**AIM:**

To implement Client and Server using Twisted Python.

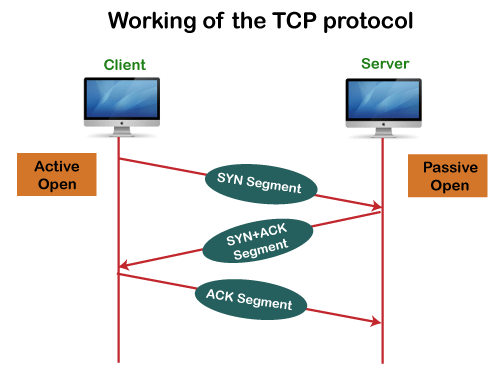
##### PROBLEM DESCRIPTION:

TCP stands for Transmission Control Protocol. It is a transport layer protocol that facilitates the transmission of packets from source to destination. It is a connection-oriented protocol that means it establishes the connection prior to the communication that occurs between the computing devices in a network. This protocol is used with an IP protocol.

##### Working:

**Server and Echo Client**

* Based on the IP Address and Port Number the connection is established between the Server and Client.
* Port Number should be same in Server and Client.
* Local Host should be specified in Client
* Client will have its data echoed back to server.



EXPLANATION:

* simple TCP server and client implemented in Python using the socket module.
* The server script establishes a TCP socket, binds it to a specified IP address and port, and listens for incoming connections.
* Once a connection is established, the server receives messages from the client, acknowledges receipt, and prints the messages along with the client's address. The connection is closed when the client finishes sending messages.
* On the client side, a TCP socket is created and connected to the server's IP address and port. The client sends a "Hello, TCP Server!" message to the server, waits for a response, and prints the received acknowledgment. Finally, the client closes the connection.

Example Output Explanation:

Server Output:

* "TCP Server listening on 127.0.0.1:5006": Indicates the server is actively listening on the specified IP address and port.
* "TCP Connection address: <client\_address>": Confirms the establishment of a connection with a client.
* "TCP Received message: <client\_message> from <client\_address>": Displays the message received from the connected client.
* "Connection from <client\_address> closed.": Indicates that the client has closed the connection.

Client Output:

* "TCP Received response: TCP Server: Message received": Shows the acknowledgment received from the server in response to the client's message.These messages illustrate the communication flow between server and client

CODE:

SERVER SIDE

import socket

TCP\_IP = "127.0.0.1"

TCP\_PORT = 5006

def simple\_tcp\_server():

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

    tcp\_socket.bind((TCP\_IP, TCP\_PORT))

    tcp\_socket.listen(1)  # Listen for one incoming connection

    print(f"TCP Server listening on {TCP\_IP}:{TCP\_PORT}")

    while True:

        conn, addr = tcp\_socket.accept()

        print(f"TCP Connection address: {addr}")

        while True:

            data = conn.recv(1024)

            if not data:

                break

            print(f"TCP Received message: {data.decode()} from {addr}")

            conn.send("TCP Server: Message received".encode())

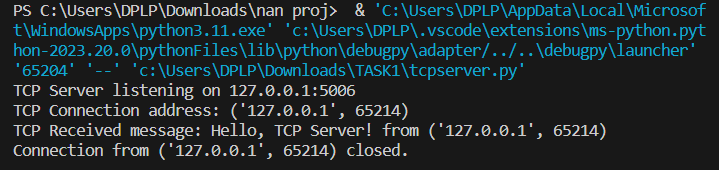
        print(f"Connection from {addr} closed.")

        conn.close()

if \_\_name\_\_ == "\_\_main\_\_":

    simple\_tcp\_server()

OUTPUT:



CLIENT SIDE:

import socket

TCP\_IP = "127.0.0.1"

TCP\_PORT = 5006

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

tcp\_socket.connect((TCP\_IP, TCP\_PORT))

message = "Hello, TCP Server!"

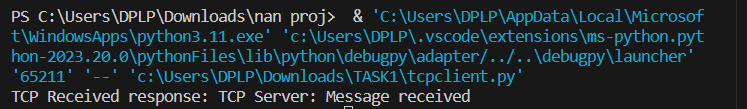
tcp\_socket.send(message.encode())

data = tcp\_socket.recv(1024)

print(f"TCP Received response: {data.decode()}")

tcp\_socket.close()

OUTPUT:



* UDP (User Datagram Protocol) server and client implemented in Python using the socket module.
* On the server side, a UDP socket is created and bound to a specified IP address and port. The server continuously listens for incoming datagrams. When a datagram is received, it prints the message and the address of the sender, and then sends a response back to the client acknowledging the receipt of the message.
* On the client side, a UDP socket is created, and a message ("Hello, UDP Server!") is sent to the server's IP address and port. The client then waits to receive a response from the server, prints the received acknowledgment along with the server's address, and finally closes the UDP socket.
* UDP server and client operate on a connectionless communication model, where messages (datagrams) are sent without establishing a persistent connection. The server responds to each received datagram individually.

