    key_t key = ftok("shmfile", 65);
    int shmID = shmget(key, 1024, 0666 | IPC_CREAT);
    char * str = (char *)shmat(shmID, (void *)0, 0);

    printf("Data read from memory: %s\n", str);
    shmdt(str);
    shmctl(shmID, IPC_RMID, NULL);

    return 0;
}
```



```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ cc SharedMemoryW.c
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
Write Data: Heyy!
Data written in memory: Heyy!
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$
```

```
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ cc SharedMemoryR.c

neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$ ./a.out
Data read from memory: Heyy!
neethu@neethu-Inspiron-15-3567:~/cs_lab/s6/NP lab$
```

4.4 Result

Implemented the program for the inter-process communication using Pipes, Message Queues and Shared Memory using C language in Ubuntu 18.04 with kernel and the above outputs were obtained.

5 First Readers-Writers Problem

5.1 Aim

To implement the First Readers-Writers Problem.

5.2 Theory

The Readers-Writers problem is a classic synchronization problem.

The problem statement : There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context: the reader and the writer. Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource. When a writer is writing data to the resource, no other process can access the resource. A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time.

Solution : Here, the first reader locks the resource if such is available. Once the file is locked from writers, it may be used by many subsequent readers without having them to re-lock it again.

Before entering the critical section, every new reader must go through the entry section. However, there may only be a single reader in the entry section at a time so as to avoid race conditions. In order to do so, every reader which enters the ENTRY Section will lock the ENTRY Section for themselves but does not lock the resource. Once the reader is done executing the entry section, it will unlock it by signalling the mutex. There can be no more than a single reader in the exit section at a time, therefore, every reader must claim and lock the Exit section for themselves before using it.

Once the first reader is in the entry section, it will lock the resource. Doing this will prevent any writers from accessing it. Subsequent readers can just utilize the locked resource. The very last reader must unlock the resource, thus making it available to writers.

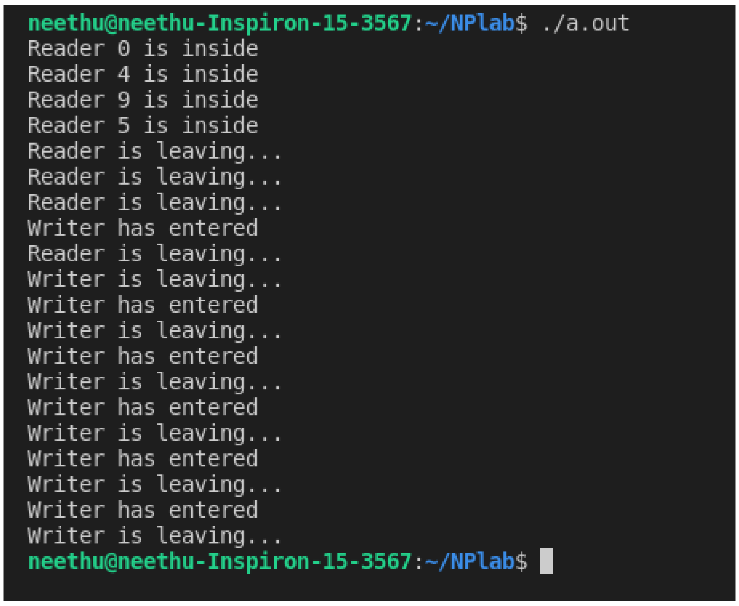
5.3 Algorithm

```
START
semaphore resource = NULL;
semaphore rmutex = NULL;
readcount =0;
procedure WRITER
  resource.P( );
  <CRITICAL SECTION>
  <EXIT SECTION>
  resource.V( );
END procedure
procedure READER
  rmutex.P( );
  <CRITICAL SECTION>
  readcount++;
  IF readcount == 1 THEN
      resource.P( );
  END IF
  <EXIT CRITICAL SECTION>
```

```
 rmutex.V( );
 rmutex.P( );
 <CRITICAL SECTION>
 readcount;
 IF readcount == 0 THEN
     resource.V();
 END IF
 <EXIT CRITICAL SECTION>
 rmutex.v();
END procedure
STOP
. resource.P() is equivalent to wait(resource)
. rmutex.P() is equivalent to wait(rmutex)
. resource.V() is equivalent to signal(resource)
. rmutex.V() is equivalent to signal(rmutex)
```

5.4 Program & Output

```
//First Readers-Writers Problem

#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<unistd.h>

pthread_mutex_t mutex, res;
pthread_t tid;
int readcount = 0;

void *reader(void *var){
    setbuf(stdout, NULL);
    pthread_mutex_lock(&mutex);
    readcount++;
    if (readcount == 1)
        pthread_mutex_lock(&res);\
    printf("Reader %d is inside\n", var);
    pthread_mutex_unlock(&mutex);
    sleep(2);
    pthread_mutex_lock(&mutex);
    readcount--;
    if(readcount == 0)
        pthread_mutex_unlock(&res);
    pthread_mutex_unlock(&mutex);
    printf("Reader is leaving...\n");
    pthread_exit(NULL);
}

void *writer(void *var){
    setbuf(stdout, NULL);
    pthread_mutex_lock(&res);
    printf("Writer has entered\n");
    sleep(2);
    printf("Writer is leaving...\n");
    fflush(stdout);
    pthread_mutex_unlock(&res);
    pthread_exit(NULL);
}

int main(){
    setbuf(stdout, NULL);
    int n1, n2, i, j;
    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&res, NULL);
    srand(time(NULL));
    for(i=0; i<10; i++){
        j = rand();
        if(j%2 == 1)
            pthread_create(&tid, NULL, reader, (void *)i);
        else
            pthread_create(&tid, NULL, writer, NULL);
    }
    pthread_exit(NULL);
}
```



