**Lab 01:** Basic Socket Client-Server Communication

## Objective:

To implement and establish a basic client-server communication using Python sockets.

## Theory:

Socket programming allows two networked devices to communicate. A server listens for incoming connections, while a client connects to the server and exchanges data using TCP/IP protocols.

## Requirements:

- Python 3 installed

- Basic understanding of networking

## Code:

File Name: server.py

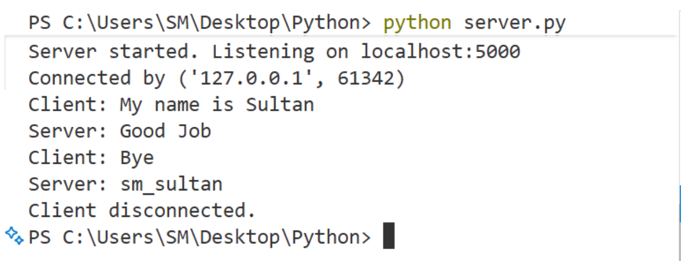
```
import socket
sultan_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
HOST = 'localhost'
PORT = 5000
sultan_server.bind((HOST, PORT))
sultan_server.listen(1)
print(f"Server started. Listening on {HOST}:{PORT}")
conn, addr = sultan_server.accept()
print(f"Connected by {addr}")
while True:
    data = conn.recv(1024).decode()
    if not data or data.lower() == 'sm_sultan':
        print("Client disconnected.")
        break
    print(f"Client: {data}")
    message = input("Server: ")
    conn.send(message.encode())
    if message.lower() == 'sm_sultan':
        break
conn.close()
sultan_server.close()
```

File Name:  client.py

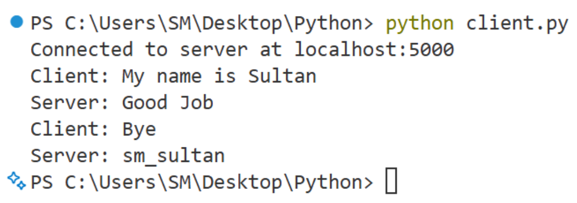
```python
import socket
sultan_server = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
HOST = 'localhost'
PORT = 5000
sultan_server.connect((HOST, PORT))
print(f"Connected to server at {HOST}:{PORT}")
while True:
    message = input("Client: ")
    sultan_server.send(message.encode())
    if message.lower() == 'sm_sultan':
        break

    data = sultan_server.recv(1024).decode()
    print(f"Server: {data}")
    if data.lower() == 'sm_sultan':
        break
sultan_server.close()
```

## Output:



```
PS C:\Users\SM\Desktop\Python> python server.py
Server started. Listening on localhost:5000
Connected by ('127.0.0.1', 61342)
Client: My name is Sultan
Server: Good Job
Client: Bye
Server: sm_sultan
Client disconnected.
PS C:\Users\SM\Desktop\Python>
```



```
PS C:\Users\SM\Desktop\Python> python client.py
Connected to server at localhost:5000
Client: My name is Sultan
Server: Good Job
Client: Bye
Server: sm_sultan
PS C:\Users\SM\Desktop\Python>
```

**Conclusion:** In this lab, basic socket communication between a client and server was successfully implemented, demonstrating how Python handles network connections.

**Lab 02:** Echo Socket Client-Server Communication

## Objective:

To implement an Echo server that returns the same message received from a client using Python sockets.

## Theory:

An Echo server reads data sent by a client and sends the same data back. This helps understand persistent TCP connections, message handling, and loop-based communication.

## Requirements:

- Python 3
- Understanding of Lab 01
- Echo server and client scripts

## Code:

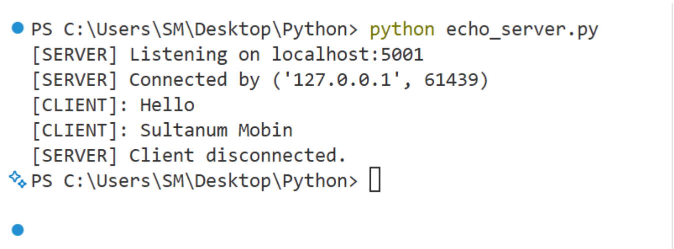
File Name: echo_server.py

```python
import socket
HOST = 'localhost'
PORT = 5001
sultan_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sultan_server.bind((HOST, PORT))
sultan_server.listen(1)
print(f"[SERVER] Listening on {HOST}:{PORT}")
conn, addr = sultan_server.accept()
print(f"[SERVER] Connected by {addr}")
while True:
    data = conn.recv(1024).decode()
    if not data or data.lower() == 'sm_sultan':
        print("[SERVER] Client disconnected.")
        break
    print(f"[CLIENT]: {data}")
    conn.send(data.encode())
conn.close()
sultan_server.close()
```