UDP Server Output:

* "UDP Server listening on 127.0.0.1:5005": Indicates that the UDP server is actively listening on the specified IP address and port.
* "UDP Received message: Hello, UDP Server! from <client\_address>": Displays the message received from the client along with the client's address.
* "UDP Server: Message received": Confirms the receipt of the message and indicates that the server has processed the client's datagram.

UDP Client Output:

* message = "Hello, UDP Server!": The client prepares a message to be sent to the server udp\_socket.sendto(message.encode(), (UDP\_IP, UDP\_PORT).
* The client sends the message as a UDP datagram to the server's IP address and port.

data, addr = udp\_socket.recvfrom(1024): The client waits to receive a response from the server.

print(f"UDP Received response: {data.decode()} from {addr}"): .

The client prints the acknowledgment received from the server along with the server's address.

UDP SERVER SIDE:

import socket

UDP\_IP = "127.0.0.1"

UDP\_PORT = 5005

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

udp\_socket.bind((UDP\_IP, UDP\_PORT))

print(f"UDP Server listening on {UDP\_IP}:{UDP\_PORT}")

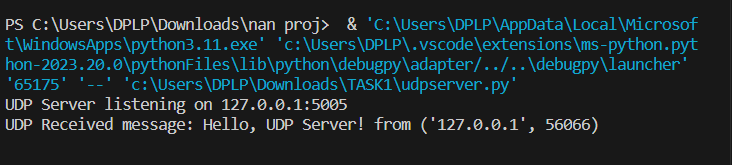
while True:

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

    print(f"UDP Received message: {data.decode()} from {addr}")

    udp\_socket.sendto("UDP Server: Message received".encode(), addr)

OUTPUT:



UDP CLIENT SIDE:

import socket

UDP\_IP = "127.0.0.1"

UDP\_PORT = 5005

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

try:

    message = "Hello, UDP Server!"

    udp\_socket.sendto(message.encode(), (UDP\_IP, UDP\_PORT))

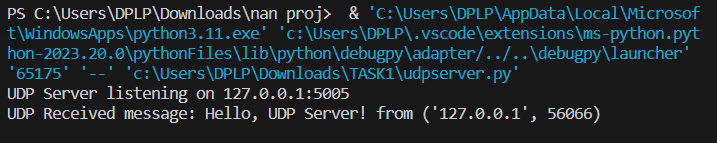
    data, addr = udp\_socket.recvfrom(1024)

    print(f"UDP Received response: {data.decode()} from {addr}")

finally:

    udp\_socket.close()

OUTPUT:



Server Side:

* The server-side code implements a ZeroMQ REP (Reply) socket using the "tcp" transport protocol on port 5000.
* The server continuously listens for incoming messages. When a message is received, it prints the content, processes it (e.g., generates an acknowledgment), and sends the response back to the client.
* The server operates in a loop, providing a persistent service to multiple client requests. It ensures proper resource cleanup using the try-finally block to close the socket and terminate the ZeroMQ context.

Client Side:

* The client-side code establishes a ZeroMQ REQ (Request) socket and connects to the server's address using the "tcp" transport protocol on port 5000.
* The client sends a message ("Hello, ZeroMQ Server!") to the server, prints the sent message, and waits for a response. Upon receiving the server's acknowledgment, the client prints the response.
* The try-finally block ensures the proper closure of the socket and termination of the ZeroMQ context, promoting good resource management. The client-server communication follows a request-reply pattern, with the client initiating the conversation and the server responding to each request.

ZEROMQ SERVER SIDE:

import zmq

def zmq\_server():

    context = zmq.Context()

    socket = context.socket(zmq.REP)

    socket.bind("tcp://\*:5555")

    print("ZeroMQ Server listening on tcp://\*:5555")

    while True:

        message = socket.recv\_string()

        print(f"ZeroMQ Received message: {message}")

        # Process the received message here (acknowledge, reply, etc.)

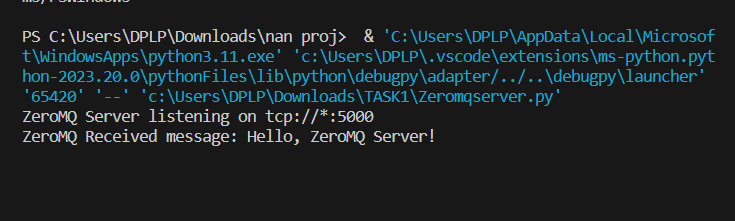
        response = f"ZeroMQ Server: Acknowledgment for {message}"

        socket.send\_string(response)

if \_\_name\_\_ == "\_\_main\_\_":

    zmq\_server()

OUTPUT:



ZEROMQ CLIENT SIDE:

import zmq

def zmq\_client():

    context = zmq.Context()

    socket = context.socket(zmq.REQ)

    socket.connect("tcp://localhost:5555")

    message = "Hello, ZeroMQ Server!"

    print(f"Sending message: {message}")

    socket.send\_string(message)

    response = socket.recv\_string()

    print(f"Received response: {response}")

if \_\_name\_\_ == "\_\_main\_\_":

    zmq\_client()

OUTPUT:

