**100071011 Computer Networks 2023-2024-2**

**Project-2**

**Building a CGI-Support Multi-Threaded Web Server**

**Specification**

|  |  |
| --- | --- |
| **学号 (Student ID)** | **1820243077** |
| **姓名 (Name)** | **Pimenov Gleb** |
| **班号 (Class No.)** | **1107** |
| **授课教师 (Instructor)** | 杨松 |

**School of Computer**

**Beijing Institute of Technology**

**Jun 4, 2024**

1. **Requirement Analysis**

The objective of this project is to develop a multi-threaded web server that supports CGI (Common Gateway Interface) scripting. This server will handle both static and dynamic content, processing HTTP GET, POST, and HEAD requests. Key functionalities include managing TCP connections, handling multiple simultaneous connections using threading, and logging detailed request and response information.

**Key Functions**

* **Static Page Handling**: Serving HTML and other static files from the webroot directory.
* **Dynamic Content Handling**: Executing CGI scripts to generate dynamic content based on user input.
* **HTTP Methods**: Support for GET, POST, and HEAD methods.
* **Thread Management**: Using a thread pool to handle multiple concurrent connections.
* **Logging**: Recording detailed information about each request and response.

1. **Design**

System Model and Flow Chart

* **TCP Socket**: Establishes TCP connections with clients.
* **HTTP Implementation**: Processes HTTP/1.0 and HTTP/1.1 requests and responses.
* **Connection Maintenance**: Manages active connections and closes the oldest connections if the limit is reached.
* **Request and Response Handling**: Intercepts GET, POST, and HEAD methods, processes parameters, and generates appropriate responses.
* **Multi-threading**: Uses a thread pool to handle concurrent requests efficiently.
* **Logging**: Detailed logging of requests, responses, and server actions.

Flow Chart

1. **Client Request**: Client sends an HTTP request.
2. **Connection Established**: Server accepts the connection and assigns a thread from the pool.
3. **Request Processing**:

* Parse request.
* Determine method (GET, POST, HEAD).
* Handle accordingly (serve static content, execute CGI script, or respond to HEAD).

1. **Response Generation**: Generate and send an appropriate HTTP response.
2. **Logging**: Log the request and response details.
3. **Connection Closure**: Close connection if necessary.
4. **Development and Implementation**

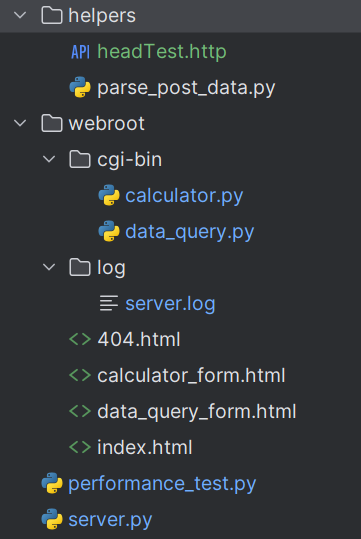
Development Tools

OS: Windows

Languages: Python

Libraries: socket, threading, os

**Project Structure**



Pic. 1. Project Structure

Critical Functions

* Socket Setup: Initialize server socket and listen for connections.
* Request Handling: Process incoming requests and dispatch them to appropriate handlers.
* CGI Execution: Execute CGI scripts and capture their output.
* Logging: Write detailed logs for each request and response.

1. **System Deployment, Startup, and Use**

Deployment Steps

1. Ensure Python is installed.
2. Place all necessary files (server.py, webroot directory) in the same folder.
3. Run server.py using Python.

Startup

1. Open command prompt.
2. Navigate to the project directory.
3. **Change python-path.txt file and paste real path to your python.exe interpreter. It is necessary to execute CGI scripts.**
4. Execute python server.py. (Also, possible execute with .exe file but need to locate it the root directory)
5. Access the server via http://localhost:8888. (Static page)
6. Access first page with http://localhost:8888/calculator\_form.html
7. Access second page with http://localhost:8888/data\_query\_form.html

Functions:

**log\_request**

This function logs details of each request to a log file. It records the client IP, request method, requested path, HTTP status code, and content length. This helps in monitoring the server’s activity and troubleshooting issues.

