



**TRIBHUVAN UNIVERSITY**  
**INSTITUTE OF ENGINEERING**  
**THAPATHALI CAMPUS**

**A Lab Report**  
**of**  
**Distributed System**  
**on**  
**Client-Server Implementation**  
**Using Python**

**Submitted By:**  
**Prinsa Joshi**  
**(THA076BCT031)**

**Submitted To:**  
Department of Electronics and Computer Engineering  
Thapathali Campus  
Kathmandu, Nepal

June, 2023

**TITLE: SIMPLE CLIENT-SERVER IMPLEMENTATION USING PYTHON****THEORY:**

**Client-server architecture** is a fundamental concept in networking and distributed computing. It is widely used in various applications, including web servers, database systems, email systems, and more. The basic components of a client-server implementation include:

- **Client:** A client is a program or device that requests services from a server. It can be a desktop application, a web browser, a mobile app, or any other entity that needs to access resources or perform tasks provided by a server. The client initiates a connection to the server, sends requests for specific services, and waits for the server's response.
- **Server:** A server is a program or device that provides services to clients. It listens for incoming requests from clients, processes those requests, and sends back the corresponding responses. Servers have dedicated hardware and software resources to handle multiple client connections simultaneously. They are designed to be robust, scalable, and capable of serving numerous clients concurrently.
- **Communication Protocol:** To facilitate communication between the client and server, a communication protocol is used. A protocol defines the rules and formats for exchanging messages between the client and server, ensuring that both parties understand and interpret the messages correctly. Some commonly used protocols in client-server communication include HTTP (Hypertext Transfer Protocol), TCP/IP (Transmission Control Protocol/Internet Protocol), and SMTP (Simple Mail Transfer Protocol).

**Client-server architecture provides several benefits, including:**

- **Scalability:** Servers can handle multiple client connections simultaneously, allowing for scalable systems that can serve a large number of users.
- **Centralized Resources:** Servers consolidate resources and provide centralized access, enabling efficient management and utilization of shared resources.
- **Security:** By centralizing access control and authentication, client-server architectures can enhance security measures, protecting sensitive data and

limiting unauthorized access.

- **Maintenance and Updates:** Servers can be updated or maintained independently, ensuring minimal disruption to clients accessing the services.
- **Fault Tolerance:** Servers can be designed with redundancy and fault tolerance mechanisms to ensure

**STEPS IN CLIENT-SERVER INTERACTION:**

The interaction between a client and server typically follows these steps:

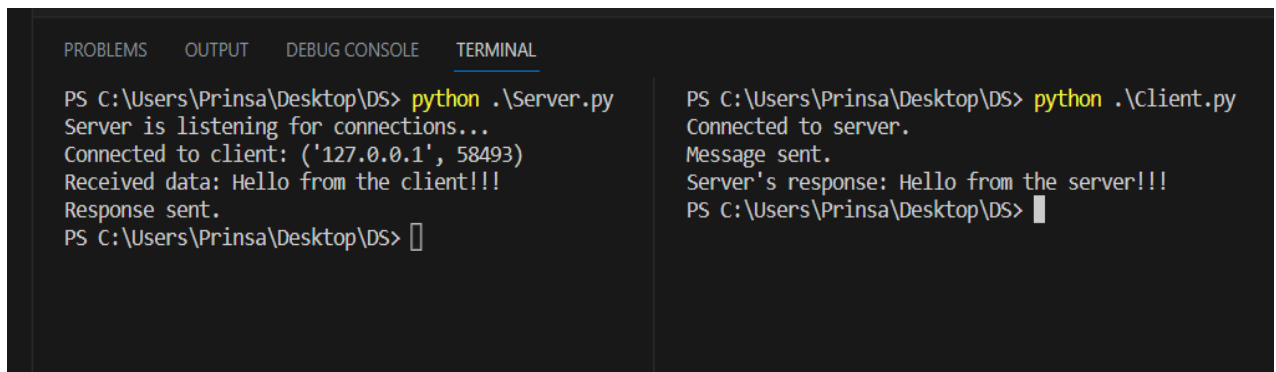
- **Establishing Connection:** The client establishes a network connection with the server. This connection can be established using various protocols, such as TCP/IP or UDP (User Datagram Protocol). The client needs to know the IP address and port number of the server to initiate the connection.
- **Sending Requests:** Once the connection is established, the client sends requests to the server. These requests can be in the form of HTTP requests, database queries, file transfers, or any other specific commands supported by the server. The client includes the necessary information or data in the request for the server to process.
- **Processing Requests:** The server receives the client's requests and processes them according to the requested services. It may access databases, perform computations, retrieve files, or execute any other required operations. The server's logic and functionality depend on the specific application it serves.
- **Generating Responses:** After processing the request, the server generates the corresponding response. The response contains the requested information, results, or actions. It may include data, error messages, status codes, or other relevant details.
- **Sending Responses:** The server sends the response back to the client through the established connection. The response is delivered to the client based on the communication protocol being used. The client receives the response and proceeds with further actions based on the received information.
- **Closing the Connection:** Once the interaction is complete, either the client or server (or both) can initiate the closure of the connection. This frees up resources and allows both parties to handle new connections or perform other tasks. Closing the connection can be done explicitly by sending a termination signal or implicitly based on predefined conditions.

