Networking with Linux Lab

Subject Code: MCAL26

A Practical Journal Submitted in Fulfilment of

the Degree of

MASTER

In

COMPUTER APPLICATION

Year 2023-2024

By

Mr. Agrawal Yash Gopal

(Application Id: - 75706)

Semester- II (CBCS)

Institute of Distance and Open Learning

Vidya Nagari, Kalina, Santacruz East – 400098.

University of Mumbai

PCP Centre

[Vidyavardhini’s College of Technology – Vasai Road, Palghar 401202]

Institute of Distance and Open Learning,

Vidya Nagari, Kalina, Santacruz (E) -400098

CERTIFICATE

This to certify that, **Mr. Agrawal Yash Gopal** appearing **Master in Computer Application (Semester II - CBCS) Application ID: 75706** has satisfactorily completed the prescribed practical of **MCAL26- Networking with Linux Lab** as laid down by the University of Mumbai for the academic year 2023-24

Teacher in charge

Examiners

Coordinator
IDOL, MCA
University of Mumbai

Date: -22/06/2024

Place: - Vasai

Index

Sr. No.	Practical	Signature
1.	Installation of NS-3 in Linux.	
2.	Installation of NetAnim.	
3.	Installation of Wireshark.	
4.	Program to simulate traffic between two nodes.	
5.	Program to simulate star topology.	
6.	Program to simulate bus topology.	
7.	Program to simulate mesh topology.	
8.	Program to simulate UDP server client.	
9.	Animate a simple network using NetAnim in Network Simulator.	
10.	Analyze the network traffic using Wireshark.	

EXPERIMENT 1

Aim: Installation of NS-3 in Linux

Theory:

Network Simulator 3: (https://www.nsnam.org/)

NS3 (Network Simulator 3) is a discrete-event network simulator primarily used for research and educational purposes. It is designed to simulate the behavior and performance of computer networks. NS3 provides a comprehensive framework for modeling a wide range of network protocols, both wired and wireless, and allows researchers to evaluate the performance of these protocols under various scenarios. It offers a high level of flexibility and extensibility, with support for custom protocols and detailed packet-level simulation. NS3 is widely used for studying network performance, testing new network technologies, and developing new network protocols.

Here's a detailed guide to help you with the process:

Step 1: Install Dependencies

Before installing NS-3, you need to install the necessary dependencies. Open a terminal and run the following commands:

-> sudo apt update

-> sudo apt install build-essential autoconf automake libxmu-dev gcc g++ python3 python3-dev python3-pip git mercurial qt5-qmake qt5-default gnuplot xgraph

Step 2: Download NS-3

Download the latest version of NS-3 from its official repository. You can use ‘git’ to clone the repository:

-> cd ~

-> git clone https://gitlab.com/nsnam/ns-3-dev.git ns-3-dev

-> cd ns-3-dev

Step 3: Verify Python Bindings

Make sure Python bindings are enabled. NS-3 requires Python for configuration and running simulations:

-> ./waf configure --enable-python-bindings

Step 4: Build NS-3

After configuring, build NS-3 using the waf build tool:

->./waf build

Step 5: Set Environment Variables

Set the environment variables to include NS-3 binaries in your PATH. You can add the following lines to your .bashrc or .bash_profile file:

-> echo 'export PATH=~/ns-3-dev/build:$PATH' >> ~/.bashrc

-> source ~/.bashrc

Step 6: Test the Installation

To verify that NS-3 has been installed correctly, run a simple example:

-> cd ~/ns-3-dev

-> ./waf --run hello-simulator

EXPERIMENT 2

Aim : Installation of NetAnim

Theory :

NetAnim : (https://code.nsnam.org/netanim)

NetAnim is a graphical user interface application designed to animate and visualize the results of network simulations conducted using the NS-3 network simulator. It provides a dynamic and interactive way to observe network topology, packet flows, and node movements over time, aiding in the analysis and understanding of network behaviors and performance. NetAnim is particularly useful for educational purposes and for researchers to validate and present their simulation outcomes in a visual format.

Install NetAnim (Network Animator)

NetAnim is a visualizer for NS-3 simulations. To install it, follow these steps:

1. Install the required packages:

-> sudo apt install qt5-qmake qt5-default

2. Download and build NetAnim:

-> cd ~

-> git clone https://gitlab.com/nsnam/netanim.git

-> cd netanim

-> qmake NetAnim.pro

-> make

3. Run NetAnim to ensure it works:

-> ./NetAnim

EXPERIMENT 3

Aim : Installation of WireShark

Theory :

Wireshark : (https://www.wireshark.org/download.html)

Wireshark is a widely-used network protocol analyzer that allows users to capture and interactively browse the traffic running on a computer network. It provides deep inspection of hundreds of protocols and is used for network troubleshooting, analysis, software and protocol development, and education. Wireshark offers features such as live capture, offline analysis, rich VoIP analysis, decryption support for many protocols, and customizable reports. It runs on multiple platforms including Windows, Linux, macOS, and UNIX.

To install Wireshark on a Linux system (here, Ubantu), follow these steps. The specific commands might vary slightly based on your Linux distribution.

1. Update your package list

-> sudo apt update

2. Install Wireshark

-> sudo apt install wireshark

3. Optional: Allow non-root users to capture packets: During the installation, you may be asked if non-superusers should be able to capture packets. If you missed this prompt, you can configure it later:

-> sudo dpkg-reconfigure wireshark-common

4. Add your user to the 'wireshark' group

-> sudo usermod -aG wireshark $USER

5. Restart your session: Log out and log back in, or reboot your system, for the group changes to take effect.

EXPERIMENT 4

Aim : Program to simulate traffic between two nodes

Theory :

Simulating traffic between two nodes can be done in various ways depending on the complexity and the level of detail required. Here's a basic example in Python that simulates data packets being sent between two nodes. This example will focus on a simple network where packets are sent from Node A to Node B and provides some basic statistics about the traffic.

We'll use the following concepts:

- **Node:** Represents a point in the network.
- **Packet:** Represents a unit of data being sent from one node to another.
- **Traffic Simulation:** Simulates the process of sending packets from one node to another over a specified duration.

Here is a basic Python script to simulate traffic between two nodes:

```
import random
import time
from collections import deque

class Node:
    def __init__(self, name):
        self.name = name

class Packet:
    def __init__(self, id, source, destination, size):
        self.id = id
        self.source = source
        self.destination = destination
        self.size = size
        self.timestamp = time.time()

class TrafficSimulator:
    def __init__(self, node_a, node_b):
        self.node_a = node_a
        self.node_b = node_b
        self.packets = deque()
        self.total_packets_sent = 0
```

```python
        self.total_data_sent = 0

    def generate_packet(self):
        packet_size = random.randint(50, 1500)  # Packet size in bytes
        packet = Packet(
            id=self.total_packets_sent + 1,
            source=self.node_a,
            destination=self.node_b,
            size=packet_size
        )
        self.packets.append(packet)
        self.total_packets_sent += 1
        self.total_data_sent += packet_size
        print(f"Packet {packet.id} sent: {packet.size} bytes")

    def simulate_traffic(self, duration, interval):
        start_time = time.time()
        while time.time() - start_time < duration:
            self.generate_packet()
            time.sleep(interval)

    def statistics(self):
        print("\nSimulation Statistics:")
        print(f"Total packets sent: {self.total_packets_sent}")
        print(f"Total data sent: {self.total_data_sent} bytes")
        if self.total_packets_sent > 0:
            print(f"Average packet size: {self.total_data_sent / self.total_packets_sent} bytes")

# Create nodes
node_a = Node("Node A")
node_b = Node("Node B")

# Initialize simulator
simulator = TrafficSimulator(node_a, node_b)

# Run simulation for 10 seconds, generating a packet every 1 second
simulator.simulate_traffic(duration=10, interval=1)

# Print statistics
simulator.statistics()
```

Explanation:

1. **Node Class**: Represents a network node with a name.

2. **Packet Class**: Represents a packet with an ID, source, destination, size, and timestamp.

3. **TrafficSimulator Class**: Manages the traffic simulation.
    - __init__: Initializes the simulator with two nodes.
    - generate_packet: Generates a packet with a random size and logs the sending.
    - simulate_traffic: Simulates the traffic for a specified duration and interval between packet generations.
    - statistics: Prints out the statistics of the simulation, such as total packets sent and total data sent.

This script will simulate traffic between two nodes for a given duration, generating packets at specified intervals, and finally print out some basic statistics.

Output:

```
Packet 1 sent: 915 bytes
Packet 2 sent: 1075 bytes
Packet 3 sent: 1146 bytes
Packet 4 sent: 1379 bytes
Packet 5 sent: 520 bytes
Packet 6 sent: 109 bytes
Packet 7 sent: 770 bytes
Packet 8 sent: 591 bytes
Packet 9 sent: 1040 bytes
Packet 10 sent: 1094 bytes

Simulation Statistics:
Total packets sent: 10
Total data sent: 8639 bytes
Average packet size: 863.9 bytes

=== Code Execution Successful ===
```

EXPERIMENT 5

Aim : Program to simulate star topology

Theory :

To simulate a star topology, we need to create a network with a central hub (or switch) and multiple nodes connected to it. We'll use Python and the networkx library to achieve this. networkx is a powerful library for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

Here's a step-by-step guide to simulate a star topology:

1. Install the required library: If you haven't already installed networkx, you can do so using pip:

-> pip install networkx matplotlib

2. Create the star topology: We'll create a function to generate and visualize a star topology.

```
import networkx as nx
import matplotlib.pyplot as plt

def create_star_topology(num_nodes):
    # Create an empty graph
    G = nx.Graph()
```

```
    # Add the central node (hub)
    hub = 0
    G.add_node(hub)

    # Add other nodes and connect them to the hub
    for i in range(1, num_nodes + 1):
        G.add_node(i)
        G.add_edge(hub, i)

    # Draw the graph
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_size=700, node_color='lightblue', font_size=10,
font_color='black', edge_color='gray')
    plt.title("Star Topology with {} nodes".format(num_nodes))
    plt.show()

# Example usage
create_star_topology(5)
```

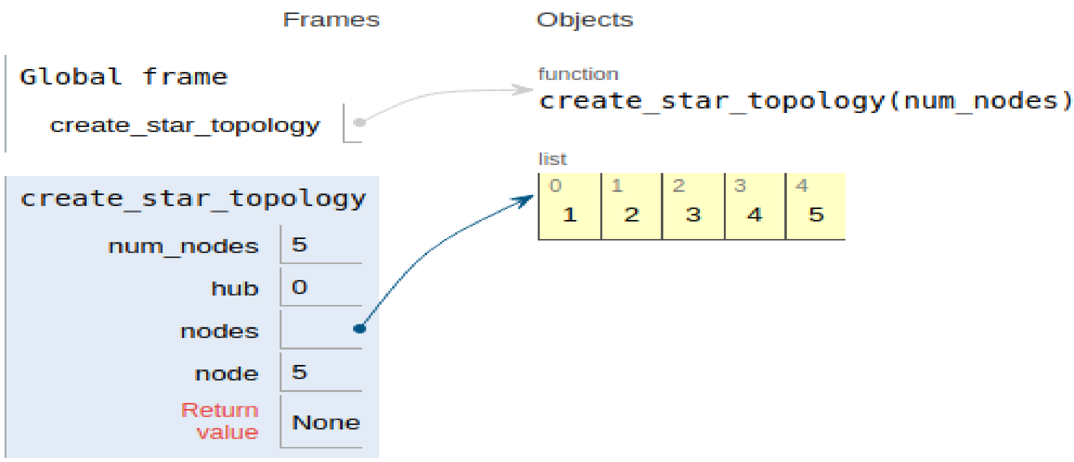
Explanation of the code:

- **Import Libraries**: We import networkx for creating and managing the network graph and matplotlib.pyplot for visualization.
- **Function Definition**: create_star_topology(num_nodes) takes the number of nodes as an argument and creates a star topology.
- **Create Graph**: We initialize an empty graph G.
- **Add Central Hub**: We add the central node (hub), which we label as 0.
- **Add Nodes and Edges**: We add the rest of the nodes and connect each to the hub.
- **Draw the Graph**: We use spring_layout to position the nodes and nx.draw to visualize the graph.

Run the Example: The example at the end creates a star topology with 5 nodes connected to a central hub.

Output:

```
Central Hub Node: 0
Peripheral Nodes:  [1, 2, 3, 4, 5]
Connections:
0 -- 1
0 -- 2
0 -- 3
0 -- 4
0 -- 5
```

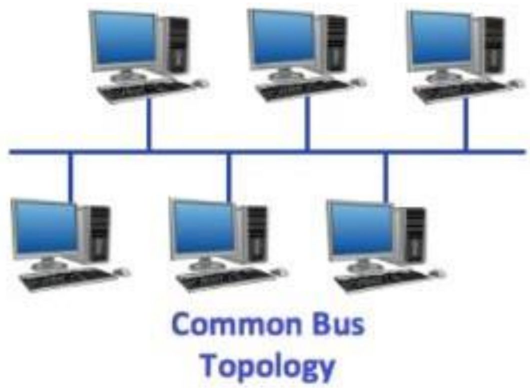


EXPERIMENT 6

Aim : Program to simulate bus topology

Theory :

Simulating a bus topology in a program involves creating a simple network where multiple devices are connected to a common communication medium (the bus).



Here's a basic Python program that simulates a bus topology:

```
import random

class Device:
    def _init_(self, name):
        self.name = name

    def send_message(self):
        message = f"Message from {self.name}"
        return message

class BusTopology:
    def _init_(self, devices):
        self.devices = devices

    def send_message(self):
        sender = random.choice(self.devices)
        message = sender.send_message()
        print(f"Bus received: {message}")
```

```
if _name___== "_main_":
    # Create some devices
    device_names = ["Device1", "Device2", "Device3", "Device4"]
    devices = [Device(name) for name in device_names]

    # Create a bus topology with these devices
    bus_network = BusTopology(devices)

    # Simulate sending messages
    for _ in range(5):
        bus_network.send_message()
```

Explanation:

1. **Device Class**: Represents a device in the network. Each device has a name attribute and a send_message method that generates a message.

