EX.NO: 1

DESIGN A BASIC LAN TOPOLOGY WITH ROUTERS AND SWITCHES

AIM:

To simulate OSI and TCP/IP layered communication using Cisco Packet Tracer by designing a basic LAN topology with routers and switches, configuring network devices, and verifying communication using command-line tools.

PROCEDURE:

1. Open Cisco Packet Tracer → **File** → **New**.
2. From the toolbox in the left-bottom corner, choose the devices required for the LAN connection.
3. Click on **End Devices** in the toolbox and select **PC**. Place 2 PCs in the blank workspace.
4. Click on **Network Devices** → **Switches** and select **2960-24TT Switch**. Place 2 switches in the workspace.
5. Click on **Network Devices** → **Routers** and select **1941 Router**. Place 1 router in the workspace.
6. Arrange the devices in the following network architecture:

   PC1 ---- Switch1 ---- Router1 ---- Switch2 ---- PC2

7. Connect:
   - PC1 to Switch1
   - Switch1 to Router1
   - Router1 to Switch2
   - Switch2 to PC2
8. While connecting the cables, select the **Fast Ethernet** ports for PCs and switches, and **Gigabit Ethernet** ports for the router.
9. If the connection is successful:

- Green light indicates successful connection
- Orange light indicates connection in progress
- Red light indicates unsuccessful connection

11. After successful connections, configure the router:

- Click on the router
- Go to **CLI**

12. Enter the following commands in the router CLI:

```
enable
configure terminal
interface g0/0
ip address 192.168.1.1 255.255.255.0
no shutdown
exit
interface g0/1
ip address 192.168.2.1 255.255.255.0
no shutdown
exit
end
write memory
```

Below are steps to enter commands in CLI

1. Press **Enter** on the keyboard.
2. The following message may appear:

   Continue with configuration dialog? [yes/no]:

8. Type the following command and press **Enter**:

   no

9. The router prompt will appear as:

   Router>

10. Type the following command to enter privileged EXEC mode:

    enable

11. The prompt changes to:

    Router#

12. Type the following command to enter global configuration mode:

    configure terminal

13. The prompt changes to:

    Router(config)#

14. Configure the first interface by typing:

    interface g0/0

15. Assign the IP address using:

ip address 192.168.1.1 255.255.255.0

16. Enable the interface using:

    no shutdown

17. Exit the interface configuration mode:

    exit

18. Configure the second interface:

    interface g0/1

19. Assign the second IP address:

    ip address 192.168.2.1 255.255.255.0

20. Enable the interface:

    no shutdown

21. Exit interface mode:

    exit

22. Return to privileged mode:

    end

23. Save the router configuration:

    write memory

24. To test the network, click on a **PC**.
25. Go to **Desktop → Command Prompt**.
26. Type the ping command:

    ping 192.168.2.10

27. Press **Enter**.
28. If replies are received, the network configuration is successful.

13. Configure PC1:

- Click on PC1
- Go to **Desktop → IP Configuration**
- Enter:
  - IP Address: 192.168.1.10
  - Subnet Mask: 255.255.255.0
  - Default Gateway: 192.168.1.1

14. Configure PC2:

- Click on PC2
- Go to **Desktop → IP Configuration**

- nter:
    - IP Address: 192.168.2.10
    - Subnet Mask: 255.255.255.0
    - Default Gateway: 192.168.2.1

15. The PCs connected in the same network must have the same first 3 octets of the IP address.
16. To verify the network connection:

- Click on PC1
- Go to **Desktop → Command Prompt**

17. Use the ping command to test connectivity:

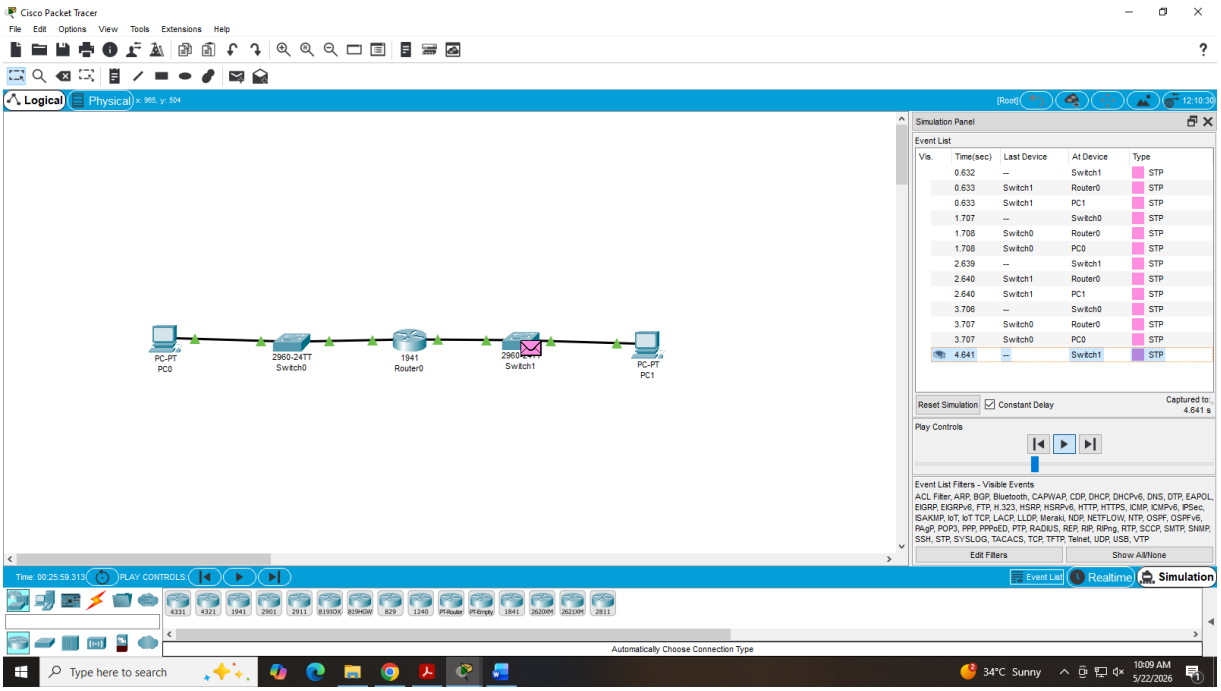
ping 192.168.2.10

18. If replies are received, the devices are successfully configured and connected.
19. To observe packet transmission and OSI/TCP-IP layer communication:

- Click on **Simulation Mode**

20. Click on **Auto Capture/Play** to view packet movement and network communication between devices.
21. Observe the packet flow through:
    - Physical Layer
    - Data Link Layer
    - Network Layer
    - Transport Layer
    - Application Layer

OUTPUT:



RESULT:

Thus, the simulation of OSI and TCP/IP layered communication using Cisco Packet Tracer was successfully performed by designing and configuring a basic LAN topology with routers and switches.

EX.NO: 2

ANALYZE SIGNAL TYPES AND BIT RATE CALCULATIONS USING PYTHON

AIM:

To observe simplex, half-duplex, and full-duplex transmission modes using Cisco Packet Tracer and to analyze analog and digital signals with bit rate calculation using Python.

PROCEDURE:

Observation of Transmission Modes Using Cisco Packet Tracer

Simplex Communication

1. Open Cisco Packet Tracer.
2. Select two PCs from End Devices:
   - PC0
   - PC1
3. Connect both PCs using Copper Cross-Over Cable.
4. Configure IP addresses:

   PC0

   - IP Address: 192.168.1.1
   - Subnet Mask: 255.255.255.0

   PC1

   - IP Address: 192.168.1.2
   - Subnet Mask: 255.255.255.0
5. Open Desktop → Command Prompt on PC0.
6. Execute the command:

ping 192.168.1.2

7. Observe one-way communication from sender to receiver representing simplex mode.

Half-Duplex Communication

1. Use the same network topology.
2. Switch to Simulation Mode in Cisco Packet Tracer.
3. Send a Simple PDU from PC0 to PC1.
4. After transmission completion, send another Simple PDU from PC1 to PC0.

5. Observe that communication occurs in both directions but only one device transmits at a time representing half-duplex mode.

Full-Duplex Communication

1. Add one switch:
   - Switch0
2. Connect:
   - PC0 to Switch0
   - PC1 to Switch0

   using Copper Straight-Through Cable.

3. Configure IP addresses if required.
4. Open Command Prompt on both PCs.
5. Execute simultaneously:

   On PC0

   ping 192.168.1.2

   On PC1

   ping 192.168.1.1

6. Observe simultaneous two-way communication representing full-duplex mode.

Signal Analysis and Bit Rate Calculation Using Python

1. Install required Python libraries using:

```
pip install numpy matplotlib
```

2. Open Python IDE or Jupyter Notebook.
3. Enter the following Python program:

```
import numpy as np
import matplotlib.pyplot as plt

# Time values
t = np.linspace(0, 1, 1000)

# Analog signal
analog_signal = np.sin(2 * np.pi * 5 * t)

# Digital signal
digital_signal = np.sign(np.sin(2 * np.pi * 5 * t))
```

```
#read input
bits = int(input("Enter number of bits transmitted: "))
time = float(input("Enter transmission time in seconds: "))


# Plotting signals
plt.figure(figsize=(10,6))

plt.subplot(2,1,1)
plt.plot(t, analog_signal, color='blue')
plt.title("Analog Signal")
plt.xlabel("Time")
plt.ylabel("Amplitude")

plt.subplot(2,1,2)
plt.plot(t, digital_signal, color='red')
plt.title("Digital Signal")
plt.xlabel("Time")
plt.ylabel("Amplitude")

plt.tight_layout()
plt.show()

# Bit rate calculation

bit_rate = bits / time

print("Bit Rate =", bit_rate, "bps")
```

4. Run the Python program.
5. Observe the analog signal as a sine wave and the digital signal as square pulses.
6. Enter sample input values:

Number of bits = 8000
Time = 2 seconds

7. Observe the calculated bit rate:

Bit Rate = 8000 / 2 = 4000 bps

Explanation :

Signal Types and Bit Rate Calculations

In data communication, information is transmitted through signals. Understanding different signal types and how to calculate bit rates is fundamental to designing and analyzing communication systems.

Signal Types

Signals can be broadly classified in several ways:

- **Analog vs. Digital:**
    - **Analog Signals:** Continuous in both time and amplitude. Examples include voice audio, radio waves.
    - **Digital Signals:** Discrete (stepwise) in both time and amplitude. Represented by a sequence of voltage pulses (e.g., square waves).
- **Periodic vs. Aperiodic:**
    - **Periodic Signals:** Complete a pattern within a measurable time frame, called a period, and repeat that pattern over subsequent identical periods. Examples include sine waves, square waves.
    - **Aperiodic Signals:** Change without exhibiting a pattern or cycle that repeats over time. Examples include human speech, data bursts.

Bit Rate Calculation

Bit rate refers to the number of bits transmitted per second. It's a fundamental measure of the speed of data transmission. The bit rate depends on several factors, including:

1. **Baud Rate (Symbol Rate):** The number of signal elements (symbols) transmitted per second. A symbol can represent one or more bits.
2. **Number of Bits per Symbol (M):** How many bits are encoded into each signal element. This is determined by the modulation scheme (e.g., BPSK, QPSK, 16-QAM).

Formula:

$$\text{Bit Rate} = \text{Baud Rate} \times \text{Bits per Symbol}$$

Or, if the number of signal levels is L, then $\text{Bits per Symbol} = \log_2(L)$.

So, $\text{Bit Rate} = \text{Baud Rate} \times \log_2(L)$

PROGRAM:

```
import math

def calculate_bit_rate(baud_rate, bits_per_symbol=None, num_signal_levels=None):
    """
    Calculates the bit rate given baud rate and either bits per symbol or number of signal levels.

    Args:
        baud_rate (float): The baud rate (symbols per second).
        bits_per_symbol (int, optional): The number of bits encoded per symbol.
        num_signal_levels (int, optional): The number of distinct signal levels (e.g., for QAM).

    Returns:
        float: The calculated bit rate (bits per second).

    Raises:
        ValueError: If neither bits_per_symbol nor num_signal_levels is provided, or if both are.
                    Also if num_signal_levels is not a power of 2 or less than 2.
    """
    if (bits_per_symbol is None and num_signal_levels is None) or \
       (bits_per_symbol is not None and num_signal_levels is not None):
        raise ValueError("Please provide either 'bits_per_symbol' or 'num_signal_levels', but not both.")

    if num_signal_levels is not None:
        if num_signal_levels < 2 or not (math.log2(num_signal_levels)).is_integer():
            raise ValueError("Number of signal levels must be a power of 2 and greater than or equal to 2.")
        bits_per_symbol = int(math.log2(num_signal_levels))

    bit_rate = baud_rate * bits_per_symbol
    return bit_rate

# Demonstration of Bit Rate Calculations

print("--- Bit Rate Calculations ---\n")

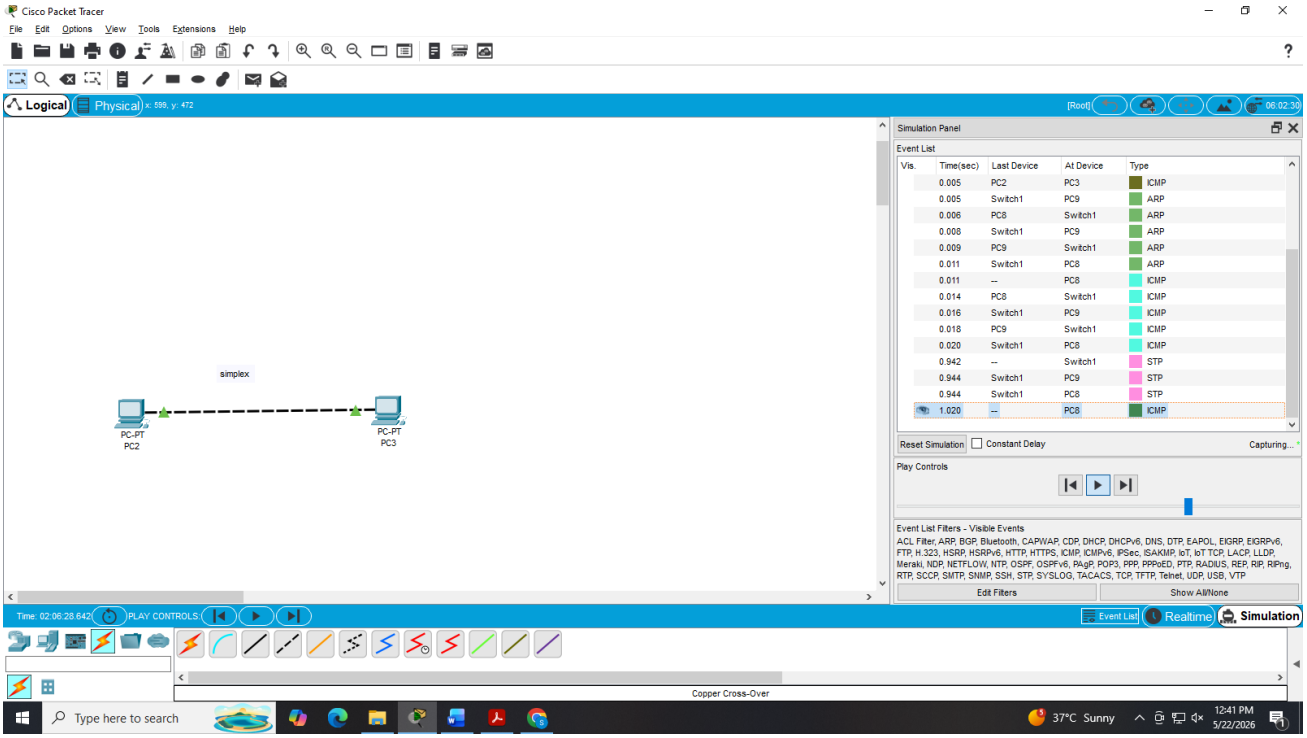
# Example 1: Simple case (1 bit per symbol)
baud1 = 1000 # 1000 symbols/sec
bits_per_sym1 = 1 # e.g., BPSK (Binary Phase Shift Keying)
bit_rate1 = calculate_bit_rate(baud1, bits_per_symbol=bits_per_sym1)
print(f"Baud Rate: {baud1} symbols/sec, Bits/Symbol: {bits_per_sym1} (e.g., BPSK)")
print(f"Calculated Bit Rate: {bit_rate1} bps\n")
```

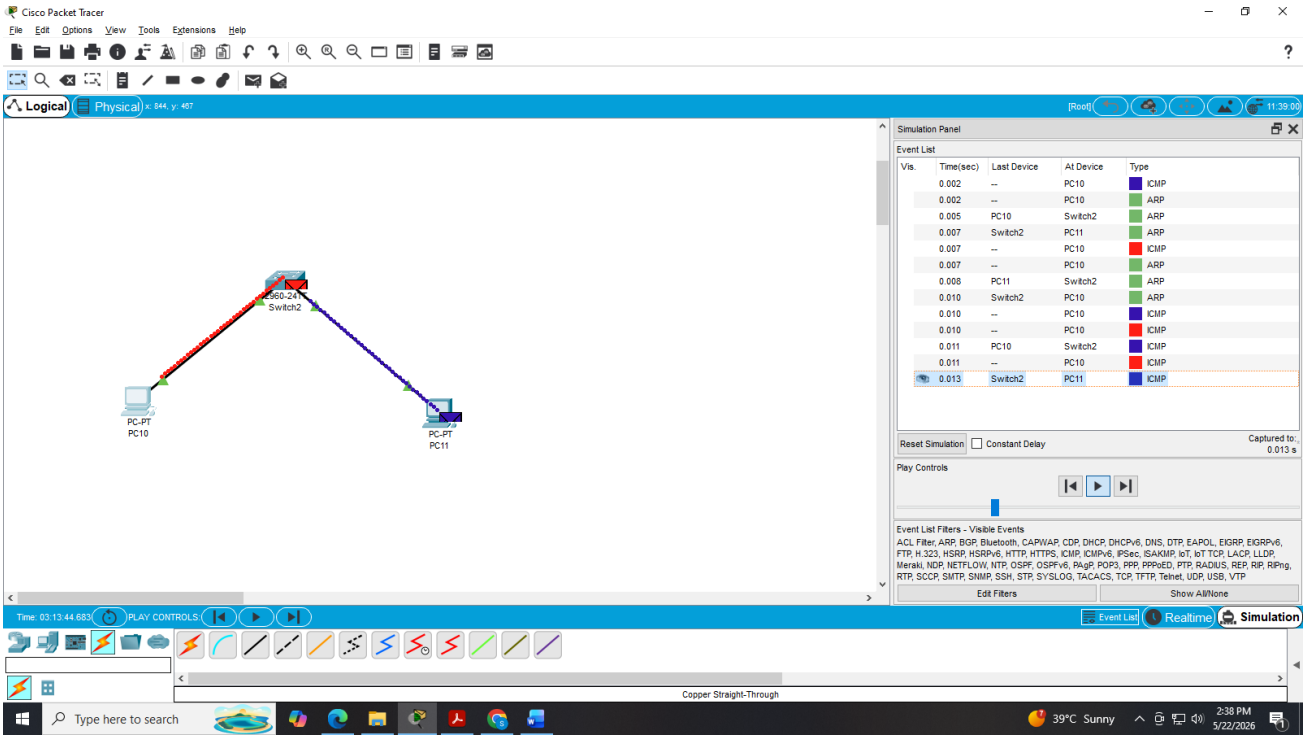
```
# Example 2: QPSK (2 bits per symbol)
baud2 = 1200
num_levels2 = 4 # QPSK uses 4 signal levels (2 bits per symbol)
bit_rate2 = calculate_bit_rate(baud2, num_signal_levels=num_levels2)
print(f"Baud Rate: {baud2} symbols/sec, Signal Levels: {num_levels2} (e.g., QPSK)")
print(f"Calculated Bit Rate: {bit_rate2} bps\n")

# Example 3: 16-QAM (4 bits per symbol)
baud3 = 2400
num_levels3 = 16 # 16-QAM uses 16 signal levels (4 bits per symbol)
bit_rate3 = calculate_bit_rate(baud3, num_signal_levels=num_levels3)
print(f"Baud Rate: {baud3} symbols/sec, Signal Levels: {num_levels3} (e.g., 16-QAM)")
print(f"Calculated Bit Rate: {bit_rate3} bps\n")

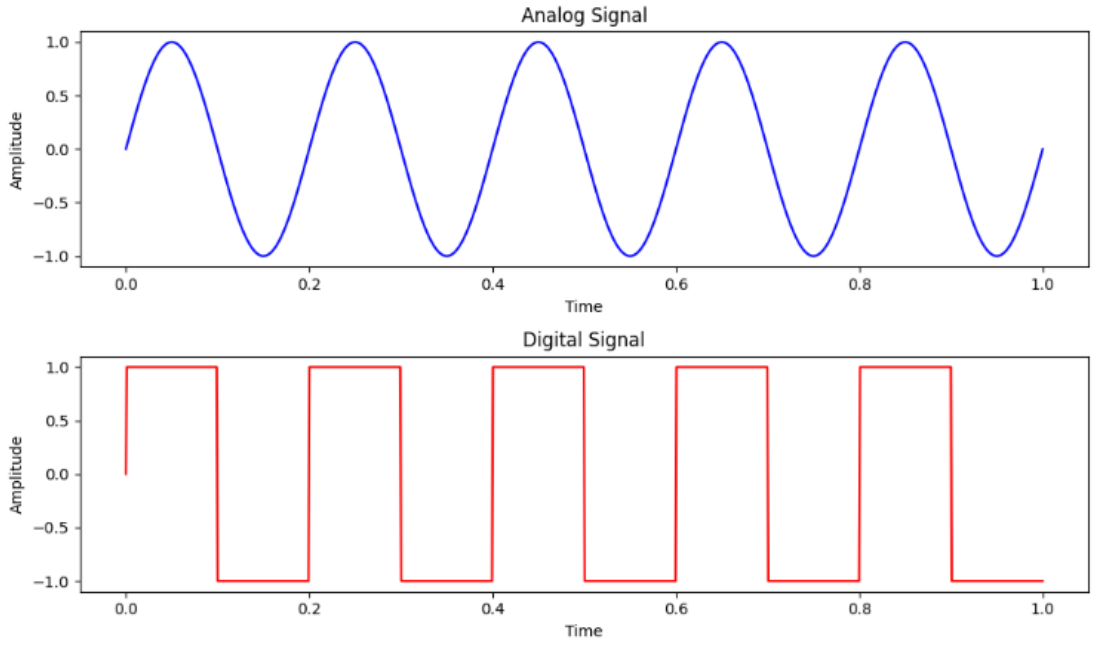
# Example 4: Higher baud rate, same bits per symbol
baud4 = 5000
bits_per_sym4 = 2
bit_rate4 = calculate_bit_rate(baud4, bits_per_symbol=bits_per_sym4)
print(f"Baud Rate: {baud4} symbols/sec, Bits/Symbol: {bits_per_sym4}")
print(f"Calculated Bit Rate: {bit_rate4} bps\n")
```

OUTPUT:


Cisco Packet Tracer
File Edit Options View Tools Extensions Help
Logical Physical
Simulation Panel
Event List
Vis. Time(sec) Last Device At Device Type
0.005 PC2 PC3 ICMP
0.005 Switch1 PC9 ARP
0.006 PC8 Switch1 ARP
0.008 Switch1 PC9 ARP
0.009 PC9 Switch1 ARP
0.011 Switch1 PC8 ARP
0.011 -- PC8 ICMP
0.014 PC8 Switch1 ICMP
0.016 Switch1 PC9 ICMP
0.018 PC9 Switch1 ICMP
0.020 Switch1 PC8 ICMP
0.942 -- Switch1 STP
0.944 Switch1 PC9 STP
0.944 Switch1 PC8 STP
1.020 -- PC8 ICMP
Reset Simulation
Constant Delay
Capturing...
Play Controls
Event List Filters - Visible Events
Edit Filters
Show All/None
simplex
PC-PT PC2
PC-PT PC3
Realtime
Simulation
Copper Cross-Over
Type here to search
37°C Sunny
12:41 PM 5/22/2026


Cisco Packet Tracer
File Edit Options View Tools Extensions Help
Logical Physical
Simulation Panel
Event List
Vis. Time(sec) Last Device At Device Type
0.002 -- PC10 ICMP
0.002 -- PC10 ARP
0.005 PC10 Switch2 ARP
0.007 Switch2 PC11 ARP
0.007 -- PC10 ICMP
0.007 -- PC10 ARP
0.008 PC11 Switch2 ARP
0.010 Switch2 PC10 ARP
0.010 -- PC10 ICMP
0.010 -- PC10 ICMP
0.011 PC10 Switch2 ICMP
0.011 -- PC10 ICMP
0.013 Switch2 PC11 ICMP
Reset Simulation
Constant Delay
Captured to: 0.013 s
Play Controls
Event List Filters - Visible Events
Edit Filters
Show All/None
2960-24TT Switch2
PC-PT PC10
PC-PT PC11
Realtime
Simulation
Copper Straight-Through
Type here to search
39°C Sunny
2:38 PM 5/22/2026

```
Enter number of bits transmitted: 8000
Enter transmission time in seconds: 2
```




```
Bit Rate = 4000.0 bps
```

--- Bit Rate Calculations ---

Baud Rate: 1000 symbols/sec, Bits/Symbol: 1 (e.g., BPSK)

Calculated Bit Rate: 1000 bps

Baud Rate: 1200 symbols/sec, Signal Levels: 4 (e.g., QPSK)

Calculated Bit Rate: 2400 bps

Baud Rate: 2400 symbols/sec, Signal Levels: 16 (e.g., 16-QAM)

Calculated Bit Rate: 9600 bps

Baud Rate: 5000 symbols/sec, Bits/Symbol: 2

Calculated Bit Rate: 10000 bps

RESULT:

Simplex, half-duplex, and full-duplex transmission modes were successfully observed using Cisco Packet Tracer. Analog and digital signals were generated and analyzed using Python, and the bit rate was calculated successfully using.

$$\text{Bit Rate} = \frac{\text{Bits Transmitted}}{\text{Transmission Time}}$$

The experiment verified different communication modes, signal behavior, and data transmission rate calculations.

