Working with Networking Command Line Tools

Ex. No. 1

Date:03.03.2023

PROBLEM STATEMENT:

To allow users to perform various network-related tasks and diagnostics directly from the command line interface (CLI) of an operating system and to provide a way to interact with network devices, troubleshoot network issues, and gather information about network configurations.

PROBLEM DESCRIPTION:

In modern computer networks, administrators often need a robust and intuitive tool to manage network devices, perform configuration tasks, and troubleshoot network-related issues.

The network command line interface (CLI) tool aims to fulfill this requirement by providing a comprehensive set of commands and functionalities.

The primary objective of the network CLI tool is to simplify the administration of network devices, such as routers, switches, and firewalls, by offering a command-based interface that allows users to interact with these devices efficiently.

The CLI tool should enable network administrators to perform tasks like configuring network interfaces, setting up routing protocols, managing access control lists (ACLs), monitoring network traffic, and diagnosing connectivity problems.

PING COMMAND:

- The ping command is a network diagnostic tool used to test the reachability and latency (response time) of a remote host or IP address. It is available on most operating systems, including Windows, macOS, and Linux.
- When you execute the ping command followed by a target host or IP address, the command sends out small packets of data to the specified destination. The target host then responds to these packets, allowing you to measure the time it takes for the packets to reach the destination and return.

1. Basic Ping:

ping <host or IP>

This command sends ICMP echo requests to the specified host or IP address and displays the round-trip time (RTT) for each reply. It continues to send echo requests until interrupted.

```
:~$ ping google.com
PING google.com (142.250.183.238) 56(84) bytes of data.
64 bytes from maa05s23-in-f14.1e100.net (142.250.183.238): icmp_seq=1 ttl=115 time=17.3 ms
64 bytes from maa05s23-in-f14.1e100.net (142.250.183.238): icmp_seq=2 ttl=115 time=7.13 ms
64 bytes from maa05s23-in-f14.1e100.net (142.250.183.238): icmp_seq=3 ttl=115 time=63.4 ms
64 bytes from maa05s23-in-f14.1e100.net (142.250.183.238): icmp_seq=4 ttl=115 time=12.0 ms
S64 bytes from maa05s23-in-f14.1e100.net (142.250.183.238): icmp_seq=5 ttl=115 time=6.22 ms
64 bytes from maa05s23-in-f14.1e100.net (142.250.183.238): icmp_seq=6 ttl=115 time=31.3 ms
64 bytes from maa05s23-in-f14.1e100.net (142.250.183.238): icmp_seq=7 ttl=115 time=4.87 ms
64 bytes from maa05s23-in-f14.1e100.net (142.250.183.238): icmp_seq=8 ttl=115 time=7.70 ms
^C
--- google.com ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7023ms
rtt min/avg/max/mdev = 4.871/18.748/63.409/18.730 ms
```

2. Specify Number of Echo Requests:

ping -c <count> <host or IP>

```
:~$ ping -c 3 www.google.com
PING www.google.com (142.250.182.68) 56(84) bytes of data.
64 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=1 ttl=58 time=15.5 ms
64 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=2 ttl=58 time=17.9 ms
64 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=3 ttl=58 time=11.3 ms

--- www.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 11.328/14.926/17.926/2.726 ms
```

The -c option allows you to specify the number of ICMP echo requests to send. It sends the specified number of requests and stops afterward, displaying the statistics.

3. Set Ping Interval:

ping -i <interval> <host or IP>

```
:~$ ping -i 2 www.google.com
PING www.google.com (142.250.195.164) 56(84) bytes of data.
64 bytes from maa03s41-in-f4.1e100.net (142.250.195.164): icmp_seq=1 ttl=117 time=19.9 ms
64 bytes from maa03s41-in-f4.1e100.net (142.250.195.164): icmp_seq=2 ttl=117 time=13.4 ms
64 bytes from maa03s41-in-f4.1e100.net (142.250.195.164): icmp_seq=3 ttl=117 time=16.4 ms
^Z
[1]+  Stopped                 ping -i 2 www.google.com
```

The -i option sets the interval in seconds between each ICMP echo request. It controls the rate at which the requests are sent.

4. DNS Reverse Lookup:

ping -a <IP>

```
:~$ ping -a www.goooogle.com
PING www.goooogle.com (142.250.76.67) 56(84) bytes of data.
64 bytes from maa05s14-in-f3.1e100.net (142.250.76.67): icmp_seq=1 ttl=58 time=7.54 ms
64 bytes from maa05s14-in-f3.1e100.net (142.250.76.67): icmp_seq=2 ttl=58 time=18.4 ms
64 bytes from maa05s14-in-f3.1e100.net (142.250.76.67): icmp_seq=3 ttl=58 time=23.4 ms
64 bytes from maa05s14-in-f3.1e100.net (142.250.76.67): icmp_seq=4 ttl=58 time=15.9 ms
64 bytes from maa05s14-in-f3.1e100.net (142.250.76.67): icmp_seq=5 ttl=58 time=8.65 ms
64 bytes from maa05s14-in-f3.1e100.net (142.250.76.67): icmp_seq=6 ttl=58 time=17.2 ms
^Z
[4]+  Stopped                 ping -a www.goooogle.com
```

The -a option is used to perform a reverse DNS lookup of the specified IP address. It attempts to determine the corresponding hostname of the given IP.

5. Specify Packet Size:

ping -s <size> <host or IP>

```
:~$ ping -s 100 www.google.com
PING www.google.com (142.250.182.68) 100(128) bytes of data.
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=1 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=2 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=3 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=4 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=5 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=6 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=7 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=8 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=9 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=10 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=11 ttl=58 (truncated)
76 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=12 ttl=58 (truncated)
^Z
[2]+  Stopped                 ping -s 100 www.google.com
```

The -s option allows you to specify the size in bytes of the ICMP echo request packet. This can be used to test the maximum transmission unit (MTU) of the network **.**

6. To Set Time Limit:

ping -w <size> <hostname>

To stop receiving a ping output after a specific amount of time, add -

w and an interval in seconds to your command.

```
:~$ ping -w 5 www.google.com
PING www.google.com (142.250.182.68) 56(84) bytes of data.
64 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=1 ttl=58 time=19.8 ms
64 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=2 ttl=58 time=11.4 ms
64 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=3 ttl=58 time=19.0 ms
64 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=4 ttl=58 time=15.9 ms
64 bytes from maa05s20-in-f4.1e100.net (142.250.182.68): icmp_seq=5 ttl=58 time=17.4 ms

--- www.google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4069ms
rtt min/avg/max/mdev = 11.432/16.697/19.772/2.953 ms
```

TRACEROUTE:

- Traceroute is a network diagnostic tool used to track the path that packets take from your computer to a target host or IP address on a network. It provides information about each hop (router) along the path, showing the IP addresses and response times for each hop.
- To use the traceroute command in Linux, follow these steps:
- Open a terminal: Launch a terminal application on your Linux system. You can typically find it in the applications menu or by using the search function.
- Run the traceroute command: In the terminal, type the following command:

traceroute <host or IP>

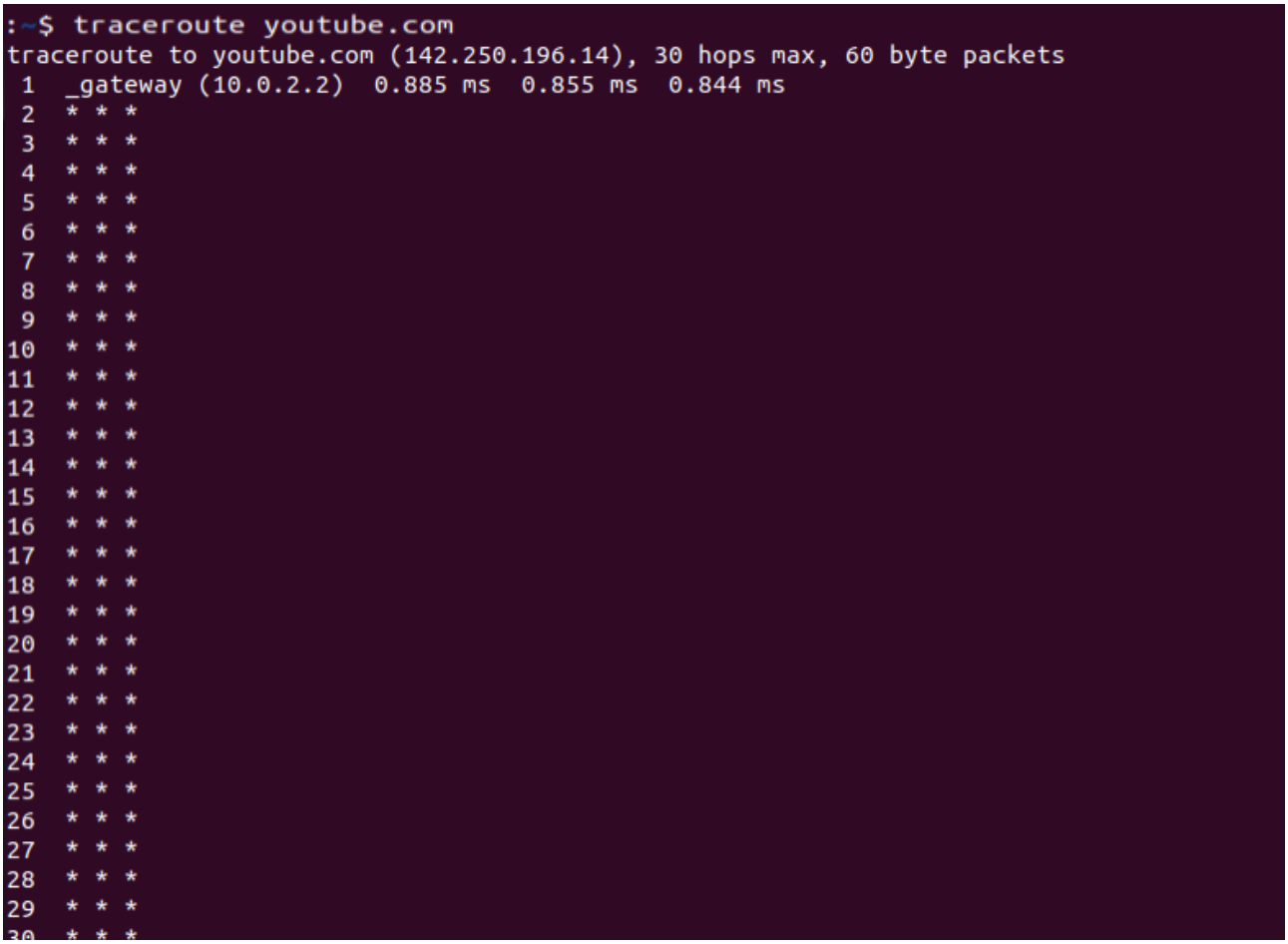
- Replace <host or IP> with the target host or IP address you want to trace the route to. For example

traceroute www.example.com

- View the output: Press Enter to execute the command. The traceroute command will start sending packets to the target and display the information about each hop in the route. The output will include the hop number, IP address, and round-trip time (RTT) for each hop.
- Analyze the output: Review the output to identify the route taken by the packets and examine the RTT values. Higher RTT values may indicate network congestion or latency at that particular hop.
- Stop the traceroute: The traceroute command will continue sending packets until it reaches the maximum hop limit (usually 30 hops) or until you interrupt it manually by pressing Ctrl+C in the terminal.
- Note: The traceroute command in Linux may require administrative (root) privileges to function properly. If you encounter any issues or receive "Permission denied" errors, try executing the command with sudo:

sudo traceroute <host or IP>

Using sudo will prompt you to enter your password before executing the command.



NETSTAT:

The netstat command allows you to monitor and analyze various aspects of network activity on your system. It provides details such as active network connections, listening ports, routing tables, and network interface statistics. By using netstat, you can gather valuable information about network performance, troubleshooting network issues, and identifying active connections.

Here are some common uses of the netstat command:

1. **Displaying Active Network Connections:**

netstat -a

This command shows all active network connections, including TCP, UDP, and Unix domain sockets.

```
$ netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:ipp           0.0.0.0:*               LISTEN
tcp        0      0 localhost:domain        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:673             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:sunrpc          0.0.0.0:*               LISTEN
tcp        0      0 10.7.5.27:46332         bom07s25-in-f14.1:https ESTABLISHED
tcp        0      0 10.7.5.27:41508         ec2-34-237-73-95.:https ESTABLISHED
tcp        0      0 10.7.5.27:49042         104.18.3.161:https      TIME_WAIT
tcp        0      0 10.7.5.27:41336         a23-201-200-135.d:https ESTABLISHED
tcp        0      0 10.7.5.27:35820         maa05s16-in-f5.1e:https ESTABLISHED
tcp        0      0 10.7.5.27:40466         55.65.117.34.bc.g:https ESTABLISHED
tcp        0      0 10.7.5.27:35812         maa05s16-in-f5.1e:https ESTABLISHED
tcp        0      0 10.7.5.27:43594         maa05s18-in-f5.1e:https ESTABLISHED
tcp6       0      0 ip6-localhost:ipp       [::]:*                  LISTEN
tcp6       0      0 [::]:674                [::]:*                  LISTEN
tcp6       0      0 [::]:sunrpc             [::]:*                  LISTEN
udp        0      0 0.0.0.0:mdns            0.0.0.0:*
udp        0      0 0.0.0.0:51090           0.0.0.0:*
udp        0      0 0.0.0.0:43983           0.0.0.0:*
udp        0      0 0.0.0.0:56422           0.0.0.0:*
udp        0      0 localhost:domain        0.0.0.0:*
udp        0      0 0.0.0.0:sunrpc          0.0.0.0:*
udp        0      0 0.0.0.0:631             0.0.0.0:*
udp        0      0 0.0.0.0:673             0.0.0.0:*
udp6       0      0 [::]:mdns               [::]:*
udp6       0      0 [::]:32802              [::]:*
udp6       0      0 [::]:sunrpc             [::]:*
udp6       0      0 [::]:674                [::]:*
raw6       0      0 [::]:ipv6-icmp          [::]:*                  7
Active UNIX domain sockets (servers and established)
```

2. Displaying Listening Ports:

netstat -l

This command lists all the ports on which the system is listening for incoming connections.

```
$ netstat -l
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:ipp           0.0.0.0:*               LISTEN
tcp        0      0 localhost:domain        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:673             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:sunrpc          0.0.0.0:*               LISTEN
tcp6       0      0 ip6-localhost:ipp       [::]:*                  LISTEN
tcp6       0      0 [::]:674                [::]:*                  LISTEN
tcp6       0      0 [::]:sunrpc             [::]:*                  LISTEN
udp        0      0 0.0.0.0:mdns            0.0.0.0:*
udp        0      0 0.0.0.0:51090           0.0.0.0:*
udp        0      0 0.0.0.0:43983           0.0.0.0:*
udp        0      0 localhost:domain        0.0.0.0:*
udp        0      0 0.0.0.0:sunrpc          0.0.0.0:*
udp        0      0 0.0.0.0:631             0.0.0.0:*
udp        0      0 0.0.0.0:673             0.0.0.0:*
udp6       0      0 [::]:mdns               [::]:*
udp6       0      0 [::]:32802              [::]:*
udp6       0      0 [::]:sunrpc             [::]:*
udp6       0      0 [::]:674                [::]:*
raw6       0      0 [::]:ipv6-icmp          [::]:*                  7
Active UNIX domain sockets (only servers)
```

3. Displaying Network Statistics

netstat -s

This command provides comprehensive statistics for various network protocols, such as TCP, UDP, ICMP, and IP.

```
$ netstat -s
Ip:
    Forwarding: 2
    147362 total packets received
    30 with invalid addresses
    0 forwarded
    0 incoming packets discarded
    145093 incoming packets delivered
    72353 requests sent out
    20 outgoing packets dropped
Icmp:
    217 ICMP messages received
    6 input ICMP message failed
    ICMP input histogram:
        destination unreachable: 128
        echo replies: 89
    450 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 272
        echo requests: 178
IcmpMsg:
        InType0: 89
        InType3: 128
        OutType3: 272
        OutType8: 178
Tcp:
    552 active connection openings
    60 passive connection openings
    8 failed connection attempts
    10 connection resets received
    7 connections established
    109541 segments received
    50710 segments sent out
    33 segments retransmitted
    0 bad segments received
    142 resets sent
Udp:
    33349 packets received
    211 packets to unknown port received
    36 packet receive errors
    21586 packets sent
    0 receive buffer errors
    0 send buffer errors
    InCsumErrors: 36
    IgnoredMulti: 1741
```

4. Displaying Routing Table

netstat -r

This command shows the system's routing table, including destination networks, gateways, and interface information.

```
$ netstat -r
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
default         _gateway        0.0.0.0         UG        0 0          0 enp3s0
10.7.0.0        0.0.0.0         255.255.0.0     U         0 0          0 enp3s0
link-local      0.0.0.0         255.255.0.0     U         0 0          0 enp3s0
$ netstat -p
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 10.7.5.27:46332         bom07s25-in-f14.1:https ESTABLISHED 2902/firefox
tcp        0      0 10.7.5.27:41508         ec2-34-237-73-95.:https ESTABLISHED 2902/firefox
tcp        0      0 10.7.5.27:41336         a23-201-200-135.d:https ESTABLISHED 2902/firefox
tcp        0      0 10.7.5.27:35820         maa05s16-in-f5.1e:https ESTABLISHED 2902/firefox
tcp        0      0 10.7.5.27:40466         55.65.117.34.bc.g:https ESTABLISHED 2902/firefox
tcp        0      0 10.7.5.27:35812         maa05s16-in-f5.1e:https ESTABLISHED 2902/firefox
tcp        0      0 10.7.5.27:43594         maa05s18-in-f5.1e:https ESTABLISHED 2902/firefox
Active UNIX domain sockets (w/o servers)
```

To use the netstat command, follow these steps:

- Open a terminal or command prompt: Launch the terminal application or command prompt on your operating system.
- Execute the netstat command: In the terminal or command prompt, type the netstat command followed by the desired options and parameters.

For example:

netstat -a

This command will display all active network connections.

```
:~$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 harish:50242            maa05s24-in-f14.1:https TIME_WAIT
tcp        0      0 harish:45068            server-65-9-25-99:https ESTABLISHED
tcp        0      0 harish:38942            55.65.117.34.bc.g:https ESTABLISHED
tcp        0      0 harish:52560            233.90.160.34.bc.:https ESTABLISHED
tcp        0      0 harish:51164            53.121.117.34.bc.:https TIME_WAIT
tcp        0      0 harish:36342            maa05s25-in-f3.1e:https TIME_WAIT
tcp        0      0 harish:59270            maa03s47-in-f4.1e:https ESTABLISHED
tcp        0      0 harish:57530            maa05s24-in-f14.1:https ESTABLISHED
tcp        0      0 harish:46300            191.144.160.34.bc:https TIME_WAIT
tcp        0      0 harish:44978            aerodent.canonical:http TIME_WAIT
tcp        0      0 harish:42420            209.100.149.34.bc:https TIME_WAIT
tcp        0      0 harish:37258            239.237.117.34.bc:https ESTABLISHED
tcp        0      0 harish:33356            maa03s41-in-f5.1e:https ESTABLISHED
tcp        0      0 harish:40848            201.181.244.35.bc:https ESTABLISHED
udp        0      0 harish:bootpc           _gateway:bootps         ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   Path
unix  2      [ ]         DGRAM      CONNECTED     22893    /run/user/1000/systemd/notify
unix  3      [ ]         DGRAM      CONNECTED     14133    /run/systemd/notify
unix  2      [ ]         DGRAM                    14147    /run/systemd/journal/syslog
unix  18     [ ]         DGRAM      CONNECTED     14156    /run/systemd/journal/dev-log
unix  8      [ ]         DGRAM      CONNECTED     14158    /run/systemd/journal/socket
unix  3      [ ]         STREAM     CONNECTED     25100    /run/user/1000/at-spi/bus
unix  3      [ ]         STREAM     CONNECTED     25632    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     25837    /run/user/1000/bus
unix  3      [ ]         STREAM     CONNECTED     22527    /run/user/1000/pulse/native
unix  3      [ ]         STREAM     CONNECTED     23833
unix  3      [ ]         STREAM     CONNECTED     23258
unix  3      [ ]         STREAM     CONNECTED     18190    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     65684
unix  2      [ ]         DGRAM                    45048
unix  3      [ ]         STREAM     CONNECTED     24937
unix  2      [ ]         DGRAM                    23389
unix  3      [ ]         STREAM     CONNECTED     19156    /run/dbus/system_bus_socket
unix  2      [ ]         STREAM     CONNECTED     67139
unix  3      [ ]         STREAM     CONNECTED     25115
unix  3      [ ]         STREAM     CONNECTED     22198    /run/user/1000/bus
unix  3      [ ]         STREAM     CONNECTED     75434
unix  3      [ ]         SEQPACKET  CONNECTED     66444
unix  3      [ ]         STREAM     CONNECTED     20443
unix  3      [ ]         SEQPACKET  CONNECTED     68382
```

DIG:

- The dig command, short for "Domain Information Groper," is a command-line network administration tool used to perform DNS (Domain Name System) queries.
- It is commonly available on Unix-like operating systems, including Linux and macOS.
- The dig command allows you to retrieve various types of DNS information from DNS servers.
- It provides a flexible and powerful way to query DNS records, troubleshoot DNS-related issues, and gather DNS-related information.

```
:~$ dig google.com

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 62831
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             177     IN      A       142.250.196.174

;; Query time: 0 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Wed Jun 07 20:44:03 IST 2023
;; MSG SIZE  rcvd: 55
```

OTHER NETWORK COMMAND LINE TOOLS:

IFCONFIG:

ifconfig (interface configuration) is a command-line tool used to configure and display the network interface parameters on a Unix-like system. It allows you to view and modify the IP address, network masks, network interfaces, and other network-related configurations.

```
:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::b4d6:88d:4ad7:2c20  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:1e:14:a1  txqueuelen 1000  (Ethernet)
        RX packets 612832  bytes 893593807 (893.5 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 157212  bytes 16550900 (16.5 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 4831  bytes 513874 (513.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4831  bytes 513874 (513.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

IWCONFIG:

iwconfig is a command-line tool used to configure and display wireless network interface parameters on Linux systems. It provides information about wireless interfaces, such as SSID (network name), mode, channel, encryption, and signal strength0.

```
:~$ iwconfig
lo        no wireless extensions.

enp0s3    no wireless extensions.
```

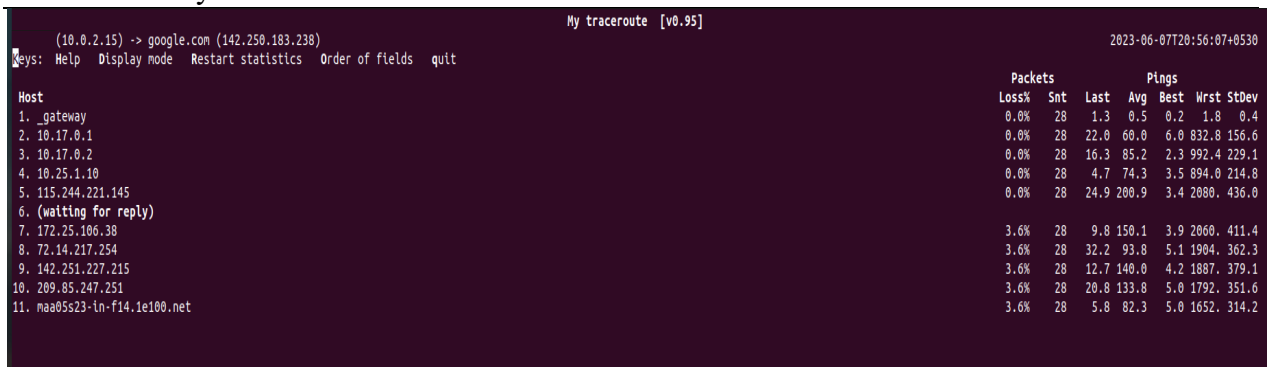
ROUTE:

Route is a command-line tool used to view and manipulate the IP routing table on a Unix-like system. It allows you to examine and modify the routing information used by the operating system to determine how network packets are forwarded between different networks or subnets.

```
:~$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    100    0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
link-local      0.0.0.0         255.255.0.0     U     1000   0        0 enp0s3
```

MTR:

MTR (My traceroute) is a command-line network diagnostic tool that combines the functionality of ping and traceroute. It continuously sends ICMP echo requests to the target host while providing detailed information about each hop in the route, including round-trip times, packet loss, and network latency.



Result:

In conclusion, network command line tools are essential utilities for network administrators, developers, and users who need to perform network-related tasks, diagnostics, and troubleshooting from the command line interface. These network command tools are tested and verified.

Simulation of Ping and Traceroute commands using Twisted Python

Ex. No. 2
Date:

PROBLEM STATEMENT:

To simulate PING and TRACEROUTE commands using Twisted Python.

PROBLEM DESCRIPTION:

a) PING

- Ping stands for **Packet Internet** or **Inter-Network Groper**.
- Relies on the Internet Control Message Protocol (ICMP) at the internet layer of TCP/IP.
- It's most basic use is to confirm network connectivity between two hosts.
- The Ping application should send ICMP Echo Request packets to a specified target host or IP address and measure the round-trip time (RTT) for each packet. The application should display the RTT for each packet received, as well as the overall statistics, including packet loss percentage and average RTT.

b) TRACEROUTE

- A Traceroute command is a command line tool that is generally used to locate the destination path from the host in the network.
- This command is useful when you want to know about the route and about all the hops that a packet takes.
- It is used in tracing and troubleshooting network problems.
- The Traceroute application should send UDP packets with increasing TTL values to a specified target host or IP address and print out the intermediate hops along the path. The application should display the IP address or hostname of each intermediate hop and the round-trip time (RTT) for each hop.

CODE:

a) PING – TWISTED PYTHON

```
import subprocess
from twisted.internet import reactor, defer

class PingProtocol:
    def init (self):
        self.deferred = defer.Deferred()

    def ping(self, host):
        process = subprocess.Popen(['ping', '-c', '4', host], stdout=subprocess.PIPE)
```

```
        output, error = process.communicate()
        if error:
            self.deferred.errback(error)
        else:
            self.deferred.callback(output)

def print_result(result):
    print(result.decode())

def print_error(failure):
    print(failure)

protocol = PingProtocol()
protocol.ping('google.com')
protocol.deferred.addCallbacks(print_result, print_error)
reactor.run()
```

b) TRACE ROUTE - TWISTED PYTHON

```
import subprocess
from twisted.internet import reactor, defer

class TracerouteCmd:
    def init (self):
        self.deferred = defer.Deferred()

    def traceroute(self, host):
        process = subprocess.Popen(['traceroute', '-m', '10', host], stdout=subprocess.PIPE)
        output, error = process.communicate()
        if error:
            self.deferred.errback(error)
        else:
            self.deferred.callback(output)

def print_result(result):
    print(result.decode())

def print_error(failure):
    print(failure)

protocol = TracerouteCmd()
protocol.traceroute('google.com')
protocol.deferred.addCallbacks(print_result, print_error)
reactor.run()
```

SAMPLE INPUT AND OUTPUT:

a) INPUT : 'google.com' (with packet limit as 4 using -c)

OUTPUT :

```
ssn@ssn-HP-Pro-Tower-400-G9-PCI-Desktop-PC:~/Downloads$ python3 Pingcmd.py
PING google.com (142.250.182.78) 56(84) bytes of data.
64 bytes from maa05s20-in-f14.1e100.net (142.250.182.78): icmp_seq=1 ttl=59 time=5.00 ms
64 bytes from maa05s20-in-f14.1e100.net (142.250.182.78): icmp_seq=2 ttl=59 time=6.88 ms
64 bytes from maa05s20-in-f14.1e100.net (142.250.182.78): icmp_seq=3 ttl=59 time=5.00 ms
64 bytes from maa05s20-in-f14.1e100.net (142.250.182.78): icmp_seq=4 ttl=59 time=6.86 ms

--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 4.995/5.933/6.880/0.934 ms
```

b) INPUT: 'google.com' (with packet limit as 10)

OUTPUT :

```
traceroute to google.com (142.250.182.78), 10 hops max, 60 byte packets
 1  _gateway (10.18.0.1)  9.746 ms  10.214 ms  10.181 ms
 2  10.18.0.2 (10.18.0.2)  5.470 ms  5.455 ms  5.442 ms
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  * * *
```

RESULT

Hence, the ping and traceroute commands are implemented using twisted python and verified the output successfully.

Simulation of Network Topologies - Bus, Star, Ring and Mesh using Twisted Python

Ex. No. 3

Date:

PROBLEM STATEMENT:

To simulate the four basic network topologies (Star, Mesh, Ring, and Bus) using Twisted Python.

PROBLEM DESCRIPTION:

a) **Star Topology:** It is a type of network topology in which every device of the network is individually connected to a central device, known as the hub. Any communication is done through this hub, and if this hub fails, then the entire network fails.
The individual connections to the central device (server), and communication between devices through the server must be done using Twisted Python.

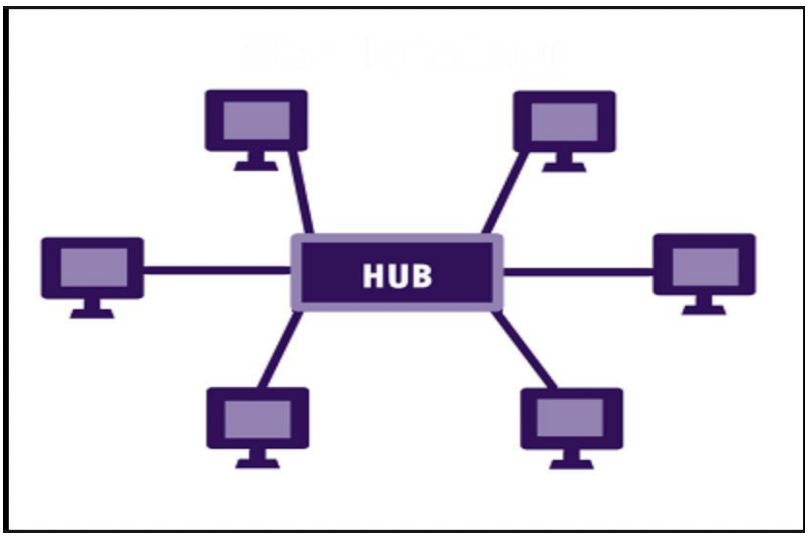


Figure 1: Star Topology

b) **Mesh Topology:** Mesh topology is a type of networking in which all the computers are interconnected to each other. In Mesh Topology, the connections between devices take place randomly. The connected nodes can be computers, switches, hubs, or any other devices. In this topology setup, even if one of the connections goes down, it allows other nodes to be distributed.
The interconnections and communication between each device must be simulated using Twisted Python.

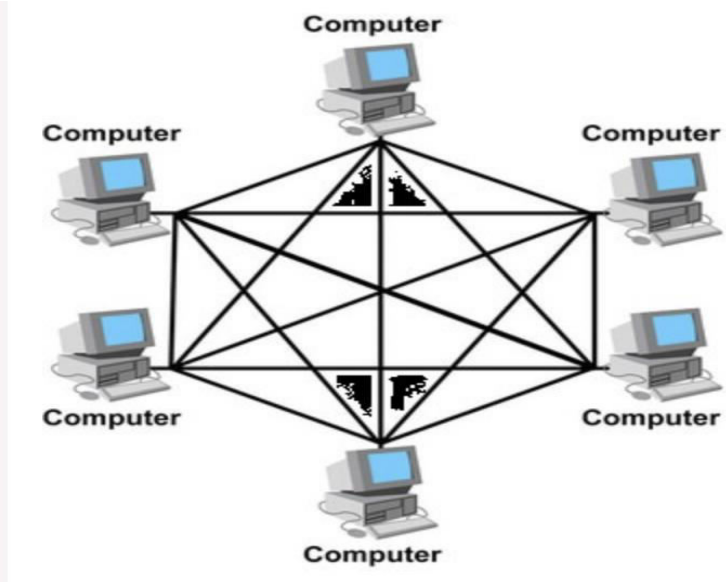


Figure 2: Mesh Topology

c) **Bus Topology:** Bus topology, also known as line topology, is a type of network topology in which all devices in the network are connected by one central RJ-45 network cable or coaxial cable. The single cable, where all data is transmitted between devices, is referred to as the bus, backbone, or trunk.

The connection of each individual device to the bus and communication between devices through the central bus must be shown using twisted python.

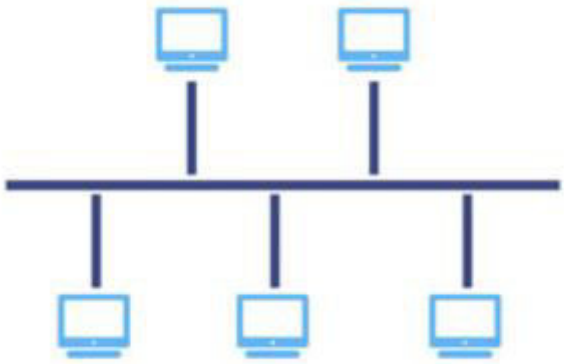


Figure 3: Bus Topology

d) **Ring Topology:** A ring topology is a network configuration where device connections create a circular data path. In a ring network, packets of data travel from one device to the next until they reach their destination. Most ring topologies allow packets to travel only in one direction, called a unidirectional ring network.

The unidirectional flow of data and the circular connection path of the ring topology network must be simulated using twisted python.

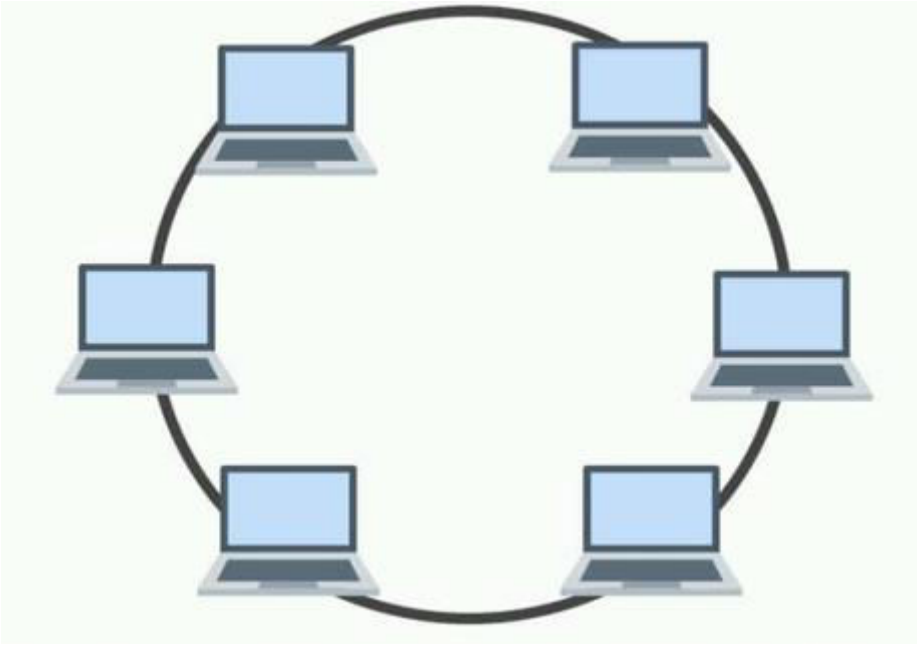


Figure 4: Ring Topology

CODE:

a) **Star topology:**

```
'''server implementation'''

from twisted.internet import protocol,reactor


class StarProtocol(protocol.Protocol):

    def __init__(self,factory):
        self.factory = factory   #factory that stores all the clients connected to the server
        self.name = None         #name of the client that will connect to the server

    def connectionMade(self):
        '''establishing a connection to the server'''

        print('New client connected: ',self.transport.getPeer())
        self.factory.clients.append(self)

    def connectionLost(self, reason):
```

```
        print("Client disconnected")
        self.factory.remove(self)

    def dataReceived(self, data):
        message = data.decode().strip()

        if not self.name:
            self.name = message
            print(self.name,' has connected to the server.')   #if the client has not connected to the server before

        else:
            if message.startswith('@'):                        #already existing client sends a message
                '''one client will send message to another client '''
                recipient, private_message = message[1:].split(":", 1)
                self.sendthroughServer(recipient, private_message)

            else:
                '''if destination is not specified, the message is simply sent to the server.'''
                self.transport.write(message)

    def sendthroughServer(self,recipient,message):
        self.transport.write(message)     #the message first goes to the server
        self.transport.write('message sending.....')
        self.sendPrivateMessage(recipient,message)      #the message is then sent to the destination through the server


    def sendPrivateMessage(self,recipient,message):
        for client in self.factory.clients:
            if client.name == recipient:
                client.transport.write(f"(Private) {self.name}: {message}\n".encode())
                break
        else:
            self.transport.write(f"Error: User {recipient} not found.\n".encode())


class StarFactory(protocol.Factory):

    def __init__(self):
        self.clients = []
    def buildProtocol(self, addr):
```

```
        return StarProtocol()


if __name__ == "__main__":
    reactor.listenTCP(8080, StarFactory())
    print("Server started. Listening on port 8080...")
    print("Enter client name to register. Enter @ before the starting of a message to send message to another client.")
    reactor.run()
```

b) Mesh Topology:

```
from twisted.internet import reactor, protocol

class MeshProtocol(protocol.Protocol):
    def __init__(self, factory):
        self.factory = factory
        self.name = None

    def connectionMade(self):
        self.factory.clients.append(self)
        print("New client connected.")

    def connectionLost(self, reason):
        self.factory.clients.remove(self)
        print("Client disconnected.")

    def dataReceived(self, data):
        message = data.decode().strip()

        if not self.name:
            self.name = message
            print(f"{self.name} joined the chat.")
        else:
            if message.startswith("@"):
                recipient, private_message = message[1:].split(":", 1)
                self.sendPrivateMessage(recipient, private_message)
            else:
                print(f"{self.name}: {message}")
                self.broadcastMessage(f"{self.name}: {message}")

    def sendPrivateMessage(self, recipient, message):
        for client in self.factory.clients:
            if client.name == recipient:
```

```
                client.transport.write(f"(Private) {self.name}: {message}\n".encode())
                break
        else:
            self.transport.write(f"Error: User {recipient} not found.\n".encode())

    def broadcastMessage(self, message):
        for client in self.factory.clients:
            if client != self:
                client.transport.write(f"{message}\n".encode())

class MeshFactory(protocol.Factory):
    def __init__(self):
        self.clients = []

    def buildProtocol(self, addr):
        return MeshProtocol(self)

if __name__ == "__main__":
    reactor.listenTCP(8000, MeshFactory())
    print("Bus server started.")
    print("Enter your name as first message to register. To send a message to a
particular username use '@username: message'.")
    reactor.run()
```

c) Ring Topology:

```
from twisted.internet import protocol,reactor

class RingProtocol(protocol.Protocol):

    def __init__(self,factory):
        self.factory = factory #factory that stores the clients connected to the server
        self.name = None       #name of the client that will connect to the server
    def connectionMade(self):
        '''establishing a connection to the server'''

        print('New client connected: ',self.transport.getPeer())
        self.factory.clients.append(self)

    def connectionLost(self, reason):

        print("Client disconnected")
        index = self.factory.clients.index(self)
```

```
        #getting the index of the client that is going to be removed

        self.factory.clients[index]=None

    def dataReceived(self, data):
        message = data.decode().strip()

        if not self.name:
            self.name = message
            print(self.name,' has connected to the server.')
            #if the client has not connected to the server before
            self.factory.names.append(self.name)

        else:
            if message.startswith('@'):
                '''one client will send message to another client '''
                recipient, private_message = message[1:].split(":", 1)
                receiver_index = self.factory.names.index(recipient)
                sender_index = self.factory.names.index(self.name)


                while sender_index != receiver_index:
                    sender_index += 1
                    if sender_index == len(self.factory.names):
                        sender_index = 0

                    if self.factory.clients[sender_index] is None:
                        '''the message can't be sent.'''
                        self.transport.write('link failure. message cannot be sent'.encode())
                        break

                    self.sendPrivateMessage(self.factory.names[sender_index],
private_message)


            else:
                '''if destination not specified, the message is sent to the server.'''
                self.transport.write(message.encode())



    def sendPrivateMessage(self,recipient,message):
        for client in self.factory.clients:
            if client is not None:
```

```
                if client.name == recipient:
                    client.transport.write(f"(Private) {self.name}: {message}\n".encode())
                    break
            else:
                self.transport.write(f"Error: User {recipient} not found.\n".encode())


class RingFactory(protocol.Factory):

    def __init__(self):
        self.clients = []
        self.names = []
    def buildProtocol(self, addr):
        return RingProtocol(self)


if __name__ == "__main__":
    reactor.listenTCP(8080, RingFactory())
    print("Server started. Listening on port 8080...")
    print('''Enter client name to register. Enter @ before the starting of a message
          to send message to another client.''')
    reactor.run()
```

d) Bus Topology:

```
from twisted.internet import reactor, protocol

class DropLink(protocol.Protocol):
    def __init__(self, factory):
        self.factory = factory
        self.name = None

    def connectionMade(self):
        self.factory.clients.append(self)
        print("New client connected to bus backbone.")

    def connectionLost(self, reason):
        self.factory.clients.remove(self)
        print("Client disconnected.")

    def dataReceived(self, data):
        message = data.decode().strip()
```

```
        if not self.name:
            self.name = message
            print(f"{self.name} connected to bus.")
        else:
            if message.startswith("@"):
                recipient, private_message = message[1:].split(":", 1)
                self.sendPrivateMessage(recipient, private_message)
            else:
                print(f"{self.name}: {message}")
                self.broadcastMessage(f"{self.name}: {message}")

    def sendPrivateMessage(self, recipient, message):
        for client in self.factory.clients:
            if client.name == recipient:
                client.transport.write(f"(Private) {self.name}: {message}\n".encode())
                break
        else:
            self.transport.write(f"Error: User {recipient} not found.\n".encode())

    def broadcastMessage(self, message):
        for client in self.factory.clients:
            if client != self:
                client.transport.write(f"{message}\n".encode())

class BusBackbone(protocol.Factory):
    def __init__(self):
        self.clients = []

    def buildProtocol(self, addr):
        return DropLink(self)

if __name__ == "__main__":
    reactor.listenTCP(8000, BusBackbone())
    print("Bus server started.")
    print("Enter your name as first message to register. To send a message to a
particular username use '@username: message'.")
    reactor.run()
```

SAMPLE INPUT/OUTPUT:

a) **Star Topology:**

Server:

```
Server started. Listening on port 8080...
Enter client name to register. Enter @ before the starting of a message to send
New client connected:  IPv4Address(type='TCP', host='127.0.0.1', port=49896)
client1  has connected to the server.
New client connected:  IPv4Address(type='TCP', host='127.0.0.1', port=59678)
client2  has connected to the server.
 hello client1
New client connected:  IPv4Address(type='TCP', host='127.0.0.1', port=41026)
client3  has connected to the server.
 hello from client3
```

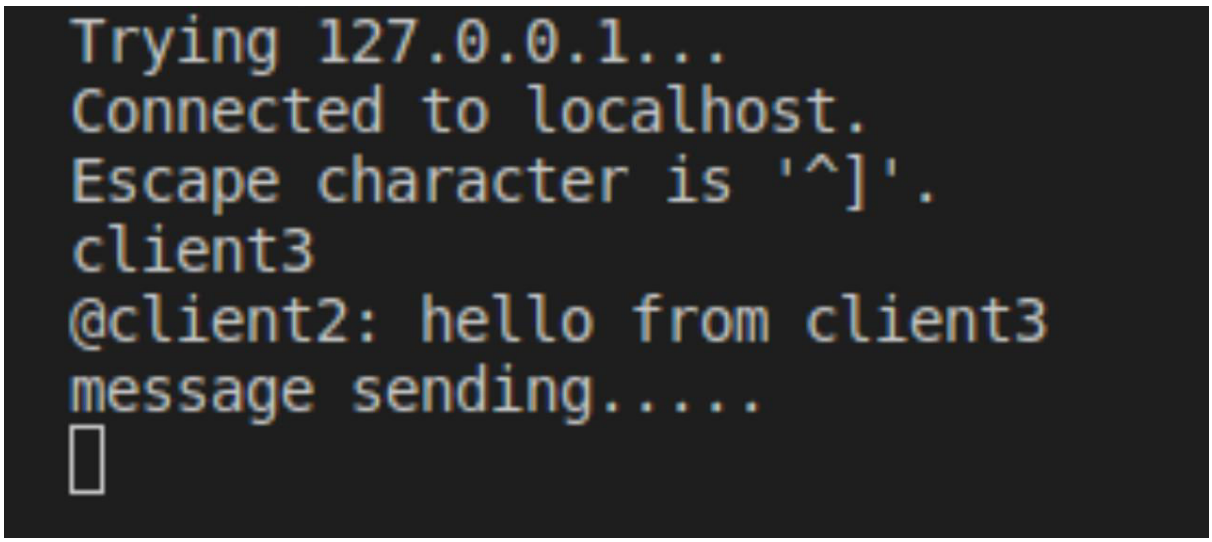
Client1:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
client1
(Private) client2:  hello client1
```

Client2:

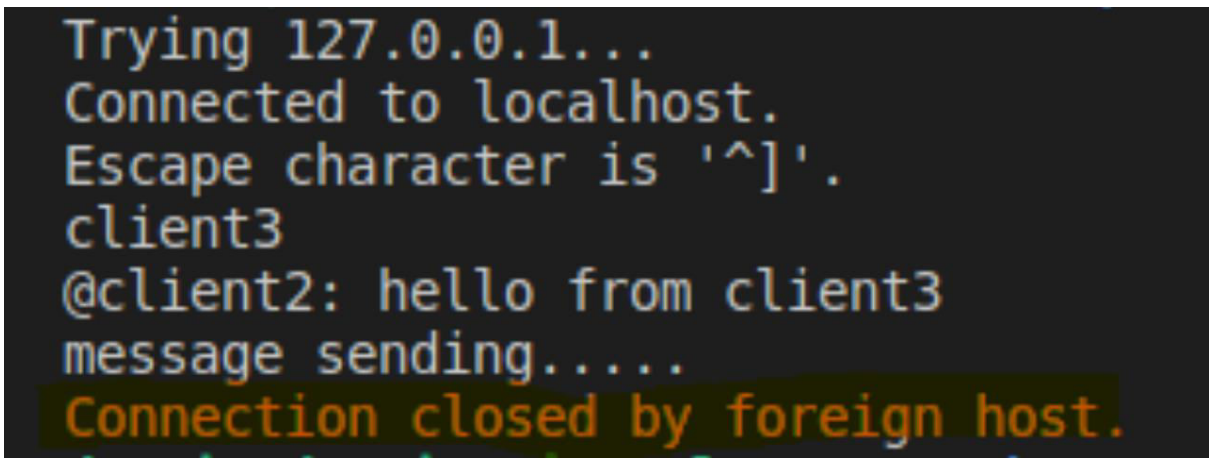
```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
client2
@client1: hello client1
message sending.....
(Private) client3:  hello from client3
```

Client3:



If the hub/server goes down, the entire network goes down:

Client

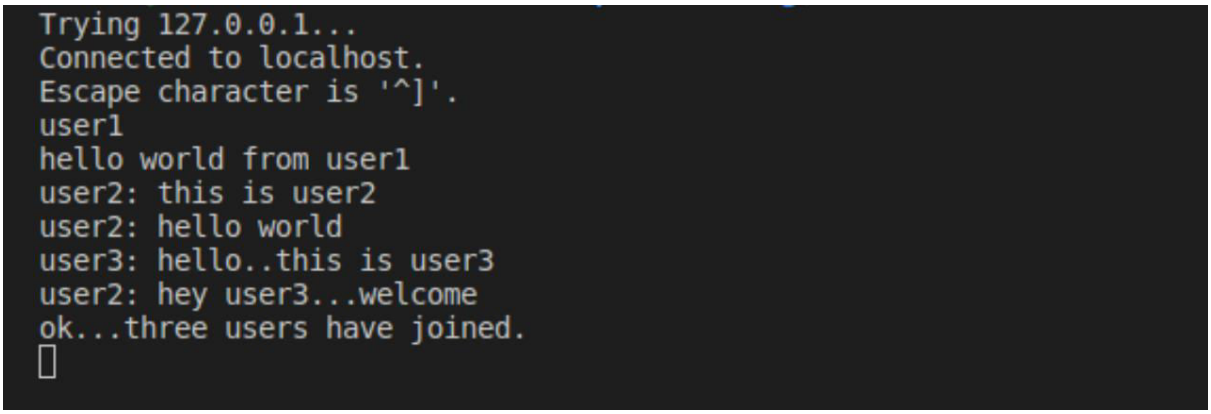


b) Mesh Topology:

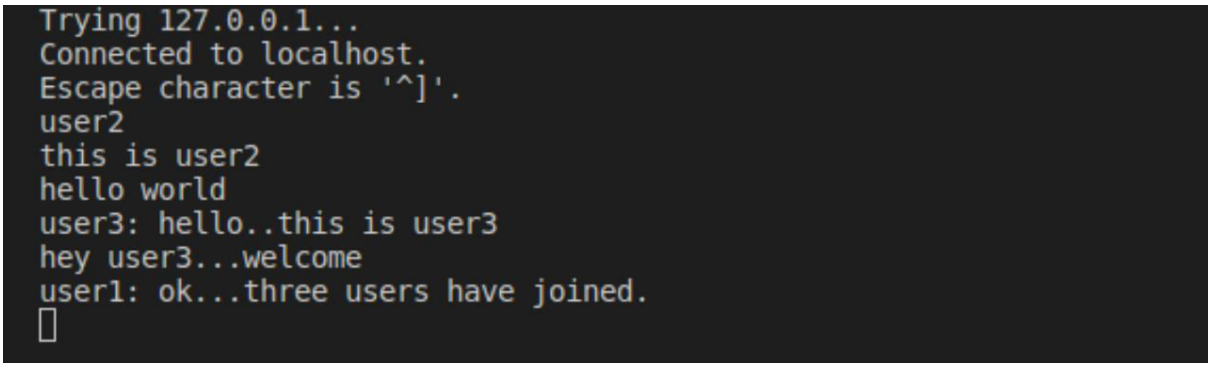
Server:

Clients-
Every node can communicate with every other node.

