CS431: Computer Communications and Networks

Programming Assignment 1

Introduction to Socket Programming in C/C++

Team

Ahmed Yakout	7
Fares Mehanna	52
Dahlia Chehata	27

Table Of Contents

I. Overview

Simple http web application implementation where a web client communicates with a web server using a restricted subset of HTTP and UNIX sockets

II. Modules Organization and Documentation

A. http_server

1. types manager

a) **std::string get_file_type (const std::string& file_name);**
return the file type from the file extension

b) **bool is_file_supported (const std::string& file_name);**
supported types are “txt”, “jpg”, ”html”, ”js”, ”css”, ”png”and ”ttf”

c) **std::string generate_file_header (const std::string& file_name);**
return the “Content-type: “ sentence

2. SOCK_RAII

a) **sock_RAII(int sock_fd): initialize the socket file descriptor**

b) **~sock_RAII():   close the socket**

3. http_server_1.0 / 1.1

a) **void handle_request(http_req_handler& request);**
increments request number, checks the method if it was supported or not,s
handles get requests by:

- creating path to the file and handling index files
- check if supported file first, try to open the file if exists
- else treat it as directory and try again , if it's a directory, then redirect the user to the dir url
- gets the file dimensions and reads data from file

handles post requests by:

- checking for “The Content-Length “ value and if the file is supported
- acknowledges the request in case of success

b) **void handle_data(const uint8_t* data, uint32_t data_length);**
increments request number and the server gives data less than wanted then gets the full path to the file.
builds directories, creates the file and writes data to file, reset the http version state and send the acknowledgment

c) **`bool is_end();`** `check if there is no requests left`
d) **`int bypass_request_checking();`** `return the expected post`
e) **`http_server_v1_0(int sock_fd);`** `initializes the socket file descriptor to 0 as well as the request number, the expected post and the post url`

4. http_responder

a) **`http_responder(int sock_fd);`**
`set the socket file descriptor with the given parameter`
b) **`~http_responder();`** `free data_`
c) **`http_responder* set_http_version (const std::string& http_version);`** `sets the http version with the given parameter`
d) **`http_responder* set_http_statuscode (const std::string& status_code);`** `sets the status code with given value`
e) **`http_responder* add_header (const std::string& header);`**
`insert new header line in the headers vectors`
f) **`http_responder* set_data (const std::string& data);`**
`set data with passed value`
g) **`http_responder* set_data (const uint8_t* data, int size);`**
`allocate the passed value of size and set data`
h) **`bool send_response();`** `sends the server response header + data in a segment format and initialize the environment variables`
i) **`void initialize_data();`** `resets the following parameters`
- `headers`
- `http_version`
- `status_code`
- `data`
- `valid = true`
- `data_size`

j) **`bool totally_send(const char* data, int data_size);`**
`send data with a specified size`

5. http request handler

a) **`void parse_request();`**
`parses the request and set the value of the fields`
b) **`http_req_handler(const std::string& request);`** `set the request with passed parameter and performs the parsing`
c) **`~http_req_handler();`** `default destructor`
d) **`HTTP_METHOD get_http_method();`** `return http method`
e) **`HTTP_VERSION get_http_version();`** `return http version`
f) **`std::string get_http_url_target();`** `return http url`
g) **`std::unordered_map<std::string, std::string> get_headers();`**
`return headers list`
h) **`std::string get_header_value(const std::string& header_type);`**
`return a specified header value`

6. http connection main handler

a) **`void handle_connection(int sock_fd);`** This function will be called from the pool-worker threadt o handle the connection.

Algorithm:

```
while(1)
       receive the request or multiple requests from the client
       if server didn't receive any input for long period of time return
       if client closed the connection or error happened return
       get the HTTP requests out of the received data
       fix the start of the buffer and the buffer itself for the next read
       transform each valid string request to object
       detect malformed requests
       start executing the requests based on HTTP_VERSION
       detect end of connection from http server
       detect POST request
end
```

b) **`static vector<string> sub_requests (const uint8_t* received_data, uint16_t data_size)`**
this function will take as a parameter the received data from the client and try to divide it to multiple requests which every request is terminated by CRLF.

c) **`static uint32_t total_data_vector_length (const vector<string>& data_list) ;`**
this function return total length of every string in the provided vector

d) **`static vector<http_req_handler> handle_requests (const vector<string>& requests_string);`**
this function convert a vector of requests represented as strings to vector of requests represented as http_request_handler objects.