EX.NO: 3.1

MEASURE AND COMPARE LATENCY,BANDWIDTH, THROUGHPUT USING PING/TRACEROUTE VISUALIZE NETWORK TOPOLOGIES (BUS, STAR, RING, HYBRID) AND PACKET FLOW

AIM:

To measure and compare **Latency** , **Bandwidth** and **Throughput** using **Ping** and **Traceroute** commands in a Cisco networking environment.

Definition :

1. Latency

Latency is the time taken for a packet to travel from source to destination.

Measured using:

ping

Unit: milliseconds (ms)

2. Bandwidth

Bandwidth is the maximum data carrying capacity of a network link.

Formula:

$$\text{Bandwidth} = \frac{\text{Data Size}}{\text{Transmission Time}}$$

Unit: Mbps and Gbps

3. Throughput

Throughput is the actual amount of data successfully transmitted per second.

Formula:

$$\text{Throughput} = \frac{\text{Successful Data Transferred}}{\text{Total Time}}$$

PROCEDURE:

Step 1: Create Network Topology

1. Open Cisco Packet Tracer
2. Add:
    - 2 PCs
    - 2 Switches
    - 1 Router
3. Connect devices using Copper Straight-through cables

Network Topology

PC1 ---- Switch ---- Router ---- Switch ---- PC2

Step2 : Configure IP Addresses

PC1

IP Address: 192.168.1.2
Subnet Mask: 255.255.255.0
Gateway: 192.168.1.1

PC2

IP Address: 192.168.2.2
Subnet Mask: 255.255.255.0
Gateway: 192.168.2.1

Step 3: Configure Router

Router CLI

```
enable
configure terminal

interface gig0/0
ip address 192.168.1.1 255.255.255.0
```

no shutdown

interface gig0/1
ip address 192.168.2.1 255.255.255.0
no shutdown

exit

Step 4: Execute Ping Command to measure Latency

From PC1 command prompt:

ping 192.168.2.2

Step 5: Execute Traceroute Command

tracert 192.168.2.2

Cisco IOS: traceroute 192.168.2.2

Step 6 :Measuring Bandwidth through calculation

Assume:

- Data transmitted = 10 MB
- Transmission time = 2 seconds

Bandwidth:

$$\text{Bandwidth} = \frac{10\ MB}{2\ s} = 5\ MB/s$$

Convert to Mbps:

$$5 \times 8 = 40\ \text{Mbps}$$

Step 7 :Measuring Throughput through calculation

Assume:

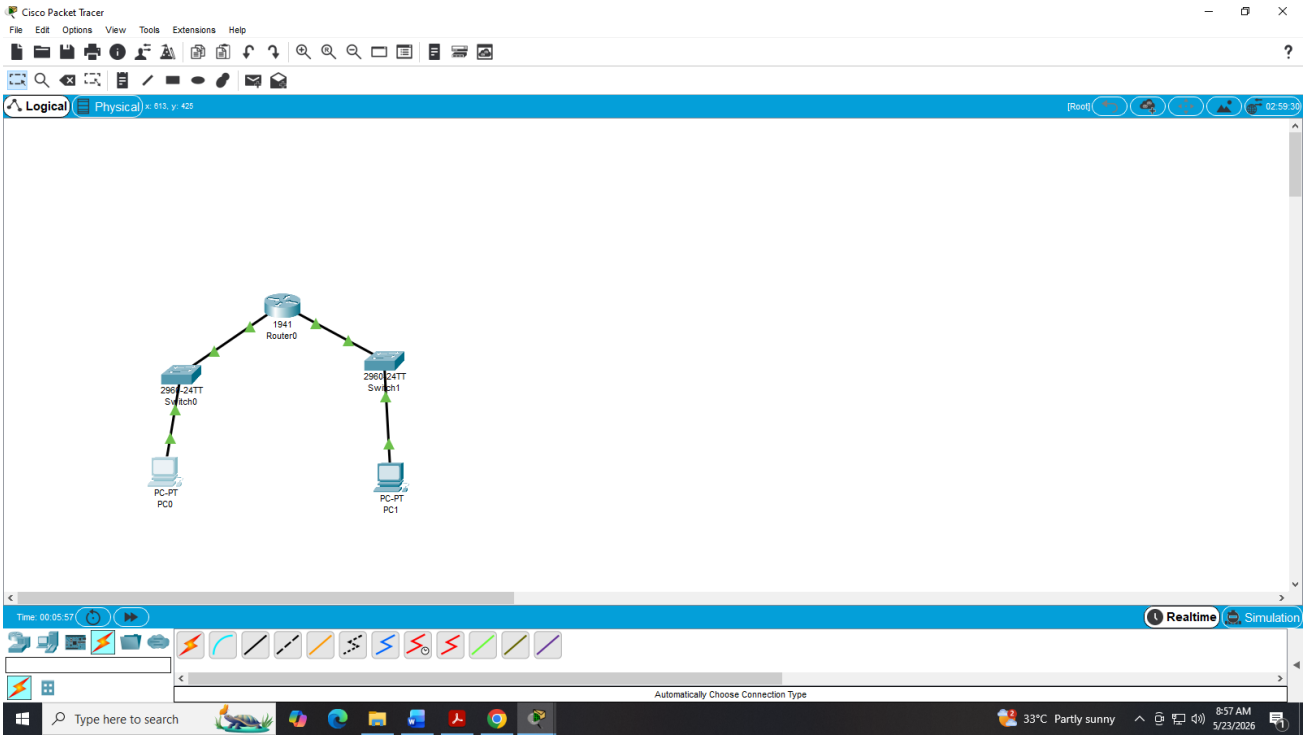
- Successfully received data = 8 MB
- Time = 2 seconds

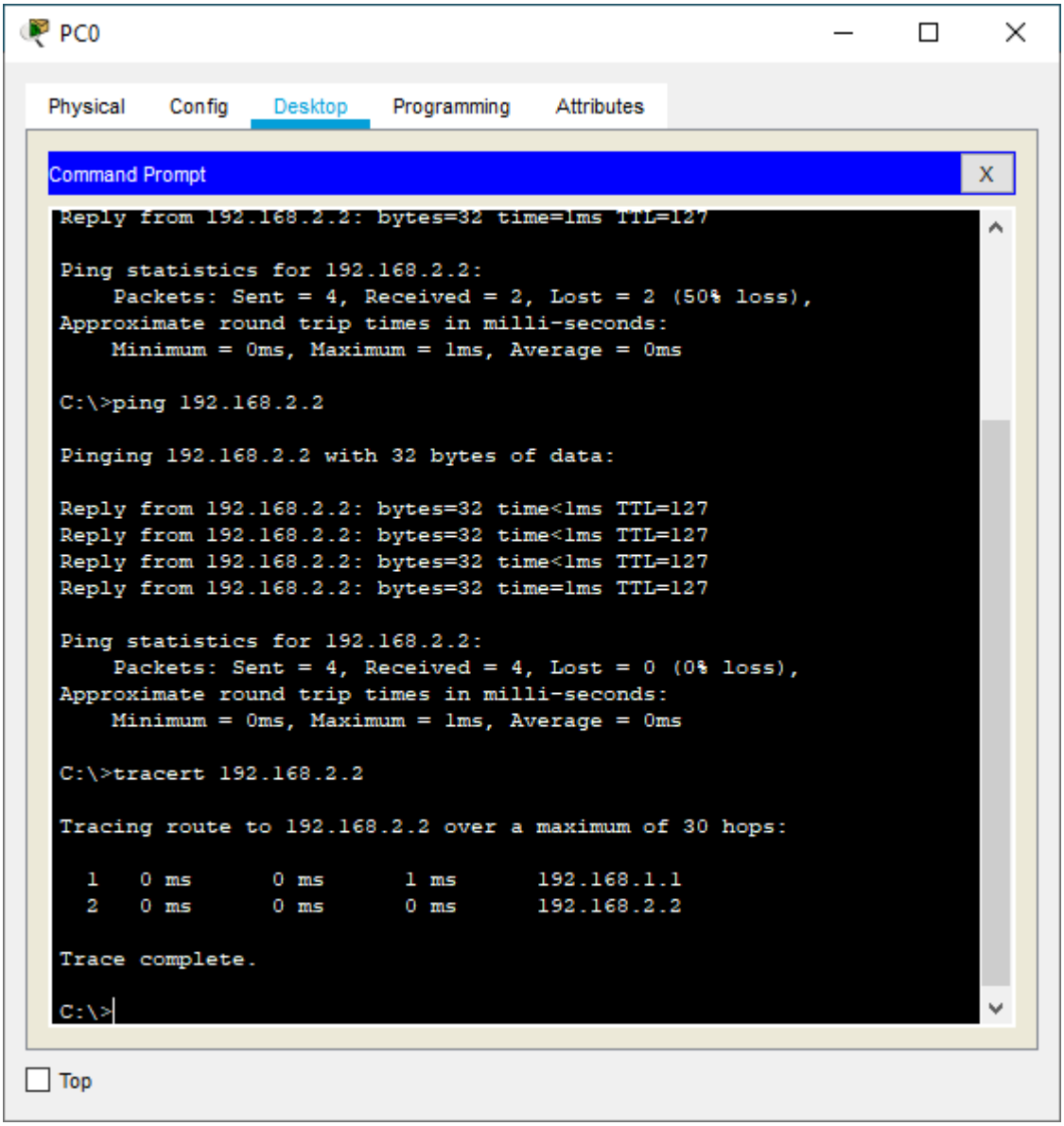
Throughput:

$$\text{Throughput} = \frac{8\ MB}{2\ s} = 4\ MB/s$$

$4 \times 8 = 32$ Mbps **Parameter**	**Latency**	**Bandwidth**	**Throughput**
Definition	Time delay in data transmission	Maximum capacity of network link	Actual data successfully transmitted
Unit	Milliseconds (ms)	Mbps / Gbps	Mbps / Gbps
Measurement Tool	Ping / Traceroute	Calculated from link capacity	File transfer / Ping analysis
Indicates	Network delay	Data carrying capability	Real network performance
Higher Value Meaning	Poor performance	Better capacity	Better performance
Lower Value Meaning	Better performance	Lower capacity	Poor performance
Affected By	Distance, congestion	Cable type, hardware	Traffic, packet loss, latency
Example	2 ms delay	100 Mbps link	75 Mbps actual speed

OUTPUT:


Cisco Packet Tracer
File Edit Options View Tools Extensions Help
Logical
Physical
1941
Router0
2960-24TT
Switch0
2960-24TT
Switch1
PC-PT
PC0
PC-PT
PC1
Realtime
Simulation
Automatically Choose Connection Type
Type here to search
33°C Partly sunny
8:57 AM
5/23/2026



RESULT:

Thus, The experiment demonstrated network performance analysis using Cisco tools with Ping command successfully measured network latency, Traceroute identified the routing path and hop delays. Also, Bandwidth and throughput were calculated and compared.

EX.NO: 3.2

VISUALIZE NETWORK TOPOLOGIES (BUS, STAR, RING, HYBRID) AND PACKET FLOW

AIM:

To visualize and study different network topologies such as Bus, Star, Ring, and Hybrid using Cisco Packet Tracer, and to observe packet flow between devices in the network.

PROCEDURE:

Step 1. Open Cisco Packet Tracer

- Launch the Cisco Packet Tracer application on the system.
- Create a new project file.

Step 2. Create Bus Topology

1. Select **End Devices** from the device menu.
2. Drag and place multiple PCs onto the workspace.
3. Select a **Hub** from the network devices section.
4. Connect all PCs to the hub using **Copper Straight-Through** cables.
5. This setup represents a **Bus Topology**.

Step 3. Create Star Topology

1. Place one **Switch** in the center of the workspace.
2. Add multiple PCs around the switch.
3. Connect each PC directly to the switch using **Copper Straight-Through** cables.
4. This forms a **Star Topology**.

Step 4. Create Ring Topology

1. Place multiple switches or PCs in a circular arrangement.
2. Connect each device to the next device using cables.
3. Connect the last device back to the first device to complete the ring.
4. This creates a **Ring Topology**.

Step 5. Create Hybrid Topology

1. Combine two or more topologies such as Star and Bus.
2. Use switches, hubs, routers, and PCs as needed.
3. Connect the devices appropriately to form a mixed network structure.
4. This setup represents a **Hybrid Topology**.

Step 6. Assign IP Addresses

1. Click on a PC.
2. Go to **Desktop → IP Configuration**.
3. Enter:
    - IP Address
    - Subnet Mask
    - Default Gateway (if required)
4. Repeat for all devices in every topology.

Example:

- PC1: 192.168.1.1
- PC2: 192.168.1.2
- Subnet Mask: 255.255.255.0

Step 7. Verify Device Connectivity

1. Open the **Command Prompt** from the Desktop tab of a PC.
2. Use the ping command to test communication between devices.

    Example:

    ping 192.168.1.2
3. Ensure successful replies are received.

Step 8. Simulate Packet Flow

1. Switch from **Realtime Mode** to **Simulation Mode**.
2. Select the **Add Simple PDU** tool (envelope icon).
3. Click the source device and then the destination device.
4. Observe packet movement through the network.
5. Analyze:
    - Packet path
    - Device communication
    - Transmission process
    - Packet success or failure

Step 9. Save the Project

- Save the Packet Tracer file for future reference and analysis.

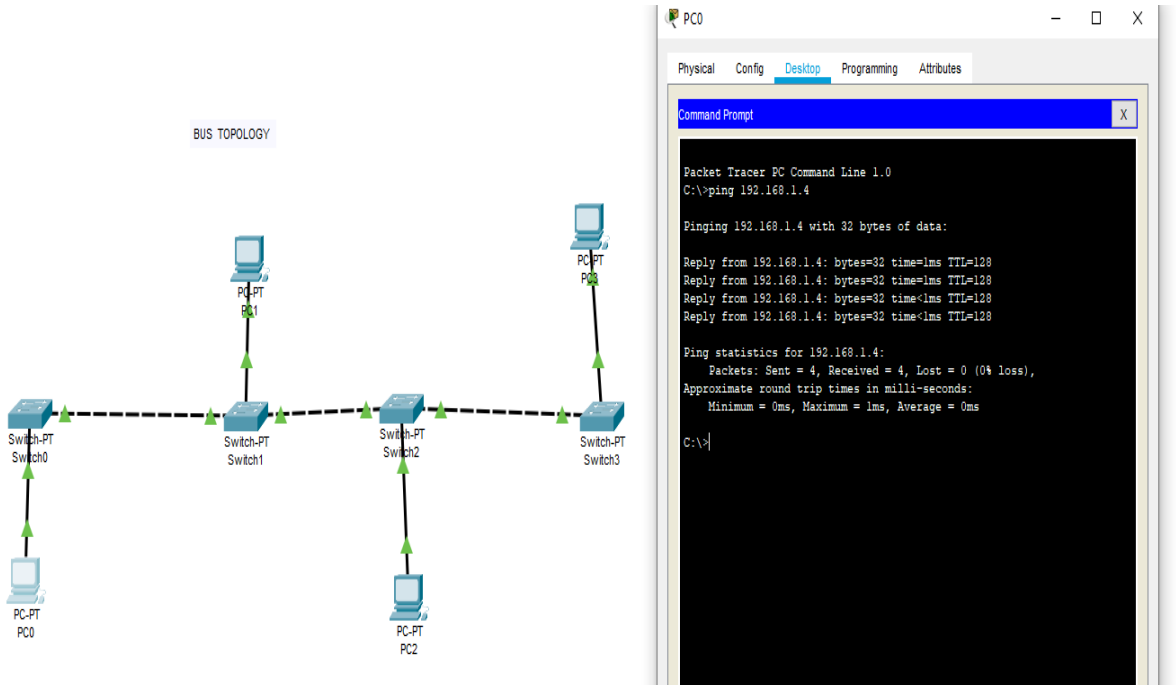
Step 9: Analyze the Topologies

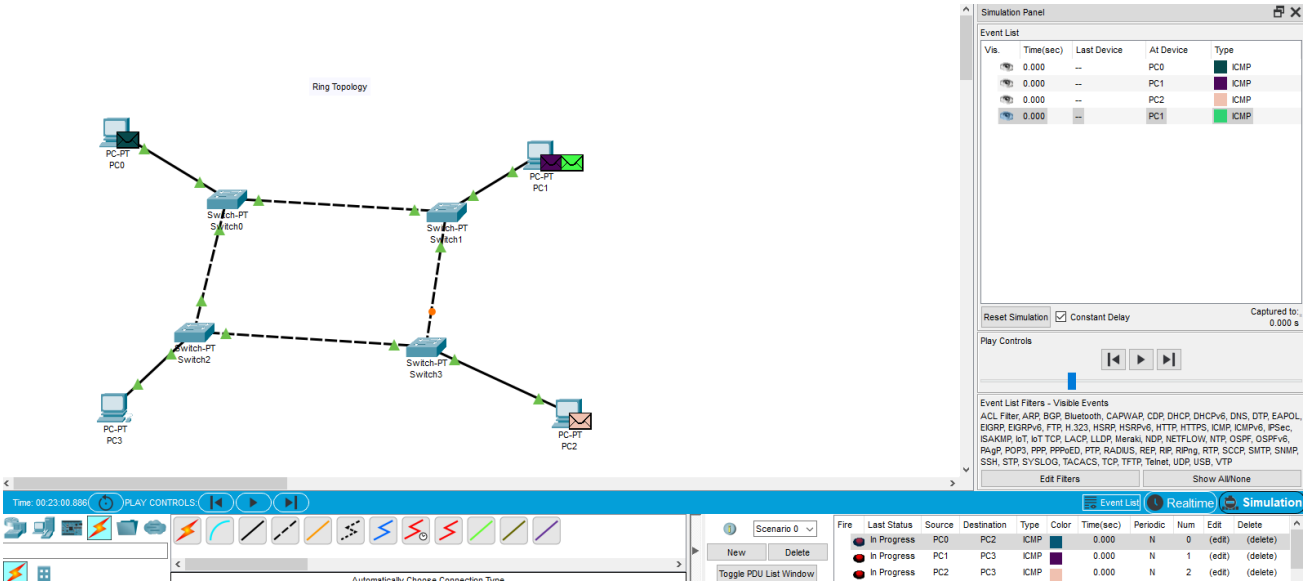
Observe how data travels in each topology and compare their characteristics based on performance, connectivity, fault tolerance, and packet transmission behavior.

