EXP:01	Study of various Network commands

Aim

To study and practice various network commands used in Linux and Windows operating systems for network configuration, troubleshooting, and analysis.

Algorithm / Procedure

1. **Identify** the operating system (Linux or Windows) to be used for the experiment.
2. **Execute** basic configuration commands (`ipconfig`/`ip address show`, `hostname`) to view the system's network identity.
3. **Test** connectivity using the `ping` command to a local host, an IP address, and a fully qualified domain name (FQDN).
4. **Analyze** network statistics and active connections using the `netstat` command.
5. **Perform** DNS lookups using the `nslookup` command to resolve FQDNs to IP addresses.
6. **Trace** the route to a remote host using `tracert` (Windows) or `traceroute`/`mtr` (Linux) to identify network path and latency.
7. **Examine** the ARP cache using `arp -a`.
8. **Document** the output of each command.

Code:

Command	Description
ifconfig	Displays and configures network interfaces (use ip addr in modern systems).
ip addr show	Shows IP address and interface details.
ping <host>	Checks connectivity with another host.
netstat -an	Displays network connections, routing tables, and interface statistics.
ss -tuln	Shows listening ports and socket statistics (modern replacement for netstat).
traceroute <host>	Displays the route packets take to reach a host.
nslookup <domain>	Queries DNS to obtain domain name or IP address.
dig <domain>	Performs detailed DNS lookups (advanced alternative to nslookup).
route -n	Displays or modifies the IP routing table.
arp -a	Displays the ARP cache (IP-to-MAC address mapping).
iwconfig	Displays wireless network interface details.
curl <URL>	Transfers data from or to a server using HTTP, FTP, etc.
wget <URL>	Downloads files from the web via HTTP, HTTPS, or FTP.
hostname -I	Shows the system's IP address.
ethtool eth0	Displays or changes Ethernet device settings.

Windows Network Commands

Command	Description
ipconfig	Displays IP configuration of all network interfaces.
ipconfig /all	Shows detailed network configuration (MAC, DNS, DHCP, etc.).
ping <host>	Tests connectivity to a specific IP or hostname.
tracert <host>	Traces the route taken by packets to reach a host.
netstat -ano	Displays active connections, ports, and associated process IDs.
nslookup <domain>	Resolves domain names to IP addresses and vice versa.
arp -a	Displays ARP cache entries.
route print	Displays the system's routing table.
getmac	Displays the MAC addresses of network interfaces.
hostname	Displays the system's hostname.
netsh interface show interface	Displays the list of network interfaces and their status.
telnet <host> <port>	Tests connectivity to a specific port (if Telnet is enabled).
net view	Displays shared resources on a network.
pathping <host>	Combines ping and tracert to identify packet loss and latency.

Command	Description
nbtstat -n	Displays NetBIOS name table of the local machine.

Result

The various network commands in Linux and Windows were successfully studied and executed, demonstrating their use in network diagnostics and configuration.

EXP:02	Study of Network cables

Aim

To understand the different types of network cables and to prepare a cross-wired cable and a straight-through cable using a crimping tool.

Algorithm / Procedure

1. **Identify** the different types of network cables (e.g., Coaxial, Twisted Pair, Fiber Optic) and their applications.
2. **Understand** the T568A and T568B wiring standards for twisted-pair cables.
3. **Prepare** a straight-through cable by terminating both ends with the same standard (e.g., T568B to T568B).
4. **Prepare** a cross-wired cable by terminating one end with T568A and the other with T568B.
5. **Test** the prepared cables using a cable tester to ensure proper connectivity.

OUTPUT:

1. UTP

2. STP

3.SSTP

4. COAXIAL CABLE

5. OPTICAL FIBER CABLE

Result

The different types of network cables were studied, and both straight-through and cross-wired cables were successfully prepared and tested according to the T568 standards.

EXP:03	Experiments on CISCO PACKET TRACER

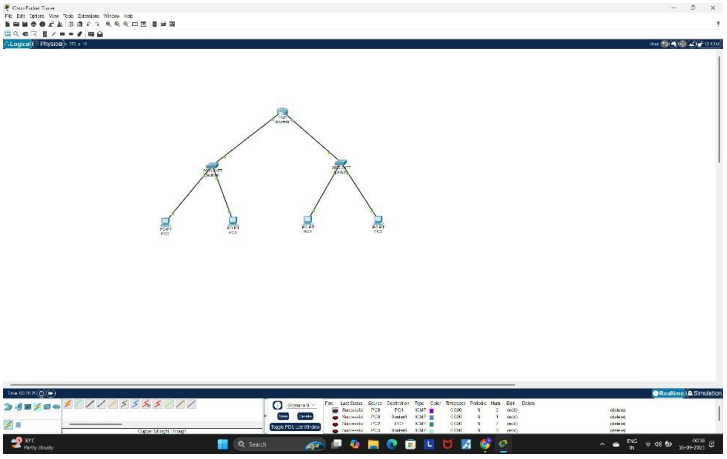
Aim

To understand the environment of CISCO PACKET TRACER, design a simple network, and analyze the behavior of network devices (HUB and SWITCH).

Algorithm / Procedure

1. **Launch** Cisco Packet Tracer and familiarize with the workspace and device library.
2. **Design** a small network (4-6 hosts) using only a **HUB** and observe packet transmission behavior (broadcast).
3. **Design** a small network (4-6 hosts) using only a **SWITCH** and observe packet transmission behavior (unicast/multicast after learning MAC addresses).
4. **Design** a network using both a **HUB and a SWITCH** to find out the functional difference between the two devices.
5. **Simulate** and trace communication between nodes in all three scenarios.

Output Images



Result

Simple networks were successfully designed and simulated in Cisco Packet Tracer. The difference in communication behavior between a HUB (Layer 1) and a SWITCH (Layer 2) was analyzed and observed.

EXP:04	**Setup and configure a LAN (Local area network) using a Switch and Ethernet cables in your lab.**

Aim

To set up and configure a Local Area Network (LAN) using a Switch and Ethernet cables and to test connectivity and file sharing.