e) **`static bool handle_post_request(int sock_fd, http_server* curr_server, uint32_t post_size) ;`**
return if the server has successfully received the data or not

B. http_client

a) **`std::vector <std::string> split_cmd(std::string command)`**
`divide the GET/POST request to extract the method type, the filename and the hostname`

b) **`std::ifstream::pos_type file_size(const char* filename)`**
`return the file size value for The Content-Length header or -1 if the file couldn't be opened`

c) **`bool rcv_ack_from_server (const int sockfd);`**
`returns code if the status code is 200 , false otherwise`

d) **`void rcv_file_from_server(const int sockfd,const std::string &filename);`**
`receives file from server in response to the GET request chunk by chunk`

e) **`void send_file_to_server(const int sockfd,const std::string &filename);`**

- `establish the connection between file and file descriptor read only`
- `fstat() stats the file pointed to by file descriptor fd and fills in buffer file_stat.`
- `get the file_stat size`
- `send file size`
- `send file to server in response to POST method chunck by chunk`

f) **`void send_header_line(const int fd,const std::string &method, std::string &filename);`**
send the header in the format:

```
Method \s filename \s HTTP version \r\n
Content-type:                     \r\n
Content-Length:                   \r\n
\r\n
```

g) **`int main(int argc, char *argv[])`**
Algorithm

```
for each command in commands.txt:
   establish TCP connection
   split the command
   send header line and waits for ACK
   case "GET":
     if ACK : receive file from server
     else   : 404 ERROR
   case "POST":
     if ACK : send file to server
     else   : POST ERROR
   close TCP connection
close the file
```

C. Multithreading handling and Main program

1. connection pool

a) **`inline conn_pool::conn_pool(size_t threads) : accept_requests(true)`**
`apply mutex lock for each request while being served`

b) **`inline conn_pool::~conn_pool()`**
`unlock the mutex, notify all threads and join`

c) **`auto conn_pool::enqueue(F&& f, Args&&... args)`**
`allows enqueuing of multiple requests while one being served`

2. Connection worker

a) **`conn_worker();`** `default constructor`

b) **`~conn_worker();`** `default destructor`

c) **`void join();`** `thread join`

3. logger

a) **`static void print_welcome_message()`**

```
"+--------------------------------------------------+"
 "|                                                  |"
 "| WELCOME TO YAWS SERVER (v0.1)  |"
 "|                                                  |"
 "+--------------------------------------------------+"
```

b) **`static void log_new_request(const std::vector<std::string>& requests)`**
`This function should log the incoming requests to the server.`

- **log format:**
  `[IP HOST] -=- [dd/mm/yyy hh:mm:ss] "[REQUEST START LINE]" [RESPONSE STATUS CODE]`
- **example:**
  `127.0.0.1 -=- [24/Feb/2014 10:26:28] "GET / HTTP/1.1" 200`

c) **`static void log_new_connection(char* cline_ip)`**
`print new connection from client ip`

4. yaws_server

a) **`static void sigchld_handler(int s);`**
`waitpid() might overwrite errno, so we save and restore it:`

b) **`static void *get_in_addr(struct sockaddr *sa);`**
`get sockaddr, IPv4 or IPv6`

c) **`yaws_server(char** args);`**`initiates new connection pool`

d) **`~yaws_server();`** `close connection`

e) **`int init( void );`**

- get address information for the server
- loop through all the results and bind to the first
- Associate server with port number on localhost
- listen and wait for incoming connections.
- BACKLOG is the number of connections allowed in the incoming queue.Any new connection will wait in the queue until we accept() it
- Reaping zombie processes as child process exit.

f) **`int run( void );`**

- Main server loop
- **Algorithm**

```
init();
print_welcome_message();
while (1)
       *Accept method  will return new socket file
       descriptor to use for this single connection, so
       we can use recv() and send().

       *log_new_connection(client_ip_address)

       *if pool: enqueue

       *to handle multiple clients we use fork.
       when there is new connection, we accept it and
       fork a child process to handle it.
end loop
```

