**WEEK-8**

**Networking in Python**

**Description:**

This unit explores the concepts of **network programming using Python**, focusing on how to build client-server applications using the built-in socket module. Students will gain hands-on experience in creating both TCP and UDP-based communication systems.

Key topics include:

* Basics of sockets and IP communication
* Creating TCP and UDP clients and servers
* Sending and receiving messages over the network
* Handling multiple connections using threading
* Implementing real-time chat or file transfer functionality

This unit is essential for understanding how distributed systems communicate. It enables students to build real-world networked applications and lays the foundation for advanced concepts such as REST APIs and web socket communication.

**1.Write a Python program to create a simple TCP client and server for sending and receiving messages.**

**Description:**

This experiment introduces basic **TCP socket programming** using Python’s built-in socket module. It demonstrates how to establish a reliable two-way connection between a client and a server over the TCP/IP protocol.

In the server script, a socket is created and bound to a specific IP address and port. It listens for incoming client connections and once connected, it receives a message, prints it, and sends a response back. On the other side, the client creates a socket, connects to the server using the same IP and port, sends a message, and waits for a response.

This experiment helps students understand:

* The concept of sockets and ports
* The flow of data in a client-server model
* Reliable communication using TCP

These fundamentals are essential for building networked applications such as chat apps, file transfers, and real-time services.

**Program:**

 **Server.py**

import socket

# Create a TCP/IP socket

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# Bind the socket to a port

server\_socket.bind(('localhost', 12345))

server\_socket.listen(1)

print("Server is listening on port 12345...")

# Wait for a connection

conn, addr = server\_socket.accept()

print(f"Connected by {addr}")

# Receive message from client

data = conn.recv(1024).decode()

print("Client says:", data)

# Send a reply

response = "Hello Client, message received!"

conn.send(response.encode())

# Close the connection

conn.close()

**Output :**

 ****

**Client.py**

import socket

# Create a socket

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# Connect to the server

client\_socket.connect(('localhost', 12345))

# Send message to server

client\_socket.send("Hello Server!".encode())

# Receive response from server

response = client\_socket.recv(1024).decode()

print("Server says:", response)

# Close the connection

client\_socket.close()

**Output:**

 ****

**2.Write a Python program to implement a UDP client and server communication.**

 **Description:**

This experiment demonstrates how to use Python’s socket module to set up a UDP (User Datagram Protocol) based communication between a client and a server.

Unlike TCP, UDP is connectionless and does not guarantee message delivery, ordering, or duplicate protection. However, it is faster and useful in scenarios like live streaming, gaming, or IoT applications where speed is more critical than reliability.

In this experiment:

* The server waits to receive a message using recvfrom() and replies back using sendto().
* The client sends a message to the server using sendto() and waits for a response using recvfrom().

This gives students practical insight into how UDP differs from TCP and when it is preferred.

**Program: Server.py**

import socket

# Create a UDP socket

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

# Bind to address and port

server\_socket.bind(('localhost', 12345))

print("UDP Server is listening...")

# Receive data from client

data, addr = server\_socket.recvfrom(1024)

print("Received from client:", data.decode())

# Send response

response = "Message received via UDP!"

server\_socket.sendto(response.encode(), addr)

# Close the server socket

server\_socket.close()

**Output:**

****

**Program : Client.py**

import socket

# Create a UDP socket

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

# Send message to server

message = "Hello UDP Server!"

client\_socket.sendto(message.encode(), ('localhost', 12345))

# Receive response from server

data, server = client\_socket.recvfrom(1024)

print("Server says:", data.decode())

# Close client socket

client\_socket.close()

**Output:**

****

**3.Write a Python program to create a simple multi-client chat server using sockets and threading.**

 **Description:**

This experiment introduces how to build a multi-client chat server in Python using socket programming and the threading module. Unlike basic single-client programs, this server can handle multiple clients at once — a critical feature in modern networking applications like chat rooms, collaborative tools, or multiplayer games.

The server listens for client connections and spawns a new thread for each client to handle communication simultaneously. When one client sends a message, the server relays it to all connected clients, creating a simple group chat effect.

This helps students learn:

* Concurrent handling of clients using threading
* Broadcasting messages to multiple sockets
* Managing client connections in real time

**Program: Server.py**

import socket

import threading

def handle\_client(conn, addr):

 print(f"New connection from {addr}")

 conn.send("Welcome to the chat! Type 'bye' to exit.".encode())

 while True:

 try:

 data = conn.recv(1024).decode()

 if not data:

 break

 print(f"{addr} says: {data}")

 if data.lower() in ['bye', 'exit']:

 conn.send("Goodbye! Connection closed.".encode())

 break

 else:

 conn.send(f"Server received: {data}".encode())

 except:

 break

 print(f"Connection closed with {addr}")

 conn.close()

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server.bind(('localhost', 12347))

server.listen()

print("Multithreaded Server is running on port 12347...")

while True:

 conn, addr = server.accept()

 thread = threading.Thread(target=handle\_client, args=(conn, addr))

 thread.start()

**Output:**

****

**Program: client.py**

import socket

client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client.connect(('localhost', 12345))

welcome = client.recv(1024).decode()

print("Server:", welcome)

while True:

 msg = input("You: ")

 client.send(msg.encode())

 response = client.recv(1024).decode()

 print("Server:", response)

 if msg.lower() in ['bye', 'exit']:

 break

client.close()

**Output:**

****

**Viva Questions – Unit 4: Networking in Python**

1. **What is the difference between TCP and UDP?**TCP (Transmission Control Protocol) is connection-oriented, provides reliable data transfer, and ensures that packets are delivered in order. UDP (User Datagram Protocol) is connectionless, does not guarantee delivery, and is faster and more lightweight.
2. **What is the purpose of the socket module in Python?**The socket module provides low-level networking interfaces in Python. It is used to create server and client programs for network communication using protocols like TCP and UDP.
3. **What does the bind() method do in socket programming?**The bind() method is used by the server to associate the socket with a specific IP address and port number so that it can listen for incoming client connections.
4. **How can a server handle multiple clients at the same time?**A server can handle multiple clients simultaneously using multithreading or multiprocessing. Each client connection is handled in a separate thread or process.
5. **What is the role of the recv() and send() methods in sockets?**The recv() method is used to receive data from the socket, and the send() method is used to send data through the socket. These methods are fundamental to data exchange between client and server.