```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./a.out
Reader 0 is inside
Reader 4 is inside
Reader 9 is inside
Reader 5 is inside
Reader is leaving...
Reader is leaving...
Reader is leaving...
Writer has entered
Reader is leaving...
Writer is leaving...
Writer has entered
Writer is leaving...
Writer has entered
Writer is leaving...
Writer has entered
Writer is leaving...
Writer has entered
Writer is leaving...
Writer has entered
Writer is leaving...
neethu@neethu-Inspiron-15-3567:~/NPlab$
```

5.5 Result

Implemented the program for the first Readers-Writers problem using C language in Ubuntu 20.04 with kernel and the above outputs were obtained.

6 Second Readers-Writers Problem

6.1 Aim

To implement the Second Readers-Writers Problem.

6.2 Theory

The Readers-Writers problem is a classic synchronization problem.

The problem statement : There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context: the reader and the writer. Any number of readers can read from the shared resource simultaneously, but only one writer can write to the shared resource.When a writer is writing data to the resource, no other process can access the resource. A writer cannot write to the resource if there are non zero number of readers accessing the resource at that time. Also, no writer, once added to the queue, shall be kept waiting longer than absolutely necessary. This is also called writers-preference.
Solution : In this solution, preference is given to the writers. This is done by forcing every reader to lock and release the readTry mutex individually. In the case of writers, only the first writer will lock the readTry and then all subsequent writers can simply use the resource as it gets freed by the previous writer. The very last writer must release the readTry semaphore, thus opening the gate for readers to try reading.

The resource semaphore can be locked by both the writer and the reader in their entry section. They are only able to do so after first locking the readTry semaphore, which can only be done by one of them at a time. If there are no writers wishing to get to the resource, as indicated to the reader by the status of the readtry semaphore, then the readers will not lock the resource. As soon as a writer shows up, it will try to set the readtry and wait for the current reader to release the readtry.

The rmutex and wmutex are used in exactly the same way as in the first readers writers problem. Their sole purpose is to avoid race conditions on the readers and writers while they are in their entry or exit sections.

6.3 Algorithm

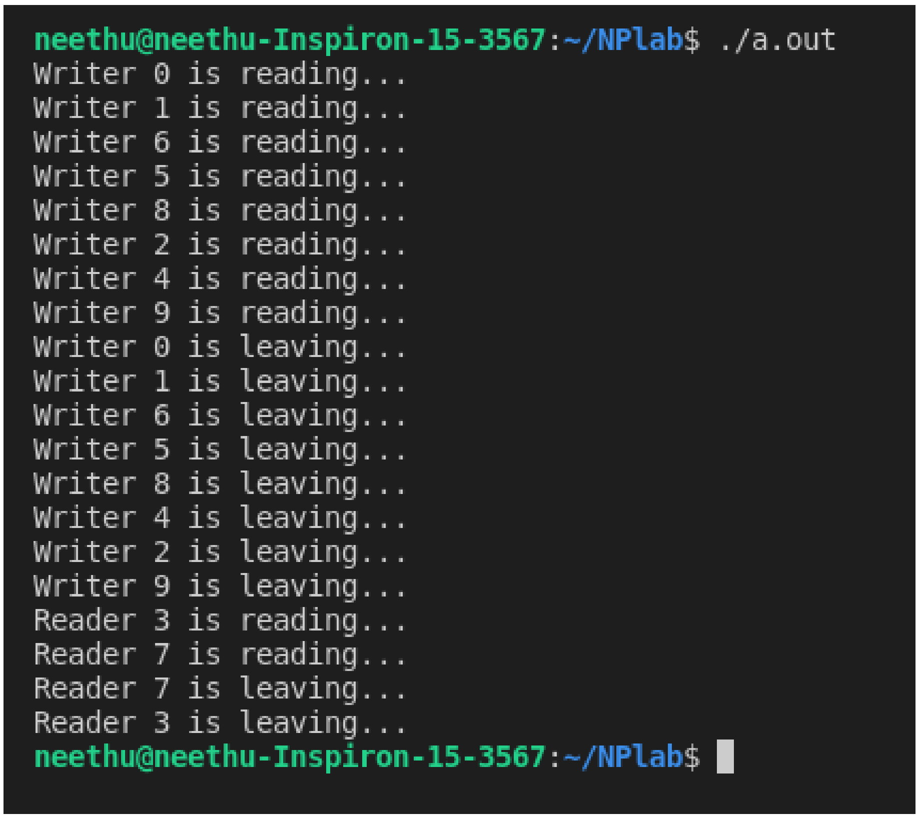
```
semaphore resource=NULL, rmutex=NULL, wmutex=NULL, res=NULL;
readcount =0, writecount=0;
procedure READER
  <ENTRY Section>
  readTry.P()
  rmutex.P()
  readcount++
  if readcount == 1 then
      resource.P()
  end if
  rmutex.V()
  readTry.V()
  <CRITICAL Section>
  <EXIT Section>
  rmutex.P()
  readcount--
  if readcount == 0 then
```

```
     resource.V()
 end if
 rmutex.V()
end procedure
procedure WRITER
 <ENTRY Section>
 wmutex.P()
 writecount++
 if writecount == 1 then
     readTry.P()
 end if
 wmutex.V()
 <CRITICAL Section>
 resource.P()
 resource.V()
 <EXIT Section>
 writecount[U+FFFD];
 if writecount == 1 then
     readTry.V() ;
 end if
 wmutex.V() ;
end procedure
```

6.4 Program & Output

```
//Second Readers-Writers Problem

#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<unistd.h>

pthread_mutex_t rmutex, wmutex, res, readTry;
pthread_t tid;
int readcount = 0, writecount = 0;

void *reader(void *var){
    setbuf(stdout, NULL);
    pthread_mutex_lock(&readTry);
    pthread_mutex_lock(&rmutex);
    readcount++;
    if(readcount == 1)
        pthread_mutex_lock(&res);
    printf("Reader %d is reading...\n",var);
    pthread_mutex_unlock(&rmutex);
    pthread_mutex_unlock(&readTry);
    sleep(2);
    pthread_mutex_lock(&rmutex);
    readcount--;
    printf("Reader %d is leaving...\n", var);
    if(readcount == 0)
```

```
        pthread_mutex_unlock(&res);
    pthread_mutex_unlock(&rmutex);
}

void *writer(void *var){
    setbuf(stdout, NULL);
    pthread_mutex_lock(&wmutex);
    writecount++;
    printf("Writer %d is reading...\n", var);
    if(writecount == 1)
        pthread_mutex_lock(&readTry);
    pthread_mutex_unlock(&wmutex);
    pthread_mutex_lock(&res);
    sleep(2);
    pthread_mutex_unlock(&res);
    pthread_mutex_lock(&wmutex);
    writecount--;
    printf("Writer %d is leaving...\n", var);
    if(writecount == 0)
        pthread_mutex_unlock(&readTry);
    pthread_mutex_unlock(&wmutex);
    pthread_exit(NULL);
}

int main(){
    setbuf(stdout, NULL);
    int n1, n2, i, j;
    pthread_mutex_init(&rmutex, NULL);
    pthread_mutex_init(&wmutex, NULL);
    pthread_mutex_init(&res,NULL);
    srand(time(NULL));
    for(i=0; i<10; i++){
        j = rand();
        if(j%2 == 1)
            pthread_create(&tid, NULL, reader, (void *)i);
        else
            pthread_create(&tid, NULL, writer, (void *)i);
    }
    pthread_exit(NULL);
}
```



```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./a.out
Writer 0 is reading...
Writer 1 is reading...
Writer 6 is reading...
Writer 5 is reading...
Writer 8 is reading...
Writer 2 is reading...
Writer 4 is reading...
Writer 9 is reading...
Writer 0 is leaving...
Writer 1 is leaving...
Writer 6 is leaving...
Writer 5 is leaving...
Writer 8 is leaving...
Writer 4 is leaving...
Writer 2 is leaving...
Writer 9 is leaving...
Reader 3 is reading...
Reader 7 is reading...
Reader 7 is leaving...
Reader 3 is leaving...
neethu@neethu-Inspiron-15-3567:~/NPlab$
```

6.5 Result

Implemented the program for the Second Readers-Writers problem using C language in Ubuntu 20.04 with kernel and the above outputs were obtained.

7 Socket Programming: TCP

7.1 Aim

To implement Client-Server communication using Socket Programming and TCP as transport layer protocol.

7.2 Theory

TCP (Transmission Control Protocol) is one of the main protocols in TCP/IP networks. Transmission control protocol (TCP) is a network communication protocol designed to send data packets over the Internet. TCP is a transport layer protocol in the OSI layer and is used to create a connection between remote computers by transporting and ensuring the delivery of messages over supporting networks and the Internet.

It is a connection-oriented protocol, which means that a connection is established and maintained until the application programs at each end have finished exchanging messages.

TCP programs are implemented in two parts:

• Server: A server program listens for a connection. On getting a request, the server accepts a connection. After the connection is established, server and client can exchange messages.

• Client: A client program requests some service. A client program request for some resources to the server and server responds to that request. Client initiates the connection establishment. The server accepts connection and client can request services by exchanging messages.

Sockets
Implementation of the above two programs (Client and Server) is to be done with the help of sockets. A socket is one endpoint of a two-way communication link between two programs running on the network. Client program and server program create and use sockets to communicate with each other.

7.3 Algorithm

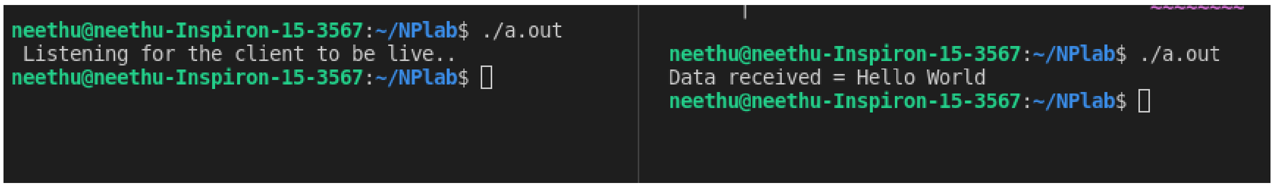
```
Algorithm for the client

1 START
2 Create the socket using the function socket()
3 Configure socket details
4 Connect the socket to the server using function connect()
5 Receive the message from the server using recv()
6 Print the received message
7 STOP


Algorithm for the Server

1 START
2 Configure socket details
3 Set all bits of the padding field to 0 using memset ()
4 Bind the address struct to the socket using bind ()
5 Listen to the socket
```