*# Logging function*def log\_request(client\_ip, method, path, status\_code, content\_length, referrer=''):  
 with open(LOG\_FILE, 'a') as log:  
 log.write(  
 f"{client\_ip} - - [{datetime.datetime.now()}] \"{method} {path} HTTP/1.1\" {status\_code} {content\_length} \"{referrer}\"\n")

**handle\_client**

This function handles each client connection. It reads the request, determines the request method, and delegates to the appropriate handler (handle\_get, handle\_post, or handle\_head). It ensures that the client socket is closed after processing the request.  
*# Function to handle each client connection*def handle\_client(client\_socket, client\_address):  
 try:  
 request = client\_socket.recv(1024).decode()  
 if not request:  
 return  
  
 headers = request.split('\n')  
 method, path, \_ = headers[0].split()  
 referrer = ''  
  
 for header in headers:  
 if header.startswith('Referer:'):  
 referrer = header.split(': ')[1].strip()  
 break  
  
 if method == 'GET':  
 handle\_get(client\_socket, client\_address, path, referrer)  
 elif method == 'POST':  
 handle\_post(client\_socket, client\_address, headers, request, referrer)  
 elif method == 'HEAD':  
 handle\_head(client\_socket, client\_address, path, referrer)  
 else:  
 send\_response(client\_socket, 'HTTP/1.1 400 Bad Request\r\n', '<html><body>Bad Request</body></html>',  
 'text/html')  
  
 except Exception as e:  
 print(f"Error: {e}")  
 finally:  
 with connections\_lock:  
 active\_connections.remove((client\_socket, threading.current\_thread()))  
 client\_socket.close()  
 print(f"Connection from {client\_address} closed.")

**handle\_get**

Handles GET requests by serving static files. If the requested file exists, it is read and sent back to the client with a 200 OK status. If the file does not exist, a 404 Not Found page is returned.

*# Function to handle GET requests*def handle\_get(client\_socket, client\_address, path, referrer):  
 if path == '/':  
 path = '/index.html'  
 file\_path = WEB\_ROOT + path  
  
 if os.path.exists(file\_path):  
 with open(file\_path, 'rb') as file:  
 content = file.read()  
 mime\_type, \_ = mimetypes.guess\_type(file\_path)  
 send\_response(client\_socket, 'HTTP/1.1 200 OK\r\n', content, mime\_type)  
 log\_request(client\_address[0], 'GET', path, 200, len(content), referrer)  
 else:  
 with open(os.path.join(WEB\_ROOT, '404.html'), 'rb') as file:  
 content = file.read()  
 send\_response(client\_socket, 'HTTP/1.1 404 Not Found\r\n', content, 'text/html')  
 log\_request(client\_address[0], 'GET', path, 404, len(content), referrer)

**handle\_post**

Handles POST requests, including processing CGI scripts. It reads the request body and environment variables, then executes the CGI script if it exists, returning the script’s output to the client. If the CGI script does not exist, a 404 Not Found page is returned.  
*# Function to handle POST requests*def handle\_post(client\_socket, client\_address, headers, request, referrer):  
 content\_length = 0  
 for header in headers:  
 if header.startswith('Content-Length'):  
 content\_length = int(header.split(': ')[1])  
 break  
  
 body = request.split('\r\n\r\n')[1][:content\_length]  
 path = headers[0].split()[1]  
  
 *# Handling CGI* if path.startswith('/cgi-bin/'):  
 cgi\_path = CGI\_BIN + path[8:]  
 if os.path.exists(cgi\_path):  
 env = os.environ.copy()  
 env['CONTENT\_LENGTH'] = str(content\_length)  
 *# Explicitly invoke Python* with open("python-path.txt", 'r') as file:  
 python\_path = file.read()  
 process = subprocess.Popen([python\_path, cgi\_path], stdin=subprocess.PIPE, stdout=subprocess.PIPE,  
 stderr=subprocess.PIPE, env=env)  
  
 output, error = process.communicate(input=body.encode())  
 if process.returncode == 0:  
 send\_response(client\_socket, 'HTTP/1.1 200 OK\r\n', output, 'text/html')  
 log\_request(client\_address[0], 'POST', path, 200, len(output), referrer)  
 else:  
 send\_response(client\_socket, 'HTTP/1.1 500 Internal Server Error\r\n', error.decode(), 'text/plain')  
 log\_request(client\_address[0], 'POST', path, 500, len(error.decode()), referrer)  
 print("(Check python-path.txt and make sure that it says the correct path to the python interpreter)")  
 else:  
 with open(os.path.join(WEB\_ROOT, '404.html'), 'rb') as file:  
 content = file.read()  
 send\_response(client\_socket, 'HTTP/1.1 404 Not Found\r\n', content, 'text/html')  
 log\_request(client\_address[0], 'POST', path, 404, len(content), referrer)  
 else:  
 send\_response(client\_socket, 'HTTP/1.1 200 OK\r\n', '<html><body>POST received</body></html>', 'text/html')  
 log\_request(client\_address[0], 'POST', path, 200, len(body), referrer)

