COMP2322 Final Project

Multi-thread Web Server

Tanya Budhrani 22097189d

Introduction

The web server implementation represents a pivotal solution in addressing the contemporary need for robust and efficient web hosting environments. This server is meticulously designed to handle concurrent connections, offering a seamless experience for both clients and servers alike.

At its core, the server leverages Python's socket and threading modules to establish and manage connections with clients. Upon receiving an HTTP request, it parses the request to determine the requested file, fetches the file from the server's file system, and crafts an appropriate HTTP response. This response includes essential header information such as status codes and content length, ensuring proper communication between the server and client.

Furthermore, the server's design encapsulates key functionalities, including error handling for file not found scenarios and logging of client requests. By logging each request, the server enables administrators to monitor traffic and diagnose potential issues effectively.

The server's configuration is highly customizable, allowing users to specify the host IP, port number, and directory from which files are served. This flexibility ensures seamless integration into various hosting environments, catering to diverse user requirements. This report will delve into the intricacies of our server's design and implementation, elucidating its core functionalities, adherence to specified requirements, and practical applications.

Design

The web server program represents a carefully crafted solution designed to facilitate seamless communication between clients and servers, offering robust handling of HTTP requests in modern web

hosting environments. In this section, we delve deep into the design of the program, exploring its architectural components, library usage, networking capabilities, and core functionalities.

Library Usage:

- **Socket Library**: The program extensively utilizes Python's `socket` library to establish and manage network connections. By creating a TCP socket (`socket.AF_INET`, `socket.SOCK_STREAM`), the server binds to a specified host and port, enabling it to listen for incoming connections.
- **Threading Library**: Leveraging the `threading` library, the program achieves concurrency by spawning multiple threads to handle incoming client requests concurrently. This multithreaded approach enhances the server's responsiveness and scalability, enabling it to serve multiple clients simultaneously without blocking the main server loop.
- **OS Library**: The program employs the `os` library to interact with the file system. When handling client requests, it utilizes file system operations to access and serve files requested by clients. This modular design allows for flexible configuration of the directory from which files are served (`BASE_DIR`), accommodating diverse hosting environments and file structures.
- **Datetime Library**: Utilizing the `datetime` library, the program accurately records timestamps for logging activities. This ensures precise tracking of client interactions, enabling administrators to monitor server traffic and diagnose potential issues effectively.

Networking Capabilities:

- **TCP/IP Communication**: The program harnesses Python's socket programming capabilities to establish TCP/IP connections with clients. This reliable and connection-oriented communication

protocol ensures robust data transmission between the server and clients, facilitating seamless exchange of HTTP requests and responses.

- **Socket Binding and Listening**: Upon initialization, the server binds to a specified host IP address (`HOST`) and port number (`PORT`), enabling it to listen for incoming connection requests from clients. By listening on a designated interface and port, the server awaits incoming connections, ready to handle client requests promptly upon arrival.
- **Concurrent Handling of Connections**: Leveraging threading, the program efficiently manages multiple client connections concurrently. Each incoming connection is accepted, and a dedicated thread is spawned to handle the client request. This concurrent processing capability enhances the server's responsiveness, ensuring optimal performance even under heavy client loads.

Core Functionalities Offered by the Code:

- **Server Configuration**: The program offers flexibility in configuring server parameters such as the host IP address and port number. Users can customize these parameters to suit specific deployment environments, allowing for seamless integration into diverse network configurations.
- **Request Parsing**: Upon receiving an HTTP request from a client, the program parses the request to extract relevant information such as the requested file path and HTTP method. This parsing mechanism enables the server to interpret client requests accurately and respond accordingly.
- **Connection Management**: The program efficiently manages client connections by accepting incoming connections, spawning dedicated threads to handle each connection, and closing connections after serving requests. This streamlined connection management ensures optimal resource utilization and prevents resource exhaustion under heavy client loads.
- **Logging Activities**: To facilitate monitoring and analysis of server traffic, the program logs each client request along with relevant details such as the client's IP address, accessed file, response

status code, and timestamp. This logging mechanism enables administrators to track server activity, identify potential issues, and analyze traffic patterns effectively.