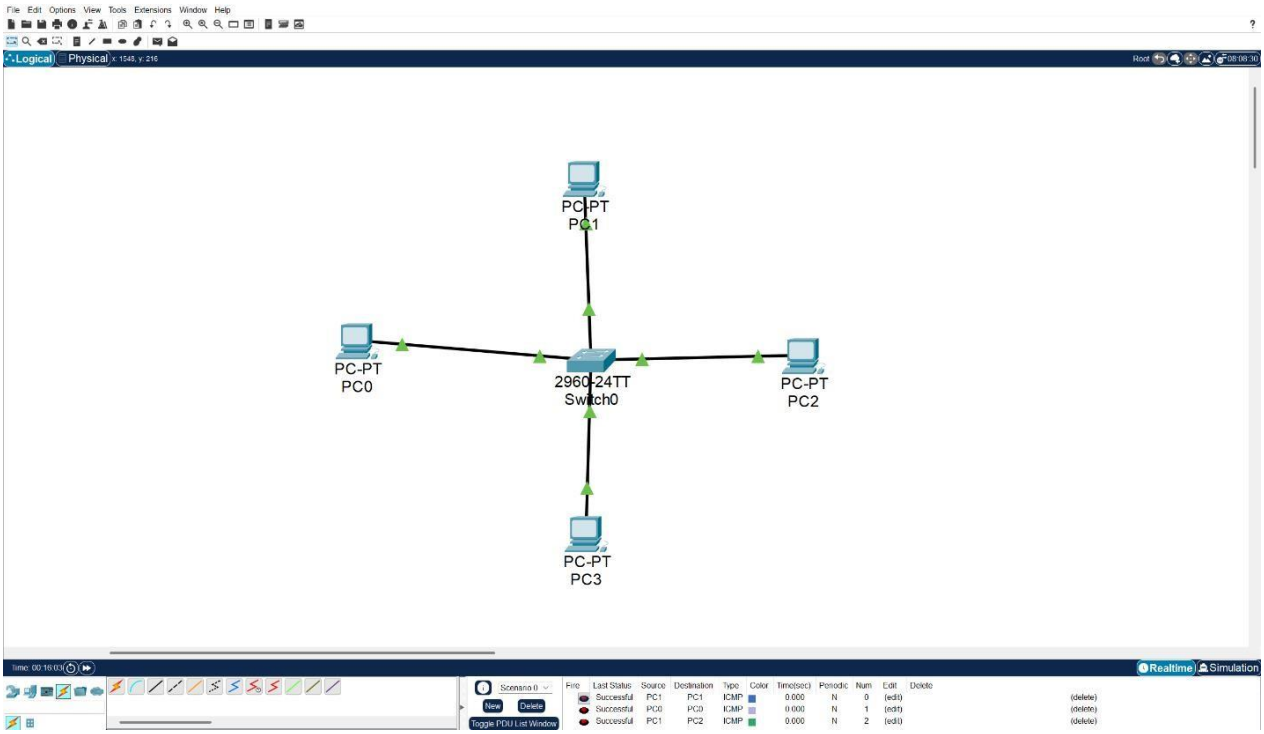
Algorithm / Procedure

1. **Connect** 3-4 host machines to a central Switch using Ethernet cables.
2. **Assign** static or dynamic IP addresses to each host machine within the same subnet.
3. **Verify** connectivity between all host machines using the `ping` command.
4. **Configure** file and folder sharing settings on one of the host machines.
5. **Access** the shared files and folders from the other machines on the LAN.

Result

A functional LAN was successfully set up and configured. Connectivity was verified using the `ping` command, and file sharing across the network was demonstrated.

Output Images



EXP:05	Experiments on Packet capture tool: Wireshark

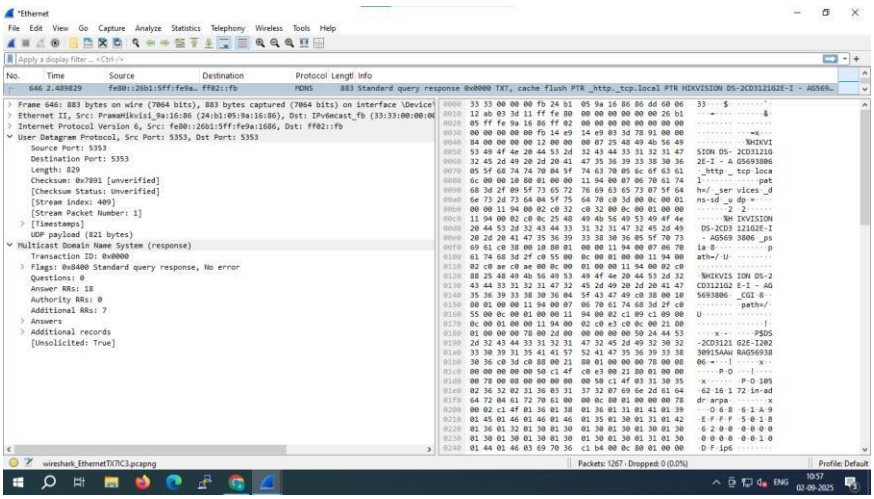
Aim

To understand the features of Wireshark as a packet capture tool and analyze the encapsulation of information at various layers of the Protocol stack.

Algorithm / Procedure

1. **Install** and launch the Wireshark packet capture tool.
2. **Select** the appropriate network interface for capturing traffic.
3. **Start** a packet capture session.
4. **Generate** network traffic (e.g., browse a website, ping a host).
5. **Stop** the capture and **analyze** the captured packets, focusing on the header information at the Data Link, Network, and Transport layers to understand encapsulation.
6. **Apply** filters (e.g., `http`, `tcp`, `ip.addr == x.x.x.x`) to isolate specific traffic.

Output Images



Result

Wireshark was successfully used to capture and analyze network traffic. The process of protocol encapsulation and the structure of packet headers at different layers were observed and understood.

EXP:06	**Error Correction at Data Link Layer**

Aim

To write a program to implement error detection and correction using the **Hamming Code** concept.

Algorithm / Procedure

1. **Determine** the number of redundant (parity) bits required for the given data size.
2. **Calculate** the positions of the parity bits (powers of 2).
3. **Implement** the Hamming Code generation algorithm:
   - Place data bits and parity bits in their respective positions.
   - Calculate the value of each parity bit based on the data bits it covers.
4. **Implement** the error detection and correction algorithm:
   - Receive the transmitted codeword.
   - Recalculate the parity bits.
   - Calculate the syndrome (error position) by combining the recalculated parity bits.
   - If the syndrome is non-zero, flip the bit at the error position.

5. **Test** the program with a data stream, introducing a single-bit error to verify the correction feature.

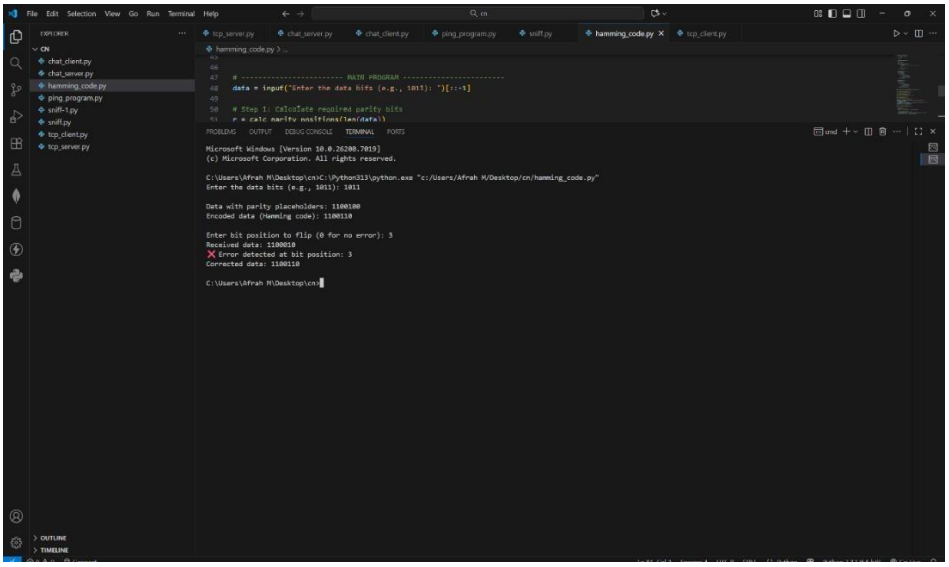
Code:

```
def calc_parity_positions(m):
    r = 0
    while (2**r) < (m + r + 1):
        r += 1
    return r


def insert_parity_bits(data, r):
    j = 0
    k = 1
    m = len(data)
    res = ''
    for i in range(1, m + r + 1):
        if i == 2**j:
            res += '0'  # parity bits start as 0 instead of 'P'
            j += 1
        else:
            res += data[-1 * k]
            k += 1
    return res[::-1]


def calc_parity_bits(arr, r):
    n = len(arr)
    arr = list(arr)
    for i in range(r):
```

```
        val = 0
        for j in range(1, n + 1):
            if j & (2**i) == (2**i):
                val ^= int(arr[-1 * j])
        arr[-1 * (2**i)] = str(val)
    return ''.join(arr)

def detect_error(arr, r):
    n = len(arr)
    res = 0
    for i in range(r):
        val = 0
        for j in range(1, n + 1):
            if j & (2**i) == (2**i):
                val ^= int(arr[-1 * j])
        res += val * (10**i)
    return int(str(res)[::-1], 2)

# ------------------------ MAIN PROGRAM ------------------------
data = input("Enter the data bits (e.g., 1011): ")[::-1]

# Step 1: Calculate required parity bits
r = calc_parity_positions(len(data))

# Step 2: Insert parity bits into data
arr = insert_parity_bits(data, r)
print("\nData with parity placeholders:", arr)
```

```
# Step 3: Calculate parity bits
arr = calc_parity_bits(arr, r)
print("Encoded data (Hamming code):", arr)

# Step 4: Introduce an error (optional)
error_index = int(input("\nEnter bit position to flip (0 for no error): "))
arr_with_error = list(arr)
if error_index != 0:
    arr_with_error[-error_index] = '1' if arr_with_error[-error_index] == '0' else '0'
arr_with_error = ''.join(arr_with_error)
print("Received data:", arr_with_error)

# Step 5: Detect error position
error_pos = detect_error(arr_with_error, r)

if error_pos == 0:
    print("No error detected.")
else:
    print(f"Error detected at bit position: {error_pos}")

    # Step 6: Correct the error
    arr_corrected = list(arr_with_error)
    arr_corrected[-error_pos] = '1' if arr_corrected[-error_pos] == '0' else '0'
    print("Corrected data:", ''.join(arr_corrected))
```