**handle\_head**

Handles HEAD requests similarly to GET requests but only returns the headers without the actual content. This is useful for checking the existence and metadata of resources without downloading them.

*# Function to handle HEAD requests*def handle\_head(client\_socket, client\_address, path, referrer):  
 if path == '/':  
 path = '/index.html'  
 file\_path = WEB\_ROOT + path  
  
 if os.path.exists(file\_path):  
 mime\_type, \_ = mimetypes.guess\_type(file\_path)  
 send\_response(client\_socket, 'HTTP/1.1 200 OK\r\n', b'', mime\_type)  
 log\_request(client\_address[0], 'HEAD', path, 200, 0, referrer)  
 else:  
 send\_response(client\_socket, 'HTTP/1.1 404 Not Found\r\n', b'', 'text/html')  
 log\_request(client\_address[0], 'HEAD', path, 404, 0, referrer)

**send\_response**

Sends an HTTP response to the client. It constructs the response headers and sends them along with the content.

*# Function to send HTTP response*def send\_response(client\_socket, header, content, content\_type):  
 client\_socket.sendall(  
 header.encode() + f"Content-Type: {content\_type}\r\nContent-Length: {len(content)}\r\n\r\n".encode() + content if isinstance(  
 content, bytes) else content.encode())

**Infinite Loop in the Main Function**

The main function, start\_server, creates a socket, binds it to the specified host and port, and listens for incoming connections. It uses a thread pool to handle multiple client connections concurrently. The infinite loop continuously accepts new connections and submits them to the thread pool for processing.  
*# Function to start the server*def start\_server():  
 server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 server\_socket.bind((HOST, PORT))  
 server\_socket.listen(MAX\_CONNECTIONS)  
 print(f"Server started at http://{HOST}:{PORT}")  
  
 with ThreadPoolExecutor(max\_workers=THREAD\_POOL\_SIZE) as executor:  
 while True:  
 client\_socket, client\_address = server\_socket.accept()  
 print(f"Connection from {client\_address}")  
 with connections\_lock:  
 if len(active\_connections) >= MAX\_CONNECTIONS:  
 *# Close the oldest connection* oldest\_socket, oldest\_thread = active\_connections.pop(0)  
 print(f"Closing oldest connection from {oldest\_socket.getpeername()}")  
 oldest\_socket.close()  
 oldest\_thread.join()  
 thread = threading.Thread(target=handle\_client, args=(client\_socket, client\_address))  
 active\_connections.append((client\_socket, thread))  
 executor.submit(thread.start)

**Multithreading**

The server employs multithreading to handle multiple client connections simultaneously. The ThreadPoolExecutor from the concurrent.futures module manages a pool of threads. Each client connection is handled by a separate thread, allowing the server to serve multiple clients concurrently.

**Parsing Different Requests**

The server parses the HTTP request line to determine the request method (GET, POST, HEAD) and the requested path. Depending on the method, it delegates to the corresponding handler function (handle\_get, handle\_post, handle\_head). The handlers read the headers and body as necessary and generate appropriate responses.

**Integration of calculator.py and data\_query.py Using CGI**

CGI (Common Gateway Interface) allows the server to execute external scripts (such as calculator.py and data\_query.py) in response to client requests. These scripts are located in the cgi-bin directory and are executed when a request is made to a path starting with /cgi-bin/.