```
6 A new socket for the incoming connections created using accept()
7 Send message to the incoming connection using send()
8 STOP
```

7.4 Program & Output

```
//Socket Programming:TCP
//Server Side

#include<stdio.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<string.h>

int main(){
    int welcomeSocket, newSocket;
    char buffer[1024];
    struct sockaddr_in serverAddr;
    socklen_t addr_size;
    struct sockaddr_storage serverStorage;

    welcomeSocket = socket(PF_INET, SOCK_STREAM, 0);
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(7891);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);

    bind(welcomeSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr));

    if (listen(welcomeSocket, 5 ) == 0)
        printf("Listening for the client to be live..\n");
    else
        printf("Error\n");

    addr_size = sizeof serverStorage;
    newSocket = accept(welcomeSocket,
                (struct sockaddr*)&serverStorage, &addr_size);

    strcpy(buffer, "Hello World\n");
    send(newSocket, buffer, 13, 0);

    return 0;
}


//Client Side

#include<stdio.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<string.h>
```

```
int main(){
    int clientSocket;
    char buffer[1024];
    struct sockaddr_in serverAddr;
    socklen_t addr_size;

    //Create socket
    clientSocket = socket(PF_INET, SOCK_STREAM, 0);

    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(7891);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);
    addr_size = sizeof serverAddr;
    connect(clientSocket, (struct sockaddr*)&serverAddr, addr_size);
    recv(clientSocket, buffer, 1024, 0);
    printf("Data received = %s", buffer);

    return 0;
}
```



```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./a.out
 Listening for the client to be live..
neethu@neethu-Inspiron-15-3567:~/NPlab$
```

```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./a.out
Data received = Hello World
neethu@neethu-Inspiron-15-3567:~/NPlab$
```

7.5 Result

Implemented the program for the Client-Server communication using Socket Programming and TCP as transport layer protocol using C language in Ubuntu 18.04 with kernel and the above outputs were obtained.

8 Socket Programming : UDP

8.1 Aim

To implement Client-Server communication using Socket Programming and UDP as transport layer protocol.

8.2 Theory

UDP (User Datagram Protocol) is an alternative communications protocol to Transmission Control Protocol (TCP) used primarily for establishing low-latency and loss-tolerating connections between applications on the internet. UDP enables process-to-process communication. UDP sends messages, called datagrams, and is considered a best-effort mode of communications. It is considered a connectionless protocol because it doesn't require a virtual circuit to be established before any data transfer occurs.

Server & Client: Since the UDP is a connectionless protocol, they do not require a connection to get established prior to data transmission or reception. Hence data can be sent between them directly.

8.3 Algorithm

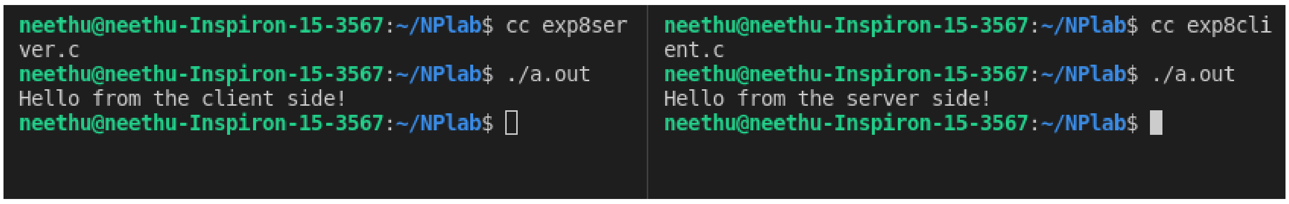
```
Algorithm for the Client

1 START
2 Create the socket using socket()
3 Configure socket details
4 Connect the socket to server using function connect()
5 Send the message to the server using sendto()
6 Receive the response from server using recvfrom()
7 Print the response
8 STOP

Algorithm for the Server

1 START
2 Create the UDP socket
3 Configure the socket details
4 Bind the address struct to the socket using bind()
5 Receive from client using recvfrom()
6 Print the received message
7 Send response using sendto()
8 STOP
```

8.4 Program & Output

```
//Socket Programming:UDP
//Server Side

#include<stdio.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<string.h>
```

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

int main(){
    char buffer[1000];
    char *message = "Hello from the server side!";
    int serversocket, len;
    struct sockaddr_in servaddr, cliaddr;
    bzero(&servaddr, sizeof(servaddr));

    serversocket = socket(AF_INET, SOCK_DGRAM, 0);
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(8080);
    servaddr.sin_family = AF_INET;

    bind(serversocket, (struct sockaddr*)&servaddr, sizeof(servaddr));

    len = sizeof(cliaddr);
    int n = recvfrom(serversocket, (char *)buffer, sizeof(buffer), 0,
            (struct sockaddr*)&cliaddr, &len);

    buffer[n] = '\0';
    printf("%s\n", buffer);

    sendto(serversocket, message, strlen(message), 0,
                        (struct sockaddr*)&cliaddr, len);
    return 0;
}

//Socket Programming:UDP
//Client Side

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

int main(){
    char buffer[1000];
    char *message = "Hello from the client side!";
    int clientsocket, n;
    struct sockaddr_in servaddr;

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(8080);
    servaddr.sin_family = AF_INET;
    clientsocket = socket(AF_INET, SOCK_DGRAM, 0);

    if(connect(clientsocket, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1){
        printf("\nError: Connection failed\n");
        exit(0);
    }

    int len = sizeof(servaddr);
    sendto(clientsocket, message, strlen(message), 0,
                                (struct sockaddr*)&servaddr, sizeof(servaddr));

    n = recvfrom(clientsocket, (char *)buffer, sizeof(buffer), 0,
                                            (struct sockaddr*)&servaddr, &len);
    buffer[n] = '\0';
    printf("%s\n", buffer);

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

8.5 Output



```
neethu@neethu-Inspiron-15-3567:~/NPlab$ cc exp8ser
ver.c
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./a.out
Hello from the client side!
neethu@neethu-Inspiron-15-3567:~/NPlab$
```

```
neethu@neethu-Inspiron-15-3567:~/NPlab$ cc exp8cli
ent.c
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./a.out
Hello from the server side!
neethu@neethu-Inspiron-15-3567:~/NPlab$
```

8.6 Result

Implemented the program for the Client-Server communication using Socket Programming and UDP as transport layer protocol using C language in Ubuntu 20.04 with kernel and the above outputs were obtained.

9 Multi user chat server using TCP

9.1 Aim

To implement a multi user chat server using TCP as transport layer protocol.

9.2 Theory

TCP (Transmission Control Protocol) works with the Internet Protocol (IP), which defines how computers send packets of data to each other. Together, TCP and IP are the basic rules defining the Internet. It is a connection-oriented protocol, which means that a connection is established and maintained until the application programs at each end have finished exchanging messages.

Server - In a simple multi user chat system, the server usually has the role to receive the messages sent by the clients and send it to all other clients. So basically, he handles the routing of the messages sent by one client to all the other clients.

Client - The client here acts from the side of the user. He sends the messages to the server, and the server sends this message to all the other clients to simulate a simple multi-user chat system.

9.3 Algorithm

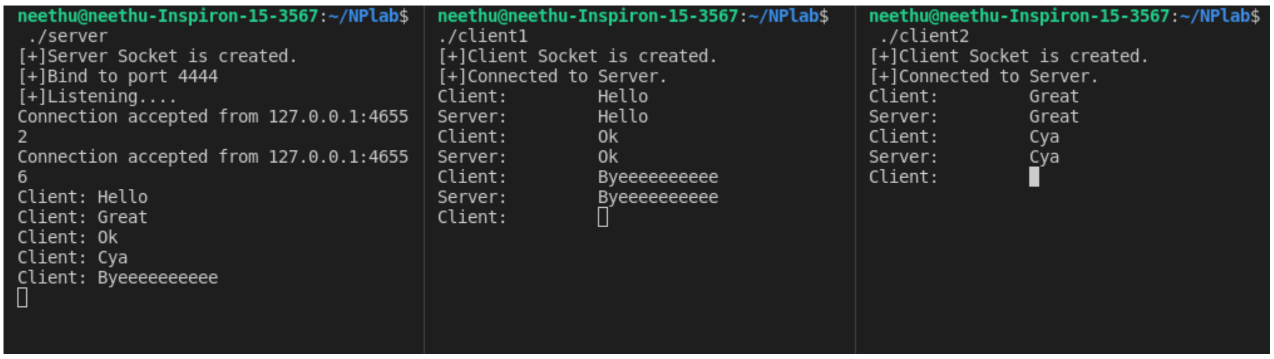
```
Algorithm for the Client