Output:



Result

A program to implement the Hamming Code was successfully written. The program demonstrated the ability to detect and correct a single-bit error in the transmitted data stream.

EXP:07	**Flow control at Data Link Layer**

Aim

To write a program to implement flow control at the data link layer using the **Sliding Window Protocol** and simulate the flow of frames from one node to another.

Algorithm / Procedure

1. **Define** the window size for the sender and receiver.
2. **Implement** the sender logic:
   - Maintain a sending window of sequence numbers.
   - Send frames within the window limit.
   - Start a timer for each unacknowledged frame.
3. **Implement** the receiver logic:
   - Maintain a receiving window.
   - Accept frames in order and send cumulative acknowledgments (ACKs).
   - Discard out-of-order frames (or buffer, depending on the specific protocol variant).

4. **Simulate** the flow of frames, including scenarios for successful transmission, lost frames, and lost ACKs, to demonstrate the flow control mechanism.

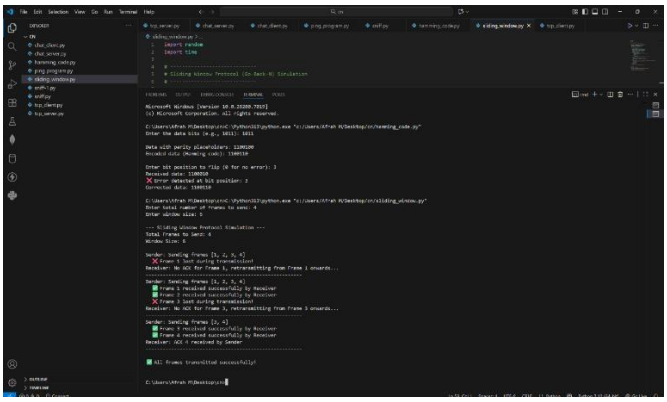
Code

```
import random
import time


# ----------------------------
# Sliding Window Protocol (Go-Back-N) Simulation
# ----------------------------


def sliding_window_simulation(total_frames, window_size):
    print("\n--- Sliding Window Protocol Simulation ---")
    print(f"Total Frames to Send: {total_frames}")
    print(f"Window Size: {window_size}\n")

    sent = 0  # Number of frames sent so far

    while sent < total_frames:
        # Determine frames in the current window
        window_end = min(sent + window_size, total_frames)
        current_window = list(range(sent + 1, window_end + 1))
        print(f"Sender: Sending frames {current_window}")

        # Simulate sending each frame in the window
        acked = True
        for frame in current_window:
            # Randomly simulate frame loss (20% chance)
            if random.random() < 0.2:
```

```
            print(f"Frame {frame} lost during transmission!")
            acked = False
            break
        else:
            print(f" Frame {frame} received successfully by Receiver")
            time.sleep(0.3)

    if acked:
        # All frames acknowledged → Slide window forward
        print(f"Receiver: ACK {window_end} received by Sender")
        sent = window_end
    else:
        # Go-Back-N retransmission
        print(f"Receiver: No ACK for Frame {frame}, retransmitting from Frame {frame} onwards...")
        time.sleep(1)
        # Sender will retransmit from the lost frame
        sent = frame - 1

    print("-" * 55)
    time.sleep(0.5)

print("\nAll frames transmitted successfully!\n")

# ----------------------------
# MAIN PROGRAM
# ----------------------------
if __name__== "__main__":
```

```
total_frames = int(input("Enter total number of frames to send: "))
window_size = int(input("Enter window size: "))

sliding_window_simulation(total_frames,  window_size)
```

Output:



Result

A program simulating the Sliding Window Protocol was successfully implemented. The simulation demonstrated effective flow control by managing the rate of frame transmission between the sender and receiver.

EXP:08	NMAP to Discover Live Hosts Using Nmap Scans (ARP, ICMP, TCP/UDP) on the TryHackMe Platform