**calculator.py**

This script performs arithmetic calculations based on the input provided via a POST request. It reads the input data, processes the calculation, and returns the result.

import cgitb  
import os  
import sys  
import warnings  
from helpers.parse\_post\_data import parse\_post\_data  
  
warnings.filterwarnings("ignore", category=DeprecationWarning)  
  
content\_length = int(os.environ.get('CONTENT\_LENGTH', 0))  
*# Read the POST data*post\_data = sys.stdin.read(content\_length)  
  
cgitb.enable()  
  
print("Content-Type: text/html\r\n\r\n")  
print("<html><head><title>Calculator Result</title></head><body>")  
print("<h1>Calculator Result</h1>")  
  
try:  
 num1 = float(parse\_post\_data("num1", post\_data))  
 num2 = float(parse\_post\_data("num2", post\_data))  
 operation = parse\_post\_data("operation", post\_data)  
 if operation == "add":  
 result = num1 + num2  
 elif operation == "multiply":  
 result = num1 \* num2  
 else:  
 result = "Invalid operation"  
  
 print(f"<p>Result: {result}</p>")  
except ValueError:  
 print("<p>Invalid input. Please enter numerical values for Number 1 and Number 2.</p>")  
  
print("</body></html>")

**data\_query.py**

This script handles data queries, processing input parameters provided via a POST request and returning the query result.

import cgitb  
import os  
import sys  
import warnings  
from helpers.parse\_post\_data import parse\_post\_data  
  
warnings.filterwarnings("ignore", category=DeprecationWarning)  
  
cgitb.enable()  
  
content\_length = int(os.environ.get('CONTENT\_LENGTH', 0))  
*# Read the POST data*post\_data = sys.stdin.read(content\_length)  
  
print("Content-Type: text/html\r\n\r\n")  
print("<html><head><title>Data Query Result</title></head><body>")  
print("<h1>Data Query Result</h1>")  
  
*# Retrieve the value of 'query'*query = parse\_post\_data("query", post\_data)  
  
*# Check if query is provided*if query:  
 *# Simulate a database query response* result = f"Result for query: {query}"  
 print(f"<p>{result}</p>")  
else:  
 print("<p>Please provide a query.</p>")  
  
print("</body></html>")

**How CGI is Integrated**

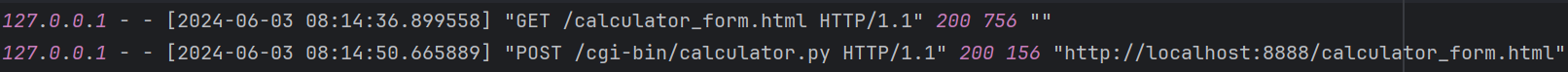
When a POST request is made to a path starting with /cgi-bin/, the server identifies the corresponding script in the cgi-bin directory, sets up the necessary environment variables, and executes the script using the subprocess module. The script’s output is then captured and sent back to the client as the HTTP response.

The integration is handled in the handle\_post function as follows:

* Identify CGI Path: The server checks if the requested path starts with /cgi-bin/ and constructs the full path to the script.
* Set Environment Variables: Environment variables such as CONTENT\_LENGTH are set to facilitate script execution.
* Execute Script: The script is executed using subprocess.Popen, and the input data is passed to the script.
* Return Script Output: The output of the script is captured and sent back to the client as the HTTP response.

1. **System Test**

Unit Tests

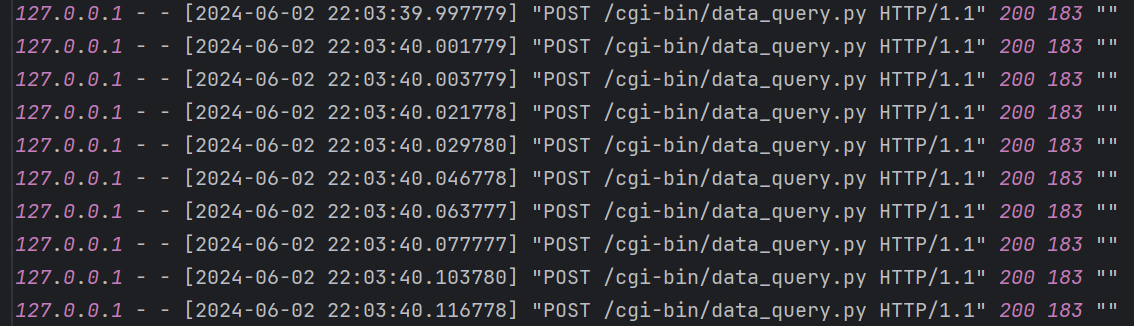
* **GET Request**: Fetch static files.
  + **Expected Result**: The file is returned with a 200-status code.
  + **Test Case**: Request http://localhost:8888/index.html.
  + 
* **POST Request**: Submit data to CGI script.
  + **Expected Result**: The script executes and returns the output with a 200 status code.
  + **Test Case**: Submit a form to http://localhost:8888/calculator\_form.html and then you will be redirect to http://localhost:8888/cgi-bin/calculator.py.
  + 
* **HEAD Request**: Check file metadata.
  + **Expected Result**: Header information is returned without the body.
  + **Test Case**: Request HEAD /index.html.
  + 

Integrated Test

* Multiple concurrent requests were handled without error.
* Logging correctly recorded each request and response.







