Faculty of Engineering, Alexandria University

Computer and Systems Engineering Department

Computer Networks and Communications: Fall 2018

Assignment 1

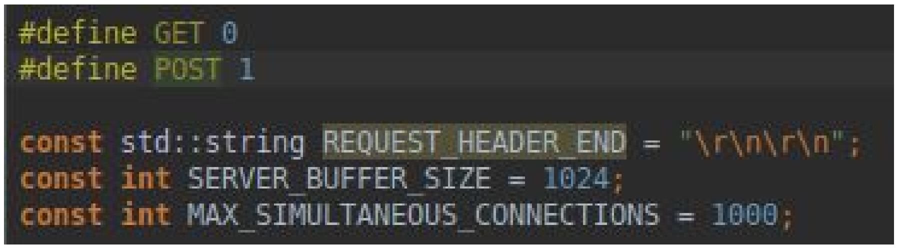
Introduction to Socket Programming

	Name	SID
(1)	Aya Aly	2
(2)	Bishoy Nader	21
(3)	Marc Magdi	55

Part 1: Server side

- *Code Organization:*

    1. Constants are gathered in server_constants.h

    ```
    #define GET 0
    #define POST 1

    const std::string REQUEST_HEADER_END = "\r\n\r\n";
    const int SERVER_BUFFER_SIZE = 1024;
    const int MAX_SIMULTANEOUS_CONNECTIONS = 1000;
    ```

    2. Request_handler.cpp

        Responsible with dealing with client's request whether it's a post or a get request.

    3. Request_parser.cpp

        Build the head map for the accepted requests.
        Provide helper functions to parse accepted requests.

    4. Socket_manager.cpp

        Handles creation of server's socket file descriptor for each accepted client.

    5. Timeout_manager.cpp

        Update timeout for open sockets depending on the percentage of the active clients to maximum number of allowed connections.

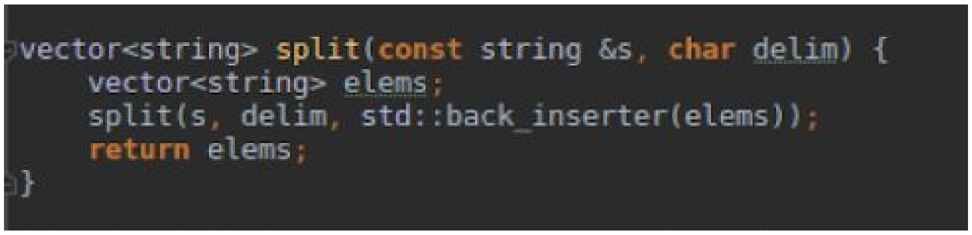
- *Major Functions:*

    1. Get_socket_fd

        Creates a socket file descriptor for the server and binds it with a specific IP address.

2. Split

   Splits the request by a delimiter.



```
vector<string> split(const string &s, char delim) {
    vector<string> elems;
    split(s, delim, std::back_inserter(elems));
    return elems;
}
```

3. handle_request

   Handling new requests coming to server, ensuring persistent connections for accepting multiple requests through same connection.

4. Get_response_header

   Helper method for handling request, Process first part of the request after reading the header.

5. Get_file_name

6. Get_headers_map

7. update_timeout

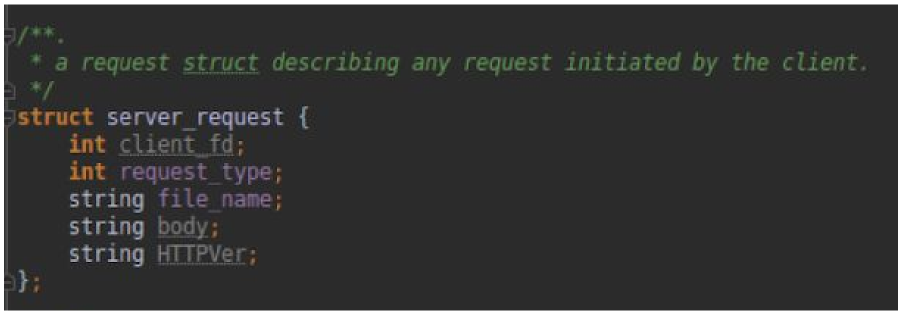
   Update the timeout for all opened sockets, by using this equation

   $(3 * \frac{MAX\_SIMULTANEOUS\_CONNECTIONS}{number\ of\ open\ sockets + 1}) + 1$

   The function is synchronized through mutex.