2. **BusTopology Class**: Represents the bus communication medium. It has a list of devices connected to it. It has a send_message method which randomly selects a device from the list of devices and calls its send_message method to simulate sending a message over the bus.

3. **Main Execution**:
    - Devices are created with names "Device1" to "Device4".
    - These devices are then instantiated and added to the BusTopology.
    - The program simulates sending 5 messages by calling bus_network.send_message() repeatedly.

How it simulates a bus topology:

- In a bus topology, all devices are connected to a single communication medium (the bus).
  In the program:
    - Devices are represented by instances of the Device class.
    - The bus is represented by the BusTopology class, which holds references to all connected devices.
    - Messages are sent by selecting a random device and simulating the message transmission over the bus (printing the message).

Output:

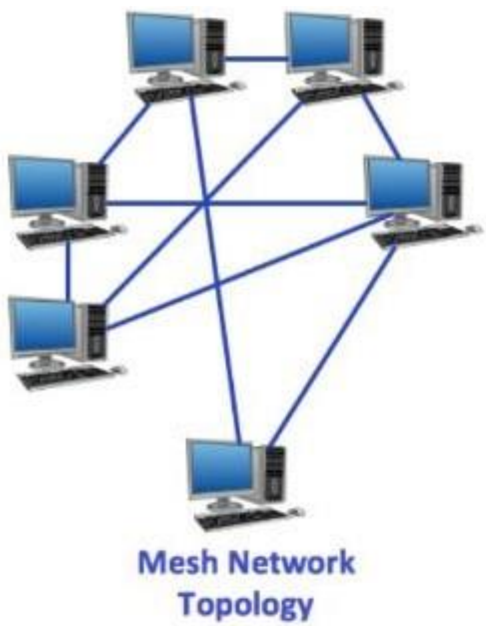
```
Bus received: Message from Device3
Bus received: Message from Device1
Bus received: Message from Device1
Bus received: Message from Device4
Bus received: Message from Device1


=== Code Execution Successful ===
```

EXPERIMENT 7

Aim : Program to simulate mesh topology

Theory :

Creating a simulation of a mesh network topology involves simulating nodes (computers/devices) connected in a decentralized, interconnected manner. Each node in a mesh network typically connects directly to several other nodes, forming a web-like structure where multiple paths between any two nodes can exist.



Here's a basic Python program to simulate a mesh network topology:

```
import networkx as nx
import matplotlib.pyplot as plt
import random

# Parameters for the mesh network
num_nodes = 10 # Number of nodes in the mesh network
avg_degree = 3  # Average degree (number of connections per node)

# Create a mesh network graph
G = nx.random_regular_graph(avg_degree, num_nodes)
```

```
# Assign random positions to nodes for visualization
pos = nx.spring_layout(G)

# Draw the mesh network
plt.figure(figsize=(8, 8))
nx.draw(G, pos, with_labels=True, node_size=500, node_color='skyblue', font_size=12,
font_weight='bold', edge_color='gray')
plt.title('Mesh Network Topology')
plt.show()

# Print some information about the network
print("Number of nodes:", G.number_of_nodes())
print("Number of edges:", G.number_of_edges())

# Example: Find shortest path between two random nodes
if G.number_of_nodes() > 1:
    node1 = random.choice(list(G.nodes()))
    node2 = random.choice(list(G.nodes()))
    shortest_path = nx.shortest_path(G, source=node1, target=node2)
    print(f"Shortest path between node {node1} and node {node2}: {shortest_path}")
else:
    print("The network contains less than two nodes, cannot find a path.")
```

Explanation:

1. **Import Libraries**: We use networkx for graph operations and matplotlib for visualization.

2. **Parameters**:
   - num_nodes: Number of nodes in the mesh network.
   - avg_degree: Average degree of nodes (approximately, how many nodes each node is connected to).

3. **Create Mesh Network Graph**:
   - nx.random_regular_graph(avg_degree, num_nodes): Generates a random graph where each node has the same number of edges (degree).

4. **Visualization**:
   - pos = nx.spring_layout(G): Positions nodes using a spring layout for better visualization.
   - nx.draw(...): Draws the mesh network graph using matplotlib.

5. **Network Information**:
   - Prints the number of nodes and edges in the network.

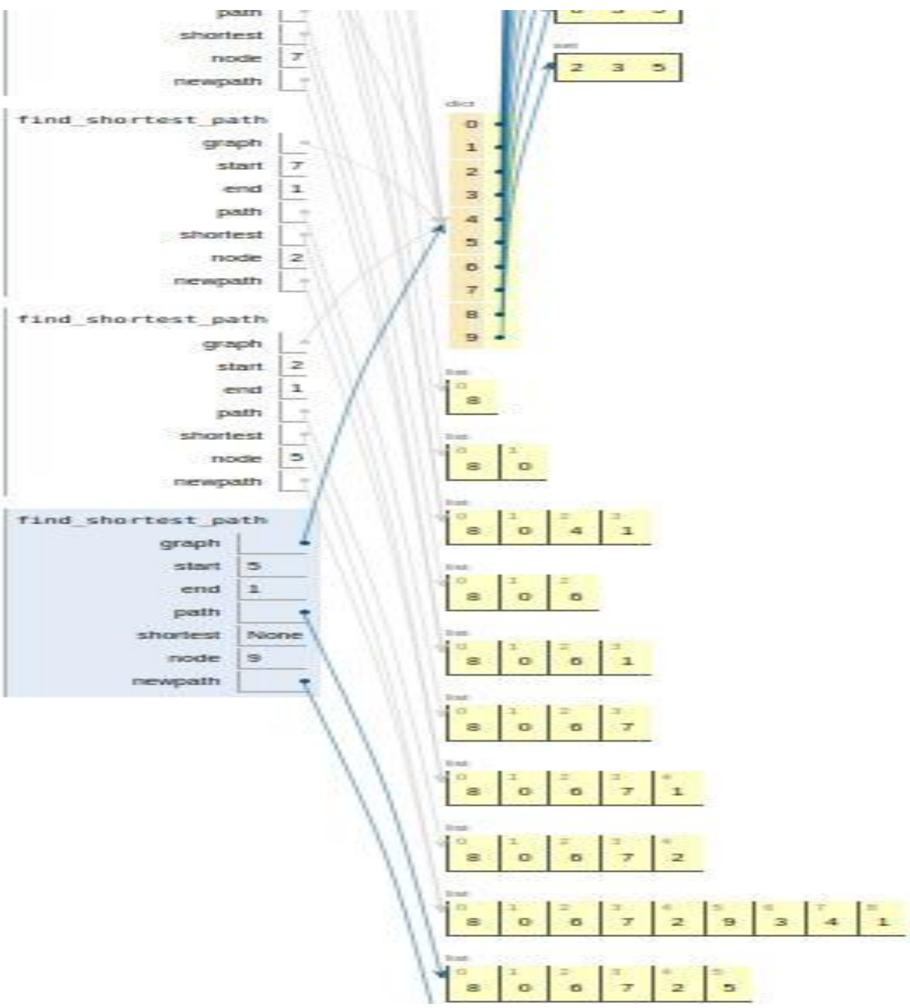
6. **Example Path Calculation**:
    - Chooses two random nodes and finds the shortest path between them using nx.shortest_path().