1 START
2 If argc < 2(no. of addresses passed), ask for address and exit
3 Create the socket using socket()
4 Configure sockaddrin struct
5 Connect the socket to server using function connect()
6 Send the message to the server using sendto()
7 Recieve the response from server using recfrom()
8 Print the response
9 STOP

Algorithm for the Server

1 START
2 Create the TCP socket
3 Configure sockaddrin struct
4 Bind the address struct to the socket using bind()
5 Accept incoming connections using accept()
6 Create child processes for these connections
7 Let these child processes handle each connected client
8 Until termination of the infinite loop, repeat steps 8-10
9 Receive the data using recvfrom()
10 Send response using sendto()
11 Print the received data
12 STOP
```

9.4 Program & Output

```
//Multi user chat server using TCP
```

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

#define PORT 4444

int main(){
    int clientSocket, ret;
    struct sockaddr_in serverAddr;
    char buffer[1024];

    clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket < 0){
        printf("[-]Error in connection.\n");
        exit(1);
    }
    printf("[+]Client Socket is created.\n");

    memset(&serverAddr, '\0', sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    ret = connect(clientSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr));
    if (ret < 0){
        printf("[-]Error in connection.\n");
        exit(1);
    }
    printf("[+]Connected to Server.\n");

    while (1){
        printf("Client: \t");
        scanf("%s", &buffer[0]);
        send(clientSocket, buffer, strlen(buffer), 0);

        if (strcmp(buffer, ":exit") == 0){
            close(clientSocket);
            printf("[-]Disconnected from server.\n");
            exit(1);
        }
        if (recv(clientSocket, buffer, 1024, 0) < 0){
            printf("[-]Error in receiving data.\n");
        }
        else{
            printf("Server: \t%s\n", buffer);
        }
    }
    return 0;
}

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

#define PORT 4444

int main(){
   int sockfd, ret;
   struct sockaddr_in serverAddr;

   int newSocket;
   struct sockaddr_in newAddr;

   socklen_t addr_size;

   char buffer[1024];
   pid_t childpid;

   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0){
      printf("[-]Error in connection.\n");
      exit(1);
   }
   printf("[+]Server Socket is created.\n");

   memset(&serverAddr, '\0', sizeof(serverAddr));
   serverAddr.sin_family = AF_INET;
   serverAddr.sin_port = htons(PORT);
   serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

   ret = bind(sockfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr));
   if (ret < 0){
      printf("[-]Error in binding1.\n");
      exit(1);
   }
   printf("[+]Bind to port %d\n", 4444);

   if (listen(sockfd, 10) == 0){
      printf("[+]Listening....\n");
   }
   else{
      printf("[-]Error in binding2.\n");
    }

    while (1){
        newSocket = accept(sockfd, (struct sockaddr *)&newAddr, &addr_size);
        if (newSocket < 0){
            exit(1);
        }
        printf("Connection accepted from %s:%d\n", inet_ntoa(newAddr.sin_addr),
        ntohs(newAddr.sin_port));

        if ((childpid = fork()) == 0){
            close(sockfd);

            while (1){
                recv(newSocket, buffer, 1024, 0);
                if (strcmp(buffer, ":exit") == 0){
                    printf("Disconnected from %s:%d\n", inet_ntoa(newAddr.sin_addr),
                    ntohs(newAddr.sin_port));
                    break;
                }
                else{
                    printf("Client: %s\n", buffer);
                    send(newSocket, buffer, strlen(buffer), 0);
                    bzero(buffer, sizeof(buffer));
                }
            }
        }
    }
    close(newSocket);
    return 0;
}
```

9.5 Output



```
neethu@neethu-Inspiron-15-3567:~/NPlab$
 ./server
[+]Server Socket is created.
[+]Bind to port 4444
[+]Listening....
Connection accepted from 127.0.0.1:4655
2
Connection accepted from 127.0.0.1:4655
6
Client: Hello
Client: Great
Client: Ok
Client: Cya
Client: Byeeeeeeeeee
```

```
neethu@neethu-Inspiron-15-3567:~/NPlab$
./client1
[+]Client Socket is created.
[+]Connected to Server.
Client:         Hello
Server:         Hello
Client:         Ok
Server:         Ok
Client:         Byeeeeeeeeee
Server:         Byeeeeeeeeee
Client:
```

```
neethu@neethu-Inspiron-15-3567:~/NPlab$
 ./client2
[+]Client Socket is created.
[+]Connected to Server.
Client:         Great
Server:         Great
Client:         Cya
Server:         Cya
Client:
```

9.6 Result

Implemented the program for a multi user chat server using TCP as transport layer protocol using C language in Ubuntu 20.04 with kernel and the above outputs were obtained.

10 Concurrent Time Server application using UDP

10.1 Aim

To implement Concurrent Time Server application using UDP to execute the program at remote server. Client sends a time request to the server, server sends its system time back to the client. Client displays the result.

10.2 Theory

UDP (User Datagram Protocol) is primarily for establishing low-latency and loss-tolerating connections between applications on the internet. UDP sends messages, called datagrams, and is considered a best-effort mode of communications. It is considered a connectionless protocol because it doesn't require a virtual circuit to be established before any data transfer occurs.

Server - The server here waits for the client's time request. When a request is received, the present system time of the server is sent to the client.

Client - The client sends the server a time request. The response from the server is received and provided as the output.

10.3 Algorithm

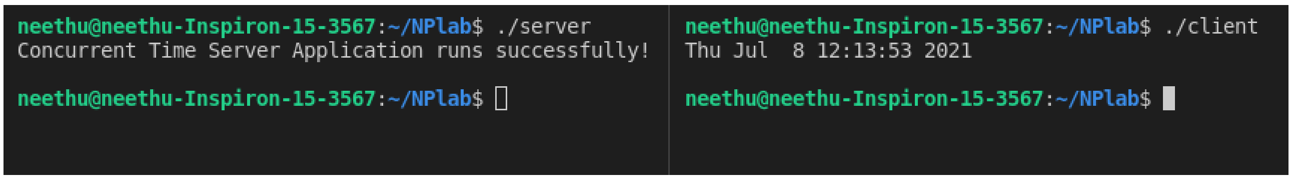
```
Algorithm for the Client

1 START
2 Create the socket using socket()
3 Configure socket details
4 Connect the socket to server using function connect()
5 Receive the response from server using recvfrom()
6 Send the acknowledge message to the server using sendto()
7 Print the response
8 STOP

Algorithm for the Server

1 START
2 We can use some time function to return the time
3 Create the UDP socket
4 Configure socket details
5 Bind the address struct to the socket using bind()
6 Receive from client using recvfrom()
7 Send the time to the client
8 STOP
```

10.4 Program & Output

```
//Concurrent time server application using UDP
//Client side
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

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

#define PORT 4444

int main(){
    char buffer[100];
    char *message = "Concurrent Time Server Application runs successfully!\n";

    int clientSocket, n;
    struct sockaddr_in servaddr;

    bzero(&servaddr, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    clientSocket = socket(AF_INET, SOCK_DGRAM, 0);

    if (connect(clientSocket, (struct sockaddr *)&servaddr,
        sizeof(servaddr)) == -1){
        printf("Error in connection.\n");
        exit(0);
    }

    sendto(clientSocket, message, 1000, 0, (struct sockaddr *)&servaddr,
        sizeof(servaddr));

    recvfrom(clientSocket, buffer, sizeof(buffer), 0, (struct sockaddr *)NULL,
        NULL);

    puts(buffer);

    close(clientSocket);
}

//Server side
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <time.h>

#define PORT 4444

int main()
{
    time_t t;
    time(&t);
    char buffer[100];
    char *message = ctime(&t);

    int serverSocket, len;
    struct sockaddr_in servaddr, cliaddr;

    bzero(&servaddr, sizeof(servaddr));
    serverSocket = socket(AF_INET, SOCK_DGRAM, 0);

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

    bind(serverSocket, (struct sockaddr *)&servaddr, sizeof(servaddr));

    len = sizeof(cliaddr);
    int n = recvfrom(serverSocket, buffer, sizeof(buffer), 0,
        (struct sockaddr *)&cliaddr, &len);

    buffer[n] = '\0';
    puts(buffer);

    sendto(serverSocket, message, 1000, 0, (struct sockaddr *)&cliaddr, len);
}
```

10.5 Output



```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./server
Concurrent Time Server Application runs successfully!

neethu@neethu-Inspiron-15-3567:~/NPlab$
```

```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./client
Thu Jul  8 12:13:53 2021

neethu@neethu-Inspiron-15-3567:~/NPlab$
```

10.6 Result

Implemented the program for the concurrent time server application using UDP using C language in Ubuntu 20.04 with kernel and the above outputs were obtained.