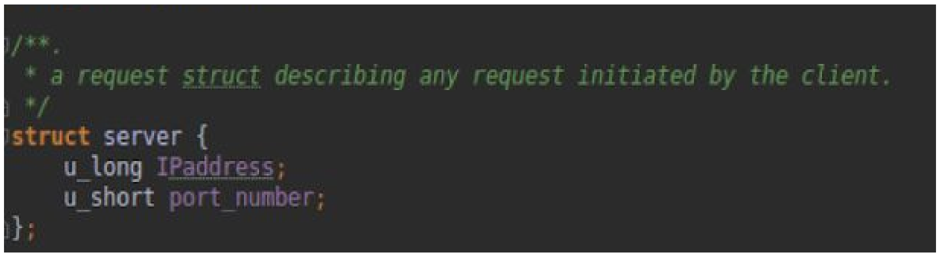
- *Data Structures:*

  1. Struct *request* for Request Data



```
/**
 * a request struct describing any request initiated by the client.
 */
struct server_request {
    int client_fd;
    int request_type;
    string file_name;
    string body;
    string HTTPVer;
};
```

2. Struct *server* for Server Info



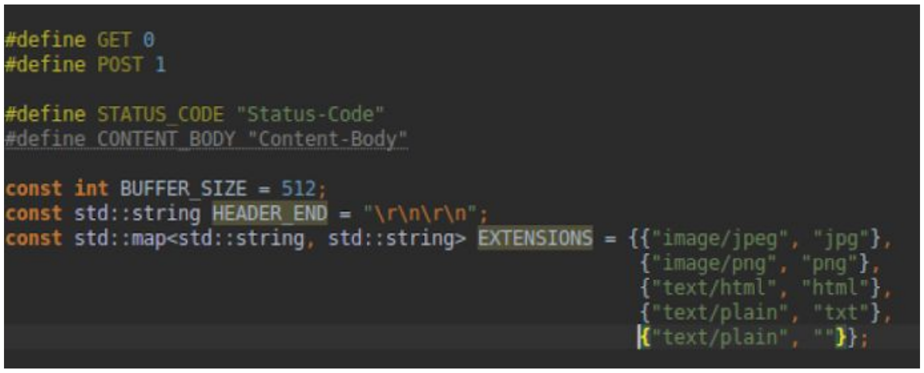
```
/**
 * a request struct describing any request initiated by the client.
 */
struct server {
    u_long IPaddress;
    u_short port_number;
};
```

We chose to make the server Multi-threaded not Multi-process as threads are lighter than processes and share the same address space, also passing data doesn't need message passing. Each Client will have a serving thread with a limit on the number of concurrent active threads.

Part 2: Client side

- *Code Organization:*

1. Constants are gathered in constants.h



```
#define GET 0
#define POST 1

#define STATUS_CODE "Status-Code"
#define CONTENT_BODY "Content-Body"

const int BUFFER_SIZE = 512;
const std::string HEADER_END = "\r\n\r\n";
const std::map<std::string, std::string> EXTENSIONS = {{"image/jpeg", "jpg"},
                                                       {"image/png", "png"},
                                                       {"text/html", "html"},
                                                       {"text/plain", "txt"},
                                                       {"text/plain", ""}};
```

2. Input_reader.cpp

   Responsible for opening and reading input file.

3. Request_parser.cpp

   Parses the request to obtain file name, port number, hostname, request type.

4. Sender.cpp

   Responsible with dealing with server whether in post or get request.

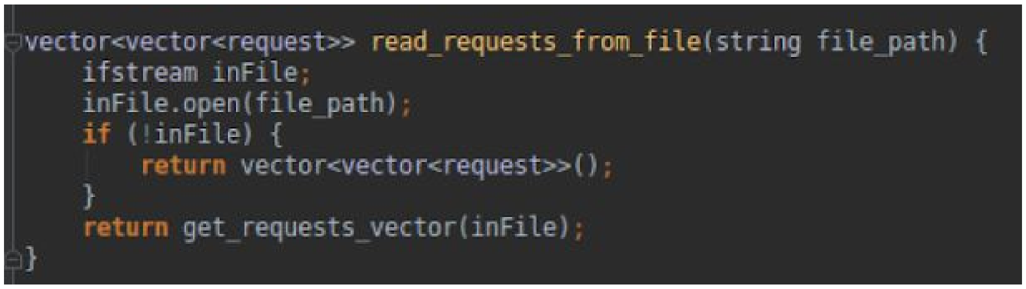
5. Sockets_manager.cpp

   Connects a client’s socket file descriptor to server with the required host name and port number.

- *Major Functions:*

1. read_requests_from_file

   Opens the input file to start reading the requests.