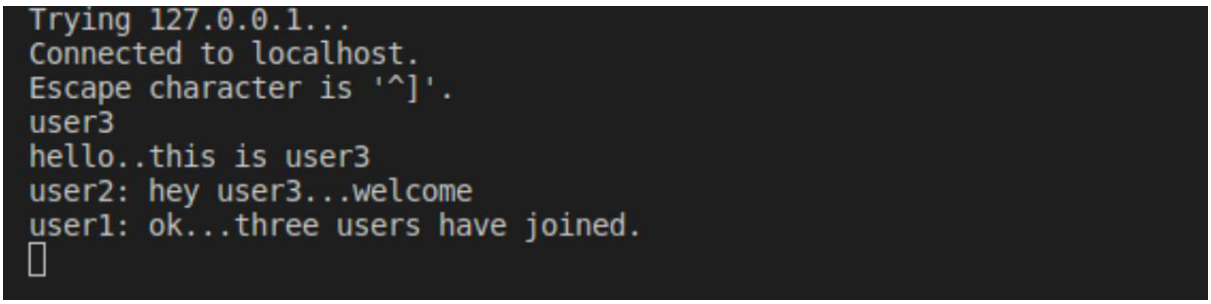
Client1:



Client2:

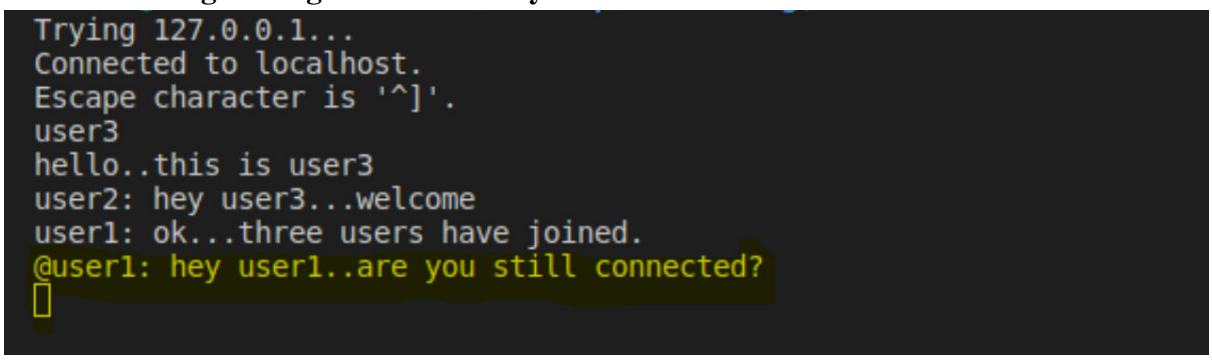


Client3-

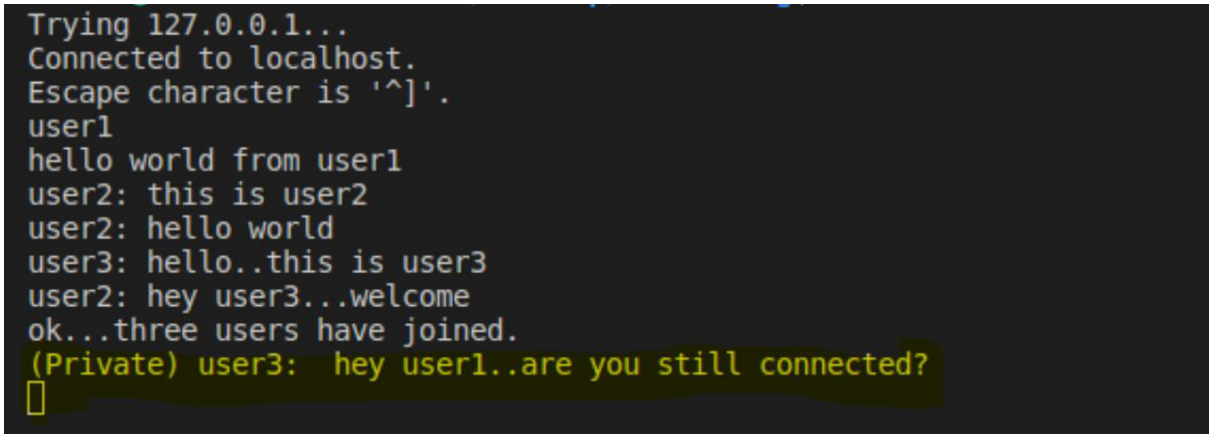


Any node can also communicate with only one specific node.

Client3 sending message to Client1 only.

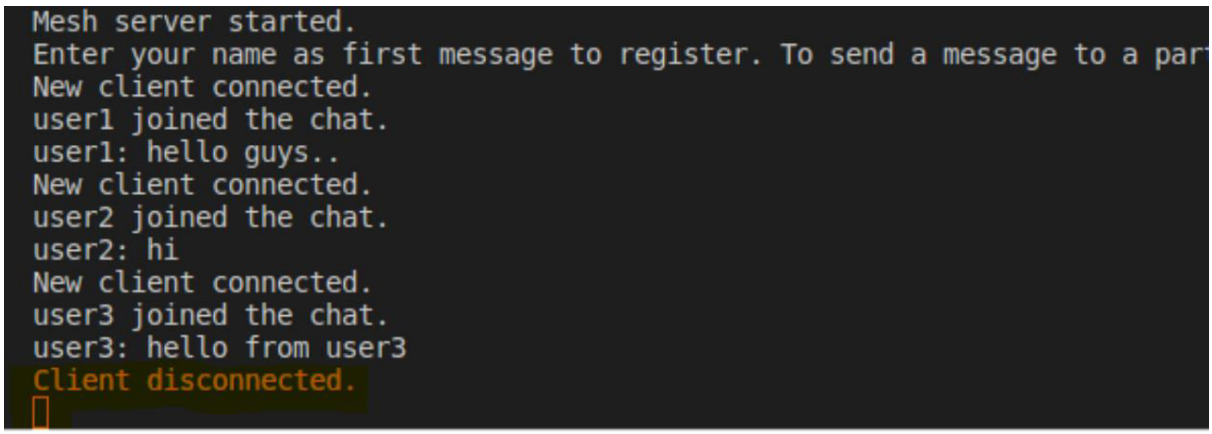


Client1:



If user2 disconnects, user1 can still send a message to user3 via another path:

Server:



User1:

```
Mesh server started.
Enter your name as first message to register. To send a message to a part
New client connected.
user1 joined the chat.
user1: hello guys..
New client connected.
user2 joined the chat.
user2: hi
New client connected.
user3 joined the chat.
user3: hello from user3
Client disconnected.
```

User2

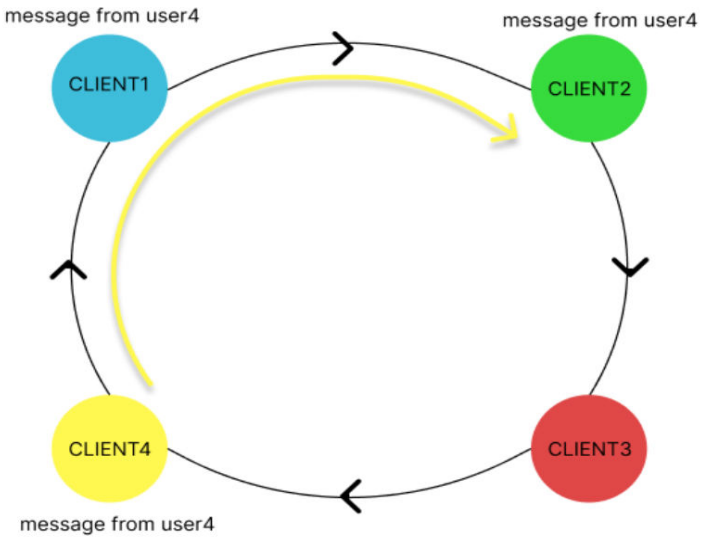
```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
user3
hello from user3
(Private) user1:  can you receive this message?
@user1: yes...user1
```

c) Ring Topology:

Server

```
Server started. Listening on port 8080...
Enter client name to register. Enter @ before the starting of a message to ser
user1
New client connected:  IPv4Address(type='TCP', host='127.0.0.1', port=46480)
user2  has connected to the server.
New client connected:  IPv4Address(type='TCP', host='127.0.0.1', port=33394)
user3  has connected to the server.
New client connected:  IPv4Address(type='TCP', host='127.0.0.1', port=41694)
user4  has connected to the server.
```

Client4 sends a message to client2. The message is travelled through the right neighbour node until client2 is reached.



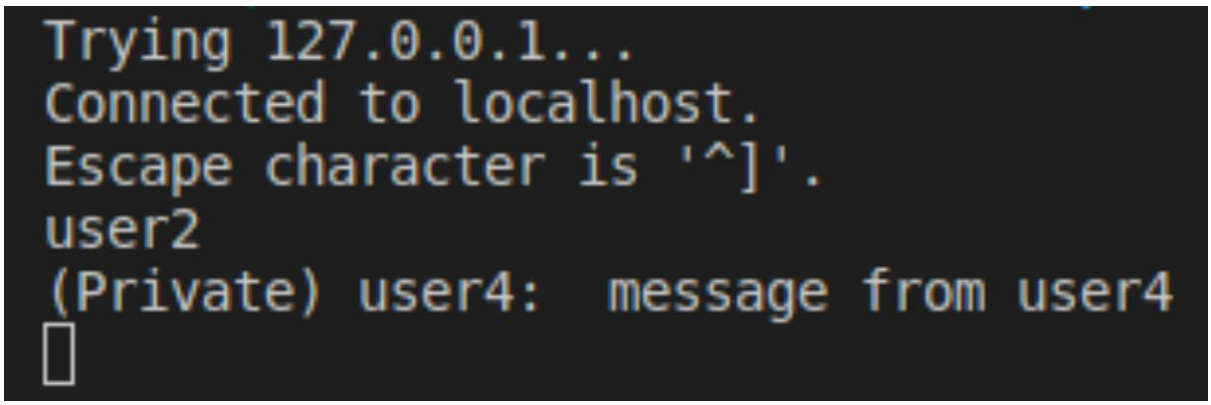
Client4:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
user4
@user2: message from user4
```

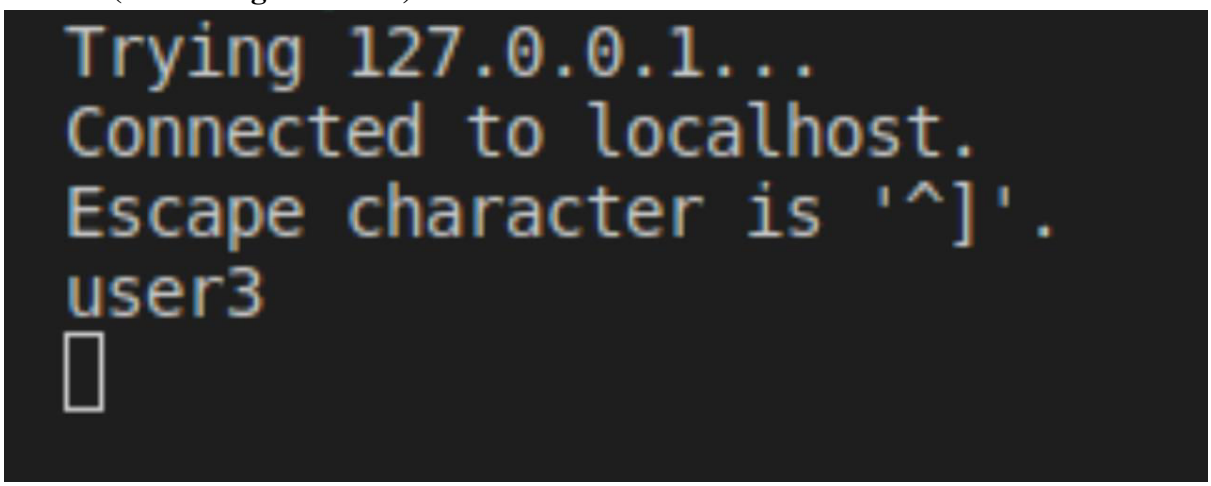
Client1:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
user1
(Private) user4:  message from user4
```

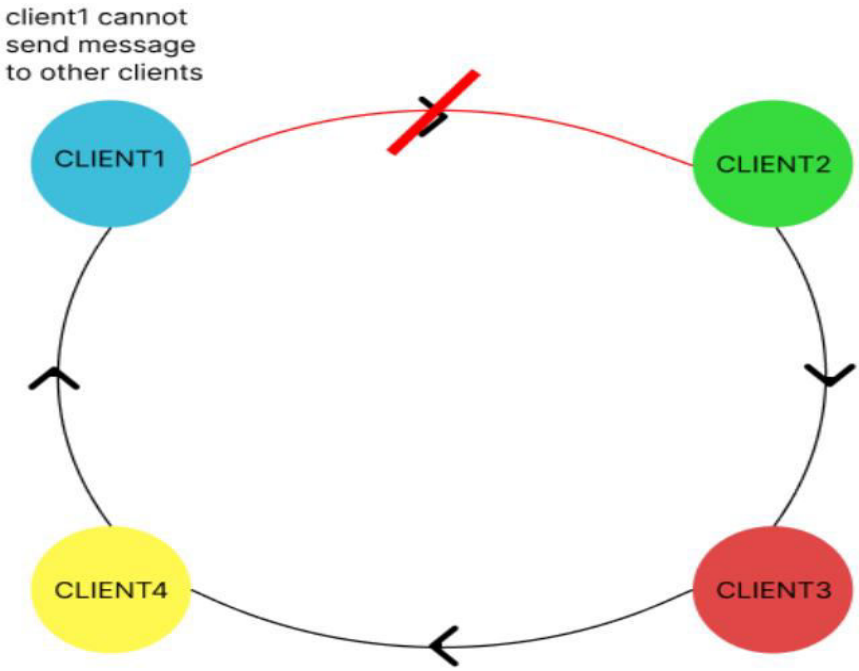
Client2:



Client3 (No message received)



If the link between any two nodes fails, the first node will not be able to send a message to any other node.



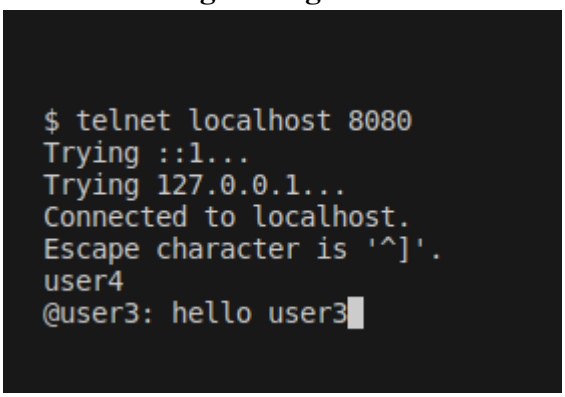
Client4 cannot send a message to Client3.

Client2 has been disconnected:

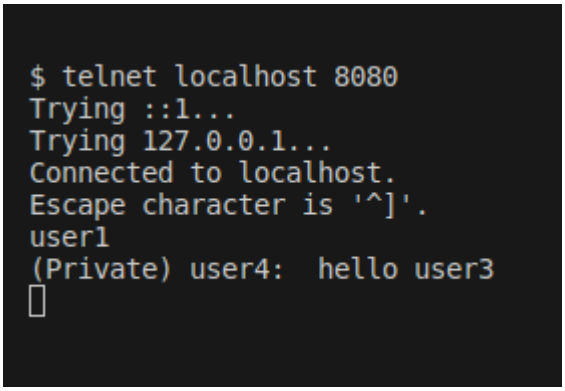
```
$ /bin/python3 /home/it4a1/Downloads/ring.py
Server started. Listening on port 8080...
Enter client name to register. Enter @ before th
e starting of a message
          to send message to another client.
New client connected:  IPv4Address(type='TCP', h
ost='127.0.0.1', port=34492)
user1  has connected to the server.
New client connected:  IPv4Address(type='TCP', h
ost='127.0.0.1', port=32984)
user2  has connected to the server.
New client connected:  IPv4Address(type='TCP', h
ost='127.0.0.1', port=36916)
user3  has connected to the server.
New client connected:  IPv4Address(type='TCP', h
ost='127.0.0.1', port=44836)
user4  has connected to the server.
Client disconnected
```

The message is successfully sent to client1, but since client2 has been disconnected, the message won't be transmitted after that.

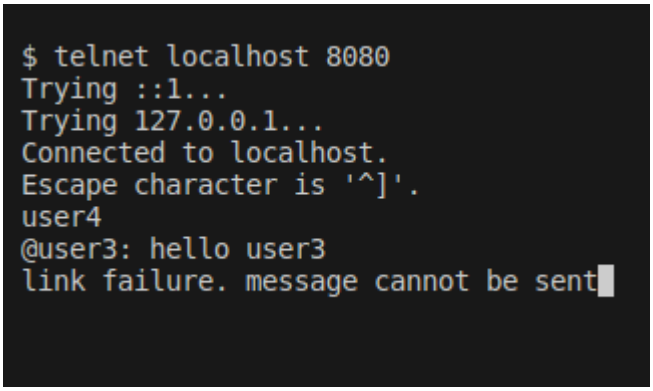
Client4 sending message to client3:



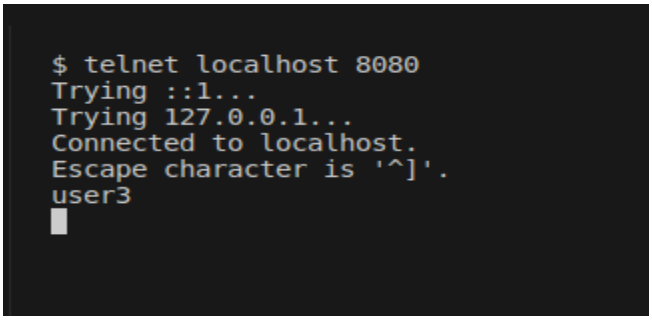
Client1 receives the message:



Client4 gets the link failure message:

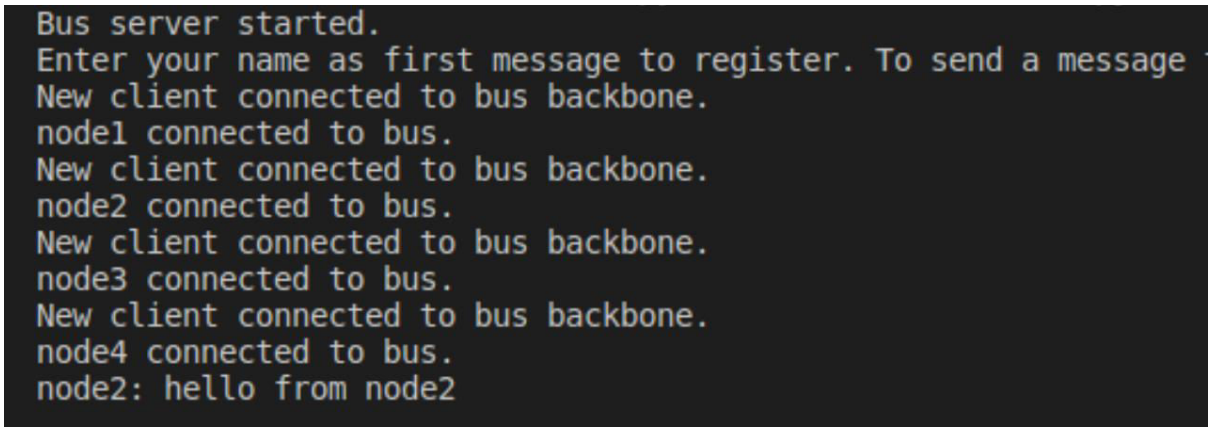


But client3 does not get the message:

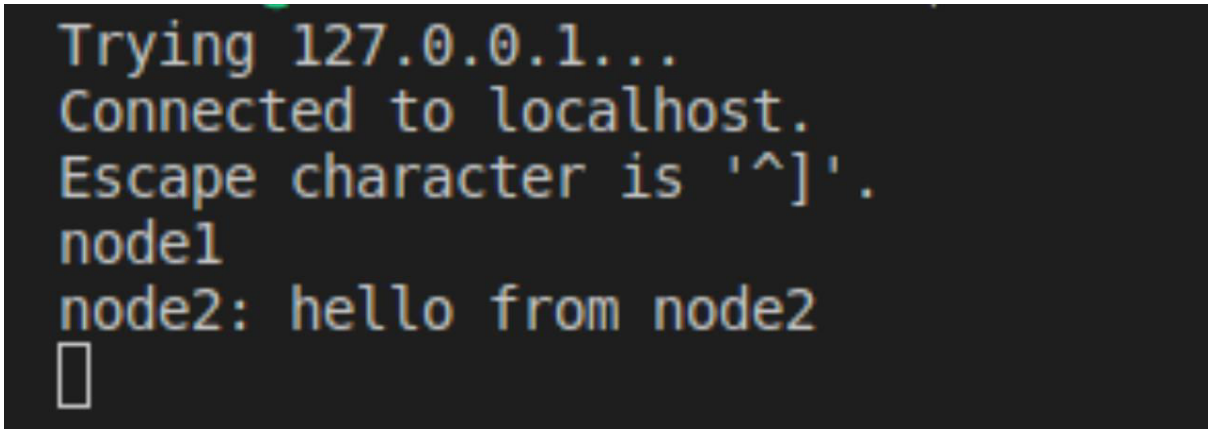


d) Bus Topology:

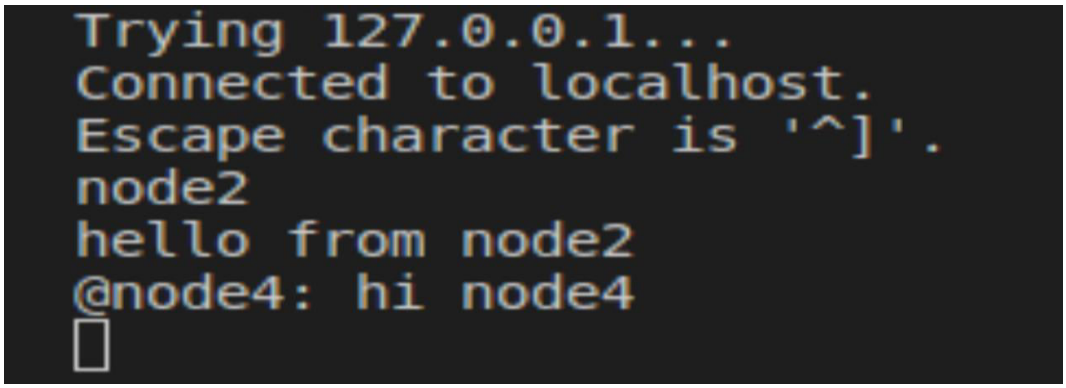
Bus Server (Backbone):



Client1:



Client2:



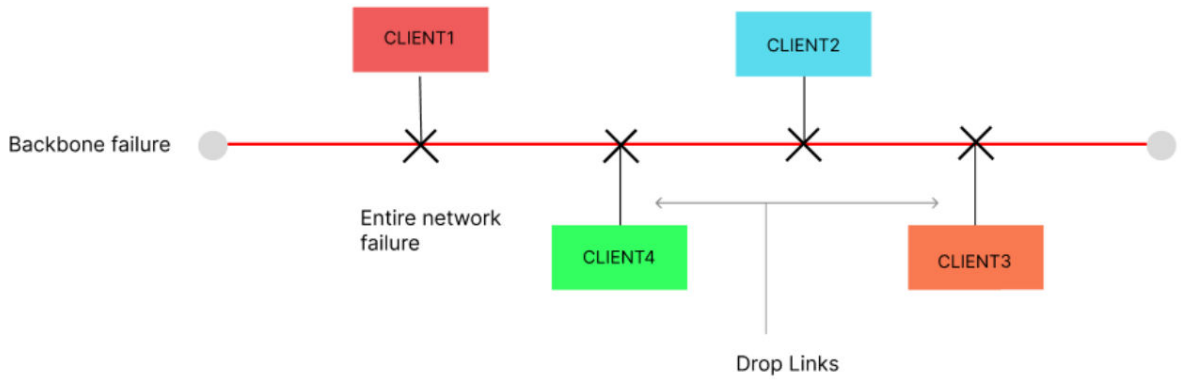
Client3:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
node3
node2: hello from node2
```

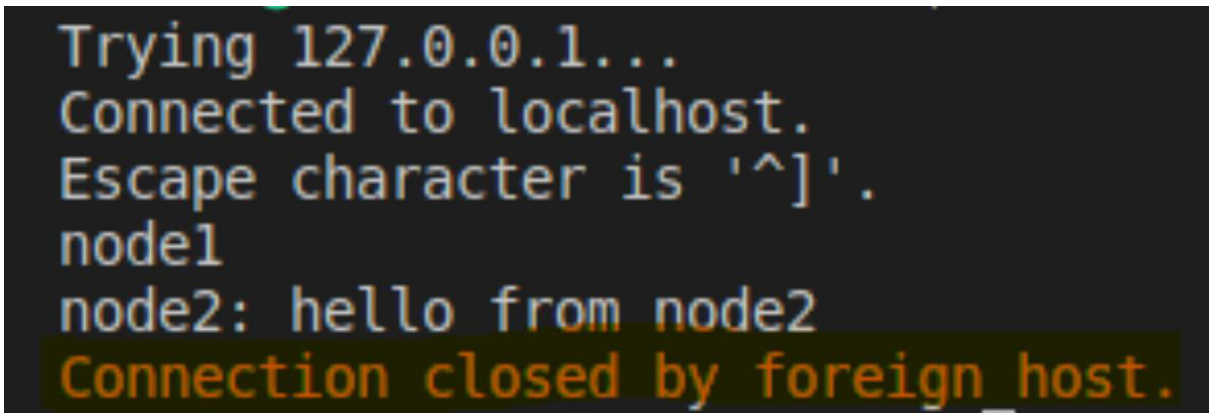
Client4:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
node4
node2: hello from node2
(Private) node2:  hi node4
```

If backbone fails:



Client:



RESULT:

Hence, the four basic network topologies such as Star, Mesh, Ring and Bus have been implemented using Twisted Python and verified the output successfully.

Network Packet Capturing and Analyzing using Wireshark

Ex. No. 4

Date:

PROBLEM STATEMENT:

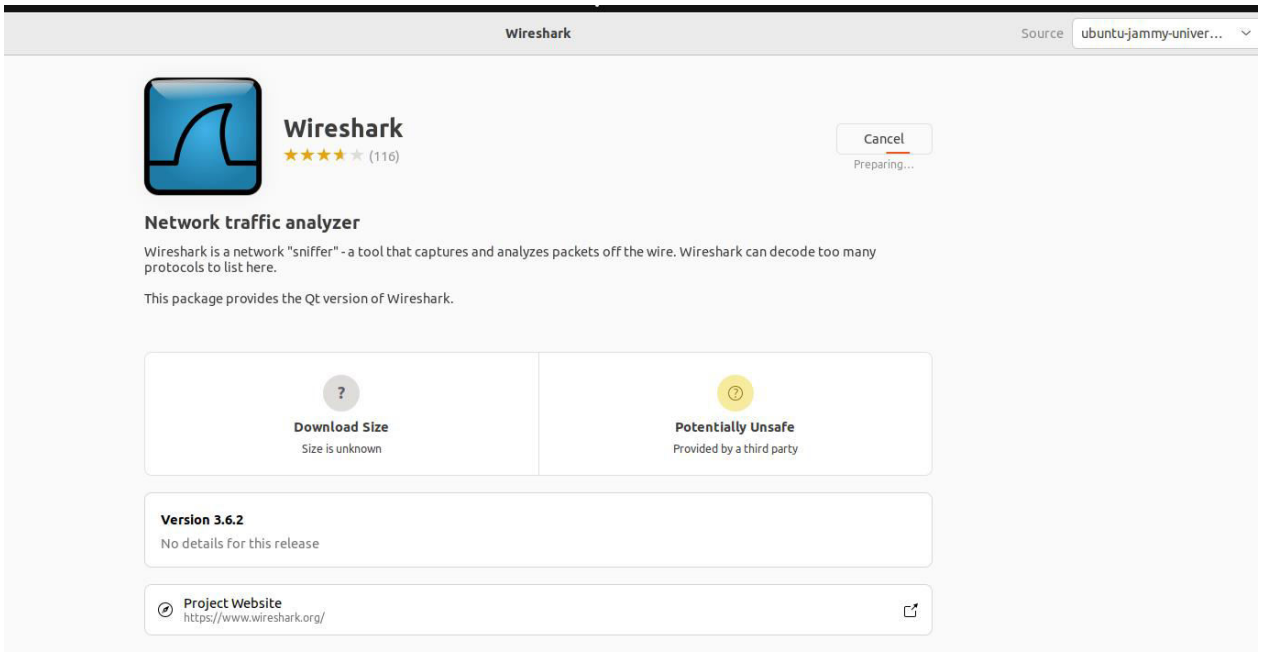
To perform network capture using Wireshark by applying various filters

PROBLEM DESCRIPTION:

Wireshark is free open-source packet analyser. It is mainly used for network troubleshooting and analysis. Using Wireshark software, network capture has to done and analysis has to be performed.

INSTALLATION:

1. Open ubuntu software and in the explore tab search for Wireshark.



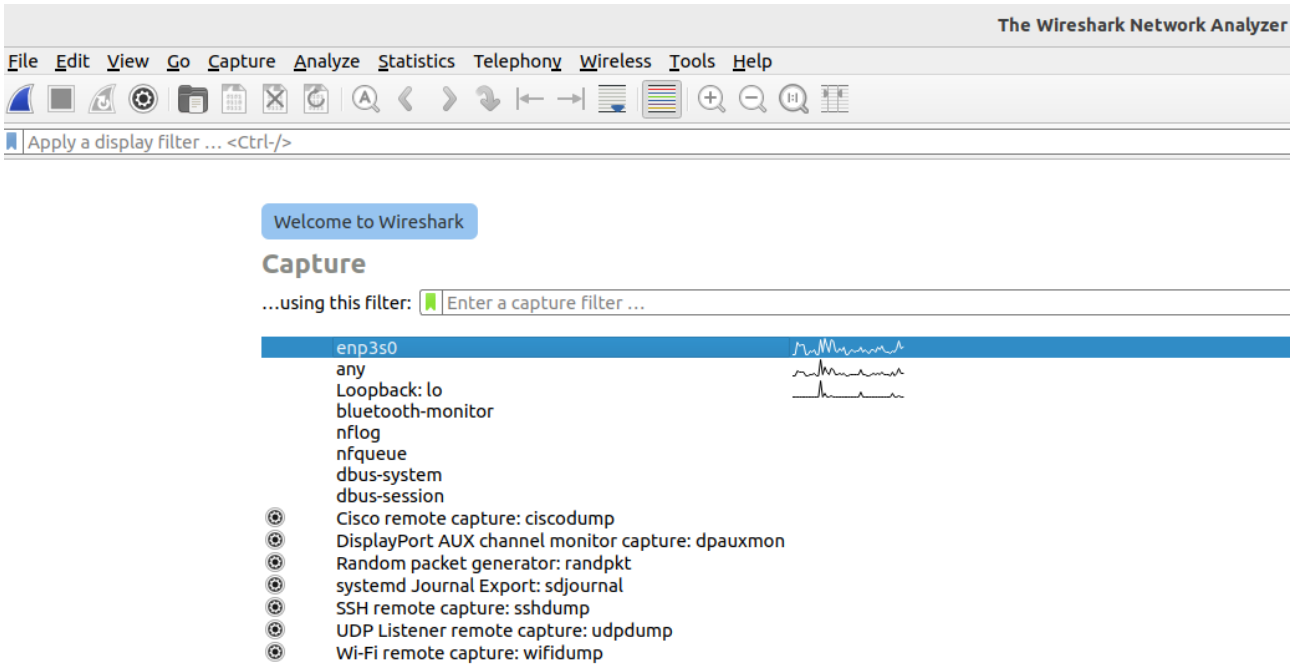
Procedure:

1. **To run**: enter the command **sudo wireshark** in ubuntu terminal

```
student@SELABIT-1:~$ sudo wireshark
 ** (wireshark:12998) 13:26:32.361945 [GUI WARNING] -- QStandardPaths: XDG_RUNTIME_DIR not set, default
 ** (wireshark:12998) 13:26:38.833273 [Capture MESSAGE] -- Capture Start ...
 ** (wireshark:12998) 13:26:38.876302 [Capture MESSAGE] -- Capture started
 ** (wireshark:12998) 13:26:38.876342 [Capture MESSAGE] -- File: "/tmp/wireshark_enp3s0ZT8N51.pcapng"
 ** (wireshark:12998) 13:26:48.811548 [GUI WARNING] -- failed to create compose table
 ** (wireshark:12998) 13:31:13.518071 [Capture MESSAGE] -- Capture Stop ...
 ** (wireshark:12998) 13:31:13.552435 [Capture MESSAGE] -- Capture stopped.
 ** (wireshark:12998) 13:31:28.436648 [Capture MESSAGE] -- Capture Start ...
 ** (wireshark:12998) 13:31:28.501626 [Capture MESSAGE] -- Capture started
 ** (wireshark:12998) 13:31:28.501680 [Capture MESSAGE] -- File: "/tmp/wireshark_enp3s09FDW51.pcapng"
 ** (wireshark:12998) 13:31:48.253581 [Capture MESSAGE] -- Capture Stop ...
 ** (wireshark:12998) 13:31:48.340245 [Capture MESSAGE] -- Capture stopped.
 ** (wireshark:12998) 13:32:36.873979 [Capture MESSAGE] -- Capture Start ...
 ** (wireshark:12998) 13:32:36.942046 [Capture MESSAGE] -- Capture started
```

2. Home screen

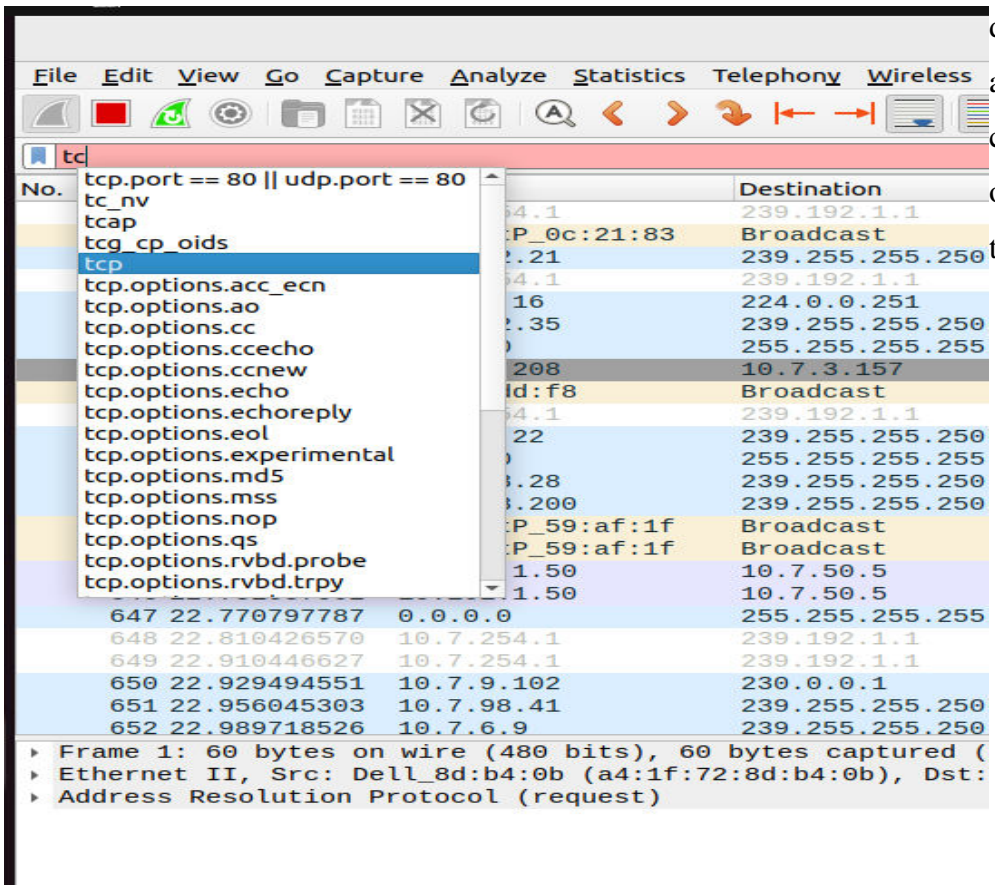
This is the home screen of the Wireshark software that appears as soon as the software opens



3. Applying Filters

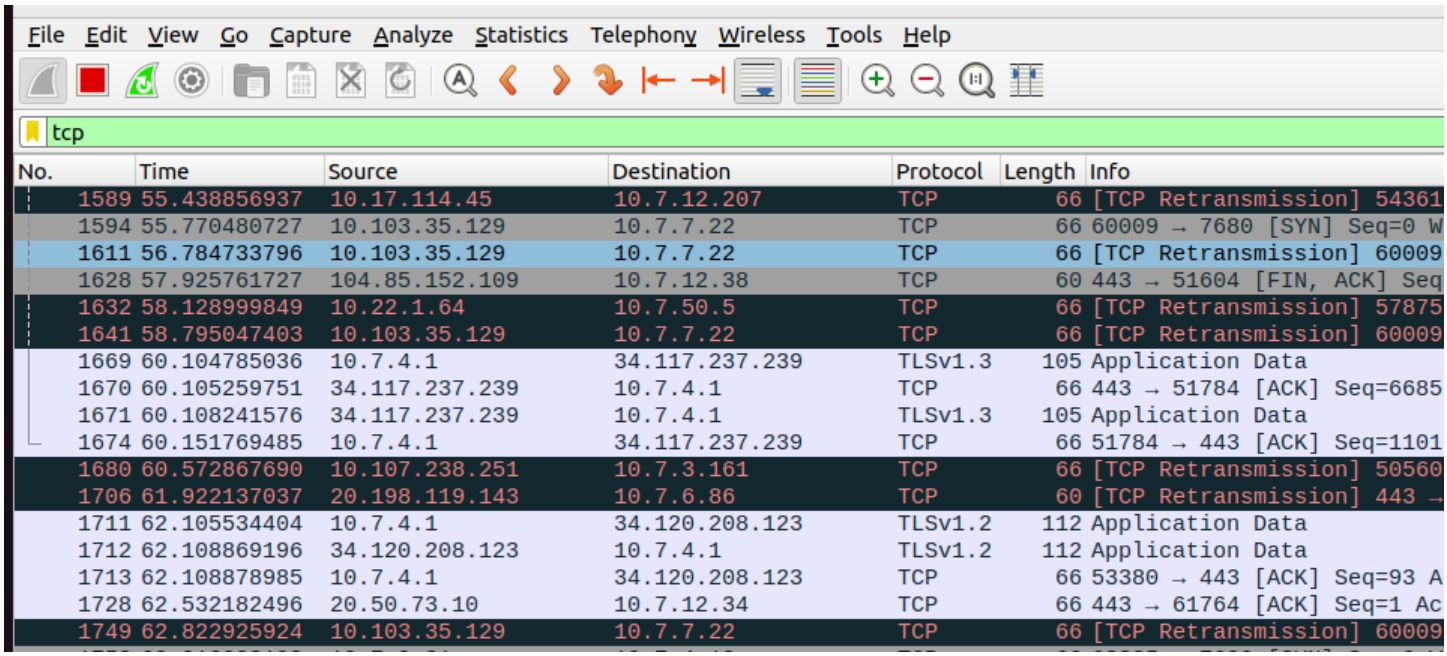
Below the menu bar, spot the display filter bar and enter the name of the protocol. This is

how a display filter is applied and the corresponding output is shown on the screen.



a) TCP packets

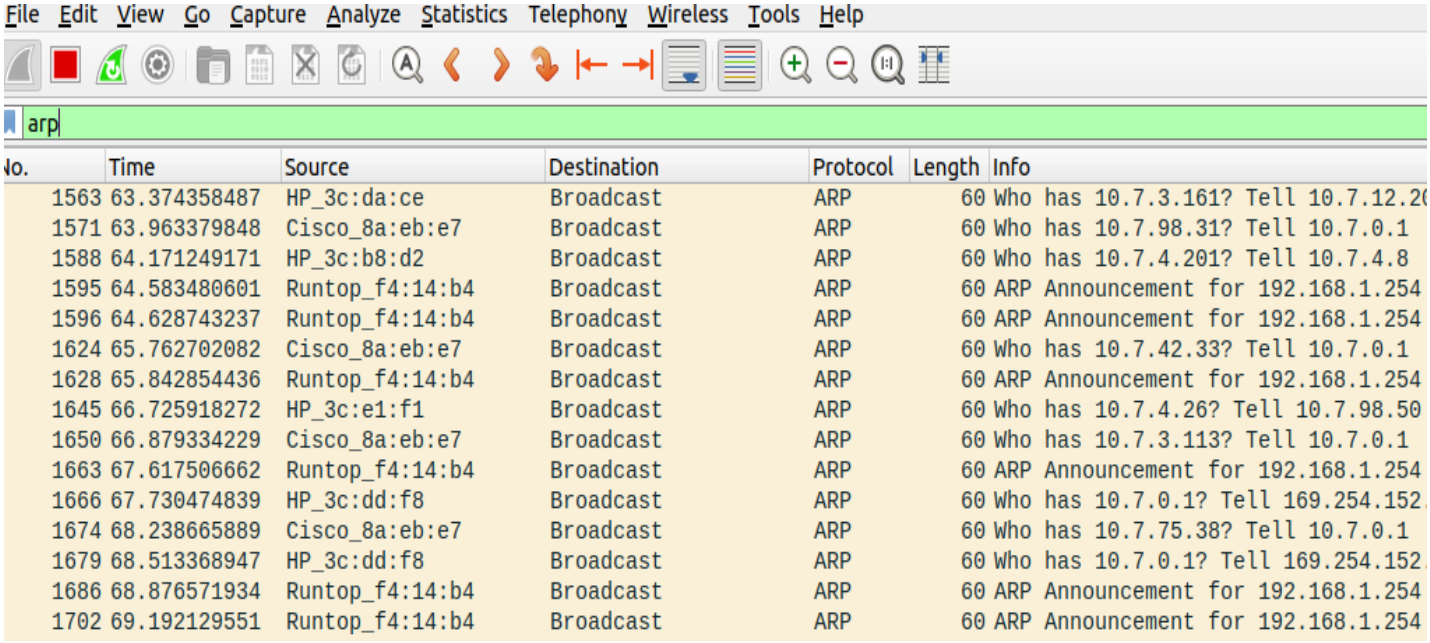
It is designed to send packets across the internet and ensure the successful delivery of data and



messages over network.