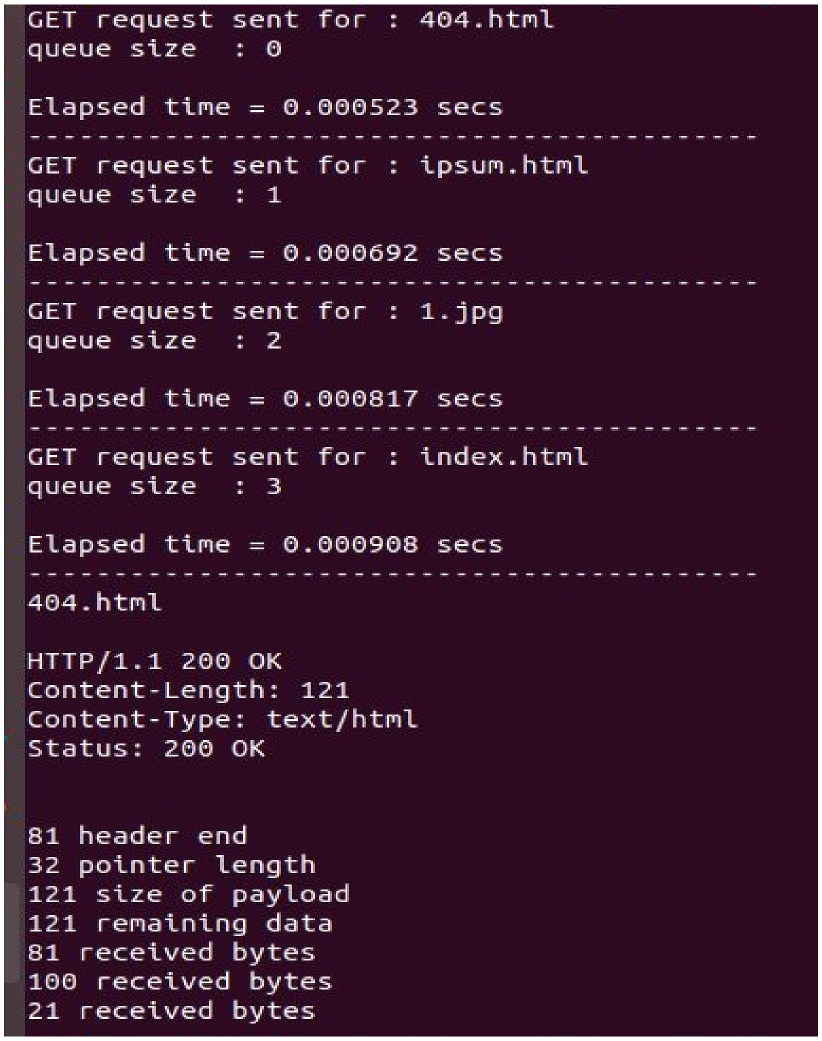
5. main

a) `initiate new object from yaws_ server and call run method`

III. HTTP1.1 Pipelining and timeout

In **timeout** a simple mechanism was used to close the connection in HTTP/1.1.
Every connection get a 15 seconds total time for every request the client might be interested in.
If the server was congested with requests, the time limit decreases for all threads to 5 seconds.
Once enough number of connections were closed, the limit return back to 15 seconds.