11 Simple Mail Transfer Protocol

11.1 Aim

To implement simulation of SMTP(Simple Mail transfer Protocol) using UDP.

11.2 Theory

SMTP is part of the application layer of the TCP/IP protocol. Using a process called "store and forward," SMTP moves your email on and across networks. SMTP makes use of a set of commands to transfer information via the client and server. Some of the important ones include:

- HELO: The HELO command is used to initiate an SMTP session.
- MAIL FROM: command is used primarily to send email addresses, it needs a way to alert the recipient host to who is sending the inbound message.
- RCPT TO: command tells the receiving host the email address of the message recipient
- DATA: When the sending host transmits the DATA command, it tells the receiving host that a stream of data will follow. End the data stream with ¡CRLF ¿.¡CRLF ¿
- QUIT: QUIT command is used to terminate an SMTP session.

Based on similar information regarding SMTP, the basic intent here is to simulate how SMTP works to move your mail on and across networks.

11.3 Algorithm

```
Algorithm for the Server

1 START
2 Define the registered users
3 Define the domain
4 Create the socket using socket()
5 Configure sockaddr_in, the structure describing an Internet Socket Address
6 Bind the address struct to the socket using bind()
7 Listen to the socket
8 A new socket for the incoming connection is created using accept()
9 Send the message with status 220 to denote that it is ready
10 If client's reply HELO command, server responds with status '250 ok' message.
11 If the response contain a 'from mail' that is registered in the server,
    status 250 is returned.
12 If the response contain a 'to mail' that is registered in the server, status
    250 is returned.
13 If the reply contains 'DATA' command, server responds with 354 to start
    mail input
14 The received body is written onto a file with from mail
15 If QUIT is received, '221 service closing' is returned to client
16 Close the socket
17 STOP

Algorithm for the Client

1 START
```

```
2 Get the from and to mail addresses
3 Create the socket using socket()
4 Configure sockaddr_in, the structure describing an Internet Socket Address
5 Connect to the server using connect()
6 If 220 status is returned from the server, connection is established
    successfully
7 Send 'HELO' command to the server
8 If the response contains 250, HELO successful
9 Send the mail to the server (MAIL FROM)
10 If the response is 250, the user is registered
11 Send the mail to the server(RCPT TP). If the response is 250, the user
    is registered
12 Send DATA command to the server. If status received is 354, goto step 13
13 Send the body of the mail to the server
14 To quit the connection, sent the command QUIT
15 If response is 221, connection is closed with the server
16 Close the socket
17 STOP
```

11.4 Program & Output

```
//Simple Mail Transfer Protocol using UDP
//Server side

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

#define MAXLINE 100
main(int argc, char *argv[])
{
    int n, sock_fd;
    struct sockaddr_in servaddr, cliaddr;
    char mesg[MAXLINE + 1];
    socklen_t len;
    char *str_ptr, *buf_ptr, *str;
    len = sizeof(cliaddr);
    if ((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        printf("SOCKET NOT CREATED\n");
        exit(1);
    }
    bzero((char *)&servaddr, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(atoi(argv[1]));
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
if (bind(sock_fd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0)
{
    perror("BIND FAILED");
    exit(1);
}
if((n=recvfrom(sock_fd,mesg,MAXLINE,0,(struct sockaddr*)&cliaddr,&len))== -1)
{
    perror("size not received:");
    exit(1);
}
mesg[n] = '\0';
printf("%s\n", mesg);
sprintf(mesg, "220 from_server_mail_server\n");

sendto(sock_fd, mesg, MAXLINE, 0,(struct sockaddr*)&cliaddr, sizeof(cliaddr));
n = recvfrom(sock_fd, mesg, MAXLINE, 0, (struct sockaddr *)&cliaddr, &len);

mesg[n] = '\0';
printf("C:%s\n", mesg);
str_ptr = strdup(mesg);
buf_ptr = strsep(&str_ptr, " ");
sprintf(mesg, "250 HELO %s", str_ptr);
free(buf_ptr);

sendto(sock_fd, mesg, MAXLINE, 0,(struct sockaddr*)&cliaddr, sizeof(cliaddr));
n = recvfrom(sock_fd, mesg, MAXLINE, 0, (struct sockaddr *)&cliaddr, &len);

mesg[n] = '\0';
printf("C: %s", mesg);
str_ptr = strdup(mesg);
buf_ptr = strsep(&str_ptr, ":");
str_ptr[strlen(str_ptr) - 1] = '\0';
sprintf(mesg, "250 HELO %s.......Sender OK\n", str_ptr);
free(buf_ptr);

sendto(sock_fd, mesg, MAXLINE, 0,(struct sockaddr*)&cliaddr, sizeof(cliaddr));
n = recvfrom(sock_fd, mesg, MAXLINE, 0, (struct sockaddr *)&cliaddr, &len);

mesg[n] = '\0';
printf("C: %s", mesg);
str_ptr = strdup(mesg);
buf_ptr = strsep(&str_ptr, ":");
str_ptr[strlen(str_ptr) - 1] = '\0';
sprintf(mesg, "250 HELO %s.......Recepient OK\n", str_ptr);
free(buf_ptr);

sendto(sock_fd, mesg, MAXLINE, 0,(struct sockaddr*)&cliaddr, sizeof(cliaddr));
n = recvfrom(sock_fd, mesg, MAXLINE, 0, (struct sockaddr *)&cliaddr, &len);
mesg[n] = '\0';
printf("C: %s\n", mesg);
sprintf(mesg, "354 Enter body of mail. End with \".\" on a newline\n");
```