**CODE:****Server.py**

```
1  import socket
2
3  # Create a socket object
4  server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5
6  # Define the host and port
7  host = 'localhost'
8  port = 12345
9
10 # Bind the socket to the host and port
11 server_socket.bind((host, port))
12
13 # Listen for incoming connections
14 server_socket.listen(1)
15 print("Server is listening for connections...")
16
17 # Accept a client connection
18 client_socket, addr = server_socket.accept()
19 print("Connected to client:", addr)
20
21 # Receive data from the client
22 data = client_socket.recv(1024).decode()
23 print("Received data:", data)
24
25 # Send a response back to the client
26 response = "Hello from the server!!!"
27 client_socket.send(response.encode())
28 print("Response sent.")
29
30 # Close the connection
31 client_socket.close()
32 server_socket.close()
```

**Client.py**

```
1  import socket
2
3  # Create a socket object
4  client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5
6  # Define the host and port to connect to
7  host = 'localhost'
8  port = 12345
9
10 # Connect to the server
11 client_socket.connect((host, port))
12 print("Connected to server.")
13
14 # Send data to the server
15 message = "Hello from the client!!!"
16 client_socket.send(message.encode())
17 print("Message sent.")
18
19 # Receive the server's response
20 response = client_socket.recv(1024).decode()
21 print("Server's response:", response)
22
23 # Close the connection
24 client_socket.close()
25
```

**Output:**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\Prinsa\Desktop\DS> python .\Server.py
Server is listening for connections...
Connected to client: ('127.0.0.1', 58493)
Received data: Hello from the client!!!
Response sent.
PS C:\Users\Prinsa\Desktop\DS>

PS C:\Users\Prinsa\Desktop\DS> python .\Client.py
Connected to server.
Message sent.
Server's response: Hello from the server!!!
PS C:\Users\Prinsa\Desktop\DS>
```

**RESULT:**

The client-server implementation was successful as the server listened on the specified IP address and port, accepted the client's connection, received the message, and sent back an acknowledgment. The client established a connection with the server, sent a message, and received the acknowledgment.

**DISCUSSION:**

The client-server architecture provides a scalable and efficient approach for distributed systems. It allows multiple clients to connect to a server and request services. Communication protocols play a crucial role in ensuring proper message exchange. This implementation demonstrated the basic steps involved in establishing a client-server connection and exchanging messages, highlighting the importance of synchronized communication between the client and server.

**CONCLUSION:**

Through this experiment, I have learned the basics of client-server architecture and its implementation. I gained knowledge about establishing network connections, sending and receiving messages between clients and servers, and the importance of communication protocols. This experience has deepened my understanding of networked systems and their practical applications.