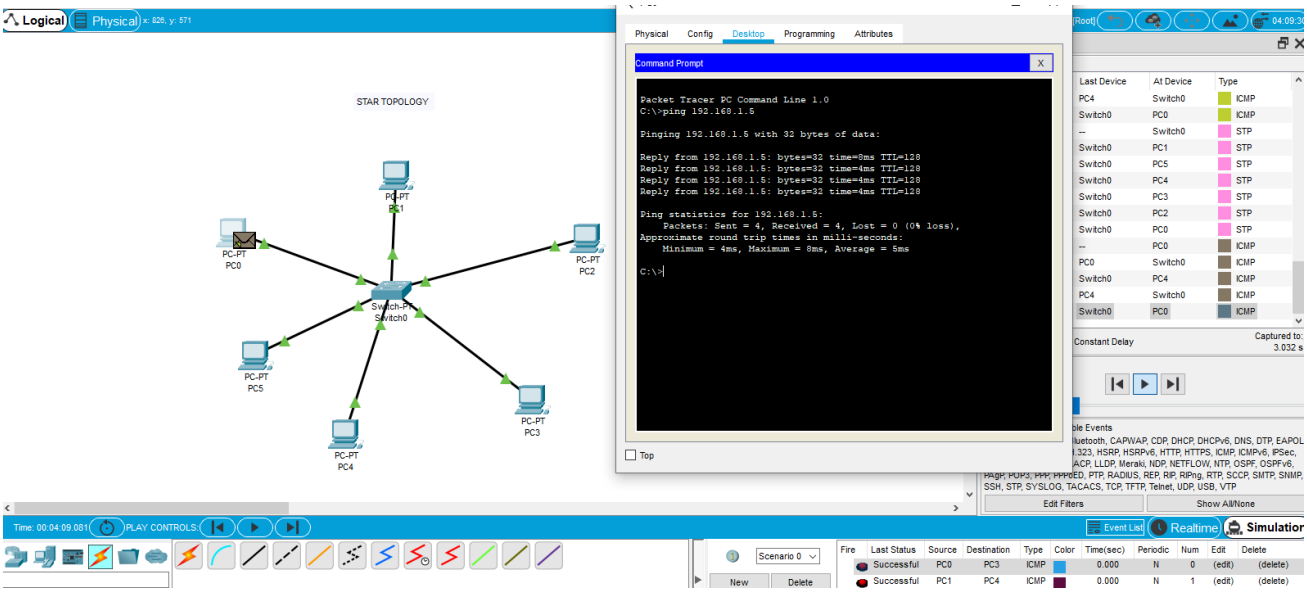
Topology	Performance	Connectivity	Fault Tolerance	Packet Transmission Behavior	Example
Bus Topology	Performance decreases when more devices are added because all devices share the same communication line.	All devices are connected to a single backbone cable.	Low fault tolerance; if the main cable fails, the entire network stops working.	Data travels through the common cable and is received by all devices, but only the destination accepts it.	In a school lab, if the backbone cable is damaged, all connected computers lose communication.
Star Topology	High performance because each device has a dedicated connection to the switch.	Devices are connected through a central switch or hub.	Better fault tolerance; failure of one PC does not affect others, but switch failure affects the whole network.	Data passes through the central switch directly to the destination device.	In an office network, if one computer fails, the remaining systems continue working normally.
Ring Topology	Moderate performance; data flows in an organized sequence.	Each device is connected to two neighboring devices	Medium fault tolerance; failure of one device or cable can	Packets travel in one direction around the ring until reaching the destination.	In a circular network setup, a broken connection between two systems can stop communication

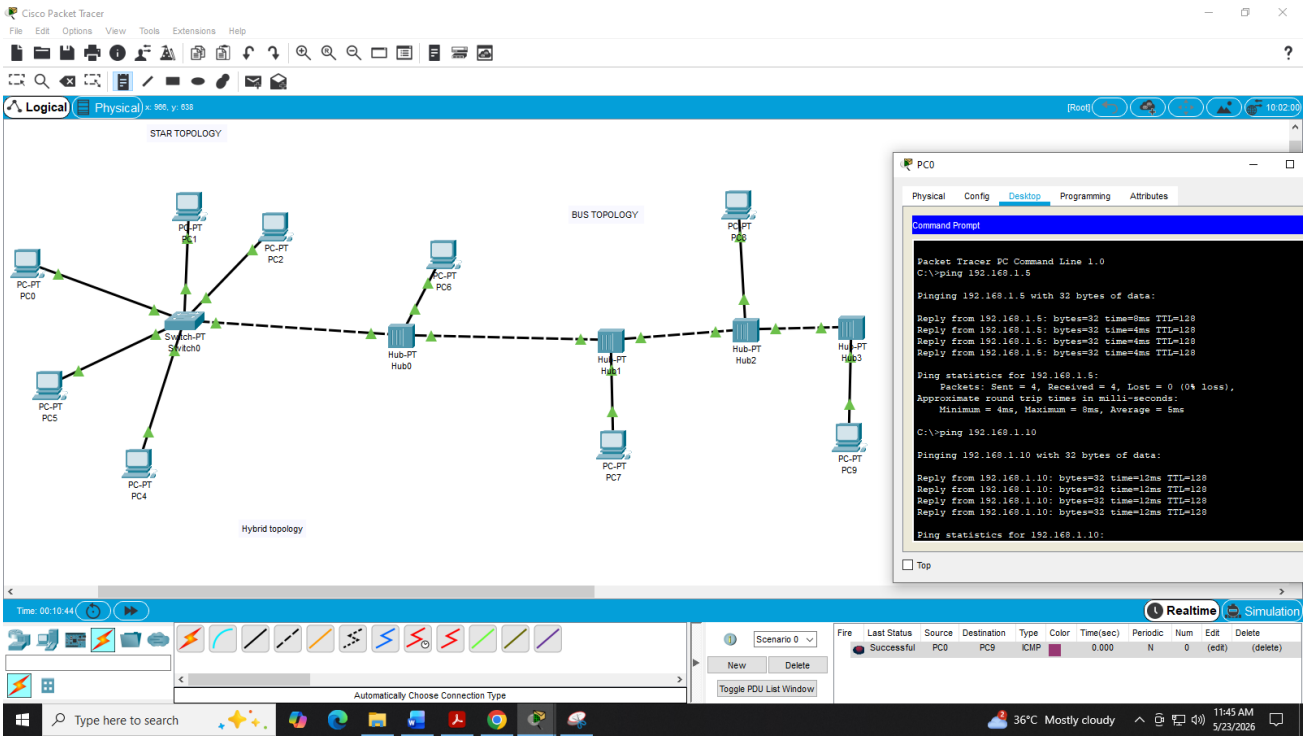
		forming a ring.	disrupt the network unless dual ring is used.		across the network.
Hybrid Topology	High performance due to the combination of multiple topologies.	Combines features of two or more topologies such as Star and Bus.	High fault tolerance depending on the topology design.	Data transmission depends on the combined topology structure.	In a company network, different departments may use Star topology while all departments are connected through a Bus backbone.

OUTPUT:




Ring Topology
PC-PT PC0
PC-PT PC1
Switch-PT Switch0
Switch-PT Switch1
Switch-PT Switch2
Switch-PT Switch3
PC-PT PC3
PC-PT PC2
Simulation Panel
Event List
Vis. Time(sec) Last Device At Device Type
0.000 -- PC0 ICMP
0.000 -- PC1 ICMP
0.000 -- PC2 ICMP
0.000 -- PC1 ICMP
Reset Simulation
Constant Delay
Play Controls
Event List Filters - Visible Events
Edit Filters
Show All/None
Time: 00:23:00.886
PLAY CONTROLS:
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 PC0 PC2 ICMP 0.000 N 0 (edit) (delete)
In Progress PC1 PC3 ICMP 0.000 N 1 (edit) (delete)
In Progress PC2 PC3 ICMP 0.000 N 2 (edit) (delete)


Logical
Physical
STAR TOPOLOGY
PC-PT PC1
PC-PT PC0
PC-PT PC2
Switch-PT Switch0
PC-PT PC5
PC-PT PC3
PC-PT PC4
Physical Config Desktop Programming Attributes
Command Prompt
Packet Tracer PC Command Line 1.0
C:\>ping 192.168.1.5
Pinging 192.168.1.5 with 32 bytes of data:
Reply from 192.168.1.5: bytes=32 time=8ms TTL=128
Reply from 192.168.1.5: bytes=32 time=4ms TTL=128
Reply from 192.168.1.5: bytes=32 time=4ms TTL=128
Reply from 192.168.1.5: bytes=32 time=4ms TTL=128
Ping statistics for 192.168.1.5:
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
Minimum = 4ms, Maximum = 8ms, Average = 5ms
C:\>
Top
Last Device At Device Type
PC4 Switch0 ICMP
Switch0 PC0 ICMP
-- Switch0 STP
Switch0 PC1 STP
Switch0 PC5 STP
Switch0 PC4 STP
Switch0 PC3 STP
Switch0 PC2 STP
Switch0 PC0 STP
-- PC0 ICMP
PC0 Switch0 ICMP
Switch0 PC4 ICMP
PC4 Switch0 ICMP
Switch0 PC0 ICMP
Constant Delay
Edit Filters
Show All/None
Time: 00:04:09.001
PLAY CONTROLS:
Event List
Realtime
Simulation
Scenario 0
New
Delete
Fire Last Status Source Destination Type Color Time(sec) Periodic Num Edit Delete
Successful PC0 PC3 ICMP 0.000 N 0 (edit) (delete)
Successful PC1 PC4 ICMP 0.000 N 1 (edit) (delete)



RESULT:

Thus, the Bus, Star, Ring, and Hybrid network topologies were successfully created and visualized using Cisco Packet Tracer. Packet flow between devices was successfully simulated and analyzed, demonstrating network communication, connectivity, and data transmission across different topologies.

EX.NO: 4

IMPLEMENT CRC AND PARITY CHECKING USING PYTHON AND DEMONSTRATE HAMMING CODE FOR ERROR CORRECTION

AIM:

To implement and demonstrate three fundamental error detection and correction techniques in data communication.

Explanation: Error Detection and Correction

Error detection and correction techniques are crucial in data transmission and storage to ensure data integrity. They involve adding redundant information to the original data, which can then be used to detect or even correct errors introduced during transmission or storage.

We will explore three common methods:

1. **Parity Checking**: A simple error detection method. Parity checking is one of the simplest error detection techniques. It involves adding a single bit, called the parity bit, to a block of data bits. The value of the parity bit is chosen such that the total number of 1s in the data block (including the parity bit) is either even (for even parity) or odd (for odd parity).

   Even Parity: The total count of 1s is even.

   Odd Parity: The total count of 1s is odd.

2. **Cyclic Redundancy Check (CRC)**: A more robust error detection method. CRC is a powerful and widely used error-detection code. It is based on polynomial division over a finite field (specifically, GF(2)). The sender computes a sequence of check bits, called the Frame Check Sequence (FCS), and appends them to the data. The receiver performs the same calculation; if the result (remainder) is zero, no error is detected. If the remainder is non-zero, an error has occurred.

   Key steps for CRC generation:

   Append n zeros to the data (where n is the degree of the generator polynomial).

Divide the resulting data polynomial by the generator polynomial using binary (XOR) division.

The remainder of this division is the CRC checksum.

Replace the appended zeros with this checksum.

3. **Hamming Code**: An error correction code capable of detecting and correcting single-bit errors. Hamming code is an error-**correcting** code that can detect and correct single-bit errors. It adds redundant bits (parity bits) at specific positions within the data block. These parity bits are calculated based on different combinations of data bits.

   Key principles:

   Parity Bit Positions: Parity bits are placed at positions that are powers of 2 (1, 2, 4, 8, ...).

   Data Bit Positions: Remaining positions are for data bits.

   Parity Calculation: Each parity bit checks a specific set of bits in the codeword.

   P1 checks bits at positions 1, 3, 5, 7, 9, 11, ...

   P2 checks bits at positions 2, 3, 6, 7, 10, 11, ...

   P4 checks bits at positions 4, 5, 6, 7, 12, 13, ...

   And so on.

   By checking the parity bits, an error syndrome (a binary number) can be generated. If the syndrome is non-zero, its value indicates the position of the erroneous bit, allowing it to be corrected.

PROGRAM:

Program 1 : Parity Checking

```
def generate_parity_bit(data_bits, parity_type='even'):
    """
```

```
    Generates a parity bit for a given string of binary data bits.

    Args:
        data_bits (str): A string of binary digits (e.g., '10110').
        parity_type (str): 'even' for even parity, 'odd' for odd parity.

    Returns:
        str: The parity bit ('0' or '1').
    """
    count_ones = data_bits.count('1')
    if parity_type == 'even':
        return '0' if count_ones % 2 == 0 else '1'
    elif parity_type == 'odd':
        return '1' if count_ones % 2 == 0 else '0'
    else:
        raise ValueError("parity_type must be 'even' or 'odd'")

def check_parity(received_data_with_parity, parity_type='even'):
    """
    Checks the parity of received data with its parity bit.

    Args:
        received_data_with_parity (str): A string of binary digits including the parity bit.
        parity_type (str): 'even' for even parity, 'odd' for odd parity.

    Returns:
        bool: True if parity is correct, False otherwise.
    """
    count_ones = received_data_with_parity.count('1')
    if parity_type == 'even':
        return count_ones % 2 == 0
    elif parity_type == 'odd':
        return count_ones % 2 != 0
    else:
        raise ValueError("parity_type must be 'even' or 'odd'")

# Demonstration
original_data = '10110101'

# Even Parity
even_parity_bit = generate_parity_bit(original_data, 'even')
even_transmitted_data = original_data + even_parity_bit
print(f"Original Data: {original_data}")
print(f"Even Parity Bit: {even_parity_bit}")
```

```
print(f"Transmitted Data (Even Parity): {even_transmitted_data}")

# Scenario 1: No error (Even Parity)
received_data_no_error_even = even_transmitted_data
print(f"Received Data (no error): {received_data_no_error_even}")
print(f"Parity Check (no error): {check_parity(received_data_no_error_even, 'even')}")

# Scenario 2: Single bit error (Even Parity)
error_index = 3 # Flip the 4th bit (0-indexed)
received_data_error_even = list(even_transmitted_data)
received_data_error_even[error_index] = '1' if received_data_error_even[error_index] == '0' else '0'
received_data_error_even = ''.join(received_data_error_even)
print(f"Received Data (single error): {received_data_error_even}")
print(f"Parity Check (single error): {check_parity(received_data_error_even, 'even')}")

print("\n" + "-"*30 + "\n")

# Odd Parity
odd_parity_bit = generate_parity_bit(original_data, 'odd')
odd_transmitted_data = original_data + odd_parity_bit
print(f"Original Data: {original_data}")
print(f"Odd Parity Bit: {odd_parity_bit}")
print(f"Transmitted Data (Odd Parity): {odd_transmitted_data}")

# Scenario 1: No error (Odd Parity)
received_data_no_error_odd = odd_transmitted_data
print(f"Received Data (no error): {received_data_no_error_odd}")
print(f"Parity Check (no error): {check_parity(received_data_no_error_odd, 'odd')}")

# Scenario 2: Single bit error (Odd Parity)
error_index = 5 # Flip the 6th bit (0-indexed)
received_data_error_odd = list(odd_transmitted_data)
received_data_error_odd[error_index] = '1' if received_data_error_odd[error_index] == '0' else '0'
received_data_error_odd = ''.join(received_data_error_odd)
print(f"Received Data (single error): {received_data_error_odd}")
print(f"Parity Check (single error): {check_parity(received_data_error_odd, 'odd')}")
```

Program 2 : Cyclic Redundancy Check (CRC)

```
def xor(a, b):
    """
    Performs a bitwise XOR operation on two binary strings.
    """
```

```
    result = []
    for i in range(1, len(b)):
        if a[i] == b[i]:
            result.append('0')
        else:
            result.append('1')
    return ''.join(result)

def crc_divide(dividend, divisor):
    """
    Performs binary division for CRC, returning the remainder.
    """
    pick = len(divisor)
    tmp = dividend[0: pick]

    while pick < len(dividend):
        if tmp[0] == '1':
            tmp = xor(divisor, tmp) + dividend[pick]
        else:
            tmp = xor('0' * pick, tmp) + dividend[pick]
        pick += 1

    if tmp[0] == '1':
        tmp = xor(divisor, tmp)
    else:
        tmp = xor('0' * pick, tmp)

    return tmp

def generate_crc(data, generator):
    """
    Generates the CRC checksum for the given data and generator polynomial.

    Args:
        data (str): The binary data string (e.g., '1010001101').
        generator (str): The binary string representing the generator polynomial (e.g., '1011').

    Returns:
        str: The CRC checksum.
    """
    n = len(generator) - 1
    appended_data = data + '0' * n
    remainder = crc_divide(appended_data, generator)
    return remainder
```

```
def check_crc(received_data_with_crc, generator):
    """
    Checks the CRC of the received data.

    Args:
        received_data_with_crc (str): The binary string of received data including CRC.
        generator (str): The binary string representing the generator polynomial.

    Returns:
        bool: True if no error is detected (remainder is all zeros), False otherwise.
    """
    remainder = crc_divide(received_data_with_crc, generator)
    return all(bit == '0' for bit in remainder)

# Demonstration
data = '1101001110'
generator_polynomial = '1011' # Corresponds to x^3 + x + 1

# Sender side
crc_checksum = generate_crc(data, generator_polynomial)
transmitted_data = data + crc_checksum
print(f"Original Data: {data}")
print(f"Generator Polynomial: {generator_polynomial}")
print(f"CRC Checksum: {crc_checksum}")
print(f"Transmitted Data: {transmitted_data}")

# Receiver side

# Scenario 1: No error
received_data_no_error = transmitted_data
print(f"\nReceived Data (no error): {received_data_no_error}")
print(f"CRC Check (no error): {check_crc(received_data_no_error, generator_polynomial)}")

# Scenario 2: Single bit error
error_index = 4 # Flip the 5th bit
received_data_error = list(transmitted_data)
received_data_error[error_index] = '1' if received_data_error[error_index] == '0' else '0'
received_data_error = ''.join(received_data_error)
print(f"Received Data (single error): {received_data_error}")
print(f"CRC Check (single error): {check_crc(received_data_error, generator_polynomial)}")

# Scenario 3: Multiple bit errors (CRC can detect most but not all)
error_index_1 = 2
```

```
error_index_2 = 7
received_data_multiple_errors = list(transmitted_data)
received_data_multiple_errors[error_index_1] = '1' if received_data_multiple_errors[error_index_1]
== '0' else '0'
received_data_multiple_errors[error_index_2] = '1' if received_data_multiple_errors[error_index_2]
== '0' else '0'
received_data_multiple_errors = ''.join(received_data_multiple_errors)
print(f"Received Data (multiple errors): {received_data_multiple_errors}")
print(f"CRC Check (multiple errors): {check_crc(received_data_multiple_errors,
generator_polynomial)}")
```

Program 3 : Hamming Code

```
def calculate_parity_bits_count(data_length):
    """
    Calculates the number of parity bits needed for a given data length.
    """
    k = data_length
    r = 0
    while (2**r) < (k + r + 1):
        r += 1
    return r

def encode_hamming(data_bits):
    """
    Encodes data bits into a Hamming codeword (even parity).
    """
    k = len(data_bits)
    r = calculate_parity_bits_count(k)
    n = k + r # Total length of codeword

    # Initialize codeword with placeholders for parity bits
    codeword = ['0'] * n
    j = 0 # Index for data_bits
    for i in range(n):
        # Positions are 1-indexed for Hamming code logic
        if (i + 1) & (i) == 0: # Check if (i+1) is a power of 2
```

```
            # This is a parity bit position
            continue
        else:
            # This is a data bit position
            codeword[i] = data_bits[j]
            j += 1

    # Calculate parity bits
    for i in range(r):
        p_pos = (2**i) - 1 # 0-indexed position of parity bit
        count_ones = 0
        for bit_pos in range(n):
            # Check if this bit is covered by the current parity bit
            if ((bit_pos + 1) >> i) & 1: # (bit_pos+1) is 1-indexed
                if (bit_pos + 1) != (p_pos + 1): # Don't include the parity bit itself
                    if codeword[bit_pos] == '1':
                        count_ones += 1
        codeword[p_pos] = '0' if count_ones % 2 == 0 else '1'

    return ''.join(codeword)

def detect_and_correct_hamming(received_codeword):
    """
    Detects and corrects a single bit error in a Hamming codeword (even parity).
    """
    n = len(received_codeword)
    r = 0
    while (2**r) < n + 1:
        r += 1

    syndrome_bits = []
    for i in range(r):
        p_pos = (2**i) - 1 # 0-indexed position of parity bit
        count_ones = 0
        for bit_pos in range(n):
            if ((bit_pos + 1) >> i) & 1: # (bit_pos+1) is 1-indexed
                if received_codeword[bit_pos] == '1':
                    count_ones += 1
        syndrome_bits.append('1' if count_ones % 2 != 0 else '0')

    syndrome = int(''.join(syndrome_bits[::-1]), 2) # Convert reversed syndrome to integer

    if syndrome == 0:
        return received_codeword, 0, "No error detected."
```

```
    else:
        # Error detected at 'syndrome' position (1-indexed)
        error_index = syndrome - 1 # 0-indexed
        corrected_codeword = list(received_codeword)
        corrected_codeword[error_index] = '1' if corrected_codeword[error_index] == '0' else '0'
        return ''.join(corrected_codeword), syndrome, f"Error detected and corrected at position
{syndrome} (1-indexed)."

# Demonstration
original_data = '1011'

# Encode
hamming_codeword = encode_hamming(original_data)
print(f"Original Data: {original_data}")
print(f"Hamming Codeword: {hamming_codeword}")

# Scenario 1: No error
received_codeword_no_error = hamming_codeword
corrected_codeword, error_pos, message =
detect_and_correct_hamming(received_codeword_no_error)
print(f"\nReceived Codeword (no error): {received_codeword_no_error}")
print(f"Corrected Codeword: {corrected_codeword}")
print(f"Error Position: {error_pos}, Message: {message}")

# Scenario 2: Single bit error
error_index = 2 # Introduce error at 3rd bit (0-indexed)
received_codeword_error = list(hamming_codeword)
received_codeword_error[error_index] = '1' if received_codeword_error[error_index] == '0' else '0'
received_codeword_error = ''.join(received_codeword_error)

corrected_codeword, error_pos, message = detect_and_correct_hamming(received_codeword_error)
print(f"\nReceived Codeword (single error at pos {error_index+1}): {received_codeword_error}")
print(f"Corrected Codeword: {corrected_codeword}")
print(f"Error Position: {error_pos}, Message: {message}")

# Scenario 3: Multiple bit errors (Hamming code can only correct single bit errors)
error_index_1 = 1
error_index_2 = 5
received_codeword_multiple_errors = list(hamming_codeword)
received_codeword_multiple_errors[error_index_1] = '1' if
received_codeword_multiple_errors[error_index_1] == '0' else '0'
received_codeword_multiple_errors[error_index_2] = '1' if
received_codeword_multiple_errors[error_index_2] == '0' else '0'
received_codeword_multiple_errors = ''.join(received_codeword_multiple_errors)
```

```
corrected_codeword, error_pos, message =
detect_and_correct_hamming(received_codeword_multiple_errors)
print(f"\nReceived Codeword (multiple errors): {received_codeword_multiple_errors}")
print(f"Corrected Codeword: {corrected_codeword}")
print(f"Error Position: {error_pos}, Message: {message}") # It will try to 'correct' one, but result will still be wrong.
```

OUTPUT:

Program 1 : Parity Checking

```
Original Data: 10110101
Even Parity Bit: 1
Transmitted Data (Even Parity): 101101011
Received Data (no error): 101101011
Parity Check (no error): True
Received Data (single error): 101001011
Parity Check (single error): False

------------------------------

Original Data: 10110101
Odd Parity Bit: 0
Transmitted Data (Odd Parity): 101101010
Received Data (no error): 101101010
Parity Check (no error): True
Received Data (single error): 101100010
Parity Check (single error): False
```

Program 2 : Cyclic Redundancy Check (CRC)

```
Original Data: 1101001110
Generator Polynomial: 1011
CRC Checksum: 001
Transmitted Data: 1101001110001

Received Data (no error): 1101001110001
CRC Check (no error): True
Received Data (single error): 1101101110001
CRC Check (single error): False
Received Data (multiple errors): 1111001010001
CRC Check (multiple errors): False
```

Program 3 : Hamming Code

```
Original Data: 1011
Hamming Codeword: 0110011

Received Codeword (no error): 0110011
Corrected Codeword: 0110011
Error Position: 0, Message: No error detected.

Received Codeword (single error at pos 3): 0100011
Corrected Codeword: 0110011
Error Position: 3, Message: Error detected and corrected at position 3 (1-indexed).

Received Codeword (multiple errors): 0010001
Corrected Codeword: 0011001
Error Position: 4, Message: Error detected and corrected at position 4 (1-indexed).
```

RESULT:

Thus, The program successfully implemented and demonstrated three important error detection and correction techniques used in data communication and storage systems: Parity Checking, Cyclic Redundancy Check (CRC), and Hamming Code.

EX.NO: 5

SIMULATE STOP-AND-WAIT AND SLIDING WINDOW PROTOCOLS FRAME CREATION USING BIT STUFFING/CHARACTER STUFFING

AIM:

To simulate the working of Stop-and-Wait and Sliding Window protocols used in data communication for reliable data transfer, and to implement frame creation techniques using Bit Stuffing and Character Stuffing for proper data framing and synchronization during transmission.

Explanation:Data Link Layer Protocols: Flow Control and Framing

The data link layer is responsible for reliable data transfer between adjacent network nodes. Key functions include:

- **Flow Control**: Managing the rate of data transmission to prevent a fast sender from overwhelming a slow receiver.
- **Framing**: Dividing the stream of bits from the network layer into discrete units called frames.
- **Error Control**: Detecting and potentially correcting errors that occur during transmission (as demonstrated previously with Parity, CRC, and Hamming Code).

