Lab Manual 1: Retrieving Local IP Address & Lab Manual 2: Multiple Client-Server Communication

Objective:

This combined lab aims to:

1. Retrieve the correct local IP address of a machine using Python's socket module.
2. Implement a server that can handle multiple client connections simultaneously using threading in Python.

Requirements:

- Python 3.x
- Basic understanding of networking concepts
- Basic knowledge of socket programming
- Text editor (e.g., VSCode, PyCharm)

Lab 1: Retrieving Local IP Address

Theory:

To identify the machine's correct local IP address, we create a Python program that uses the socket module to create a UDP socket and force it to connect to an external server (Google's DNS server, 8.8.8.8). Although no actual data is exchanged, this operation allows the machine to expose its valid local IP address.

Program:

```
import socket

# Function to retrieve the correct local IP address of the machine
def get_local_ip():
    # Create a UDP socket
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
        try:
            # Use a dummy address to force the socket to use the correct interface
            s.connect(('8.8.8.8', 80))  # Google's public DNS server
            ip = s.getsockname()[0]     # Get the local IP address
        except Exception:
            ip = '127.0.0.1'  # Default to localhost in case of error
    return ip
```

```
# Get and print the correct local IP address
host_ip = get_local_ip()
print(f"Correct IP Address: {host_ip}")
```

Explanation:

1. **Socket Creation:**
   The `socket.socket(socket.AF_INET, socket.SOCK_DGRAM)` creates a UDP socket for communication. UDP is used because it is lightweight and doesn't require a connection.
2. **Connecting to DNS Server:**
   The socket attempts to connect to Google's public DNS server (8.8.8.8), which allows the system to reveal the active network interface and retrieve the local IP address.
3. **Retrieving the IP Address:**
   After the "connection" is made, `s.getsockname()[0]` retrieves the machine's local IP address.
4. **Error Handling:**
   If there’s no internet connection or any error, the code defaults to 127.0.0.1 (localhost).

Lab 2: Multiple Client-Server Communication

Theory:

In a real-world environment, servers often need to handle multiple clients at the same time. This can be achieved using **multithreading**, where each client connection is managed by a separate thread, allowing the server to handle concurrent clients without blocking.

In this lab, we will:

- Implement a server that can accept multiple clients simultaneously.
- Implement a client that connects to the server and exchanges messages with it.

Program Overview:

1. **Server Program:**
   - The server listens for incoming client connections on a specific IP address and port.
   - Each connection is handled in a new thread, allowing the server to serve multiple clients concurrently.

- Each client’s message is received and acknowledged by the server.

2. **Client Program:**
   - The client connects to the server using its IP address and port.
   - The client sends messages to the server and waits for the server's response.
   - The communication continues until the client sends the "exit" command.

Server Code:

```
import socket
import threading  # Import threading to enable handling multiple clients at the same time

# Function to handle individual client connections
def handle_client(client_socket, client_address):
    print(f"Connected to {client_address}")
    while True:
        try:
            data = client_socket.recv(1024)  # Receive data from the client
            if not data:
                print(f"Connection closed by {client_address}")
                break
            print(f"Received from {client_address}: {data.decode()}")
            client_socket.sendall(b"Server received: " + data)  # Acknowledge message
        except ConnectionResetError:
            print(f"Connection lost with {client_address}")
            break
    client_socket.close()  # Close the client socket

# Function to retrieve the local IP address of the machine running the server
def get_local_ip():
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
        try:
            s.connect(('8.8.8.8', 80))  # Connect to Google's DNS to get the correct interface IP
            ip = s.getsockname()[0]  # Get the local IP address
        except Exception:
            ip = '127.0.0.1'  # Fallback to localhost if something goes wrong
    return ip

# Main server function to start the server and handle client connections
def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Create a TCP socket
    host_ip = get_local_ip()  # Get the local IP address
    server_address = (host_ip, 65432)  # Server will listen on this IP and port
    server_socket.bind(server_address)  # Bind the socket to the IP and port
    server_socket.listen(5)  # Listen for incoming connections (up to 5 in the queue)
    print(f"Server is running on {host_ip}:{65432}, waiting for connections...")

    while True:
        client_socket, client_address = server_socket.accept()  # Accept a new client connection
        client_thread = threading.Thread(target=handle_client, args=(client_socket, client_address))  #
Create a new thread for the client
```

```
        client_thread.start()  # Start the thread to handle the client

start_server()  # Start the server
```

Client Code:

```
import socket

def start_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Create a TCP client socket
    server_address = ('127.0.0.1', 65432)  # Define server IP and port (replace with actual server IP for
remote testing)

    try:
        client_socket.connect(server_address)  # Connect to the server
        print(f"Connected to server at {server_address[0]}:{server_address[1]}")

        while True:
            message = input("You (Client): ")  # Get input from the user
            if message.lower() == 'exit':
                print("Closing connection...")
                break
            client_socket.sendall(message.encode())  # Send the message to the server
            data = client_socket.recv(1024)  # Receive response from the server
            print(f"Server: {data.decode()}")  # Print server's response

    except ConnectionRefusedError:
        print("Failed to connect to the server. Is the server running?")
    finally:
        client_socket.close()  # Close the socket when done

if __name__ == "__main__":
    start_client()  # Start the client
```

Step-by-Step Execution:

1. Run the Server:

- Save the server code to a file, e.g., server.py.
- Open a terminal and run the server script:

  ```
  bash
  Copy code
  python server.py
  ```

- The server will start and wait for incoming connections.

2. Run the Client:

- Save the client code to a file, e.g., client.py.

- Open another terminal (or multiple terminals for multiple clients) and run the client script:

  bash
  Copy code
  python client.py

- Connect the client to the server and start sending messages.

3. Test with Multiple Clients:

- Open multiple terminals, run client.py in each, and observe how the server handles communication from different clients simultaneously.

Key Concepts:

1. **Socket Programming:**
   - socket.socket() creates a socket for network communication.
   - bind() binds the socket to an IP and port.
   - listen() makes the server wait for incoming connections.
   - accept() accepts client connections.
   - recv() receives data from the client.
   - sendall() sends data to the client.
2. **Multithreading:**
   - threading.Thread() allows creating a new thread for each client.
   - This ensures that multiple clients can be served concurrently without blocking the server.
3. **Local IP Detection:**
   - The get_local_ip() function retrieves the correct local IP of the server by connecting to an external DNS, ensuring it works on different network interfaces.

Exercise:

Broadcast Messages:

Hint:

- Maintain a list or set of connected clients (sockets) on the server.
- When a message is received from one client, iterate through this list and send the message to all other clients.

Example Steps:

1. Create a list to store connected client sockets (e.g., clients = []).
2. When a new client connects, append its socket to this list.
3. In the handle_client function, after receiving a message, iterate through the clients list and send the message to each client except the one that sent it.

3. Client Identification:

Hint:

- Modify the client to prompt the user for a username upon connection.
- Send the username along with messages to the server so that the server can include it in responses.

Example Steps:

1. Ask the user for their username when the client connects.
2. Send the username as part of the initial message or as a separate message.
3. On the server side, modify the message handling to extract and acknowledge the username when sending responses.