TCP-Based Web Server: Design Document

Introduction

This document outlines the design and implementation of a TCP-based web server application. The web server is built using Python and socket programming techniques to handle HTTP requests. The server processes client requests for web files, responds with the appropriate content, and can handle multiple requests concurrently using a multi-threaded design.

The implementation consists of three main components:

1. **Single-client Web Server:** This handles one HTTP request at a time.
2. **Multi-threaded Web Server:** This extends the basic server to handle multiple clients simultaneously.
3. **Custom HTTP Client:** A client that interacts with the server, sending HTTP requests and displaying the server's responses.

Design Assumptions

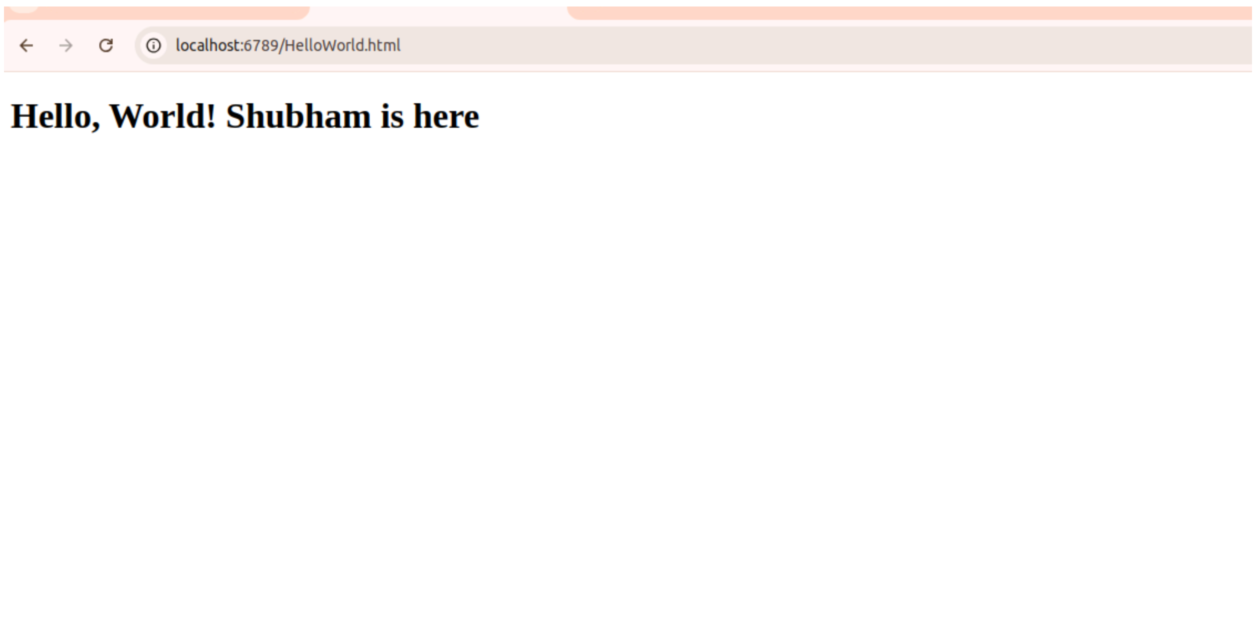
The following assumptions were made during the design and implementation of the web server:

- **HTTP Version:** The server and client both use HTTP/1.1 protocol to send and receive messages.
- **Port Number:** The server runs on a non-reserved port number (e.g., 6789). This port is specified manually in the server code.
- **File System Structure:** The server looks for requested files in the same directory where the server is running. This keeps the system simple for the lab environment.
- **File Type:** The server assumes that the requested file is a standard HTML file, though it can be extended to handle other file types such as CSS, JavaScript, or images.
- **Client Requests:** The client requests files using HTTP GET methods only. Other methods (POST, PUT, DELETE, etc.) are not handled in this version.
- **Error Handling:** If a file requested by the client does not exist, the server responds with a "404 Not Found" status.

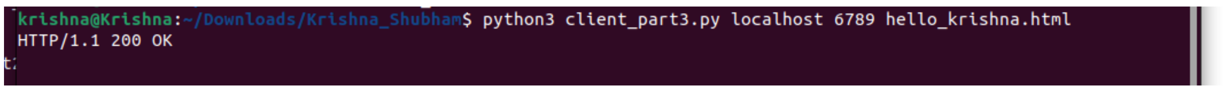
System Components

1. **Socket Creation:** The web server is built using the Python `socket` module. A socket is created using the `socket()` function with the following parameters:
   - **Address Family:** `AF_INET`, which specifies IPv4.
   - **Socket Type:** `SOCK_STREAM`, which enables TCP communication.
2. The server socket is bound to the local machine's IP address (or an empty string `''` to accept requests on all available interfaces) and a specific port (e.g., 6789).
3. **Handling Client Requests:** The server waits for incoming client connections using the `accept()` method. Once a client connects, the server:

    - Receives the HTTP request using the `recv()` method.
    - Parses the request to extract the requested file.
    - Attempts to open the file in the server's directory.
    - Sends an HTTP response with a status line, headers, and the file content (if the file exists) or a 404 error message (if it doesn't).
4. After handling the request, the server closes the connection using `close()`.
5. **HTTP Response Format:** The server's response follows the HTTP/1.1 standard. The structure is as follows:
    - **Status Line:** `HTTP/1.1 200 OK` or `HTTP/1.1 404 Not Found`.
    - **Headers:** Basic headers (such as content type) could be added to inform the client about the file being sent.
    - **Body:** The content of the requested file (for successful responses) or an error message (for failed requests).



Hello, World! Shubham is here



```
krishna@Krishna:~/Downloads/Krishna_Shubham$ python3 client_part3.py localhost 6789 hello_krishna.html
HTTP/1.1 200 OK
```

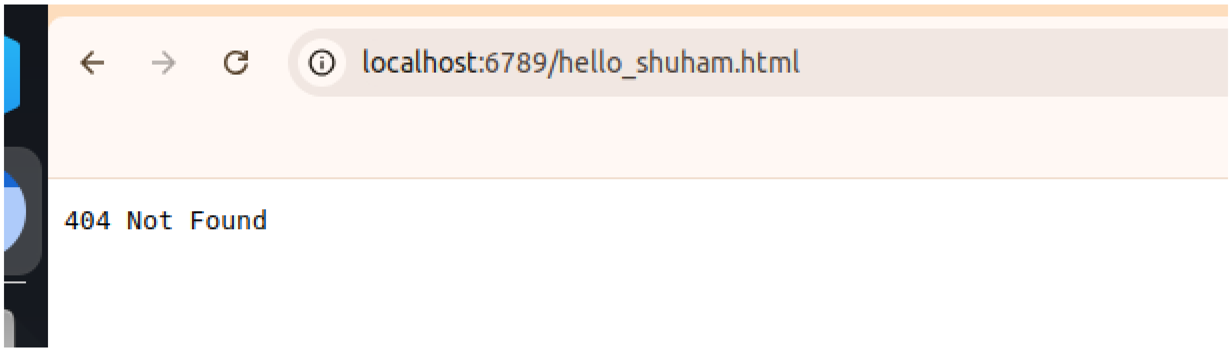
Multi-Threaded Server

The single-client server handles one request at a time, which limits its functionality in real-world use cases. To improve scalability, we implemented a **multi-threaded server** that can handle multiple clients concurrently.

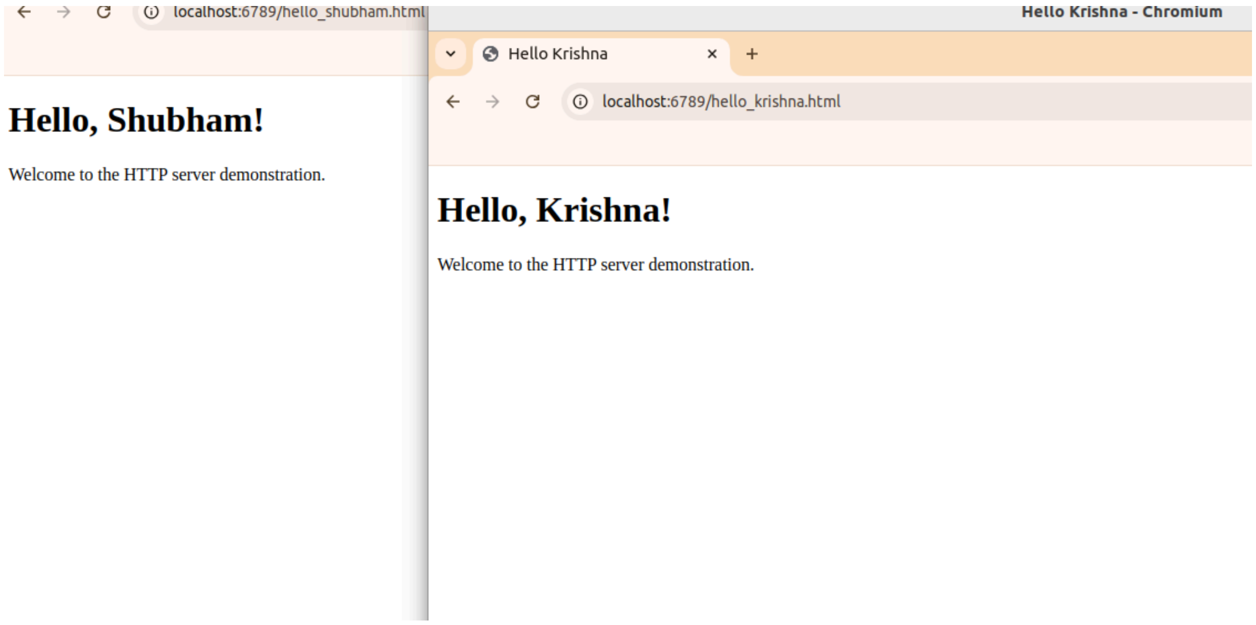
1. **Thread Creation:** For each new client connection, a new thread is created using Python's `threading.Thread()` method. Each thread is responsible for handling the entire lifecycle of a single client request, including receiving the HTTP request, serving the file, and sending the response back.

