|  |  |  |
| --- | --- | --- |
|  | Alexandria University  Faculty of Engineering  Computer & Systems Engineering  CSE352: Computer Networks | A logo of a globe  Description automatically generated |

**Introduction to Socket Programming in C/C++**

**HTTP Reboot**

|  |  |
| --- | --- |
| **Student ID** | **Student Name** |
| 20011800 | Mahmoud Tarek Mahmoud Embaby |
| 20012180 | Hesham Youssef Ahmed Abo El-Qamasan |

**1. Program Organization**

**1.1. Program Architecture**

* Architecture Pattern: Client-Server Model

**Client**: The web client sends HTTP requests and handles responses.

**Server**: The server listens for requests, processes them and sends back HTTP responses.

* Components:
* *For the Web Server*

**Concurrency Model**: It uses a multi-threading approach, where each connection is handled in a separate thread, allowing for concurrent processing of client requests.

**Connection Handling**: It has a mechanism to handle persistent connections as per HTTP/1.1 specifications and close them after a dynamically calculated timeout or when an error occurs.

**Request Processing**: It can process GET and POST requests and respond with appropriate HTTP status codes and messages.

**Dynamic Timeout:** It adjusts the timeout for each connection dynamically based on the number of active connections.

**Error Handling:** It has functions to handle invalid requests and server overload situations, sending corresponding HTTP responses.

**Logging:** A simple logging mechanism is used to log server activity and connection statuses.

* *For the Web Client*

**Client-Side Communication**: Establishes TCP connections with a web server and handles network operations such as connecting, sending and receiving data over sockets.

**Request Handling**: Constructs HTTP requests based on user commands.

**Response Processing**: Parses the HTTP response to extract the headers and content.

**Error Handling**: Includes based error handling for socket operations and file I/O.

**2.2. Directory Structure**

- client/: Contains the web client source code files.

- server/: Contains the web server source code files.

- tests/: Contains the test scripts.

**2.3. Code Organization**

*Web Server*

|  |  |
| --- | --- |
| **Method Name** | **Function** |
| connection\_handler | Handles persistent connections as per HTTP/1.1 specifications and close them after a dynamically calculated timeout or when an error occurs. |
| handle\_get\_request | Handles GET requests, reads the local file, processes and the appropriate HTTP response |
| handle\_post\_request | Handles POST requests, writes the content to a local file, processes and the appropriate HTTP response |
| handle\_invalid\_request | Returns appropriate HTTP error codes and messages. |
| handle\_server\_overload | Returns error 503 (Service Unavailable) with Retry-After header, used when number of active connections exceeds the max server capacity. |
| calculate\_timeout | Dynamically calculates the timeout in seconds based on the heuristic explained below. |
| read\_next\_block | Reads POST content in blocks, one at a time. |
| mytok | Custom string tokenizer |
| write\_token\_to\_file | Writes token to a file |
| increment\_connection\_count | Increments the connection count while using semaphores. |
| decrement\_connection\_count | Decrements the connection count while using semaphores. |
| logger | Logs server actions (Connections, Requests) to external file. |

*Web Client*

|  |  |
| --- | --- |
| **Method Name** | **Function** |
| process\_commands | Processes user commands from external file `input.txt` and directs the commands to specified handlers. |
| handle\_get\_request | Sends GET requests to a server, waits for the response, reads the content and stores it in a local file. |
| handle\_post\_request | Reads local file content and sends POST requests to a server. |
| connect\_to\_server | Establishes a TCP connection with a server |
| getContentLength | Parses Content-Length value from server response headers |

**2.4. Data Structures**

|  |  |
| --- | --- |
| **Data Structure** | **Usages** |
| Standard Types and Structures | char[]  FILE\*  struct sockaddr\_in  pthread\_t  pthread\_mutex\_t |
| Socket Programming Types | Int sockfd |
| Dynamic Memory | char\* (pointer to char) |
| Select System Call | fd\_set  struct timeval |

**2.5. Input/Output**

*Web Server*

* Writes the server logs to `logs.txt` file.

*Web Client*

* Reads user-specified commands from `input.txt` file with the following command structure:

GET Command: client\_get file-path host-name (port-number)

POST Command: client\_post file-path host-name (port-number)

**2. Program Design**

**2.1. Using multi-threading rather than multi-processing**

The web server uses multi-threading for handling multiple requests from different users. For each open socket, a thread is assigned with persistent connection.

The reasons for choosing multi-threading rather than multi-processing are:

1. Simplicity, handles blocking I/O well and utilizes multi-core processors.
2. Threads are usually cheaper than processes.
3. To start up a process, you need to duplicate all the resources of the parent process. When it terminates, everything has to be torn down.

**2.2. Dynamic Timeout Heuristic**

A mechanism to dynamically manage connection timeouts based on server load.

* If the server is idle, it allows connections to remain open for a longer period.
* If the server is busy, it shortens the time an idle connection is kept open to free up resources.

The heuristic function depends on the system specifications and requirements. Therefore, we designed a function that can dynamically assign timeout periods based on 3 factors:

* *Server Config*: MAX\_TIMEOUT
* *Server Config*: MAX\_CONNECTIONS
* *Current State*: ACTIVE\_CONNNECTIONS

Final Heuristic Function:

Timeout = MAX\_TIMEOUT – (MAX\_TIMEOUT / MAX\_CONNECTIONS) \* ACTIVE\_CONNECTIONS

**2.3. Configurations**

**MAX\_TIMEOUT** and **MAX\_CONNECTIONS** are constants that can be modified based on the server hardware specification and system admins requirements.

**OVERLOAD\_RETRY\_AFTER** is the value set to Retry-After header when server returns error 503 due to server overload (Max connection capacity reached). This value asks the user to retry establishing the connection after X seconds.

**3. Testing**

**3.1. Basic Testing**

|  |  |
| --- | --- |
| **Client Request** | **Server Response** |
| GET /file.txt HTTP/1.1 | HTTP/1.1 200 OK  Content-Length: 11  Hello World |
| GET /file2.txt HTTP/1.1 *(DOESN’T EXIST)* | HTTP/1.1 404 Not Found |
| POST /file2.txt HTTP/1.1 | HTTP/1.1 200 OK |

Establishing a connection from a web browser (Safari) to fetch HTML content and 2 images.

A screenshot of a web page

Description automatically generated

**3.2. Stress Testing**

The following stress tests are performed with the following configuration:

*Server Settings:*

MAX\_CONNECTIONS set to **100**.

MAX\_TIMEOUT set to **30**.

*Source Settings:*

REQUEST\_INTERVAL set to **0.1**

NUM\_CLIENTS set to **200**

Please refer to **log-test1.txt** to see the test results.

1. **Performance Evaluation**

**TO BE DONE SOON**

1. **References**

* Online: “Beej’s Guide to Network Programming: Using Internet Sockets”
* ‘TCP\IP Sockets in C’ book