Running the Program:

- Ensure you have networkx and matplotlib installed (pip install networkx matplotlib).
- Execute the script in a Python environment (python mesh_topology_simulation.py).
-

Output:

```
Mesh Network Topology:
0 -- [4, 6, 8]
1 -- [4, 6, 7]
2 -- [5, 7, 9]
3 -- [4, 8, 9]
4 -- [0, 1, 3]
5 -- [2, 8, 9]
6 -- [0, 1, 7]
7 -- [1, 2, 6]
8 -- [0, 3, 5]
9 -- [2, 3, 5]
```



EXPERIMENT 8

Aim : Program to simulate UDP server client

Theory :

To simulate a UDP server-client interaction in Python, we need to create both the server and client scripts. UDP (User Datagram Protocol) is a connectionless protocol, meaning it doesn't establish a connection before sending data. The server listens for incoming messages, and the client sends messages to the server.

UDP Server

This server will listen for incoming messages on a specific port and print them to the console.

```
import socket

def udp_server(host='127.0.0.1', port=12345):
    # Create a UDP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Bind the socket to the address and port
    server_address = (host, port)
    print(f'Starting UDP server on {host}:{port}')
    sock.bind(server_address)

    while True:
        print('\nWaiting to receive message...')
        data, address = sock.recvfrom(4096)  # Buffer size is 4096 bytes

        print(f'Received {len(data)} bytes from {address}')
        print(data.decode())

        if data:
            sent = sock.sendto(data, address)
            print(f'Sent {sent} bytes back to {address}')

if__name__== '_main_':
    udp_server()
```

UDP Client

This client will send a message to the server and wait for a response.

```
import socket

def udp_client(host='127.0.0.1', port=12345, message='Hello, UDP server!'):
    # Create a UDP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    server_address = (host, port)
    try:
        # Send data
        print(f'Sending: {message}')
        sent = sock.sendto(message.encode(), server_address)

        # Receive response
        print('Waiting for a response...')
        data, server = sock.recvfrom(4096)
        print(f'Received {data.decode()} from {server}')

    finally:
        print('Closing socket')
        sock.close()

if__name__== '__main__':
    udp_client()
```

Explanation

1. **UDP Server**:
    - Creates a socket with socket.socket(socket.AF_INET, socket.SOCK_DGRAM).
    - Binds the socket to the provided host and port.
    - Enters an infinite loop to receive messages using recvfrom.
    - Prints the received message and sends it back to the client.

2. **UDP Client**:
    - Creates a socket with socket.socket(socket.AF_INET, socket.SOCK_DGRAM).
    - Sends a message to the server using sendto.
    - Waits for a response from the server using recvfrom.
    - Prints the received message and closes the socket.

Running the Scripts

1. **Run the server script**: Execute udp_server.py to start the server.
2. **Run the client script**: Execute udp_client.py to send a message to the server.

This will simulate a basic UDP communication between a client and a server.

Output:

```
Starting UDP server simulator

Waiting to receive message...
Enter a message to simulate receiving (or 'exit' to stop): hii
Received 3 bytes from ('127.0.0.1', 12345)
hii
Sent 3 bytes back to ('127.0.0.1', 12345)

Waiting to receive message...
Enter a message to simulate receiving (or 'exit' to stop): hello
Received 5 bytes from ('127.0.0.1', 12345)
hello
Sent 5 bytes back to ('127.0.0.1', 12345)

Waiting to receive message...
Enter a message to simulate receiving (or 'exit' to stop): all working
Received 11 bytes from ('127.0.0.1', 12345)
all working
Sent 11 bytes back to ('127.0.0.1', 12345)

Waiting to receive message...
Enter a message to simulate receiving (or 'exit' to stop):
```

EXPERIMENT 9

Aim : Animate a simple network using NetAnim in Network Simulator

Theory :

Animating a network simulation using NetAnim in Network Simulator (ns-3) involves several steps to set up and visualize the network scenario. Here's a basic guide to get you started:

Prerequisites:

1. **Install ns-3**: Make sure you have ns-3 installed on your system. NetAnim is a visualization tool that comes bundled with ns-3.

2. **Create a Network Scenario**: You need to write an ns-3 script (here, Python) that defines your network topology, nodes, and traffic.

3. **Install NetAnim**: Ensure NetAnim is installed and available in your ns-3 setup.

Steps to Animate a Simple Network

Step 1: Write an ns-3 Script

Create a Python script (simple_network.py) to set up a simple network topology and run the simulation.

```
import ns.core
import ns.network
import ns.internet
import ns.point_to_point

# Setup the simulation environment sim
= ns.core.NodeContainer()
sim.Create(2)

pointToPoint = ns.point_to_point.PointToPointHelper()
pointToPoint.SetDeviceAttribute("DataRate", ns.core.StringValue("5Mbps"))
pointToPoint.SetChannelAttribute("Delay", ns.core.StringValue("2ms"))

devices = pointToPoint.Install(sim)

stack = ns.internet.InternetStackHelper() stack.Install(sim)
```

address = ns.internet.Ipv4AddressHelper()
address.SetBase(ns.network.Ipv4Address("10.1.1.0"), ns.network.Ipv4Mask("255.255.255.0"))

interfaces = address.Assign(devices) #

Generate a trace file for animation
ns.core.Simulator.Stop(ns.core.Seconds(1.0))
ns.core.Simulator.Run()
ns.core.Simulator.Destroy()

