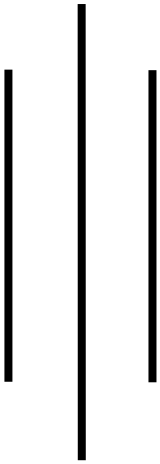
TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS



A Lab Report

On

Simple Client Server Implementation Using Python

Submitted By:

Name: Adhip Bhattarai

Roll No: THA076BCT004

Date: 7th June, 2023

Submitted To:

Department of Electronics and Computer Engineering

SIMPLE CLIENT-SERVER IMPLEMENTATION USING PYTHON

THEORY:

In a simple client-server architecture, there are two main components: the client and the server. The client is a device or program that requests and consumes services from the server. The server, on the other hand, is a device or program that provides services to clients.

Here's a breakdown of how the client-server architecture works:

- The client sends a request: The client initiates communication by sending a request to the server. This request can be for various services such as accessing a file, retrieving data from a database, or performing a specific action.
- The server receives the request: The server listens for incoming requests and receives the request sent by the client.
- The server processes the request: Once the server receives the request, it processes the request based on the type of service being requested. This may involve retrieving data, performing calculations, or executing specific operations.
- The server sends a response: After processing the request, the server generates a response containing the requested information or the result of the operation. The response is then sent back to the client.
- The client receives the response: The client receives the response from the server, which contains the requested data or the outcome of the requested action.
- Client consumes the response: Upon receiving the response, the client consumes and utilizes the information as needed. This can involve displaying the data to the user, performing further operations based on the received result, or making subsequent requests to the server.

This client-server model allows for a distributed system where multiple clients can request services from a centralized server. It promotes scalability, as additional clients can be added without impacting the server's functionality. It also allows for a clear separation of concerns, as the server focuses on providing services while the client handles user interactions and presentations.

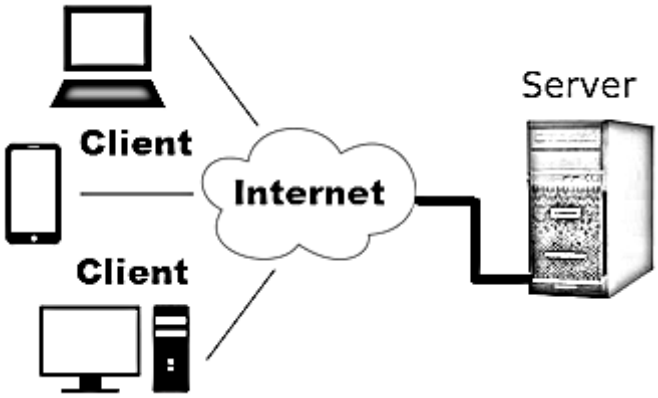


Figure 1. Client Server Architecture

ALGORITHM:

For Server Side

Step 1: Start

Step 2: Create a socket object for the server

Step 3: Bind the socket into a unique address

Step 4: Set the socket to listen for incoming connections with a maximum backlog of 1 client

Step 5: Enter an infinite loop to continuously accept client connections and handle them

Step 6: Accept a new client connection using the "accept" method. This will return a new socket object "c_socket" representing the client socket, and the address of the client "addr"

Step 7: Receive data from the client. Also, specify the maximum number of bytes to receive and decode the received bytes.

Step 8: Print a message indicating that the client is successfully connected, along with their name, address, and the client socket.

Step 9: Send a welcome message to the client using the "send" method of the client socket, by converting the message to bytes.

Step 10: Close the client socket

For Client Side

Step 1: Start

Step 2: Create a socket object for the client.

Step 3: Connect the client socket to the server by providing the server address and the port number

Step 4: Prompt the user to enter their name

Step 5: Convert the name given by the user into bytes and send it through the client socket

Step 6: Receive the response from the server. Also, specify the maximum number of bytes to receive. Decode the received bytes.

Step 7: Print the decoded message from the server.

Step 8: End

CODE:

For Server Side:

```
import socket

# Create a socket object

s = socket.socket()

# Print a message indicating that the socket was created successfully

print("Socket successfully created")

# Bind the socket to the address ('localhost', 9999)

s.bind(('localhost', 9999))

# Set the socket to listen for incoming connections with a backlog of 1 client

s.listen(1)

# Print a message indicating that the server is waiting for clients to connect

print("Waiting for clients to be connected.....")

# Enter an infinite loop to continuously accept client connections and handle them

while (1):

    # Accept a new client connection
```

```
c_socket, addr = s.accept()

# Receive data from the client, assuming the maximum number of bytes to receive
    #is 2048

name = c_socket.recv(2048).decode()

# Print a message indicating that the client is successfully connected, along with
    #their name, address, and the client socket

print(f"{name} is successfully connected {addr} {c_socket}")

# Send a welcome message to the client, converting the message to bytes using the
    #UTF-8 encoding

c_socket.send(bytes('Welcome to our Server', 'utf-8'))

# Close the client socket

c_socket.close()
```

For Client Side:

```
import socket

# Create a socket object for the client

c_socket = socket.socket()

# Connect the client socket to the server at 'localhost' on port 9999

c_socket.connect(('localhost', 9999))

# Prompt the user to enter their name

name = input("Enter your name: ")

# Send the user's name to the server, converting it to bytes using the UTF-8 encoding

c_socket.send(bytes(name, 'utf-8'))

# Receive the response from the server, assuming the maximum number of bytes to
    #receive is 2048
```

```
message = c_socket.recv(2048).decode()

# Print the message received from the server

print(message)
```

RESULT:

```
Socket successfully created
Waiting for clients to be connected.....
Adhip is successfully connected ('127.0.0.1', 61367) <socket.socket fd=1480, family=AddressFamily.AF_INET, type=SocketKind.SOCK
_STREAM, proto=0, laddr=('127.0.0.1', 9999), raddr=('127.0.0.1', 61367)>
```

Figure 2. Result of Server

```
Enter your name: Adhip
Welcome to our Server
```

Figure 3. Result of Client

DISCUSSION:

In this way, we implemented a simple client-server architecture using Python. This architecture relies on the built-in "socket" library in Python, which provides the necessary functionality for network communication. The "socket" library in Python allows us to create sockets, which are the endpoints for sending and receiving data over a network. With sockets, we can establish communication channels between the client and server, enabling them to exchange information.