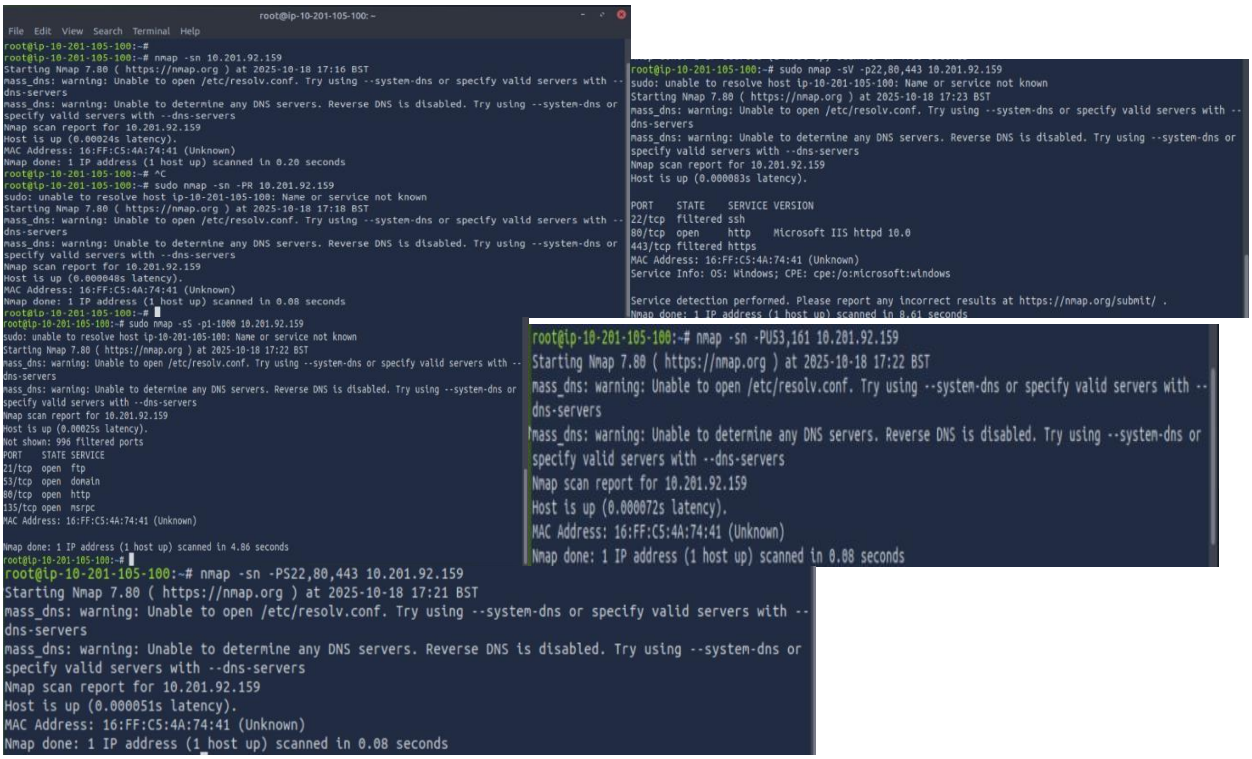
Aim

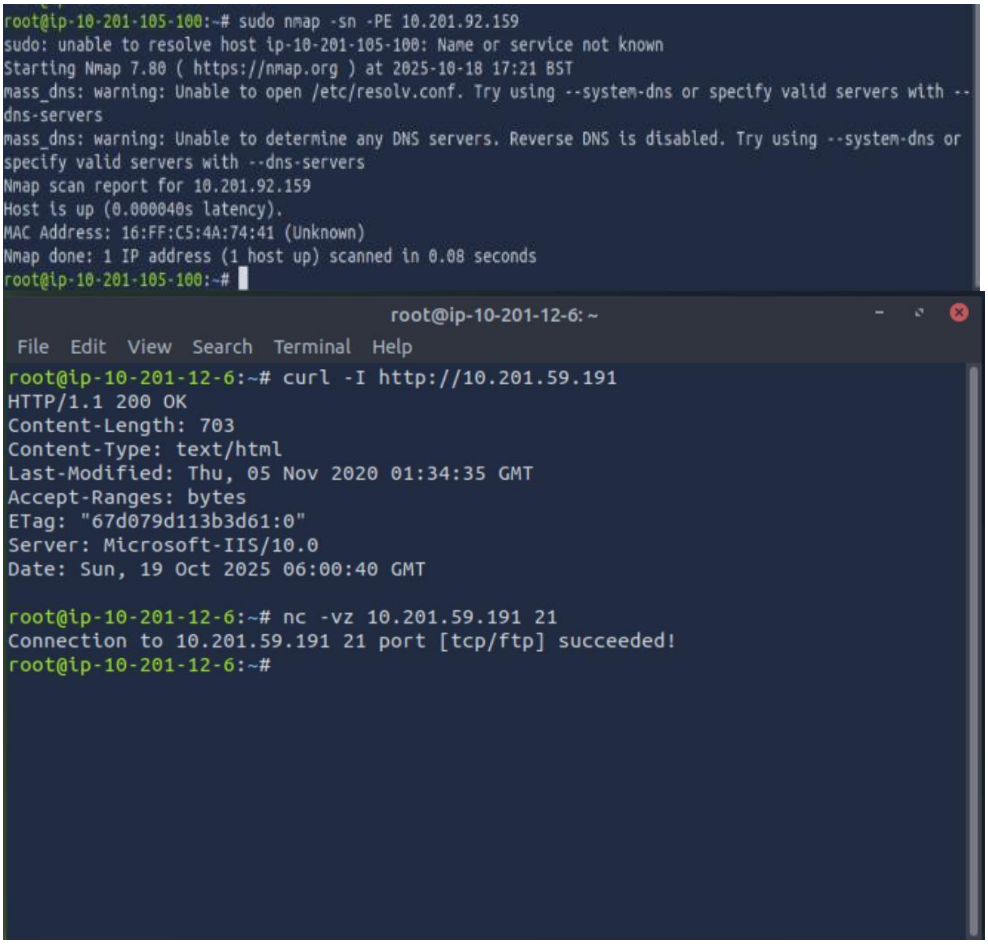
To use the **Nmap** tool to discover live hosts and analyze network services using various scan types (ARP, ICMP, TCP/UDP).

Algorithm / Procedure

1. **Access** the TryHackMe platform environment (or a similar controlled lab environment).
2. **Execute** an ARP scan (`nmap -sn -PR`) on the local subnet to discover active hosts.
3. **Execute** an ICMP echo request scan (`nmap -sn -PE`) to discover hosts that respond to ping.
4. **Perform** a TCP SYN scan (`nmap -sS`) on a target host to identify open ports.
5. **Perform** a UDP scan (`nmap -sU`) on a target host to identify open UDP services.
6. **Analyze** the results of each scan to understand the host's status and running services.

Output:





Result

Nmap was successfully used to perform various network scans. Live hosts were discovered, and the status of TCP and UDP ports was analyzed, demonstrating the utility of Nmap for network reconnaissance.

EXP:09	**Implementation of SUBNETTING in CISCO PACKET TRACER simulator**

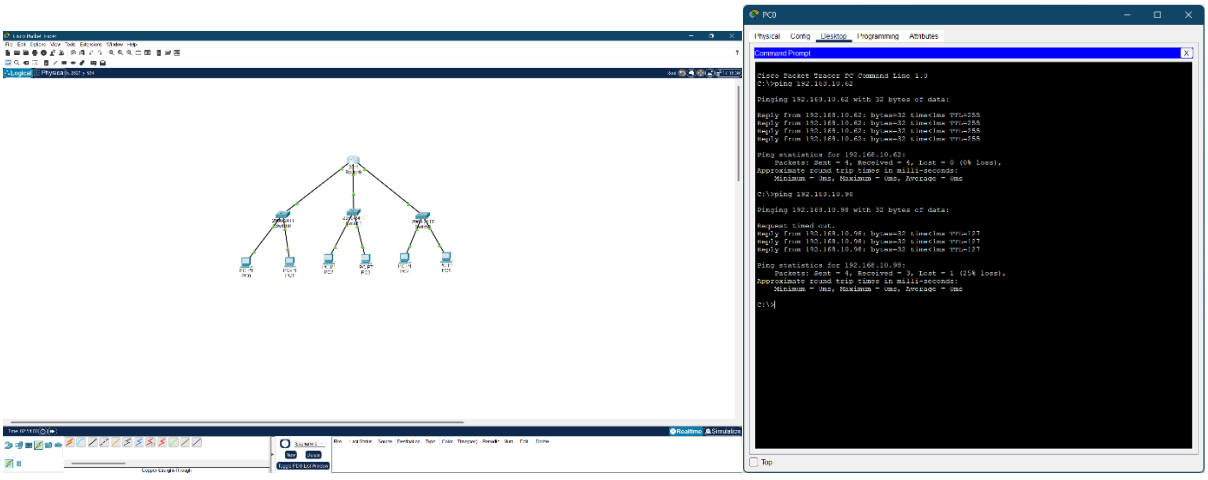
Aim

To implement and simulate **Subnetting** in the CISCO PACKET TRACER simulator and observe packet transmission across subnets.

Algorithm / Procedure

1. **Design** a network topology requiring multiple subnets (e.g., a single class C network divided into two or more subnets).
2. **Calculate** the subnet mask, network address, and host address range for each subnet.
3. **Assign** static IP addresses and the calculated subnet mask to hosts in each subnet.
4. **Connect** the subnets using a **Router**.
5. **Simulate** packet transmission:
   - Between hosts in the same subnet (should succeed).
   - Between hosts in different subnets **connected via a router** (should succeed).
   - Between hosts in different subnets **without a router** (should fail).

Output:



Result

Subnetting was successfully implemented and configured in Cisco Packet Tracer. The simulation verified that communication between different subnets requires a Layer 3 device (Router), while communication within a subnet does not.

EXP:10	**Internetworking with routers in CISCO PACKET TRACER simulator.**

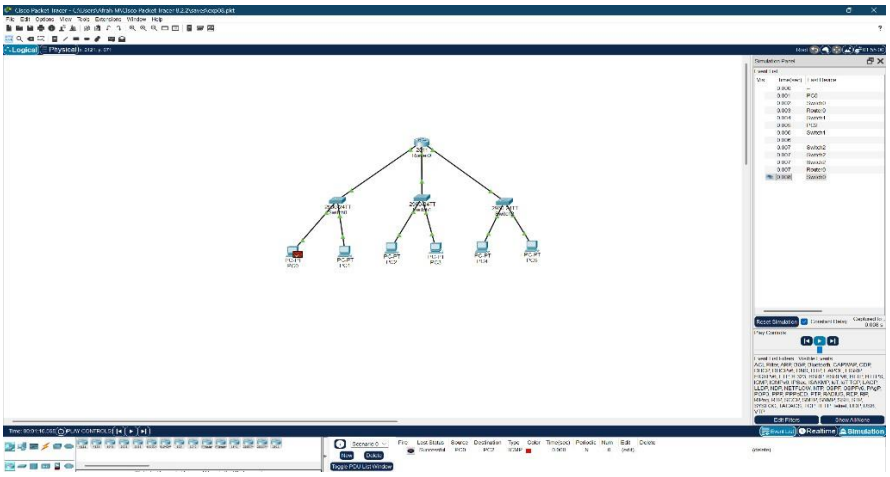
Aim

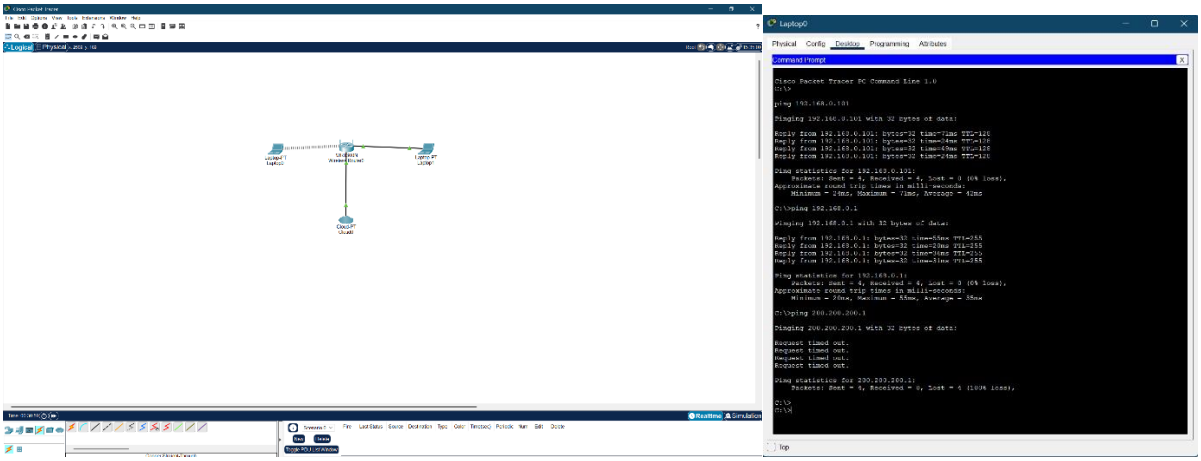
To design and configure a simple internetwork using a router, including static IP addressing, routing, and a wireless network with DHCP.

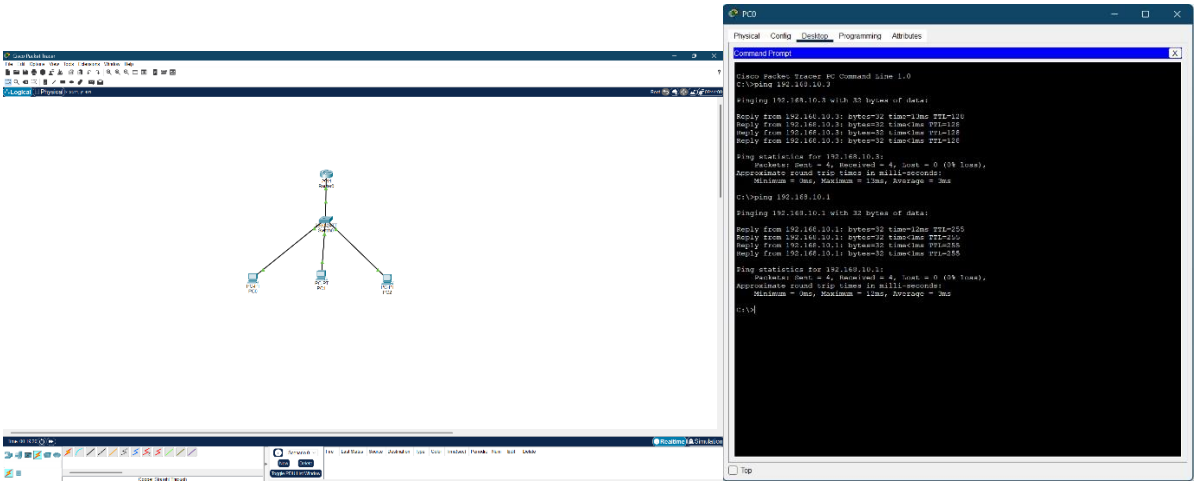
Algorithm / Procedure

1. **Design** a simple internetwork topology with two or more different networks (3-4 hosts each) connected via a Router.
2. **Configure** static IP addresses on all host machines and the router interfaces.
3. **Perform** simulation and trace how routing is done in packet transmission between the different networks.
4. **Design** and configure a separate internetwork using a **wireless router**, **DHCP server**, and an **internet cloud** to simulate a home/office network.
5. **Verify** connectivity and IP address assignment in the wireless network.

Output:







Result

A simple internetwork with a router was successfully designed and configured, demonstrating the process of inter-network routing. A wireless network with DHCP was also configured and verified.

EXP:11	Routing at Network Layer

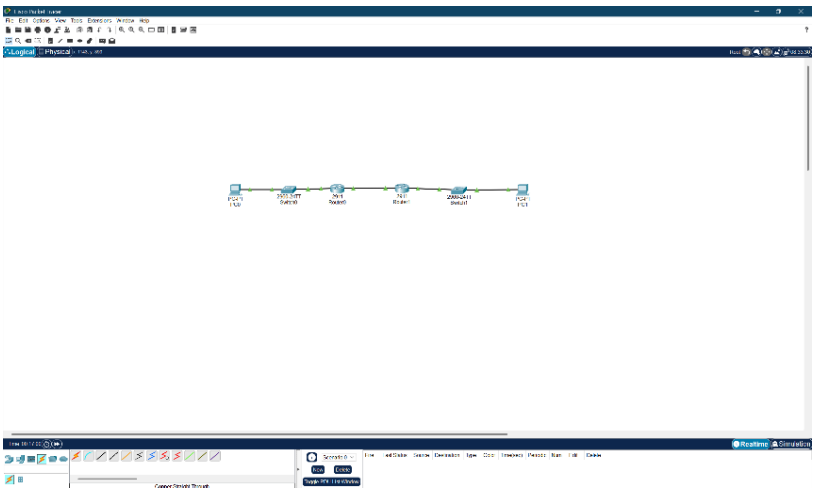
Aim

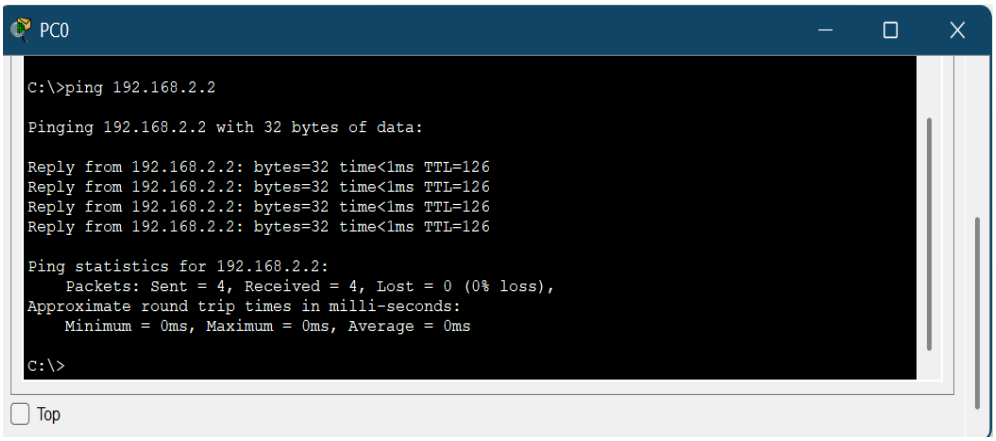
To simulate **Static Routing Protocol** and **RIP (Routing Information Protocol)** configuration using CISCO Packet Tracer.

Algorithm / Procedure

1. **Design** a multi-network topology (e.g., three routers and three networks).
2. **Simulate Static Routing:**
   - Configure the router interfaces with IP addresses.
   - Manually add static routes to the routing table of each router to reach all other networks.
   - Verify end-to-end connectivity.
3. **Simulate RIP (Dynamic Routing):**
   - Remove the static routes.
   - Enable the RIP routing protocol on all routers.
   - Advertise the directly connected networks.
   - Verify that the routing tables are automatically populated and check end-to-end connectivity.

Output:





Result

Both Static Routing and RIP dynamic routing protocols were successfully simulated in Cisco Packet Tracer. The experiment demonstrated the manual configuration of static routes versus the automatic route discovery of RIP.

EXP:12	End –End Communication at Transport Layer

Aim

To implement an echo client-server using TCP/UDP sockets and to implement a chat program using socket programming.

Algorithm / Procedure

1. **Implement** a basic **TCP Echo Server** that listens for connections, receives a message, and sends the same message back to the client.
2. **Implement** a basic **TCP Echo Client** that connects to the server, sends a message, and prints the echoed response.
3. **Implement** a simple **Chat Program** using TCP sockets, allowing two separate programs (client and server) to send and receive messages interactively.
4. *(Optional)* Implement the same Echo Client-Server using **UDP** sockets to compare connection-oriented vs. connectionless communication.

Code:

tcp_client.py

```
import socket


s = socket.socket()
s.connect(('localhost', 12345))


while True:
    msg = input("Client: ")
    if msg.lower() == 'exit':
        break
    s.send(msg.encode())
    data = s.recv(1024).decode()
    print("From Server:", data)
```

```
s.close()
```

tcp_server.py

```
import socket

s = socket.socket()
s.bind(('localhost', 12345))
s.listen(1)
print("Server ready, waiting for connection...")
conn, addr = s.accept()
print("Connected with", addr)

while True:
   data = conn.recv(1024).decode()
   if not data:
      break
   print("From Client:", data)
   conn.send(data.encode())

conn.close()
```

Client_server.py

```
import socket

s = socket.socket()
s.bind(('localhost', 12345))
```

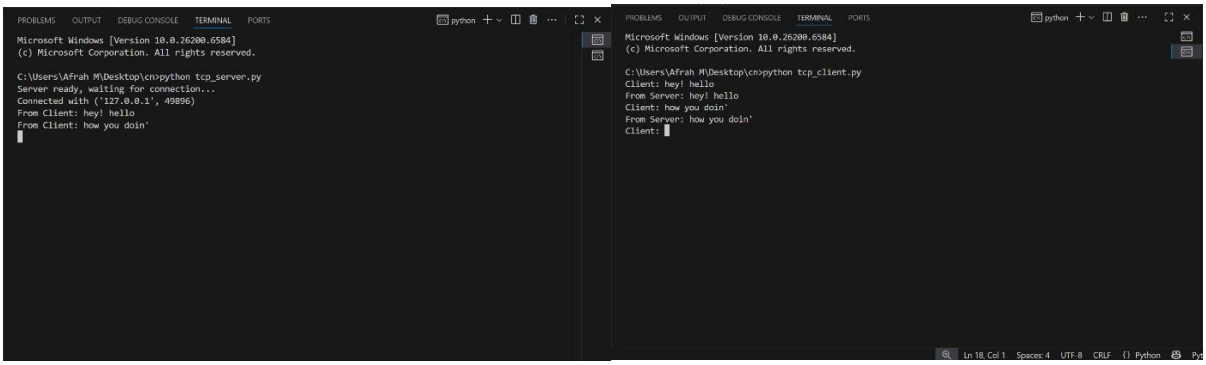
```
s.listen(1)
print("Chat Server waiting for connection...")
conn, addr = s.accept()
print("Connected with", addr)

while True:
    msg = conn.recv(1024).decode()
    if msg.lower() == 'bye':
        print("Client ended chat.")
        break
    print("Client:", msg)
    reply = input("Server: ")
    conn.send(reply.encode())

conn.close()
```