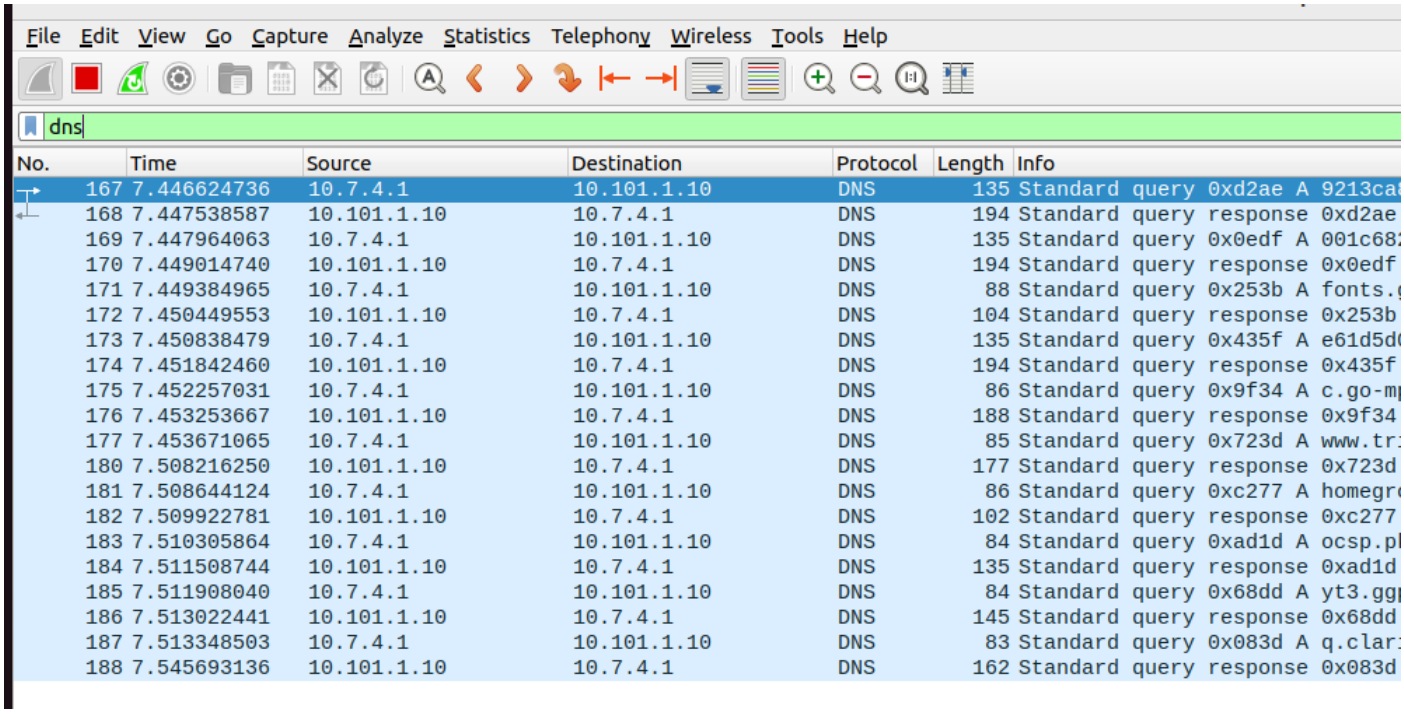
b) ARP packets

The Address Resolution Protocol is a communication protocol used for discovering the link layer address, such as a MAC address, associated with a given internet layer address.



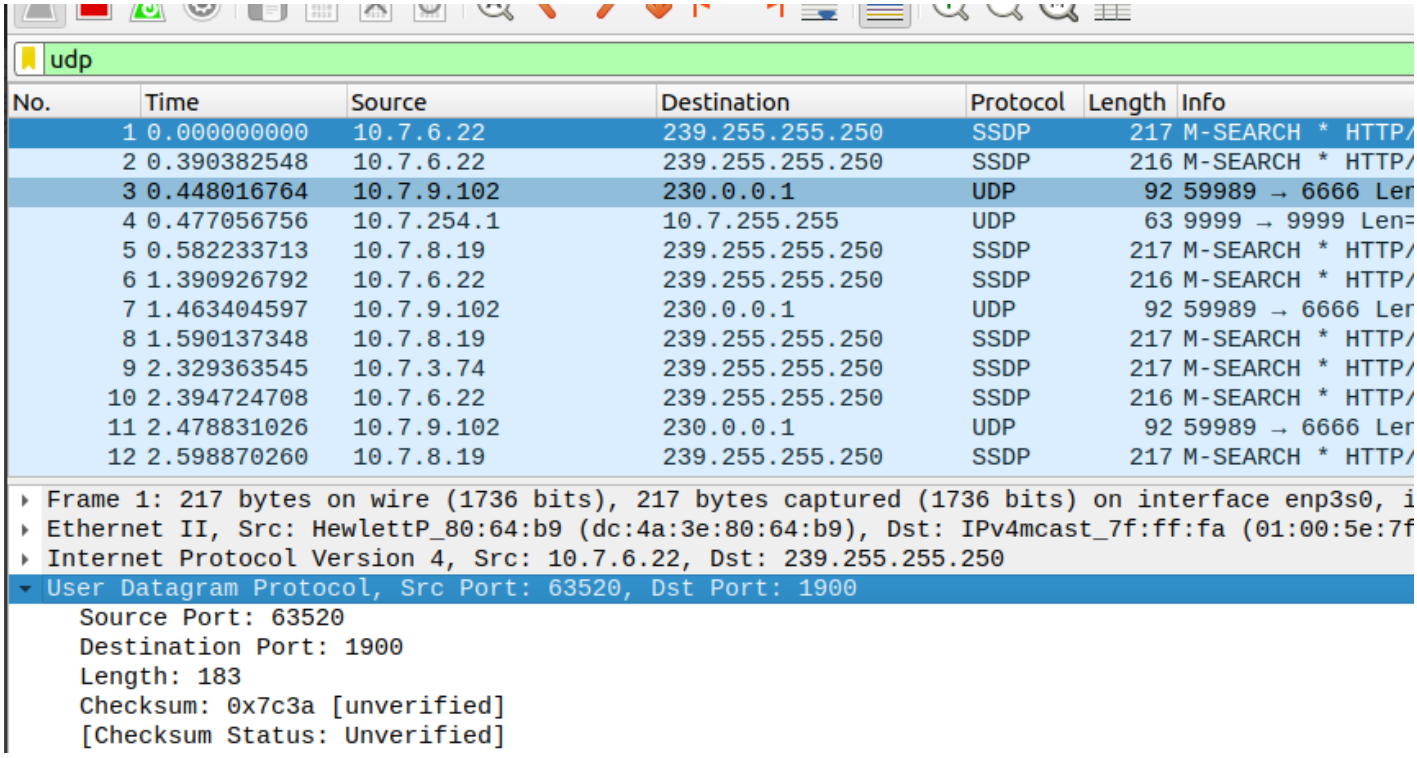
c) DNS

Domain Name System (DNS) turns domain names into IP addresses, which allow browsers to get to websites and other internet resources. Every device on the internet has an IP address, which other devices can use to locate the device.



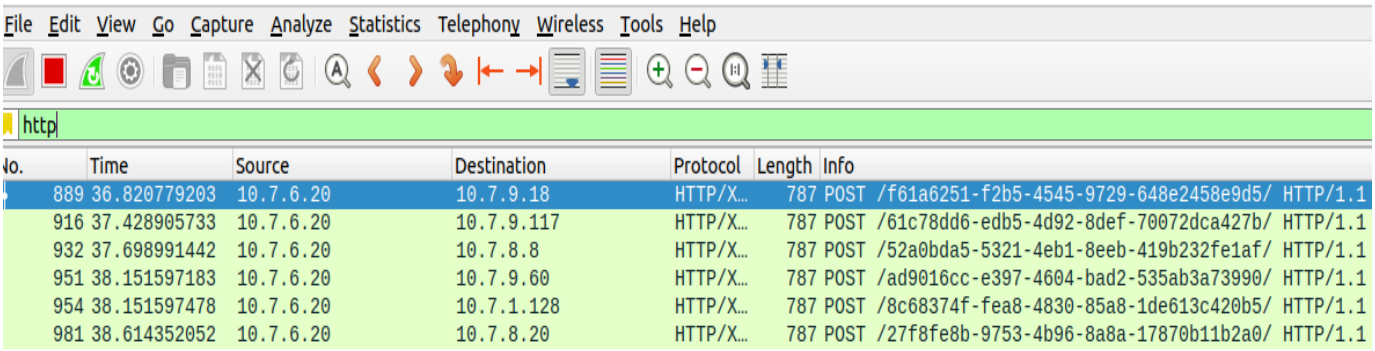
d) UDP

UDP is a short form for User Datagram protocol. It is one of the simplest transport layer protocol. It is a connectionless and unreliable transport protocol**.**



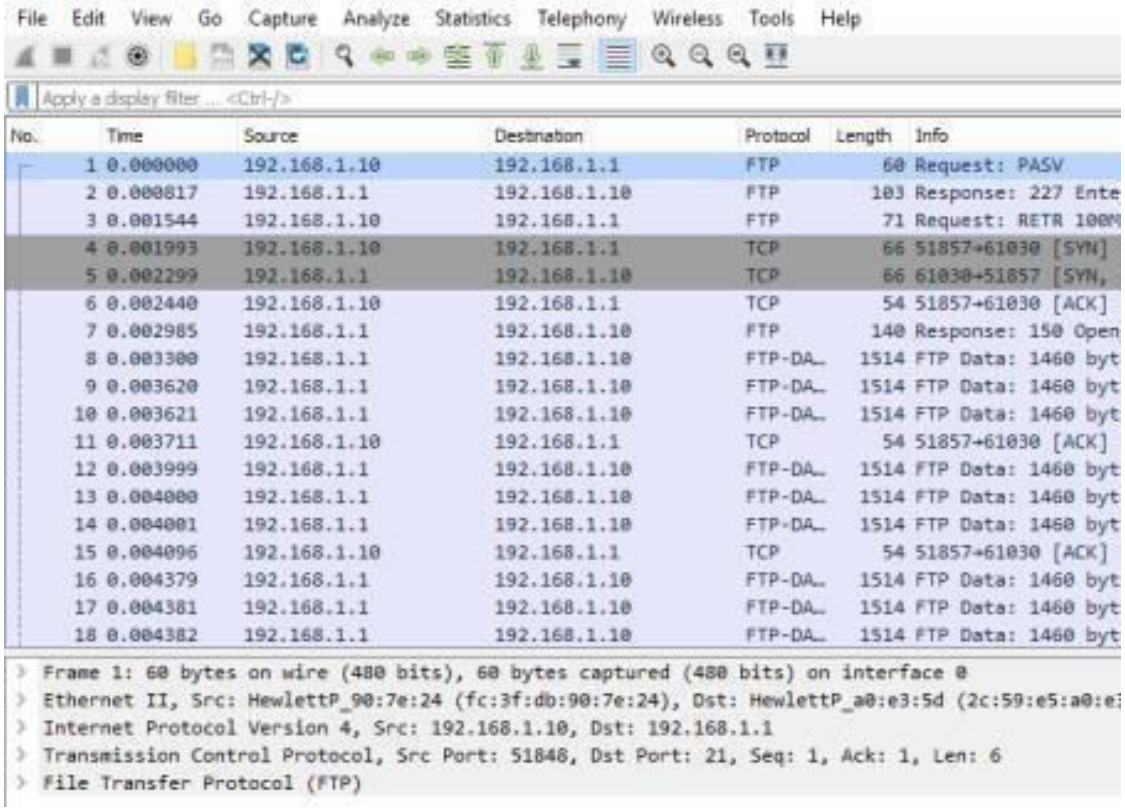
e) HTTP

Hypertext Transfer Protocol,It uses encryption for secure communication over a computer network, and is widely used on the Internet. In HTTPS, the communication protocol is encrypted using Transport Layer Security or, formerly, Secure Sockets Layer.



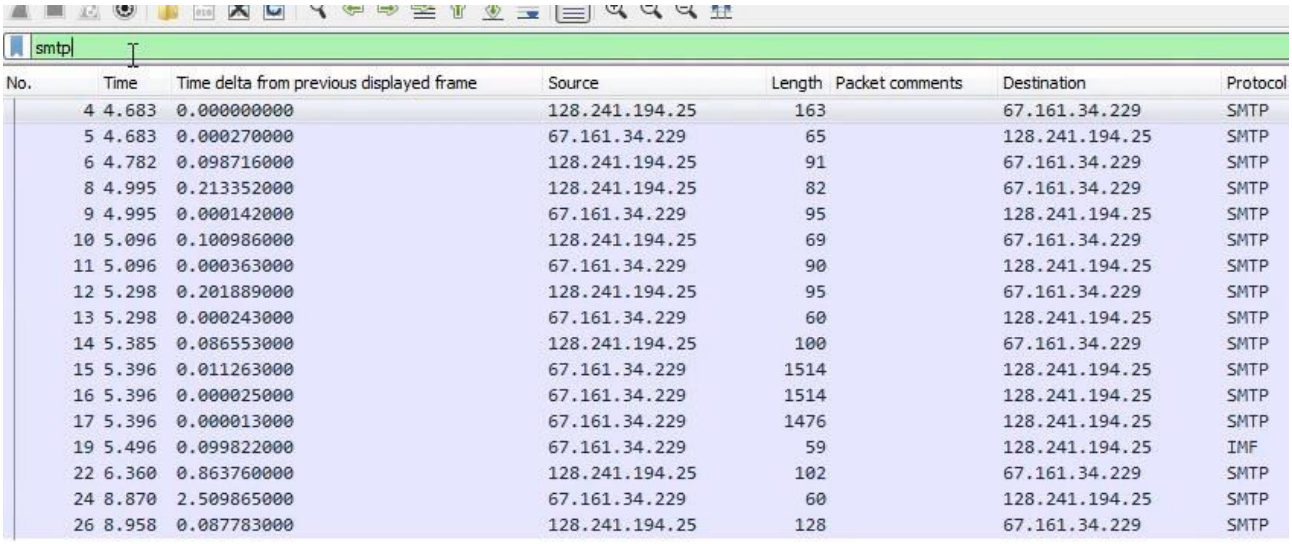
f) FTP

FTP (File Transfer Protocol) is a network protocol for transmitting files between computers over Transmission Control Protocol/Internet Protocol (TCP/IP) connections.



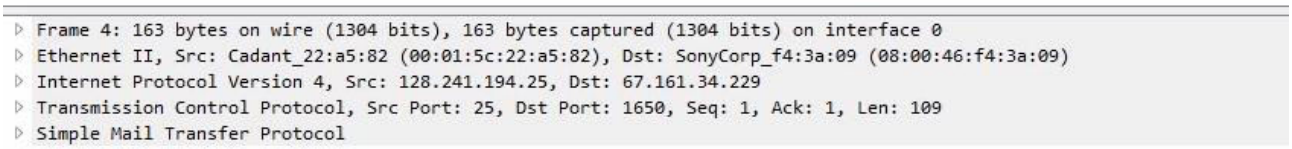
g) SMTP

The Simple Mail Transfer Protocol is an Internet standard communication protocol for electronic mail transmission. Mail servers and other message transfer agents use SMTP to send and receive mail messages



smtp

No.	Time	Time delta from previous displayed frame	Source	Length	Packet comments	Destination	Protocol
4	4.683	0.000000000	128.241.194.25	163		67.161.34.229	SMTP
5	4.683	0.000270000	67.161.34.229	65		128.241.194.25	SMTP
6	4.782	0.098716000	128.241.194.25	91		67.161.34.229	SMTP
8	4.995	0.213352000	128.241.194.25	82		67.161.34.229	SMTP
9	4.995	0.000142000	67.161.34.229	95		128.241.194.25	SMTP
10	5.096	0.100986000	128.241.194.25	69		67.161.34.229	SMTP
11	5.096	0.000363000	67.161.34.229	90		128.241.194.25	SMTP
12	5.298	0.201889000	128.241.194.25	95		67.161.34.229	SMTP
13	5.298	0.000243000	67.161.34.229	60		128.241.194.25	SMTP
14	5.385	0.086553000	128.241.194.25	100		67.161.34.229	SMTP
15	5.396	0.011263000	67.161.34.229	1514		128.241.194.25	SMTP
16	5.396	0.000025000	67.161.34.229	1514		128.241.194.25	SMTP
17	5.396	0.000013000	67.161.34.229	1476		128.241.194.25	SMTP
19	5.496	0.099822000	67.161.34.229	59		128.241.194.25	IMF
22	6.360	0.863760000	128.241.194.25	102		67.161.34.229	SMTP
24	8.870	2.509865000	67.161.34.229	60		128.241.194.25	SMTP
26	8.958	0.087783000	128.241.194.25	128		67.161.34.229	SMTP

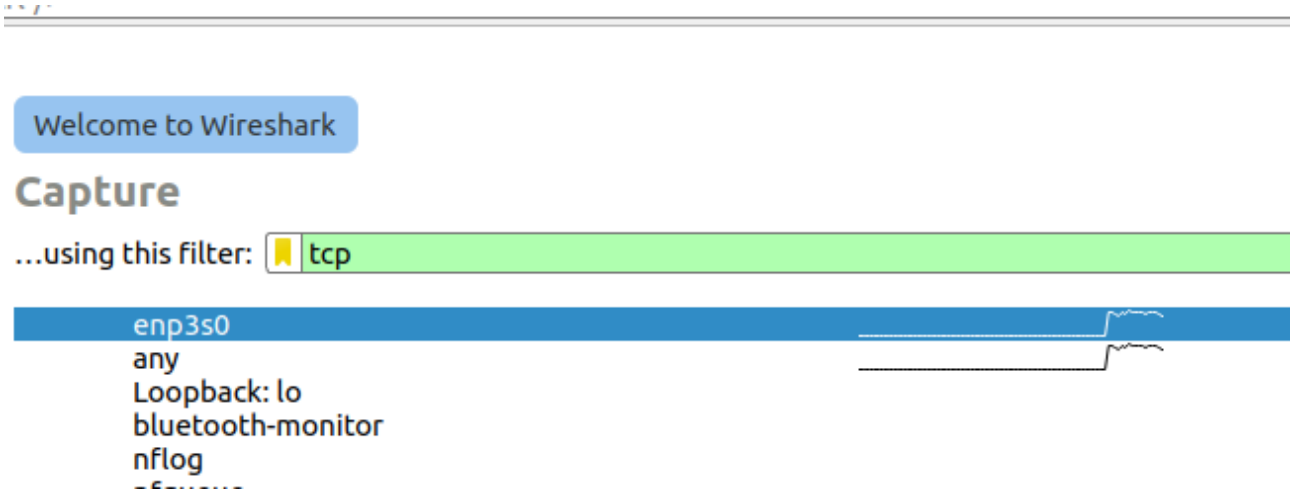


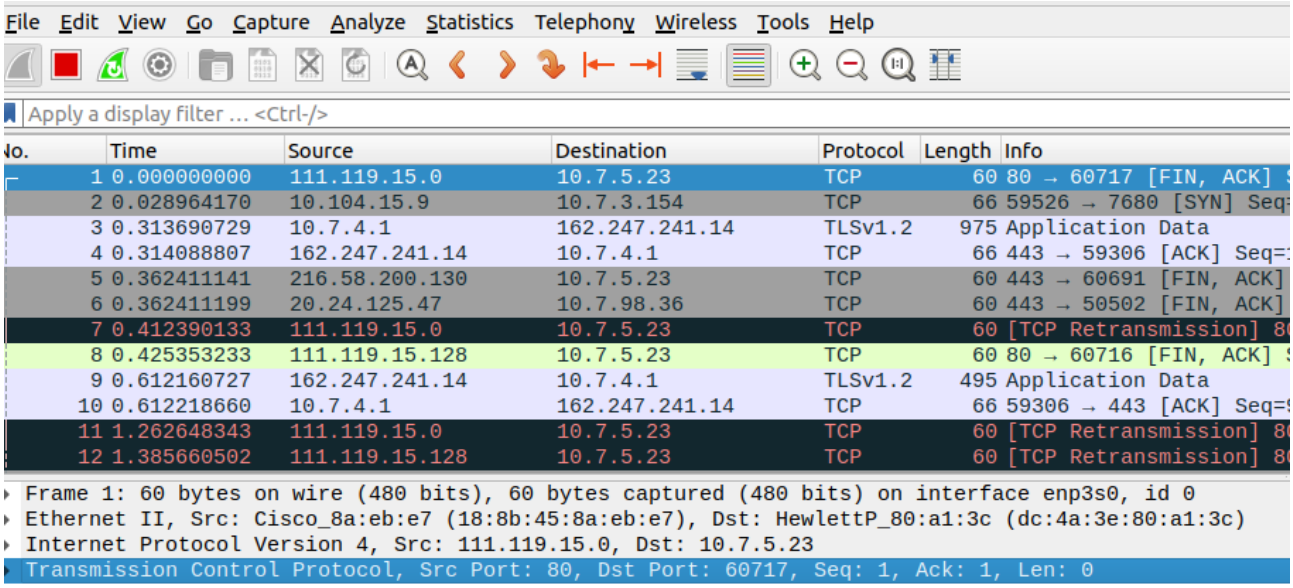
```
▷ Frame 4: 163 bytes on wire (1304 bits), 163 bytes captured (1304 bits) on interface 0
▷ Ethernet II, Src: Cadant_22:a5:82 (00:01:5c:22:a5:82), Dst: SonyCorp_f4:3a:09 (08:00:46:f4:3a:09)
▷ Internet Protocol Version 4, Src: 128.241.194.25, Dst: 67.161.34.229
▷ Transmission Control Protocol, Src Port: 25, Dst Port: 1650, Seq: 1, Ack: 1, Len: 109
▷ Simple Mail Transfer Protocol
```

Capture filters are different from display filters. Capture filters are applied before capturing the packets, it is used to capture only the mentioned protocol packets. Whereas display filters are used

To display only the mentioned protocol packets from the captured packets.

Applying Capture Filters





Result:

Thus, the required network packets has been captured and analysed using Wireshark.

Network Simulation using Packet Tracer

Ex. No. 5
Date:

PROBLEM STATEMENT:

To implement a simple network and different cases of the same using the CISCO packet tracer application.

PROBLEM DESCRIPTION:

1. Creating and transfer of packets in mesh topology when all cables are intact, using packet tracer.
2. Creating and transfer of packets in mesh topology when one cable is damaged or removed, using packet tracer.
3. Creating and transfer of packets in star topology when all cables are intact, using packet tracer.
4. Creating and transfer of packets in star topology when one cables is damaged or removed, using packet tracer.

Installation Procedure:

1. Create an account on CISCO website.
2. Click on the installation link after logging in: https://skillsforall.com/resources/lab-downloads?courseLang=en-US

Configuring a Simple Network Using Packet Tracer:

Objectives:

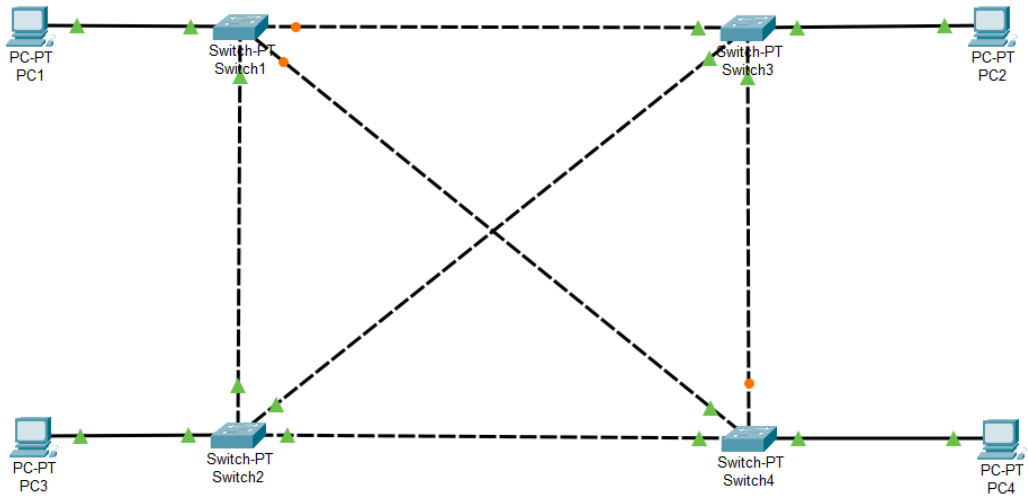
1. Connect devices as per the topology.
2. Basic configuration of the devices.
3. Assign IP address to the PC's.
4. Test and verify the network connectivity (ping command)

TOPOLOGY 1

IP Addressing Table:

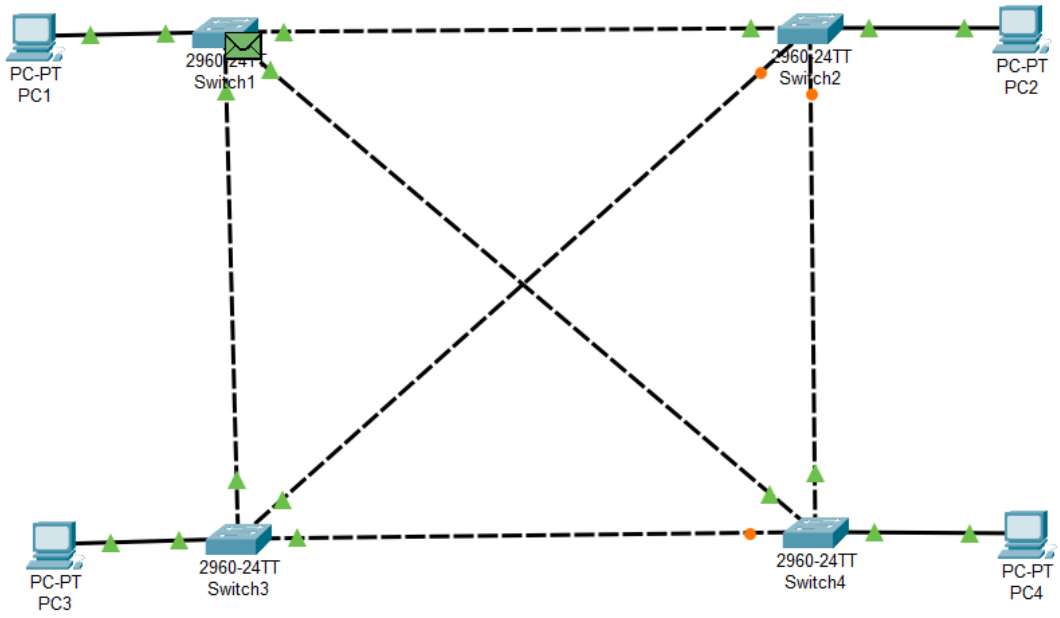
S.no	Devices	IP Address	Subnet Mask
1.	PC1	192.168.0.1	255.255.255.0
2.	PC2	192.168.0.2	255.255.255.0
3.	PC3	192.168.0.3	255.255.255.0
4.	PC4	192.168.0.4	255.255.255.0

Mesh Topology (CASE 1: All cables intact):

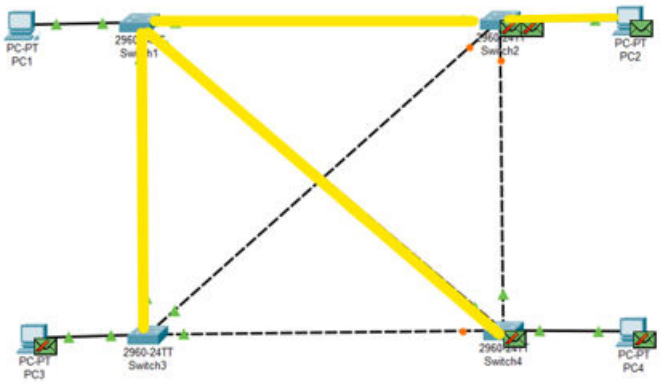


k
ets from source address: PC3 to other PC's]

Initial:



Final:



Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	Edit	Delete
	Successful	PC1	PC2	ICMP		0.000	N	0	(edit)	(delete)

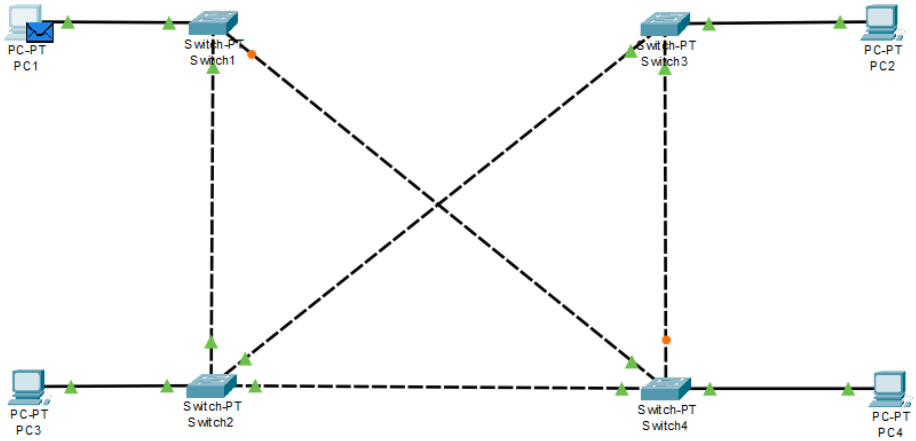
Using ping command to verify connectivity:

```
C:\>ping 192.168.0.1

Pinging 192.168.0.1 with 32 bytes of data:

Reply from 192.168.0.1: bytes=32 time=11ms TTL=128
Reply from 192.168.0.1: bytes=32 time=4ms TTL=128
Reply from 192.168.0.1: bytes=32 time=16ms TTL=128
Reply from 192.168.0.1: bytes=32 time=7ms TTL=128

Ping statistics for 192.168.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 4ms, Maximum = 16ms, Average = 9ms
```

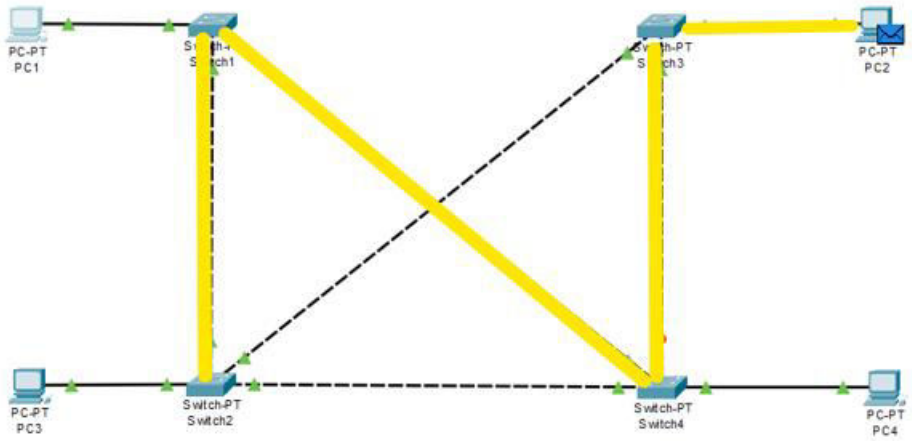
Mesh Topology (CASE 2: One cable is damaged):

[Mesh topology when one cable is removed/damaged]

Initial:



Final:



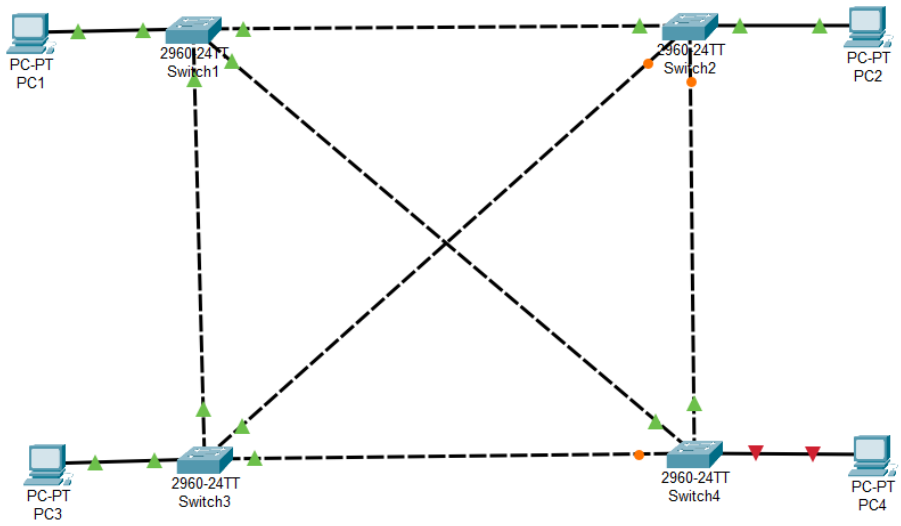
Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	Edit	Delete
	Successful	PC1	PC2	ICMP		0.000	N	0	(edit)	

Using Ping Command to Verify Connectivity:

```
C:\>ping 192.168.0.1

Pinging 192.168.0.1 with 32 bytes of data:

Reply from 192.168.0.1: bytes=32 time=11ms TTL=128
Reply from 192.168.0.1: bytes=32 time=4ms TTL=128
Reply from 192.168.0.1: bytes=32 time=16ms TTL=128
Reply from 192.168.0.1: bytes=32 time=7ms TTL=128

Ping statistics for 192.168.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 4ms, Maximum = 16ms, Average = 9ms
```

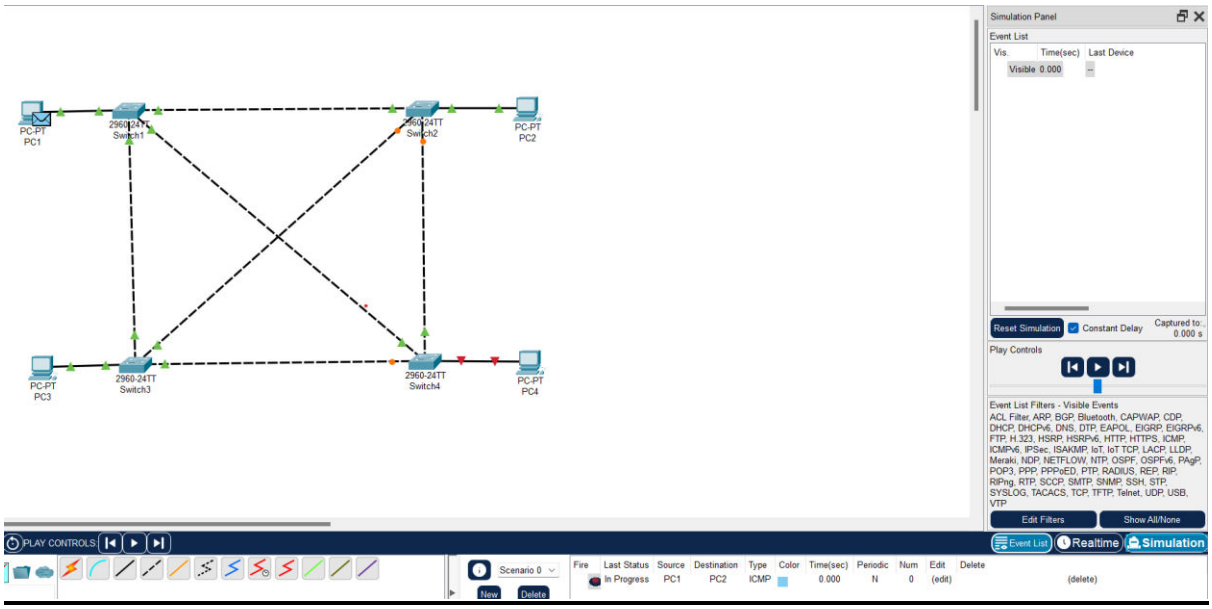
Mesh Topology (CASE 3: One PC is disabled When one PC is disabled (other than server and receiver)):

:

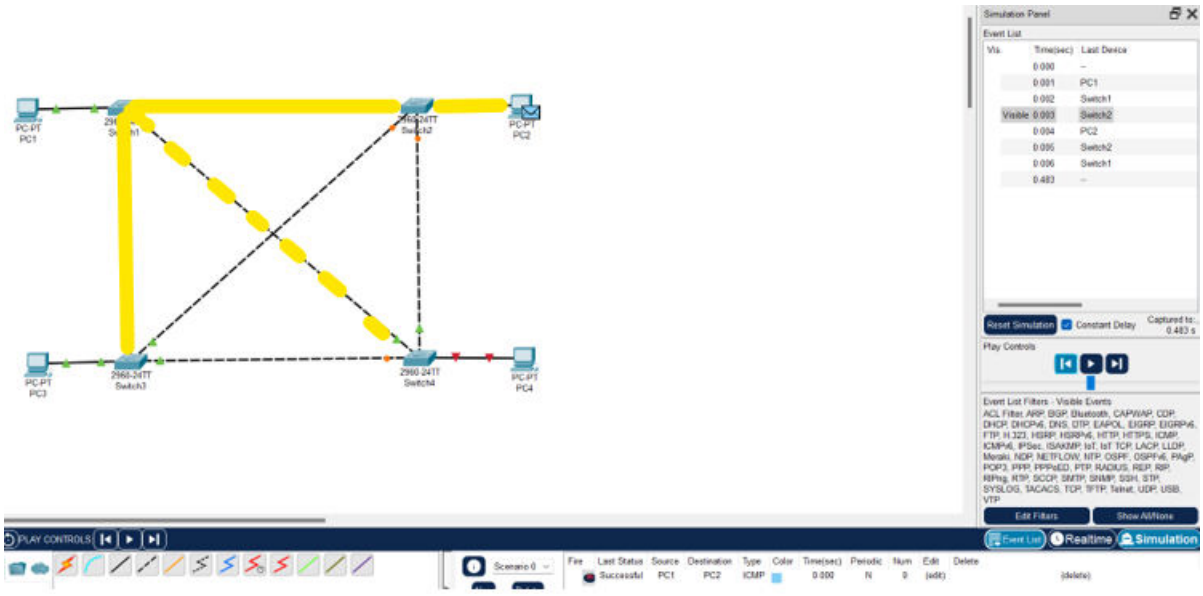


Here PC 4 is disabled.

