Name: Ahaan Desai
Roll no.: 231070016
Batch: A

Computer Network Lab
Experiment 10

Aim: Implementation of Socket Programming - TCP and UDP sockets

Theory

Socket programming is a way to enable communication between two or more devices over a network. It allows programs (called processes) running on different computers to exchange data using standard network protocols like TCP (Transmission Control Protocol) or UDP (User Datagram Protocol).

A socket is an endpoint of a two-way communication link between two programs running on a network. In simple terms, it acts as a “door” through which data can be sent or received.

Socket programming is mainly used for client-server communication, where:

- The server waits for client requests and responds to them.
- The client initiates communication with the server.

Types of Sockets

1. Stream Sockets (TCP Sockets)
    - Use TCP (Transmission Control Protocol).

- Provide reliable, connection-oriented communication.

- Data is transmitted in a continuous stream, ensuring that packets arrive in order and without loss.
- Examples: Web browsing (HTTP), email (SMTP), file transfer (FTP).

2. Datagram Sockets (UDP Sockets)
    - Use UDP (User Datagram Protocol).
    - Provide connectionless communication.
    - Data is sent as individual packets (datagrams) that may arrive out of order, be duplicated, or even lost.
    - Faster but less reliable than TCP.
    - Examples: Online gaming, video streaming, DNS queries.

Program

1) TCP Server and Client

a) Server

```python
import socket, threading

clients = []

def handle_client(conn):
    while True:
        try:
            msg = conn.recv(1024)
            if not msg:
                break

            for c in clients:
                if c != conn:
                    c.send(msg)

        except:
            break
```

```
    conn.close()
    clients.remove(conn)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', 9999))
s.listen()

while True:
    conn, addr = s.accept()
    clients.append(conn)
    threading.Thread(target=handle_client, args=(conn,)).start()
```

b) Client

Python

```
import socket, threading

def recv_msg(s):
    while True:
        try:
            print(s.recv(1024).decode())
        except:
            break

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('localhost', 9999))
threading.Thread(target=recv_msg, args=(s,), daemon=True).start()

while True:
    msg = input()
    s.send(msg.encode())
```

2) UDP Server and CLient

a) Server

Python

```python
import socket, threading

clients = []


def listen():
    while True:
        data, addr = s.recvfrom(1024)
        clients.append(addr)
        for c in clients:
            if c != addr:
                s.sendto(data, c)

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('0.0.0.0', 9999))
threading.Thread(target=listen).start()
```

b) Client

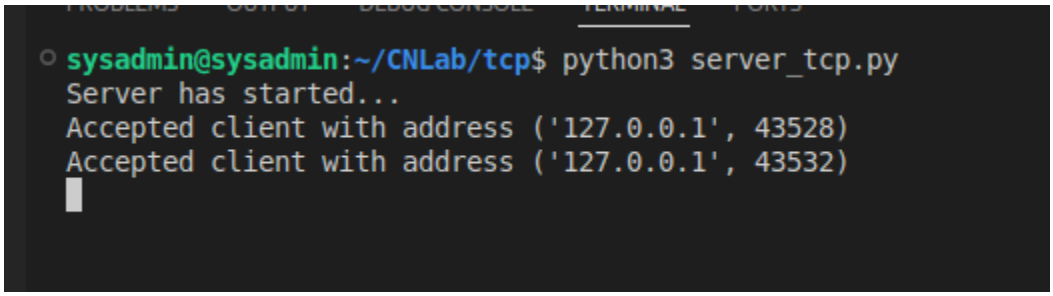
Python

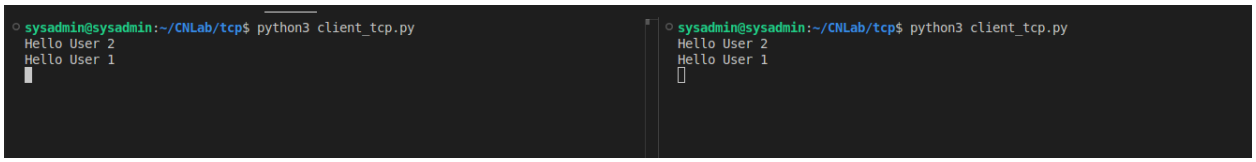
```python
import socket, threading

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 0))
server = ('localhost', 9999)

def recv_msg():
    while True:
        try:
            data, _ = s.recvfrom(1024)
            print(data.decode())
        except:
            break
```

```
threading.Thread(target=recv_msg, daemon=True).start()

while True:
    msg = input()
    s.sendto(msg.encode(), server)
```

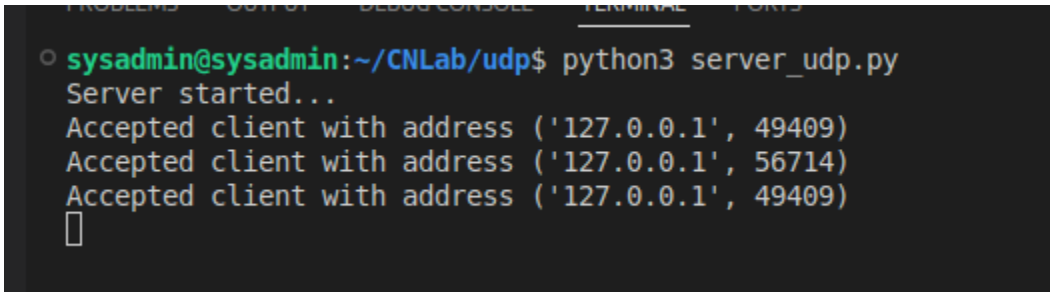
Output

1) TCP





2) UDP



Conclusion

This experiment demonstrates the implementation of TCP and UDP socket programming in Python for real-time communication. TCP ensures reliable, connection-oriented data transfer suitable for chat applications requiring message integrity, while UDP offers faster, connectionless communication ideal for broadcasting. By comparing both, we understand trade-offs between reliability and speed, and how sockets enable efficient, low-level network communication in distributed systems.