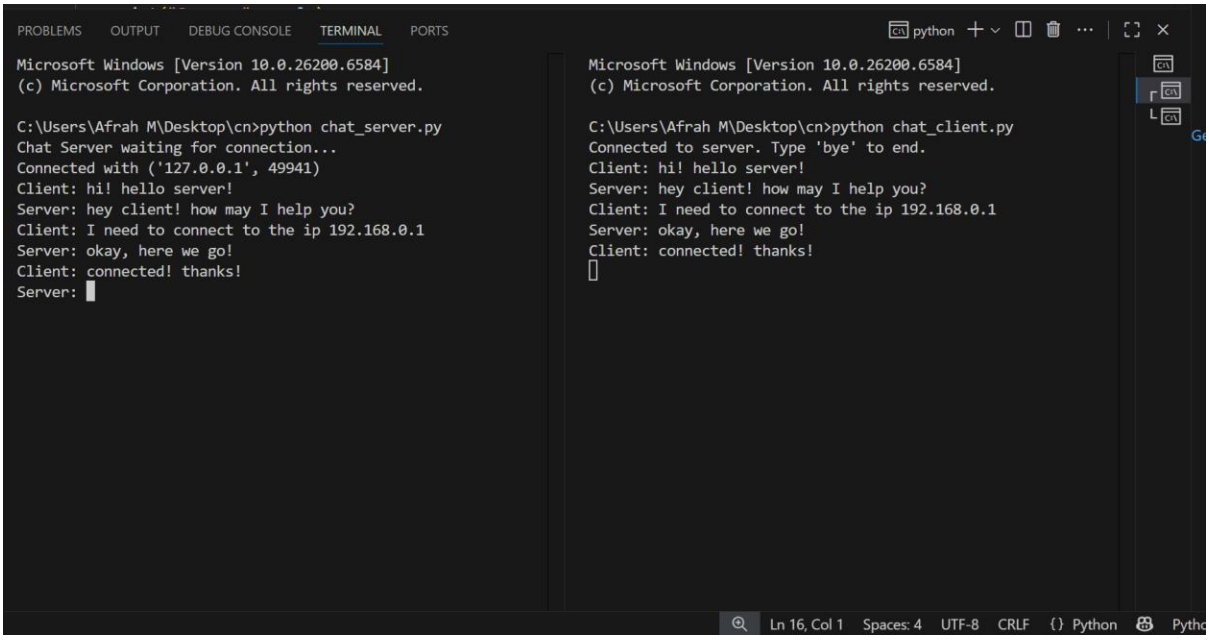
client.py

```
import socket

s = socket.socket()
s.connect(('localhost', 12345))
print("Connected to server. Type 'bye' to end.")

while True:
    msg = input("Client: ")
    s.send(msg.encode())
    if msg.lower() == 'bye':
```

```
        break
    reply = s.recv(1024).decode()
    print("Server:", reply)

s.close()
```

Output:





Result

An echo client-server and a chat program were successfully implemented using socket programming. The experiment demonstrated end-to-end communication at the Transport Layer using TCP sockets.

EXP:13	Implement your own ping program

Aim

To implement a custom **ping program** to test connectivity to a remote host.

Algorithm / Procedure

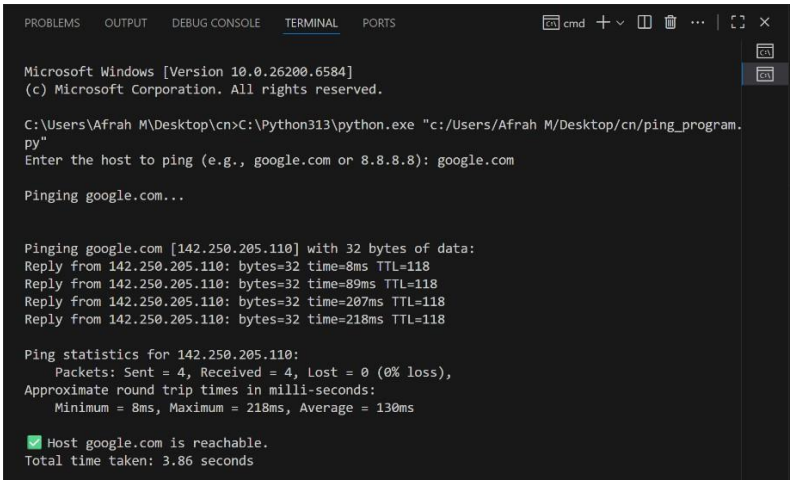
1. **Understand** the structure of an ICMP Echo Request and Echo Reply message.
2. **Implement** a program using raw sockets (or a suitable library) to construct and send an ICMP Echo Request packet to a target host.
3. **Implement** logic to listen for and receive the corresponding ICMP Echo Reply packet.
4. **Calculate** the Round Trip Time (RTT) by measuring the time difference between sending the request and receiving the reply.
5. **Display** the results, including the RTT and packet loss statistics, similar to the standard `ping` utility.

Code

```
import os

import platform

import time


# Get the target host

host = input("Enter the host to ping (e.g., google.com or 8.8.8.8): ")


# Detect OS command

param = '-n' if platform.system().lower() == 'windows' else '-c'


# Ping 4 times

print(f"\nPinging {host}...\n")

start = time.time()

response = os.system(f"ping {param} 4 {host}")

end = time.time()
```

```
if response == 0:
    print(f"\nHost {host} is reachable.")
else:
    print(f"\nHost {host} is unreachable.")

print(f"Total time taken: {round(end - start, 2)} seconds")
```

Output:



Result

A custom ping program was successfully implemented. The program demonstrated the ability to send and receive ICMP packets and calculate the Round Trip Time to a target host.

EXP:14	Write a code using RAW sockets to implement packet sniffing

Aim

To write a program using **RAW sockets** to implement a basic **packet sniffer** and capture network traffic.

Algorithm / Procedure

1. **Understand** the concept of raw sockets, which allow access to the data link layer or network layer headers.
2. **Create** a raw socket that can capture all incoming packets on a specified interface.
3. **Implement** a loop to continuously receive packets from the raw socket.
4. **Parse** the captured raw data to extract and display key information from the Ethernet, IP, and TCP/UDP headers (e.g., Source/Destination MAC, Source/Destination IP, Source/Destination Port).
5. **Test** the sniffer by generating traffic (e.g., browsing, pinging) and observing the captured packet details.

Code

```
import socket

import struct

import textwrap

import time

import sys

import ctypes


# Windows-only constants

SIO_RCVALL = 0x98000001

RCVALL_ON = 1

RCVALL_OFF = 0
```

```
def mac_addr(bytes_addr):
    return ':'.join('{:02x}'.format(b) for b in bytes_addr)


def ipv4(addr_bytes):
    return '.'.join(map(str, addr_bytes))


def hexdump(src, length=16):
    results = []
    digits = 2
    for i in range(0, len(src), length):
        chunk = src[i:i+length]
        hexa = ' '.join(f'{b:02x}' for b in chunk)
        text = ''.join((chr(b) if 32 <= b < 127 else '.') for b in chunk)
        results.append(f'{i:04x}  {hexa:<{length*(digits+1)}}  {text}')
    return '\n'.join(results)


def parse_ip_header(data):
    # Unpack first 20 bytes of IP header
    iph = struct.unpack('!BBHHHBBH4s4s', data[:20])
    version_ihl = iph[0]
    version = version_ihl >> 4
    ihl = (version_ihl & 0xF) * 4
    tos = iph[1]
    total_length = iph[2]
    identification = iph[3]
    flags_offset = iph[4]
    ttl = iph[5]
    protocol = iph[6]
```