Initial



Final



Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	Edit	Delete	
	Successful	PC1	PC2	ICMP		0.000	N	0	(edit)		(delete)

Using ping command to verify connectivity:

```
Cisco Packet Tracer PC Command Line 1.0
C:\>ping 192.168
Ping request could not find host 192.168. Please check the name and try again.
C:\>ping 192.168.0.4

Pinging 192.168.0.4 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.0.4:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

C:\>
```

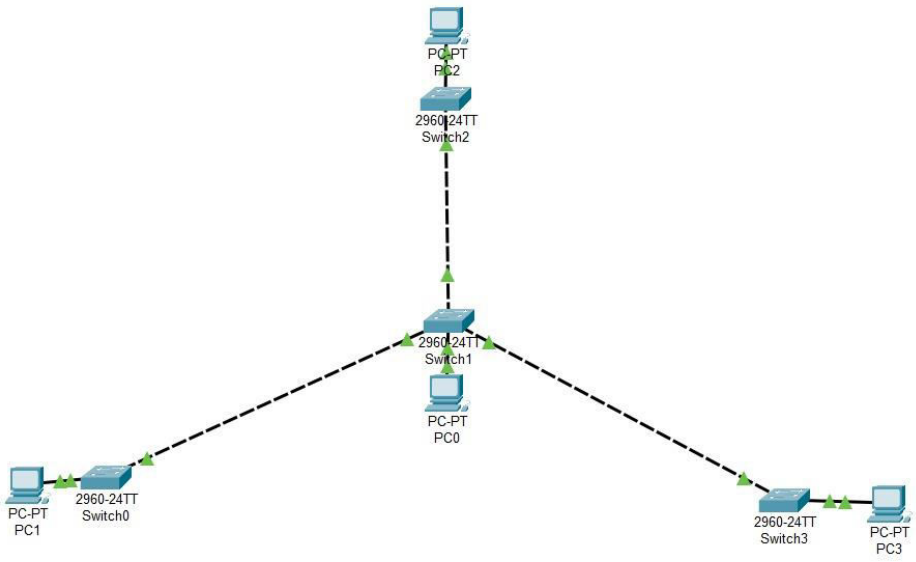
Topology 2

IP Addressing Table:

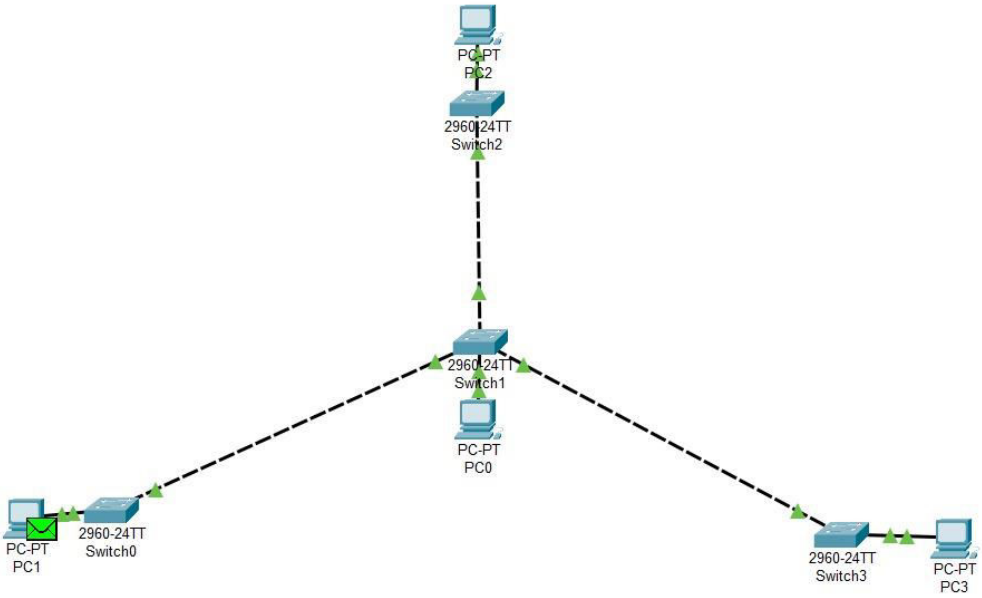
S.no	Devices	IP Address	Subnet Mask
1.	PC0	192.168.0.1	255.255.255.0
2.	PC1	192.168.0.2	255.255.255.0
3.	PC2	192.168.0.3	255.255.255.0
4.	PC3	192.168.0.4	255.255.255.0

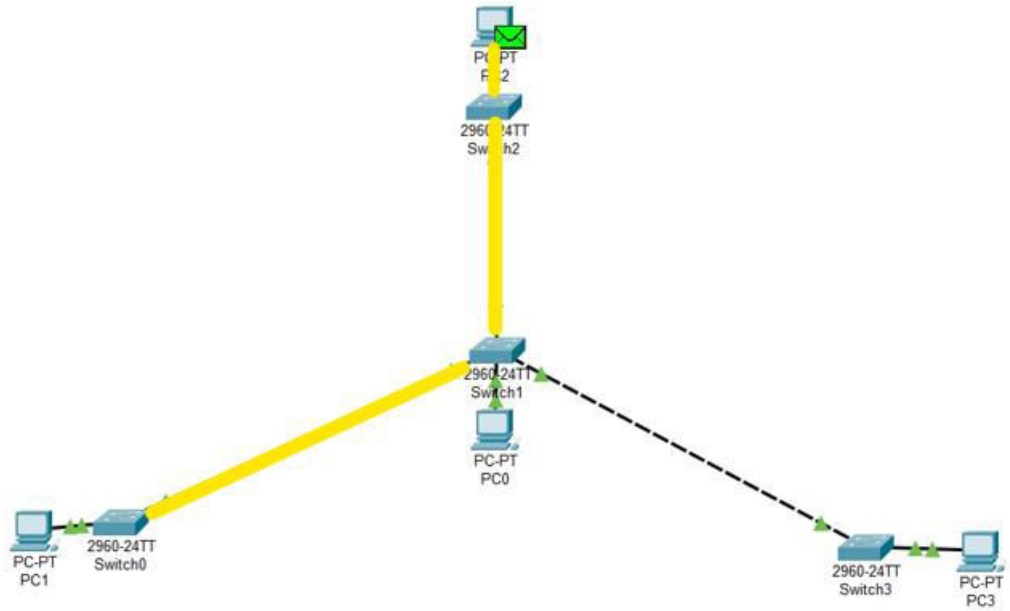
Star Topology (CASE 1: All cables are intact):

Initial:



Final:


PC-PT
PC2
2960-24TT
Switch2
2960-24TT
Switch1
PC-PT
PC0
PC-PT
PC1
2960-24TT
Switch0
2960-24TT
Switch3
PC-PT
PC3


PC-PT
PC0
PC-PT
PC1
2960-24TT
Switch0
2960-24TT
Switch3
PC-PT
PC3

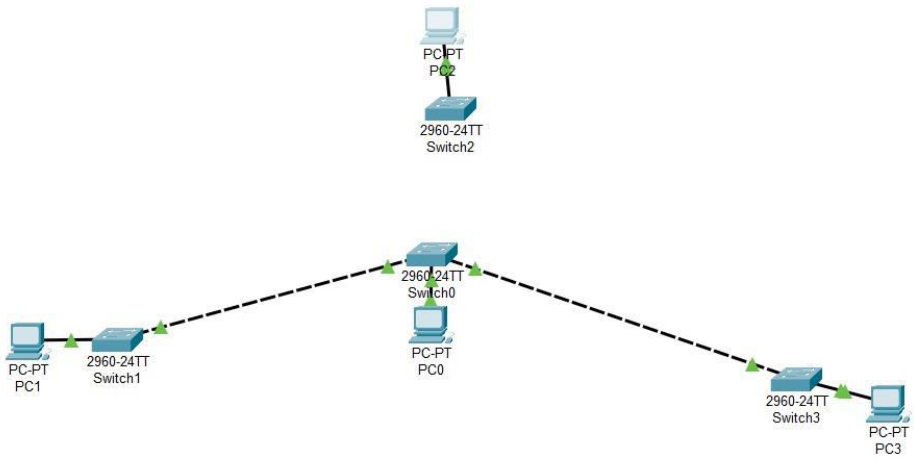
Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	Edit	Delete
	Successful	PC1	PC2	ICMP		0.000	N	0	(edit)	(delete)

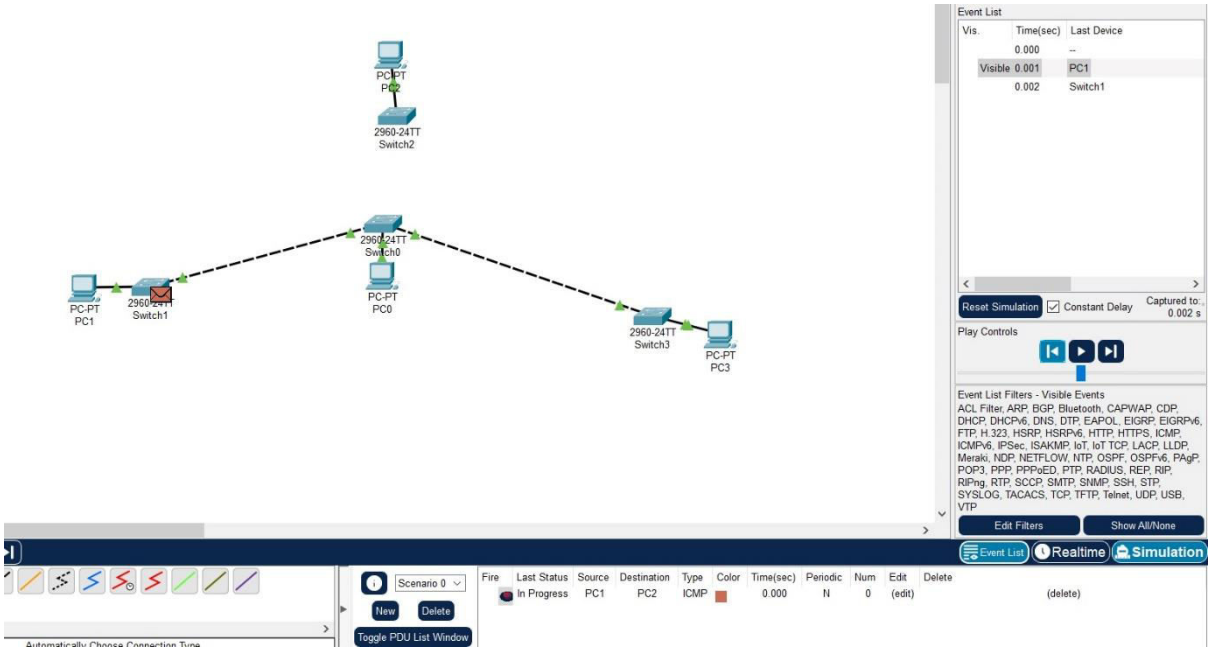
Using ping command to verify connectivity:

```
Cisco Packet Tracer PC Command Line 1.0
C:\>ping 192.168.0.1

Pinging 192.168.0.1 with 32 bytes of data:

Reply from 192.168.0.1: bytes=32 time=8ms TTL=128
Reply from 192.168.0.1: bytes=32 time=8ms TTL=128
Reply from 192.168.0.1: bytes=32 time=8ms TTL=128
Reply from 192.168.0.1: bytes=32 time=8ms TTL=128

Ping statistics for 192.168.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 8ms, Maximum = 8ms, Average = 8ms

C:\>
```

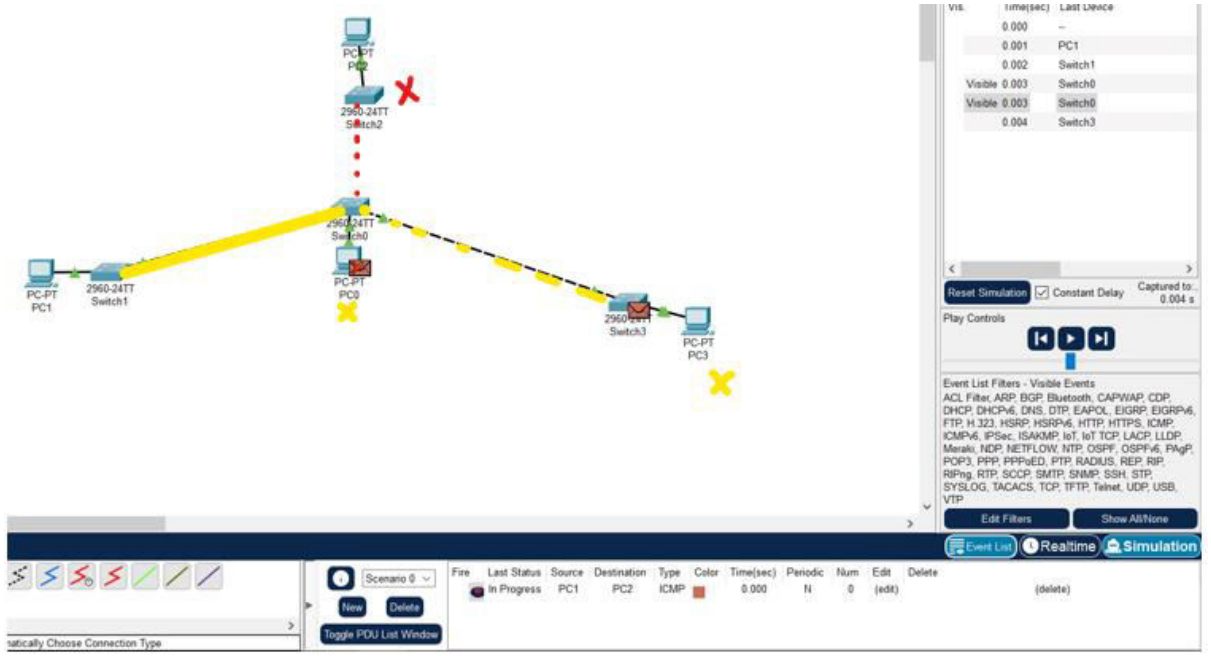
Star Topology (Case2: When one cable is damaged):

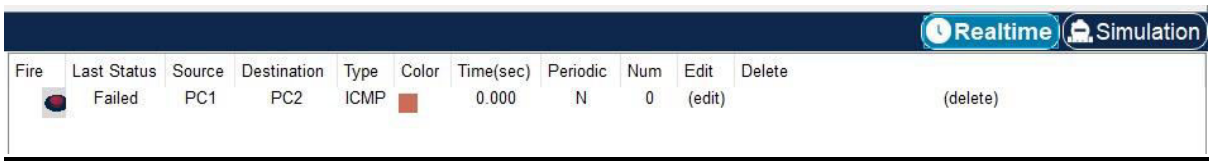
Initial:




PC-PT
PC2
2960-24TT
Switch2
2960-24TT
Switch0
PC-PT
PC1
2960-24TT
Switch1
PC-PT
PC0
2960-24TT
Switch3
PC-PT
PC3
Event List
Vis. Time(sec) Last Device
0.000 --
Visible 0.001 PC1
0.002 Switch1
Reset Simulation
Constant Delay
Captured to:
0.002 s
Play Controls
Event List Filters - Visible Events
ACL Filter, ARP, BGP, Bluetooth, CAPWAP, CDP, DHCP, DHCPv6, DNS, DTP, EAPOL, EIGRP, EIGRPv6, FTP, H.323, HSRP, HSRPv6, HTTP, HTTPS, ICMP, ICMPv6, IPSec, ISAKMP, IoT, IoT TCP, LACP, LLDP, Meraki, NDP, NETFLOW, NTP, OSPF, OSPFv6, PAgP, POP3, PPP, PPPoED, PTP, RADIUS, REP, RIP, RIPng, RTP, SCCP, SMTP, SNMP, SSH, STP, SYSLOG, TACACS, TCP, TFTP, Telnet, UDP, USB, VTP
Edit Filters
Show All/None
Event List
Realtime
Simulation
Scenario 0
New
Delete
Toggle PDU List Window
Fire Last Status Source Destination Type Color Time(sec) Periodic Num Edit Delete
In Progress PC1 PC2 ICMP 0.000 N 0 (edit) (delete)

Final:

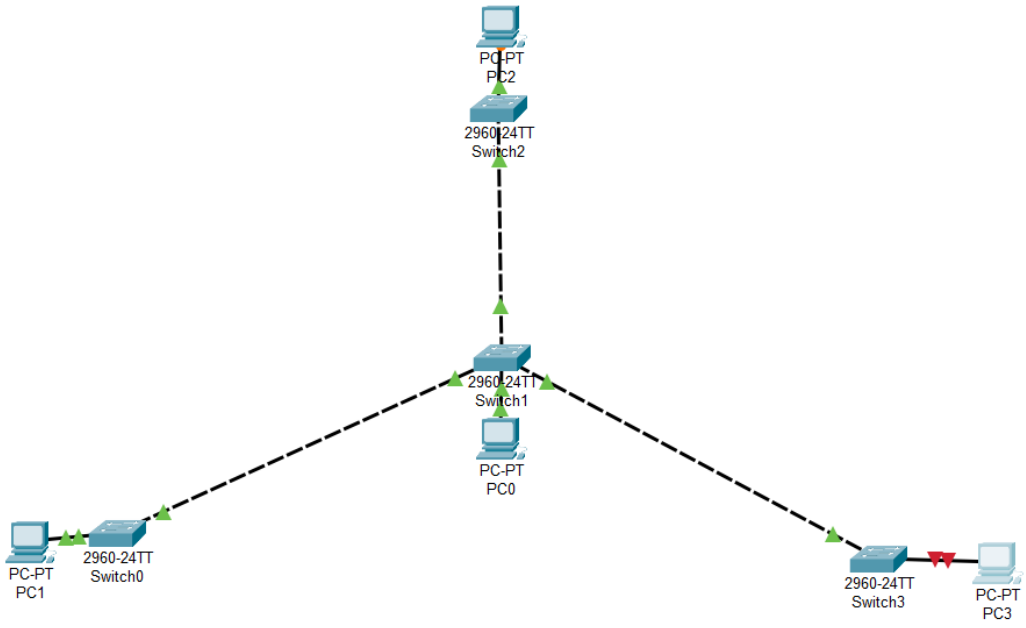

PC-PT
PC2
2960-24TT
Switch2
2960-24TT
Switch0
PC-PT
PC1
2960-24TT
Switch1
PC-PT
PC0
2960-24TT
Switch3
PC-PT
PC3
Vis. Time(sec) Last Device
0.000 --
0.001 PC1
0.002 Switch1
Visible 0.003 Switch0
Visible 0.003 Switch0
0.004 Switch3
Reset Simulation
Constant Delay
Captured to:
0.004 s
Play Controls
Event List Filters - Visible Events
ACL Filter, ARP, BGP, Bluetooth, CAPWAP, CDP, DHCP, DHCPv6, DNS, DTP, EAPOL, EIGRP, EIGRPv6, FTP, H.323, HSRP, HSRPv6, HTTP, HTTPS, ICMP, ICMPv6, IPSec, ISAKMP, IoT, IoT TCP, LACP, LLDP, Meraki, NDP, NETFLOW, NTP, OSPF, OSPFv6, PAgP, POP3, PPP, PPPoED, PTP, RADIUS, REP, RIP, RIPng, RTP, SCCP, SMTP, SNMP, SSH, STP, SYSLOG, TACACS, TCP, TFTP, Telnet, UDP, USB, VTP
Edit Filters
Show All/None
Event List
Realtime
Simulation
Scenario 0
New
Delete
Toggle PDU List Window
Fire Last Status Source Destination Type Color Time(sec) Periodic Num Edit Delete
In Progress PC1 PC2 ICMP 0.000 N 0 (edit) (delete)


Realtime
Simulation
Fire Last Status Source Destination Type Color Time(sec) Periodic Num Edit Delete
Failed PC1 PC2 ICMP 0.000 N 0 (edit) (delete)

Using ping command to verify connectivity:

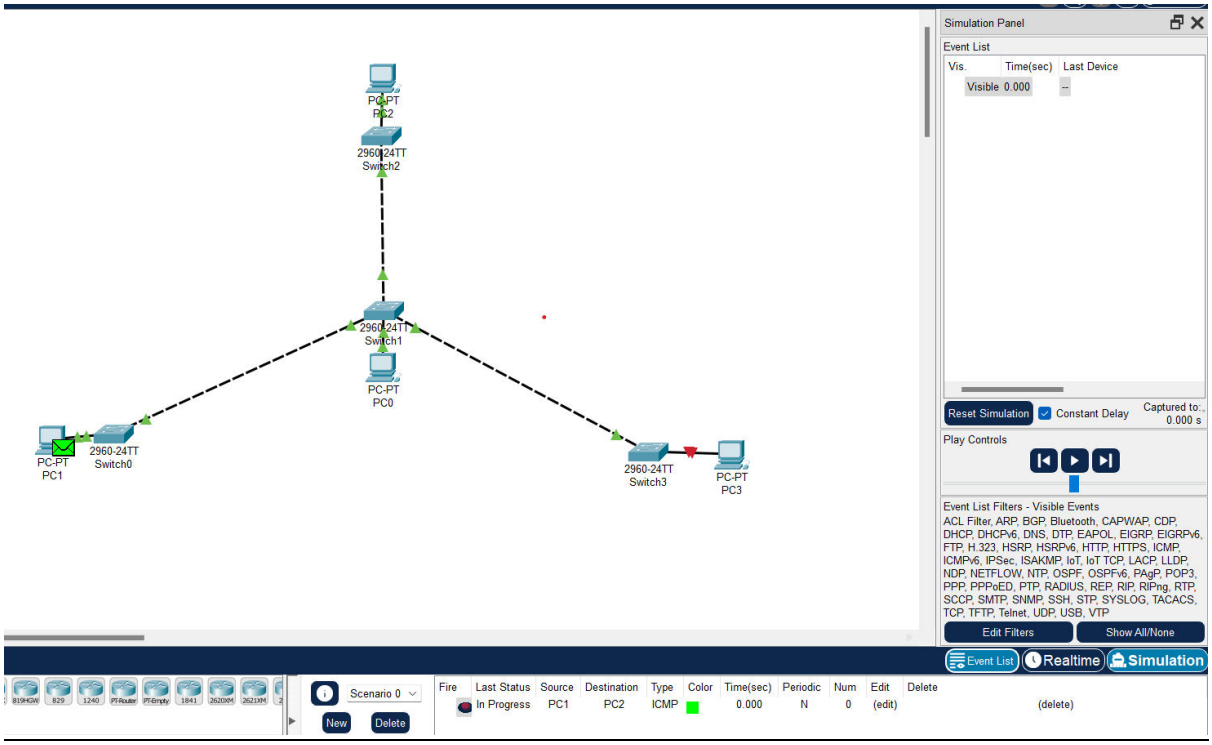
```
C:\>ping 192.168.0.2

Pinging 192.168.0.2 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.0.2:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),
```

Star Topology (Case3: When one PC is disabled (other than server and receiver)):

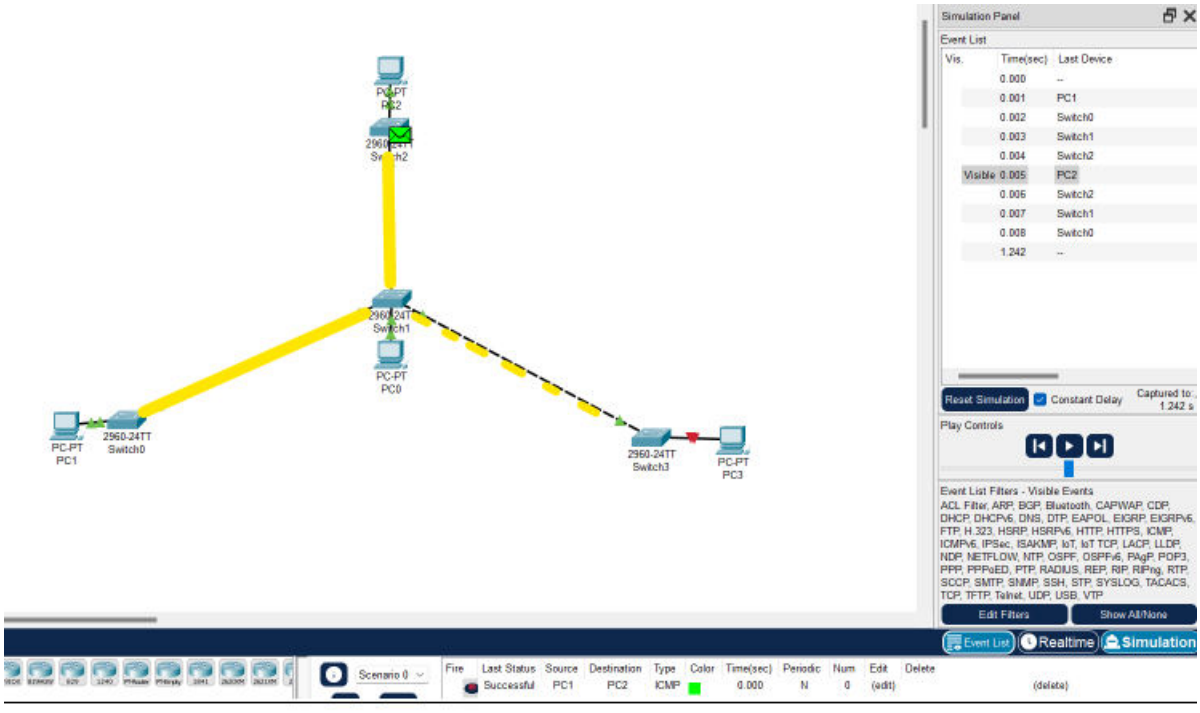


PC3 is disabled here.

Initial

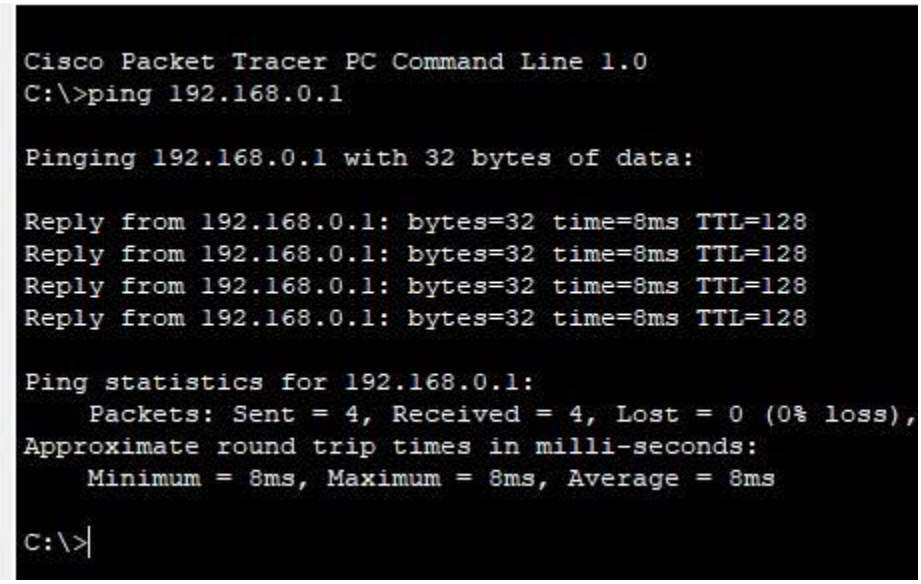


Final



Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	Edit	Delete
	Successful	PC1	PC2	ICMP		0.000	N	0	(edit)	(delete)

Using ping command to verify Connectivity:



Result:

Hence, the given topologies and different cases for each were implemented successfully using CISCO packet tracer.

Study of Socket Programming and Client – Server Model

Ex. No. 6
Date:

PROBLEM STATEMENT:

To study the client-server model using socket programming to enable communication between a client program and a server program over a network.

PROBLEM DESCRIPTION:

The task is to create a client-server application using socket programming to establish communication between a client program and a server program. The client program should be able to initiate a connection to the server, send data to the server, and receive a response from the server. The server program should listen for incoming connections, receive data from the client, process the data, and send a response back to the client. The implementation can be based on either TCP or UDP protocols, depending on the specific requirements of the application.

SOCKET PROGRAMMING:

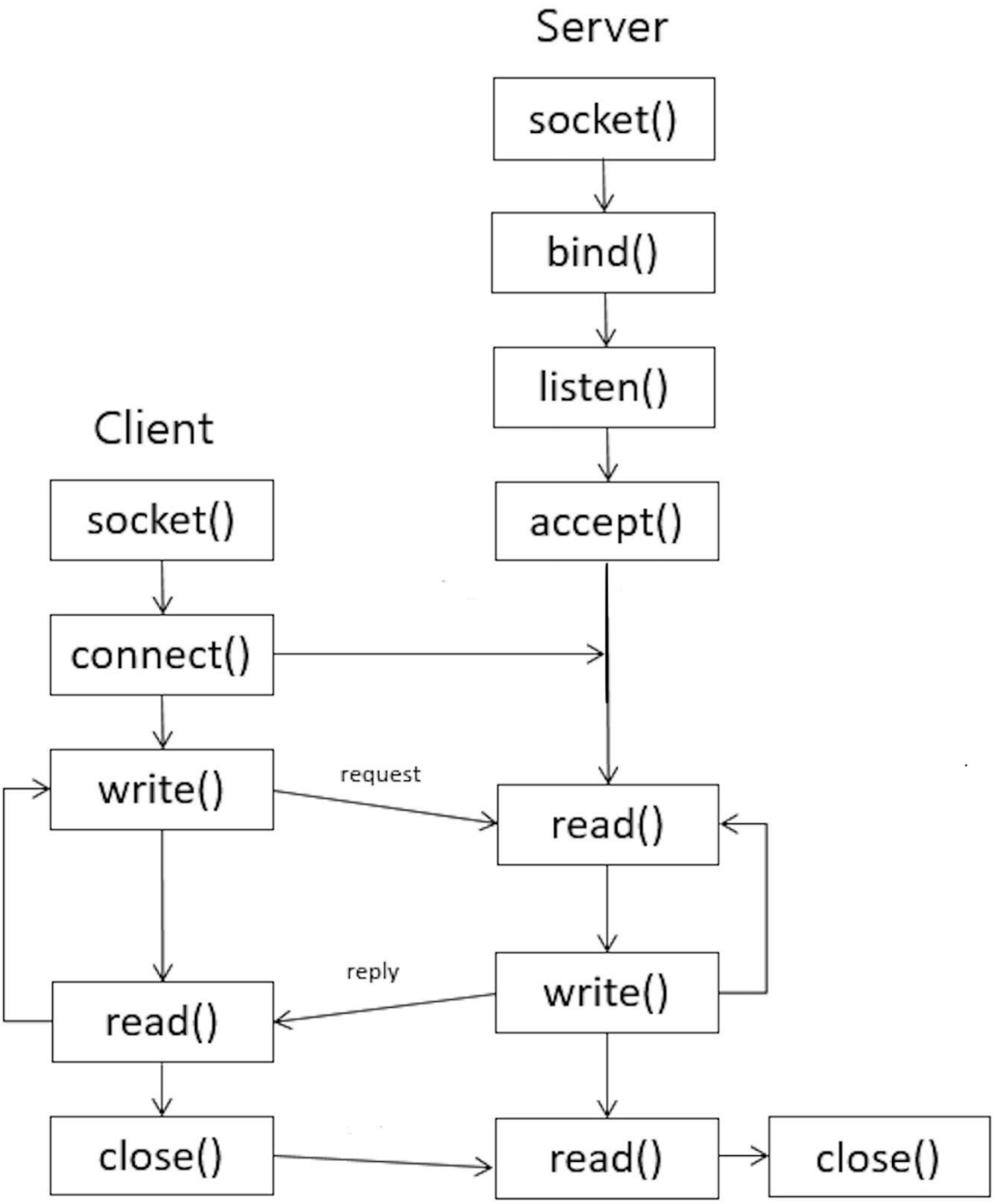
- A socket is a communications connection point (endpoint) that you can name and address in a network.

- Socket programming shows how to use socket APIs to establish communication links between remote and local processes. One socket (node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server.

- In socket programming, there are typically two roles: the client and the server.

Client: The client initiates a connection to a server. It can send requests or messages to the server and receive responses.

Server: The server listens for incoming connections from clients. It accepts connections, processes client requests, and sends back responses.



FLOW DIAGRAM FOR SERVER AND CLIENT MODEL OF SOCKET

FUNCTIONS USED IN SERVER/CLIENT MODEL:

- **Create a socket**:
  Both the client and server need to create their respective sockets. A socket is created using the socket() function, specifying the address family (such as IPv4 or IPv6) and the socket type (such as TCP or UDP).

- **Bind** (for server):
  In the server program, the socket needs to be bound to a specific address and port on the machine. This is done using the bind() function, which associates a socket with an address.

- **Listen** (for server):
  The server socket needs to listen for incoming connections. The listen() function is used to make the server socket ready to accept client connections.

- **Connect** (for client):
  The client program needs to establish a connection to the server. The connect() function is used to connect to the server's address and port.

- **Accept** (for server):
  When a client attempts to connect to the server, the server program uses the accept() function to accept the connection and create a new socket for communication with that client.

- **Send and receive data**:
  Once the connection is established, both the client and server can use the send() and receive() functions to send and receive data between them. The data can be in the form of bytes or strings.

- **Close the connection**:
  After the communication is complete, both the client and server should close their sockets using the close() function.

To implement client-server model using twisted python, here are some of the key functions and classes:

Server Side:

- **reactor.listenTCP(port_number, factory)**: Starts a TCP server listening on the specified ***port_number***. It takes a ***Factory*** instance, which is responsible for creating protocol instances for each client connection.

- **twisted.protocols.Protocol:** A base class that you can subclass to create your own protocol for handling server-side logic. It provides methods like ***connectionMade(), dataReceived(data),*** and ***connectionLost(reason)*** to handle different events in the communication process.

- **twisted.internet.protocol.Factory:** A base class for creating protocol instances. You can subclass it to define your own factory, which is responsible for creating instances of your protocol class.

- **reactor.run()**: Starts the Twisted event loop, allowing your server to run and handle incoming connections and data.

Client Side:

- **reactor.connectTCP(server_ip, port_number, protocol_instance):** Connects to a server with the specified ***server_ip*** and ***port_number*** using the given ***protocol_instance***. The protocol instance should be an instance of your custom protocol class, which extends ***twisted.protocols.Protocol.***

- **twisted.protocols.Protocol**: Similar to the server side, this base class is used to create your own protocol for handling client-side logic. You can implement methods like ***connectionMade(), dataReceived(data),*** and ***connectionLost(reason)*** to handle the corresponding events.

- **reactor.run()**: Starts the Twisted event loop, allowing your client to run and interact with the server.

Result:

Thus, the study of socket programming and client-server model have been done.

APPLICATIONS USING TCP SOCKETS LIKE

A) ECHO CLIENT/SERVER B) MULTIPLE CLIENT/SERVER C) CHAT D) FILE TRANSFER USING TWISTED PYTHON

Ex. No. 7

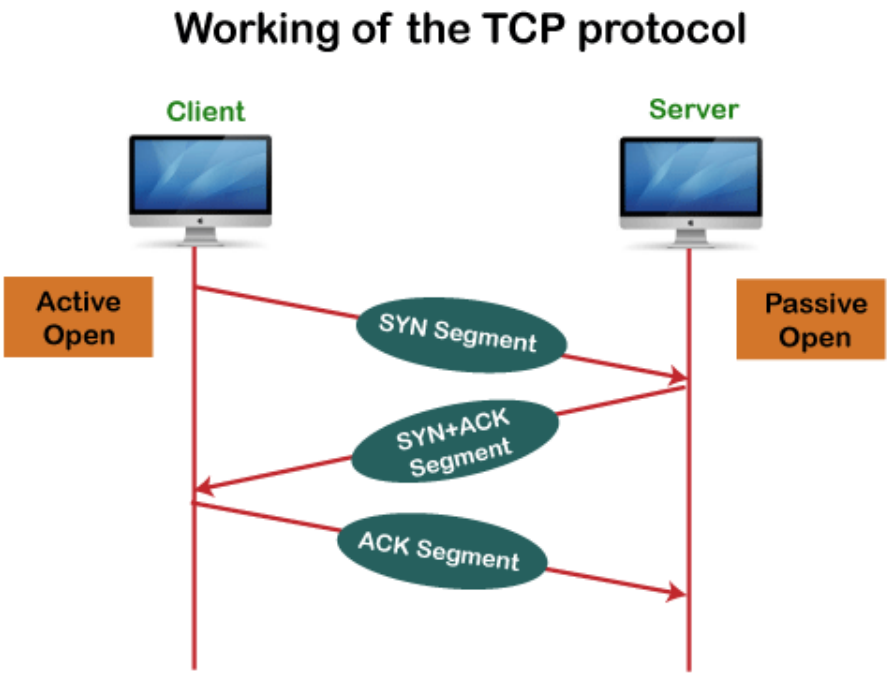
Date:

PROBLEM STATEMENT:

To implement TCP Echo Server and Echo Client for Single Server, Single Client and Single Server, Multiple Client, Chat Server and File Transfer Applications using Twisted Python.

PROBLEM DESCRIPTION:

TCP stands for Transmission Control Protocol. It is a transport layer protocol that facilitates the transmission of packets from source to destination. It is a connection-oriented protocol that means it establishes the connection prior to the communication that occurs between the computing devices in a network. This protocol is used with an IP protocol.



A) ECHO CLIENT/SERVER
Working:

For Single Echo Server and Echo Client

- Based on the IP Address and Port Number the connection is established between the Server and Client.
- Port Number should be same in Server and Client.
- Local Host should be specified in Client.
- Client will have its data echoed back to server.

B) MULTIPLE CLIENT/ SINGLE SERVER
For Single Echo Server and Multiple Echo Clients

- Run the server script first and then run the client script in separate terminals to observe the communication between multiple clients and the server.
- Each Client will have its data echoed back to the Server.

CODE:

echoServer.py
```
from twisted.internet import protocol,reactor

class echo(protocol.Protocol):

    def dataReceived(self, data):
        print("Message from Client -", data.decode())
        print("Client Connected!")
        ack_msg = f"{data.decode()}"
        ack = "ACK[" + ack_msg + "]"
        print("Acknoledgement Sent!")
        self.transport.write(ack.encode())

class echofactory(protocol.Factory):

    def buildProtocol(self, addr):
        return echo()

reactor.listenTCP(8000,echofactory())
reactor.run()
```

echoClient.py
```
from twisted.internet import reactor, protocol

class EchoClient(protocol.Protocol):

    def connectionMade(self):
        msg = input("Enter the message to Server - ")
        self.transport.write(msg.encode())

    def dataReceived(self, data):
        print ("Acknoledgement from Server -", data.decode())
        self.transport.loseConnection()

class EchoFactory(protocol.ClientFactory):

    def buildProtocol(self, addr):
```

```
        return EchoClient()


    def clientConnectionFailed(self, connector, reason):
        print ("Connection failed.")
        reactor.stop()

    def clientConnectionLost(self, connector, reason):
        print ("Connection lost.")
        reactor.stop()

reactor.connectTCP("localhost", 8000, EchoFactory())
reactor.run()
```

CODE EXPLANATION:

Echo Server:

CLASS/FUNCTION	USE
Class Echo	This handles the communication with Client.
dataReceived()	Echoes the received data back to the Client.
EchoFactory Class	This creates instances of the Echo Protocol for each Client connection.
BuildProtocol()	It is called when a connection to the server is established.

Echo Client:

CLASS/FUNCTION	USE
Class EchoClient	This handles the communication with the Server.
connectionMade()	It will be called when a connection to the server is established.
DataReceived()	It will be called when a data is received from the Server.
EchoClientFactory class	It creates instances of EchoClient().
clientConnectionFailed()	It will be called when the connection to the Server fails.
clientConnectionLost()	It will be called when the connection to the Server lost.

OUTPUT:

<u>Single Server – Single Client</u>

Client:

```
d/.vscode/extensions/ms-python.python-2023.6.0/pythonFiles/lib/python/debugpy/ad
ED/MY\ FILES/SEM\ 4/Network\ Lab/code_arun/tcp/echoclient.py
Enter the message to Server - Hi Server, Want to Connect.
Acknoledgement from Server - ACK[Hi Server, Want to Connect.]
Connection lost.
```

Server:

```
d/.vscode/extensions/ms-python.python-2023.6.0/pythonFiles/lib/python/debugpy/a
ED/MY\ FILES/SEM\ 4/Network\ Lab/code_arun/tcp/echoserver.py
Message from Client - Hi Server, Want to Connect.
Client Connected!
Acknoledgement Sent!
```

Single Server – Multiple Client

Client 1:

```
apter/../../debugpy/launcher 51611 -- /home/ahamed/Documents/echoclient.py
Enter the message to Server - Hi Server, Want to connect.
Acknoledgement from Server - ACK[Hi Server, Want to connect.]
Connection lost.
```

Client 2:

```
nFiles/lib/python/debugpy/adapter/../../debugpy/launcher 33403
Enter the message to Server - Hi Srever, Connect with me.
Acknoledgement from Server - ACK[Hi Srever, Connect with me.]
Connection lost.
```

Client 3:

```
nFiles/lib/python/debugpy/adapter/../../debugpy/launcher 42197 -- /hom
Enter the message to Server - Hi Server, Please connect with me.
Acknoledgement from Server - ACK[Hi Server, Please connect with me.]
Connection lost.
```

Server:

```
Message from Client - Hi Server, Want to connect.
Client Connected!
Acknoledgement Sent!
Message from Client - Hi Srever, Connect with me.
Client Connected!
Acknoledgement Sent!
Message from Client - Hi Server, Please connect with me.
Client Connected!
Acknoledgement Sent!
```

C) CHAT

PROBLEM STATEMENT:

To design and implement a TCP-based chat room application that allows multiple clients to connect to a central server and engage in real time communication with one another. The chat room should provide a platform for users to exchange text messages and foster interactive sessions.

PROBLEM DESCRIPTION:

The objective is to develop a TCP-based chat room consisting of a chat server and multiple chat clients. The goal is to allow users to connect to the server using a client application and communicate with each other in real-time through text-based messages.

For every client joining the server should be displayed and notified in the chatroom with their names, for the clients who are already available.

All the messages sent by each client is listed out with their name.

Every message sent by each client is displayed to all the others present in the chat room.
'/quit' when entered can make the client lose connection with the chat server.

Code:

```
from twisted.internet import reactor, protocol
from twisted.protocols.basic import LineOnlyReceiver


class ChatProtocol(LineOnlyReceiver):
    def __init__(self, factory):
        self.factory = factory
        self.name = None
        self.state = "GETNAME"
        self.client = None


    def connectionMade(self):
        self.sendLine("What's your name?".encode())


    def connectionLost(self, reason):
        if self.name in self.factory.users:
            del self.factory.users[self.name]
            self.broadcastMessage(f"{self.name} has left the chat room.")
```

```
    def lineReceived(self, line):
        if self.state == "GETNAME":
            self.handle_GETNAME(line.decode())
        else:
            self.handle_CHAT(line.decode())

    def handle_GETNAME(self, name):
        if name in self.factory.users:
            self.sendLine("Name already taken, please choose another name.".encode())
            return
        self.sendLine(f"Welcome, {name}!".encode())
        self.broadcastMessage(f"{name} has joined the chat room.")
        self.name = name
        self.factory.users[name] = self
        self.state = "CHAT"

    def handle_CHAT(self, message):
        if message.lower() == "/quit":
            self.transport.loseConnection()
        else:
            message = f"<{self.name}> {message}"
            self.broadcastMessage(message)

    def broadcastMessage(self, message):
        for name, protocol in self.factory.users.items():
            if protocol != self:
                protocol.sendLine(message.encode())

    def connectionMade(self):
        self.sendLine("Connected to the chat server. Type '/quit' to exit.".encode())
        self.factory.clients.append(self)

    def connectionLost(self, reason):
        self.factory.clients.remove(self)

class ChatFactory(protocol.Factory):
    def __init__(self):
        self.users = {}
        self.clients = []
```

```
    def buildProtocol(self, addr):
        return ChatProtocol(self)

    def broadcastMessage(self, message):
        for client in self.clients:
            client.sendLine(message.encode())

if __name__ == "__main__":
    reactor.listenTCP(9000, ChatFactory())
    print("Chat server started. Listening on port 9000...")
    reactor.run()
```

command to run in the terminal is telnet ip address port number . create multiple terminals or use different machines.

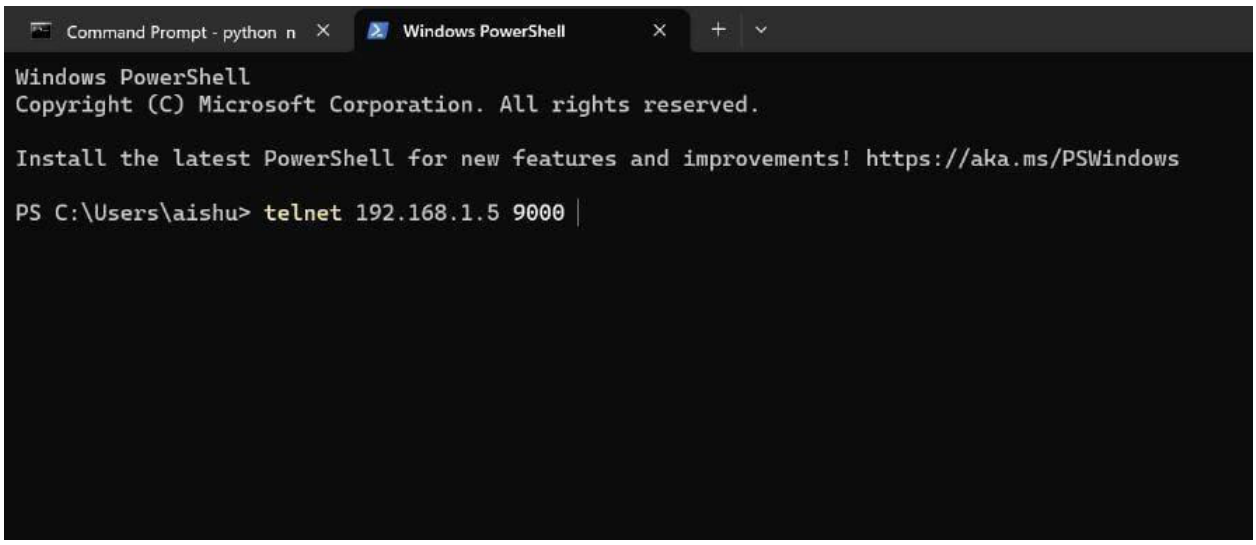
To execute the code:

1. Run the program in a code editor or in the terminal.
2. The server will run the program i.e the code provided above.
3. For every (including the server) user that wants to connect to this server, he/she has to type:
   'telnet' <IP address of the host node> <port address >
   (Port address:- which is in the program in his/her node).
   Example:
   telnet 10.4.7.32 8000 (10.4.7.32 - IP address of the system running the server program).
   To find ip address of the system use the following command in terminal:
   'hostname -I'
   [in their system].
4. Users can continue chatting and joining the server until the server connection is closed. Closing the server is done by closing the respective terminal that is running the server or by giving /quit.

SAMPLE INPUT/OUTPUT:

To start the server:

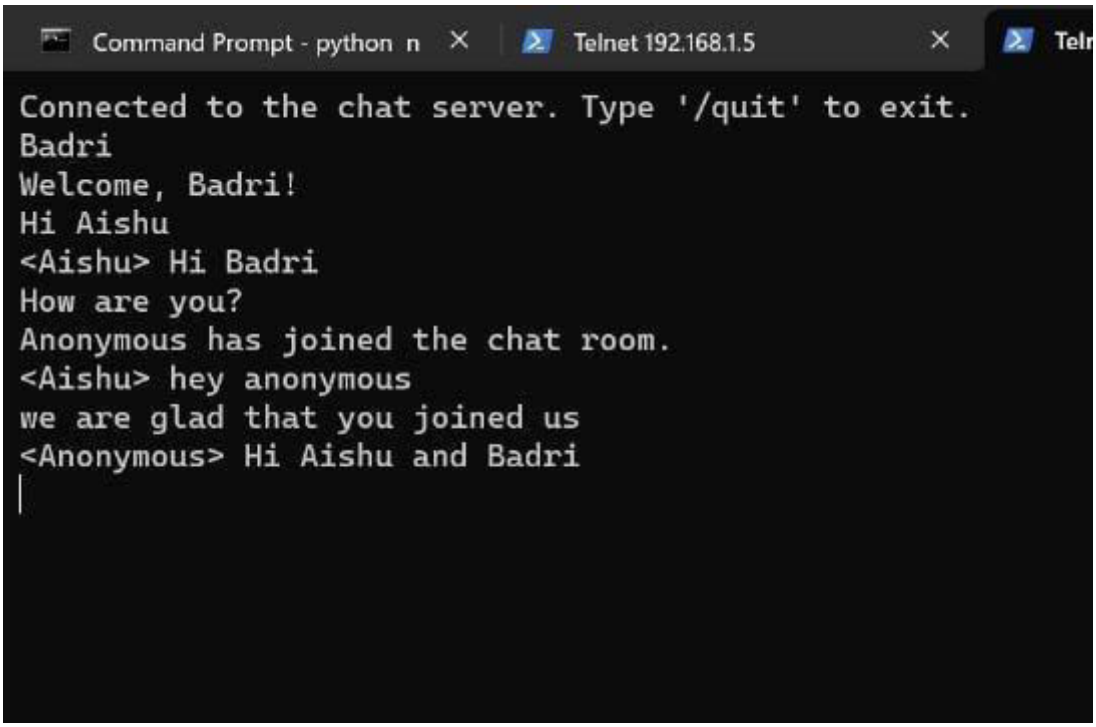
To join the chatroom for each client:



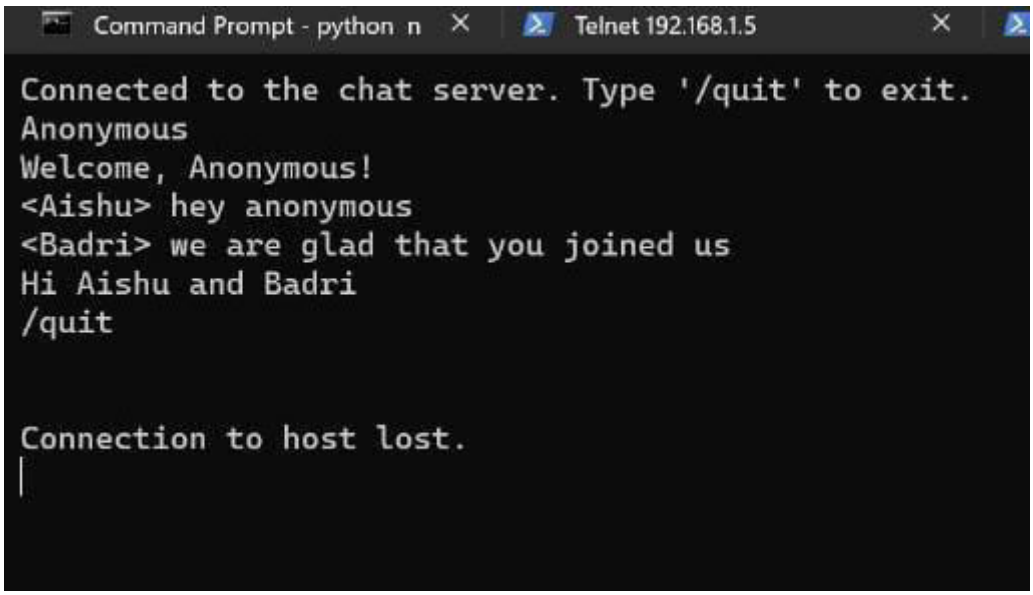
Once the user joins the chatroom by default enter username and will be welcomed and the user can start chatting.

When the any new client joins the server the all the present clients will get notified about the new client.

Second client joins:



The third client joins and loses connection.