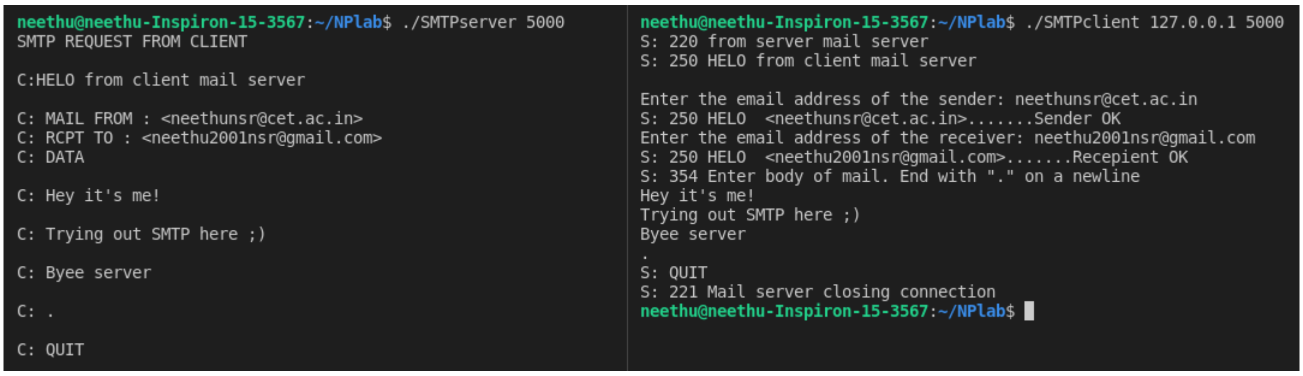
```
    sendto(sock_fd, mesg, MAXLINE, 0,(struct sockaddr*)&cliaddr, sizeof(cliaddr));
    while (1)
    {
        n = recvfrom(sock_fd, mesg, MAXLINE, 0, (struct sockaddr *)&cliaddr, &len);
        mesg[n] = '\0';
        printf("C: %s\n", mesg);
        mesg[strlen(mesg) - 1] = '\0';

        str = mesg;
        while (isspace(*str++))
            ;
        if (strcmp(--str, ".") == 0)
            break;
        sprintf(mesg, "250 Messages accepted for delivery \n");
        sendto(sock_fd, mesg, MAXLINE, 0, (struct sockaddr *)&cliaddr,
            sizeof(cliaddr));
        n = recvfrom(sock_fd, mesg, MAXLINE, 0, (struct sockaddr*)&cliaddr, &len);
        mesg[n] = '\0';
        printf("C: %s\n", mesg);
        sprintf(mesg, "221 Mail server closing connection\n");
        sendto(sock_fd, mesg, MAXLINE, 0, (struct sockaddr *)&cliaddr,
            sizeof(cliaddr));
    }
}

//Client side

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

#define MAXLINE 100
main(int argc, char *argv[])
{
    int n;
    int sock_fd;
    int i = 0;
    struct sockaddr_in servaddr;
    char buf[MAXLINE + 1];

    char address_buf[MAXLINE], message_buf[MAXLINE];
    char *str_ptr, *buf_ptr, *str;
    if (argc != 3)
    {
        fprintf(stderr, "Command is :./SMTPclient <address_port>\n");
        exit(1);
    }
```

```
if ((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
{
    printf("SOCKET NOT CREATED\n");
    exit(1);
}
bzero((char *)&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(atoi(argv[2]));
inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
sprintf(buf, "SMTP REQUEST FROM CLIENT\n");

n = sendto(sock_fd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr,
    sizeof(servaddr));
if (n < 0)
{
    perror("ERROR");
    exit(1);
}

if ((n = recvfrom(sock_fd, buf, MAXLINE, 0, NULL, NULL)) == -1)
{
    perror("UDP READ ERROR");
    exit(1);
}
buf[n] = '\0';
printf("S: %s", buf);
sprintf(buf, "HELO from_client_mail_server\n");

n = sendto(sock_fd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr,
    sizeof(servaddr));

if ((n = recvfrom(sock_fd, buf, MAXLINE, 0, NULL, NULL)) == -1)
{
    perror("UDP READ ERROR");
    exit(1);
}
buf[n] = '\0';
printf("S: %s", buf);
printf("\nEnter the email address of the sender: ");
fgets(address_buf, sizeof(address_buf), stdin);
address_buf[strlen(address_buf) - 1] = '\0';

sprintf(buf, "MAIL FROM : <%s>\n", address_buf);

sendto(sock_fd, buf, sizeof(buf), 0, (struct sockaddr *)&servaddr,
    sizeof(servaddr));
if ((n = recvfrom(sock_fd, buf, MAXLINE, 0, NULL, NULL)) == -1)
{
    perror("UDP READ ERROR");
    exit(1);
}
buf[n] = '\0';
```

```
printf("S: %s", buf);
printf("Enter the email address of the receiver: ");
fgets(address_buf, sizeof(address_buf), stdin);
address_buf[strlen(address_buf) - 1] = '\0';
sprintf(buf, "RCPT TO : <%s>\n", address_buf);
sendto(sock_fd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr,
    sizeof(servaddr));
if ((n = recvfrom(sock_fd, buf, MAXLINE, 0, NULL, NULL)) == -1)
{
    perror("UDP read error");
    exit(1);
}
buf[n] = '\0';
printf("S: %s", buf);
sprintf(buf, "DATA\n");
sendto(sock_fd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr,
    sizeof(servaddr));
if ((n = recvfrom(sock_fd, buf, MAXLINE, 0, NULL, NULL)) == -1)
{
    perror("UDP READ ERROR");
    exit(1);
}
buf[n] = '\0';
printf("S: %s", buf);
do
{
    fgets(message_buf, sizeof(message_buf), stdin);
    sprintf(buf, "%s", message_buf);
    sendto(sock_fd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr,
        sizeof(servaddr));
    message_buf[strlen(message_buf) - 1] = '\0';
    str = message_buf;
    while (isspace(*str++))
        ;
    if (strcmp(--str, ".") == 0)
        break;
} while (1);

if ((n = recvfrom(sock_fd, buf, MAXLINE, 0, NULL, NULL)) == -1)
{
    perror("UDP READ ERROR");
    exit(1);
}

buf[n] = '\0';
sprintf(buf, "QUIT\n");
printf("S: %s", buf);
sendto(sock_fd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr,
    sizeof(servaddr));
if ((n = recvfrom(sock_fd, buf, MAXLINE, 0, NULL, NULL)) == -1)
{
    perror("UDP READ ERROR");
```

```
        exit(1);
    }

    buf[n] = '\0';
    printf("S: %s", buf);
}
```

11.5 Output



```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./SMTPserver 5000
SMTP REQUEST FROM CLIENT

C:HELO from client mail server

C: MAIL FROM : <neethunsr@cet.ac.in>
C: RCPT TO : <neethu2001nsr@gmail.com>
C: DATA

C: Hey it's me!

C: Trying out SMTP here ;)

C: Byee server

C: .

C: QUIT
```

```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./SMTPclient 127.0.0.1 5000
S: 220 from server mail server
S: 250 HELO from client mail server

Enter the email address of the sender: neethunsr@cet.ac.in
S: 250 HELO  <neethunsr@cet.ac.in>.......Sender OK
Enter the email address of the receiver: neethu2001nsr@gmail.com
S: 250 HELO  <neethu2001nsr@gmail.com>.......Recepient OK
S: 354 Enter body of mail. End with "." on a newline
Hey it's me!
Trying out SMTP here ;)
Byee server
.
S: QUIT
S: 221 Mail server closing connection
neethu@neethu-Inspiron-15-3567:~/NPlab$
```

11.6 Result

Implemented the program for the simulation of Simple Mail Transfer Protocol using UDP, using C language in Ubuntu 20.04 with kernel and the above outputs were obtained.

12 Concurrent File Server

12.1 Aim

To develop concurrent file server which will provide the file requested by client if it exists. If not server sends appropriate message to the client. Server should also send its process ID (PID) to clients for display along with file or the message.

12.2 Theory

Server - A server is a software that waits for client requests and serves or processes them accordingly.

Client - A client is requester of this service. A client program request for some resources to the server and server responds to that request.

Socket - It is the endpoint of a bidirectional communication channel between a server and a client. Sockets may communicate within a process, between processes on the same machine, or between processes on different machines. For any communication with a remote program, we have to connect through a socket port.

FTP (File Transfer Protocol) is a client-server protocol that may be used to transfer files between computers on the internet. The client asks for the files and the server provides them. It is a standard network protocol used for the transfer of computer files between a client and server on a computer network. FTP is built on a client-server model architecture and uses separate control and data connections between the client and the server.FTP users may authenticate themselves with a clear-text sign-in protocol, normally in the form of a username and password, but can connect anonymously if the server is configured to allow it.