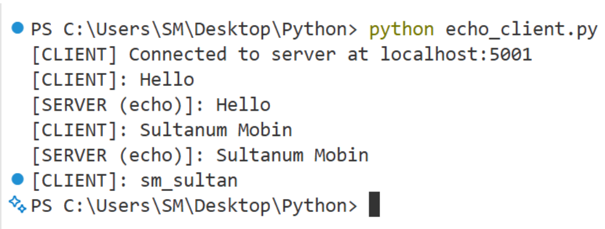
File Name: echo_client.py

```python
import socket
HOST = 'localhost'
PORT = 5001
sultan_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sultan_server.connect((HOST, PORT))
print(f"[CLIENT] Connected to server at {HOST}:{PORT}")
while True:
    message = input("[CLIENT]: ")
    sultan_server.send(message.encode())
    if message.lower() == 'sm_sultan':
        break
    echoed = sultan_server.recv(1024).decode()
    print(f"[SERVER (echo)]: {echoed}")
sultan_server.close()
```

**Output:**



```
PS C:\Users\SM\Desktop\Python> python echo_server.py
[SERVER] Listening on localhost:5001
[SERVER] Connected by ('127.0.0.1', 61439)
[CLIENT]: Hello
[CLIENT]: Sultanum Mobin
[SERVER] Client disconnected.
PS C:\Users\SM\Desktop\Python>
```



```
PS C:\Users\SM\Desktop\Python> python echo_client.py
[CLIENT] Connected to server at localhost:5001
[CLIENT]: Hello
[SERVER (echo)]: Hello
[CLIENT]: Sultanum Mobin
[SERVER (echo)]: Sultanum Mobin
[CLIENT]: sm_sultan
PS C:\Users\SM\Desktop\Python>
```

**Conclusion:** The Echo Server and Client successfully demonstrated two-way communication. This lab helps understand message loops, persistent connections, and data handling in socket programming.

**Experiment No:** 3
**Experiment Name:** Implementation of Socket Server and multi-threded Client communication in Python.

**Theory:** Computer networking is one of the core concepts in computer science. A socket allows communication between two endpoints over a network. In this assignment, we implement a **Python TCP Socket Server** which can handle multiple clients simultaneously using **multi-threading**. This enables parallel communication—multiple users can connect, send messages, and receive responses at the same time. Multi-threading ensures each client runs in a separate execution thread without blocking others.

**2 . Objectives**

This assignment aims to:

1. Implement a TCP socket server in Python
2. Establish a connection between server and multiple clients
3. Use **threading** to handle multiple clients concurrently
4. Demonstrate message sending & receiving
5. Execute and test the system in a Linux environment

**Server.py:**

```
import socket
import threading
def sultan_client(client_socket, address):
    print(f"[NEW CONNECTION] {address} connected.")
    while True:
        try:
            message = client_socket.recv(1024).decode()
            if not message:
                break
            print(f"[{address}] {message}")
            reply = f"Server received: {message}"
            client_socket.send(reply.encode())
        except:
            break
    print(f"[DISCONNECTED] {address}")
    client_socket.close()
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
HOST = '127.0.0.1'   # localhost
PORT = 12345
server_socket.bind((HOST, PORT))
server_socket.listen()
print(f"[SERVER STARTED] Listening on {HOST}:{PORT}")
while True:
    client_socket, address = server_socket.accept()
    thread = threading.Thread(
        target=sultan_client,
        args=(client_socket, address)
    )
    thread.start()
```

**Client.py:**

```
import socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
HOST = '127.0.0.1'
PORT = 12345
client_socket.connect((HOST, PORT))
while True:
    message = input("Enter message (type 'exit' to quit): ")
    if message.lower() == 'exit':
        break
    client_socket.send(message.encode())
    reply = client_socket.recv(1024).decode()
    print("Server:", reply)
client_socket.close()
```

**Sample Output**

**Server Terminal**

```
[ SERVER STARTED] Listening on
0.0.0.0:1012 [ NEW CONNECTION]
('127.0.0.1', 54532) connected.
 From ('127.0.0.1', 54532): Hello Server
[ACTIVE CONNECTIONS] 1
```

**Client Terminal**

```
Connected to server
You: Hello Server
Server: Server received: Hello Server
```

**8. Conclusion**

In this assignment, we successfully implemented:

- A TCP socket server
- Multiple clients communicating concurrently
- A fully working multi-threaded network communication system

This practical demonstrates real-world server-client communication, threading, and basic networking principles essential in distributed systems.