D) FILE TRANSFER

PROBLEM STATEMENT:

To create an application (File Transfer) using TCP socket using Twisted Python.

PROBLEM DESCRIPTION:

This problem describes the implementation of File Transfer Protocol (FTP) using Python. Here, the TCP Socket has been used for this, i.e. a connection-oriented socket. FTP (File Transfer Protocol) is a network protocol for transmitting files between computers over Transmission Control Protocol/Internet Protocol (TCP/IP) connections.

To implement a file transfer mechanism using Twisted, which is an event-driven networking engine framework for Python, there are 2 key sides: **Client-side** and **Server-side.**

Functions used in Server Side:

The FileTransferProtocol class is a subclass of protocol.Protocol, which represents the behavior of the server in response to events. In this case, it handles the connection being made and data being received from the client.

the "connectionMade" method is called. In this implementation, it simply prints "Client connected."

The "dataReceived" method is called when the server receives data from the client. The server checks if the received data is equal to b"SEND", indicating that the client wants to initiate a file transfer. If that's the case, the server responds with b"READY" to acknowledge that it is ready to receive the file.

The FileTransferFactory class is a subclass of protocol.Factory that is responsible for creating instances of the FileTransferProtocol class when a connection is made. It has a buildProtocol method that creates a new protocol instance for each client connection.

The script checks if it is being run directly (not imported as a module) by using if __name__ == "__main__". If so, it creates a TCP server using reactor.listenTCP() and passes it the port number (in this case, 7000) and an instance of FileTransferFactory. It also prints "Server started."

The reactor.run() call starts the Twisted event loop, which allows the server to handle client connections and data events.

Functions used in Client Side:

The FileTransferClientProtocol class is defined, which inherits from protocol.Protocol. This class represents the client-side protocol for file transfer.
The “connectionMade” method is overridden and called when a connection is established with the server. It opens a file called "myfile.txt" in binary mode and reads its contents into the self.fileData attribute.
The “dataReceived” method is overridden and called whenever data is received from the server.

The client sends the file data by callingself.transport.write(self.fileData).

Theclient prints a message indicating that the file transfer is complete and closes the connection by calling self.transport.loseConnection().

The FileTransferClientFactory class is defined, which inherits from protocol.ClientFactory. This class is responsible for creating instances of FileTransferClientProtocol whenever a connection is established.

the Twisted reactor is started by calling reactor.connectTCP to establish a TCP connection with the server at the address "localhost" on port 7000.

Finally, reactor.run() is called to start the event loop and begin the network communication.

CODE:

SERVER:

```
from twisted.internet import reactor, protocol
import os
import time

class FileTransferProtocol(protocol.Protocol):
    def connectionMade(self):
        print("Client connected.")

    def dataReceived(self, data):
        if data == b"SEND":
            self.transport.write(b"READY")
            self.transferFile = True
```

```
            self.startTime = time.time()  # Start measuring time
        elif self.transferFile:
            with open("received_file", "wb") as f:
                f.write(data)
            self.transport.write(b"RECEIVED")
            self.transferFile = False
            endTime = time.time()  # Stop measuring time
            rtt = endTime - self.startTime
            print("Round trip time:", rtt)
        else:
            self.transport.write(b"ERROR")

class FileTransferFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return FileTransferProtocol()

if __name__ == "__main__":
    reactor.listenTCP(7000, FileTransferFactory())
    print("Server started.")
    reactor.run()
```

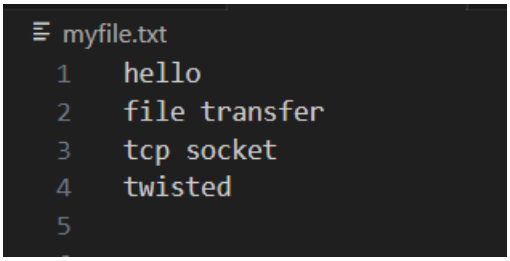
CLIENT:

```
from twisted.internet import reactor, protocol
import os
import time

class FileTransferClientProtocol(protocol.Protocol):
    def connectionMade(self):
        self.startTime = time.time()  # Start measuring time
        try:
            f = open("myfile.txt", "rb")
            self.fileData = f.read()
            f.close()
            if len(self.fileData) == 0:
                print("File is empty.")
                self.transport.loseConnection()
            else:
                self.transport.write(b"SEND")
        except FileNotFoundError:
            print("File not found.")
            self.transport.loseConnection()

    def dataReceived(self, data):
        if data == b"READY":
            if self.fileData:
                self.transport.write(self.fileData)
            else:
                self.transport.loseConnection()
        elif data == b"RECEIVED":
            endTime = time.time()  # Stop measuring time
            rtt = endTime - self.startTime
```

```
            print("Round trip time:", rtt)
            print("File transfer complete.")
            self.transport.loseConnection()
        else:
            print("Error:", data.decode())
            self.transport.loseConnection()

class FileTransferClientFactory(protocol.ClientFactory):
    protocol = FileTransferClientProtocol

if __name__ == "__main__":
    reactor.connectTCP("localhost", 7000, FileTransferClientFactory())
    reactor.run()
```

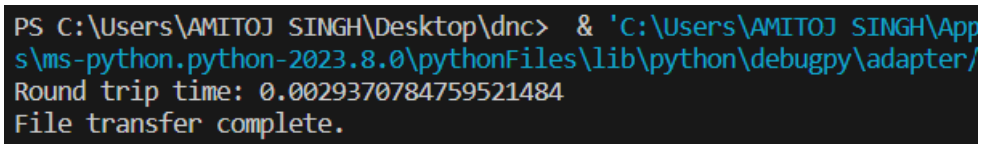
SAMPLE INPUT/OUTPUT:

FILE CONTENT(myfile.txt)



SERVER SIDE:

CLIENT SIDE:



RECEIVED FILE:

WHEN FILE IS EMPTY:

SERVER SIDE:

CLIENT SIDE:

WHEN FILE NOT FOUND:

SERVER SIDE:

CLIENT SIDE:

```
PS C:\Users\AMITOJ SINGH\Desktop\dnc>  & 'C:\Users\AMI
s\ms-python.python-2023.8.0\pythonFiles\lib\python\debu
File not found.
```

RESULT:

The TCP Echo Server and Echo Client for Single Server, Single Client and Single Server, Multiple Client, Chat Server and File Transfer applications have been implemented and tested successfully using Twisted Python.

APPLICATIONS USING UDP SOCKETS LIKE

A) ECHO CLIENT/SERVER B) MULTIPLE CLIENT/SERVER C) CHAT D) FILE TRANSFER USING TWISTED PYTHON

Ex. No. 8
Date:

PROBLEM STATEMENT:

To implement UDP Echo Server and Echo Client for Single Server, Single Client and Single Server, Multiple Client, Chat Server and File Transfer Applications using Twisted Python.

PROBLEM DESCRIPTION:

User Datagram Protocol (UDP) is a simple transport-layer protocol. The application writes a message to a UDP socket, which is then encapsulated in a UDP datagram, which is further encapsulated in an IP datagram, which is sent to the destination.

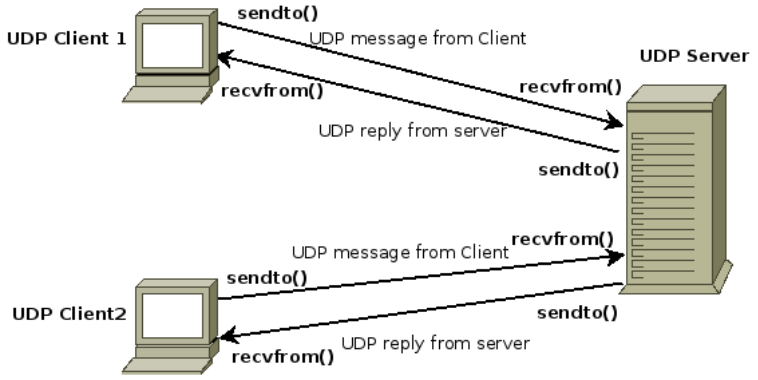
There is no guarantee that a UDP will reach the destination, that the order of the datagrams will be preserved across the network or that datagrams arrive only once.

The problem of UDP is its lack of reliability: if a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not automatically retransmitted.

Each UDP datagram is characterized by a length. The length of a datagram is passed to the receiving application along with the data.

No connection is established between the client and the server and, for this reason, we say that UDP provides a *connection-less service*.

Working:



A) ECHO CLIENT/SERVER
Working:

For Single Echo Server and Echo Client

- Based on the IP Address and Port Number the connection is established between the Server and Client.
- Port Number should be same in Server and Client.
- Local Host should be specified in Client.
- Client will have its data echoed back to server.

B) MULTIPLE CLIENT/ SINGLE SERVER
For Single Echo Server and Multiple Echo Clients

- Run the server script first and then run the client script in separate terminals to observe the communication between multiple clients and the server.
- Each Client will have its data echoed back to the Server.

CODE:

 # udp server

```
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor

class EchoUDP(DatagramProtocol):
    def datagramReceived(self, datagram, address):
        print("Datagram: ",datagram.decode())
        self.transport.write(datagram, address)

def main():
    reactor.listenUDP(8000, EchoUDP())
    print("Started to listen")
    reactor.run()

if __name__ == '__main__':
    main()
```

 # udp client

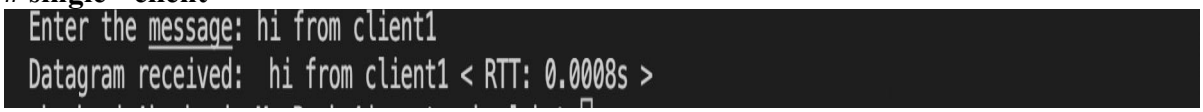
```
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor
import time

class EchoClientDatagramProtocol(DatagramProtocol):
    a = input("Enter the message: ")
    strings = [f"{a}".encode()]

    def startProtocol(self):
        self.transport.connect('127.0.0.1', 8000)
        self.sendDatagram()
```

```
    def sendDatagram(self):
        if len(self.strings):
            datagram = self.strings.pop(0)
            self.timestamp = time.time()
            self.transport.write(datagram)
        else:
            reactor.stop()

    def datagramReceived(self, datagram, host):
        rtt = time.time() - self.timestamp
        print('Datagram received: ', datagram.decode(), f"< RTT: {rtt:.4f}s >")
        self.sendDatagram()

def main():
    protocol = EchoClientDatagramProtocol()
    t = reactor.listenUDP(0, protocol)
    reactor.run()

if __name__ == '__main__':
    main()
```

Output :

 # server

 # single - client



For multi-client server open multiple terminals and run client file in it .

 # server

```
Started to listen
Datagram:  hi from client1
Datagram:  hello from client 2
Datagram:  heyyy from client3
```

#client 1

```
Enter the message: hi from client1
Datagram received:  hi from client1 < RTT: 0.0008s >
```

#client 2

```
Enter the message: hello from client 2
Datagram received:  hello from client 2 < RTT: 0.0050s >
```

#client 3

```
Enter the message: heyyy from client3
Datagram received:  heyyy from client3 < RTT: 0.0005s >
```

All the messages sent by the client(s) will be reflected back in the server (simple echo client/server).

C) CHAT

PROBLEM STATEMENT:

To design and implement a chat server using UDP (User Datagram Protocol) that allows multiple clients to communicate with each other in real-time.

PROBLEM DESCRIPTION:

The goal is to design and implement a chat server using UDP to facilitate real-time communication between multiple clients. UDP is a connectionless protocol that provides fast and lightweight data transmission but does not guarantee reliable delivery or ordered messages. Despite these limitations, we aim to develop a chat server that can handle packet loss, reordering, and concurrent client connections.

The chat server will allow clients to connect using their respective IP addresses and port numbers. Once connected, clients can send messages to the server, which will then relay the messages to the intended recipients. The server should maintain a list of active clients and their connection details.

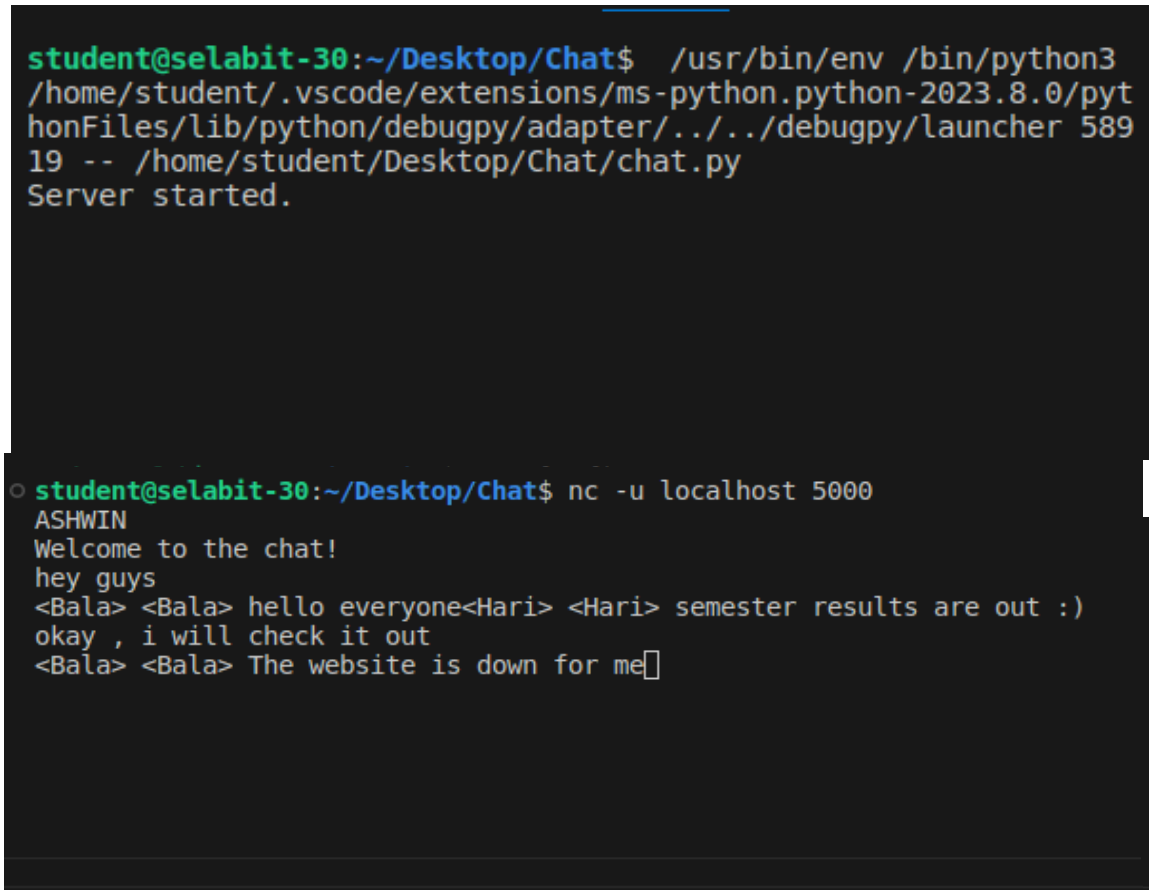
To ensure message delivery, the server should handle potential packet loss and reordering that may occur during transmission. This may involve implementing error detection and retransmission mechanisms or utilizing acknowledgments from clients. The server should also guarantee that messages are delivered to clients in the same order they were received, even if they arrive out of order due to the nature of UDP.

Overall, our aim is to build a reliable and efficient chat server using UDP that enables real-time communication among multiple clients while accounting for the limitations of the UDP protocol.

CODE:

```
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor

class ChatServer(DatagramProtocol):
    def __init__(self):
        self.users = {} # maps user addresses to usernames

    def datagramReceived(self, data, addr):
        message = data.decode('utf-8').strip()
        if addr in self.users:
            # if the user is already registered, broadcast the message to all other users
            username = self.users[addr]
            message = f"<{username}> {message}"
            for user_addr in self.users:
                if user_addr != addr:
                    self.transport.write(message.encode('utf-8'), user_addr)
        else:
            # if the user is not registered, use the first message as their username
            self.users[addr] = message.split()[0]
            self.transport.write(b"Welcome to the chat!\n", addr)

if __name__ == "__main__":
    reactor.listenUDP(5000, ChatServer())
    print("Server started.")
    reactor.run()
```

SAMPLE INPUT/OUTPUT:



D) FILE TRANSFER

PROBLEM STATEMENT:

To implement file transfer in UDP using twisted python.

PROBLEM DESCRIPTION:
UDP is a connectionless, unreliable protocol. UDP (User Datagram Protocol) file transfer is a method of transferring files between a client and a server using UDP as the underlying transport protocol. In a UDP file transfer, the client breaks the file into smaller chunks, known as packets, and sends them to the server over the network. The server receives these packets and reconstructs the original file.

What if transferring file gets corrupted?

- In a UDP file transfer, if the transferred file gets corrupted, it will not be automatically detected or corrected by the UDP protocol itself as it is unreliable. UDP does not provide built-in mechanisms for error detection, error correction, or retransmission of lost or corrupted packets. If a packet is lost or corrupted during transmission, UDP does not attempt to recover or retransmit it.
- Therefore, if a file gets corrupted during a UDP transfer, the receiver will not be aware of the corruption unless additional measures are implemented. It is the responsibility of the application layer to handle any error detection or correction.

CODE:

SERVER:

```
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor
import pickle
import time
import os

class EchoUDP(DatagramProtocol):
    def datagramReceived(self, datagram, address):
        receive_time = time.time()                                  # Get the receive timestamp
        file_name, file_data, send_time = pickle.loads(datagram)
        print(f'{file_name} received!')

        rtt = receive_time - send_time                              # Calculate RTT
        print(f'RTT: {rtt} seconds')

        file_path = file_name.rstrip('.txt') + 'Server.txt'
        with open(file_path, 'wb') as file:
            file.write(b'file_data')
        file_size = os.path.getsize(file_path)                      #File size
        print(f'{file_name} saved. File size: {file_size} bytes')

def main():
```

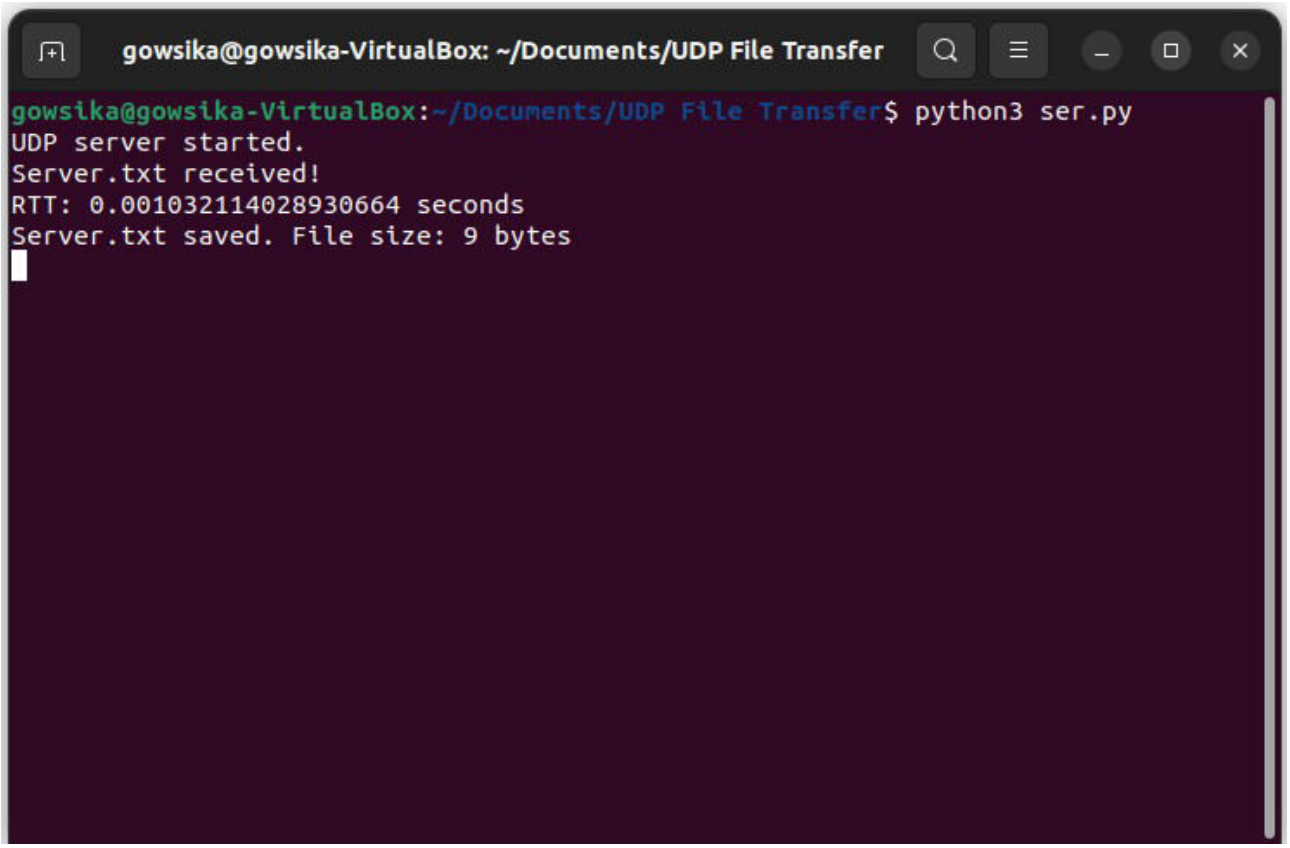
```
    reactor.listenUDP(1234, EchoUDP())
    print("UDP server started.")
    reactor.run()

if __name__ == '__main__':
    main()
```

CLIENT:

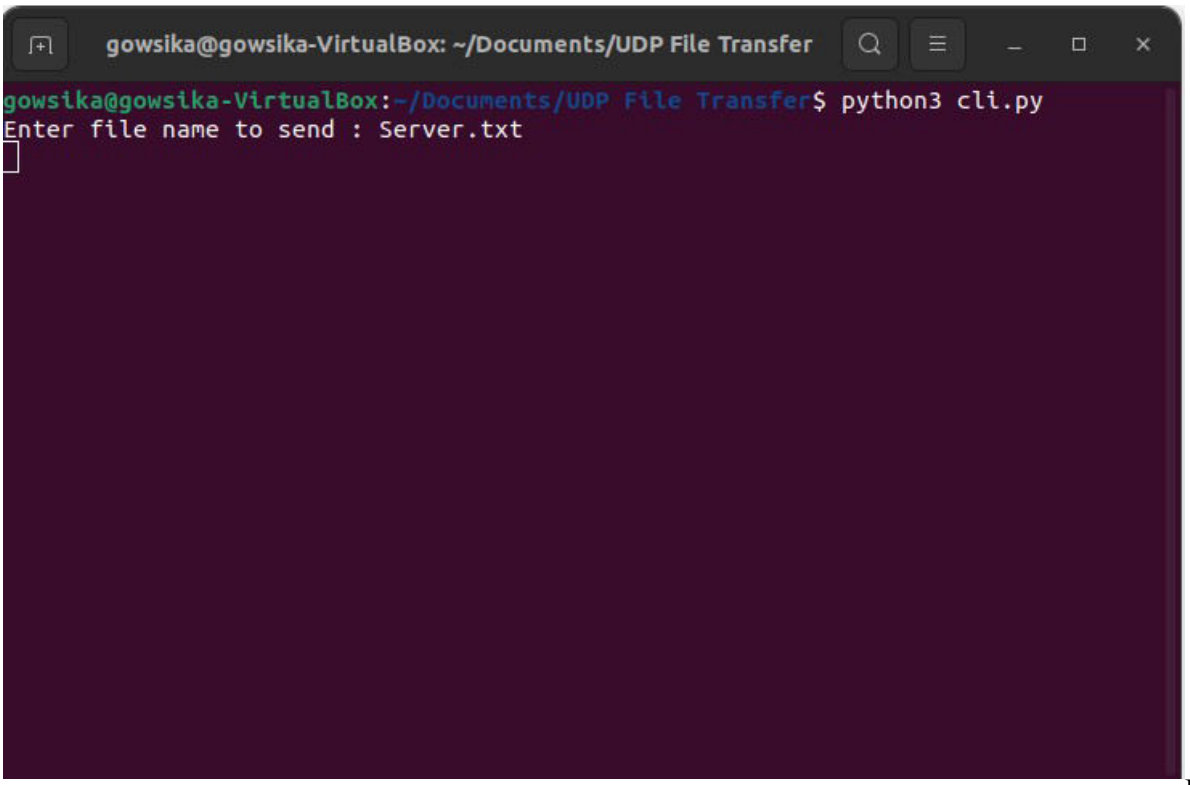
```
from __future__ import print_function
import pickle
import time
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor

class EchoClientDatagramProtocol(DatagramProtocol):

    def startProtocol(self):
        self.transport.connect('127.0.0.1', 1234)
        self.sendDatagram()

    def sendDatagram(self):
        file_name = input('Enter file name to send : ')
        file = open(f'{file_name}','r')
        file_data = file.read()
        send_time = time.time()                                  # Get the send timestamp
        data = (file_name, file_data, send_time)
        if file_name:
            self.transport.write(pickle.dumps(data))
        else:
            reactor.stop()

    def datagramReceived(self, datagram, host):
        print('Datagram received: ', repr(datagram))
        self.sendDatagram()


def main():
    protocol = EchoClientDatagramProtocol()
    t = reactor.listenUDP(0, protocol)
    reactor.run()

if __name__ == '__main__':
    main()
```

SAMPLE INPUT/OUTPUT:

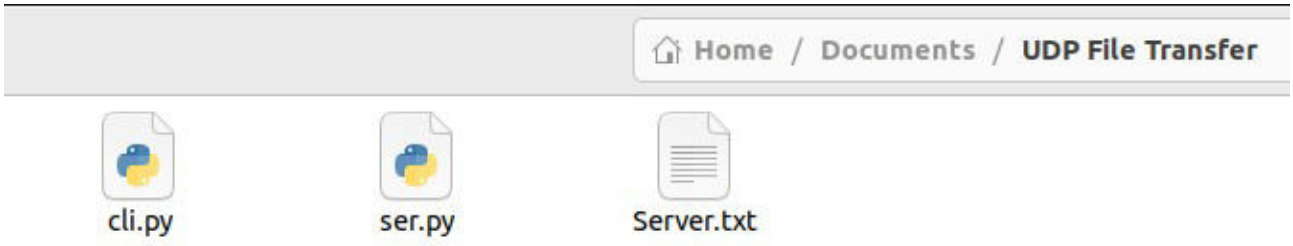
SERVER:



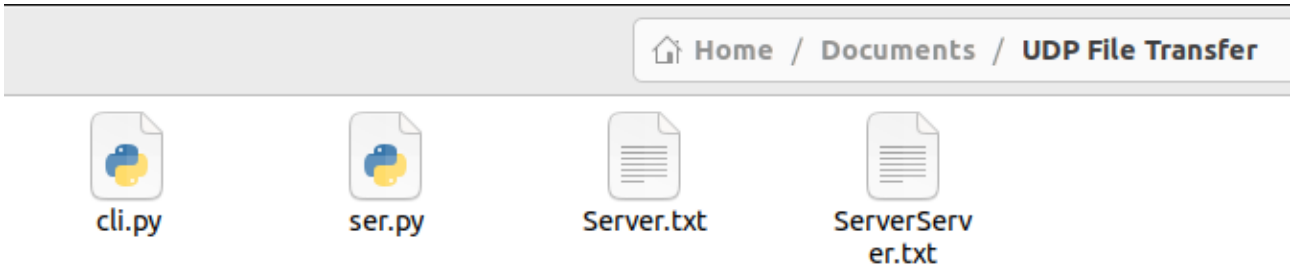
CLIENT:



Before executing:



After Executing:



- After executing the client, the text file is received by the server as ServerServer.txt.

RESULT:

The UDP Echo Server and Echo Client for Single Server, Single Client and Single Server, Multiple Client, Chat Server and File Transfer applications have been implemented and tested successfully using Twisted Python.

Simulation of ARP/RARP protocols using Twisted Python

Ex. No. 9
Date:

a) Simulation of ARP Protocol

PROBLEM STATEMENT:

To design and develop a Twisted Python application that listens for ARP requests from devices on the network and enables these devices to dynamically retrieve their MAC addresses.

PROBLEM DESCRIPTION:

Address Resolution Protocol (ARP) is a network protocol used to obtain an MAC address based on a device's IP address. It functions in the reverse direction of the traditional RARP (Reverse Address Resolution Protocol). ARP allows devices with unknown MAC addresses to send a request to a ARP server on the network, which maps the IP address to an MAC address and provides it to the requesting device.

ARP Request Scenario:

In an ARP request scenario, when the server's MAC address is unknown, a ARP request is broadcasted to all systems in the network. The ARP request packet contains the IP address of the requesting device, but the MAC address is set to 0:0:0:0:0:0 to indicate that the device does not yet have an MAC address assigned.

When the ARP request is received by the systems in the network, they compare the IP address in the request with their own IP addresses. If a system matches the IP address, it responds to the request with its own MAC address, providing the requested device with the necessary MAC address mapping.

This allows the device initiating the ARP request to obtain its MAC address dynamically, based on its IP address, without prior configuration or manual assignment.

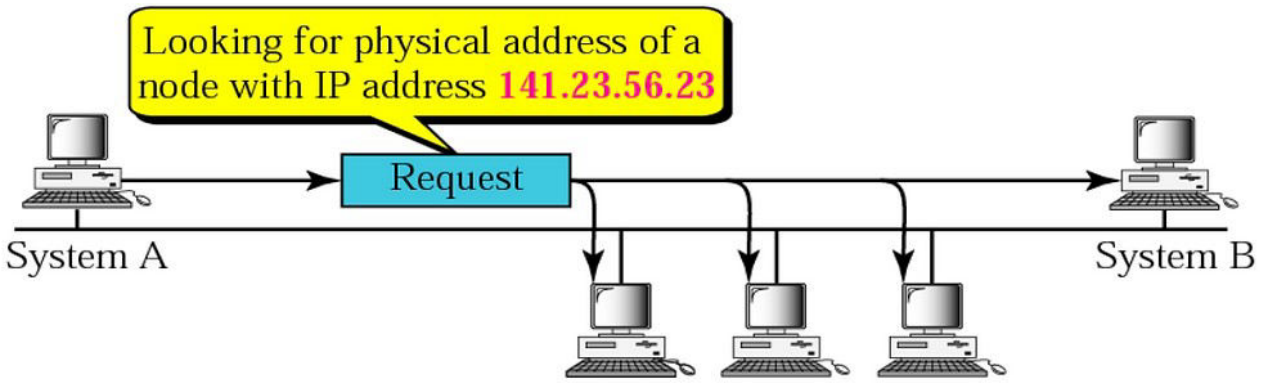
ARP Reply Scenario:

In an ARP reply scenario, an ARP server receives a ARP request from a device that is seeking its MAC address based on its IP address. The ARP server has a mapping of IP addresses to MAC addresses in its configuration.

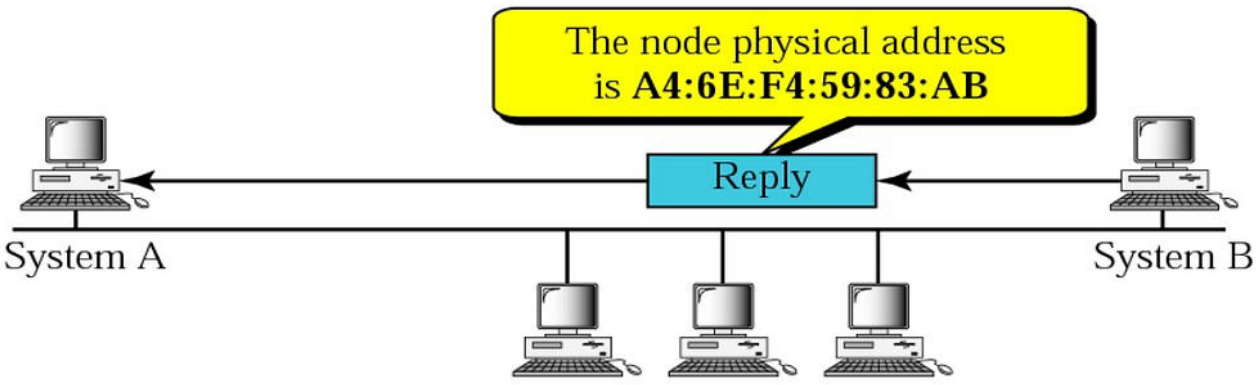
Upon receiving the ARP request, the ARP server checks its mapping and identifies the corresponding MAC address for the IP address provided in the request. It then constructs a ARP reply packet containing the IP and MAC addresses.

The ARP reply packet is sent directly to the requesting device, using its IP address as the destination address. Unlike ARP requests that are broadcasted, ARP replies are unicast and sent only to the specific device that made the request.

Upon receiving the ARP reply, the device updates its network configuration with the provided MAC address, allowing it to communicate on the network using the assigned MAC address.



a. ARP request is broadcast



b. ARP reply is unicast

ARP Request and Response

CODE:

Server:

```
from twisted.internet import reactor, protocol
import struct
class ARPServer(protocol.Protocol):
    def connectionMade(self):
        print("client connected")

    def dataReceived(self, data):

        global arp_tabel
        rec=eval(data.decode())
        mac_address='0:0:0:0:0:0'
        # get_parts=data.split()
        arp_packet_format="!6s4s6s4s"

        arp_data=struct.unpack(arp_packet_format,rec.get('req_format'))    #unpacking the format
        (
            Source_Hardware_Address,
            Source_Protocol_Address,
            Target_Hardware_Address,
            Target_Protocol_Address
        ) = arp_data

        print("Received ARP packet:")
        print("Source Hardware Address:", ":".join("{:02x}".format(byte) for byte in
Source_Hardware_Address))
        print("Source Protocol Address:", ".".join(str(byte) for byte in Source_Protocol_Address))
        print("Target Hardware Address:", ":".join("{:02x}".format(byte) for byte in
Target_Hardware_Address))
        print("Target Protocol Address:", ".".join(str(byte) for byte in Target_Protocol_Address))



        if rec.get('req')=="ARP_REQUEST":

            for i in arp_tabel:
                if i==rec.get('ip'):
```

```
                mac_address=arp_tabel[i]
            else:
                continue
        l=[]
        for i in mac_address.split(':'):
            l.append(int(i))                #list contains MAC address

        ip_address =rec.get('ip')  # Example IP address
        response_packet = struct.pack(          #packing the data to client now source and destination are swapped
        arp_packet_format,
        Target_Hardware_Address,
        Target_Protocol_Address,
        Source_Hardware_Address,
        bytes(l),

    )
        to_client={'reply_format':response_packet}          # dict to differntiate reply format and ip addres to be sent
        if mac_address !='0:0:0:0:0:0':
            arp_reply = f'ARP_REPLY {ip_address} {mac_address}\n'

            to_client['data']=arp_reply
            self.transport.write(str(to_client).encode())     # encoded data is send
            print("MAC Address sent")

        else:
            self.transport.write(b'hi')
            print("invalid IP recieved ")

    def connectionLost(self, reason):
        print("client removed")
        return
class ARPServerFactory(protocol.Factory):
    def buildProtocol(self, addr):
```

```
        return ARPServer()

arp_tabel={}
arp_tabel['192.168.1.1']='00:11:22:33:44:55'

reactor.listenTCP(1234, ARPServerFactory())
reactor.run()
```

Client:

```
from twisted.internet import reactor, protocol
import struct
class ARPClient(protocol.Protocol):

    def connectionMade(self):
        arp_packet_format="!6s4s6s4s"
        request_packet=struct.pack(arp_packet_format,

            bytes([00, 11, 22, 33, 44, 55]),            # Example source hardware address
            bytes([0, 0, 0, 0]),                # Example source protocol address
            bytes([17, 18, 19, 20, 21, 22]),        # Example target hardware address
            bytes([26, 27, 28, 29])                # Example target protocol address
        )

        a=input("enter IP address:")
        to_serv={'ip':a,'req_format':request_packet,'req':'ARP_REQUEST'}  #dict is used
        self.transport.write(str(to_serv).encode())

    def dataReceived(self, data):
        recv=eval(data.decode())            #data is decoded with same structure sent(dict)
        arp_packet_format = "!6s4s6s4s"
        (
            Source_Hardware_Address,
            Source_Protocol_Address,
            Target_Hardware_Address,
```

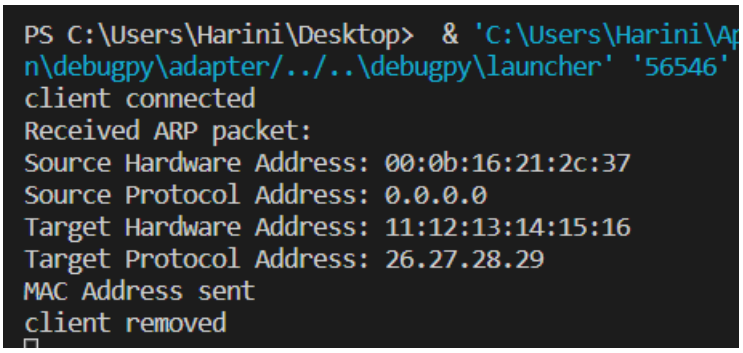
```
            Target_Protocol_Address
        ) = struct.unpack(arp_packet_format,recv.get('reply_format'))  #unpacking the format

        print("Received ARP reply:")
        print("Source Hardware Address:", ":".join("{:02x}".format(byte) for byte in
Source_Hardware_Address))
        print("Source Protocol Address:", ".".join(str(byte) for byte in Source_Protocol_Address))
        print("Target Hardware Address:", ":".join("{:02x}".format(byte) for byte in
Target_Hardware_Address))
        print("Target Protocol Address:", ".".join(str(byte) for byte in Target_Protocol_Address))

        if recv.get('data').startswith('ARP_REPLY'):

            reply_parts = recv.get('data').split()
            if len(reply_parts) == 3:

                mac_address = reply_parts[2]
                ip_address = reply_parts[1]
                print(f"Received ARP reply: IP = {ip_address }, MAC = {mac_address}")
                self.transport.loseConnection()          # to terminate after the data is print
            else:
                print("Invalid ARP reply")
                self.transport.loseConnection()
        else:
            print("invalid IP Address given !!!")
            self.transport.loseConnection()

class ARPClientFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return ARPClient()

    def clientConnectionFailed(self, connector, reason):
        print("Connection failed.")
        reactor.stop()
```

```
    def clientConnectionLost(self, connector, reason):
        print("Connection lost.")
        reactor.stop()
reactor.connectTCP('localhost', 1234, ARPClientFactory())
reactor.run()
```

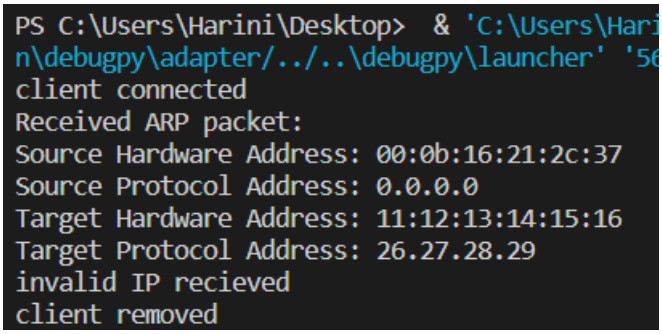
Sample Input/Output:

Server:

Valid IP address:



Invalid IP address:



Client:

Valid IP address:

```
PS C:\Users\Harini\Desktop>  & 'C:\Users\Harini\AppData\Local\Micro
n\debugpy\adapter/../..\debugpy\launcher' '56555' '--' 'c:\Users\Ha
enter IP address:192.168.1.1
Received ARP reply:
Source Hardware Address: 11:12:13:14:15:16
Source Protocol Address: 26.27.28.29
Target Hardware Address: 00:0b:16:21:2c:37
Target Protocol Address: 0.11.22.33
Received ARP reply: IP = 192.168.1.1, MAC = 00:11:22:33:44:55
Connection lost.
PS C:\Users\Harini\Desktop>
```

Invalid IP address:

```
\adapter/../..\debugpy\launche
enter IP address:168.2.3.1
invalid IP Address given !!!
```

If Server is not ready:

```
PS C:\Users\Harini\Desktop>  &
/../..\debugpy\launcher' '56735
Connection failed.
PS C:\Users\Harini\Desktop>
```

After server is terminated:

```
PS C:\Users\Harini\Desktop>  &
n\debugpy\adapter/../..\debugp
enter IP address:192.168.1.1
Connection lost.
PS C:\Users\Harini\Desktop>
```

b) Simulation of RARP Protocol

PROBLEM STATEMENT:

To design and develop a Twisted Python application that listens for RARP requests from devices on the network and enables these devices to dynamically retrieve their IP addresses.

PROBLEM DESCRIPTION:

Reverse Address Resolution Protocol (RARP) is a network protocol used to obtain an IP address based on a device's physical MAC address. It functions in the reverse direction of the traditional ARP (Address Resolution Protocol). RARP allows devices with unknown IP addresses to send a

request to a RARP server on the network, which maps the MAC address to an IP address and provides it to the requesting device.

RARP Request scenario:

In a RARP request scenario, when the device's IP address is unknown, a RARP request is broadcasted to all systems in the network. The RARP request packet contains the MAC address of the requesting device, but the IP address is set to 0.0.0.0 to indicate that the device does not yet have an IP address assigned.

When the RARP request is received by the systems in the network, they compare the MAC address in the request with their own MAC addresses. If a system matches the MAC address, it responds to the request with its own IP address, providing the requested device with the necessary IP address mapping.

This allows the device initiating the RARP request to obtain its IP address dynamically, based on its MAC address, without prior configuration or manual assignment.

RARP Reply Scenario:

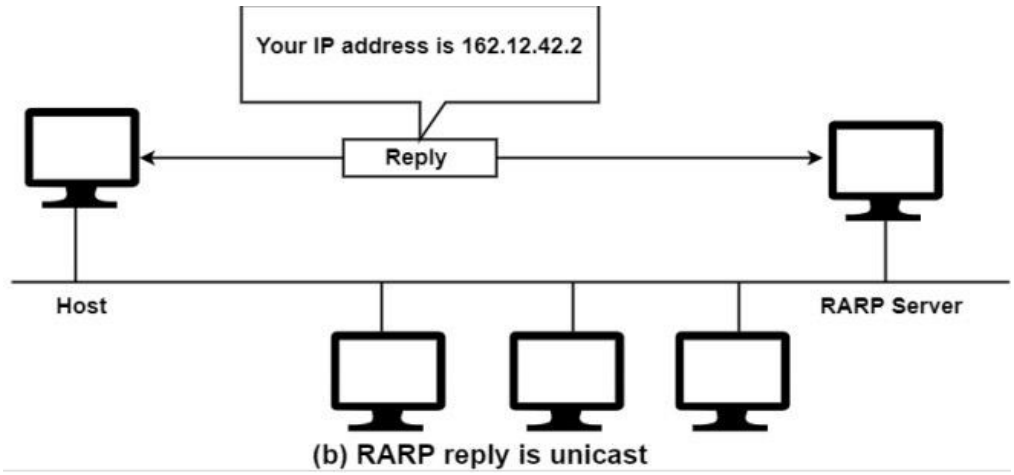
In a RARP reply scenario, a RARP server receives a RARP request from a device that is seeking its IP address based on its MAC address. The RARP server has a mapping of MAC addresses to IP addresses in its configuration.

Upon receiving the RARP request, the RARP server checks its mapping and identifies the corresponding IP address for the MAC address provided in the request. It then constructs a RARP reply packet containing the MAC and IP addresses.

The RARP reply packet is sent directly to the requesting device, using its MAC address as the destination address. Unlike RARP requests that are broadcasted, RARP replies are unicast and sent only to the specific device that made the request.

Upon receiving the RARP reply, the device updates its network configuration with the provided IP address, allowing it to communicate on the network using the assigned IP address.

RARP Request and Reply:



RARP Table:

Sampl

Serve

from t

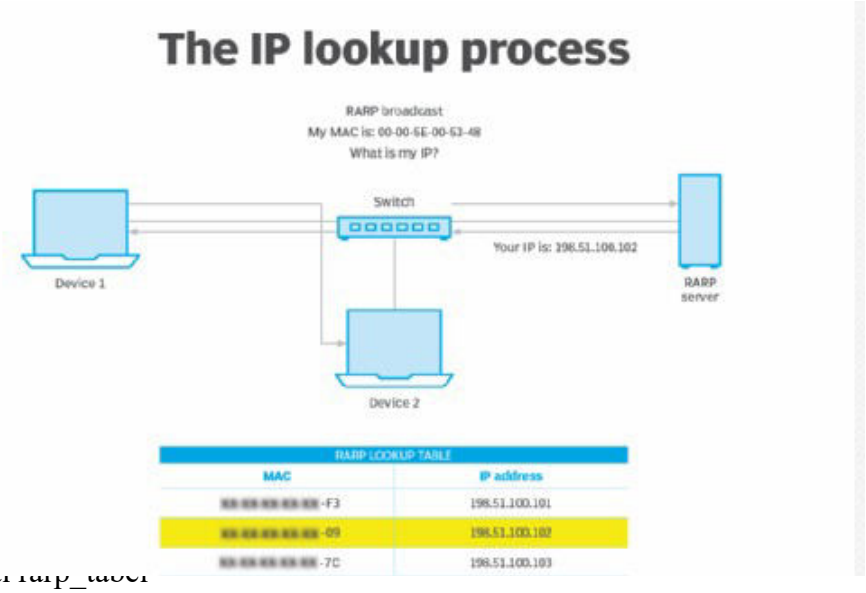
import

class R

def

p

def




```
    global rarp_table
    rec=eval(data.decode())
    ip_address='0.0.0.0'
    # get_parts=data.split()
    rarp_packet_format="!6s4s6s4s"

    rarp_data=struct.unpack(rarp_packet_format,rec.get('req_format'))    #unpacking the format
    (
        Source_Hardware_Address,
        Source_Protocol_Address,
        Target_Hardware_Address,
        Target_Protocol_Address
    ) = rarp_data
```

```
        print("Received RARP packet:")
        print("Source Hardware Address:", ":".join("{:02x}".format(byte) for byte in
Source_Hardware_Address))
        print("Source Protocol Address:", ".".join(str(byte) for byte in Source_Protocol_Address))
        print("Target Hardware Address:", ":".join("{:02x}".format(byte) for byte in
Target_Hardware_Address))
        print("Target Protocol Address:", ".".join(str(byte) for byte in Target_Protocol_Address))



        if rec.get('req')=="RARP_REQUEST":

            for i in rarp_tabel:
                if i==rec.get('mac'):
                    ip_address=rarp_tabel[i]
                else:
                    continue
            l=[]
            for i in ip_address.split('.'):
                l.append(int(i))                #list contains ip address

            mac_address =rec.get('mac')  # Example MAC address
            response_packet = struct.pack(          #packing the data to client now source and destination
are swapped
            rarp_packet_format,
            Target_Hardware_Address,
            Target_Protocol_Address,
            Source_Hardware_Address,
            bytes(l),


        )

            to_client={'reply_format':response_packet}          # dict to differntiate reply format and ip
addres to be sent
            if ip_address !='0.0.0.0':
                rarp_reply = f'RARP_REPLY {mac_address} {ip_address}\n'
```

```
            to_client['data']=rarp_reply
            self.transport.write(str(to_client).encode())     # encoded data is send
            print("IP Address sent")

        else:
            self.transport.write(b'hi')
            print("invalid MAC recieved ")

    def connectionLost(self, reason):
        print("client removed")
        return
class RARPServerFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return RARPServer()

rarp_tabel={}
rarp_tabel['00:11:22:33:44:55']='192.168.1.1'

reactor.listenTCP(1234, RARPServerFactory())
reactor.run()
```

Client:

```
from twisted.internet import reactor, protocol
import struct
class RARPClient(protocol.Protocol):

    def connectionMade(self):
        rarp_packet_format="!6s4s6s4s"
        request_packet=struct.pack(rarp_packet_format,

            bytes([00, 11, 22, 33, 44, 55]),            # Example source hardware address
            bytes([0, 0, 0, 0]),              # Example source protocol address
            bytes([17, 18, 19, 20, 21, 22]),       # Example target hardware address
            bytes([26, 27, 28, 29])              # Example target protocol address
```

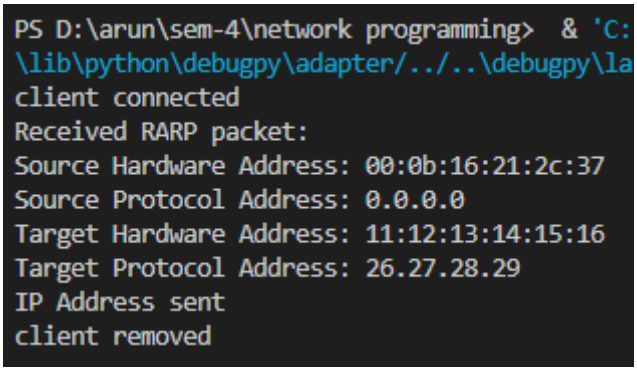
```
        )

        a=input("enter MAC address:")
        to_serv={'mac':a,'req_format':request_packet,'req':'RARP_REQUEST'}  #dict is used
        self.transport.write(str(to_serv).encode())

    def dataReceived(self, data):
        recv=eval(data.decode())            #data is decoded with same structure sent(dict)
        rarp_packet_format = "!6s4s6s4s"

        (
            Source_Hardware_Address,
            Source_Protocol_Address,
            Target_Hardware_Address,
            Target_Protocol_Address
        ) = struct.unpack(rarp_packet_format,recv.get('reply_format'))  #unpacking the format

        print("Received RARP reply:")
        print("Source Hardware Address:", ":".join("{:02x}".format(byte) for byte in
Source_Hardware_Address))
        print("Source Protocol Address:", ".".join(str(byte) for byte in Source_Protocol_Address))
        print("Target Hardware Address:", ":".join("{:02x}".format(byte) for byte in
Target_Hardware_Address))
        print("Target Protocol Address:", ".".join(str(byte) for byte in Target_Protocol_Address))

        if recv.get('data').startswith('RARP_REPLY'):

            reply_parts = recv.get('data').split()
            if len(reply_parts) == 3:

                mac_address = reply_parts[1]
                ip_address = reply_parts[2]
                print(f"Received RARP reply: MAC = {mac_address}, IP = {ip_address}")
                self.transport.loseConnection()            # to terminate after the data is print
            else:
```

```
            print("Invalid RARP reply")
            self.transport.loseConnection()
        else:
            print("invalid MAC Address given !!!")
            self.transport.loseConnection()

class RARPClientFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return RARPClient()

    def clientConnectionFailed(self, connector, reason):
        print("Connection failed.")
        reactor.stop()

    def clientConnectionLost(self, connector, reason):
        print("Connection lost.")
        reactor.stop()
reactor.connectTCP('localhost', 1234, RARPClientFactory())
reactor.run()
```

Sample Input/Output:

Server:

Valid MAC address:



Invalid MAC address:

```
PS D:\arun\sem-4\network programming>  & 'C
\lib\python\debugpy\adapter/../..\debugpy\l
client connected
Received RARP packet:
Source Hardware Address: 00:0b:16:21:2c:37
Source Protocol Address: 0.0.0.0
Target Hardware Address: 11:12:13:14:15:16
Target Protocol Address: 26.27.28.29
invalid MAC recieved
client removed
```

Client:

Valid MAC address:

```
PS D:\arun\sem-4\network programming>  & 'C:\Python310\python.exe
\lib\python\debugpy\adapter/../..\debugpy\launcher' '62983' '--'
enter MAC address:00:11:22:33:44:55
Received RARP reply:
Source Hardware Address: 11:12:13:14:15:16
Source Protocol Address: 26.27.28.29
Target Hardware Address: 00:0b:16:21:2c:37
Target Protocol Address: 192.168.1.1
Received RARP reply: MAC = 00:11:22:33:44:55, IP = 192.168.1.1
Connection lost.
```

Invalid MAC address:

```
PS D:\arun\sem-4\network programming>  & 'C:\
\lib\python\debugpy\adapter/../..\debugpy\lau
enter MAC address:00:11
Received RARP reply:
Source Hardware Address: 11:12:13:14:15:16
Source Protocol Address: 26.27.28.29
Target Hardware Address: 00:0b:16:21:2c:37
Target Protocol Address: 0.0.0.0
invalid MAC Address given !!!
Connection lost.
```

If Server is not ready:

```
PS D:\arun\sem-4\network programming>
y\adapter/../..\debugpy\launcher' '58
Connection failed.
```

RESULT:

Thus, the simulation of ARP/RARP protocols have been implemented and tested successfully using Twisted Python.

Implementation of Stop and Wait Protocol and Sliding Window Protocol using Twisted Python

Ex. No. 10
Date:

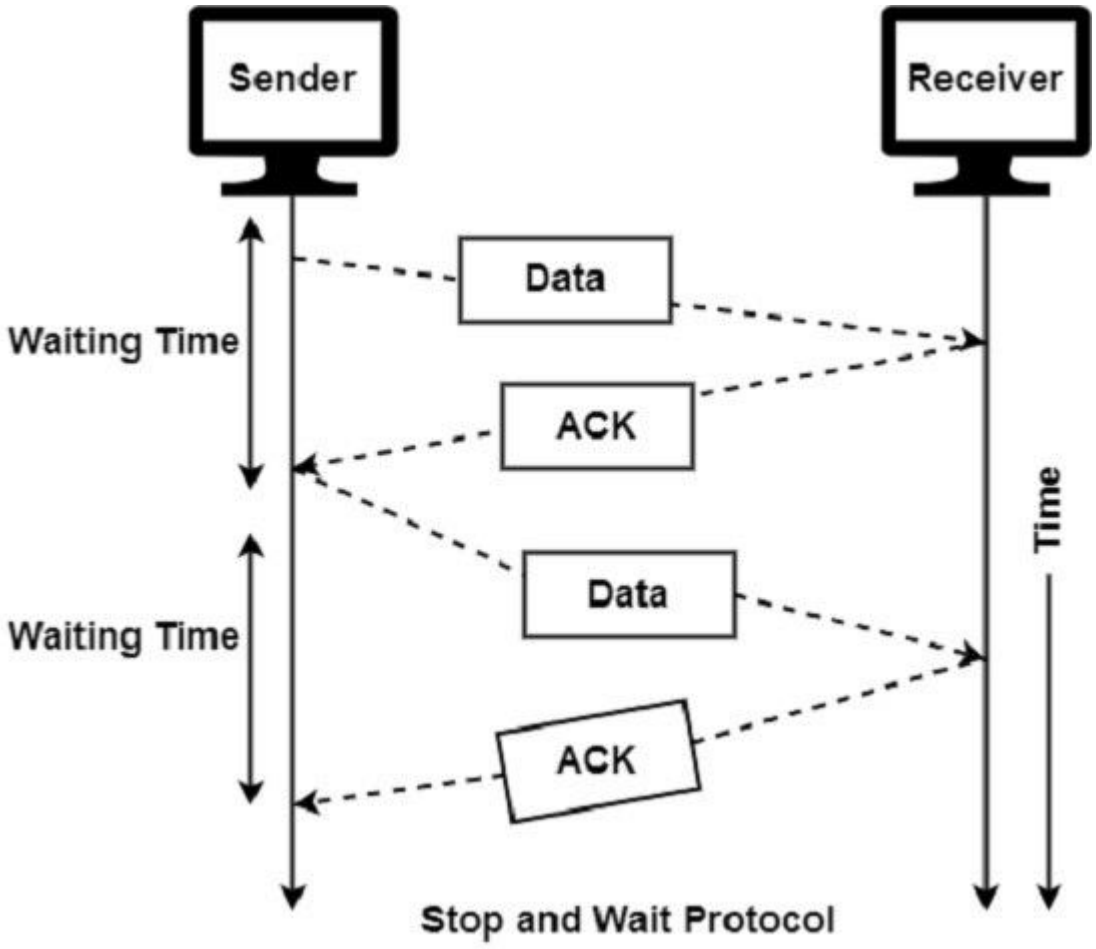
a) Implementation of Stop and Wait Protocol

PROBLEM STATEMENT:

To write a program to implement the stop and wait protocol using Twisted Python.

PROBLEM DESCRIPTION:

Stop and Wait protocol is where the client and server communicate based on the acknowledgement received. The server first gets connected to the client and the client sends an acknowledgement message “ACK”. If “ACK” is received then communication begins, else it must again try to get connected. Once the connection is made, the server starts sending messages. Every time a message is sent to the client, it sends back an acknowledgement if it has received it. If “ACK” is received from the client, the server sends the next message, else it waits for a specific duration called as the timeout period. If it is a timeout, then the server resends the same message.



CODE:

#Server

```
from twisted.internet import reactor, protocol

class StopAndWaitServer(protocol.Protocol):
    def connectionMade(self):
        print("Client connected:", self.transport.getPeer())
        self.send_message()

    def send_message(self):
        message = input("Enter message: ")
        self.transport.write(message.encode())
        print("Message sent to client:", message)
        self.expected_ack = "ACK"
        #self.schedule_resend()

    def schedule_resend(self):
        self.resend_call = reactor.callLater(5, self.resend_message)

    def resend_message(self):
        print("ACK not received. Resending message...")
        self.send_message()

    def dataReceived(self, data):
        ack = data.decode()
        print("ACK received:", ack)
        if ack == self.expected_ack:
            #self.resend_call.cancel()
            print("ACK received. Message acknowledged.")
            self.send_message()
        else:
            print("Invalid ACK received.")
            self.schedule_resend()

    def connectionLost(self, reason):
        print("Client disconnected:")

class StopAndWaitServerFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return StopAndWaitServer()

server_port = 8000

factory = StopAndWaitServerFactory()
reactor.listenTCP(server_port, factory)

reactor.run()
```

#Client

```
from twisted.internet import reactor, protocol

class StopAndWaitClient(protocol.Protocol):
    def connectionMade(self):
        print("Connected to server.")
        self.send_ack()

    def send_ack(self):
        self.transport.write(input("Enter ack: ").encode())
        print("ACK sent")

    def dataReceived(self, data):
        message = data.decode()
        print("Message received:", message)
        self.send_ack()

    def connectionLost(self, reason):
        print("Connection lost:", reason.getErrorMessage())

class StopAndWaitClientFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return StopAndWaitClient()

    def clientConnectionFailed(self, connector, reason):
        print("Connection failed:", reason.getErrorMessage())
        reactor.stop()

    def clientConnectionLost(self, connector, reason):
        print("Connection lost:", reason.getErrorMessage())
        reactor.stop()

server_address = 'localhost'
server_port = 8000

factory = StopAndWaitClientFactory()
reactor.connectTCP(server_address, server_port, factory)
reactor.run()
```

SAMPLE INPUT/OUTPUT:

Server:

```
student@student-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ python3 stop_n_wait_server.py
Client connected: IPv4Address(type='TCP', host='127.0.0.1', port=39390)
Enter message: Hi
Message sent to client: Hi
ACK received: ACKack
Invalid ACK received.
ACK not received. Resending message...
Enter message: Hi
Message sent to client: Hi
ACK received: ack
Invalid ACK received.
ACK not received. Resending message...
Enter message: Hello
Message sent to client: Hello
ACK received: ACK
ACK received. Message acknowledged.
Enter message: ^Z
[1]+  Stopped                 python3 stop_n_wait_server.py
student@student-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$
```

Client:

```
student@student-HP-ProDesk-400-G7-Microtower-PC:~/Downloads$ python3 stop_n_wait_client.py
Connected to server.
Enter ack: ACK
ACK sent
Message received: Hi
Enter ack: ack
ACK sent
Message received: Hi
Enter ack: ack
ACK sent
Message received: Hello
Enter ack: ACK
ACK sent
```

b) Implementation of Sliding Window Protocol - Go-Back-N

PROBLEM STATEMENT:

To write a program to implement the sliding window protocol (Go-Back-N) using Twisted Python.

PROBLEM DESCRIPTION:

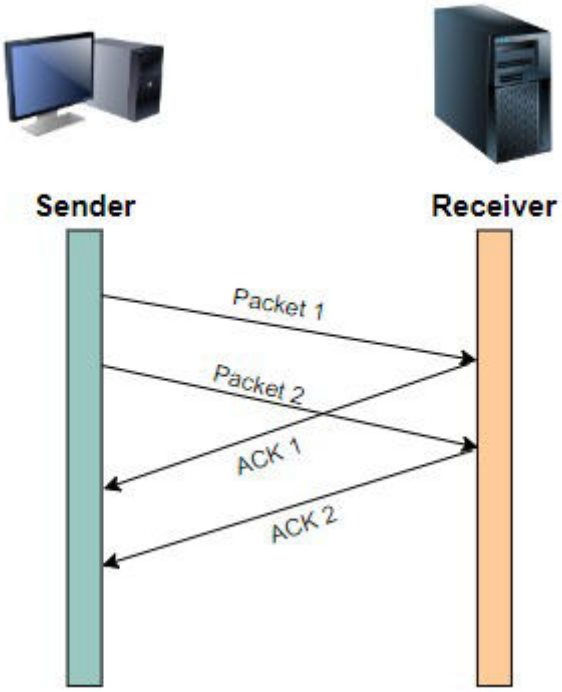
What Is Go-Back-N Protocol?

- This protocol is based on the method of using the Sliding Window Protocol as the basis of data exchange, where the 'N' in the protocol represents the window size.

- The number of frames transmitted while the sender waits for an acknowledgment is also determined by the value of N. The sender sends frames in sequential order. The receiver can only buffer one frame at a time since the size of the receiver window is equal to 1, but the sender can buffer N frames.

- The Go-Back-N sender uses a retransmission timer to detect the lost segments. The Go-Back-N receiver uses both independent acknowledgment and cumulative acknowledgment.

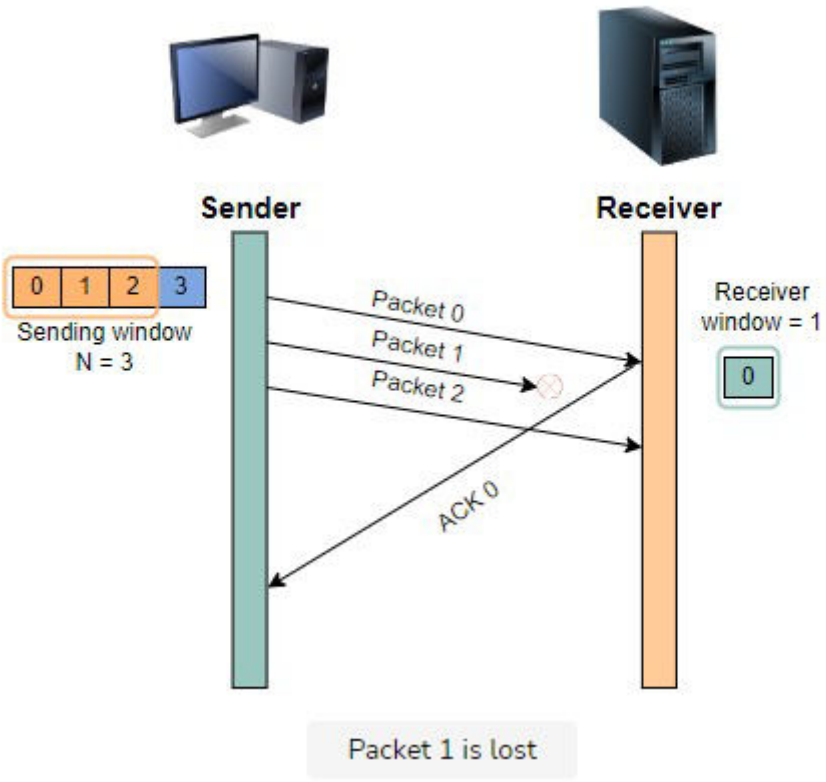
- The following illustration assists in grasping the concept of the Go-Back-N protocol:



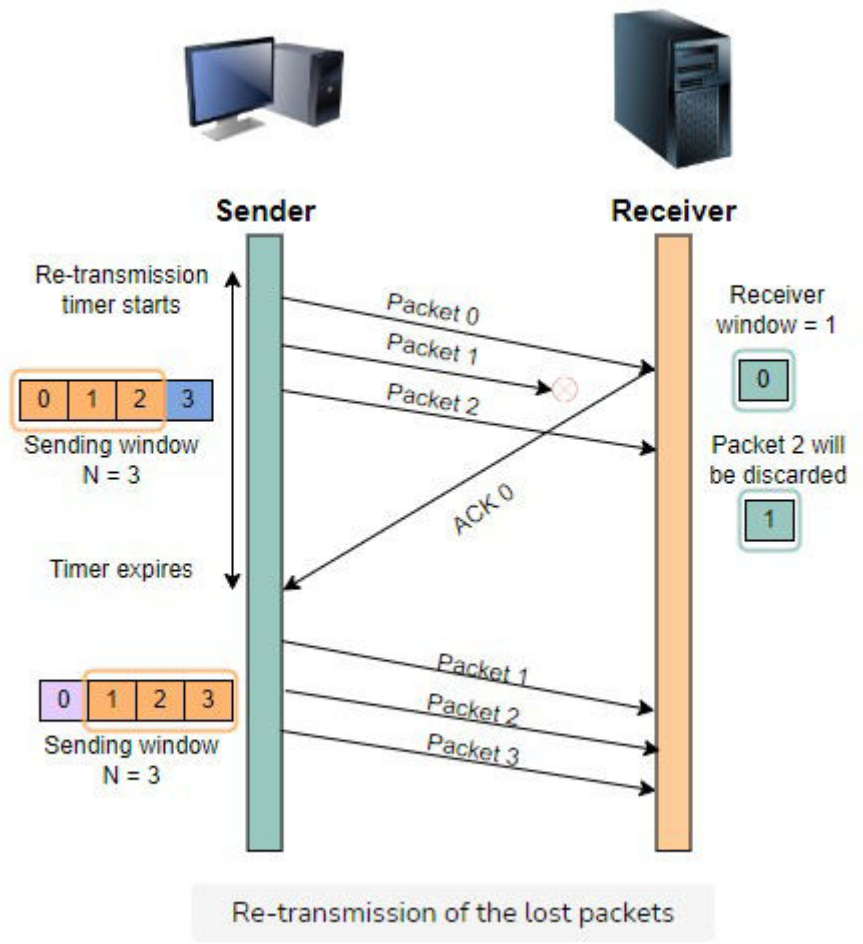
How it works?

Now, with the aid of an illustration, we'll understand how Go-Back-N operates.

- The sender's window has a size of 3 or Go-Back-3.
- The sender has sent out packets 0, 1, and 2.
- The receiver is now anticipating packet 1 after acknowledging packet 0.
- Packet 1 is lost somewhere in the network.



- Since it's waiting for the packet with sequence number 1, the receiver will ignore every packet the sender delivered after packet one.
- On the sender's side, a retransmission timer is set to expire.
- The sender will re-transmit all packets from one up to N, which is 3.



CODE

SENDER:

```
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor, task

class Sender(DatagramProtocol):
    def __init__(self):
        self.seq_num = 0
        self.window_size = 4
        self.base = 0
        self.timer = None
        self.packets = [
            b"Packet 0",
            b"Packet 1",
            b"Packet 2",
            b"Packet 3",
            b"Packet 4",
            b"Packet 5",
            b"Packet 6",
            b"Packet 7"
        ]
        self.sent = [False] * len(self.packets)
```

```
    def startProtocol(self):
        self.sendPackets()

    def sendPackets(self):
        while self.seq_num < min(self.base + self.window_size, len(self.packets)):
            if not self.sent[self.seq_num]:
                print("Sending packet", self.seq_num)
                self.transport.write(self.packets[self.seq_num], ("127.0.0.1",
8000))
                #print(f'SP SEQ:{self.seq_num} BASE: {self.base} TIME:
{self.timer}')
                if self.base == self.seq_num:
                    self.startTimer()
                self.sent[self.seq_num] = True
            self.seq_num += 1

    def startTimer(self):
        if self.timer is not None and self.timer.active():
            self.timer.cancel()
        self.timer = reactor.callLater(5, self.handleTimeout)

    def handleTimeout(self):
        print("Timeout, resending packets from", self.base)
        self.seq_num = self.base
        # print(f'SEQ :{self.seq_num},BASE:{self.base}')
        for i in range(self.seq_num,self.window_size + 1) :
            self.sent[i] = False

        self.sendPackets()

    def datagramReceived(self, datagram, address):

        ack_num = int(datagram.decode())
        print("Received ACK", ack_num)
        if ack_num >= self.base:
            self.base = ack_num + 1
            if self.base == self.seq_num:
                if self.timer is not None and self.timer.active():
                    self.timer.cancel()
                self.sendPackets()
            else:
                self.startTimer()

if __name__ == "__main__":
    reactor.listenUDP(9000, Sender())
    reactor.run()
```

RECIEVER:

```
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor
import time

class Receiver(DatagramProtocol):
    status = False
    expected_packet = 0

    def datagramReceived(self, datagram, address):
        packet_num = int(datagram.decode()[-1])



        if packet_num == 2 and not self.status:
            self.status = True
            print('break')
            time.sleep(7)
            return

        if packet_num == self.expected_packet:
            print("Received packet", packet_num)
            ack = str(packet_num)
            self.transport.write(ack.encode(), address)
            self.expected_packet += 1

        else :
            print("Discarded packet", packet_num)

if __name__ == "__main__":
    reactor.listenUDP(8000, Receiver())
    reactor.run()
```

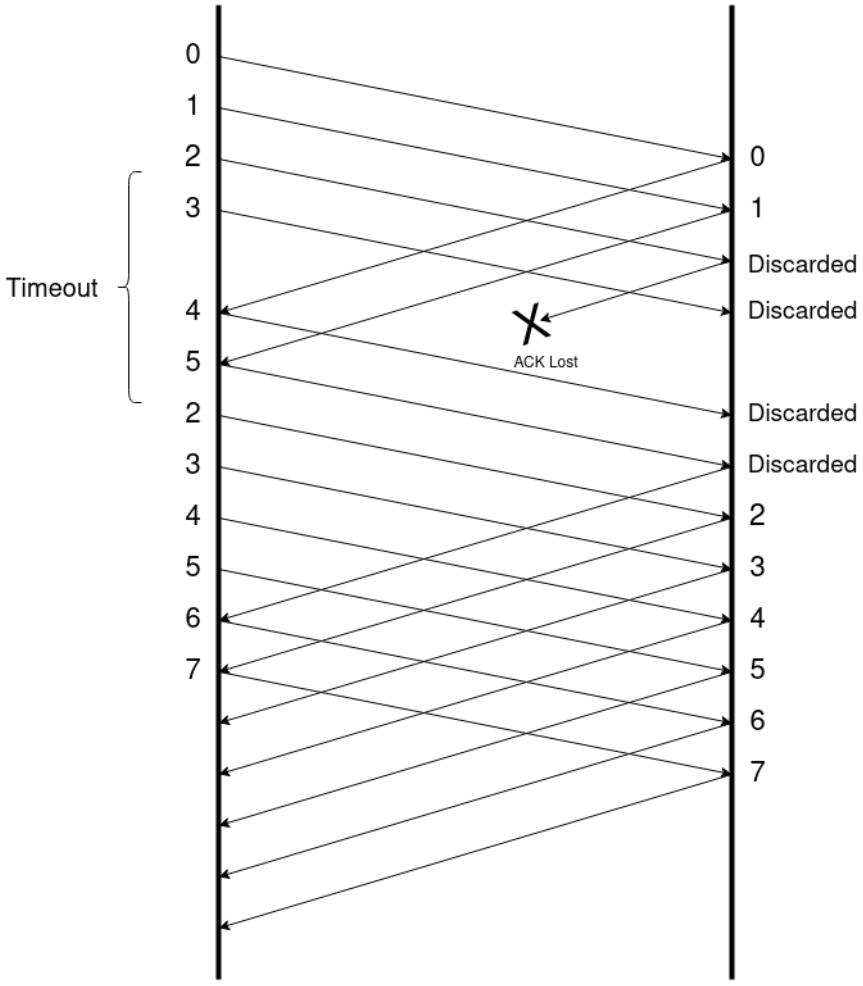
SAMPLE INPUT/OUTPUT:

(The program execution is put to sleep for 7 seconds if the receiver receives packet 2, to simulate loss of ACK for packet 2)

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\mitul\OneDrive\Desktop\SEM 4\Network lab> python3 Sender.py
Sending packet 0
Sending packet 1
Sending packet 2
Sending packet 3
Received ACK 0
Received ACK 1
Timeout, resending packets from 2
Sending packet 2
Sending packet 3
Sending packet 4
Sending packet 5
Received ACK 2
Received ACK 3
Received ACK 4
Received ACK 5
Sending packet 6
Sending packet 7
Received ACK 6
Received ACK 7
PS C:\Users\mitul\OneDrive\Desktop\SEM 4\Network lab>
```

```
PS C:\Users\mitul\OneDrive\Desktop\SEM 4\Network lab> python3 Reciever.py
Received packet 0
Received packet 1
break
Discarded packet 3
Received packet 2
Received packet 3
Received packet 4
Received packet 5
Received packet 6
Received packet 7
PS C:\Users\mitul\OneDrive\Desktop\SEM 4\Network lab>
```

Illustration:



c) Implementation of Sliding Window Protocol – Selective Repeat

PROBLEM STATEMENT:
To write a program to implement the sliding window protocol (Selective Repeat) using Twisted Python using TCP and UDP sockets.

PROBLEM DESCRIPTION:
Selective repeat protocol is a sliding window protocol that uses the concept of pipelining where multiple packets can be sent while the sender is waiting for the acknowledgement for the first sent packet. The selective repeat protocol manages error and flows control between the sender and receiver.

Why Selective Repeat ARQ?

- The Go-Back-N protocol is efficient if the error rate and loss of packets is less. If the connection is poor, there will be frequent loss of packets and the sender would have to retransmit all the outstanding packets. This wastes the bandwidth of the channel.

- In Go-Back-N, the size of the receiver window is 1 so it buffers only one packet in order that it has to acknowledge next and if this expected packet is lost or corrupted all the received out of order packets are discarded. And the sender has to retransmit all the outstanding packets though some of these may have arrived at the receiver safely but out of order.

Working of Selective Repeat Protocol

Like go-back-n, selective repeat protocol is also a sliding window protocol. Unlike the go-back-n protocol, the selective repeat protocol resends only a selective packet that is either lost or corrupted. Let us first discuss the windows in selective repeat.

Window

In selective repeat, both sender and receiver have a sliding window. On the sender side, the window covers the sequence of packets that are either sent or can be sent. At the receiver, the sliding window covers the sequence number of the packets that are either received or are expected to be received. In selective repeat, the size of the sender and receiver window is the same.

1. Sender Window

The sender window in selective repeat is much smaller as compared to the go-back-n protocol. The size of the sender window here is 2^{m-1}. Here m is the number of bits used by the packet header to express the sequence number of the corresponding packet.

The sender window covers the packets that are sent but not yet acknowledged, one that is acknowledged out of order and the one that can be sent once the data for the corresponding are received by the sender's application layer.

2. Receiver Window

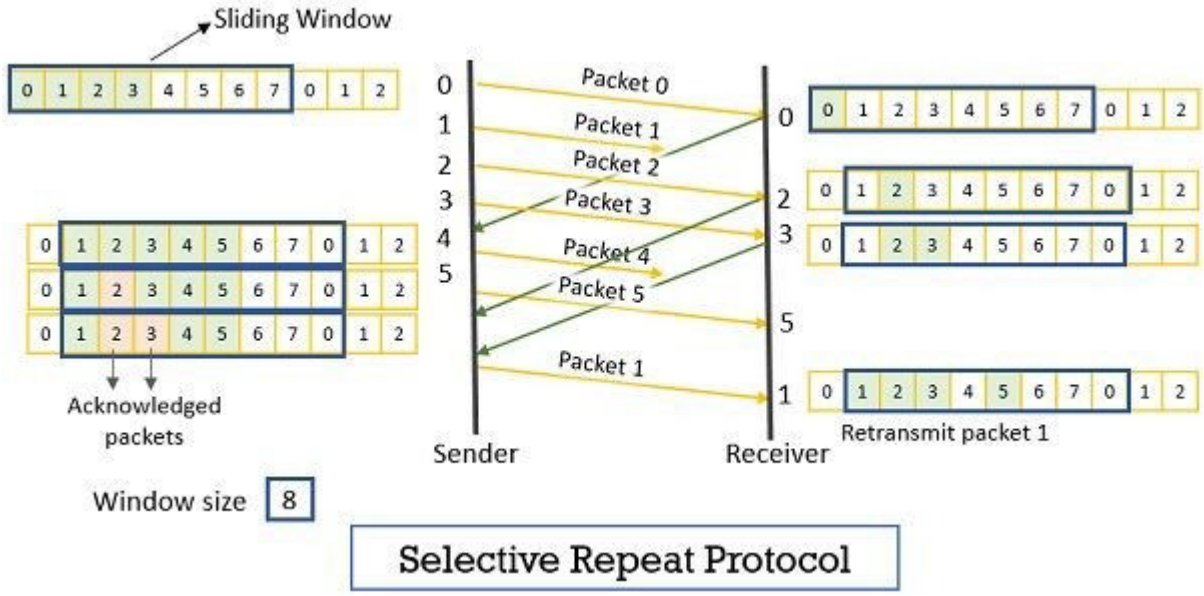
The maximum size of the receiver window is 2^{m-1} which is the same as the sender window. The receiver window covers the sequence number of the packets that are received out of order and are waiting for the packets that were sent earlier but are not yet received.

The receiver transport layer does not deliver packets out of order to the application layer. It waits until a set of consecutive packets are received so that they can be delivered to the application layer.

n selective repeat, at the sender side, a timer is attached to each sent packet and if the acknowledgement is not received before the timer expire the corresponding packet is resent.

Let understand selective repeat protocol with the help of an example. Consider that the header of a packet has allotted 3 bits to resemble the sequence number of the corresponding packet so m=3 here. Now as m is 3 the packets are sequenced using modulo 2^m. So the sequence number of packets lie from 0, 1, 2, 3, …, 7.

The size of the receiver and sender window will be 2^{m-1} i.e. $2^{3-1} = 4$. So the size of the receiver and sender window is 4.



Algorithm:
Using TCP:

Sender Algorithm:

1. Initialize the window size, send base, send next, and receiver base variables.

2. Establish a connection.

3. Upon connection, call the sendPackets() function.

4. In the sendPackets() function: a. While send_next is within the window and there are packets in the buffer: i. Get the packet at send_next from the buffer. ii. Send the packet to the receiver. iii. Increment send_next.

5. When data is received from the receiver: a. Check if the received data is an acknowledgment (ACK). b. If not an ACK, consider it as a packet loss. c. If it is an ACK: i. Increment send_base and send_next. ii. Call the sendPackets() function.

6. Handle the connectionLost() event.

Receiver Algorithm:

1. Initialize the window size, receive base, and sender base variables.

2. Upon receiving a packet: a. Check if the packet is within the window. b. If yes, process the packet. c. If not, consider it as a duplicate packet and ignore it.

3. Send an acknowledgment (ACK) for the received packet.

4. Move the receive base forward to the next expected packet.

5. Handle the connectionLost() event.

Using UDP:

1. Import the required modules: `twisted.internet.protocol` for the base class `DatagramProtocol`, and `twisted.internet` for the reactor.

2. Define the server class `SelectiveRepeatServer`, inheriting from `DatagramProtocol`. Initialize the server with an expected sequence number and an empty buffer.

3. Implement the `datagramReceived` method in the server class. This method is called when a datagram is received by the server. It extracts the sequence number and data from the received datagram. If the sequence number matches the expected sequence number, the server acknowledges the packet by sending the sequence number back to the client. It then checks the buffer for any previously received out-of-order packets and delivers them if found.

4. Define the client class `SelectiveRepeatClient`, inheriting from `DatagramProtocol`. Initialize the client with the filename to be sent, the current sequence number, window size, and a dictionary to store sent packets.

5. Implement the `startProtocol` method in the client class. This method is called when the client protocol is started. It initiates the sending of packets by calling the `sendPackets` method.

6. Implement the `sendPackets` method in the client class. It reads data from the file in chunks of 1024 bytes and sends each packet to the server. It stores the sent packet in the dictionary along with its sequence number. The method is scheduled to be called repeatedly using `reactor.callLater` with a delay of 1 second.

7. Implement the `datagramReceived` method in the client class. This method is called when an acknowledgment (ACK) is received from the server. It checks if the ACK number corresponds to a sent packet and removes it from the dictionary.

8. Implement the `connectionLost` method in the client class. This method is called when the client's connection is lost. It closes the file and stops the reactor.

9. Start the server by listening on UDP port 8000 using `reactor.listenUDP`.

10. Create an instance of the client class and start the client by listening on a random UDP port using `reactor.listenUDP`. Provide the filename of the file to be sent.

11. Run the reactor using `reactor.run()` to start the event loop and handle network events.

CODE:

Using TCP:

```
from twisted.internet import protocol, reactor

class SelectiveRepeatProtocol(protocol.Protocol):
    def __init__(self):
        self.window_size = 4  # Size of the sender and receiver window
        self.send_base = 0    # Sequence number of the oldest unacknowledged packet
        self.send_next = 0    # Sequence number of the next packet to be sent
        self.recv_base = 0    # Sequence number of the oldest unprocessed packet

        self.buffer = ['Packet0', 'Packet1', 'Packet2', 'Packet3']  # Example packet buffer

    def connectionMade(self):
        print("Connection established.")
        self.sendPackets()

    def dataReceived(self, data):
        # Simulate packet loss
        if data != "ACK":
            print("Packet lost!")
            return

        print("ACK received.")
        self.send_base += 1
        self.send_next += 1
        self.sendPackets()

    def sendPackets(self):
        while self.send_next < self.send_base + self.window_size and self.send_next <
len(self.buffer):
            packet = self.buffer[self.send_next]
            print("Sending packet:", packet)
            self.transport.write(packet.encode())
            self.send_next += 1

    def connectionLost(self, reason):
        print("Connection lost.")

class SelectiveRepeatFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return SelectiveRepeatProtocol()
```

```
reactor.listenTCP(8000, SelectiveRepeatFactory())
reactor.run()
```

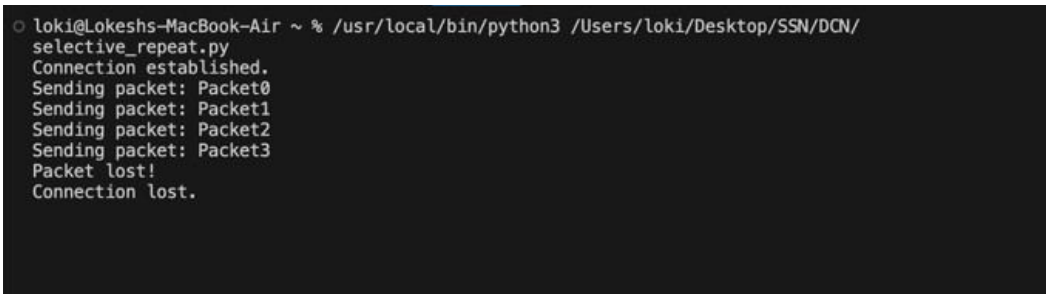
Using UDP:

```
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor

class SelectiveRepeatServer(DatagramProtocol):
    def __init__(self):
        self.expected_seq_num = 0
        self.buffer = {}

    def datagramReceived(self, datagram, address):
        seq_num, data = datagram.split(b':', 1)
        seq_num = int(seq_num)

        if seq_num == self.expected_seq_num:
            print("Received packet with sequence number:", seq_num)
            self.transport.write(str(seq_num).encode(), address)
            self.expected_seq_num += 1

            while self.expected_seq_num in self.buffer:
                packet = self.buffer.pop(self.expected_seq_num)
                print("Delivering buffered packet with sequence number:",
self.expected_seq_num)
                self.transport.write(str(self.expected_seq_num).encode(), address)
                self.expected_seq_num += 1

        else:
            print("Received out-of-order packet with sequence number:", seq_num)
            self.buffer[seq_num] = datagram

class SelectiveRepeatClient(DatagramProtocol):
    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename, 'rb')
        self.seq_num = 0
        self.window_size = 4
        self.packets = {}

    def startProtocol(self):
        self.sendPackets()

    def sendPackets(self):
        for i in range(self.window_size):
            data = self.file.read(1024)
            if not data:
                break

            packet = str(self.seq_num).encode() + b':' + data
            self.transport.write(packet, ('127.0.0.1', 8000))
```

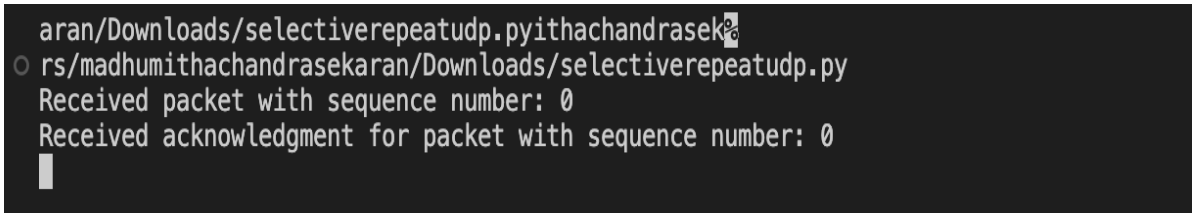
```
            self.packets[self.seq_num] = packet
            self.seq_num += 1

        reactor.callLater(1, self.sendPackets)

    def datagramReceived(self, data, address):
        ack_num = int(data.decode())

        if ack_num in self.packets:
            print("Received acknowledgment for packet with sequence number:", ack_num)
            del self.packets[ack_num]
        else:
            print("Received duplicate acknowledgment for packet with sequence number:",
ack_num)

    def connectionLost(self, reason):
        self.file.close()
        reactor.stop()

# Start the server
reactor.listenUDP(8000, SelectiveRepeatServer())

# Start the client
client = SelectiveRepeatClient("/Users/madhumithachandrasekaran/Desktop/file.txt")
reactor.listenUDP(0, client)
reactor.run()
```

SAMPLE INPUT/OUTPUT:

1) **Using TCP:**



2) **Using UDP:**



RESULT:

Thus, the Stop and Wait and Sliding Window protocols (Go-Back-N and Selective Repeat) have been implemented and tested successfully using Twisted Python.

Implementation of HTTP Web client program to download a web page using TCP sockets using Twisted Python

Ex. No. 11
Date:

PROBLEM STATEMENT:

To write a program for creating socket for HTTP for web page upload and download.

PROBLEM DESCRIPTION:

The goal is to implement a client-server system that allows users to upload web pages from the client-side and download them from the server-side using the HTTP protocol using socket programming in Python. The client and server should communicate using the HTTP protocol and adhere to the HTTP request-response model. The client should be able to select a web page file from their local system and send it to the server for uploading. The server should receive the web page file, save it on the server-side, and send a response indicating the success or failure of the upload. The client should be able to download web pages from the server.

CODE:

SERVER:

```
import socket
HOST = 'localhost'
PORT = 8080


def handle_upload(connection):
    with open('uploaded.html', 'wb') as file:
        while True:
            data = connection.recv(1024)
            if not data:
                break
            file.write(data)
    connection.sendall(b'Webpage uploaded successfully.')
    connection.close()

def handle_download(connection):
    with open('example.html', 'rb') as file:
        data = file.read()
    connection.sendall(data)
```

```
    connection.close()

def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((HOST, PORT))
    server_socket.listen(1)
    print(f'Server started. Listening on {HOST}:{PORT}...')
    while True:
        connection, address = server_socket.accept()
        request = connection.recv(1024).decode()
        if request == 'UPLOAD':

          handle_upload(connection)
        elif request == 'DOWNLOAD':
            handle_download(connection)
        connection.close()
start_server()
```

CLIENT:

```
import socket
HOST = 'localhost'
PORT = 8080

def upload_webpage(filename):
    with open(filename, 'rb') as file:
        data = file.read()
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
        client_socket.connect((HOST, PORT))
        client_socket.sendall(b'UPLOAD')
        client_socket.sendall(data)
        response = client_socket.recv(1024).decode()
        print(response)
        print("Webpage uploaded successfully..")

def download_webpage():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
        client_socket.connect((HOST, PORT))

       client_socket.sendall(b'DOWNLOAD')
        print("Webpage downloaded successfully..")
        data = client_socket.recv(1024)
        print(data.decode())
upload_webpage('example.html')
download_webpage()
```

Example.html :

```
<!DOCTYPE html>
```

```
<html>
<head>
   <title>Example Webpage</title>
</head>
<body>
   <h1>Welcome to the Example Webpage!</h1>
   <p>This is a sample webpage for testing purposes.</p>
</body>
</html>
```

SAMPLE INPUT/OUTPUT:

Server:

```
C:\Users\SSN\Desktop>python3 server.py
Server started. Listening on localhost:8080...
```

Client:

```
C:\Users\SSN\Desktop>python3 client.py

Webpage uploaded successfully..
Webpage downloaded successfully..