PROGRAM:

1. **Stop-and-Wait Protocol (Flow Control)**

The Stop-and-Wait protocol is a simple flow control mechanism where the sender transmits one frame and then waits for an acknowledgment (ACK) from the receiver before sending the next frame. This ensures that the receiver is not overwhelmed. It's often used with a timer to handle lost frames or ACKs.

Key characteristics:

- **Simplicity**: Easy to implement.
- **Inefficiency**: Can be very inefficient, especially over long distances or high-latency links, as the sender spends a lot of time waiting.

- **Error Handling**: With sequence numbers and timers, it can recover from lost frames and ACKs.

```
import time
import random
def simulate_stop_and_wait(frames, loss_rate=0.2, corrupt_rate=0.1, timeout=2):
    """
    Simulates the Stop-and-Wait protocol.
    Args:
        frames (list): A list of data frames to be sent.
        loss_rate (float): Probability of a frame or ACK being lost (0 to 1).
        corrupt_rate (float): Probability of a frame or ACK being corrupted (0 to 1).
        timeout (int): Timeout duration in seconds for waiting for an ACK.
    """
    print("--- Stop-and-Wait Protocol Simulation ---")
    sender_sequence_number = 0
    receiver_expected_sequence_number = 0
    sent_frames = {}
    for frame_data in frames:
        while True:
            frame = {
                'seq_num': sender_sequence_number,
                'data': frame_data,
```

```
            'checksum': f"checksum_for_{frame_data}" # Simplified checksum
        }
        print(f"\nSender: Sending Frame {frame['seq_num']} with data '{frame['data']}'")
        # Simulate transmission
        if random.random() < loss_rate:
            print(f"Sender: Frame {frame['seq_num']} LOST in transmission!")
            time.sleep(1) # Simulate delay for retransmission
            continue
        if random.random() < corrupt_rate:
            print(f"Sender: Frame {frame['seq_num']} CORRUPTED in transmission! (Simulating error)")
            # Receiver would detect corruption and discard, leading to timeout
        else:
            print(f"Sender: Frame {frame['seq_num']} successfully sent.")
        sent_frames[sender_sequence_number] = frame
        # Simulate receiver's actions
        ack_received = False
        start_time = time.time()
        while time.time() - start_time < timeout:
            # Simulate receiver processing and sending ACK
            if 'received_frames' not in locals():
                received_frames = [] # Initialize if not already present
```

```
            if frame['seq_num'] == receiver_expected_sequence_number and random.random() >= corrupt_rate: # Simulate successful reception and uncorrupted frame
                print(f"Receiver: Received Frame {frame['seq_num']} correctly.")
                if 'data' in frame: # Only append if it's a valid data frame
                    received_frames.append(frame['data'])
                ack = {
                    'seq_num': receiver_expected_sequence_number,
                    'type': 'ACK',
                    'checksum': f"checksum_for_ACK_{receiver_expected_sequence_number}"
                }
                # Simulate ACK loss/corruption
                if random.random() < loss_rate:
                    print(f"Receiver: ACK {ack['seq_num']} LOST!")
                    break # Wait for sender timeout
                if random.random() < corrupt_rate:
                    print(f"Receiver: ACK {ack['seq_num']} CORRUPTED!")
                    break # Wait for sender timeout
                print(f"Receiver: Sending ACK {ack['seq_num']}")
                # Assuming ACK reaches sender if not lost/corrupted
                ack_received = True
                receiver_expected_sequence_number = 1 - receiver_expected_sequence_number # Flip sequence number
                break
```

```
            elif frame['seq_num'] != receiver_expected_sequence_number and random.random() >= corrupt_rate:
                print(f"Receiver: Received Frame {frame['seq_num']} (duplicate or out-of-order). Discarding and Resending ACK {1 - receiver_expected_sequence_number}")
                # Send ACK for the *expected* frame again
                ack = {
                    'seq_num': 1 - receiver_expected_sequence_number,
                    'type': 'ACK',
                    'checksum': f"checksum_for_ACK_{1 - receiver_expected_sequence_number}"
                }
                if random.random() < loss_rate:
                    print(f"Receiver: ACK {ack['seq_num']} LOST (re-sent ACK)!")
                    break
                if random.random() < corrupt_rate:
                    print(f"Receiver: ACK {ack['seq_num']} CORRUPTED (re-sent ACK)!")
                    break
                print(f"Receiver: Sending ACK {ack['seq_num']} (for expected frame {1 - receiver_expected_sequence_number})")
                ack_received = True
                break
            else:
                # Frame corrupted or something else, receiver would ignore
                break
```

```
            if ack_received and ack['seq_num'] == sender_sequence_number: # Check if received ACK matches sent frame's sequence number
                print(f"Sender: Received ACK {ack['seq_num']}. Moving to next frame.")
                sender_sequence_number = 1 - sender_sequence_number # Flip sequence number
                break # Move to next frame in the outer loop
            else:
                print(f"Sender: Timeout or unexpected ACK. Retransmitting Frame {frame['seq_num']}.")
                time.sleep(0.5) # Small delay before retransmission
    print("\n--- Simulation End ---")
    print(f"Successfully received data: {received_frames}")

# Demonstration
data_to_send = ['A', 'B', 'C', 'D']
simulate_stop_and_wait(data_to_send, loss_rate=0.3, corrupt_rate=0.1, timeout=3)
```

2. **Sliding Window Protocol (Flow Control)**

Sliding Window protocols improve efficiency over Stop-and-Wait by allowing the sender to transmit multiple frames before receiving acknowledgments. This is achieved using a "window" of frames that can be in transit. Both the sender and receiver maintain a window of acceptable sequence numbers.

Key characteristics:

- **Increased Throughput**: Can utilize the channel more efficiently, especially over high-latency links.
- **Window Size**: Determines the number of unacknowledged frames a sender can transmit.
- **Go-Back-N** and **Selective Repeat** are common variants. This simulation will be a simplified Go-Back-N like approach.

```
def simulate_sliding_window(frames, window_size, loss_rate=0.2, corrupt_rate=0.1, timeout=5):
    """
    Simulates a simplified Sliding Window protocol (Go-Back-N style).
    Args:
        frames (list): A list of data frames to be sent.
        window_size (int): The maximum number of unacknowledged frames.
        loss_rate (float): Probability of a frame or ACK being lost (0 to 1).
        corrupt_rate (float): Probability of a frame or ACK being corrupted (0 to 1).
        timeout (int): Timeout duration in seconds for waiting for an ACK.
    """
    print(f"--- Sliding Window Protocol Simulation (Window Size: {window_size}) ---")
    # Sender state
    send_base = 0
    next_seq_num = 0
    buffered_frames = {}
    sender_timers = {}
    # Receiver state
    receive_expected_seq_num = 0
    received_data_in_order = []
    num_frames = len(frames)
    while len(received_data_in_order) < num_frames:
        # Sender: Send new frames within the window
```

```
while next_seq_num < send_base + window_size and next_seq_num < num_frames:
    frame = {
        'seq_num': next_seq_num,
        'data': frames[next_seq_num],
        'checksum': f"checksum_for_{frames[next_seq_num]}"
    }
    buffered_frames[next_seq_num] = frame
    sender_timers[next_seq_num] = time.time() # Start timer for this frame
    print(f"Sender: Sending Frame {frame['seq_num']} with data '{frame['data']}'")
    if random.random() < loss_rate:
        print(f"Sender: Frame {frame['seq_num']} LOST in transmission!")
    elif random.random() < corrupt_rate:
        print(f"Sender: Frame {frame['seq_num']} CORRUPTED in transmission!")
    else:
        print(f"Sender: Frame {frame['seq_num']} successfully sent.")
    next_seq_num += 1
    time.sleep(0.2) # Small delay to simulate sending frames
# Simulate ACKs from receiver
# In a real scenario, ACKs would arrive asynchronously
# Here, we'll just check if receiver processed anything
# Receiver side processing (simplified)
# For each frame the sender *tried* to send, simulate receiver's perspective
for seq_num_to_check in range(send_base, next_seq_num):
```

```
        if seq_num_to_check in buffered_frames:
            frame_to_check = buffered_frames[seq_num_to_check]
            # Simulate reception conditions
            if random.random() >= loss_rate and random.random() >= corrupt_rate: # Frame not lost and not corrupted
                if frame_to_check['seq_num'] == receive_expected_seq_num:
                    print(f"Receiver: Received Frame {frame_to_check['seq_num']} correctly (expected). ACK {receive_expected_seq_num} sent.")
                    received_data_in_order.append(frame_to_check['data'])
                    receive_expected_seq_num += 1
                    # Simulate ACK transmission
                    if random.random() < loss_rate:
                        print(f"Receiver: ACK {frame_to_check['seq_num']} LOST!")
                    elif random.random() < corrupt_rate:
                        print(f"Receiver: ACK {frame_to_check['seq_num']} CORRUPTED!")
                    else:
                        # Sender processes ACK for send_base
                        if frame_to_check['seq_num'] == send_base:
                            print(f"Sender: Received ACK {frame_to_check['seq_num']}. Sliding window.")
                            del sender_timers[send_base]
                            del buffered_frames[send_base]
                            send_base += 1 # Slide window
                        elif frame_to_check['seq_num'] > send_base:
```

```
                    # This is a cumulative ACK, means all frames up to this seq_num are
acknowledged
                    print(f"Sender: Received ACK {frame_to_check['seq_num']} (cumulative).
Sliding window.")
                    for acked_seq in range(send_base, frame_to_check['seq_num'] + 1):
                        if acked_seq in sender_timers: # Only if it was sent by this sender
                            del sender_timers[acked_seq]
                            if acked_seq in buffered_frames: # Also remove from buffer
                                del buffered_frames[acked_seq]
                    send_base = frame_to_check['seq_num'] + 1
                elif frame_to_check['seq_num'] < receive_expected_seq_num:
                    print(f"Receiver: Received Frame {frame_to_check['seq_num']} (duplicate).
Resending ACK for {receive_expected_seq_num - 1}.")
                    # Simulate ACK transmission for the already received frame
                    if random.random() < loss_rate:
                        print(f"Receiver: ACK {receive_expected_seq_num - 1} LOST (duplicate)!")
                    elif random.random() < corrupt_rate:
                        print(f"Receiver: ACK {receive_expected_seq_num - 1} CORRUPTED
(duplicate)!")
                    else:
                        # Simulate sender processing cumulative ACK
                        if (receive_expected_seq_num - 1) == send_base -1:
                            print(f"Sender: Received ACK {receive_expected_seq_num - 1} (duplicate).
Reaffirming window.")
```

```
                    # No window slide, but confirms previous frames
                elif (receive_expected_seq_num - 1) > send_base -1:
                    print(f"Sender: Received ACK {receive_expected_seq_num - 1} (cumulative duplicate). Reaffirming window.")
                    # No window slide
            else:
                print(f"Receiver: Received Frame {frame_to_check['seq_num']} (out-of-order). Discarding and waiting for expected {receive_expected_seq_num}.")
                # In Go-Back-N, receiver discards out-of-order frames
                # Sender will timeout and retransmit from 'send_base'
        # Sender: Check for timeouts (for the oldest unacknowledged frame)
        if send_base in sender_timers and (time.time() - sender_timers[send_base]) > timeout:
            print(f"Sender: Timeout for Frame {send_base}. Retransmitting from Frame {send_base}.")
            next_seq_num = send_base # Go-Back-N: retransmit all frames from send_base
            for i in range(send_base, num_frames):
                if i in sender_timers:
                    sender_timers[i] = time.time() # Reset timer for retransmitted frames
            time.sleep(0.5) # Simulate retransmission delay
        if len(received_data_in_order) == num_frames:
            break
        # Small sleep to prevent busy-waiting in simulation
        time.sleep(0.1)
    print("\n--- Simulation End ---")
```

```
    print(f"Successfully received data: {received_data_in_order}")

# Demonstration

data_to_send_sw = ['P', 'Q', 'R', 'S', 'T', 'U']

simulate_sliding_window(data_to_send_sw, window_size=3, loss_rate=0.3, corrupt_rate=0.1,
timeout=3)
```

3. **Framing: Bit Stuffing and Character Stuffing**

Framing is the process of dividing a stream of bits into distinct blocks or frames. This is essential for multiplexing data from different sources, error control, and flow control. The receiver needs a way to determine the start and end of each frame.

Bit Stuffing

Bit stuffing is used in synchronous data transmission. It ensures that the frame delimiter (a special bit pattern, e.g., 01111110) does not appear in the data itself. Whenever the data contains a sequence of bits that matches the frame delimiter's prefix (e.g., five consecutive 1s for the 01111110 delimiter), an extra 0 bit is "stuffed" into the data. This extra 0 is removed by the receiver.

```
import re

def bit_stuffing(data, flag='01111110'):
    """
    Performs bit stuffing on the data to avoid the flag sequence.
    Args:
        data (str): The binary data string.
        flag (str): The flag sequence (e.g., '01111110').
    Returns:
```

```
        str: The framed data with bit stuffing.
    """
    # The bit stuffing rule for HDLC-like flags (01111110) is to
    # insert a '0' after five consecutive '1's in the data.
    # The pattern to look for is '11111' (five ones).
    # r'\g<1>0' correctly appends a literal '0' after the matched group 1.
    stuffed_data = re.sub(r'(1{5})', r'\g<1>0', data)
    # Add flags at the beginning and end
    return flag + stuffed_data + flag
def bit_unstuffing(stuffed_data, flag='01111110'):
    """
    Performs bit unstuffing on the received data.
    Args:
        stuffed_data (str): The received framed data with bit stuffing.
        flag (str): The flag sequence.
    Returns:
        str: The original data after unstuffing.
    """
    # Remove start and end flags
    if stuffed_data.startswith(flag) and stuffed_data.endswith(flag):
        unstuffed_data = stuffed_data[len(flag):-len(flag)]
    else:
        raise ValueError("Invalid frame: missing start/end flag.")
```

```
    # Remove the stuffed '0's (after five consecutive '1's).
    # The pattern to look for is '111110'.
    # r'\g<1>' replaces '111110' with just '11111' (the captured group).
    unstuffed_data = re.sub(r'(1{5})0', r'\g<1>', unstuffed_data)
    return unstuffed_data
# Demonstration of Bit Stuffing
original_bit_data = "011111111110011111101"
print(f"Original Bit Data: {original_bit_data}")
stuffed_frame = bit_stuffing(original_bit_data)
print(f"Stuffed Frame:     {stuffed_frame}")
unstuffed_data = bit_unstuffing(stuffed_frame)
print(f"Unstuffed Data:    {unstuffed_data}")
print(f"Match original:    {original_bit_data == unstuffed_data}")

# Test with data that contains the flag pattern (but should be stuffed correctly)
original_bit_data_with_flag = "0111111011111110"
print(f"\nOriginal Bit Data (with flag pattern): {original_bit_data_with_flag}")
stuffed_frame_with_flag = bit_stuffing(original_bit_data_with_flag)
print(f"Stuffed Frame:                         {stuffed_frame_with_flag}")
unstuffed_data_with_flag = bit_unstuffing(stuffed_frame_with_flag)
print(f"Unstuffed Data:                        {unstuffed_data_with_flag}")
print(f"Match original:                        {original_bit_data_with_flag == unstuffed_data_with_flag}")
```

4. **Character Stuffing (Byte Stuffing)**

Character stuffing is used in asynchronous data transmission. It involves inserting an ESC (Escape) character whenever the data contains a character that is identical to the frame delimiter or the ESC character itself. The receiver removes the stuffed ESC characters.

#program

```
def character_stuffing(data, flag='DLESTX', esc='DLE'):
    """
    Performs character stuffing on the data.
    Args:
        data (list): A list of characters/bytes (strings).
        flag (str): The start/end flag character.
        esc (str): The escape character.
    Returns:
        list: The framed data with character stuffing.
    """
    stuffed_data = []
    for char in data:
        if char == flag or char == esc:
            stuffed_data.append(esc) # Stuff an ESC character
        stuffed_data.append(char)
    return [flag] + stuffed_data + [flag]
def character_unstuffing(stuffed_frame, flag='DLESTX', esc='DLE'):
    """
```

```
Performs character unstuffing on the received data.

Args:
    stuffed_frame (list): The received framed data with character stuffing.
    flag (str): The start/end flag character.
    esc (str): The escape character.
Returns:
    list: The original data after unstuffing.
"""
# Remove start and end flags
if stuffed_frame[0] == flag and stuffed_frame[-1] == flag:
    unstuffed_data = stuffed_frame[1:-1]
else:
    raise ValueError("Invalid frame: missing start/end flag.")


result = []
i = 0
while i < len(unstuffed_data):
    if unstuffed_data[i] == esc:
        i += 1 # Skip the stuffed ESC character
        if i < len(unstuffed_data):
            result.append(unstuffed_data[i])
        else:
            raise ValueError("Invalid stuffing: ESC at end of data.")
```

```
        else:
            result.append(unstuffed_data[i])
        i += 1
    return result
# Demonstration of Character Stuffing
original_char_data = ['A', 'B', 'DLE', 'C', 'DLESTX', 'D']
print(f"Original Character Data: {original_char_data}")
stuffed_frame_char = character_stuffing(original_char_data)
print(f"Stuffed Frame:           {stuffed_frame_char}")
unstuffed_char_data = character_unstuffing(stuffed_frame_char)
print(f"Unstuffed Data:          {unstuffed_char_data}")
print(f"Match original:          {original_char_data == unstuffed_char_data}")

# Test with data that does not contain the flag or esc characters
original_char_data_clean = ['X', 'Y', 'Z']
print(f"\nOriginal Character Data (clean): {original_char_data_clean}")
stuffed_frame_char_clean = character_stuffing(original_char_data_clean)
print(f"Stuffed Frame:                   {stuffed_frame_char_clean}")
unstuffed_char_data_clean = character_unstuffing(stuffed_frame_char_clean)
print(f"Unstuffed Data:                  {unstuffed_char_data_clean}")
print(f"Match original:                  {original_char_data_clean == unstuffed_char_data_clean}")
```

OUTPUT:

```
--- Stop-and-Wait Protocol Simulation ---

Sender: Sending Frame 0 with data 'A'
Sender: Frame 0 successfully sent.
Receiver: Received Frame 0 correctly.
Receiver: Sending ACK 0
Sender: Received ACK 0. Moving to next frame.

Sender: Sending Frame 1 with data 'B'
Sender: Frame 1 successfully sent.
Sender: Timeout or unexpected ACK. Retransmitting Frame 1.

Sender: Sending Frame 1 with data 'B'
Sender: Frame 1 successfully sent.
Receiver: Received Frame 1 correctly.
Receiver: Sending ACK 1
Sender: Received ACK 1. Moving to next frame.

Sender: Sending Frame 0 with data 'C'
Sender: Frame 0 LOST in transmission!

Sender: Sending Frame 0 with data 'C'
Sender: Frame 0 successfully sent.
Receiver: Received Frame 0 correctly.
Receiver: Sending ACK 0
Sender: Received ACK 0. Moving to next frame.

Sender: Sending Frame 1 with data 'D'
Sender: Frame 1 successfully sent.
Receiver: Received Frame 1 correctly.
Receiver: Sending ACK 1
Sender: Received ACK 1. Moving to next frame.

--- Simulation End ---
Successfully received data: ['A', 'B', 'C', 'D']
```

```
--- Sliding Window Protocol Simulation (Window Size: 3) ---
Sender: Sending Frame 0 with data 'P'
Sender: Frame 0 LOST in transmission!
Sender: Sending Frame 1 with data 'Q'
Sender: Frame 1 LOST in transmission!
Sender: Sending Frame 2 with data 'R'
Sender: Frame 2 successfully sent.
Receiver: Received Frame 1 (out-of-order). Discarding and waiting for expected 0.
Receiver: Received Frame 2 (out-of-order). Discarding and waiting for expected 0.
Receiver: Received Frame 0 correctly (expected). ACK 0 sent.
Sender: Received ACK 0. Sliding window.
Receiver: Received Frame 1 correctly (expected). ACK 1 sent.
Sender: Received ACK 1. Sliding window.
Receiver: Received Frame 2 correctly (expected). ACK 2 sent.
Sender: Received ACK 2. Sliding window.
Sender: Sending Frame 3 with data 'S'
Sender: Frame 3 LOST in transmission!
Sender: Sending Frame 4 with data 'T'
Sender: Frame 4 successfully sent.
Sender: Sending Frame 5 with data 'U'
Sender: Frame 5 successfully sent.
Receiver: Received Frame 3 correctly (expected). ACK 3 sent.
Sender: Received ACK 3. Sliding window.
Receiver: Received Frame 4 correctly (expected). ACK 4 sent.
Sender: Received ACK 4. Sliding window.
Receiver: Received Frame 5 correctly (expected). ACK 5 sent.
Sender: Received ACK 5. Sliding window.

--- Simulation End ---
Successfully received data: ['P', 'Q', 'R', 'S', 'T', 'U']
```

- ```
  Original Bit Data: 01111111110011111101
  Stuffed Frame:     011111100111110111110001111101010111110
  Unstuffed Data:    01111111110011111101
  Match original:    True

  Original Bit Data (with flag pattern): 0111111011111110
  Stuffed Frame:                         01111110011111010111111011001111110
  Unstuffed Data:                        0111111011111110
  Match original:                        True
  ```

  ```

```
Original Character Data: ['A', 'B', 'DLE', 'C', 'DLESTX', 'D']
Stuffed Frame:           ['DLESTX', 'A', 'B', 'DLE', 'DLE', 'C', 'DLE', 'DLESTX', 'D', 'DLESTX']
Unstuffed Data:          ['A', 'B', 'DLE', 'C', 'DLESTX', 'D']
Match original:          True

Original Character Data (clean): ['X', 'Y', 'Z']
Stuffed Frame:                   ['DLESTX', 'X', 'Y', 'Z', 'DLESTX']
Unstuffed Data:                  ['X', 'Y', 'Z']
Match original:                  True
```

RESULT:

Thus, The program successfully simulated Stop-and-Wait and Sliding Window protocols for reliable data transmission. It also implemented Bit Stuffing and Character Stuffing techniques for proper frame creation and data transparency in communication networks.

EX.NO: 6

CONFIGURE ETHERNET LAN AND VLANS IN PACKET TRACER AND CAPTURE MAC LAYER PACKETS USING WIRESHARK

AIM:

To configure an Ethernet LAN and VLANs using Cisco Packet Tracer and to capture and analyze MAC layer packets using Wireshark.

PROCEDURE:

PART 1 : To configure an Ethernet LAN and VLANs using Cisco Packet Tracer

Step A: Configure Ethernet LAN in Cisco Packet Tracer

1. Open Cisco Packet Tracer

- Launch Cisco Packet Tracer.
- Create a new workspace.

2. Add Network Devices

From the device section:

- Add:
    - 1 Switch (e.g., Cisco 2960)
    - 4 PCs

Arrange them properly on the workspace.

Example:

- PC0
- PC1
- PC2

- PC3
- Switch0

3. Connect the Devices

1. Select **Connections**.
2. Choose **Copper Straight-Through Cable**.
3. Connect:
    - PC0 → Switch0
    - PC1 → Switch0
    - PC2 → Switch0
    - PC3 → Switch0

4. Configure IP Addresses

For each PC:

1. Click the PC.
2. Go to:
    - Desktop → IP Configuration
3. Assign IP addresses.

Example:

Device IP Address Subnet Mask

PC0 192.168.1.1 255.255.255.0

PC1 192.168.1.2 255.255.255.0

PC2 192.168.1.3 255.255.255.0

Device IP Address Subnet Mask

PC3 192.168.1.4 255.255.255.0

5. Verify Ethernet LAN Connectivity

1. Open Command Prompt on PC0.
2. Use: ping 192.168.1.4
3. Verify successful replies.

Step B: Configure VLANs in Cisco Packet Tracer

1. **Create VLANs on the Switch**

Open Switch CLI

1. Click Switch0.
2. Select the **CLI** tab.
3. Enter the following commands:

```
enable
configure terminal

vlan 10
name SALES



vlan 20
name HR
```

2. **Assign Ports to VLANs**

Assign FastEthernet Ports

```
interface fastethernet 0/1
switchport mode access
switchport access vlan 10

interface fastethernet 0/2
switchport mode access
switchport access vlan 10

interface fastethernet 0/3
switchport mode access
switchport access vlan 20

interface fastethernet 0/4
switchport mode access
switchport access vlan 20
```

3. **Connect PCs to VLANs**

PC	VLAN
PC0	VLAN 10
PC1	VLAN 10
PC2	VLAN 20
PC3	VLAN 20

4. **Configure IP Addresses for VLANs**

VLAN 10 Devices

- PC0 → 192.168.10.1
- PC1 → 192.168.10.2

VLAN 20 Devices

- PC2 → 192.168.20.1

- PC3 → 192.168.20.2

Subnet Mask:

- 255.255.255.0

5. **Test VLAN Communication**

Same VLAN Communication

From PC0:

ping 192.168.10.2

Expected:

- Successful communication.

Different VLAN Communication

From PC0:

ping 192.168.20.1

Expected:

- Request timed out (without routing).

PART 2 : To capture and analyze MAC layer packets using Wireshark.

Detailed Steps to Capture MAC Layer Packets in Wireshark

Step 1 — Install Wireshark

1. Download Wireshark from:

Wireshark Official Website

2. Run the installer.
3. During installation:
    - Keep default settings

- Install **Npcap** when prompted (required for packet capture)

4. Finish installation and restart the computer if required.

Step 2 — Open Wireshark

1. Launch Wireshark.
2. The home screen displays available network interfaces.

Example interfaces:

- Ethernet
- Wi-Fi
- VMware adapters
- Loopback

Step 3 — Select the Correct Network Interface

Choose the interface currently carrying traffic.

How to identify it

- If using LAN cable → select **Ethernet**
- If using wireless → select **Wi-Fi**

You will notice moving graph lines beside the active interface.

Step 4 — Start Packet Capture

1. Double-click the active interface.
2. Wireshark immediately begins capturing packets.

You will see packets scrolling continuously.

Columns displayed:

- No.
- Time

- Source
- Destination
- Protocol
- Length
- Info

Step 5 — Generate Network Traffic

To capture MAC layer frames, create some traffic.

Method 1 — Ping another device

Open Command Prompt:

Windows

ping 192.168.1.1

Linux

ping 192.168.1.1

Method 2 — Open websites

Open a browser and visit any webpage.

Method 3 — Transfer files

Send files across the network or use shared folders.

Step 6 — Stop Packet Capture

1. Return to Wireshark.
2. Click the **red square Stop button** on the toolbar.

The captured packets remain displayed.

Step 7 — Filter Ethernet Packets

To focus on MAC layer traffic:

In the filter bar type:

eth

Press **Enter**.

This shows Ethernet frames only.

Step 8 — Analyze Ethernet Frame Details

Click any packet.

In the middle pane:

Expand:

Ethernet II

You will see MAC layer information.

Step 9 — Observe Important MAC Layer Fields

1. Destination MAC Address

Example:

Destination: 00:1A:2B:3C:4D:5E

Indicates the receiving device.

2. Source MAC Address

Example:

Source: 00:0C:29:AA:BB:CC

Indicates the sending device.

3. EtherType

Example:

Type: IPv4 (0x0800)

Shows the upper-layer protocol.

Common values:

EtherType	Protocol
0x0800	IPv4
0x0806	ARP
0x86DD	IPv6

Step 10 — Capture ARP Packets (MAC Layer Communication)

In the filter bar:

arp

ARP packets help map IP addresses to MAC addresses.

You can clearly observe:

- Sender MAC address
- Target MAC address

Step 11 — View VLAN Tags (If VLANs Are Configured)

If your network uses VLANs:

1. Capture packets from a trunk link.
2. Open a packet.
3. Expand:

802.1Q Virtual LAN

You can observe:

- VLAN ID
- Priority
- VLAN tagging information

Example:

VLAN ID: 10

Step 12 — Use Useful Filters

Show Ethernet traffic only

eth

Show ARP packets

arp

Show packets from a specific MAC

eth.addr == 00:0C:29:AA:BB:CC

Show broadcast packets

eth.dst == ff:ff:ff:ff:ff:ff

Step 13 — Save the Capture

1. Click:

File → Save As

2. Save as:

.pcapng

This file can be reopened later.

Step 14 — Export Specific Packets (Optional)

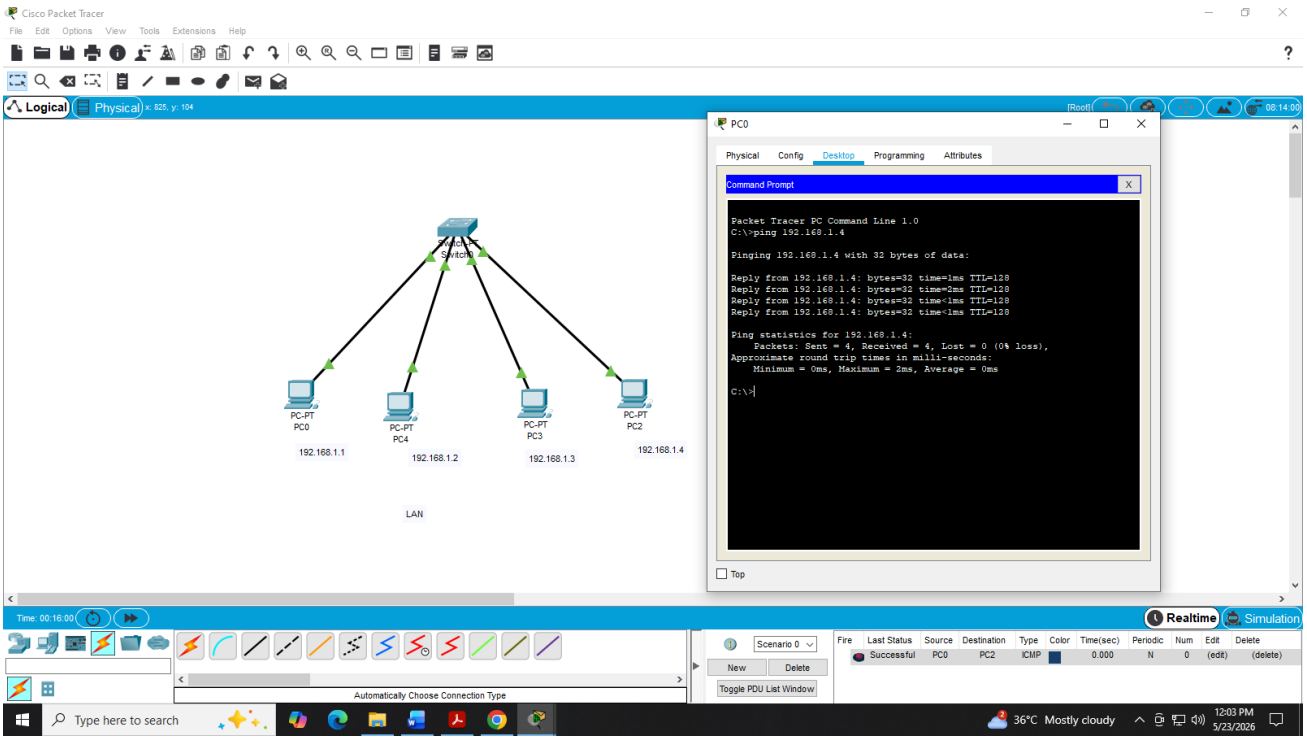
1. Select packets.
2. Go to:

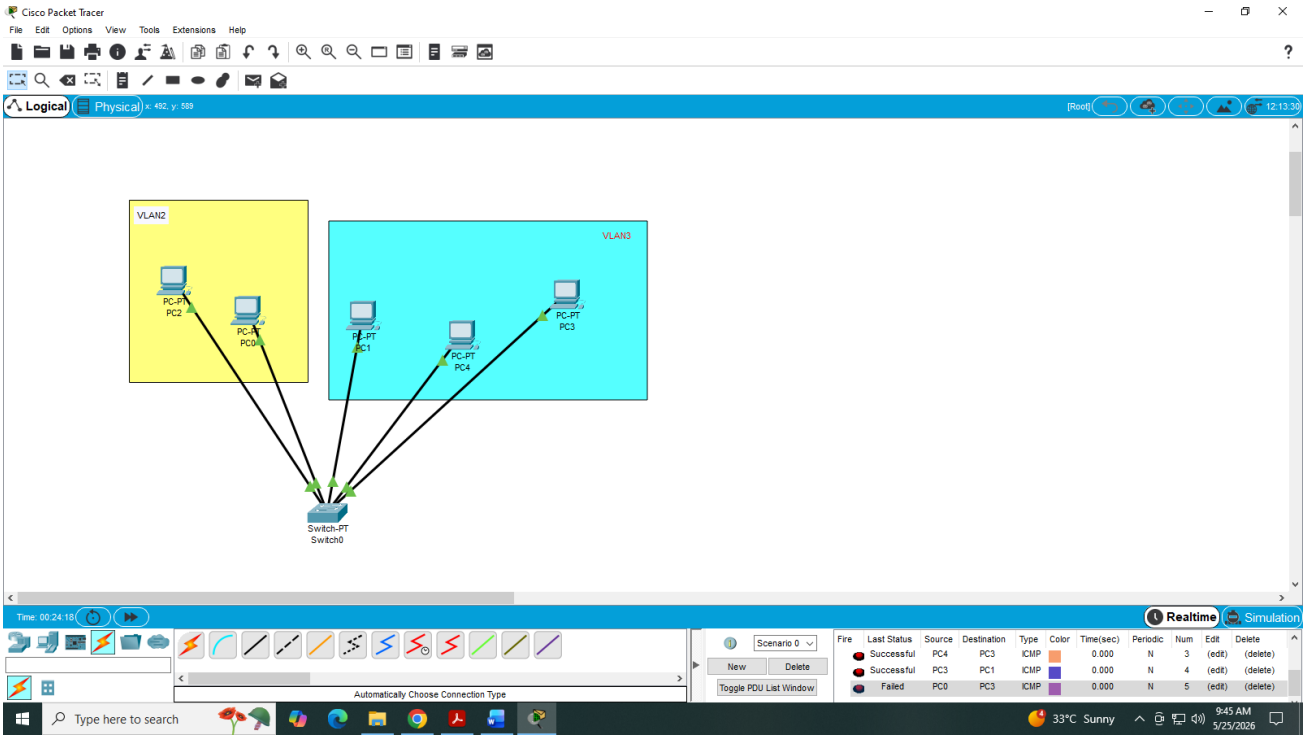
File → Export Specified Packets

Useful for reports and assignments.

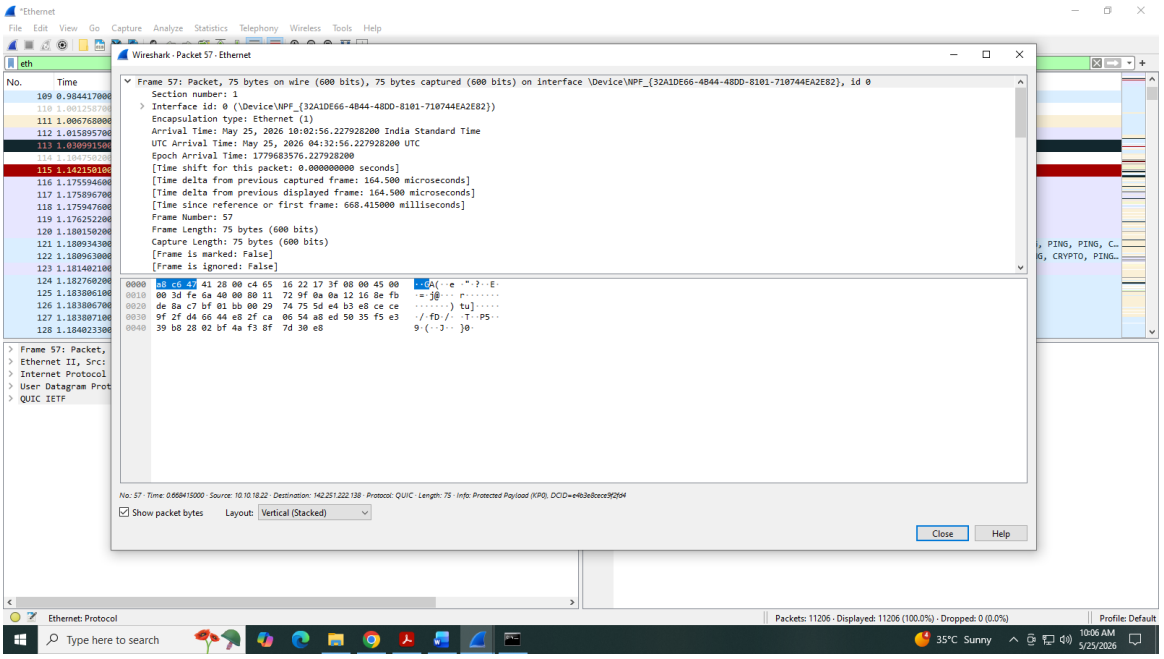
OUTPUT:

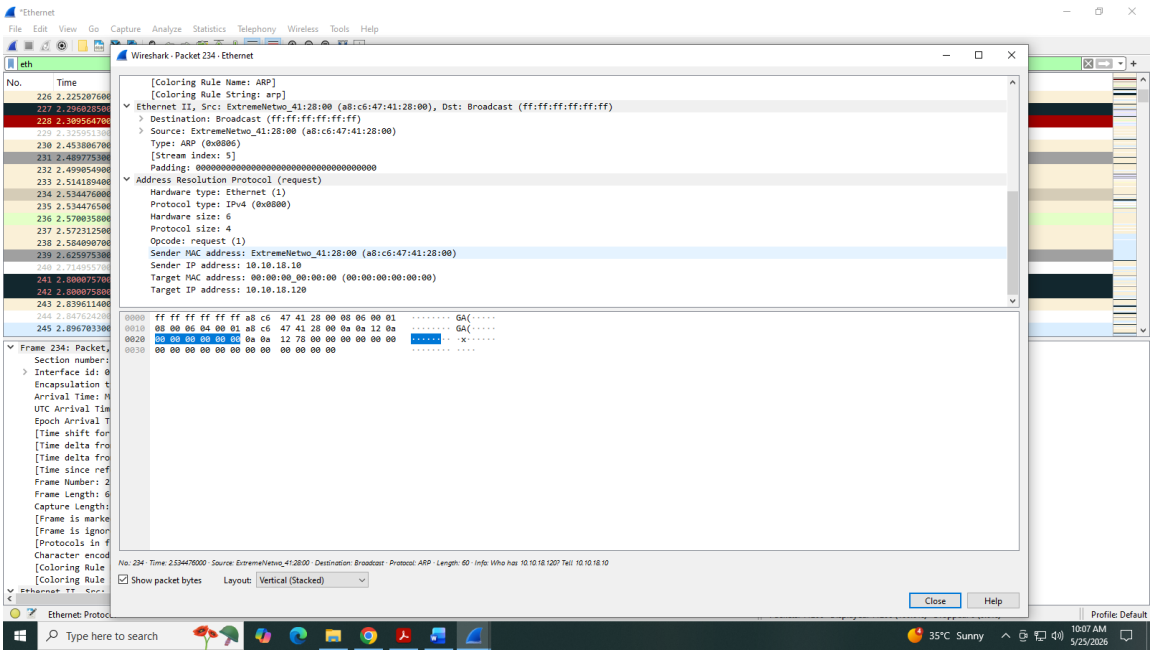
Part 1 : output VLAN





PART 2 : output





RESULT:

Thus, the Ethernet LAN and VLANs were successfully configured in Cisco Packet Tracer. Devices within the same VLAN communicated properly, while inter-VLAN communication required routing. Using Wireshark, MAC layer packets were captured and analyzed, including source/destination MAC addresses, EtherType, and VLAN tags, confirming proper VLAN segmentation and Ethernet communication.

EX.NO: 7

PRACTICE IPV4 AND IPV6 ADDRESSING AND SUBNETTING CONFIGURE STATIC AND DYNAMIC ROUTING (RIP/OSPF) IN PACKET TRACER

AIM:

To practice IPv4 and IPv6 addressing and subnetting, configure static and dynamic routing protocols (RIP and OSPF) using Cisco Packet Tracer.

PROCEDURE:

Step 1 — Open Packet Tracer

1. Launch Cisco Packet Tracer
2. Create a new project.

Step 2 — Build the Network Topology

Add:

- 2 Routers
- 2 Switches
- 4 PCs

Example topology:

```
PC0 --- SW1 --- R1 ----- R2 --- SW2 --- PC2
          PC1 ---/                  \--- PC3
```

Step 3 — Connect Devices

Use:

- Copper Straight-through cables for:
  - PC ↔ Switch
  - Switch ↔ Router
- Serial cable for:
  - Router ↔ Router

Step 4 — IPv4 Addressing Plan

Device	Interface	IP Address	Subnet Mask
PC0	FastEthernet	192.168.1.2	255.255.255.0
PC1	FastEthernet	192.168.1.3	255.255.255.0
R1	G0/0	192.168.1.1	255.255.255.0
R1	S0/0/0	10.0.0.1	255.255.255.252
R2	S0/0/0	10.0.0.2	255.255.255.252
R2	G0/0	192.168.2.1	255.255.255.0
PC2	FastEthernet	192.168.2.2	255.255.255.0
PC3	FastEthernet	192.168.2.3	255.255.255.0

Step 5 — Configure PCs

Example for PC0

1. Click PC0
2. Desktop → IP Configuration
3. Enter:

IP Address: 192.168.1.2

Subnet Mask: 255.255.255.0

Default Gateway: 192.168.1.1

Repeat for all PCs.

Step 6 — Configure Router Interfaces

Router R1

```
enable
configure terminal

interface g0/0
ip address 192.168.1.1 255.255.255.0
no shutdown

interface s0/0/0
ip address 10.0.0.1 255.255.255.252
```

clock rate 64000

no shutdown

Router R2

enable

configure terminal

interface g0/0

ip address 192.168.2.1 255.255.255.0

no shutdown

interface s0/0/0

ip address 10.0.0.2 255.255.255.252

no shutdown

Step 7 — Verify Connectivity

From PC0:

ping 192.168.2.2

Initially, it may fail because routing is not configured.

PART 2 — Configure Static Routing

Step 8 — Configure Static Routes

On R1

ip route 192.168.2.0 255.255.255.0 10.0.0.2

On R2

ip route 192.168.1.0 255.255.255.0 10.0.0.1

Step 9 — Test Static Routing

From PC0:

ping 192.168.2.2

Result:

- Ping successful

PART 3 — Configure Dynamic Routing (RIP)

Step 10 — Remove Static Routes (Optional)

no ip route 192.168.2.0 255.255.255.0 10.0.0.2

Step 11 — Configure RIP on R1

router rip

version 2

network 192.168.1.0

network 10.0.0.0

no auto-summary

Step 12 — Configure RIP on R2

router rip

version 2

network 192.168.2.0

network 10.0.0.0

no auto-summary

Step 13 — Verify RIP

Use:

show ip route

Routes marked with:

R

indicate RIP routes.

PART 4 — Configure Dynamic Routing (OSPF)

Step 14 — Configure OSPF on R1

router ospf 1

network 192.168.1.0 0.0.0.255 area 0

network 10.0.0.0 0.0.0.3 area 0

Step 15 — Configure OSPF on R2

router ospf 1

network 192.168.2.0 0.0.0.255 area 0

network 10.0.0.0 0.0.0.3 area 0

Step 16 — Verify OSPF

show ip route

Routes marked:

O

represent OSPF routes.

PART 5 — Practice IPv6 Addressing

Step 17 — Enable IPv6 Routing

On routers:

ipv6 unicast-routing

Step 18 — Configure IPv6 Addresses

R1

interface g0/0

ipv6 address 2001:DB8:1::1/64

no shutdown

R2

interface g0/0

ipv6 address 2001:DB8:2::1/64

no shutdown

Step 19 — Configure PCs with IPv6

Example:

Device	**IPv6 Address**
PC0	2001:DB8:1::2/64
PC2	2001:DB8:2::2/64

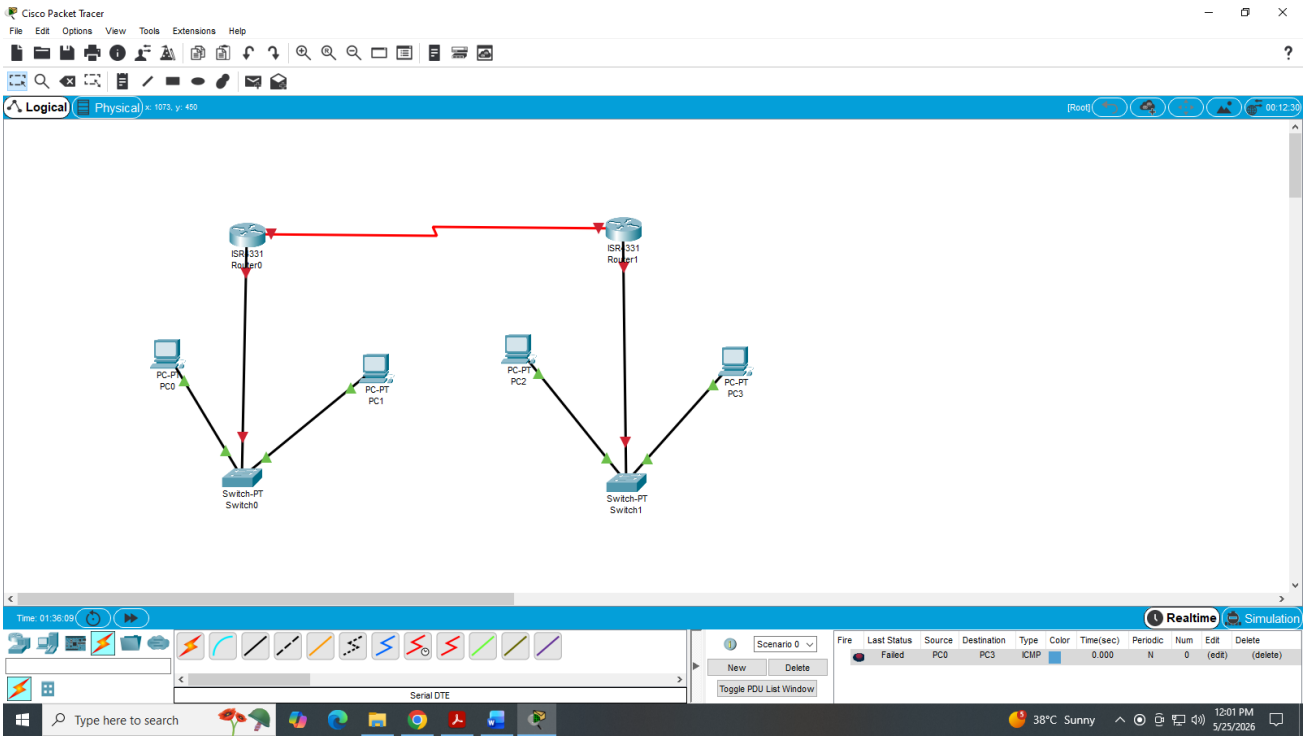
Default gateways:

- 2001:DB8:1::1
- 2001:DB8:2::1

Step 20 — Test IPv6 Connectivity

ping 2001:DB8:2::2

OUTPUT(IC)



RESULT:

Thus, IPv4 and IPv6 addressing and subnetting were configured successfully in Cisco Packet Tracer. Static routing, RIP, and OSPF dynamic routing protocols were implemented and verified successfully.

EX.NO: 8

ANALYZE ICMP AND ARP PACKETS USING WIRESHARK CONFIGURE DHCP AND NAT IN PACKET TRACER

AIM:

To analyze ICMP and ARP packets using Wireshark and to configure DHCP and NAT using Cisco Packet Tracer.

PROCEDURE:

Part A: To analyze ICMP and ARP packets using Wireshark

Step 1 — Install and Open Wireshark

1. Install Wireshark from:

Wireshark Download Page

2. Open Wireshark.
3. Select the active network interface:
    - Ethernet
    - Wi-Fi
4. Double-click the interface to start packet capture.

Step 2 — Generate ICMP Traffic

Open Command Prompt.

Run:

ping 8.8.8.8

This generates ICMP packets.

Step 3 — Capture ICMP Packets

In Wireshark filter bar:

icmp

Press Enter.

Step 4 — Analyze ICMP Packet Fields

Click an ICMP packet.

Expand:

- Ethernet II
- Internet Protocol
- Internet Control Message Protocol

Observe:

- Source IP address
- Destination IP address
- ICMP Type
- Sequence number
- TTL

Common ICMP Types

Type Meaning

8 Echo Request

0 Echo Reply

Step 5 — Generate ARP Traffic

Open Command Prompt.

Run:

arp -d *

Then ping another device:

ping 192.168.1.1

This forces ARP packet generation.

Step 6 — Capture ARP Packets

In Wireshark filter:

arp

Step 7 — Analyze ARP Packet Fields

Click ARP packet.

Observe:

- Sender MAC address
- Sender IP address
- Target MAC address

- Target IP address

ARP Packet Types

Operation	**Meaning**
Request	Asking for MAC address
Reply	Returning MAC address

Part B: to configure DHCP and NAT using Cisco Packet Tracer.

Step 8 — Create Network Topology

Add:

- 1 Router
- 1 Switch
- 2 PCs

Topology:

```
PC0 ----\
        Switch ---- Router
PC1 ----/
```

Step 9 — Configure Router Interface

Click Router → CLI.