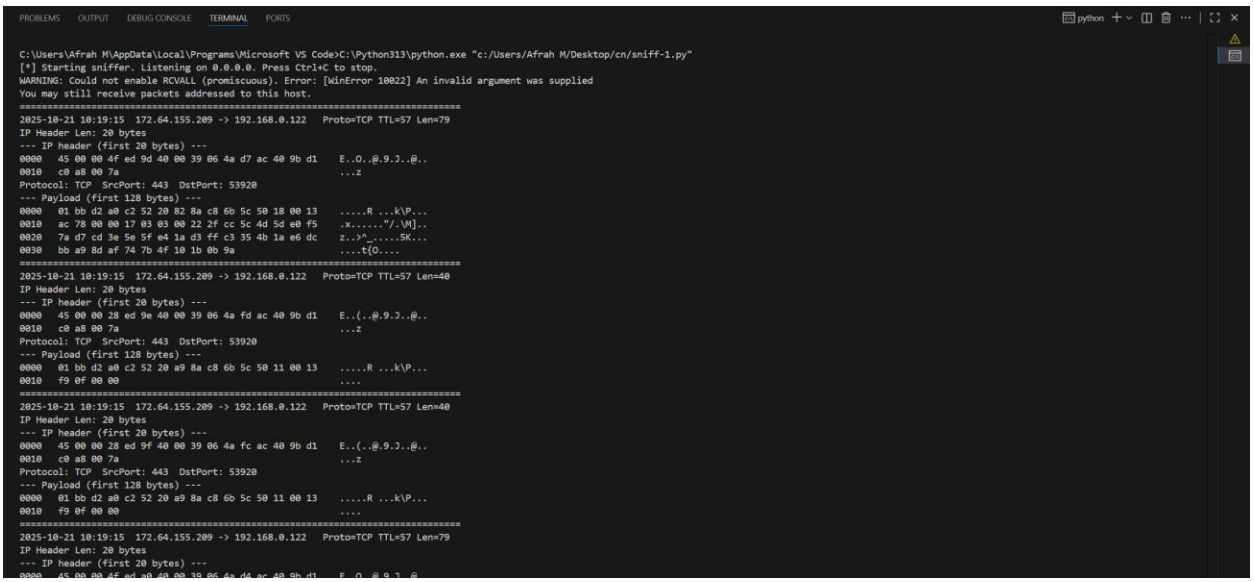
```
    checksum = iph[7]
    src = ipv4(iph[8])
    dst = ipv4(iph[9])
    return {
        'version': version,
        'ihl': ihl,
        'tos': tos,
        'total_length': total_length,
        'id': identification,
        'flags_offset': flags_offset,
        'ttl': ttl,
        'protocol': protocol,
        'checksum': checksum,
        'src': src,
        'dst': dst,
        'payload': data[ihl:total_length]
    }

def protocol_name(p):
    return {1: 'ICMP', 6: 'TCP', 17: 'UDP'}.get(p, str(p))

def main(listen_addr='0.0.0.0'):
    print(f'[*] Starting sniffer. Listening on {listen_addr}. Press Ctrl+C to stop.')
    try:
        # Create RAW socket
        s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
    except PermissionError:
        print('ERROR: Must run as Administrator to create raw socket.')
```

```
        sys.exit(1)


    # Bind to interface
    s.bind((listen_addr, 0))


    # Include IP header in captured packets
    s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)


    # Enable promiscuous mode (Windows-specific)
    # Use ioctls via socket.ioctl
    try:
        s.ioctl(SIO_RCVALL, RCVALL_ON)
    except Exception as e:
        print('WARNING: Could not enable RCVALL (promiscuous). Error:', e)
        print('You may still receive packets addressed to this host.')


    try:
        while True:
            raw_data, addr = s.recvfrom(65535)
            ts = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
            ip = parse_ip_header(raw_data)
            print('='*80)
            print(f'{ts}  {ip["src"]} -> {ip["dst"]}   Proto={protocol_name(ip["protocol"])}
TTL={ip["ttl"]} Len={ip["total_length"]}')
            # Show first 64 bytes of payload as hex dump
            header_len = ip['ihl']
            print(f'IP Header Len: {header_len} bytes')
            print('--- IP header (first 20 bytes) ---')
```

```
        print(hexdump(raw_data[:header_len], length=16))
        # Determine protocol and print simple port info for TCP/UDP
        if ip['protocol'] == 6 and len(ip['payload']) >= 4:  # TCP
            src_port, dst_port = struct.unpack('!HH', ip['payload'][:4])
            print(f'Protocol: TCP  SrcPort: {src_port}  DstPort: {dst_port}')
        elif ip['protocol'] == 17 and len(ip['payload']) >=4:  # UDP
            src_port, dst_port = struct.unpack('!HH', ip['payload'][:4])
            print(f'Protocol: UDP  SrcPort: {src_port}  DstPort: {dst_port}')
        else:
            print(f'Protocol: {protocol_name(ip["protocol"])}')

        # Show payload (first 128 bytes)
        payload = ip['payload'][:128]
        if payload:
            print('--- Payload (first 128 bytes) ---')
            print(hexdump(payload, length=16))
        else:
            print('No payload (or payload skipped).')
    except KeyboardInterrupt:
        print('\n[*] Stopping sniffer.')
    finally:
        # Turn off promiscuous
        try:
            s.ioctl(SIO_RCVALL, RCVALL_OFF)
        except Exception:
            pass
        s.close()
```

```
if __name__== '__main__':
    # If you want to listen on a specific local IP, pass it here; default is 0.0.0.0
    main()
```

Output:



Result

A basic packet sniffer was successfully implemented using raw sockets. The program demonstrated the ability to capture and parse raw network packets, extracting header information from different protocol layers.

EXP:15	Analyse various types of servers using Webalizer tool

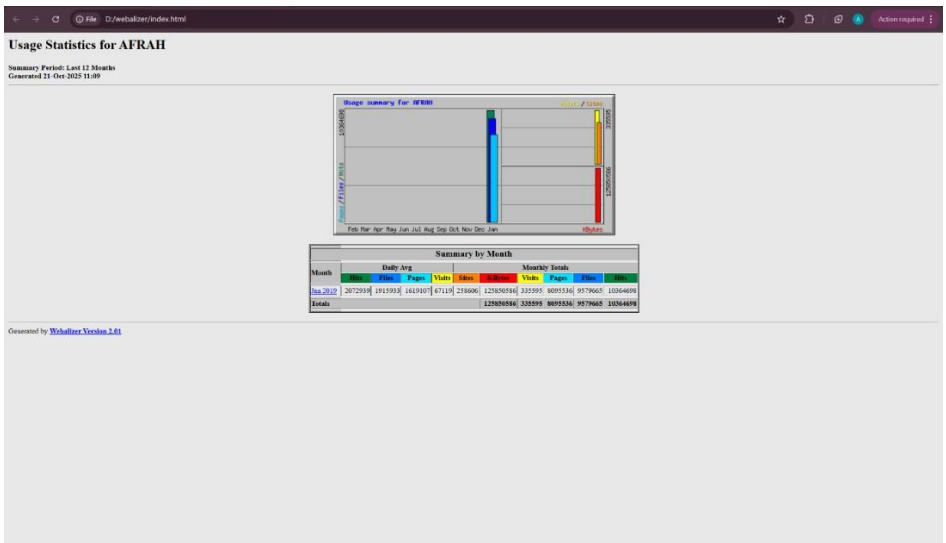
Aim

To analyze various types of servers (e.g., Web, FTP) by processing their log files using the **Webalizer** tool.

Algorithm / Procedure

1. **Install** and configure the Webalizer tool (or a similar log analysis tool).
2. **Obtain** a sample log file from a server (e.g., Apache access log, FTP log).
3. **Run** Webalizer on the log file, specifying the output directory for the generated reports.
4. **Analyze** the generated HTML reports, focusing on key metrics such as:
   - Monthly/Daily/Hourly statistics.
   - Top URLs, referrers, and search strings.
   - User agents (browsers/OS).
   - Country statistics.
5. **Document** the key findings and insights about the server usage.

Output:



Result

The Webalizer tool was successfully used to process server log files. A comprehensive analysis of server traffic and user behavior was performed, demonstrating the utility of log analysis tools.