<!DOCTYPE html>
<html>
<head>
    <title>Example Webpage</title>
</head>
<body>
    <h1>Welcome to the Example Webpage!</h1>
    <p>This is a sample webpage for testing purposes.</p>
</body>
</html>
```

DOWNLOADED WEB PAGE:

Welcome to the Example Webpage!

This is a sample webpage for testing purposes.

RESULT:

Thus, the HTTP protocol for webpage upload and download has been implemented using socket programming in python.

Implementation of Remote Procedure Call (RPC) using Twisted Python

Ex. No. 12
Date:

PROBLEM STATEMENT:

To write a program to implement Remote Procedure Call (RPC) for Client-Server Communication using Twisted Python.

PROBLEM DESCRIPTION:

RPC allows a program to call a procedure or function on a remote computer as if it were a local function call. This enables distributed computing and facilitates communication between different processes or systems. It should provide the necessary functionality to enable remote procedure calls, handling the communication between the client and the server. The RPC system should handle the underlying network communication and provide a seamless interface for remote procedure invocation.

1. **Client-Server Communication:** Implement a reliable communication channel between the client and server. Establish a connection or session between the client and server.

2. **Interface Definition Language (IDL):** Design a language-independent Interface Definition Language (IDL) to specify server interfaces and operations.
   Ensure client and server can agree on procedure signatures.

3. **Stub Generation:** Develop client-side stubs and server-side skeletons based on the IDL. Client stubs package procedure parameters, transmit them to the server, and handle responses. Server skeletons receive procedure calls, unpack parameters, and invoke server-side procedures.

4. **Marshalling and Unmarshalling:** Implement mechanisms for marshalling (serialization) and unmarshalling (deserialization) of procedure parameters and return values. Convert parameters into a suitable format for network transmission and reconstruct on the server side.

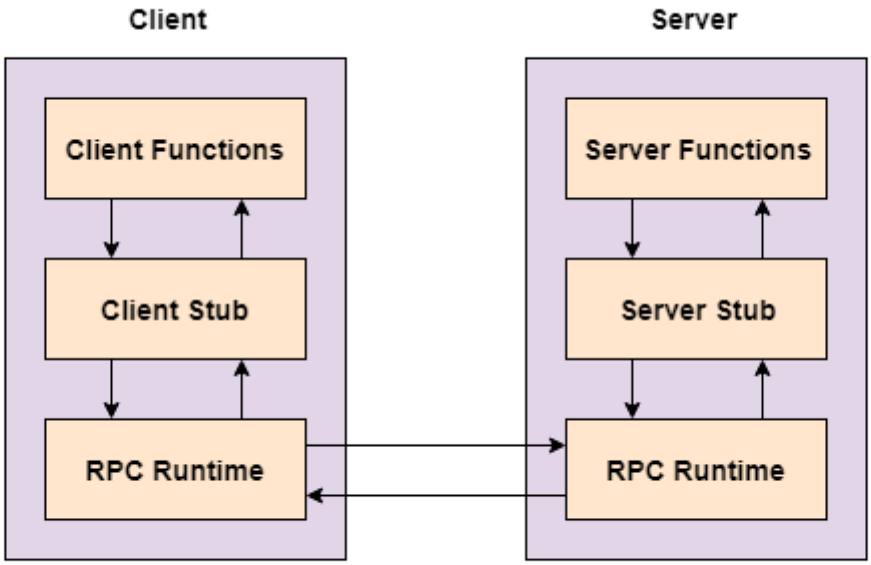
5. **Procedure Invocation and Execution:** Enable the client to invoke remote procedures by calling client stubs. Server skeletons receive procedure calls, unpack parameters, and execute server-side procedures.

PROCEDURE:

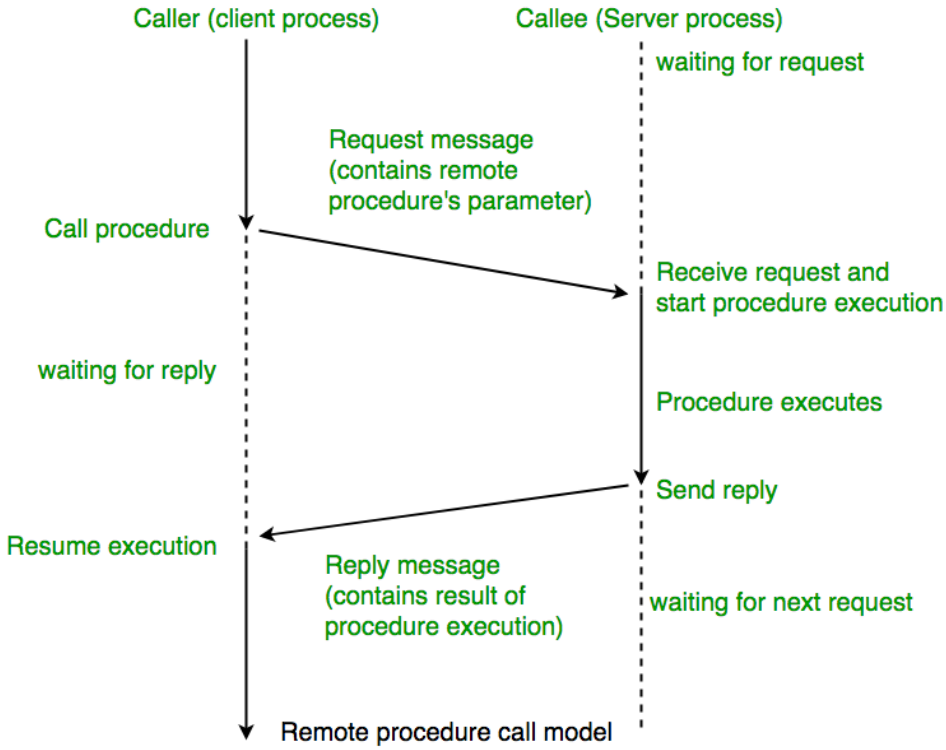
- In RPC, the sender makes a request in the form of a procedure, function, or method call. RPC translates these calls into requests sent over the network to the intended destination.
- The RPC recipient then processes the request based on the procedure name and argument list, sending a response to the sender when complete.
- The process is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters.

- The remote server sends a response to the client, and the application continues its process.
- While the server is processing the call, the client is blocked, it waits until the server has finished processing before resuming execution.

A diagram that demonstrates this is as follows −



When making a Remote Procedure Call:



CODE:

SERVER:

```
from twisted.spread import pb
from twisted.internet import reactor

class MyService(pb.Root):
    def remote_add(self, x, y):
        print("ADDITION:\n",x+y)
        return x + y

    def remote_subtract(self, x, y):
        print("SUBTRACTION:\n",x-y)
        return x - y

    def remote_multiply(self, x, y):
        print("MULTIPLICATION:\n",x*y)
        return x * y

    def remote_divide(self, x, y):

        if y != 0:
            print("DIVISION:\n",x/y)
            return x / y
        else:
            raise ValueError("Cannot divide by zero.")

service = MyService()

factory = pb.PBServerFactory(service)

reactor.listenTCP(6473, factory)
reactor.run()
```

CLIENT:

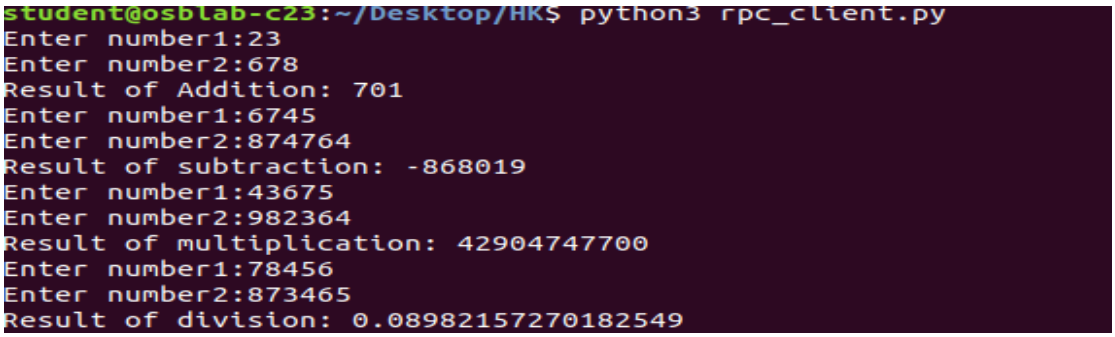
```
from twisted.spread import pb
from twisted.internet import reactor

def add_handle_result(result):
    print("Result of Addition:", result)

def sub_handle_result(result):
    print("Result of subtraction:", result)

def mul_handle_result(result):
    print("Result of multiplication:", result)

def div_handle_result(result):
    print("Result of division:", result)
```

```
def connection_error(err):
    print("Connection error:",err)
    reactor.stop()


def connect():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 6473, factory)
    d = factory.getRootObject()

    d.addCallback(lambda obj: obj.callRemote("add",int(input("Enter number1:")) ,int(input("Enter
number2:"))))

    d.addCallback(add_handle_result)
    d.addCallback(lambda _: factory.getRootObject())
    d.addCallback(lambda obj: obj.callRemote("subtract",int(input("Enter
number1:")),int(input("Enter number2:"))))

    d.addCallback(sub_handle_result)
    d.addCallback(lambda _: factory.getRootObject())
    d.addCallback(lambda obj: obj.callRemote("multiply",int(input("Enter
number1:")),int(input("Enter number2:"))))

    d.addCallback(mul_handle_result)
    d.addCallback(lambda _: factory.getRootObject())
    d.addCallback(lambda obj: obj.callRemote("divide",int(input("Enter
number1:")),int(input("Enter number2:"))))

    d.addCallback(div_handle_result)

reactor.callWhenRunning(connect)
reactor.run()
```

SAMPLE INPUT/OUTPUT:

CLIENT SIDE:

CASE 1:



CASE 2:

```
student@osblab-c23:~/Desktop/HK$ python3 rpc_client.py
Enter number1:627435
Enter number2:325
Result of Addition: 627760
Enter number1:124
Enter number2:546
Result of subtraction: -422
Enter number1:234
Enter number2:67
Result of multiplication: 15678
Enter number1:465
Enter number2:0
Unhandled error in Deferred:

Traceback from remote host -- builtins.ValueError: Cannot divide by zero.
```

SERVER SIDE:

CASE 1:

```
student@osblab-c23:~/Desktop/HK$ python3 rpc_server.py
ADDITION:
 701
SUBTRACTION:
 -868019
MULTIPLICATION:
 42904747700
DIVISION:
 0.08982157270182549
```

CASE 2:

```
ADDITION:
 627760
SUBTRACTION:
 -422
MULTIPLICATION:
 15678
Peer will receive following PB traceback:
Unhandled Error
Traceback (most recent call last):
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/banana.py", line 176, in gotItem
    self.callExpressionReceived(item)
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/banana.py", line 137, in callExpressionReceived
    self.expressionReceived(obj)
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/pb.py", line 607, in expressionReceived
    method(*sexp[1:])
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/pb.py", line 1006, in proto_message
    netKw,
--- <exception caught here> ---
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/pb.py", line 1053, in _recvMessage
    netResult = object.remoteMessageReceived(self, message, netArgs, netKw)
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/flavors.py", line 131, in remoteMessageReceived
    state = method(*args, **kw)
  File "rpc_server.py", line 23, in remote_divide
    raise ValueError("Cannot divide by zero.")
builtins.ValueError: Cannot divide by zero.
```

RESULT: Thus, the RPC using Twisted and Perspective Broker in Python has been implemented and tested successfully.

Implementation of Subnetting using Twisted Python

Ex. No. 13
Date:

PROBLEM STATEMENT:

To write a program to implement subnetting and find the number of hosts using Twisted Python.

PROBLEM DESCRIPTION:

A subnet is a smaller network, also referred to as a sub network. An IP network is logically divided into several smaller network components by subnets. A subnet is used to divide a large network into a number of smaller, linked networks, which helps to minimize traffic. Subnets reduce the need for traffic to use unnecessary routes, which speeds up the network. To help with the lack of IP addresses on the internet, subnets were developed.

Advantages of Subnetting

- Subnetting is used to decrease the presence of Internet Protocol (IP) range.
- Subnets helps in stopping the devices or gadgets from occupying the whole network, only allowing the hosts to control which kind of user can have access to the important information. Simply, we can tell that network is safe just because of the subnetting concept.
- Subnetting concept increases the performance of the total network by deleting the repeated traffic causing errors.
- We can convert the whole big network into smaller networks by using the concept of subnetting as discussed earlier.

Example

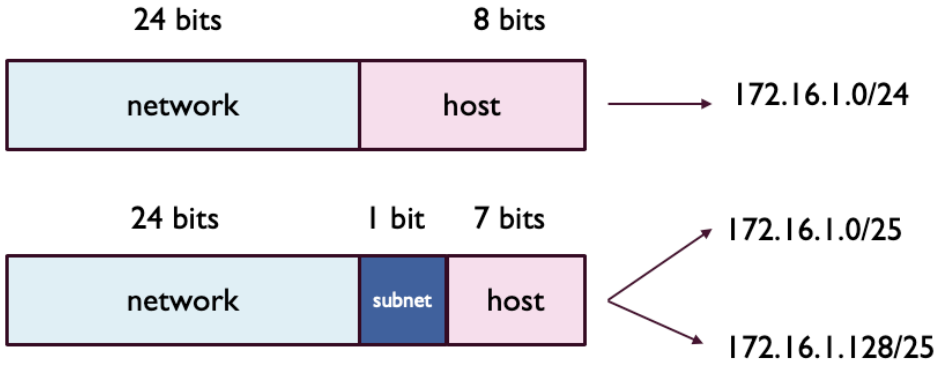
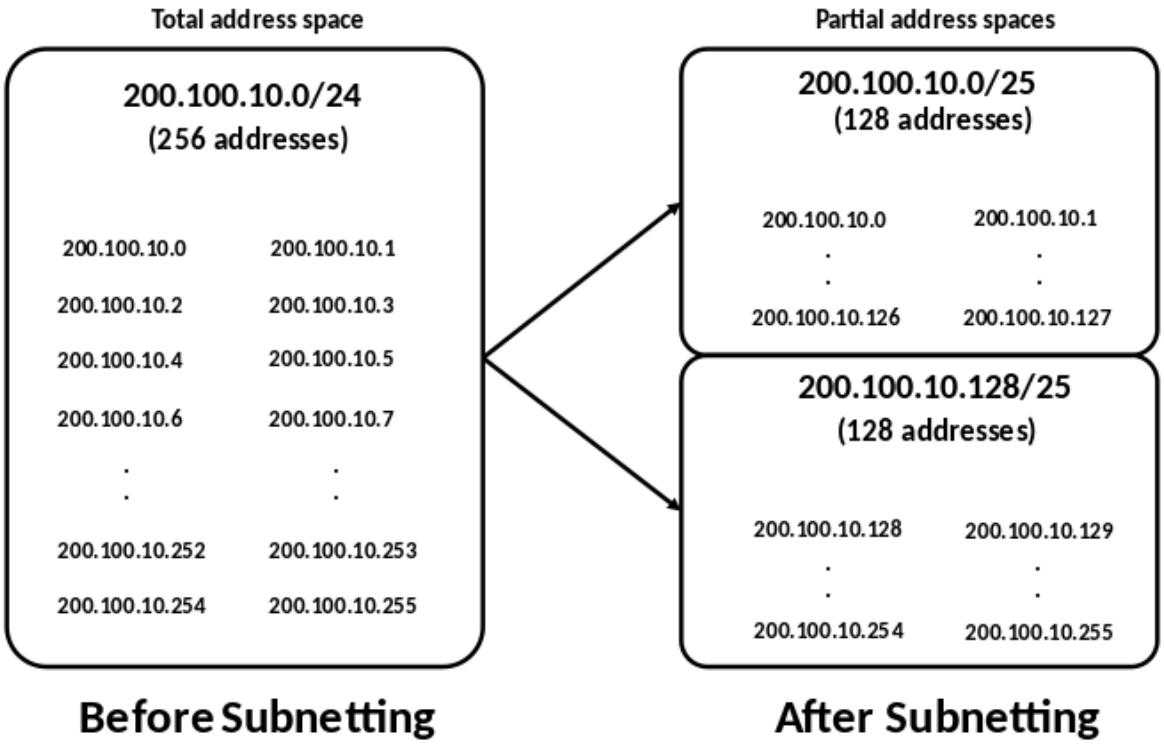


Illustration of Subnetting:

Subnetting is the process of dividing a network into smaller subnetworks or subnets. It involves manipulating the IP address and subnet mask to create distinct network addresses for each subnet.



ALGORITHM:

Here's an algorithm for subnetting a network:

Step 1: Determine the network requirements

- Determine the total number of required subnets.
- Determine the number of hosts needed per subnet.
- Choose an appropriate IP address range.

Step 2: Calculate the number of bits needed for subnetting

- Determine the number of bits required to represent the total number of subnets (2^N >= Total Subnets).
- Determine the number of bits required to represent the number of hosts per subnet (2^N >= Hosts Per Subnet + 2).

Step 3: Choose a suitable subnet mask

- Convert the number of subnet bits to a subnet mask by setting the leftmost N bits to 1 and the remaining bits to 0.
- Common subnet masks include /24 (255.255.255.0), /16 (255.255.0.0), etc.

Step 4: Divide the IP address range into subnets

- Start with the given IP address range and subnet mask.
- Increment the network address by the number of hosts per subnet for each subnet.
- Assign the first usable host address and the last usable host address for each subnet.

Step 5: Determine the broadcast address for each subnet

- Calculate the broadcast address by taking the network address of the subnet and setting all host bits to 1.

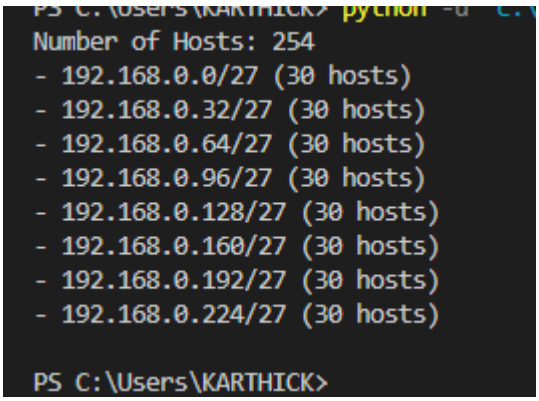
Step 6: Repeat steps 4 and 5 until all subnets are created.

CODE:

```
from twisted.internet import reactor
from ipaddress import ip_network, ip_interface

def compute_number_of_hosts_and_subnets(network_id, subnet_mask, subnet_bits):
    try:
        network = ip_network(network_id + '/' + subnet_mask)
        num_hosts = network.num_addresses - 2  # Subtracting network and broadcast
addresses

        subnets = list(network.subnets(new_prefix=subnet_bits))

        return num_hosts, subnets
    except ValueError as e:
        return None, None

def main():
    network_id = '192.168.0.0'  # Example network ID
    subnet_mask = '255.255.255.0'  # Example subnet mask
    subnet_bits = 27  # Number of bits to be borrowed for subnets

    num_hosts, subnets = compute_number_of_hosts_and_subnets(network_id,
subnet_mask, subnet_bits)

    if num_hosts is not None and subnets is not None:
        print("Number of Hosts:", num_hosts)
        print("Subnets:")
        for subnet in subnets:
            subnet_hosts = subnet.num_addresses - 2  # Subtracting network and
broadcast addresses per subnet
            print(f"- {subnet} ({subnet_hosts} hosts)")
        print("")
    else:
```

```
        print("Error occurred during computation.")

if __name__ == '__main__':
    main()
```

SAMPLE INPUT/OUTPUT:



RESULT: Thus, the subnetting has been implemented and tested successfully using Twisted Python.

CODE:

SERVER:

```
from twisted.spread import pb
from twisted.internet import reactor

class MyService(pb.Root):
    def remote_add(self, x, y):
        print("ADDITION:\n",x+y)
        return x + y

    def remote_subtract(self, x, y):
        print("SUBTRACTION:\n",x-y)
        return x - y

    def remote_multiply(self, x, y):
        print("MULTIPLICATION:\n",x*y)
        return x * y

    def remote_divide(self, x, y):

        if y != 0:
            print("DIVISION:\n",x/y)
            return x / y
        else:
            raise ValueError("Cannot divide by zero.")

service = MyService()

factory = pb.PBServerFactory(service)

reactor.listenTCP(6473, factory)
reactor.run()
```

CLIENT:

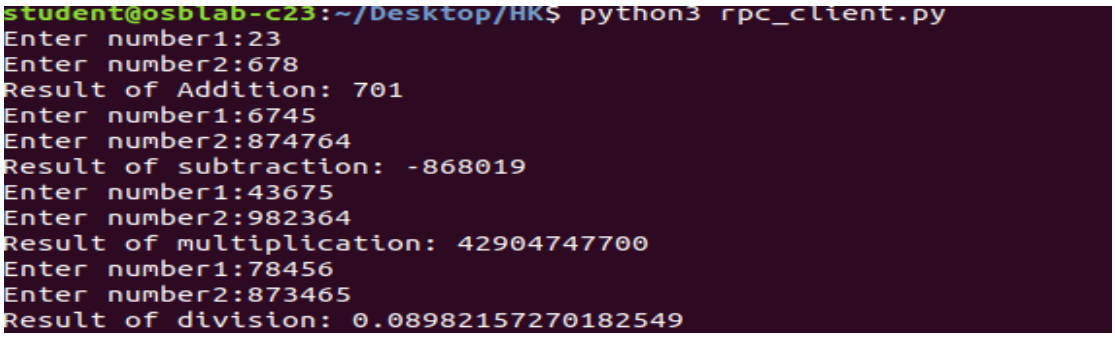
```
from twisted.spread import pb
from twisted.internet import reactor

def add_handle_result(result):
    print("Result of Addition:", result)

def sub_handle_result(result):
    print("Result of subtraction:", result)

def mul_handle_result(result):
    print("Result of multiplication:", result)

def div_handle_result(result):
    print("Result of division:", result)
```

```
def connection_error(err):
    print("Connection error:",err)
    reactor.stop()


def connect():
    factory = pb.PBClientFactory()
    reactor.connectTCP("localhost", 6473, factory)
    d = factory.getRootObject()

    d.addCallback(lambda obj: obj.callRemote("add",int(input("Enter number1:")) ,int(input("Enter
number2:"))))

    d.addCallback(add_handle_result)
    d.addCallback(lambda _: factory.getRootObject())
    d.addCallback(lambda obj: obj.callRemote("subtract",int(input("Enter
number1:")),int(input("Enter number2:"))))

    d.addCallback(sub_handle_result)
    d.addCallback(lambda _: factory.getRootObject())
    d.addCallback(lambda obj: obj.callRemote("multiply",int(input("Enter
number1:")),int(input("Enter number2:"))))

    d.addCallback(mul_handle_result)
    d.addCallback(lambda _: factory.getRootObject())
    d.addCallback(lambda obj: obj.callRemote("divide",int(input("Enter
number1:")),int(input("Enter number2:"))))

    d.addCallback(div_handle_result)

reactor.callWhenRunning(connect)
reactor.run()
```

SAMPLE INPUT/OUTPUT:

CLIENT SIDE:

CASE 1:



CASE 2:

```
student@osblab-c23:~/Desktop/HK$ python3 rpc_client.py
Enter number1:627435
Enter number2:325
Result of Addition: 627760
Enter number1:124
Enter number2:546
Result of subtraction: -422
Enter number1:234
Enter number2:67
Result of multiplication: 15678
Enter number1:465
Enter number2:0
Unhandled error in Deferred:

Traceback from remote host -- builtins.ValueError: Cannot divide by zero.
```

SERVER SIDE:

CASE 1:

```
student@osblab-c23:~/Desktop/HK$ python3 rpc_server.py
ADDITION:
 701
SUBTRACTION:
 -868019
MULTIPLICATION:
 42904747700
DIVISION:
 0.08982157270182549
```

CASE 2:

```
ADDITION:
 627760
SUBTRACTION:
 -422
MULTIPLICATION:
 15678
Peer will receive following PB traceback:
Unhandled Error
Traceback (most recent call last):
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/banana.py", line 176, in gotItem
    self.callExpressionReceived(item)
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/banana.py", line 137, in callExpressionReceived
    self.expressionReceived(obj)
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/pb.py", line 607, in expressionReceived
    method(*sexp[1:])
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/pb.py", line 1006, in proto_message
    netKw,
--- <exception caught here> ---
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/pb.py", line 1053, in _recvMessage
    netResult = object.remoteMessageReceived(self, message, netArgs, netKw)
  File "/usr/local/lib/python3.6/dist-packages/twisted/spread/flavors.py", line 131, in remoteMessageReceived
    state = method(*args, **kw)
  File "rpc_server.py", line 23, in remote_divide
    raise ValueError("Cannot divide by zero.")
builtins.ValueError: Cannot divide by zero.
```

RESULT: Thus, the RPC using Twisted and Perspective Broker in Python has been implemented and tested successfully.

Applications using TCP and UDP Sockets like
a) DNS b) SNMP c) SMTP

Ex. No. 14 a)
Date:

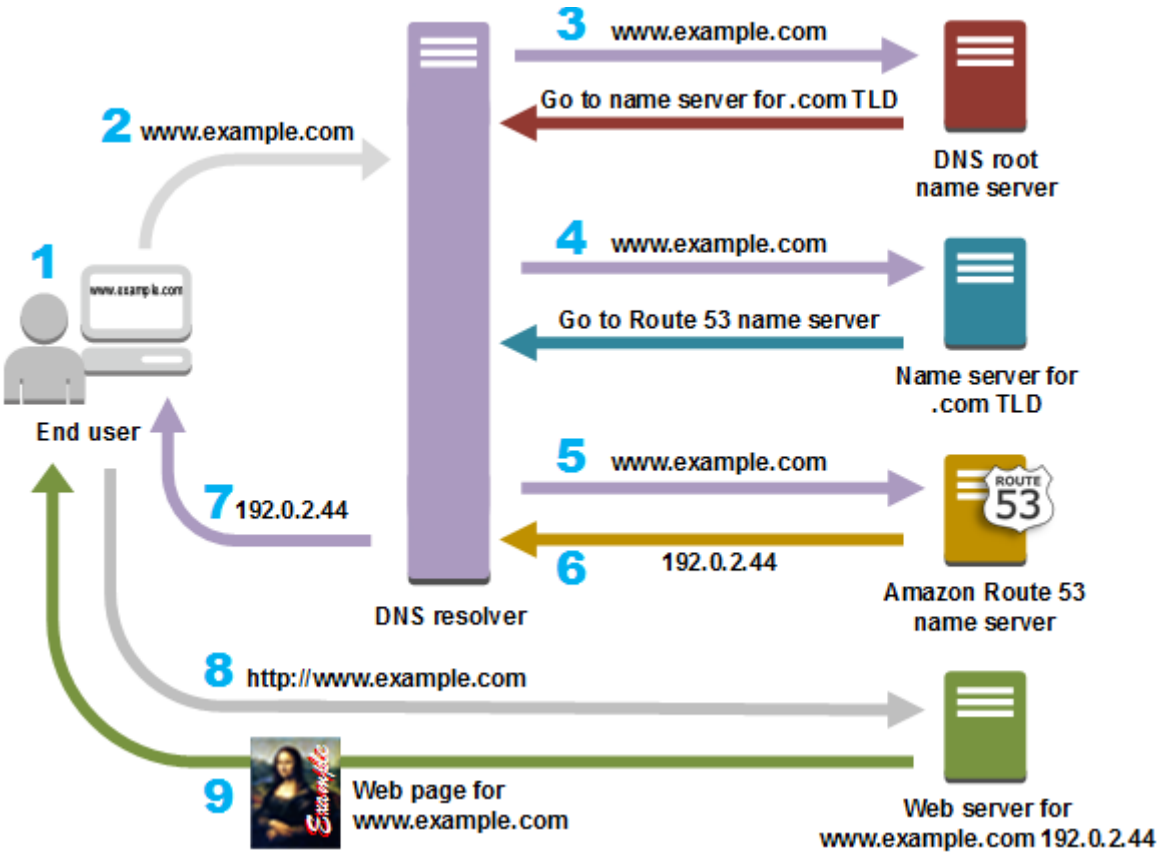
PROBLEM STATEMENT:

To write a program to implement Domain Name System (DNS) using UDP sockets for resolving the domain name into IP addresses using Python.

PROBLEM DESCRIPTION:

The DNS server listens for the DNS query from clients and respondswith the corresponding IP address if it is present in the dns_table dictionary. If the IP address is not found it responds not found.

How Does DNS Route Traffic to Your Web Application?
The following diagram gives an overview of how recursive and authoritative DNS services work together to route an end user to your website or application.



1. A user opens a web browser, enters www.example.com in the address bar, and presses Enter.

2. The request for www.example.com is routed to a DNS resolver, which is typically managed by the user's Internet service provider (ISP), such as a cable Internet provider, a DSL broadband provider, or a corporate network.

3. The DNS resolver for the ISP forwards the request for www.example.com to a DNS root name server.

4. The DNS resolver for the ISP forwards the request for www.example.com again, this time to one of the TLD name servers for .com domains. The name server for .com domains responds to the request with the names of the four Amazon Route 53 name servers that are associated with the example.com domain.

5. The DNS resolver for the ISP chooses an Amazon Route 53 name server and forwards the request for www.example.com to that name server.

6. The Amazon Route 53 name server looks in the example.com hosted zone for the www.example.com record, gets the associated value, such as the IP address for a web server, 192.0.2.44, and returns the IP address to the DNS resolver.

7. The DNS resolver for the ISP finally has the IP address that the user needs. The resolver returns that value to the web browser. The DNS resolver also caches (stores) the IP address for example.com for an amount of time that you specify so that it can respond more quickly the next time someone browses to example.com. For more information, see time to live (TTL).

8. The web browser sends a request for www.example.com to the IP address that it got from the DNS resolver. This is where your content is, for example, a web server running on an Amazon EC2 instance or an Amazon S3 bucket that's configured as a website endpoint.

9. The web server or other resource at 192.0.2.44 returns the web page for www.example.com to the web browser, and the web browser displays the page.

Algorithm:

Step1: Import the socket module.

Step 2: Define a dictionary dns_table with domain names as keysand corresponding IP addresses as values.

Step 3: Create a UDP socket using socket.socket.

Step 4: Bind the socket to the local address "127.0.0.1" and port1235.

Step 5: Start an infinite loop to continuously receive and respond to DNS queries

Step 6: Receive data and the client's address using s.recvfrom(1024).The buffer size is set to 1024 bytes.

Step7: Decode the received data into a string.

Step 8: Look up the received domain name in the dns_table using dns_table.get(data, "NOT Found!"). If the domain name is found, it returns the corresponding IP address; otherwise, it returns the string"NOT Found!".

Step 9: Encode the IP address string into bytes.

Step 10: Send the IP address back to the client using s.sendto(ip,address), where ip is the encoded IP address and address is the client's address.

Step 11: Repeat the loop to continue receiving and responding toDNS queries.

The DNS client:

Step1: Get the hostname using socket.gethostname() and assign it tothe variable hostname.

Step 2: Set the IP address (ipaddr) to "127.0.0.1" (localhost) in thisexample. You can change it to the desired IP address of the DNS server.

Step 3: Create a UDP socket using socket.socket(socket.AF_INET, socket.SOCK_DGRAM).

Step 4: Create a tuple addr with the IP address and port number (1234in this example) of the DNS server.

Step 5: Initialize the variable c to "Y" to enter the loop.

Step 6: Start a loop to allow the user to make multiple DNS queries.

Step 7: Ask the user to enter a domain name for which the IP addressis needed using input(). Store it in the variable req_domain.

Step 8: Send the encoded domain name to the server usings.sendto(req_domain.encode(), addr).

Step 9: Receive the response data and the server's address usings.recvfrom(1024). The buffer size is set to 1024 bytes.

Step 10: Decode the received data into a string and remove anyleading/trailing whitespaces.

Step 11: Print the domain name and the corresponding IP address tothe console using print().

Step 12: Close the socket using s.close().

CODE:

SERVER:

```
import socket
dns_table={"www.google.com":"192'165'1'1","www.youtube.com":"192.165.1.2",
        "www.python.org":"192.165.1.3",
        "www.aurcc.ac.in":"192.165.1.5",
```

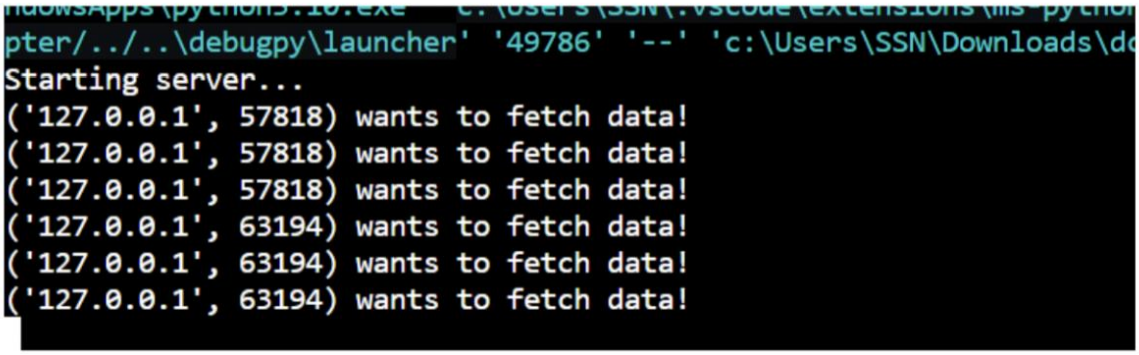
```
        "www.amazon.com":"192.165.1.5"}
s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)print("Starting server...")
s.bind(("127.0.0.1",1235))
while True:
    data,address=s.recvfrom(1024) print(f"{address} wants
    to fetch data!")data=data.decode()
ip=dns_table.get(data,"NOT Found!").encode()send=s.sendto(ip,address)
```

CLIENT:

```
import socket
hostname=socket.gethostname()
ipaddr="127.0.0.1"
s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)addr=(ipaddr,1234)
c="Y"
while c.upper()=="Y":
   req_domain=input("enter domain name for which the ip isneeded:")
    send=s.sendto(req_domain.encode(),addr)data,ad-
    dress=s.recvfrom(1024)
    reply_ip=data.decode().strip()
    print(f"the ip for the domain name {req_domain}:{reply_ip}")c=(input("continue?
    (y/n)"))
s.close()
```

SAMPLE INPUT/OUTPUT:

SERVER SIDE:



CLIENT SIDE:

```
pter/../..\debugpy\launcher' '49888' '--' 'c:\Users\SSN\Downloads\dcn_lap\DNS_client.py'
enter domain name for which the ip is needed:www.facebook.com
the ip for the domain name www.facebook.com:NOT Found!
continue? (y/n)y
enter domain name for which the ip is needed:www.google.com
the ip for the domain name www.google.com:192'165'1'1
continue? (y/n)y
enter domain name for which the ip is needed:www.amazon.com
the ip for the domain name www.amazon.com:192.165.1.5
continue? (y/n)n
```

RESULT: Hence using the DNS, the domain name www.google.com is resolved into an IP address 192.165.1.1

Applications using TCP and UDP Sockets like a) DNS b) SNMP c) SMTP

Ex. No. 14b)
Date:

PROBLEM STATEMENT:

To write a program to implement a Simple Network Management Protocol (SNMP) server and client using Twisted framework to enable communication between network devices for managing and monitoring purposes.

PROBLEM DESCRIPTION:

Simple Network Management Protocol (SNMP) is an application-layer protocol for monitoring and managing network devices on a local area network (LAN) or wide area network (WAN).

The purpose of SNMP is to provide network devices, such as routers, servers and printers, with a common language for sharing information with a network management system (NMS).

SNMP's client-server architecture has the three following components:

1. an SNMP manager;
2. an SNMP agent; and
3. a management information base (MIB).

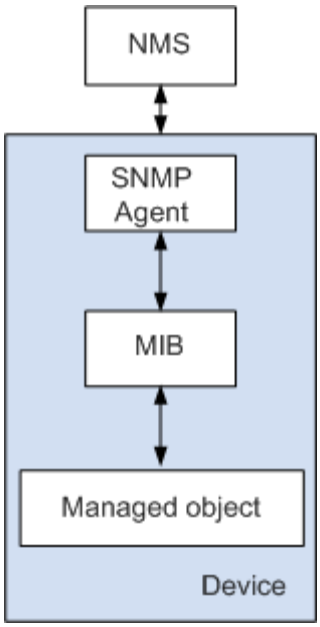
The SNMP manager acts as the client, the SNMP agent acts as the server and the MIB acts as the server's database. When the SNMP manager asks the agent a question, the agent uses the MIB to supply the answer.

Components of SNMP

There are four main components in an SNMP-managed network.

- ***SNMP agent***
  - Agent software runs on the hardware or service being monitored, collecting data about disk space, bandwidth use and other important network performance metrics.
- ***SNMP-managed network nodes***
  - These are the network devices and services upon which the agents run.

- ***SNMP manager***
    - The NMS is a software platform that functions as a centralized console to which agents feed information. The NMS will actively request agents to send updates at regular intervals.
- ***Management information base***
    - This MIB database is a text file (.mib) that itemizes and describes all objects on a particular device that can be queried or controlled using SNMP. Each MIB item is assigned an object identifier (OID).



Model of SNMP System

SNMP defines five types of messages: **GetRequest**, **GetNextRequest**, **SetRequest**, **GetResponse**, and **Trap**.

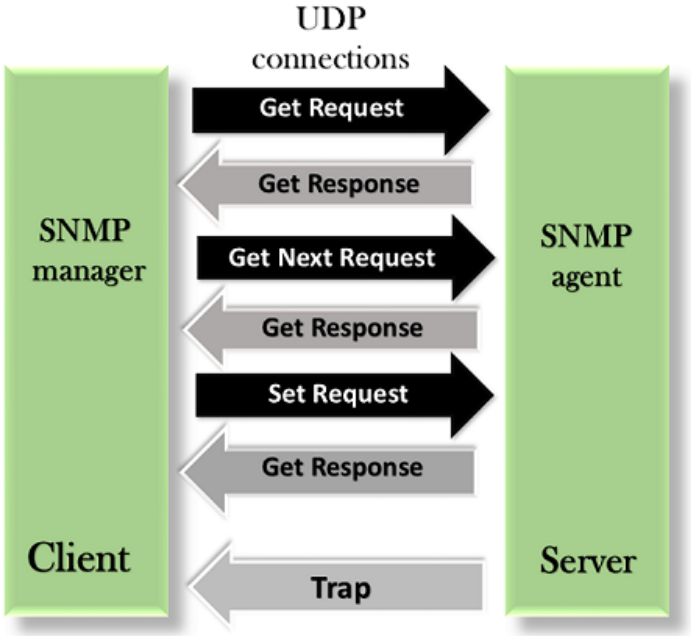


Illustration:

ALGORITHM:

Server Side

Step 1: Create a SNMP server using the Twisted framework.
Step 2: Listen for incoming SNMP requests on a specific port.
Step 3: When a request is received, extract the data from the request.
Step 4: Process the SNMP request data to determine the appropriate response.
Step 5: Prepare the response message based on the processed request data.
Step 6: Send the response message back to the client that made the request.

Client Side

Step 1: Create a SNMP client using the Twisted framework.
Step 2: Establish a connection with the SNMP server by specifying the server's IP addressand port number.
Step 3: Construct an SNMP request message.
Step 4: Send the SNMP request message to the server.
Step 5: Wait for the response from the server.
Step 6: Upon receiving the response, extract and process the response data as needed.

CODE:

SENDER SIDE:

```
zrom twisted.internet import reactor

from twisted.internet.protocol import DatagramProtocol
class SNMPProtocol(DatagramProtocol): def data-

    gramReceived(self, data, addr):

        print("Received data from {}: {}".format(addr, data))

        # Process the SNMP request here and prepare the responseresponse =

        "SNMP response"

        self.transport.write(response.encode(), addr)


def run_server():
```

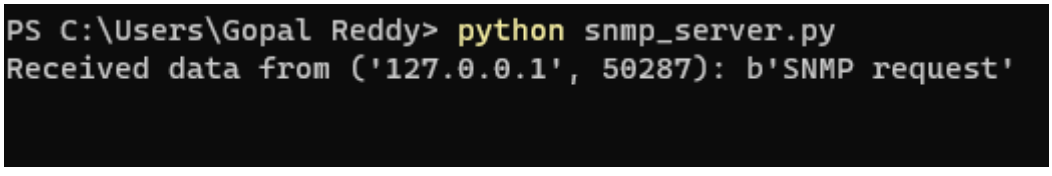
```
    reactor.listenUDP(161, SNMPProtocol())reactor.run()

run_server
```

RECEIVER SIDE:

```
from twisted.internet import reactor
from twisted.internet.protocol import DatagramProtocol


class SNMPClientProtocol(DatagramProtocol):def
    startProtocol(self):
        self.transport.connect('127.0.0.1', 161)self.sendRe-
        quest()


    def sendRequest(self):
        request = "SNMP request"
        self.transport.write(request.encode())


    def datagramReceived(self, data, addr):
        print("Received data from {}: {}".format(addr, data))# Pro-
        cess the SNMP response here


def run_client():
    reactor.listenUDP(0, SNMPClientProtocol())reac-
    tor.run()


run_client()
```

SAMPLE INPUT/OUTPUT:

SERVER SIDE



CLIENT SIDE

```
PS C:\Users\Gopal Reddy> python snmp_client.py
Received data from ('127.0.0.1', 161): b'SNMP response'
```

RESULT: Thus, the Simple Network Management Protocol (SNMP) using Twisted Python was implemented and tested successfully.

Applications using TCP and UDP Sockets like
a) DNS b) SNMP c) SMTP

Ex. No. 14c)
Date:

PROBLEM STATEMENT:

To write a program to implement Simple Mail Transfer Protocol (SMTP) using Twisted Python with an aim to develop a reliable email delivery system.

PROBLEM DESCRIPTION:

Email is emerging as one of the most valuable services on the internet today. Sending email is the most common and necessary requirement for most of the applications. Most internet systems use SMTP as a method to transfer mail from one user to another. SMTP is a push protocol and is used to send the mail whereas Post Office Protocol (POP) or Internet Message Access Protocol (IMAP) is used to retrieve those emails at the receiver's side.

SMTP is an application layer protocol. The client who wants to send the mail opens a TCP connection to the SMTP server and then sends the mail across the connection. The SMTP server is an always-on listening mode. As soon as it listens for a TCP connection from any client, the SMTP process initiates a connection through port 25. After successfully establishing a TCP connection the client process sends the mail instantly.

Components of SMTP

- **Mail User Agent (MUA):** It is a computer application that helps you in sending and retrieving mail. It is responsible for creating email messages for transfer to the mail transfer agent (MTA).
- **Mail Submission Agent (MSA):** It is a computer program that basically receives mail from a Mail User Agent (MUA) and interacts with the Mail Transfer Agent (MTA) for the transfer of the mail.
- **Mail Transfer Agent (MTA):** It is basically software that has the work to transfer mail from one system to another with the help of SMTP.
- **Mail Delivery Agent (MDA): A mail** Delivery agent or Local Delivery Agent is basically a system that helps in the delivery of mail to the local system.

Model of SMTP System

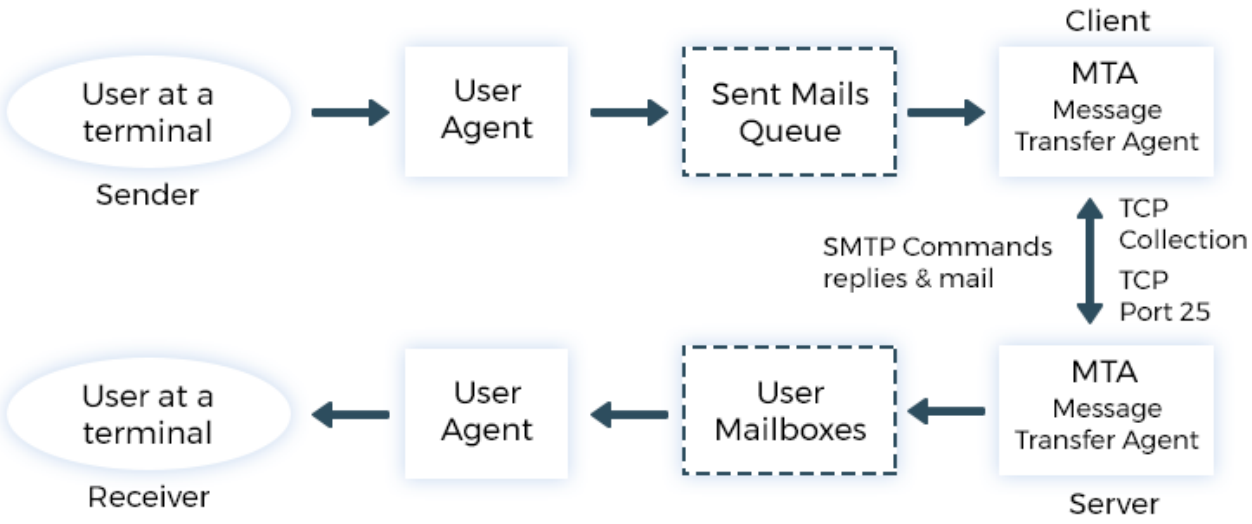
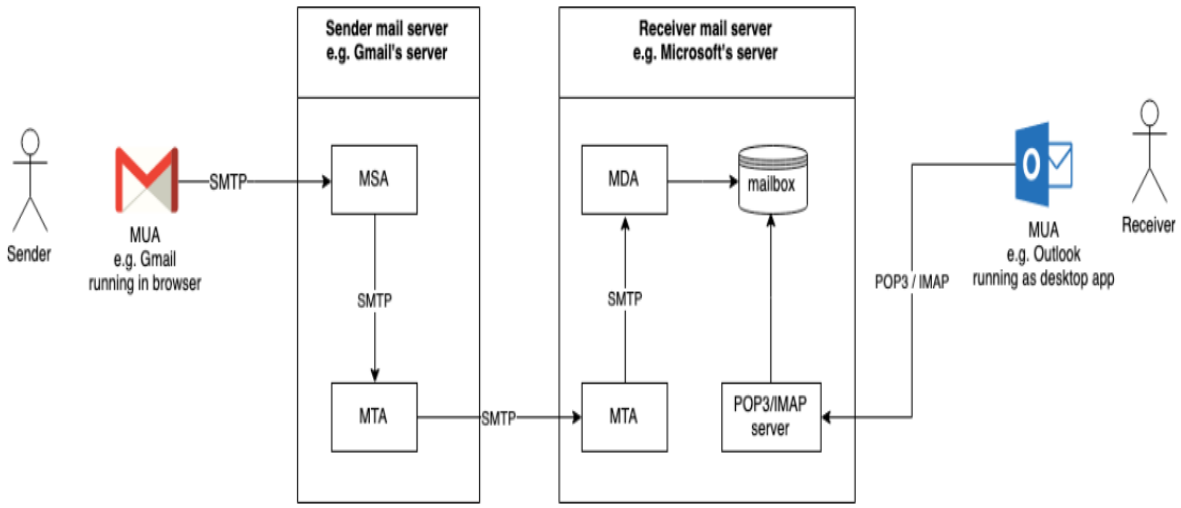


Illustration:



CODE:

SENDER SIDE:

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

def send_email(sender_email, sender_password, recipient_email, subject,message):
    # SMTP server configuration
    smtp_server = "smtp.gmail.com"
    smtp_port = 587   # Use the appropriate port for your SMTP server

    # Create a MIME multipart message
    msg = MIMEMultipart()
    msg['From'] = sender_email
    msg['To'] = recipient_email
    msg['Subject'] = subject

    # Add the message body
    msg.attach(MIMEText(message, 'plain'))

    try:
        # Connect to the SMTP server
        server = smtplib.SMTP(smtp_server, smtp_port)
        server.starttls()
        server.login(sender_email, sender_password)

        # Send the email
        server.sendmail(sender_email, recipient_email, msg.as_string())

        # Disconnect from the server
        server.quit()

        print('Email sent successfully!')except
    smtplib.SMTPException as e:
        print('Error sending email:', str(e))# Example
usage
sender_email = 'jai2110682@ssn.edu.in'
sender_password = '22Jan2022@ssn' recipient_email =
'harsh2110303@ssn.edu.in'
subject = 'successful implementation of SMTP protocol'
```

```
message = 'SMTP protocol was successfully implemented Harsh and Jai'

send_email(sender_email, sender_password, recipient_email, subject, message)
```

SAMPLE INPUT/OUTPUT:

RESULT: Thus, the Simple Mail Transfer Protocol (SMTP) using Twisted Python was implemented and tested successfully.