12.3 Algorithm

```
Algorithm for the Client

1 START
2 Create the socket using the function socket()
3 Configure socket details
4 Connect the socket to server using function connect()
5 Send the name of the file to searchfor to the server
6 If found, receive the file
7 If not found, console out the appropriate message
8 Receive the server’s PID
9 Terminate connection
10 STOP
```

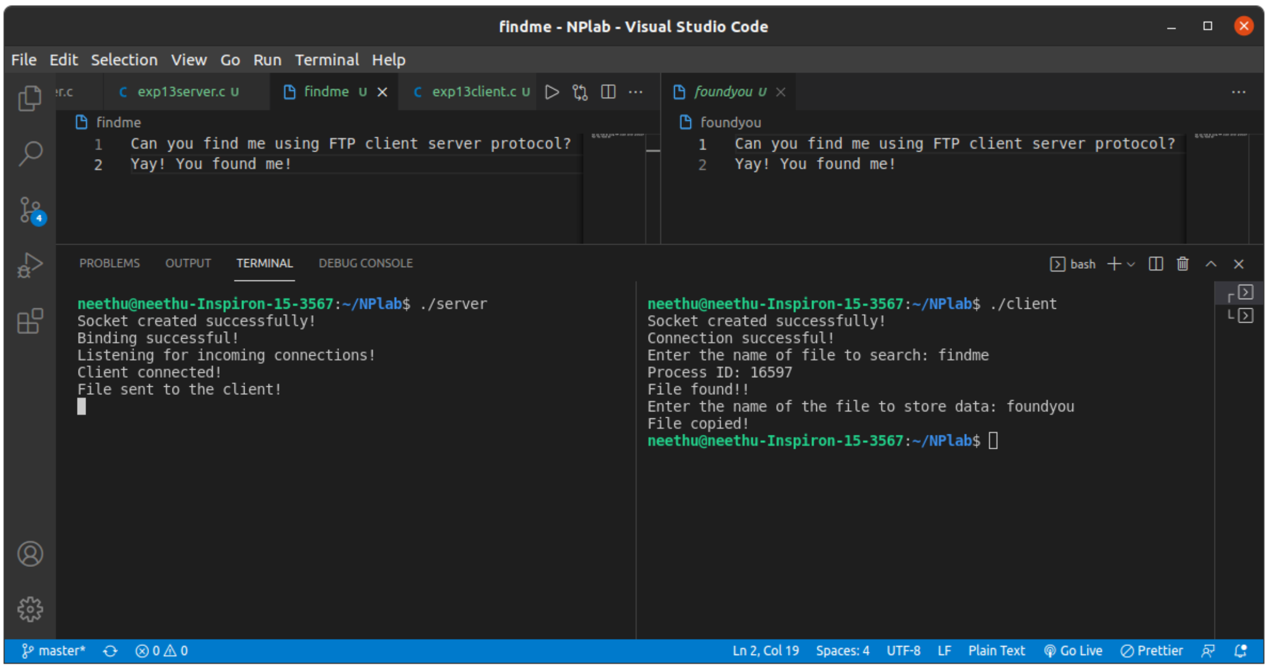
```
Algorithm for the Server

1 START
2 Configure socket details
3 Bind the address struct to the socket using bind()
4 Listen to the socket
5 A new socket for the incoming connection is created using accept()
```

```
6 Reciee the name of the file to search for
7 Check for the file using access()
8 If the file is found, goto step 9, else goto step 10
9 Send the file to the client using sendfile()
10 Send the response "NO" to the client
11 Send the PID to the client
12 STOP
```

12.4 Program & Output

```
//Concurrent File Server
//Client side

#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/stat.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/sendfile.h>

#define PORT 8080

int main()
{
    int socketServer;
    struct sockaddr_in server;
    char requestMessage[BUFSIZ], replyMessage[BUFSIZ];

    int filesize, file;
    char *data, filename[20], storageFile[20];

    socketServer = socket(AF_INET, SOCK_STREAM, 0);
    if (socketServer == -1)
    {
        perror("SOCKET NOT CREATED");
        return 1;
    }
    printf("Socket created successfully!\n");

    server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);

    if (connect(socketServer, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        perror("Connection failed...");
        return 1;
    }
    printf("Connection successful!\n");
    printf("Enter the name of file to search: ");
    scanf("%s", filename);

    write(socketServer, filename, strlen(filename));
    recv(socketServer, replyMessage, 10, 0);
    int pid = atoi(replyMessage);
    printf("Process ID: %d\n", pid);
    if (pid > 0)
    {
        printf("File found!!\n");
        printf("Enter the name of the file to store data: ");
        scanf("%s", storageFile);

        recv(socketServer, &filesize, sizeof(int), 0);
        data = malloc(filesize);
        file = open(storageFile, O_CREAT | O_EXCL | O_WRONLY, 0666);
        recv(socketServer, data, filesize, 0);
        write(file, data, filesize);

        printf("File copied!\n");
        close(file);
    }
    else if (strcmp(replyMessage, "NO") == 0)
    {
        fprintf(stderr, "File not found!\n");
    }
    return 0;
}

//Server side

#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include <sys/stat.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/sendfile.h>

#define PORT 8080

void Sender(int socket, char *filename)
{
    struct stat obj;
    int file, filesize;
    stat(filename, &obj);
    file = open(filename, O_RDONLY);
    filesize = obj.st_size;
    send(socket, &filesize, sizeof(int), 0);
    sendfile(socket, file, NULL, filesize);
    printf("File sent to the client!\n");
}

void *callBack(void *socket)
{
    int socketfd = *(int *)socket;
    char serverResponse[BUFSIZ], filename[BUFSIZ];
    recv(socketfd, filename, BUFSIZ, 0);
    //If the file exists, notify the client and send it
    if (access(filename, F_OK) != -1)
    {
        snprintf(serverResponse, 10, "%d", getpid());
        write(socketfd, serverResponse, 10);
        Sender(socketfd, filename);
    }
    else
    {
        strcpy(serverResponse, "NO");
        write(socketfd, serverResponse, strlen(serverResponse));
    }
    free(socket);
}

int main(int argc, char **argv)
{
    int socketServer, socketClient, *newSocket, len = sizeof(struct sockaddr_in);
    struct sockaddr_in server, client;

    socketServer = socket(AF_INET, SOCK_STREAM, 0);
    if (socketServer == -1)
    {
        printf("SOCKET NOT CREATED");
        exit(0);
    }
    printf("Socket created successfully!\n");

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

    if (bind(socketServer, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        printf("BIND FAILED");
        exit(0);
    }
    printf("Binding successful!\n");
    listen(socketServer, 3);
    printf("Listening for incoming connections!\n");
    while (socketClient = accept(socketServer, (struct sockaddr *)&client,
        (socklen_t *)&len))
    {
        printf("Client connected!\n");
        pthread_t sniffer;
        newSocket = malloc(1);
        *newSocket = socketClient;
        pthread_create(&sniffer, NULL, callBack, (void *)newSocket);
        pthread_join(sniffer, NULL);
    }
    if (socketClient < 0)
    {
        printf("Accept failed!\n");
        return 1;
    }
    return 0;
}
```

12.5 Output


findme - NPlab - Visual Studio Code
File Edit Selection View Go Run Terminal Help
exp13server.c U
findme U
exp13client.c U
foundyou U
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
bash
master* 0 0
Ln 2, Col 19 Spaces: 4 UTF-8 LF Plain Text Go Live Prettier

```
findme
Can you find me using FTP client server protocol?
Yay! You found me!
```

```
foundyou
Can you find me using FTP client server protocol?
Yay! You found me!
```

```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./server
Socket created successfully!
Binding successful!
Listening for incoming connections!
Client connected!
File sent to the client!
```

```
neethu@neethu-Inspiron-15-3567:~/NPlab$ ./client
Socket created successfully!
Connection successful!
Enter the name of file to search: findme
Process ID: 16597
File found!!
Enter the name of the file to store data: foundyou
File copied!
neethu@neethu-Inspiron-15-3567:~/NPlab$
```

12.6 Result

Implemented the program to develop concurrent file server which will provide the file requested by client if it exists, along with the pid, using C language in Ubuntu 20.04 with kernel and the above outputs were obtained.

13 Network with multiple subnets with wired and wireless LANs

13.1 Aim

Design and configure a network with multiple subnets with wired and wireless LANs using required network devices. Configure the following services in the network- TELNET, SSH, FTP server, Web server, File server, DHCP server and DNS server.

13.2 Theory

Subnet
A subnet is a logical partition of an IP network into multiple, smaller network segments. It is typically used to subdivide large networks into smaller, more efficient subnetworks. The internet is composed of many networks that are run by many organizations. In turn, each organization's network can be composed of many smaller networks, or subnets. Each subnet allows its connected devices to communicate with each other, and routers are used to communicate between subnets. The size of a subnet depends on the connectivity requirements and the network technology employed. A point-to-point subnet allows two devices to connect, while a data center subnet might be designed to connect many more devices.