```
enable
configure terminal

interface g0/0
ip address 192.168.1.1 255.255.255.0
no shutdown
```

Step 10 — Configure DHCP Pool

```
ip dhcp pool LANPOOL
network 192.168.1.0 255.255.255.0
default-router 192.168.1.1
dns-server 8.8.8.8
```

Step 11 — Exclude Router Address

```
ip dhcp excluded-address 192.168.1.1
```

Step 12 — Configure PCs for DHCP

For each PC:

1. Desktop
2. IP Configuration
3. Select:

DHCP

PCs automatically receive IP addresses.

Step 13 — Verify DHCP

On PC command prompt:

```
ipconfig
```

You should see:

- Assigned IP address
- Subnet mask
- Default gateway

Configure NAT in Packet Tracer

Step 14 — Build NAT Topology

Add:

- 2 Routers
- 1 Switch
- 2 PCs

Example:

LAN ---- R1 ---- R2 ---- INTERNET

Step 15 — Configure IP Addresses

LAN Interface

```
interface g0/0
ip address 192.168.1.1 255.255.255.0
ip nat inside
no shutdown
```

WAN Interface

```
interface s0/0/0
ip address 200.1.1.1 255.255.255.252
ip nat outside
no shutdown
```

Step 16 — Configure Access List

```
access-list 1 permit 192.168.1.0 0.0.0.255
```

Step 17 — Configure NAT Overload (PAT)

```
ip nat inside source list 1 interface s0/0/0 overload
```

Step 18 — Configure Default Route

```
ip route 0.0.0.0 0.0.0.0 200.1.1.2
```

Step 19 — Verify NAT

Use:

show ip nat translations

You can observe:

- Inside local addresses
- Inside global addresses

Step 20 — Test Connectivity

From PC:

ping 8.8.8.8

Successful reply indicates proper NAT operation.

Step 21 — Capture DHCP, ARP, and ICMP in Wireshark

Apply filters:

DHCP

bootp

ICMP

icmp

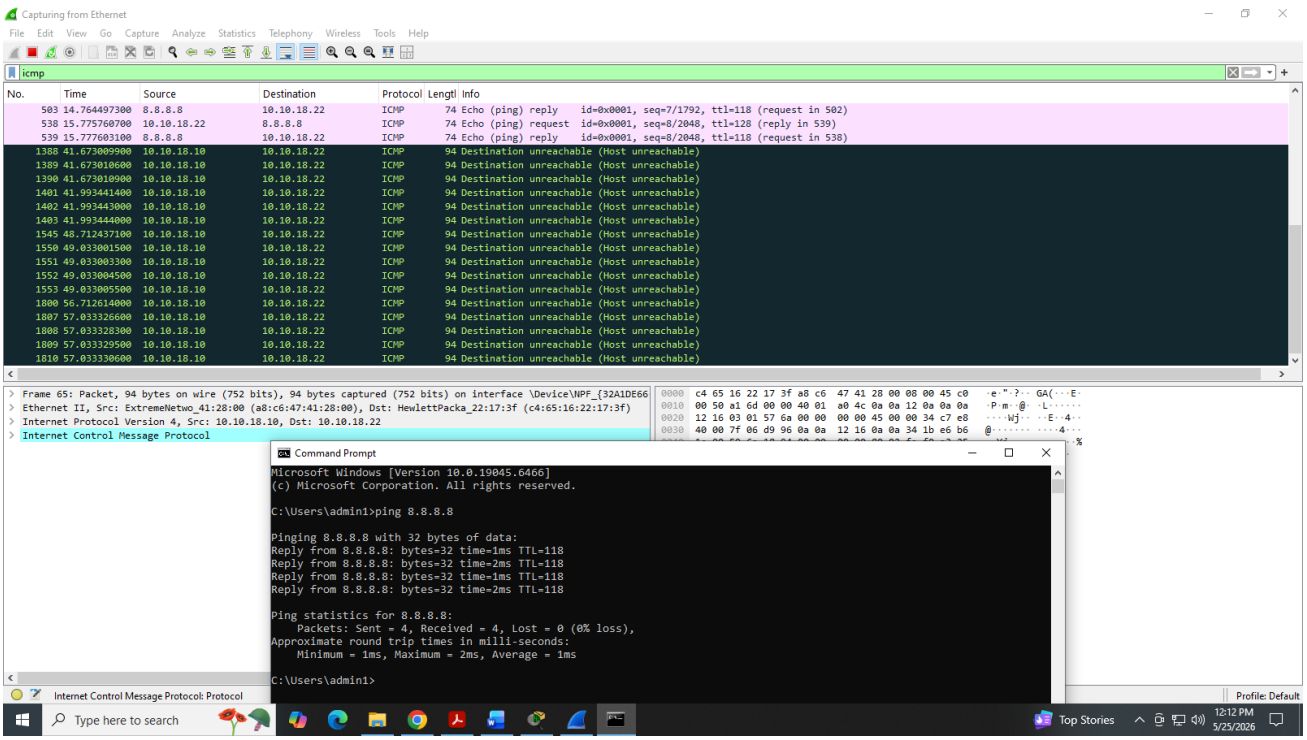
ARP

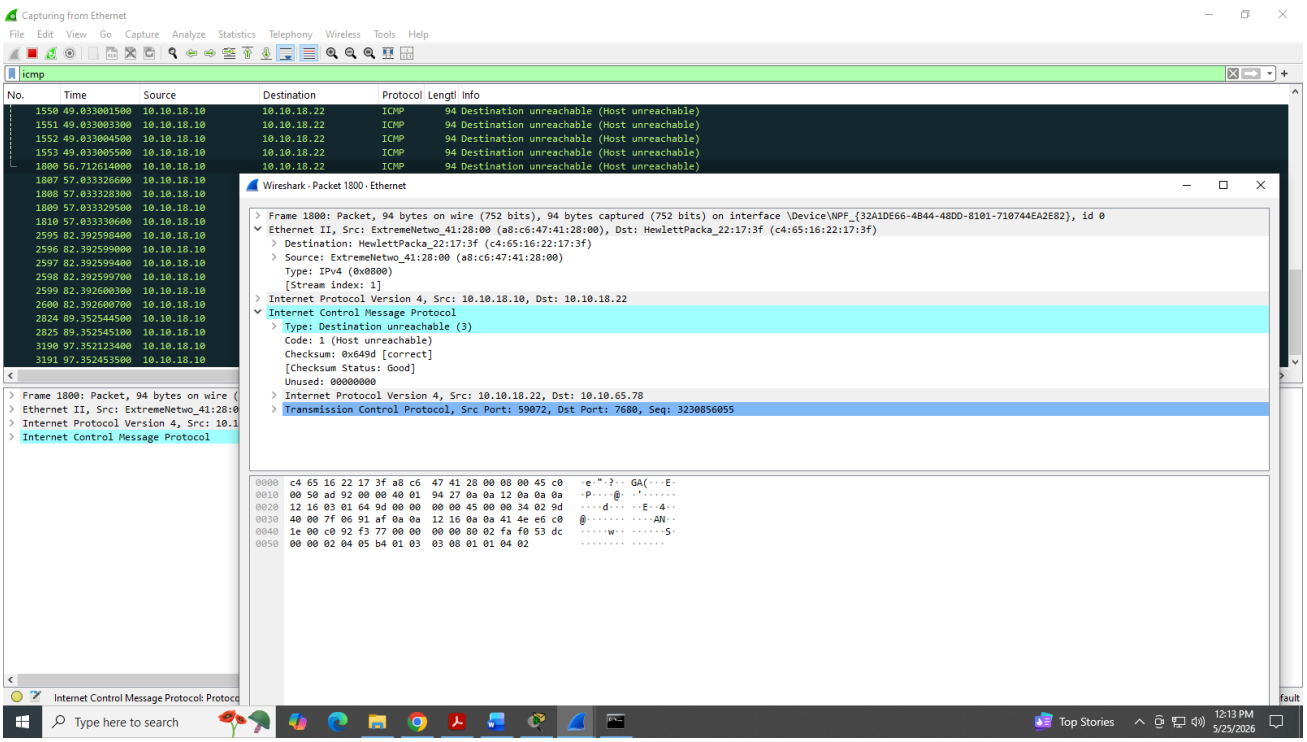
arp

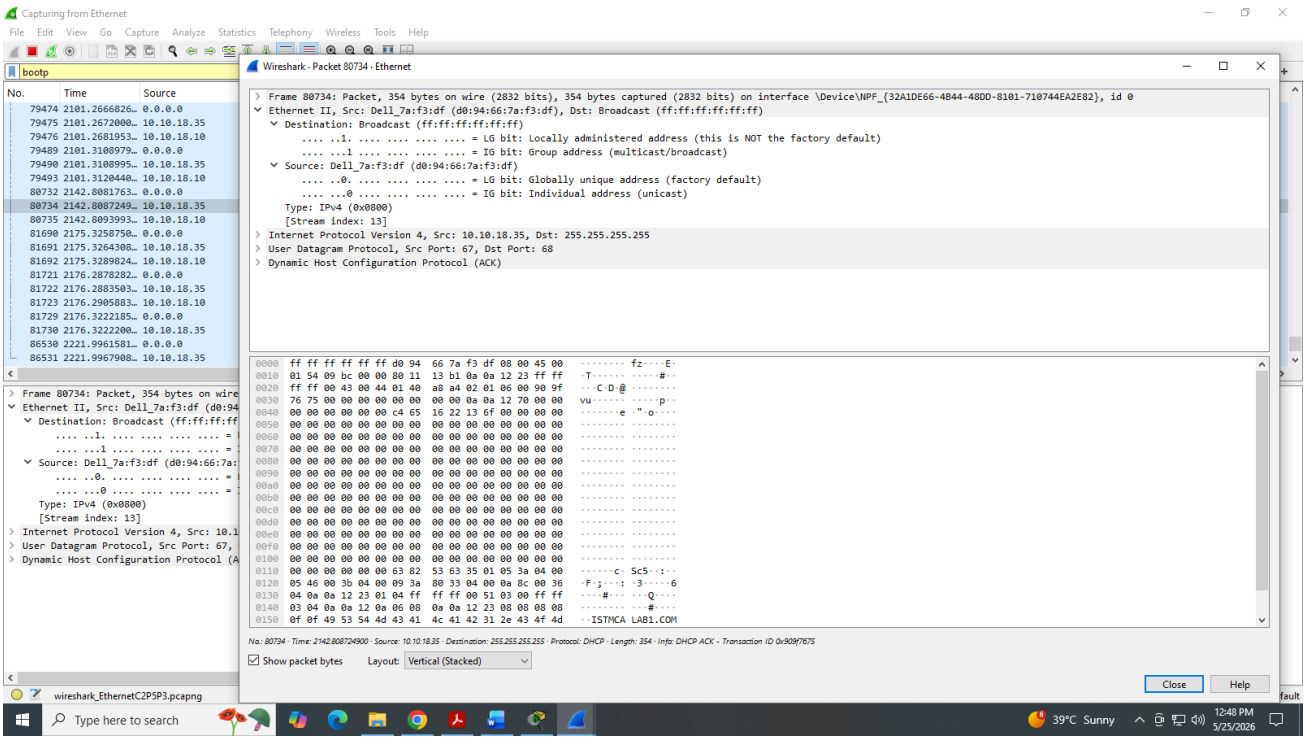
Observe packet exchange and protocol behavior.

OUTPUT:

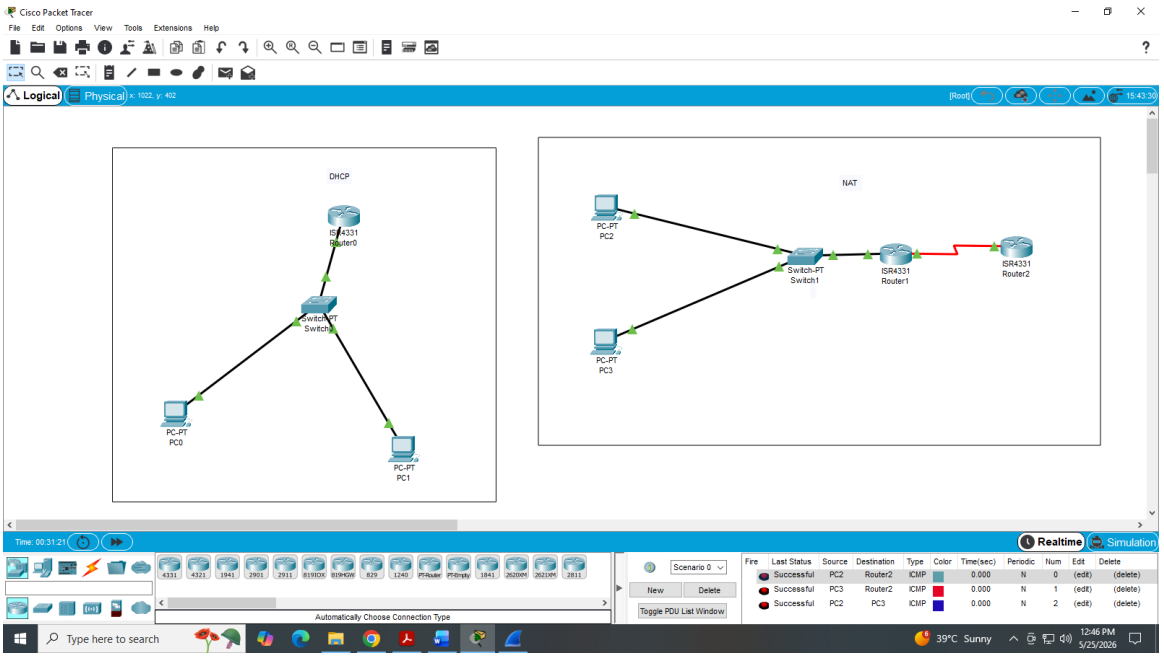
Part A: To analyze ICMP and ARP packets using Wireshark







Part B: to configure DHCP and NAT using Cisco Packet Tracer.



RESULT:

ICMP and ARP packets were successfully captured and analyzed using Wireshark. DHCP was configured successfully in Cisco Packet Tracer and NAT configuration was also implemented successfully.

EX.NO: 9

COMPARE TCP AND UDP CONNECTIONS USING WIRESHARK SIMULATE PORT COMMUNICATION AND CONGESTION CONTROL USING SOCKET PROGRAMMING

AIM:

To study and compare TCP and UDP connections using Wireshark, and to simulate port-based communication and congestion control behavior using socket programming.

PROCEDURE:

PART A: TCP CLIENT–SERVER PROGRAM

1. TCP Server (Python Example)

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('127.0.0.1', 5000))
server.listen(5)

print("TCP Server waiting for connection...")

conn, addr = server.accept()
print("Connected to:", addr)

data = conn.recv(1024).decode()
print("Received:", data)

conn.send("Message received via TCP".encode())
conn.close()
```

2. TCP Client

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('127.0.0.1', 5000))

client.send("Hello Server (TCP)".encode())
response = client.recv(1024).decode()
```

```
print("Server:", response)
client.close()
```

Wireshark Steps (TCP)

1. Open Wireshark.
2. Select interface (Wi-Fi / Ethernet / Loopback).
3. Start capture.
4. Run TCP server and client.
5. Apply filter:

tcp.port == 5000

Observations (TCP)

- 3-way handshake observed:
  - SYN
  - SYN-ACK
  - ACK
- Data packets are acknowledged.
- Retransmissions visible if delay introduced.
- Ordered delivery ensured.

PART B: UDP CLIENT–SERVER PROGRAM

1. UDP Server (python)

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(('127.0.0.1', 6000))
```

```
print("UDP Server ready...")

data, addr = server.recvfrom(1024)
print("Received:", data.decode())

server.sendto("Message received via UDP".encode(), addr)
```

2. UDP Client

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

client.sendto("Hello Server (UDP)".encode(), ('127.0.0.1', 6000))

data, addr = client.recvfrom(1024)
print("Server:", data.decode())
```

Wireshark Steps (UDP)

1. Start Wireshark capture.
2. Run UDP server and client.
3. Apply filter:

```
udp.port == 6000
```

Observations (UDP)

- No handshake observed.
- Packets sent directly.
- No acknowledgment packets.
- Faster transmission compared to TCP.
- Packet loss may occur under simulated delay/load.

PART C: CONGESTION CONTROL SIMULATION (TCP)

Concept

Congestion control in TCP is observed indirectly by:

- Delayed packet acknowledgment
- Retransmission
- Window size adaptation

1.SERVER PROGRAM (TCP - Congestion Simulation)

```
import socket
import time

# Create TCP socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('127.0.0.1', 5001))
server.listen(1)

print("TCP Server ready and waiting...")

conn, addr = server.accept()
print("Connected by:", addr)

total_received = 0

while True:
    data = conn.recv(4096)

    if not data:
        break

    total_received += len(data)
    print(f"Received {len(data)} bytes | Total: {total_received}")

    # Artificial delay to simulate congestion processing
    time.sleep(0.5)

conn.close()
server.close()

print("Connection closed")
```

2.CLIENT PROGRAM (TCP - Congestion Simulation)

```
import socket
import time

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('127.0.0.1', 5001))

# Large data packet to simulate congestion
message = "A" * 1000000  # 1 MB data

chunk_size = 4096
index = 0

print("Sending data...")

while index < len(message):
    chunk = message[index:index + chunk_size]
    client.send(chunk.encode())

    print(f"Sent {len(chunk)} bytes")