Study of Network simulator (NS-2)

Ex. No. 15a)
Date:

PROBLEM STATEMENT:

To study the use of network simulator (NS-2).

PROBLEM DESCRIPTION:

The given problem statement requires us to use Network Simulator like NS-2 to simulate the congestion control algorithms.

NS-2 is an open-source network simulation tool which allows us to simulate and analyse the behaviour of computer networks. It is primarily written in C++ and provides a TCL (Tool Command Language) scripting interface to configure and control the simulations.

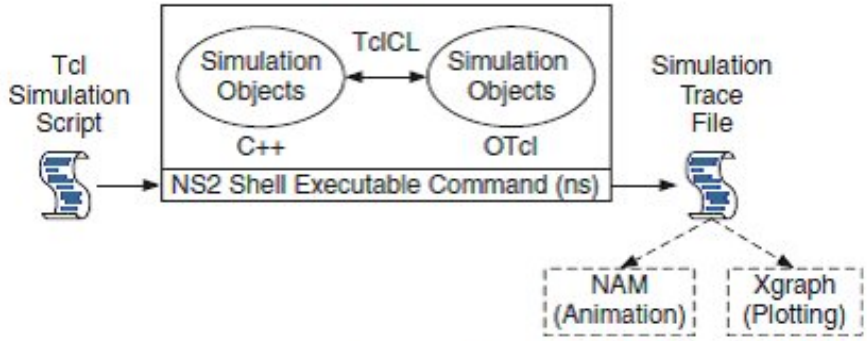
NAM (Network Animator) is a graphical visualization tool that works alongside NS-2 to provide a visual representation of the simulated network and allows users to observe the flow of packers.

Features of NS-2

- It is a discrete event simulator for networking research.
- It provides substantial support to simulate bunch of protocols like TCP, FTP, UDP, https and DSR.
- It simulates wired and wireless network.
- It is primarily Unix based.
- Uses TCL as its scripting language.
- Otcl: Object oriented support
- Tclcl: C++ and otcl linkage
- Discrete event scheduler

NS-2 consists of two key languages: C++ and Object-oriented Tool Command Language (OTcl). While the C++ defines the internal mechanism (i.e., a backend) of the simulation objects, the OTcl sets up simulation by assembling and configuring the objects as well as scheduling discrete events. The C++ and the OTcl are linked together using TclCL

BASIC ARCHITECTURE:



Basic architecture of NS.

PROCEDURE TO RUN TCL FILE:

- ✓ Install NS-2, NAM and TCL one after another
- ✓ Save a TCL file with the necessary algorithm implementation in desired destination
- ✓ Run Script using “ns filename_.tcl”

TCL FILE ALGORITHM:

- ✓ Start
- ✓ Create a Network Topology
- ✓ Enable Trace File Generation
- ✓ Create Nodes
- ✓ Establish links between nodes
- ✓ Set up traffic sources
- ✓ Implement Congestion Control Algorithm
- ✓ Set parameters for simulation
- ✓ Run the Simulation
- ✓ Close Trace File
- ✓ Launch NAM for visualisation
- ✓ Exit NS-2
- ✓ Stop

CODE:

```
# Create a network topology
set ns [new Simulator]

# Enable trace file generation
set tracefile [open "congestion_control.tr" w]
$ns trace-all $tracefile

# Create nodes
set node(0) [$ns node]
set node(1) [$ns node]

# Create links
```

```
$ns duplex-link $node(0) $node(1) 10Mb 10ms DropTail

# Set up traffic sources
set tcp [new Agent/TCP]
$tcp set class_ 2
$ns attach-agent $node(0) $tcp

set sink [new Agent/TCPSink]
$ns attach-agent $node(1) $sink

$ns connect $tcp $sink

set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns at 0.1 "$ftp start"

# Implement congestion control algorithm
$tcp set cong_algorithm NewReno

# Set simulation parameters
$ns duplex-link-op $node(0) $node(1) orient right-down
$ns color 1 Blue
$ns color 2 Red
$ns at 10.0 "$ns finish"

# Run the simulation
$ns run

# Close the trace file
$ns flush-trace
close $tracefile

# Launch Nam for visualization
set namfile "congestion_control.nam"
set nam [open $namfile w]
$ns namtrace-all $nam
$ns nam-end-wireless $nam

# Exit NS2
$ns halt
$ns delete
```

SAMPLE INPUT/OUTPUT:

A Trace file was generated and a NAM window was opened simulating the network congestion control.

RESULT: Thus, the Network Simulator (NS-2) is studied in detail.

Simulation of Congestion Control Algorithms using NS-2

Ex. No. 15b)
Date:

PROBLEM STATEMENT:

To simulate network congestion control algorithm using network simulator (NS-2).

PROBLEM DESCRIPTION:

Congestion control is a state in which a part of a network message traffic is so heavy that it slows down network response time. When one part of the subnet (e. g. one or more routers in an area) becomes overloaded, congestion results. Congestion control is a global issue which involves every host and router within the subnet becomes overloaded.

Goals of TCP Congestion Control:

- To increase the source's send rate when the network is not congested.
- To share the network resources with other TCP flows in a "fair" way.
- To reduce the source's send rate when the network is congested.

The given problem statement is to use Network Simulator like NS-2 to simulate the congestion control algorithms.

NS-2 is an open-source network simulation tool which provides a simulation environment where users can design, implement, and evaluate network protocols, architectures, and applications.

NAM (Network Animator) is a graphical visualization tool that works alongside NS-2 to provide a visual representation of the simulated network and allows users to observe the flow of packers.

ALGORIHTM:

Step 1: Initialize the simulator: Create a new instance of the NS-2 simulator and open trace and nam output files for recording simulation events and visualization.

Step 2: Set colors: Define colors for nodes and links in the simulation.

Step 3: Create nodes: Create six nodes (n0 to n5) representing network entities in the simulation.

Step 4: Create links: Create duplex links between nodes with specified bandwidth, delay, and queue parameters.

Step 5: Create TCP agents: Create four TCP agents (tcp1, tcp2, tcp3, tcp4) using the Reno algorithm and attach them to corresponding sending nodes (n0, n1, n2, n1).

Step6: Create TCP sinks: Create four TCP sink agents (sink1, sink2, sink3, sink4) and attach them to corresponding receiving nodes (n4, n5, n3, n4).
Step 7: Establish connections: Connect the TCP agents to their corresponding TCP sinks to establish the traffic flow.

Step 8: Setup FTP traffic generators: Create four FTP application objects (ftp1, ftp2, ftp3, ftp4) and attach them to the TCP agents to generate FTP traffic.

Step 9: Calculate RTT using Ping: Create two Ping agents (p0, p1) and attach them to nodes n0 and n4 respectively. Connect the two Ping agents to measure the round-trip time (RTT) between them.

Step 10: Start/stop traffic and send pings: Schedule the start and stop times for FTP traffic and ping messages.

Step 11: Plot congestion window: Define a procedure plotWindow to record the congestion window (cwnd_) of a TCP agent and plot it over time. The congestion window data is saved in the congestion.xg file.

Step 12: Simulation completion and visualization: Define a finish procedure to execute after the simulation ends. It opens the nam visualization tool to display the network animation (congestion.nam) and xgraph to plot the congestion window data (congestion.xg).

Step 13: Run simulation: Run the NS-2 simulation using the run command.

CODE:

```
set ns [new Simulator]
set f [ open congestion.tr w ]
$ns trace-all $f
set nf [ open congestion.nam w ]
$ns namtrace-all $nf

$ns color 1 Red
$ns color 2 Blue
$ns color 3 White
$ns color 4 Green

#to create nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

# to create the link between the nodes with bandwidth, delay and queue
$ns duplex-link $n0 $n2   2Mb  10ms DropTail
$ns duplex-link $n1 $n2   2Mb  10ms DropTail
$ns duplex-link $n2 $n3 0.3Mb 200ms DropTail
```

```
$ns duplex-link $n3 $n4 0.5Mb  40ms DropTail
$ns duplex-link $n3 $n5 0.5Mb  30ms DropTail

# Sending node with agent as Reno Agent
set tcp1 [new Agent/TCP/Reno]
$ns attach-agent $n0 $tcp1
set tcp2 [new Agent/TCP/Reno]
$ns attach-agent $n1 $tcp2
set tcp3 [new Agent/TCP/Reno]
$ns attach-agent $n2 $tcp3
set tcp4 [new Agent/TCP/Reno]
$ns attach-agent $n1 $tcp4

$tcp1 set fid_ 1
$tcp2 set fid_ 2
$tcp3 set fid_ 3
$tcp4 set fid_ 4

# receiving (sink) node
set sink1 [new Agent/TCPSink]
$ns attach-agent $n4 $sink1
set sink2 [new Agent/TCPSink]
$ns attach-agent $n5 $sink2
set sink3 [new Agent/TCPSink]
$ns attach-agent $n3 $sink3
set sink4 [new Agent/TCPSink]
$ns attach-agent $n4 $sink4

# establish the traffic between the source and sink
$ns connect $tcp1 $sink1
$ns connect $tcp2 $sink2
$ns connect $tcp3 $sink3
$ns connect $tcp4 $sink4

# Setup a FTP traffic generator on "tcp"
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1
$ftp1 set type_ FTP

set ftp2 [new Application/FTP]
$ftp2 attach-agent $tcp2
$ftp2 set type_ FTP

set ftp3 [new Application/FTP]
$ftp3 attach-agent $tcp3
$ftp3 set type_ FTP

set ftp4 [new Application/FTP]
$ftp4 attach-agent $tcp4
$ftp4 set type_ FTP
```

```
# RTT Calculation Using Ping ----------------------------------------------

set p0 [new Agent/Ping]
$ns attach-agent $n0 $p0
set p1 [new Agent/Ping]
$ns attach-agent $n4 $p1

#Connect the two agents
$ns connect $p0 $p1

# Method call from ping.cc file
Agent/Ping instproc recv {from rtt} {
$self instvar node_
puts "node [$node_ id] received ping answer from \
$from with round-trip-time $rtt ms."
}

# ---------------------------------------------------------------------------


# start/stop the traffic
$ns at 0.2 "$p0 send"
$ns at 0.3 "$p1 send"
$ns at 0.5   "$ftp1 start"
$ns at 0.6   "$ftp2 start"
$ns at 0.7 "$ftp3 start"
$ns at 0.8   "$ftp4 start"
$ns at 66.0 "$ftp4 stop"
$ns at 67.0  "$ftp3 stop"
$ns at 68.0 "$ftp2 stop"
$ns at 70.0 "$ftp1 stop"
$ns at 70.1 "$p0 send"
$ns at 70.2 "$p1 send"

# Set simulation end time
$ns at 80.0 "finish"

# procedure to plot the congestion window
# cwnd_ used from tcp-reno.cc file
proc plotWindow {tcpSource outfile} {
   global ns
   set now [$ns now]
   set cwnd_ [$tcpSource set cwnd_]

# the data is recorded in a file called congestion.xg.
   puts  $outfile  "$now $cwnd_"
   $ns at [expr $now+0.1] "plotWindow $tcpSource  $outfile"
}

set outfile [open  "congestion.xg"  w]
$ns  at  0.0  "plotWindow $tcp1  $outfile"
proc finish {} {
```

```
 exec nam congestion.nam &
 exec xgraph congestion.xg -geometry 300x300 &
exit 0
}
# Run simulation
$ns run
```

SAMPLE INPUT/OUTPUT:

Output NAM file was generated and simulation of network control was observed in NAM window.

```
merudhula@Merudhula-PC:/mnt/c/Users/Admin/OneDrive/Desktop/NS2$ ns congestion.tcl
node 0 received ping answer from  4 with round-trip-time 506.0 ms.
node 4 received ping answer from  0 with round-trip-time 506.0 ms.
node 0 received ping answer from  4 with round-trip-time 506.0 ms.
node 4 received ping answer from  0 with round-trip-time 506.0 ms.
nam:
ns: finish: couldn't execute "xgraph": no such file or directory
    while executing
"exec xgraph congestion.xg -geometry 300x300 &"
    (procedure "finish" line 3)
    invoked from within
"finish"
merudhula@Merudhula-PC:/mnt/c/Users/Admin/OneDrive/Desktop/NS2$
```

Trace File:

+ 0.2 0 2 ping 64 ------- 0 0.1 4.2 -1 0
- 0.2 0 2 ping 64 ------- 0 0.1 4.2 -1 0
r 0.210256 0 2 ping 64 ------- 0 0.1 4.2 -1 0
+ 0.210256 2 3 ping 64 ------- 0 0.1 4.2 -1 0
- 0.210256 2 3 ping 64 ------- 0 0.1 4.2 -1 0
+ 0.3 4 3 ping 64 ------- 0 4.2 0.1 -1 1
- 0.3 4 3 ping 64 ------- 0 4.2 0.1 -1 1
r 0.341024 4 3 ping 64 ------- 0 4.2 0.1 -1 1
+ 0.341024 3 2 ping 64 ------- 0 4.2 0.1 -1 1
- 0.341024 3 2 ping 64 ------- 0 4.2 0.1 -1 1
r 0.411963 2 3 ping 64 ------- 0 0.1 4.2 -1 0
+ 0.411963 3 4 ping 64 ------- 0 0.1 4.2 -1 0
- 0.411963 3 4 ping 64 ------- 0 0.1 4.2 -1 0
r 0.452987 3 4 ping 64 ------- 0 0.1 4.2 -1 0
+ 0.452987 4 3 ping 64 ------- 0 4.2 0.1 -1 2
- 0.452987 4 3 ping 64 ------- 0 4.2 0.1 -1 2
r 0.494011 4 3 ping 64 ------- 0 4.2 0.1 -1 2
+ 0.494011 3 2 ping 64 ------- 0 4.2 0.1 -1 2
- 0.494011 3 2 ping 64 ------- 0 4.2 0.1 -1 2
+ 0.5 0 2 tcp 40 ------- 1 0.0 4.0 0 3
- 0.5 0 2 tcp 40 ------- 1 0.0 4.0 0 3
r 0.51016 0 2 tcp 40 ------- 1 0.0 4.0 0 3
+ 0.51016 2 3 tcp 40 ------- 1 0.0 4.0 0 3
- 0.51016 2 3 tcp 40 ------- 1 0.0 4.0 0 3
r 0.542731 3 2 ping 64 ------- 0 4.2 0.1 -1 1

+ 0.542731 2 0 ping 64 ------- 0 4.2 0.1 -1 1
- 0.542731 2 0 ping 64 ------- 0 4.2 0.1 -1 1
r 0.552987 2 0 ping 64 ------- 0 4.2 0.1 -1 1
+ 0.552987 0 2 ping 64 ------- 0 0.1 4.2 -1 4
- 0.552987 0 2 ping 64 ------- 0 0.1 4.2 -1 4
r 0.563243 0 2 ping 64 ------- 0 0.1 4.2 -1 4
+ 0.563243 2 3 ping 64 ------- 0 0.1 4.2 -1 4
- 0.563243 2 3 ping 64 ------- 0 0.1 4.2 -1 4
+ 0.6 1 2 tcp 40 ------- 2 1.0 5.0 0 5
- 0.6 1 2 tcp 40 ------- 2 1.0 5.0 0 5
r 0.61016 1 2 tcp 40 ------- 2 1.0 5.0 0 5
+ 0.61016 2 3 tcp 40 ------- 2 1.0 5.0 0 5
- 0.61016 2 3 tcp 40 ------- 2 1.0 5.0 0 5
r 0.695717 3 2 ping 64 ------- 0 4.2 0.1 -1 2
+ 0.695717 2 0 ping 64 ------- 0 4.2 0.1 -1 2
…..
…..

RESULT: Thus, the Simple Network Management Protocol (SNMP) using Twisted Python was implemented and tested successfully.

Implementation of the Routing Algorithms – Link State Routing

Ex. No. 16c)
Date:

PROBLEM STATEMENT:

To write a program to implement link state routing protocol using Twisted Python.

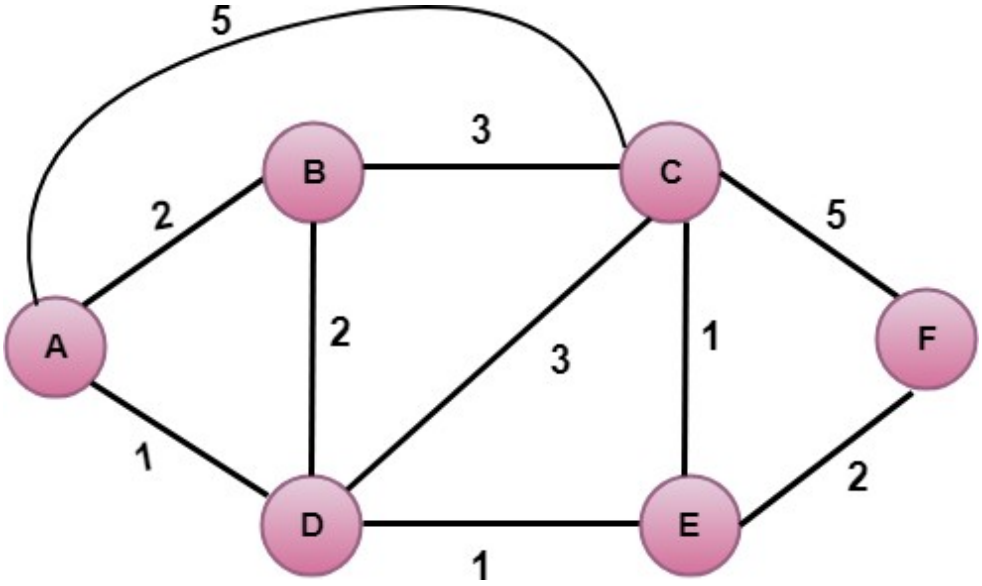
PROBLEM DESCRIPTION:

The link-state routing algorithm is a network routing approach where each router has complete knowledge of the network. It operates in several steps: routers exchange link-state advertisements (LSAs) to discover the network's status, build a database of the topology, calculate the shortest path to every other router using a shortest path algorithm, generate routing tables based on the calculations, and flood updates to all routers when there are network changes.

WORKING:

- Each router in the network sends out a special message calleda "link-state advertisement" (LSA) to all other routers.
- Upon receiving LSAs from other routers, each router creates adatabase of the network's topology.
- Using the information in the topology database, each routerindependently calculates the shortest path to every other router in the network.
- Based on the shortest path calculations, each router constructs its own routing table, which specifies the next hop(next router) for each destination in the network.
- Once the routing tables are updated, each router can use the information to forward packets to their destinations along theshortest paths.

GRAPHICAL REPRESENTATION:



INITIAL TABLE:

Step	N	D(B),P(B)	D(C),P(C)	D(D),P(D)	D(E),P(E)	D(F),P(F)
1	A	2,A	5,A	1,A	∞	∞
2	AD	2,A	4,D		2,D	∞
3	ADE	2,A	3,E			4,E
4	ADEB		3,E			4,E
5	ADEBC					4,E

Final table:

Step	N	D(B),P(B)	D(C),P(C)	D(D),P(D)	D(E),P(E)	D(F),P(F)
1	A	2,A	5,A	1,A	∞	∞
2	AD	2,A	4,D		2,D	∞
3	ADE	2,A	3,E			4,E
4	ADEB		3,E			4,E
5	ADEBC					4,E
6	ADEBCF					

CODE:

```
from twisted.internet import reactor, protocol
import pickle
import heapq
class Router(protocol.Protocol):
    def __init__(self,factory):
        self.name=factory.name
    def connectionMade(self):
       #self.transport.write(self)
       d={}
       s=self.name
       d['name']=s
       n=int(input("Enter how many neighbour's u want to connect: "))
       l={}
       for i in range(n):
            p=input("Enter Neighbour Name: ")
            print("Enter",p,"Cost:")
            s=int(input())
            l[p]=s
            print("Connected to ",p)
       d['neighbours']=l
       k=pickle.dumps(d)
       u=input(' Compute shortest distance: ')
       if u==1:
        h=[self.name]
        self.transport.write(pickle.dumps(h))
```

```
    return distances
    def dataReceived(self, data):
        k=data.loads()
        if type(k)==dict:
            #print("Neighbours routing table:")
            for k,v in k:
                if v!=self.name:
                    print(k,"-",v)

class RouterFactory(protocol.ClientFactory):
    def __init__(self):
        self.name=input("Enter name of ur node: ")
    def buildProtocol(self, addr):
        return Router(self)
    def routerConnectionFailed(self, connector, reason):
        print ("Connection failed.")
        reactor.stop()
    def routerConnectionLost(self, connector, reason):
        print ("Connection lost.")
        reactor.stop()
```

ALGORITHM IMPLEMENTATION:

```
def shortest_paths(self,start):
  graph=self.n
  distances = {node: float('inf') for node in graph}  # Initialize distances to infinity
  distances[start] = 0  # Distance from start node to itself is 0
  priority_queue = [(0, start)]  # Use a priority queue to store nodes based on their distance from the s

  while priority_queue:
    current_distance, current_node = heapq.heappop(priority_queue)

    # Check the neighbors of the current node
    for neighbor, weight in graph[current_node].items():
        distance = current_distance + weight

        # If a shorter path is found, update the distance and add the neighbor to the priority queue
        if distance < distances[neighbor]:
            distances[neighbor] = distance
            heapq.heappush(priority_queue, (distance, neighbor))

  return distances
```

SAMPLE INPUT/OUTPUT:

INITIAL CONNECTION OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

nadish@nadish-Inspiron-13-5310:~/Desktop/link_state$ python3 link_cl.py
Enter name of ur node: Node1
Enter how many neighbour's u want to connect: 2
Enter Neighbour Name: Node2
Enter Node2 Cost:
3
Connected to  Node2
Enter Neighbour Name: Node3
Enter Node3 Cost:
7
Connected to  Node3
```

Output:

```
nk_state/link_cl.py
Enter the source node: Node1
The shortest distance from Node1 to Node1 is 0
The shortest distance from Node1 to Node2 is 3
The shortest distance from Node1 to Node3 is 2
The shortest distance from Node1 to Node4 is 8
The shortest distance from Node1 to Node5 is 11
nadish@nadish-Inspiron-13-5310:~/Desktop/link_state$ ~
```

RESULT:

Thus, the Link state Routing Protocol has been implemented using Twisted Python and tested successfully.

Implementation of the Routing Algorithms – Flooding

Ex. No. 16b)
Date:

PROBLEM STATEMENT:

To write a program to implement routing protocol using flooding in Twisted Python.

PROBLEM DESCRIPTION:

In computer networks, flooding is an easy and straightforward routing technique in which the source or node sends packets over each of the outgoing links. Flooding is a very simple routing algorithm that sends all the packets arriving via each outgoing link. Flooding is used in computer networking routing algorithms where each incoming packet is transmitted through every outgoing link, except for the one on which it arrived. Flooding algorithms are guaranteed to find and exploit the shortest paths to the sent packets, as floods use each route in a network naturally.

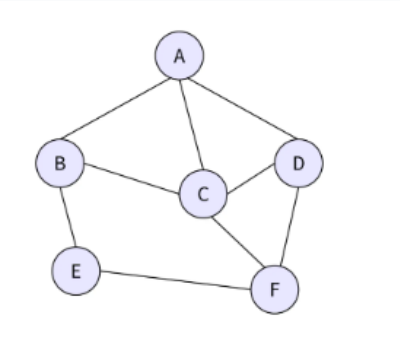
ADVANTAGES OF FLOODING TECHNIQUE

- The benefit of this technique is it is very easy to implement. If the packet is able to be delivered, it will be delivered (probably more than once), and since flooding naturally uses every path through the network, the shortest path will also be used.
- It's highly durable. Even if a high number of routers fail, the packets find a way to get to their destination.
- All nodes that are connected to each other, whether directly or indirectly, are visited. As a result, there is no way for any node to be missed. In the case of messaging, this is an important criterion.

In a client-server network architecture, where multiple clients connect to a central server, there is a need to establish efficient communication between the server and clients. The network employs a flooding routing protocol, where the server broadcasts messages to all connected clients. However, this flooding routing protocol presents certain challenges and problems that need to be addressed.

WORKING:

Assume there is a network with 6 routers connected through transmission lines, as shown in the figure ahead.



Following are the Events that Take Place in Flooding:

- Any packet incoming to A is sent to D, C, and B.
- B sends this packet to E and C.
- C sends this packet to F, D, and B.
- D sends this packet to F and C.
- E sends the packet to F.
- F sends the packet to E and C.

CODE:

SERVER

```
from twisted.internet import reactor, protocol

class FloodingProtocol(protocol.Protocol):
def connectionMade(self):
print("New client connected:", self.transport.getPeer())
self.transport.write(b"Welcome to the server!\n")

def dataReceived(self, data):
print("Received data from client:", data.decode())
self.floodClients(data)

def connectionLost(self, reason):
print("Client disconnected:", self.transport.getPeer())

def floodClients(self, data):
for client in self.factory.clients:
client.transport.write(data)


class FloodingFactory(protocol.Factory):
def __init__(self):
self.clients = []
```

```
def buildProtocol(self, addr):
protocol = FloodingProtocol()
protocol.factory = self
self.clients.append(protocol)
return protocol


if __name__ == '__main__':
port = 8000
print("Starting TCP server on port", port)
factory = FloodingFactory()
reactor.listenTCP(port, factory)
reactor.run()
```

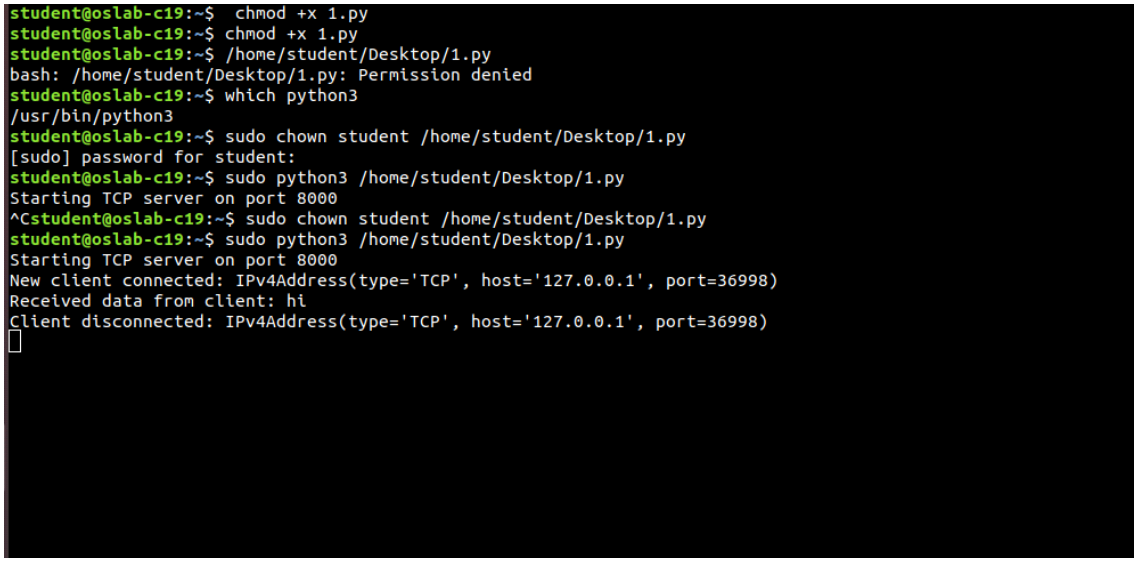
CLIENT

```
from twisted.internet import reactor, protocol
class FloodingClient(protocol.Protocol):
def connectionMade(self):
print("Connected to server. You can start sending messages.")
self.sendData()

def dataReceived(self, data):
print("Received data from server:", data.decode())

def connectionLost(self, reason):
print("Connection lost.")

def sendData(self):
message = input("Enter a message to send (or 'quit' to exit): ")
if message == 'quit':
self.transport.loseConnection()
else:
self.transport.write(message.encode())
reactor.callLater(0, self.sendData)


class FloodingClientFactory(protocol.ClientFactory):
def buildProtocol(self, addr):
return FloodingClient()


if __name__ == '__main__':
host = 'localhost'
port = 8000
print("Starting TCP client...")
from twisted.internet import reactor, protocol


class FloodingClient(protocol.Protocol):
```

```
    def connectionMade(self):
        print("Connected to server. You can start sending messages.")
        self.sendData()

    def dataReceived(self, data):
        print("Received data from server:", data.decode())

    def connectionLost(self, reason):
        print("Connection lost.")

    def sendData(self):
        message = input("Enter a message to send (or 'quit' to exit): ")
        if message == 'quit':
            self.transport.loseConnection()
        else:
            self.transport.write(message.encode())
            reactor.callLater(0, self.sendData)


class FloodingClientFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return FloodingClient()


if __name__ == '__main__':
    host = 'localhost'
    port = 8000
    print("Starting TCP client...")
    reactor.connectTCP(host, port, FloodingClientFactory())
    reactor.run()reactor.connectTCP(host, port, FloodingClientFactory())
reactor.run()
```

SAMPLE INPUT/OUTPUT:

SERVER:



CLIENT:

```
student@oslab-c19:~$ python3 2.py
student@oslab-c19:~$ sudo chown student /home/student/Desktop/2.py
[sudo] password for student:
student@oslab-c19:~$ sudo python3 /home/student/Desktop/2.py
Starting TCP client...
Connected to server. You can start sending messages.
Enter a message to send (or 'quit' to exit): hi
Enter a message to send (or 'quit' to exit): quit
Connection lost.
```

RESULT:

Thus, the Routing Protocol using Flooding has been implemented using Twisted Python and tested successfully.

Implementation of the routing algorithms – Distance Vector Routing

Ex. No. 16c)

Date:

PROBLEM STATEMENT:

To write a program to implement distance vector routing protocol using Twisted Python.

PROBLEM DESCRIPTION:

Distance vector routing protocols are used in computer networks to determine the best paths for data packets to travel from source to destination. The main goal of such protocols is to minimize the time taken and the resources utilized for packet transmission. However, there are several challenges that need to be addressed for effective routing in a network.

- Network Topology
- Link Costs
- Dynamic Network Conditions
- Scalability

To address these challenges, a distance vector routing protocol uses a distributed approach, where each router maintains its own routing table and periodically exchanges information with its neighbouring routers. The routers share information about their distance (cost) to various destinations in the network, allowing each router to calculate the shortest path to each destination using the Bellman-Ford algorithm or a similar algorithm.

By continuously exchanging routing information, routers can keep their routing tables up to date, adapting to changes in network topology and link costs. The distance vector protocol iteratively converges towards the optimal paths for packet transmission.

Efficient distance vector routing protocols, such as Routing Information Protocol (RIP) and Interior Gateway Routing Protocol (IGRP), strike a balance between simplicity and effectiveness. They provide a reliable mechanism for routing packets in computer networks while considering network topology, link costs, and dynamic network conditions.

CODE:

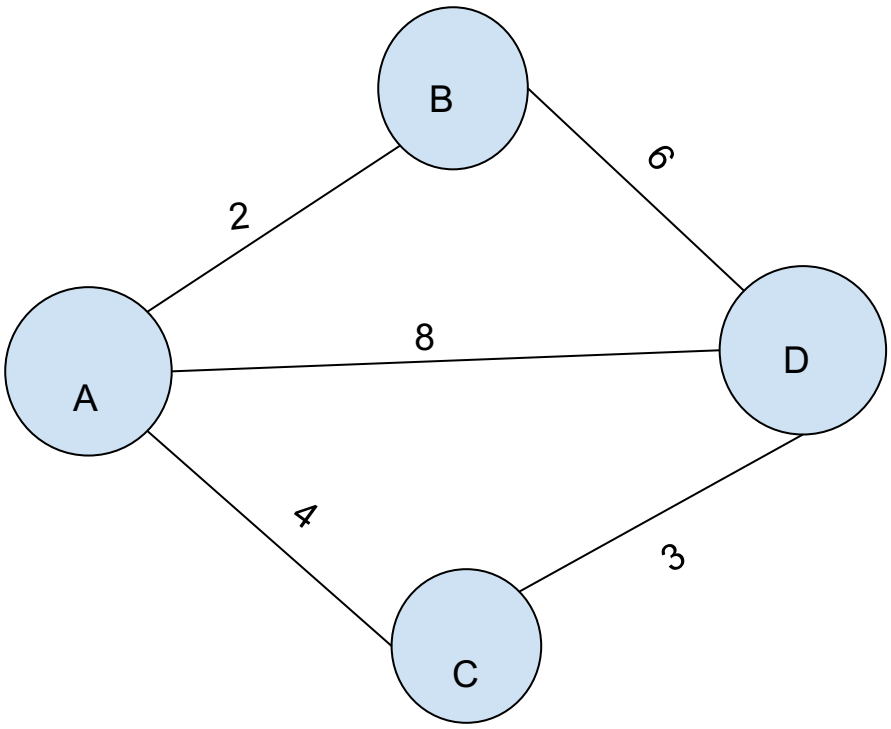
```
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor


class DistanceVectorRoutingProtocol(DatagramProtocol):
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.routing_table = {}
```

```
        self.neighbor_routers = []

    def startProtocol(self):
        self.transport.joinGroup("224.0.0.0")
        print(f"Started routing protocol on {self.host}:{self.port}")

        if self.port == 8000:
            self.updateRoutingTable("A", "A", 0)
            self.sendRoutingUpdate("A", "B", 2)
            self.sendRoutingUpdate("A", "C", 4)
            self.sendRoutingUpdate("A", "D", 7)

    def sendRoutingUpdate(self, source, destination, cost):
        routing_update = f"{source},{destination},{cost}"
        self.transport.write(routing_update.encode(), ("224.0.0.0", self.port))

    def datagramReceived(self, datagram, address):
        routing_update = datagram.decode()
        source, destination, cost = routing_update.split(",")
        self.updateRoutingTable(source, destination, int(cost))
        print(f"Received routing update from {address}: {routing_update}")

    def updateRoutingTable(self, source, destination, cost):
        if destination not in self.routing_table or cost < self.routing_table[destination][1]:
            self.routing_table[destination] = (source, cost)

    def calculateShortestPaths(self):
        infinity = float("inf")
        shortest_paths = {router: (infinity, None) for router in self.routing_table.keys()}
        shortest_paths[self.host] = (0, self.host)

        updated = True
        while updated:
            updated = False
            for neighbor, (nh, cost) in self.routing_table.items():
                for dest, (nh_dest, cost_dest) in self.routing_table.items():
                    if neighbor != self.host and dest != self.host:
                        new_cost = cost + cost_dest
                        if new_cost < shortest_paths[dest][0]:
                            shortest_paths[dest] = (new_cost, neighbor)
                            updated = True

        self.routing_table = {dest: (nh_dest, cost_dest) for dest, (cost_dest, nh_dest) in
shortest_paths.items()}
```

```
        print("Routing table updated:")
        print("Destination   Next Hop   Cost")
        for dest, (nh, cost) in self.routing_table.items():
            print(f"{dest}\t\t{nh}\t\t{cost}")


        self.broadcastRoutingTable()


    def broadcastRoutingTable(self):
        print("Broadcasting routing table:")
        for dest, (nh, cost) in self.routing_table.items():
            routing_update = f"{self.host},{dest},{cost}"
            self.transport.write(routing_update.encode(), ("224.0.0.0", self.port))


if __name__ == "__main__":
    host = "127.0.0.1"
    port = 8000


    protocol = DistanceVectorRoutingProtocol(host, port)
    reactor.listenMulticast(port, protocol, listenMultiple=True)


    reactor.callLater(5, protocol.calculateShortestPaths)  # Delayed execution of shortest path
calculation
    reactor.run()
```

SAMPLE INPUT/OUTPUT:

```
Started routing protocol on 127.0.0.1:8000
Received routing update from ('192.168.1.6', 8000): A,B,2
Received routing update from ('192.168.1.6', 8000): A,C,4
Received routing update from ('192.168.1.6', 8000): A,D,7
Routing table updated:
Destination   Next Hop   Cost
A               A               0
B               A               2
C               A               4
D               A               7
127.0.0.1               127.0.0.1               0
Broadcasting routing table:
Received routing update from ('192.168.1.6', 8000): 127.0.0.1,A,0
Received routing update from ('192.168.1.6', 8000): 127.0.0.1,B,2
Received routing update from ('192.168.1.6', 8000): 127.0.0.1,C,4
Received routing update from ('192.168.1.6', 8000): 127.0.0.1,D,7
Received routing update from ('192.168.1.6', 8000): 127.0.0.1,127.0.0.1,0
```



ROUTING GROUP: 224.0.0.0

RESULT:

Thus, the Distance Vector Routing Protocol has been implemented using Twisted Python and tested successfully.

Mini-Project:
Simulation of SSL Client/Server Application using Twisted Python

Ex. No. 17
Date:

PROBLEM STATEMENT:

To write a program to set up an Secure Shell (SSH) server and client

PROBLEM DESCRIPTION:

NEED FOR SSH:

Secure Shell (SSH) is needed for several reasons:

1. **Secure Remote Access:** SSH provides a secure method for remotely accessing and managing systems and network devices. It encrypts the communication between the client and server, protecting sensitive information such as passwords, commands, and data from being intercepted or tampered with by attackers.

2. **Data Confidentiality:** SSH encrypts all communication, ensuring that data transmitted over the network cannot be easily understood by eavesdroppers. This is particularly important when accessing or transferring sensitive data, such as login credentials, financial information, or confidential documents.

3. **Authentication and Access Control:** SSH uses various authentication methods, including passwords and public-key cryptography, to verify the identity of users before granting them access to the system. This helps prevent unauthorized access and ensures that only trusted individuals can connect to the server.

4. **Secure File Transfer:** SSH provides secure file transfer capabilities, allowing users to transfer files between systems over an encrypted channel. This ensures that files are transmitted securely, protecting their integrity and confidentiality during transit.

5. **Remote Administration:** SSH enables system administrators to remotely manage and administer systems and network devices. They can securely execute commands, manage configurations, troubleshoot issues, and perform administrative tasks from anywhere, reducing the need for physical access to the machines.

6. **Tunneling and Port Forwarding:** SSH supports tunneling, which allows users to securely access services running on remote machines by forwarding ports over an encrypted SSH connection. This enables users to access services such as databases, web servers, and email servers securely, even if they are behind firewalls or on private networks.

7. **Strong Security Standards:** SSH is based on strong cryptographic algorithms and security standards. It has evolved over time to address vulnerabilities and emerging threats, making it a reliable and secure choice for remote access and data transfer.

Overall, SSH is needed to establish secure and encrypted communication channels, protect data privacy and integrity, authenticate users, and enable secure remote administration and file transfer. It is an essential tool for maintaining the security and privacy of systems and networks.

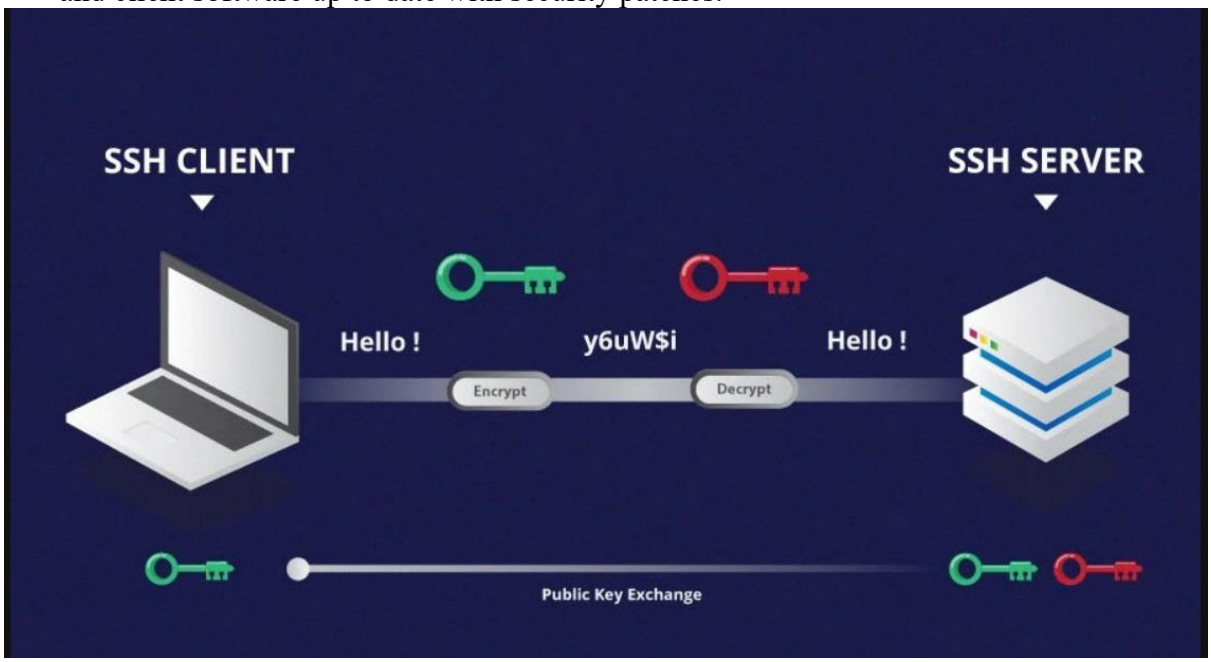
To set up an SSH server and client, the following steps need to be followed.

SSH Server:

1. Install an SSH server software on the remote system where you want to enable SSH access. OpenSSH is the most commonly used SSH server implementation and is available on various operating systems.
2. Configure the SSH server settings, including network port, authentication methods, and other security options. The configuration file is usually located at `/etc/ssh/sshd_config` on Linux-based systems.
3. Start the SSH server service. On Linux, you can usually use a command like 'sudo service ssh start' or 'sudo systemctl start sshd' to start the SSH server.

SSH Client:

1. Ensure that you have an SSH client installed on your local system. Most Linux distributions and macOS have an SSH client pre-installed. For Windows, you can use popular clients like OpenSSH or PuTTY.
2. Open a terminal or command prompt on your local system and enter the SSH client command followed by the server's IP address or domain name. The command typically looks like 'ssh username@server_ip' or 'ssh username@server_domain'.
3. If connecting to the server for the first time, you may be prompted to confirm the server's authenticity by verifying its host key fingerprint. Once confirmed, the server's key will be stored on your local system for future connections.
4. Provide the username and password (if password authentication is enabled) or your private key passphrase (if using key-based authentication) to establish the SSH connection.
5. Once connected, you can execute commands on the remote server, transfer files, or perform other tasks as per your requirements.
6. It's important to note that SSH is a powerful tool, and you should take appropriate security measures to protect your systems, such as using strong passwords or passphrase-protected private keys, disabling password authentication if possible, and keeping your SSH server and client software up to date with security patches.



CODE:

SERVER:

```
from twisted.conch import error
from twisted.conch.ssh import factory, keys, userauth, connection
from twisted.cred import portal
from twisted.internet import reactor
from twisted.python import log

class SSHDemoAvatar(userauth.SSHUserAuthServer):
    def getPassword(self):
        # Authenticate the user and return the password
        # or None if authentication fails
        return "password"  # Replace with your authentication logic

    def getPublicKey(self):
        # Return a public key or None for password-based authentication
        return None  # Replace with your public key logic

    def getPrivateKey(self):
        # Return a private key for public key-based authentication
        # or None for password-based authentication
        return None  # Replace with your private key logic

class SSHDemoRealm:
    def requestAvatar(self, avatarId, mind, *interfaces):
        if connection.IConchUser in interfaces:
            return interfaces[0], SSHDemoAvatar(), lambda: None
        raise NotImplementedError("No supported interfaces found.")

class SSHDemoFactory(factory.SSHFactory):
    def _init_(self):
        self.privateKeys = {'ssh-rsa': keys.Key.fromString(data='...')}
        self.publicKeys = {'ssh-rsa': keys.Key.fromString(data='...')}
        self.portal = portal.Portal(SSHDemoRealm())

    def buildProtocol(self, addr):
        p = self.protocol()
        p.factory = self
        return p

log.startLogging(open("ssh_server.log", "w"))
reactor.listenTCP(2222, SSHDemoFactory())
reactor.run()
```

CLIENT

```
from twisted.internet import reactor
from twisted.conch.client import options
from twisted.conch.client import default, forward
from twisted.conch.ssh import transport
```

```
from twisted.conch.ssh.keys import Key
from twisted.python import log

class SSHDemoClientTransport(transport.SSHClientTransport):
    def verifyHostKey(self, hostKey, fingerprint):
        # Verify the server's host key
        # Return True to accept or False to reject
        return True  # Replace with your host key verification logic

    def connectionSecure(self):
        # Connection is secure, start authentication
        self.requestService(SSHUserAuthClient('username', 'password'))  # Replace with your
authentication details

class SSHUserAuthClient(userauth.SSHUserAuthClient):
    def getPassword(self):
        # Return the password for authentication
        return self.password

log.startLogging(open("ssh_client.log", "w"))
options.ConchOptions().parseOptions([])
reactor.connectTCP('server_ip', 2222, SSHDemoClientTransport())
reactor.run()
```

SAMPLE INPUT/OUTPUT:

Server:

```
SSH server started. Listening on port 2222...
```

Client:

```
SSH client started. Connecting to server...
Connected to server. Performing host key verification...
Host key verification successful. Starting authentication...
Authentication successful. SSH client connected.
```

RESULT:

The SSL Client/Server Application has been implemented and tested successfully using Twisted Python.