The main thread continuously listens for new client connections, while individual threads are spawned to process client requests concurrently.

2. **Thread Management:** Each client connection is handled in its own thread using a function `handle_client()`. This function performs the same steps as the single-client server, but in parallel with other threads. Once a thread finishes serving a client, it terminates.
   The threading design ensures that:
   - Multiple clients can connect to the server simultaneously.
   - Each client is served independently without blocking others.
3. The multi-threaded approach enhances the server's ability to scale and handle more complex workloads, such as serving multiple large files simultaneously.


localhost:6789/hello_shuham.html

```
404 Not Found
```



localhost:6789/hello_shubham.html

Hello, Shubham!

Welcome to the HTTP server demonstration.

localhost:6789/hello_krishna.html

Hello, Krishna!

Welcome to the HTTP server demonstration.

Client Design

The client is a simple Python program that sends HTTP GET requests to the server and displays the server's response.

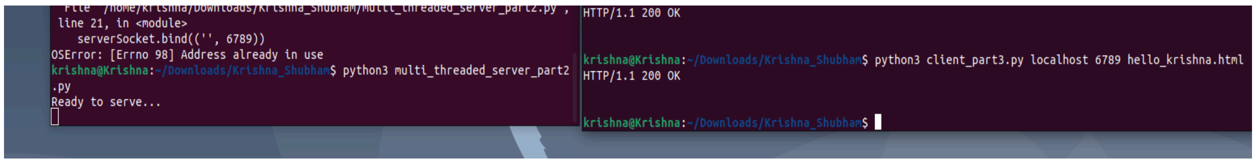
1. **TCP Connection:** The client establishes a TCP connection to the server using the `connect()` method. The server's IP address and port are provided as command-line arguments.
2. **HTTP Request:** The client sends an HTTP GET request formatted as:
3. **Response Handling:** The client receives the server's response using `recv()`. It then prints the response to the terminal. If the requested file exists on the server, the client displays the content of the file. If the file does not exist, it displays the 404 error message.

Error Handling

1. **File Not Found (404 Error):** If the requested file is not found in the server's directory, the server returns a `404 Not Found` status with a simple HTML message.
2. **Connection Errors:** If the client cannot connect to the server (e.g., the server is not running or the port is incorrect), the client catches the connection error and prints an appropriate error message.

Testing the System

1. **Single-client Testing:** To test the single-client server, the server was run on the local machine, and a browser or custom client was used to request an HTML file. The server successfully responded with the file content or a 404 error when the file was not found.
2. **Multi-threaded Testing:** The multi-threaded server was tested by simulating multiple clients connecting to the server simultaneously using the custom client. The server successfully handled multiple requests at the same time, with no delays or crashes.
3. **Custom Client Testing:** The custom client was tested by sending requests for different files to the server. The client was able to receive and display both successful responses and 404 error messages.



```
  File "/home/krishna/Downloads/Krishna_Shubham/multi_threaded_server_part2.py",
 line 21, in <module>
    serverSocket.bind(('', 6789))
OSError: [Errno 98] Address already in use
krishna@Krishna:~/Downloads/Krishna_Shubham$ python3 multi_threaded_server_part2
.py
Ready to serve...
```

```
HTTP/1.1 200 OK


krishna@Krishna:~/Downloads/Krishna_Shubham$ python3 client_part3.py localhost 6789 hello_krishna.html
HTTP/1.1 200 OK


krishna@Krishna:~/Downloads/Krishna_Shubham$
```

Conclusion

This project demonstrates the use of TCP socket programming in Python to implement a basic web server capable of handling HTTP requests. The initial design handles one request at a time, while the multi-threaded extension allows the server to handle multiple clients concurrently. The system was tested thoroughly using both a web browser and a custom client.

This design can be further extended to handle more advanced features, such as support for additional HTTP methods, file types, and security features like HTTPS.