13.3 Configuring the Services

The following shows how the different services can be configured in an Ubuntu PC:

Telnet

Telnet is a user command and an underlying TCP/IP protocol for accessing remote computers. Through Telnet, an administrator or another user can access someone else's computer remotely. On the Web, HTTP and FTP protocols allow you to request specific files from remote computers, but not to actually be logged on as a user of that computer. With Telnet, you log on as a regular user with whatever privileges you may have been granted to the specific application and data on that computer. Telnet is most likely to be used by program developers and anyone who has a need to use specific applications or data located at a particular host computer. Take the following steps to configure Telnet:

- Install Telnet

```
sudo apt install telnet xinetd
```

- Edit /etc/inetd.conf with root permission,add this line:

```
telnet stream tcp nowait telnetd /usr/sbin/tcpd /usr/sbin/in.telnet
```

- Edit /etc/xinetd.conf, copy the following configuration:

```
# Simple  configuration file for xinetd
#
# Some defaults , and include /etc/xinetd.d
defaults
{
# Please note the you need a log_type line to be able to use
# log_on_success and log_on_failure . The default is the following
# log_type = SYSLOG daemon info
instances = 60
log_type = SYSLOG authpriv
log_on_success = HOST PID
log_on_failure = HOST
cps = 25 30
}
```

- Change telnet port by using the following command in the terminal:

  ```
  tellnet 23/tcp
  ```

- Then restart the service

  ```
  sudo /etc/init.d/xinetd restart
  ```

SSH

The SSH protocol (also referred to as Secure Shell) is a method for secure remote login from one computer to another. It provides several alternative options for strong authentication, and it protects the communications security and integrity with strong encryption. It is a secure alternative to the non-protected login protocols (such as telnet, rlogin) and insecure file transfer methods (such as FTP).
Take the following steps to configure SSH:

- Install SSH:

  ```
  sudo apt-get installl openssh-server
  ```

  (Installing the client can be done by replacing openssh-server by openssh-client)

- Configure SSH:

  ```
  sudo nano /etc/ssh/sshd_config
  ```

  Then make the changes you want to make

- Restart SSH:

  ```
  sudo systemctl restart ssh
  ```

We can login to the SSH server from an SSH client.

FTP server

File Transfer Protocol (FTP) is the commonly used protocol for exchanging files over the Internet. FTP uses the Internet's TCP/IP protocols to enable data transfer. FTP uses a client-server architecture, often secured with SSL/TLS. FTP promotes sharing of files via remote computers with reliable and efficient data transfer. FTP uses a client-server architecture. Users provide authentication using a sign-in protocol, usually a username and password, however some FTP servers may be configured to accept anonymous FTP logins where you don't need to identify yourself before accessing files. Most often, FTP is secured with SSL/TLS.
The following steps show setting up an FTP server on the computer:

- Install FTP daemon:

  ```
  sudo apt install vsftpd
  ```

- Configure FTP can be done by editing the following file:

  ```
  /etc/vsftpd.conf
  ```

- Restart the service

  ```
  sudo systemctl restart vsftpd.service
  ```

Web Server

A Web server is a program that uses HTTP (Hypertext Transfer Protocol) to serve the files that form Web pages to users, in response to their requests, which are forwarded by their computers' HTTP clients. Dedicated computers and appliances may be referred to as Web servers as well. The process is an example of the client/server model. All computers that host Web sites must have Web server programs. Leading Web servers include Apache (the most widely-installed Web server), Microsoft's Internet Information Server (IIS) and nginx (pronounced engine X) from NGNIX. Other Web servers include Novell's NetWare server, Google Web Server (GWS) and IBM's family of Domino servers.
A web server can be hosted on the localhost of the PC by following the following steps:

- Installing the server: The most common server on Linux systems and it is called the LAMP server. It can be installed on Ubuntu by:

  ```
  sudo apt install lamp-server
  ```

- Hosting a website: By creating a .conf file in the */etc/apache2/sites-available/* folder, we inform the server of the location of the code for our website.
- Enabling the website by using the command:

  ```
  sudo a2ensite <nameOfFile.conf>
  ```

- By editing */etc/hosts* file, we can give the domain name for the website
- The configuration the server is in the file: */etc/apache2/apache2.conf*
- Restart the server by the command:

  ```
  sudo systemctl restart apache2 service
  ```

File Server

In the client/server model, a file server is a computer responsible for the central storage and management of data files so that other computers on the same network can access the files. A file server allows users to share information over a network without having to physically transfer files by floppy diskette or some other external storage device. Any computer can be configured to be a host and act as a file server. In its simplest form, a file server may be an ordinary PC that handles requests for files and sends them over the network. In a more sophisticated network, a file server might be a dedicated network-attached storage (NAS) device that also serves as a remote hard disk drive for other computers, allowing anyone on the network to store files on it as if to their own hard drive. The following steps can be followed to setup a file server:

- Installing Samba File Server:

  ```
  sudo apt install samba
  ```

- Configuring the file server by editing */etc/samba/smb.conf* First, edit the following key/value pairs in the [global] section of /etc/samba/smb.conf:

  ```
  workgroup = EXAMPLE
  ...
  security = user
  Create a new section at the bottom of the file, or uncomment one
  of the examples, for the directory to be shared:
  [ share]
  comment = Ubuntu File Server Share
  path = /srv/samba/share
  browsable = yes
  guest ok = yes
  read only = no
  create mask = 0755
  ```

- Make a directory for hosting files and setting permission for the directory:

  ```
  sudo mkdir -p /srv/samba/share
  sudo chown nobody : nogroup /srv/samba/ share/
  ```

- Restart Samba service:

  ```
  sudo systemctl restart smbd. service nmbd. service
  ```

DHCP Server

DHCP (Dynamic Host Configuration Protocol) is a network management protocol used to dynamically assign an Internet Protocol (IP) address to any device, or node, on a network so they can communicate using IP. DHCP automates and centrally manages these configurations rather than requiring network administrators to manually assign IP addresses to all network devices. DHCP can be implemented on small local networks as well as large enterprise networks. DHCP will assign new IP addresses in each location when devices are moved from place to place, which means network administrators do not have to manually initially configure each device with a valid IP address or reconfigure the device with a new IP address if it moves to a new location on the network. Versions of DHCP are available for use in Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6).
The following steps shows how DHCP server can be run:

- Install DHCP server:

```
sudo apt-get install isc-dhcp-server
```

- Configure DHCP server, the config file is */etc/dhcp/dhcpd.conf:*

```
# Sample /etc/dhcpd.conf
# (add your comments here)
default |lease |time 600;
max|lease|time 7200;
option subnet|mask 255.255.255.0;
option broadcast|address 192.168.1.255;
option routers 192.168.1.254;
option domain|name|servers 192.168.1.1 , 192.168.1.2;
option domain|name "mydomain . example ";

subnet 192.168.1.0 netmask 255.255.255.0 {
range 192.168.1.10 192.168.1.100;
range 192.168.1.150 192.168.1.200;
}
```

- Starting and stopping services can be achieved using:

```
sudo service isc|dhcp|server restart
sudo service isc|dhcp|server start
sudo service isc|dhcp|server stop
```

After editing configuration files, we have to restart the service.

DNS Server

The Domain Name Systems (DNS) is the phonebook of the Internet. Humans access information online through domain names, like nytimes.com or espn.com. Web browsers interact through Internet Protocol (IP) addresses. DNS translates domain names to IP addresses so browsers can load Internet resources.

Each device connected to the Internet has a unique IP address which other machines use to find the device. DNS servers eliminate the need for humans to memorize IP addresses such as 192.168.1.1 (in IPv4), or more complex newer alphanumeric IP addresses such as 2400:cb00:2048:1::c629:d7a2 (in IPv6).
The following steps show the setup:

- Installing

```
sudo apt install bind9
```

- The configuration is in the /etc/bind folder
- Setting as a catching name server by editing the file */etc/bind/named.conf.options*:

```
forwarders {
1.2.3.4; # replace with ip address
5.6.7.8; # of the name servers
}
```

- BIND9 can be configured with the primary and the secondary master as a custom DNS server to access all the subnets.
- Restarting bind9:

```
sudo systemctl restart bind9.service
```

13.4 Result

Implemented the program for accessing the different nodes in the subnet, TELNET, SSH, FTP server, Web server, File server, DHCP server and DNS server have been configured in Ubuntu 20.04 with kernel and the above outputs were obtained.