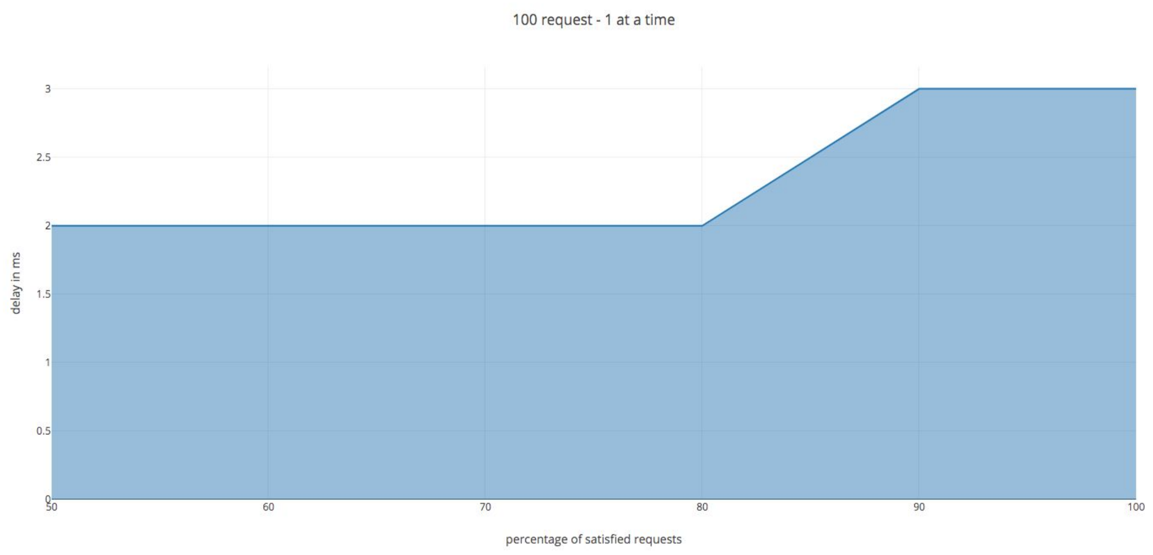
In pipelining we wrote different class to handle HTTP/1.1 with the assumption that only we will send GET requests as the required mechanism to handle POST request is not pipelining friendly.
In the client two threads were created, the sender thread and the receiver thread.
The sender will keep sending the requests in the file and the receiver will keep receiving until they both end.

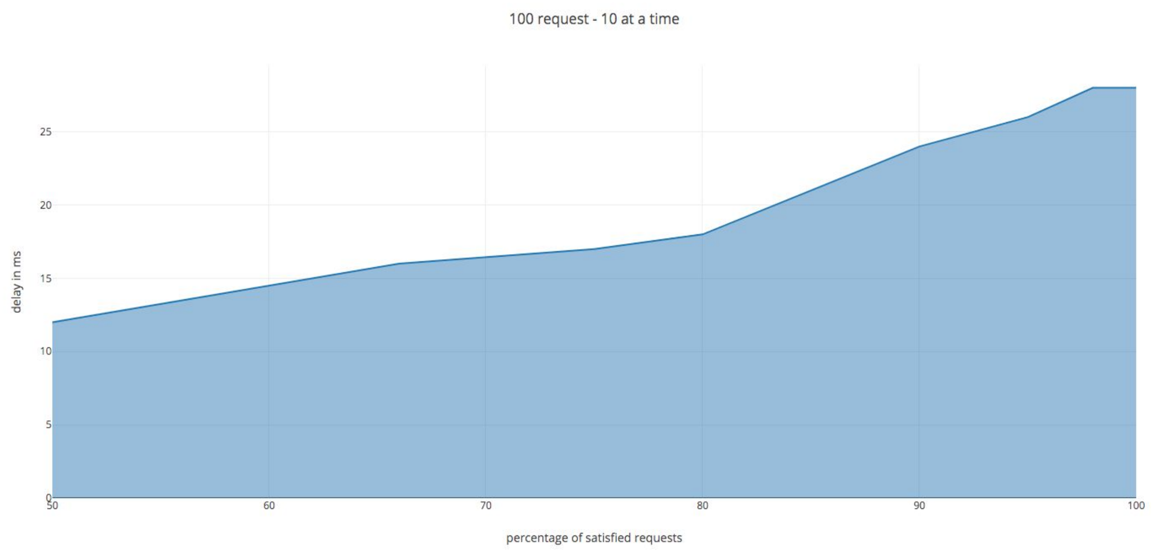


```
GET request sent for : 404.html
queue size  : 0

Elapsed time = 0.000523 secs
-------------------------------------------
GET request sent for : ipsum.html
queue size  : 1

Elapsed time = 0.000692 secs
-------------------------------------------
GET request sent for : 1.jpg
queue size  : 2

Elapsed time = 0.000817 secs
-------------------------------------------
GET request sent for : index.html
queue size  : 3

Elapsed time = 0.000908 secs
-------------------------------------------
404.html

HTTP/1.1 200 OK
Content-Length: 121
Content-Type: text/html
Status: 200 OK


81 header end
32 pointer length
121 size of payload
121 remaining data
81 received bytes
100 received bytes
21 received bytes
```