# Write the trace file
ns.network.AnimationInterface("simple_network.xml")

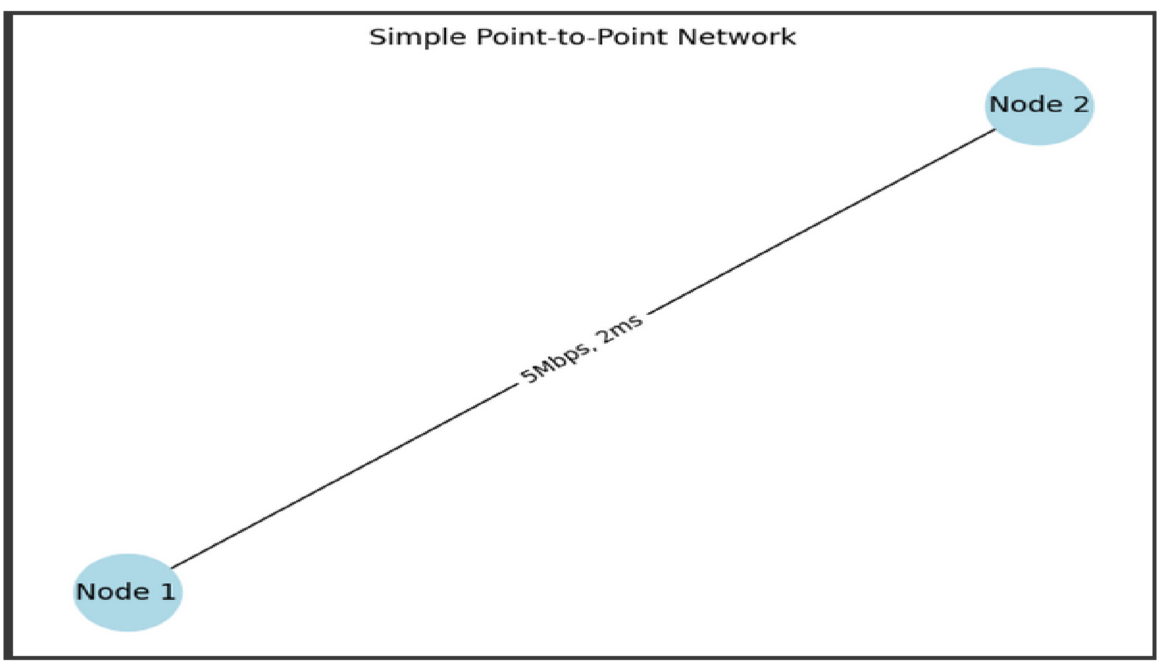
Step 2 : Generate trace files

When you run this script (python simple_network.py), ns-3 will generate a trace file named simple_network.xml.

Step 3 : Use NetAnim to animate

Launch NetAnim and open the simple_network.xml file to visualize the network animation.

Output:



EXPERIMENT 10

Aim : Analyze the network traffic using WireShark

Theory :

Wireshark is a powerful tool used by network administrators, security analysts, and developers alike to diagnose network problems, analyze network protocols, and investigate security incidents.

Analyzing network traffic using WireShark (now Wireshark) involves several steps to understand what data is being transmitted over your network.

Here’s a basic guide to get you started:

Step 1: Capture Traffic

1. **Start Wireshark**: Open Wireshark on your computer. It may prompt you to choose a network interface to start capturing traffic on.

2. **Select Interface**: Choose the network interface that corresponds to your network connection (Ethernet, Wi-Fi, etc.).

3. **Begin Capture**: Click on the interface and then click the green "Start" button to begin capturing packets.

Step 2: Analyze Captured Traffic

Once you've captured some packets:

1. **Packet List**: The main window in Wireshark shows a list of captured packets. Each row represents a single packet with information such as source and destination addresses, protocol used, and more.

2. **Packet Details**: Clicking on a packet in the list will show detailed information in the lower part of the Wireshark window. This includes protocol-specific details, packet headers, and sometimes even packet contents if they are not encrypted.

Step 3: Filter and Focus

To make sense of the data:

1. **Apply Filters**: Use Wireshark's filtering capabilities to focus on specific types of traffic (e.g., HTTP, DNS). Filters can be applied using the filter bar at the top of the Wireshark window.

   - Example filters:
     - http: Filters HTTP traffic.
     - ip.addr == 192.168.1.1: Filters traffic to or from a specific IP address.
     - tcp.port == 80: Filters traffic on a specific TCP port (e.g., port 80 for HTTP).
     -

2. **Follow Streams**: Wireshark allows you to follow a TCP or UDP stream to see all packets related to a particular conversation between two endpoints. This can provide a more coherent view of data exchanges.

Step 4: Interpret Results

To analyze the traffic effectively:

1. **Protocol Analysis**: Understand which protocols are in use (TCP, UDP, HTTP, etc.) and their role in the network communication.

2. **Traffic Patterns**: Look for patterns in the traffic, such as large data transfers, frequent connections to specific servers, or unusual protocols.

3. **Analyze Headers**: Check packet headers for information like source and destination addresses, ports, sequence numbers (for TCP), and flags.

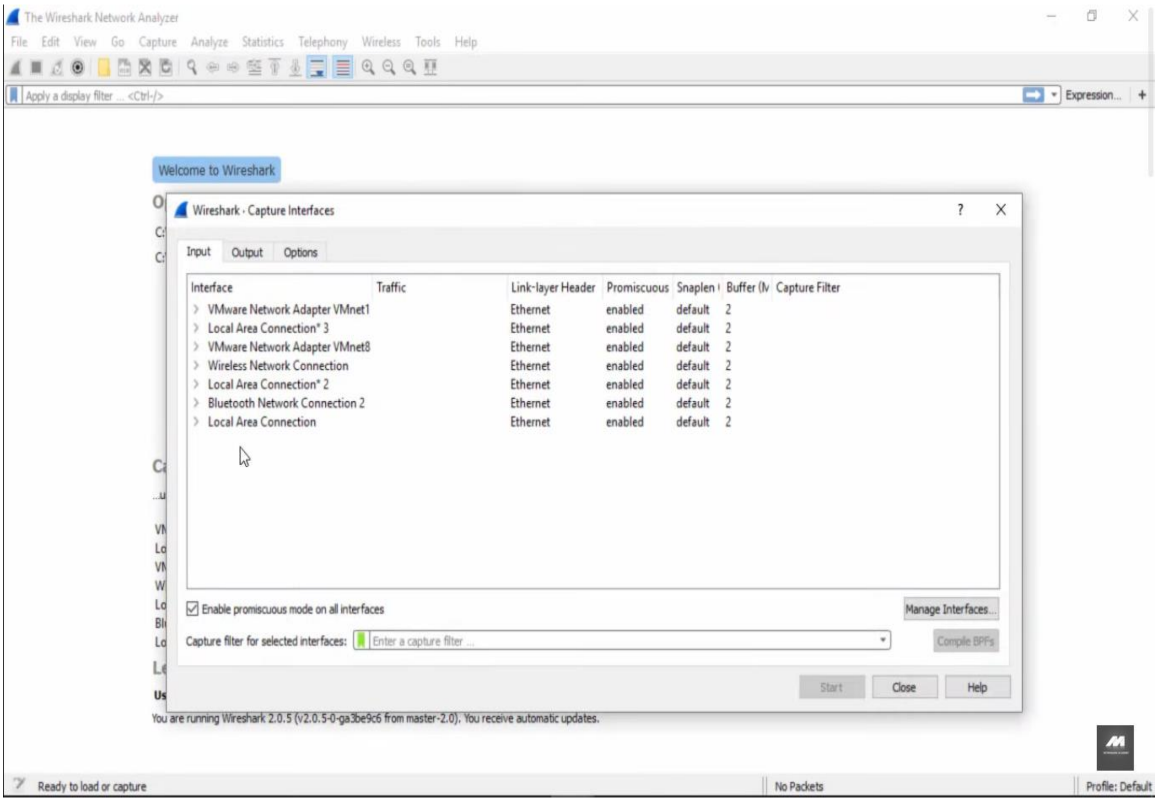
4. **Detect Issues**: Identify any anomalies or suspicious activities (e.g., unexpected traffic, high bandwidth usage).