    index += chunk_size

    # Delay to simulate slow network / congestion scenario
    time.sleep(0.2)

client.close()
print("Data transmission complete")
```

Wireshark Filter

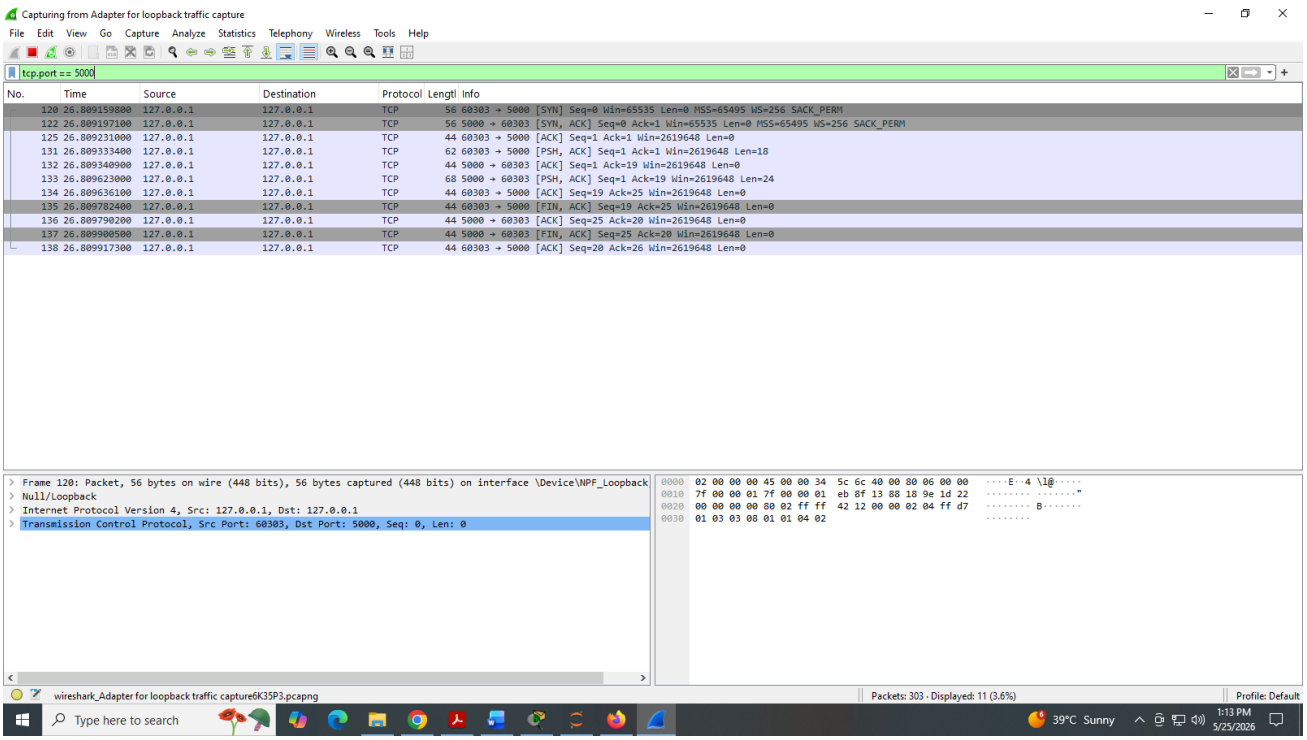
```
tcp.port == 5001
```

Observations

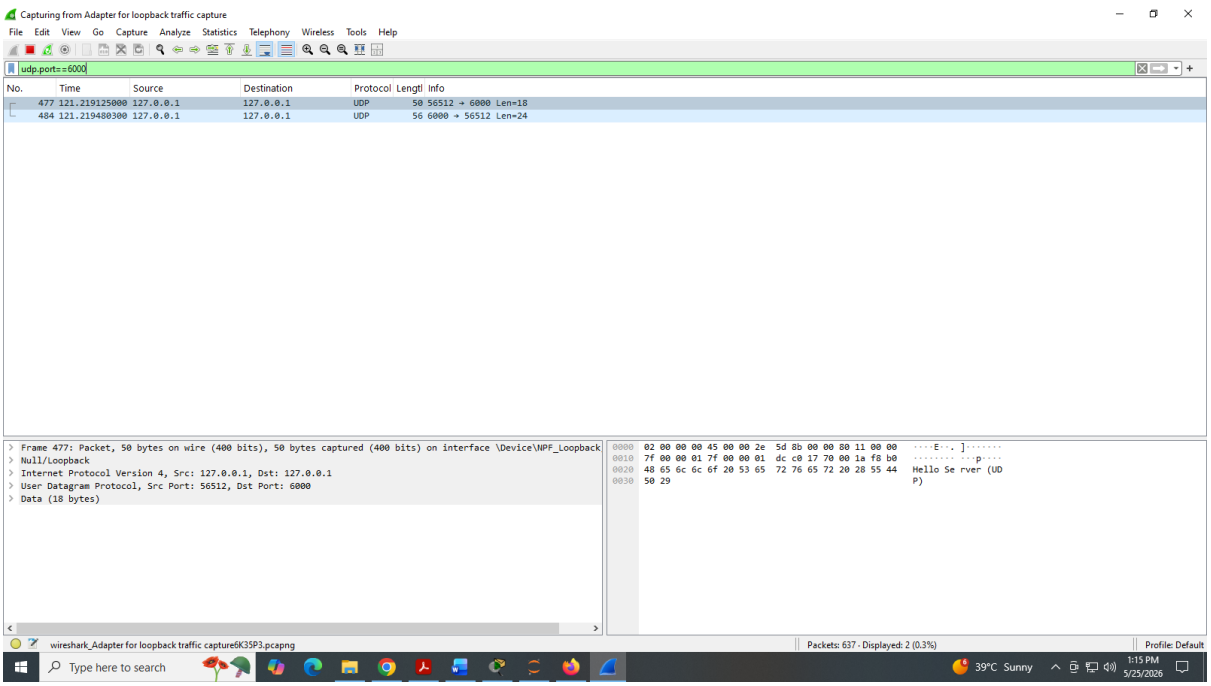
- TCP slows transmission when delays occur.
- Retransmission packets appear.
- ACK timing increases under load.
- Flow becomes controlled instead of continuous.

OUTPUT:

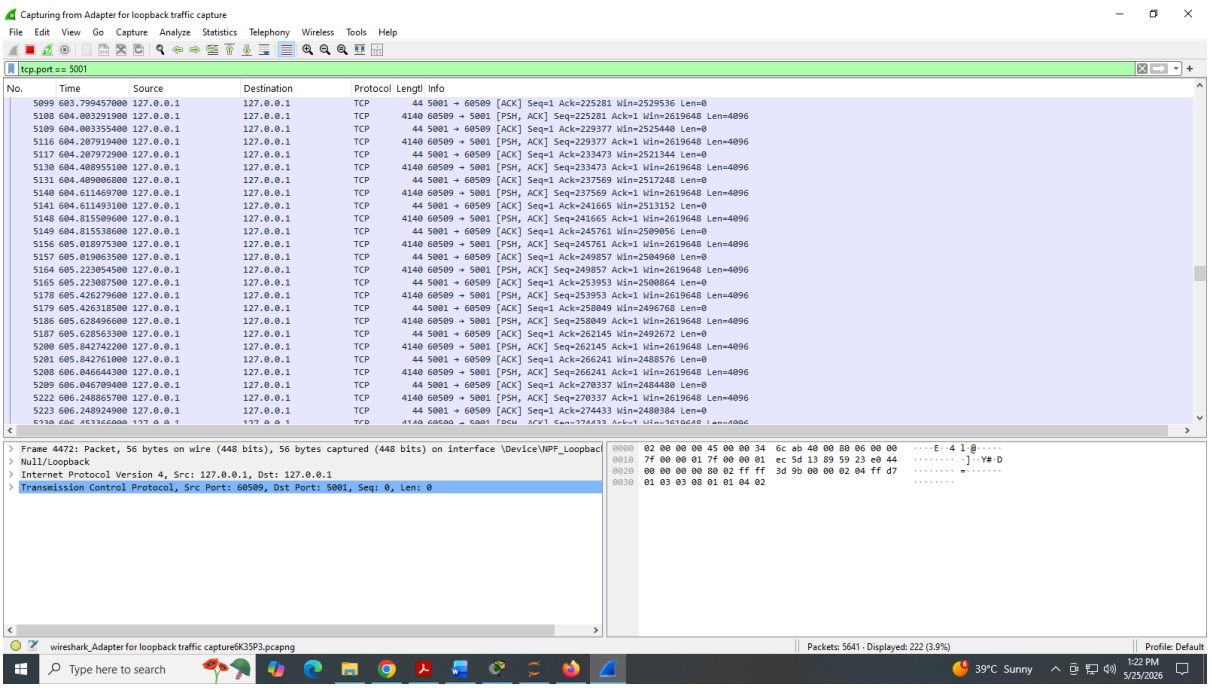
PART A: TCP CLIENT–SERVER PROGRAM



PART B: UDP CLIENT–SERVER PROGRAM



PART C: CONGESTION CONTROL SIMULATION (TCP)



RESULT:

Thus ,the experiment successfully demonstrated TCP and UDP communication using socket programming, Packet analysis using Wireshark. , Differences in reliability, speed, and structure ,TCP congestion control behavior under network load condition .

EX.NO: 10

PERFORM DNS QUERIES USING NSLOOKUP AND DIG TEST HTTP/HTTPS REQUESTS USING POSTMAN AND BROWSER DEV TOOLS

AIM:

To test **DNS queries and HTTP/HTTPS requests**, in order to understand client-server communication, name resolution, and web request/response behavior.

PROCEDURE:

PART A: DNS QUERIES USING nslookup AND dig

1. Using nslookup

Open terminal / command prompt and run:

nslookup google.com

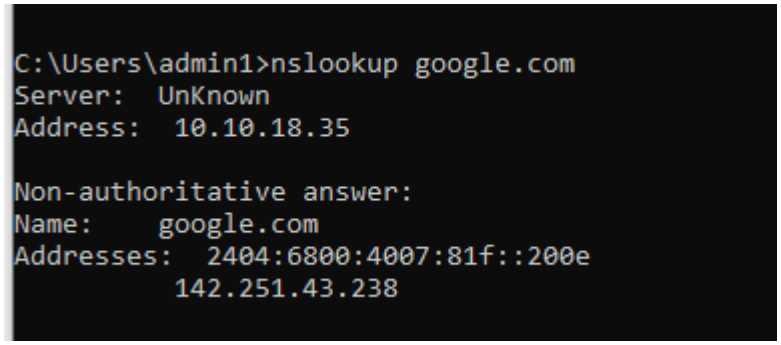
2. Using dig command

dig google.com

To get only IP address

dig +short google.com

OUTPUT:



PROCEDURE:

PART B: HTTP/HTTPS TESTING USING BROWSER DEVTOOLS

Open Web Browser

- Launch **Google Chrome** or **Mozilla Firefox** on your system.

Open Developer Tools

- Press **F12** key on the keyboard
  OR
- Right-click on the webpage → select **Inspect**

Navigate to Network Tab

- In Developer Tools, click on the **Network** tab.
- This tab records all network activity between the browser and server.

Enable Network Logging

- Ensure the recording button (● red dot) is **ON**.
- If it is off, click it to start capturing network traffic.

Clear Previous Logs (Optional but recommended)

- Click the **Clear (🚫 or 🧹 icon)** to remove previous requests.
- This helps in observing fresh requests clearly.

Visit a Website

- In the browser address bar, enter a website such as:

  ```
  https://example.com
  ```

- Press Enter.

Observe Network Activity

- As the page loads, multiple requests will appear in the Network tab.
- These include HTML, CSS, JavaScript, images, and API calls.

Filter Requests (Optional)

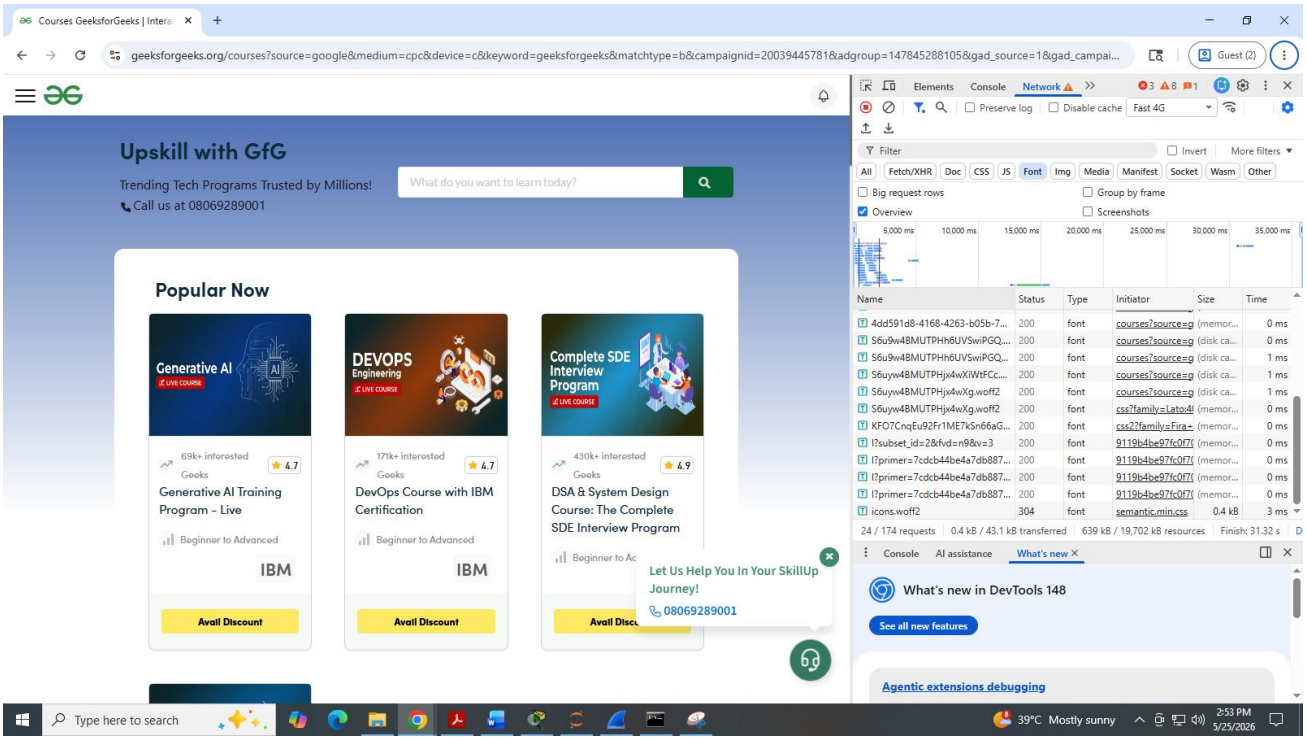
- Use filters like:

- **Fetch/XHR** → for API calls
- **Doc** → for main HTML page
- **JS / CSS / Img** → for static resources

Inspect Individual Request

- Click on any request to view details:
  - Headers
  - Response
  - Timing
  - Payload (if applicable)

OUTPUT:



RESULT:

Thus, The experiment successfully demonstrated DNS resolution using nslookup and real-time network inspection using browser developer tools

EX.NO: 11

SET UP FTP AND EMAIL (SMTP, POP3) SERVERS USING FILEZILLA AND THUNDERBIRD

AIM:

To set up and configure FTP and Email servers using FileZilla Server and Mozilla Thunderbird, and to demonstrate Telnet and SSH remote sessions for secure and non-secure communication with remote devices.

PROCEDURE:

PART A: FTP SERVER CONFIGURATION USING FILEZILLA

PROCEDURE

1. Install FileZilla Server

1. Download and install FileZilla Server.
2. Launch FileZilla Server Interface.

2. Configure FTP Server

1. Click **Edit → Users**.
2. Click **Add User**.
3. Enter username:

student

4. Set password for authentication.
5. Enable password option.

3. Configure Shared Folder

1. Go to **Shared Folders** section.
2. Click **Add Folder**.
3. Select folder to share.
4. Assign permissions:
    - Read
    - Write
    - Delete
    - Append

4. Start FTP Service

1. Ensure server is running on:

Port 21

2. Verify server status is active.

5. Connect using FileZilla Client

1. Open FileZilla Client.
2. Enter:
    - Host: 127.0.0.1
    - Username: student
    - Password: configured password
    - Port: 21

3. Click **Quickconnect**.

6. Transfer Files

1. Drag files from local panel to server panel.
2. Observe upload/download progress.

OBSERVATIONS

- FTP connection established successfully.
- Authentication verified using username/password.
- Files transferred between client and server.
- Upload and download speeds displayed.

RESULT

FTP server was successfully configured using FileZilla Server, and files were transferred using FileZilla Client.

PART B: EMAIL CONFIGURATION USING THUNDERBIRD (SMTP & POP3)

AIM

To configure email services using SMTP and POP3 protocols in Thunderbird.

PROCEDURE

1. Install Thunderbird

1. Download and install Mozilla Thunderbird.

2. Open Thunderbird application.

2. Create Mail Account

1. Click:

Account Settings → Add Mail Account

2. Enter:
    - Name
    - Email address
    - Password

3. Configure SMTP Server

1. Set outgoing mail server:
    - Server Type: SMTP
    - Port: 587 or 25
    - Security: STARTTLS/SSL

4. Configure POP3 Server

1. Set incoming mail server:
    - Protocol: POP3
    - Port: 110
    - Server address
    - Username/password

5. Send Test Email

1. Click **Compose**.
2. Enter recipient email.
3. Type subject and message.
4. Click **Send**.

6. Receive Email

1. Click **Get Messages**.
2. Observe incoming messages downloaded from POP3 server.

OBSERVATIONS

- SMTP successfully sent emails.
- POP3 retrieved emails from server.
- Inbox synchronized with mail server.
- Email headers and attachments handled properly.

RESULT

SMTP and POP3 services were successfully configured and tested using Thunderbird email client.

PART C: TELNET SESSION DEMONSTRATION

AIM

To establish a Telnet session with a remote device.

PROCEDURE

1. Enable Telnet Client (Windows)

1. Open:

Control Panel → Programs → Turn Windows features on/off

2. Enable:

Telnet Client

2. Open Command Prompt

Run:

telnet 127.0.0.1 23

3. Login to Remote Device

1. Enter username/password if prompted.
2. Execute commands remotely.

OBSERVATIONS

- Telnet session established successfully.
- Communication occurs in plain text.
- Remote commands executed successfully.

RESULT

Remote access was established using Telnet protocol.

PART D: SSH SESSION DEMONSTRATION

AIM

To establish a secure SSH session with a remote device.

PROCEDURE

1. Open PuTTY / Terminal

1. Launch PuTTY or terminal.

2. Configure SSH Connection

Enter:

- Host/IP Address:

127.0.0.1

- Port:

22

- Protocol:

SSH

3. Connect to Remote Device

1. Click **Open**.
2. Accept SSH key if prompted.
3. Enter username and password.

4. Execute Remote Commands

Example:

ls
pwd
ipconfig

OBSERVATIONS

- SSH connection encrypted successfully.
- Secure authentication performed.
- Remote shell access obtained.
- Commands executed securely.

OUTPUT:

RESULT:

The experiment successfully demonstrated FTP server setup and file transfer using FileZilla , Email communication using SMTP and POP3 in Thunderbird , Remote login using Telnet , Secure remote access using SSH . The experiment helped understand file transfer, email communication, and remote administration protocols in networking.

EX.NO:12

CREATE SIMPLE WEB SERVICE CLIENT USING REST API USING PYTHON AND USE PING, TRACEROUTE, NETSTAT TO ANALYZE PERFORMANCE

AIM:

To create a simple web service client using REST API in Python, and to analyze network connectivity and performance using Ping, Traceroute, and Netstat commands.

PROCEDURE:

PART A: SIMPLE WEB SERVICE CLIENT USING REST API

pip install requests

```
import requests
# REST API URL
url = "https://jsonplaceholder.typicode.com/posts/1"
# Send GET request
response = requests.get(url)
# Display status code
print("Status Code:", response.status_code)
# Display JSON response
print("Response Data:")
print(response.json())
```

OUTPUT:

```
Status Code: 200
```

Response Data:

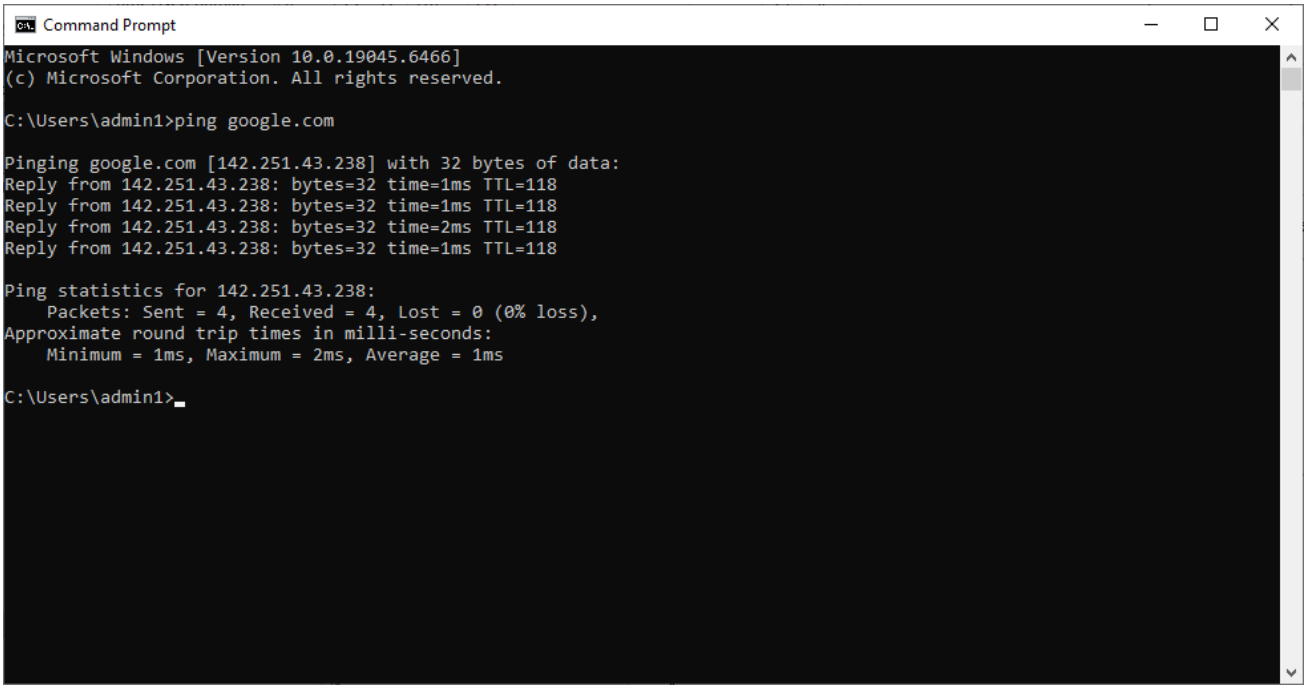
{'userId': 1, 'id': 1, 'title': 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit', 'body': 'quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto'}

PART B: NETWORK PERFORMANCE ANALYSIS USING PING

1. Open Command Prompt / Terminal

Run: ping google.com

OUTPUT:

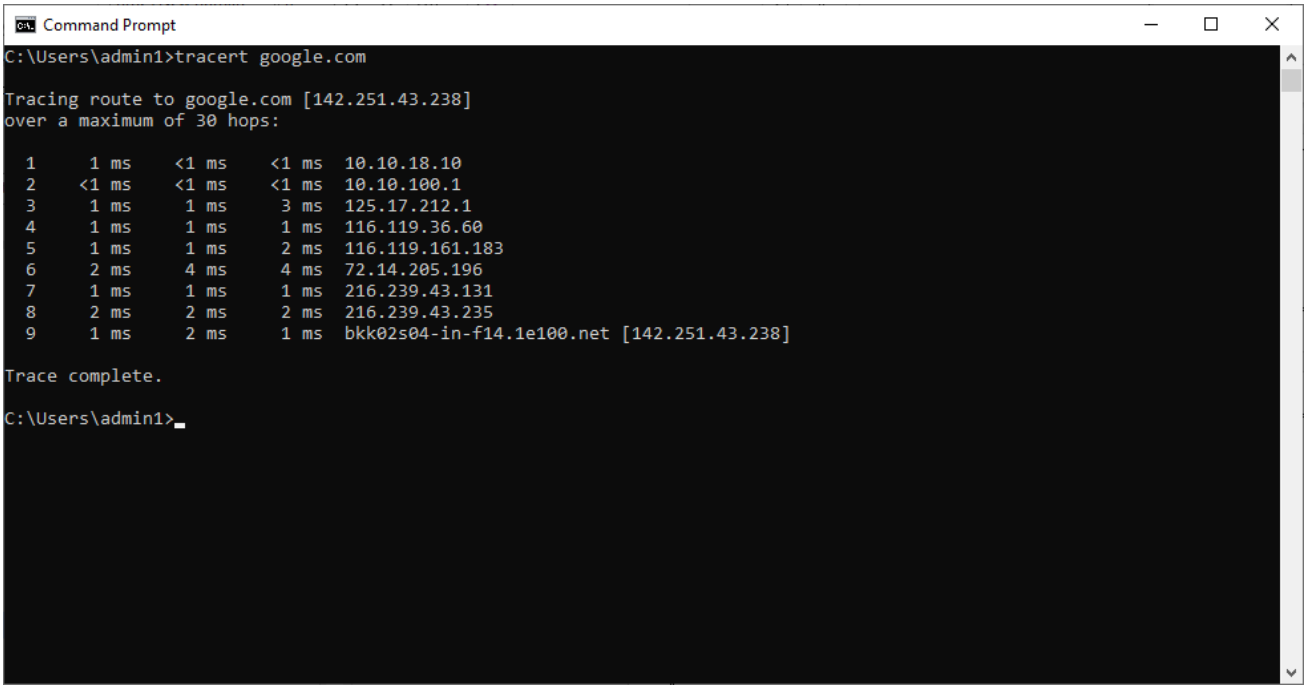


PART C: NETWORK PATH ANALYSIS USING TRACEROUTE

1. Open Command Prompt / Terminal

RUN: tracert google.com

OUTPUT:



PART D: NETWORK CONNECTION ANALYSIS USING NETSTAT

1. Open Terminal / Command Prompt

Run: netstat

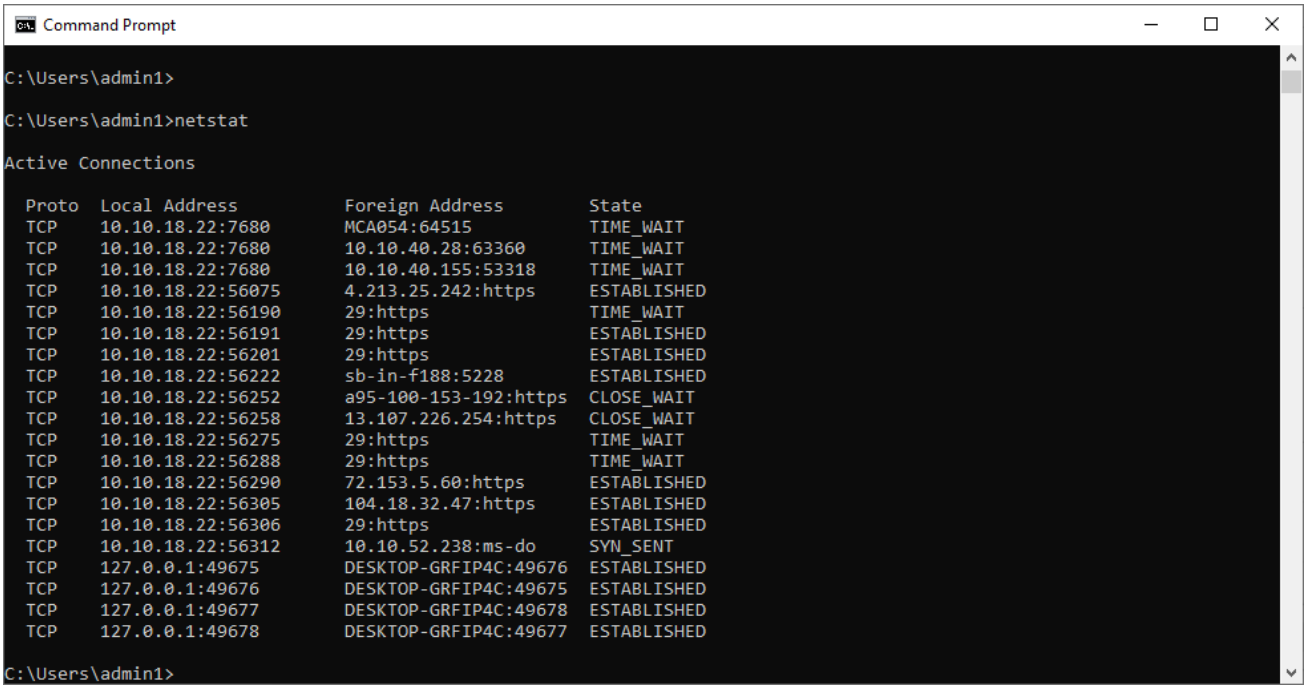
2. Display All Connections and Ports

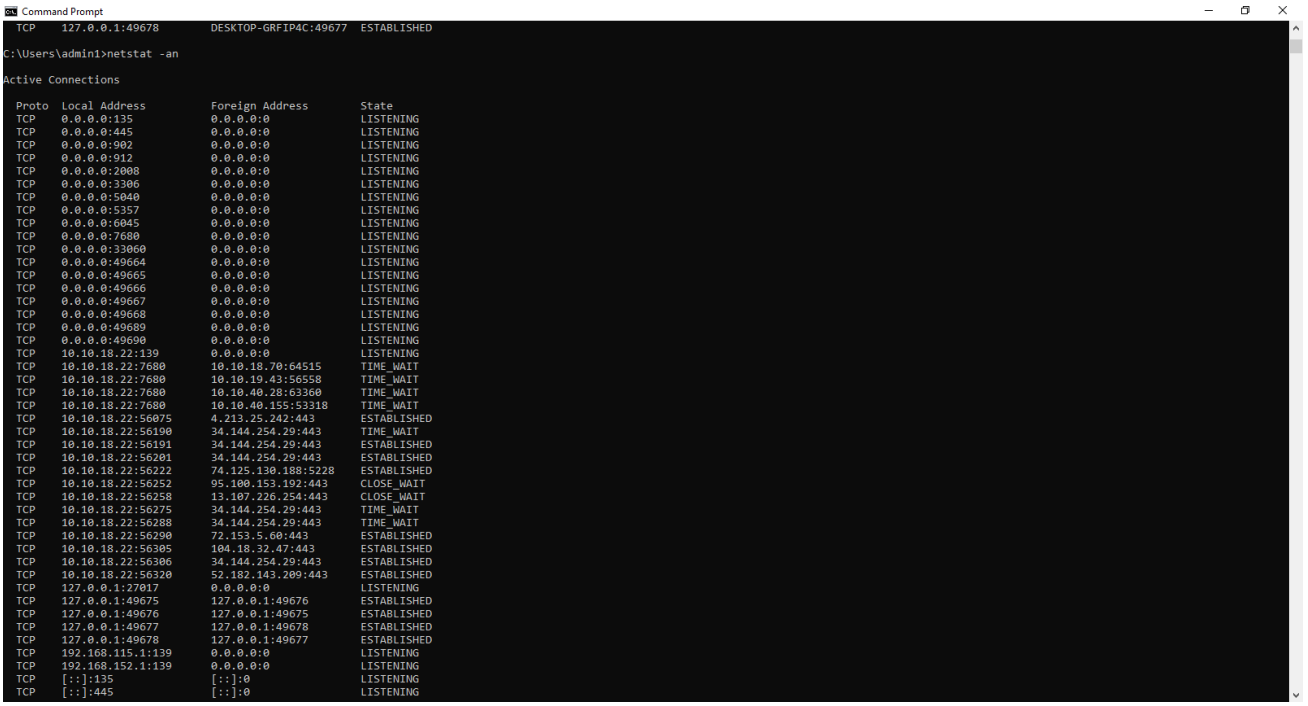
Run: netstat -an

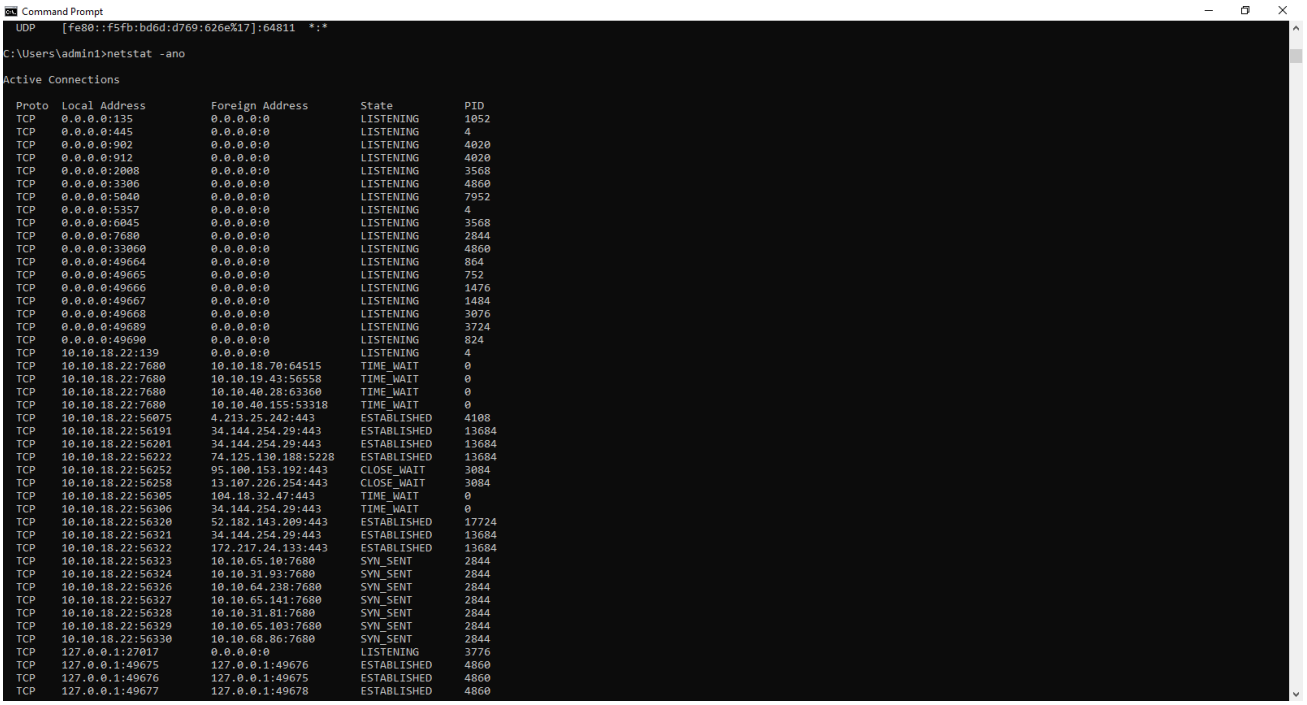
3. Display Process IDs

Run: netstat -ano

OUTPUT:


Command Prompt
C:\Users\admin1>
C:\Users\admin1>netstat
Active Connections
Proto Local Address Foreign Address State
TCP 10.10.18.22:7680 MCA054:64515 TIME_WAIT
TCP 10.10.18.22:7680 10.10.40.28:63360 TIME_WAIT
TCP 10.10.18.22:7680 10.10.40.155:53318 TIME_WAIT
TCP 10.10.18.22:56075 4.213.25.242:https ESTABLISHED
TCP 10.10.18.22:56190 29:https TIME_WAIT
TCP 10.10.18.22:56191 29:https ESTABLISHED
TCP 10.10.18.22:56201 29:https ESTABLISHED
TCP 10.10.18.22:56222 sb-in-f188:5228 ESTABLISHED
TCP 10.10.18.22:56252 a95-100-153-192:https CLOSE_WAIT
TCP 10.10.18.22:56258 13.107.226.254:https CLOSE_WAIT
TCP 10.10.18.22:56275 29:https TIME_WAIT
TCP 10.10.18.22:56288 29:https TIME_WAIT
TCP 10.10.18.22:56290 72.153.5.60:https ESTABLISHED
TCP 10.10.18.22:56305 104.18.32.47:https ESTABLISHED
TCP 10.10.18.22:56306 29:https ESTABLISHED
TCP 10.10.18.22:56312 10.10.52.238:ms-do SYN_SENT
TCP 127.0.0.1:49675 DESKTOP-GRFIP4C:49676 ESTABLISHED
TCP 127.0.0.1:49676 DESKTOP-GRFIP4C:49675 ESTABLISHED
TCP 127.0.0.1:49677 DESKTOP-GRFIP4C:49678 ESTABLISHED
TCP 127.0.0.1:49678 DESKTOP-GRFIP4C:49677 ESTABLISHED
C:\Users\admin1>


Command Prompt
TCP 127.0.0.1:49678 DESKTOP-GRFIP4C:49677 ESTABLISHED
C:\Users\admin1>netstat -an
Active Connections
Proto Local Address Foreign Address State
TCP 0.0.0.0:135 0.0.0.0:0 LISTENING
TCP 0.0.0.0:445 0.0.0.0:0 LISTENING
TCP 0.0.0.0:902 0.0.0.0:0 LISTENING
TCP 0.0.0.0:912 0.0.0.0:0 LISTENING
TCP 0.0.0.0:2008 0.0.0.0:0 LISTENING
TCP 0.0.0.0:3306 0.0.0.0:0 LISTENING
TCP 0.0.0.0:5040 0.0.0.0:0 LISTENING
TCP 0.0.0.0:5357 0.0.0.0:0 LISTENING
TCP 0.0.0.0:6045 0.0.0.0:0 LISTENING
TCP 0.0.0.0:7680 0.0.0.0:0 LISTENING
TCP 0.0.0.0:33060 0.0.0.0:0 LISTENING
TCP 0.0.0.0:49664 0.0.0.0:0 LISTENING
TCP 0.0.0.0:49665 0.0.0.0:0 LISTENING
TCP 0.0.0.0:49666 0.0.0.0:0 LISTENING
TCP 0.0.0.0:49667 0.0.0.0:0 LISTENING
TCP 0.0.0.0:49668 0.0.0.0:0 LISTENING
TCP 0.0.0.0:49689 0.0.0.0:0 LISTENING
TCP 0.0.0.0:49690 0.0.0.0:0 LISTENING
TCP 10.10.18.22:139 0.0.0.0:0 LISTENING
TCP 10.10.18.22:7680 10.10.18.70:64515 TIME_WAIT
TCP 10.10.18.22:7680 10.10.19.43:56558 TIME_WAIT
TCP 10.10.18.22:7680 10.10.40.28:63360 TIME_WAIT
TCP 10.10.18.22:7680 10.10.40.155:53318 TIME_WAIT
TCP 10.10.18.22:56075 4.213.25.242:443 ESTABLISHED
TCP 10.10.18.22:56190 34.144.254.29:443 TIME_WAIT
TCP 10.10.18.22:56191 34.144.254.29:443 ESTABLISHED
TCP 10.10.18.22:56201 34.144.254.29:443 ESTABLISHED
TCP 10.10.18.22:56222 74.125.130.188:5228 ESTABLISHED
TCP 10.10.18.22:56252 95.100.153.192:443 CLOSE_WAIT
TCP 10.10.18.22:56258 13.107.226.254:443 CLOSE_WAIT
TCP 10.10.18.22:56275 34.144.254.29:443 TIME_WAIT
TCP 10.10.18.22:56288 34.144.254.29:443 TIME_WAIT
TCP 10.10.18.22:56290 72.153.5.60:443 ESTABLISHED
TCP 10.10.18.22:56305 104.18.32.47:443 ESTABLISHED
TCP 10.10.18.22:56306 34.144.254.29:443 ESTABLISHED
TCP 10.10.18.22:56320 52.182.143.209:443 ESTABLISHED
TCP 127.0.0.1:27017 0.0.0.0:0 LISTENING
TCP 127.0.0.1:49675 127.0.0.1:49676 ESTABLISHED
TCP 127.0.0.1:49676 127.0.0.1:49675 ESTABLISHED
TCP 127.0.0.1:49677 127.0.0.1:49678 ESTABLISHED
TCP 127.0.0.1:49678 127.0.0.1:49677 ESTABLISHED
TCP 192.168.115.1:139 0.0.0.0:0 LISTENING
TCP 192.168.152.1:139 0.0.0.0:0 LISTENING
TCP [::]:135 [::]:0 LISTENING
TCP [::]:445 [::]:0 LISTENING



RESULT:

The experiment was successfully conducted to analyze network communication and services using various networking tools and protocols.

EX.NO:13

SIMULATE SYMMETRIC/ASYMMETRIC ENCRYPTION ALGORITHMS USING PYTHON

AIM:

To simulate and study symmetric and asymmetric encryption algorithms using Python, and to understand secure data encryption and decryption techniques used in network security.

PROGRAM:

PROGRAM 1: ASYMMETRIC ENCRYPTION SIMULATION

 # Symmetric Encryption using XOR

```
def encrypt(text, key):
    encrypted = ""
    for char in text:
        encrypted += chr(ord(char) ^ key)
    return encrypted
def decrypt(cipher, key):
    decrypted = ""
    for char in cipher:
        decrypted += chr(ord(char) ^ key)
    return decrypted
# Original message
message = "NETWORK SECURITY"
# Secret key
key = 7
# Encryption
```

```
encrypted_text = encrypt(message, key)
# Decryption
decrypted_text = decrypt(encrypted_text, key)
print("Original Message :", message)
print("Encrypted Message:", encrypted_text)
print("Decrypted Message:", decrypted_text)
```

OUTPUT:

Original Message : NETWORK SECURITY

Encrypted Message: IBSPHUL'TBDRUNS^

Decrypted Message: NETWORK SECURITY

PROGRAM 2 : ASYMMETRIC ENCRYPTION SIMULATION

```
pip install cryptography
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes
# Generate private key
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)
# Generate public key
public_key = private_key.public_key()
# Original message
message = b"HELLO NETWORK SECURITY"
# Encrypt using public key
```

```
encrypted = public_key.encrypt(
    message,
    padding.PKCS1v15()
)
# Decrypt using private key
decrypted = private_key.decrypt(
    encrypted,
    padding.PKCS1v15()
)
print("Original Message :", message.decode())
print("Encrypted Message:", encrypted)
print("Decrypted Message:", decrypted.decode())
```

OUTPUT:

Original Message : HELLO NETWORK SECURITY

Encrypted Message: b'W7q\tt\xce\xe3\x01K\x17\x98\xf5t\xb6l\x1fT\xe9i}\x90X-N\x88\x08S\xec\xbaX\x06\xd2\xf4\xe5\x1c\x0cr\x0c4+c5\xb1c\xbc\x9a\xfcs\x13\xb5\x13g\xfd<\xe9\xd4eN\x877z\x8dxwM\xc6\x13QO\r_\x03\x9b\xc2\xb5\xada\x04\xf5>\xfd&\xa9\xc9\xc6b\xcf\x97*d\x8d\x9a\x029D\xa8!u&\xd8\xadh\xeaY\xab\x84\xdd`J\xf7\x906\xa9\x9a\x8e\xb7\xc1"\x13\xd9\xa5\xa8\xc9\x1cA\x83\x82W\xca\xbaJ\xf3?\xea\xa9%\x98\xd4\x90\xba<8$\x08\x84\x91\x80x\xfb>\x12-z)\xd5\x987&*\xe3\x8f>\xe9\x93\x0c\xff\x0f\xbem\xcb_\xe5Q_\x18\xe9\xa7\x81\x19\xdd\x8e\xcd\xa2?\xdcB\x1dU\xc3\xb8\x1bu\xc6&\x8d\x9c\xd3\x18\xb8@\xe5\xf2\x8c\x15\x92J\x9e\xe4\x9de\xfc\xbcI\xeb\xef\xa4\xb0~s*\xbd\x83*\xbd@u\x17\xa1\x1c\xe1\xb4\xc3\xec\xe9e\x04\x86S\xac\x17\x8a\x9a\x08U\xbdL\x88s\x11\xff\xfeN\x9f\xa4\xba\x7f'

Decrypted Message: HELLO NETWORK SECURITY

RESULT:

Symmetric and asymmetric encryption algorithms was successfully simulated using Python XOR encryption logic.

EX.NO:14

CAPTURE WPA2 AUTHENTICATION TRAFFIC USING WIRESHARK

AIM:

To capture and analyze WPA2 authentication traffic using Wireshark in order to study the wireless authentication process, including the WPA2 four-way handshake and encrypted wireless communication between client and access point.

PROCEDURE:

1. Install and Open Wireshark

1. Install Wireshark on the system.
2. Launch Wireshark application.

2. Select Wireless Interface

1. Choose the Wi-Fi adapter from the interface list.
2. Ensure monitor mode/promiscuous mode is enabled if supported.

3. Start Packet Capture

1. Click **Start Capturing Packets**.
2. Keep Wireshark running.

4. Reconnect to WPA2 Wi-Fi Network

1. Disconnect the device from Wi-Fi.

2. Reconnect to the WPA2-secured network.
3. Enter Wi-Fi password if prompted.

5. Capture Authentication Traffic

During reconnection, Wireshark captures:

- Beacon frames
- Probe requests/responses
- Authentication frames
- Association requests/responses
- WPA2 four-way handshake packets

6. Apply Wireshark Filter

Use the following filter:

eapol

OR

wlan.fc.type_subtype == 0x08

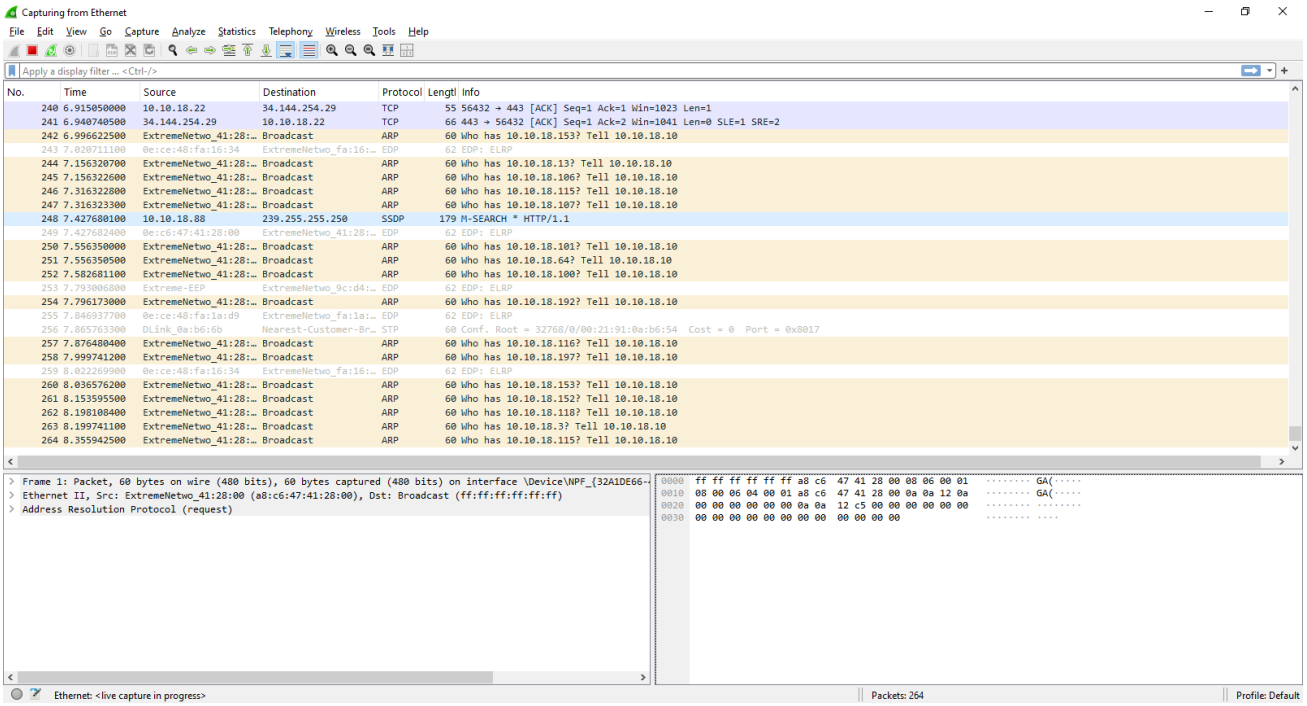
7. Analyze WPA2 Handshake

Observe:

- Message 1
- Message 2
- Message 3
- Message 4

These packets represent the WPA2 four-way handshake process.

OUTPUT:



RESULT:

WPA2 authentication traffic was successfully captured and analyzed using Wireshark. The experiment demonstrated the WPA2 four-way handshake process, wireless authentication mechanisms, and encrypted communication between the wireless client and access point.

.

EX.NO:15

AUTOMATE ROUTER CONFIG USING NETMIKO USING PYTHON

AIM:

To automate router configuration using the Netmiko Python library by establishing an SSH connection to a network device and executing configuration commands remotely

PROGRAM:

```
pip install netmiko

from netmiko import ConnectHandler

# Router connection details

router = {

    "device_type": "cisco_ios",

    "host": "192.168.1.1",

    "username": "admin",

    "password": "admin",

    "secret": "admin"

}


# Establish SSH connection

connection = ConnectHandler(**router)

# Enter enable mode

connection.enable()
```

```
print("Connected to Router Successfully")

# Configuration commands

config_commands = [

    "interface loopback 0",

    "ip address 10.10.10.1 255.255.255.0",

    "no shutdown"

]

# Send configuration commands

output = connection.send_config_set(config_commands)

print("\nConfiguration Output:\n")

print(output)

# Verify configuration

verify = connection.send_command("show ip interface brief")

print("\nVerification Output:\n")

print(verify)

# Disconnect session

connection.disconnect()

print("\nConnection Closed")
```

OUTPUT:

```
config term

Enter configuration commands, one per line.
```

Router(config)#interface loopback0

Router(config-if)#ip address 10.10.10.1 255.255.255.255

Router(config-if)#description Configured_by_Netmiko

Router(config-if)#no shutdown

Router(config-if)#end

RESULT:

Thus the program executed successfully and automate router configuration is established.