**Experiment No:** 4

**Experiment Name:** Implement and Establish an HTTP Server That Responds with a Webpage in Python.

**Theory:** An HTTP server is a software module that listens for incoming HTTP requests from clients and responds with web resources such as HTML pages, images, or files. In Python, we can easily create a simple HTTP server using built-in modules like http.server without installing any external libraries. This assignment demonstrates how to build a basic HTTP server in Python that returns a custom HTML webpage when accessed from a browser.

**Implementation Procedure:**

1. Import the required socket module in Python.
2. Create a TCP socket using IPv4 addressing.
3. Bind the socket to a local IP address and a port number.
4. Put the server socket into listening mode.
5. Accept incoming client connections.
6. Receive the HTTP request from the client.
7. Create an HTTP response containing HTML content.
8. Send the response back to the client.
9. Close the client connection.

**HTTP Server Program(server.py):**

```
import socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
HOST = "127.0.0.1"
PORT = 8080
server_socket.bind((HOST, PORT))
server_socket.listen(1)
print("HTTP Server running at http://127.0.0.1:8080")
while True:
    client_socket, client_address = server_socket.accept()
    print("Connection received from:", client_address)
    request = client_socket.recv(1024).decode()
    print("Client Request:")
    print(request)
    html_content = """
    <html>
    <head>
        <title>Python HTTP Server</title>
    </head>
    <body>
        <h1>Hello from Python HTTP Server</h1>
        <p>This webpage is served using socket programming.</p>
    </body>
    </html>
    """
    response = (
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/html\r\n"
        f"Content-Length: {len(html_content)}\r\n"
        "\r\n"
        + html_content
    )
    client_socket.sendall(response.encode())
    client_socket.close()
```

**Output:**

- When the server program is executed, it starts listening on port 8080.
- Upon entering http://127.0.0.1:8080 in a web browser, the HTML webpage is displayed.
- The webpage shows a heading and a paragraph served by the Python HTTP server.

**Sample Output (Browser):**

```
Hello from Python HTTP Server
This webpage is served using socket programming.
```

**Conclusion:**

In this experiment, an HTTP server was successfully implemented using Python socket programming. The server was able to receive HTTP requests from a web browser and respond with an HTML webpage. This experiment demonstrates the basic working principle of HTTP communication and provides a clear understanding of how web servers handle client requests and responses.

**Experiment No:** 7

**Experiment Name:** Testing HTML Code and a Web Page Using Simple HTTP Server and Localhost HTTP Server in Python.

**Theory:** An HTTP server is responsible for handling client requests and delivering web content such as HTML pages. Python provides a built-in module called http.server which allows easy creation of a **Localhost-only HTTP Server** – serves web pages only on the local machine using IP address 127.0.0.1. In this experiment, an HTML file is tested using both servers to verify proper webpage rendering in a web browser. This confirms that the servers can correctly serve static HTML content.

**Implementation Procedure:**

1. Create an HTML file (index.html) containing basic webpage content.
2. Place the HTML file in the same directory as the server programs.
3. Write a Python program to create a Simple HTTP Server using the http.server module.
4. Run the Simple HTTP Server and access it using a web browser.
5. Write another Python program to create a Localhost-only HTTP Server.
6. Run the Localhost HTTP Server and access it using 127.0.0.1.
7. Observe the output displayed in the browser for both servers.

**Code:**

**HTML File (index.html)**

```html
<!DOCTYPE html>
<html>
<head>
   <title>Lab 07 HTML Test</title>
</head>
<body>
   <h1>HTML Page Test Successful</h1>
   <p>Python HTTP Server Made by Sultan.</p>
</body>
</html>
```

**Localhost HTTP Server**

```python
from http.server import HTTPServer, SimpleHTTPRequestHandler
HOST = "127.0.0.1"
PORT = 8080
server = HTTPServer((HOST, PORT), SimpleHTTPRequestHandler)
server.serve_forever()
```

**Output:**

- The HTML webpage is successfully displayed in the web browser when accessed using:
    - http://127.0.0.1:8000 (Simple HTTP Server)
    - http://127.0.0.1:8080 (Localhost HTTP Server)
- The webpage renders correctly with the given HTML content.
- The Localhost HTTP Server is accessible only from the local machine.