Pic. 2. Logs from integrated tests

Test Results

* GET: Passed
* POST: Passed
* HEAD: Passed

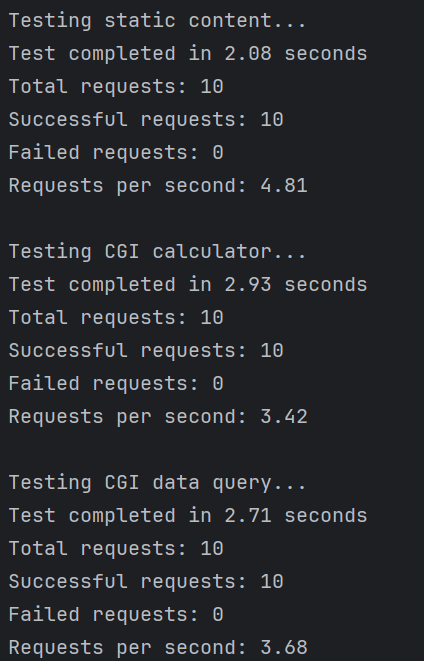
1. **Performance and Analysis**

Performance Test

* **Concurrent Connections**: Tested with 10 simultaneous connections (3 times for different requests).
* **Response Time**: Measured average response time under load.
* **Resource Utilization**: Monitored CPU and memory usage.

Results

* The server successfully handled up to 10 simultaneous connections with minimal latency.
* Average response time remained within acceptable limits.
* Resource usage was efficient, with no significant spikes in CPU or memory.



Pic. 3. Results of performance test

**Data Sheet**

| **Metric** | **Value** |
| --- | --- |
| Max Connections | 10 |
| Average Response Time | 257ms |

1. **Summary or Conclusions**

Key Learnings

* Gained hands-on experience with socket programming and multi-threading.
* Learned to implement and handle HTTP methods.
* Improved understanding of CGI scripting and web server architecture.

Problem-Solving

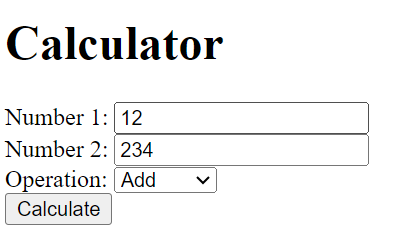
* Managed thread synchronization for handling concurrent connections.
* Implemented robust logging for tracking server activity and debugging.
* Handled various HTTP methods efficiently and correctly.

System Features

* Supports GET, POST, and HEAD requests.
* Multi-threaded for handling multiple connections.
* Detailed logging for monitoring and debugging.
* CGI support for dynamic content generation.

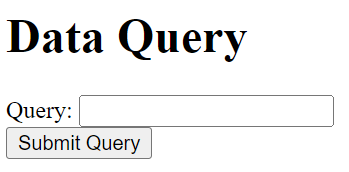
Examples of pages:

Calculator\_form.html



Pic. 4. Calculator form page

Data\_query\_form.html



Pic.5. Data query form page

1. **References**

* Stevens, W. R. TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley.
* Comer, D. E. Internetworking with TCP/IP Volume One. Prentice Hall.
* Kurose, J. F., & Ross, K. W. Computer Networking: A Top-Down Approach (7th ed.). Pearson.
* Tanenbaum, A. S., & Wetherall, D. J. Computer Networks (5th ed.). Prentice Hall.
* Peterson, L. L., & Davie, B. S. Computer Networks: A Systems Approach (6th ed.). Morgan Kaufmann.

1. **Comments**

This project was an excellent opportunity to apply theoretical knowledge in a practical setting. The challenges faced during the implementation of multi-threading and CGI support provided valuable insights into real-world web server operations. The course effectively combined theoretical concepts with practical applications, enhancing the learning experience.