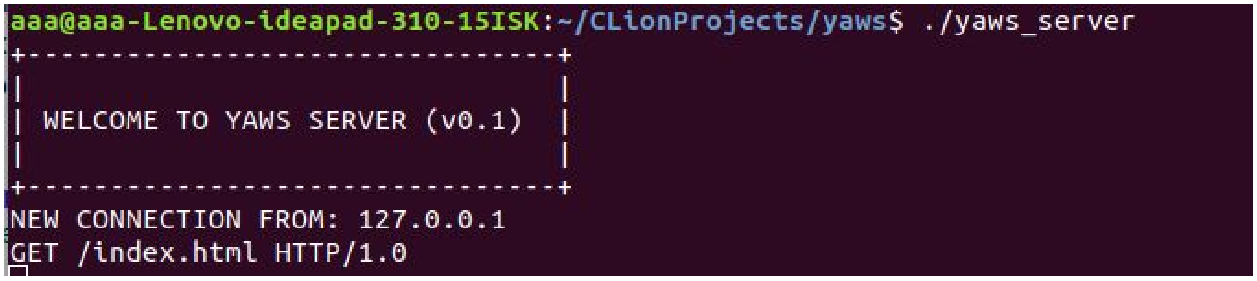
IV. Performance graphs

We used “ab” tool from Apache to benchmark our server, then we plotted the results.

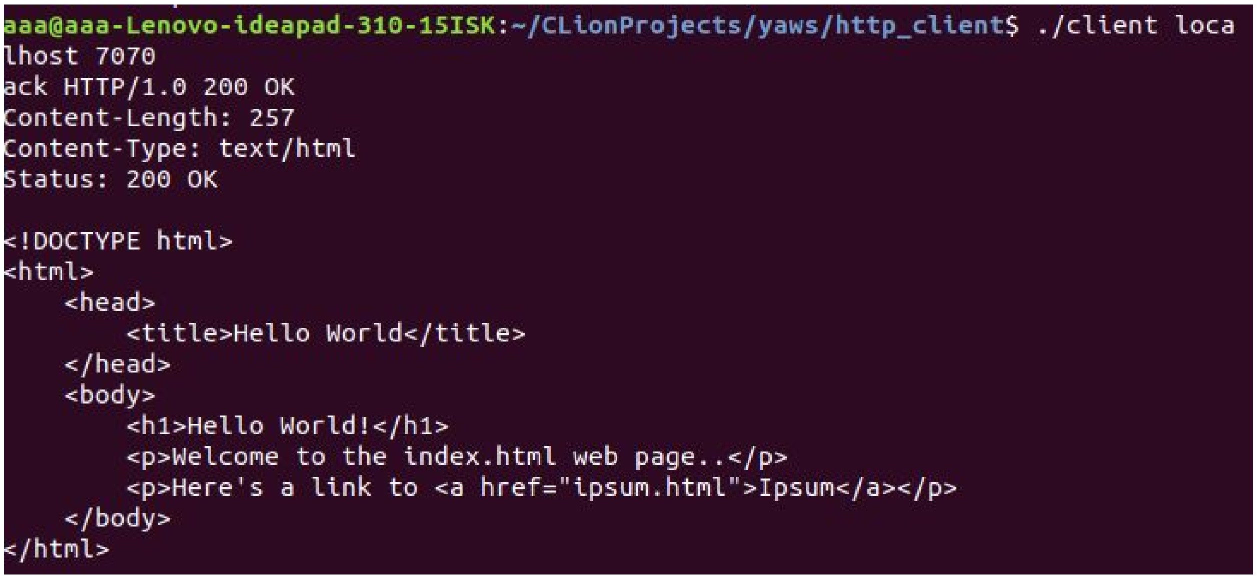




V. Screenshots



```
aaa@aaa-Lenovo-ideapad-310-15ISK:~/CLionProjects/yaws$ ./yaws_server
+--------------------------------+
|                                |
| WELCOME TO YAWS SERVER (v0.1)  |
|                                |
+--------------------------------+
NEW CONNECTION FROM: 127.0.0.1
GET /index.html HTTP/1.0
```



```
aaa@aaa-Lenovo-ideapad-310-15ISK:~/CLionProjects/yaws/http_client$ ./client loca
lhost 7070
ack HTTP/1.0 200 OK
Content-Length: 257
Content-Type: text/html
Status: 200 OK

<!DOCTYPE html>
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <h1>Hello World!</h1>
        <p>Welcome to the index.html web page..</p>
        <p>Here's a link to <a href="ipsum.html">Ipsum</a></p>
    </body>
</html>
```