Step 5: Save and Share

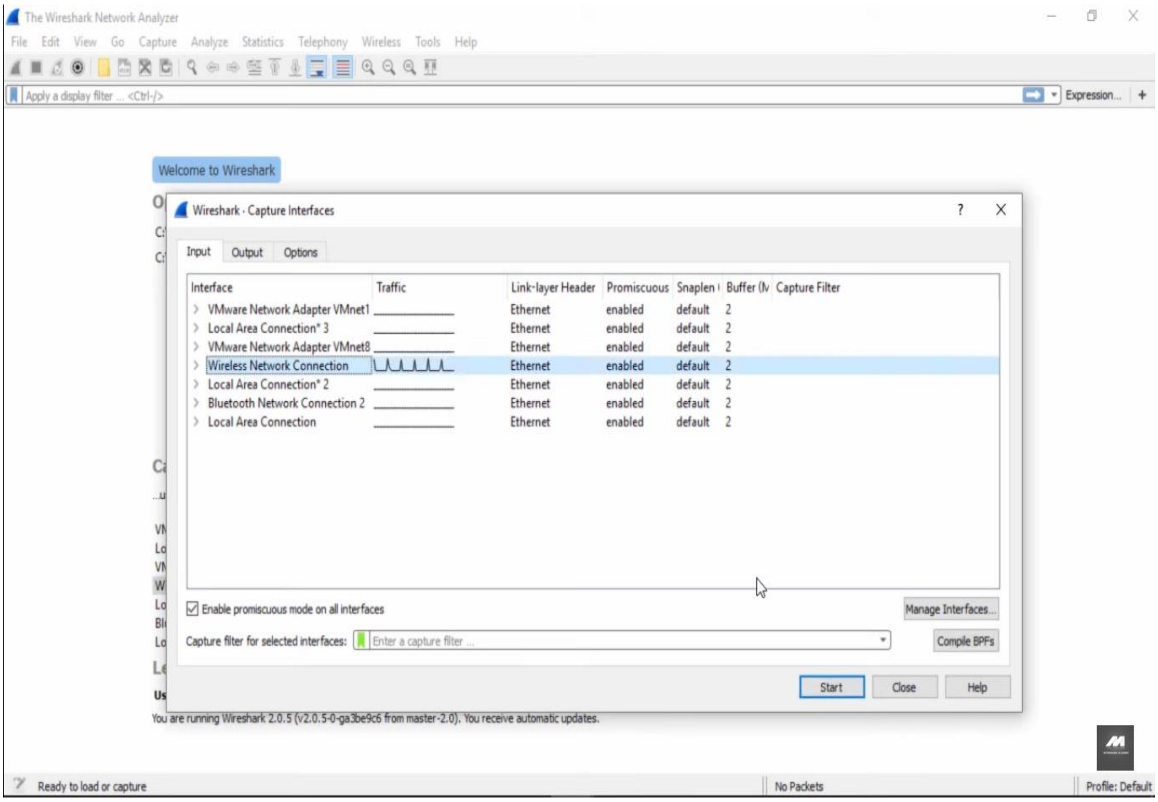
1. **Save Captures**: Save your capture for further analysis or for sharing with others. Wireshark allows you to save captures in its native .pcap format or export in other formats like .csv.

2. **Share Results**: If needed, share your findings with colleagues or support teams for troubleshooting or security analysis.

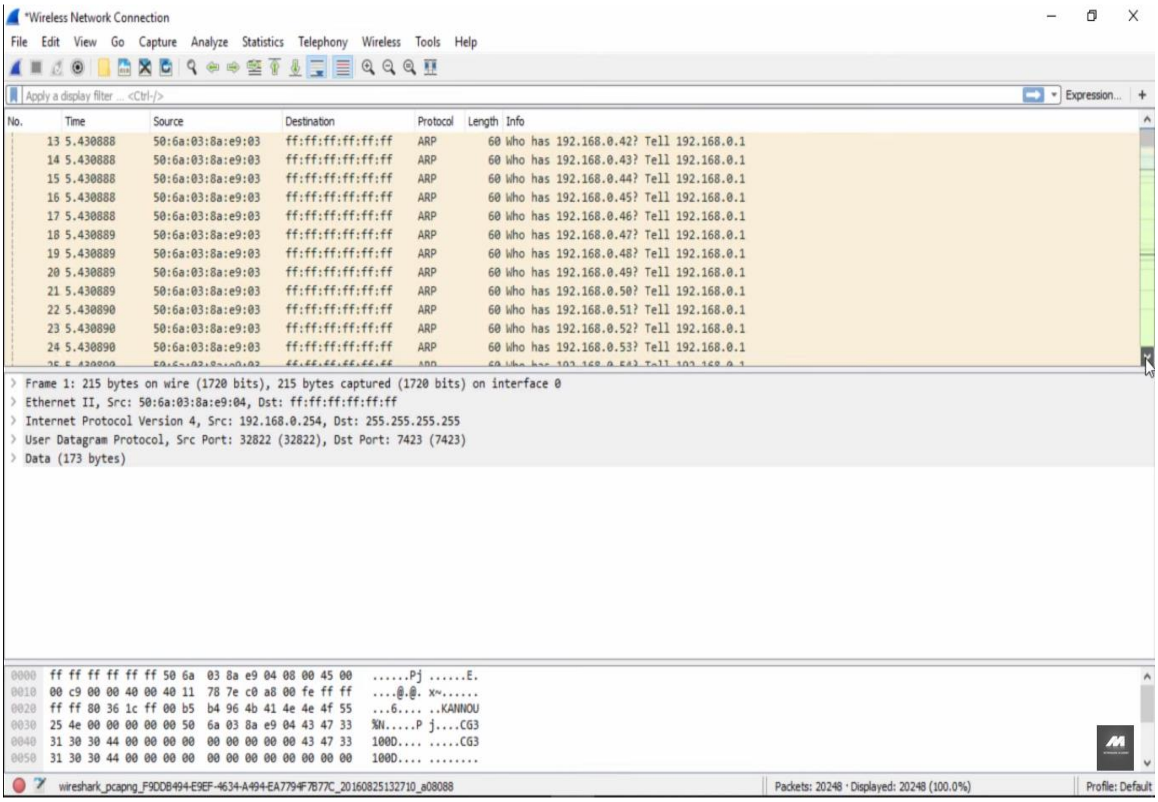
Output:

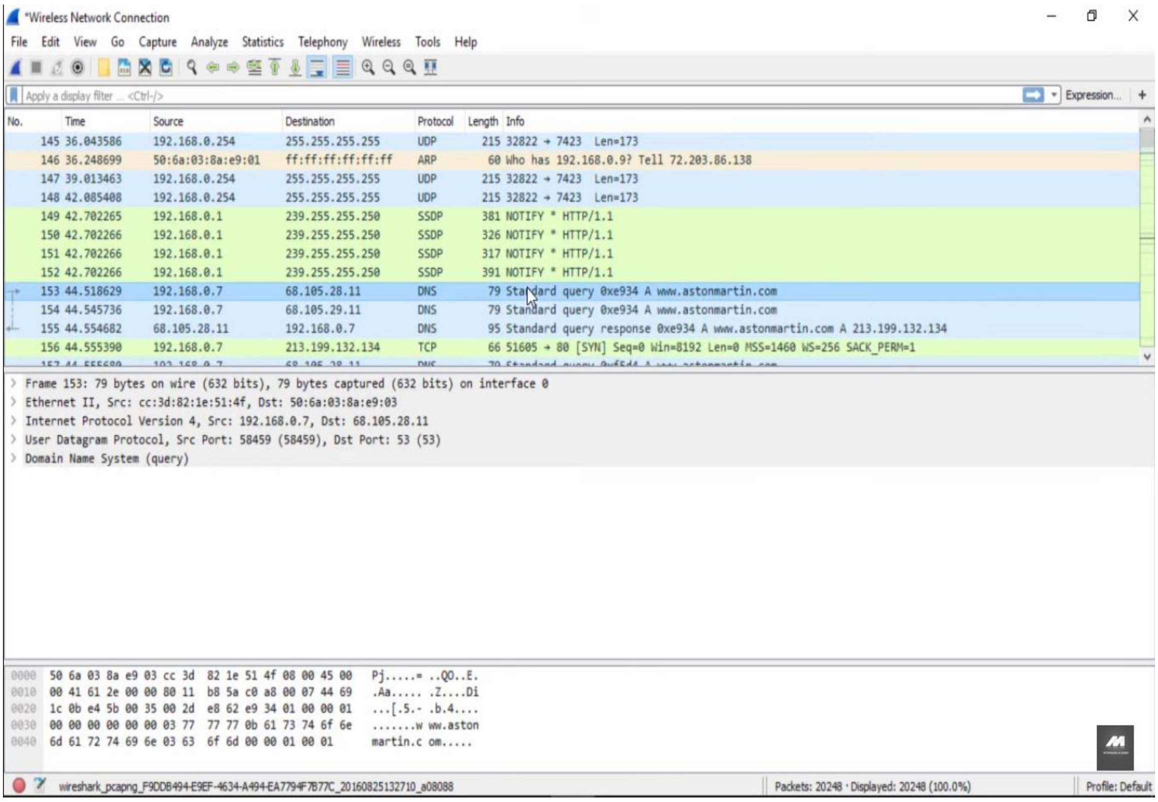


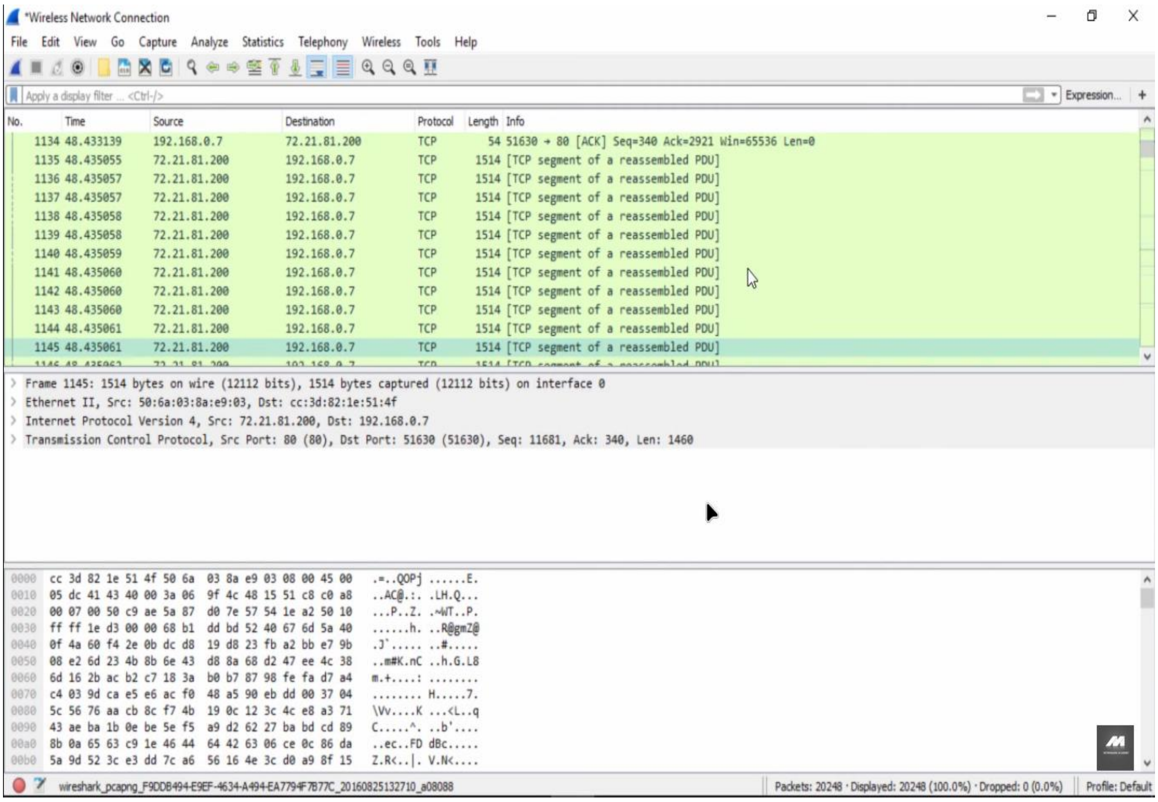
The Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
Welcome to Wireshark
Wireshark · Capture Interfaces
Input Output Options
Interface Traffic Link-layer Header Promiscuous Snaplen Buffer Capture Filter
VMware Network Adapter VMnet1 Ethernet enabled default 2
Local Area Connection* 3 Ethernet enabled default 2
VMware Network Adapter VMnet8 Ethernet enabled default 2
Wireless Network Connection Ethernet enabled default 2
Local Area Connection* 2 Ethernet enabled default 2
Bluetooth Network Connection 2 Ethernet enabled default 2
Local Area Connection Ethernet enabled default 2
Enable promiscuous mode on all interfaces
Manage Interfaces...
Capture filter for selected interfaces:
Enter a capture filter ...
Compile BPFs
Start Close Help
You are running Wireshark 2.0.5 (v2.0.5-0-ga3be9c6 from master-2.0). You receive automatic updates.
Ready to load or capture
No Packets
Profile: Default



The Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
Welcome to Wireshark
Wireshark · Capture Interfaces
Input Output Options
Interface Traffic Link-layer Header Promiscuous Snaplen Buffer Capture Filter
VMware Network Adapter VMnet1 Ethernet enabled default 2
Local Area Connection* 3 Ethernet enabled default 2
VMware Network Adapter VMnet8 Ethernet enabled default 2
Wireless Network Connection Ethernet enabled default 2
Local Area Connection* 2 Ethernet enabled default 2
Bluetooth Network Connection 2 Ethernet enabled default 2
Local Area Connection Ethernet enabled default 2
Enable promiscuous mode on all interfaces
Manage Interfaces...
Capture filter for selected interfaces:
Enter a capture filter ...
Compile BPFs
Start Close Help
You are running Wireshark 2.0.5 (v2.0.5-0-ga3be9c6 from master-2.0). You receive automatic updates.
Ready to load or capture
No Packets
Profile: Default