**Conclusion:**

In this experiment, an HTML webpage was successfully tested using both the Simple HTTP Server and the Localhost-only HTTP Server implemented in Python. The results confirm that Python HTTP servers can efficiently serve static HTML content and that restricting the server to localhost improves security by preventing external access.

**Experiment No:** 9

**Experiment Name:** Configure Your Mobile With New DNS Address

**Theory**: DNS (Domain Name System) is a naming system that translates **domain names** (e.g., google.com) into **IP addresses** that computers and mobile devices can understand. By default, mobile devices use DNS servers provided by the ISP or mobile network operator. However, users can manually configure **custom DNS servers** (such as public DNS) to achieve:

- Faster browsing
- Better reliability
- Improved security
- Content filtering (in some cases)

**Implementation Procedure:**

**Steps to Configure DNS on Android Mobile (Wi-Fi):**

1. Open **Settings** on the mobile device.
2. Go to **Wi-Fi** settings.
3. Long press on the connected Wi-Fi network.
4. Select **Modify network**.
5. Enable **Advanced options**.
6. Change **IP settings** from *DHCP* to *Static*.
7. Enter the following DNS addresses:
    - DNS 1: 8.8.8.8
    - DNS 2: 8.8.4.4
8. Save the settings.
9. Reconnect to the Wi-Fi network.

**Testing Method:**

1. Open a web browser.
2. Visit websites such as:
    - www.google.com
    - www.openai.com
3. Check if websites load correctly.

4. Optionally, use a DNS testing website to confirm the DNS server in use.

**Observation / Output:**

- Internet connection works normally after DNS configuration.
- Websites resolve faster in some cases.
- Domain names are successfully converted to IP addresses using the new DNS server.
- No connection errors observed.

**Conclusion:**

In this experiment, a mobile device was successfully configured with a new DNS address. The device was able to resolve domain names correctly and access the internet without issues. This experiment demonstrates the importance of DNS in network communication and shows how custom DNS servers can be used to improve reliability and performance.

**Experiment No:** 10

**Experiment Name:** Accessing a Website Using a Local DNS Server (Sultan_1021.com)

**Theory:**

The Domain Name System (DNS) is used to translate domain names into IP addresses. Normally, this translation is handled by public or ISP-provided DNS servers. However, it is also possible to configure a **local DNS server** that resolves specific domain names within a local system or network.

A **local DNS server** allows custom domain names (such as Sultan_1021.com) to be mapped to a local IP address like 127.0.0.1. This technique is commonly used for testing websites, development environments, and learning DNS concepts without accessing the internet.

**Implementation Procedure:**

1. Create and run a simple DNS server using Python.
2. Add a DNS entry for the domain Sultan_1021.com mapped to 127.0.0.1.
3. Configure the client system to use the local DNS server.
4. Create a simple HTML webpage to represent the website.
5. Run a local HTTP server on localhost.
6. Access the website using the custom domain name in a web browser.
7. Verify that the site loads successfully using the local DNS resolution.

**Code:**

**DNS Server Program (dns_server.py)**

```
import socket
dns_records = {
    "Sultan_1021.com": "127.0.0.1"
}
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
HOST = "127.0.0.1"
PORT = 5353
server_socket.bind((HOST, PORT))
print("Local DNS Server running...")
while True:
    data, addr = server_socket.recvfrom(1024)
    domain = data.decode().strip()
    ip = dns_records.get(domain, "Domain not found")
    server_socket.sendto(ip.encode(), addr)
```

**HTTP Server Program (http_server.py)**

```
from http.server import HTTPServer, SimpleHTTPRequestHandler
HOST = "127.0.0.1"
PORT = 8080
server = HTTPServer((HOST, PORT), SimpleHTTPRequestHandler)
print("HTTP Server running on localhost...")
server.serve_forever()
```

**HTML File (index.html)**

```
<!DOCTYPE html>
<html>
<head>
   <title>Local DNS Test</title>
</head>
<body>
   <h1>Welcome to Sultan_1021.com</h1>
   <p>This website is accessed using a local DNS server.</p>
</body>
</html>
```

**Output:**

- The domain name Sultan_1021.com is successfully resolved to 127.0.0.1.
- The webpage loads correctly in the browser when accessed using:
- http://Sultan_1021.com:8080
- The site is accessible only on the local machine.

**Conclusion:**

In this experiment, a custom domain name Sultan_1021.com was successfully accessed using a local DNS server. The local DNS server correctly resolved the domain name to the localhost IP address, and the HTTP server served the webpage without using any public DNS service. This experiment demonstrates the practical use of local DNS configuration and helps in understanding DNS resolution mechanisms.