Implementation

In this section, we provide a comprehensive analysis of the implementation of our Python-based web server program, delving into the functionality and inner workings of each function.

1. **handle_client Function**:
    - The `handle_client` function serves as the core handler for incoming client requests.
    - Upon receiving a connection object (`conn`) and client address (`addr`), it prints a message indicating the client's connection.
    - It then receives the HTTP request from the client using the `recv` method, decoding the received bytes into a UTF-8 encoded string.
    - The function parses the HTTP request to extract the requested file path using string manipulation techniques. Specifically, it splits the request string and retrieves the second element, which represents the requested file path.
    - If the requested file path is '/', indicating the root directory, the function redirects the request to serve the default file, which in this implementation is 'html.html'.
    - Next, the function constructs the full file path by joining the base directory path (`BASE_DIR`) with the requested file path, excluding the leading slash.
    - It then checks if the requested file exists on the server's file system using the `os.path.exists` function.
    - If the file exists, it reads the contents of the file using a binary read operation (`rb`) and stores the content in a variable named `content`.

- Subsequently, the function constructs an HTTP response containing the file content along with relevant headers, including the `Content-Length` header indicating the size of the content.
- In case the requested file is not found, the function constructs a 404 Not Found error response.
- Finally, the function sends the HTTP response back to the client using the `sendall` method, logs the request using the `log_request` function, and closes the connection using the `close` method.

2. **log_request Function**:
    - The `log_request` function is responsible for logging each client request along with relevant details such as the client's IP address, accessed file, response status code, and timestamp.
    - It accepts the client's IP address (`addr`), requested file path (`requested_file`), and response data (`response`) as input parameters.
    - Using Python's `datetime` module, the function retrieves the current timestamp in the format `'%Y-%m-%d %H:%M:%S'`, representing the year, month, day, hour, minute, and second.
    - It extracts the status code from the HTTP response data using string manipulation techniques, splitting the response string and retrieving the second element.
    - The function constructs a log entry string containing the client's IP address, access time, requested file path, and response status code.
    - Finally, it appends the log entry to a text file named `server_log.txt` using the `write` method in append mode (`'a'`), ensuring that each log entry is recorded sequentially.

3. **main Function**:

- The `main` function serves as the entry point of the script, responsible for initializing the server and handling incoming client connections.
- It creates a TCP socket using the `socket.socket` constructor, specifying the address family (`socket.AF_INET`) and socket type (`socket.SOCK_STREAM`) for IPv4 and TCP communication, respectively.
- The function then binds the socket to the specified host IP address (`HOST`) and port number (`PORT`) using the `bind` method.
- It sets the socket to listen for incoming connections using the `listen` method, allowing the server to accept incoming connection requests from clients.
- Within a continuous loop, the function accepts incoming connection requests using the `accept` method, which returns a new socket object (`conn`) and the client's address (`addr`).
- For each accepted connection, the function spawns a new thread using the `threading.Thread` constructor, passing the `handle_client` function as the target and the connection object (`conn`) and client address (`addr`) as arguments.
- This multithreaded approach enables the server to handle multiple client connections concurrently, ensuring optimal performance and responsiveness.

Demonstration

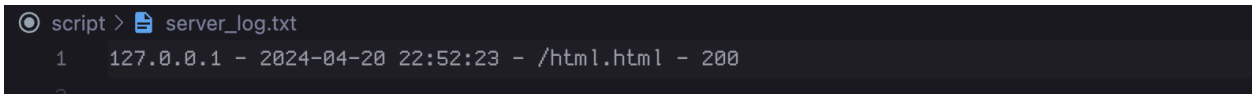
Running the server file using python3 server.py

After typing http://localhost:8080/ into browser

Output and results

```
Tanyas-MacBook-Air:script tanyabudhrani$ python3 server.py
Server listening on 127.0.0.1:8080
Connected by ('127.0.0.1', 55674)
Request:
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
sec-ch-ua: "Not-A.Brand";v="99", "Chromium";v="124"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "macOS"
DNT: 1
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9
```

Log file


script > server_log.txt
1 127.0.0.1 - 2024-04-20 22:52:23 - /html.html - 200