*Wireless Network Connection
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
No. Time Source Destination Protocol Length Info
13 5.430888 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.42? Tell 192.168.0.1
14 5.430888 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.43? Tell 192.168.0.1
15 5.430888 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.44? Tell 192.168.0.1
16 5.430888 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.45? Tell 192.168.0.1
17 5.430888 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.46? Tell 192.168.0.1
18 5.430889 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.47? Tell 192.168.0.1
19 5.430889 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.48? Tell 192.168.0.1
20 5.430889 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.49? Tell 192.168.0.1
21 5.430889 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.50? Tell 192.168.0.1
22 5.430890 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.51? Tell 192.168.0.1
23 5.430890 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.52? Tell 192.168.0.1
24 5.430890 50:6a:03:8a:e9:03 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.53? Tell 192.168.0.1
Frame 1: 215 bytes on wire (1720 bits), 215 bytes captured (1720 bits) on interface 0
Ethernet II, Src: 50:6a:03:8a:e9:04, Dst: ff:ff:ff:ff:ff:ff
Internet Protocol Version 4, Src: 192.168.0.254, Dst: 255.255.255.255
User Datagram Protocol, Src Port: 32822 (32822), Dst Port: 7423 (7423)
Data (173 bytes)
0000 ff ff ff ff ff ff 50 6a 03 8a e9 04 08 00 45 00 ......Pj ......E.
0010 00 c9 00 00 40 00 40 11 78 7e c0 a8 00 fe ff ff ....@.@. x~......
0020 ff ff 80 36 1c ff 00 b5 b4 96 4b 41 4e 4e 4f 55 ...6.... ..KANNOU
0030 25 4e 00 00 00 00 00 50 6a 03 8a e9 04 43 47 33 %N.....P j....CG3
0040 31 30 30 44 00 00 00 00 00 00 00 00 00 43 47 33 100D.... .....CG3
0050 31 30 30 44 00 00 00 00 00 00 00 00 00 00 00 00 100D.... ....
wireshark_pcapng_F9DDB494-E9EF-4634-A494-EA7794F7B77C_20160825132710_a08088
Packets: 20248 · Displayed: 20248 (100.0%)
Profile: Default


*Wireless Network Connection
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
No. Time Source Destination Protocol Length Info
145 36.043586 192.168.0.254 255.255.255.255 UDP 215 32822 → 7423 Len=173
146 36.248699 50:6a:03:8a:e9:01 ff:ff:ff:ff:ff:ff ARP 60 Who has 192.168.0.9? Tell 72.203.86.138
147 39.013463 192.168.0.254 255.255.255.255 UDP 215 32822 → 7423 Len=173
148 42.085408 192.168.0.254 255.255.255.255 UDP 215 32822 → 7423 Len=173
149 42.702265 192.168.0.1 239.255.255.250 SSDP 381 NOTIFY * HTTP/1.1
150 42.702266 192.168.0.1 239.255.255.250 SSDP 326 NOTIFY * HTTP/1.1
151 42.702266 192.168.0.1 239.255.255.250 SSDP 317 NOTIFY * HTTP/1.1
152 42.702266 192.168.0.1 239.255.255.250 SSDP 391 NOTIFY * HTTP/1.1
153 44.518629 192.168.0.7 68.105.28.11 DNS 79 Standard query 0xe934 A www.astonmartin.com
154 44.545736 192.168.0.7 68.105.29.11 DNS 79 Standard query 0xe934 A www.astonmartin.com
155 44.554682 68.105.28.11 192.168.0.7 DNS 95 Standard query response 0xe934 A www.astonmartin.com A 213.199.132.134
156 44.555390 192.168.0.7 213.199.132.134 TCP 66 51605 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
Frame 153: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on interface 0
Ethernet II, Src: cc:3d:82:1e:51:4f, Dst: 50:6a:03:8a:e9:03
Internet Protocol Version 4, Src: 192.168.0.7, Dst: 68.105.28.11
User Datagram Protocol, Src Port: 58459 (58459), Dst Port: 53 (53)
Domain Name System (query)
0000 50 6a 03 8a e9 03 cc 3d 82 1e 51 4f 08 00 45 00 Pj.....= ..QO..E.
0010 00 41 61 2e 00 00 80 11 b8 5a c0 a8 00 07 44 69 .Aa..... .Z....Di
0020 1c 0b e4 5b 00 35 00 2d e8 62 e9 34 01 00 00 01 ...[.5.- .b.4....
0030 00 00 00 00 00 00 03 77 77 77 0b 61 73 74 6f 6e ....w ww.aston
0040 6d 61 72 74 69 6e 03 63 6f 6d 00 00 01 00 01 martin.c om.....
wireshark_pcapng_F9DDB494-E9EF-4634-A494-EA7794F7B77C_20160825132710_a08088
Packets: 20248 · Displayed: 20248 (100.0%)
Profile: Default


*Wireless Network Connection
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
No. Time Source Destination Protocol Length Info
1134 48.433139 192.168.0.7 72.21.81.200 TCP 54 51630 → 80 [ACK] Seq=340 Ack=2921 Win=65536 Len=0
1135 48.435055 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1136 48.435057 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1137 48.435057 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1138 48.435058 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1139 48.435058 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1140 48.435059 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1141 48.435060 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1142 48.435060 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1143 48.435060 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1144 48.435061 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
1145 48.435061 72.21.81.200 192.168.0.7 TCP 1514 [TCP segment of a reassembled PDU]
Frame 1145: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
Ethernet II, Src: 50:6a:03:8a:e9:03, Dst: cc:3d:82:1e:51:4f
Internet Protocol Version 4, Src: 72.21.81.200, Dst: 192.168.0.7
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51630 (51630), Seq: 11681, Ack: 340, Len: 1460
0000 cc 3d 82 1e 51 4f 50 6a 03 8a e9 03 08 00 45 00
0010 05 dc 41 43 40 00 3a 06 9f 4c 48 15 51 c8 c0 a8
0020 00 07 00 50 c9 ae 5a 87 d0 7e 57 54 1e a2 50 10
0030 ff ff 1e d3 00 00 68 b1 dd bd 52 40 67 6d 5a 40
0040 0f 4a 60 f4 2e 0b dc d8 19 d8 23 fb a2 bb e7 9b
0050 08 e2 6d 23 4b 8b 6e 43 d8 8a 68 d2 47 ee 4c 38
0060 6d 16 2b ac b2 c7 18 3a b0 b7 87 98 fe fa d7 a4
0070 c4 03 9d ca e5 e6 ac f0 48 a5 90 eb dd 00 37 04
0080 5c 56 76 aa cb 8c f7 4b 19 0c 12 3c 4c e8 a3 71
0090 43 ae ba 1b 0e be 5e f5 a9 d2 62 27 ba bd cd 89
00a0 8b 0a 65 63 c9 1e 46 44 64 42 63 06 ce 0c 86 da
00b0 5a 9d 52 3c e3 dd 7c a6 56 16 4e 3c d0 a9 8f 15
wireshark_pcapng_F9DDB494-E9EF-4634-A494-EA7794F7B77C_20160825132710_a08088
Packets: 20248 · Displayed: 20248 (100.0%) · Dropped: 0 (0.0%)
Profile: Default