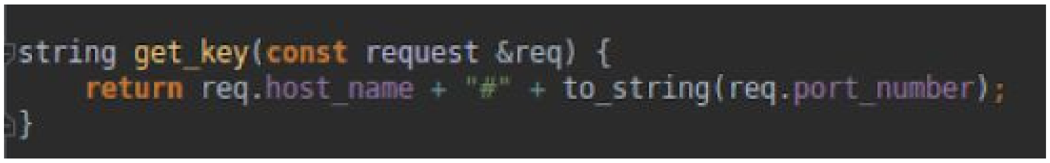
```
vector<vector<request>> read_requests_from_file(string file_path) {
    ifstream inFile;
    inFile.open(file_path);
    if (!inFile) {
        return vector<vector<request>>();
    }
    return get_requests_vector(inFile);
}
```

2. Get_requests_vector

   Reads input file line by line and returns a vector<vector<request>>.

3. Get_key

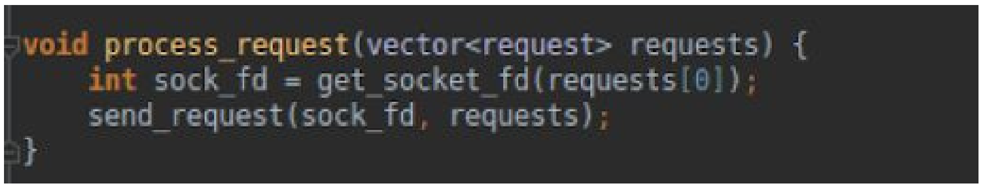
   Creates a request key in the formate HostName#PortNumber.



```
string get_key(const request &req) {
    return req.host_name + "#" + to_string(req.port_number);
}
```

4. Process_requests

   Process each request using the socket fd created.



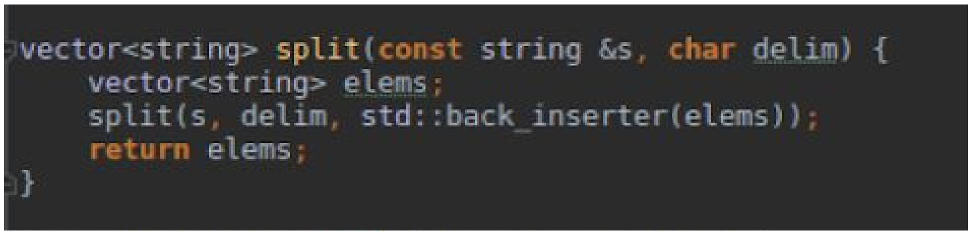
```
void process_request(vector<request> requests) {
    int sock_fd = get_socket_fd(requests[0]);
    send_request(sock_fd, requests);
}
```

5. Get_socket_fd

   Returns a client's socket file descriptor that is connected to a desired server.

6. Split

   Splits the request by a delimiter.



```
vector<string> split(const string &s, char delim) {
    vector<string> elems;
    split(s, delim, std::back_inserter(elems));
    return elems;
}
```

7. Parse_request

   Extracts file name, port number, hostname, request type out of a request.

8. Process_data

9. Process_header

10. Send_request

    Determines if post is get or post and calls appropriate method upon determining.

11. send _post_request

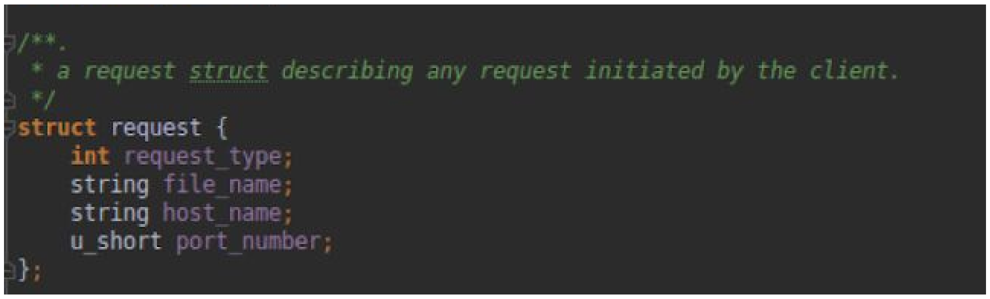
    Responsible for sending the request if -post- to server.

12. Send_get_request

    Responsible for sending the request if -get- to server.

- *Data Structures:*

  1. Struct *request* for Request Data



```
/**
 * a request struct describing any request initiated by the client.
 */
struct request {
    int request_type;
    string file_name;
    string host_name;
    u_short port_number;
};
```

  2. vector<vector<request>>
     Contains the requests read from the input file and processed later.

Part 3: Bonus

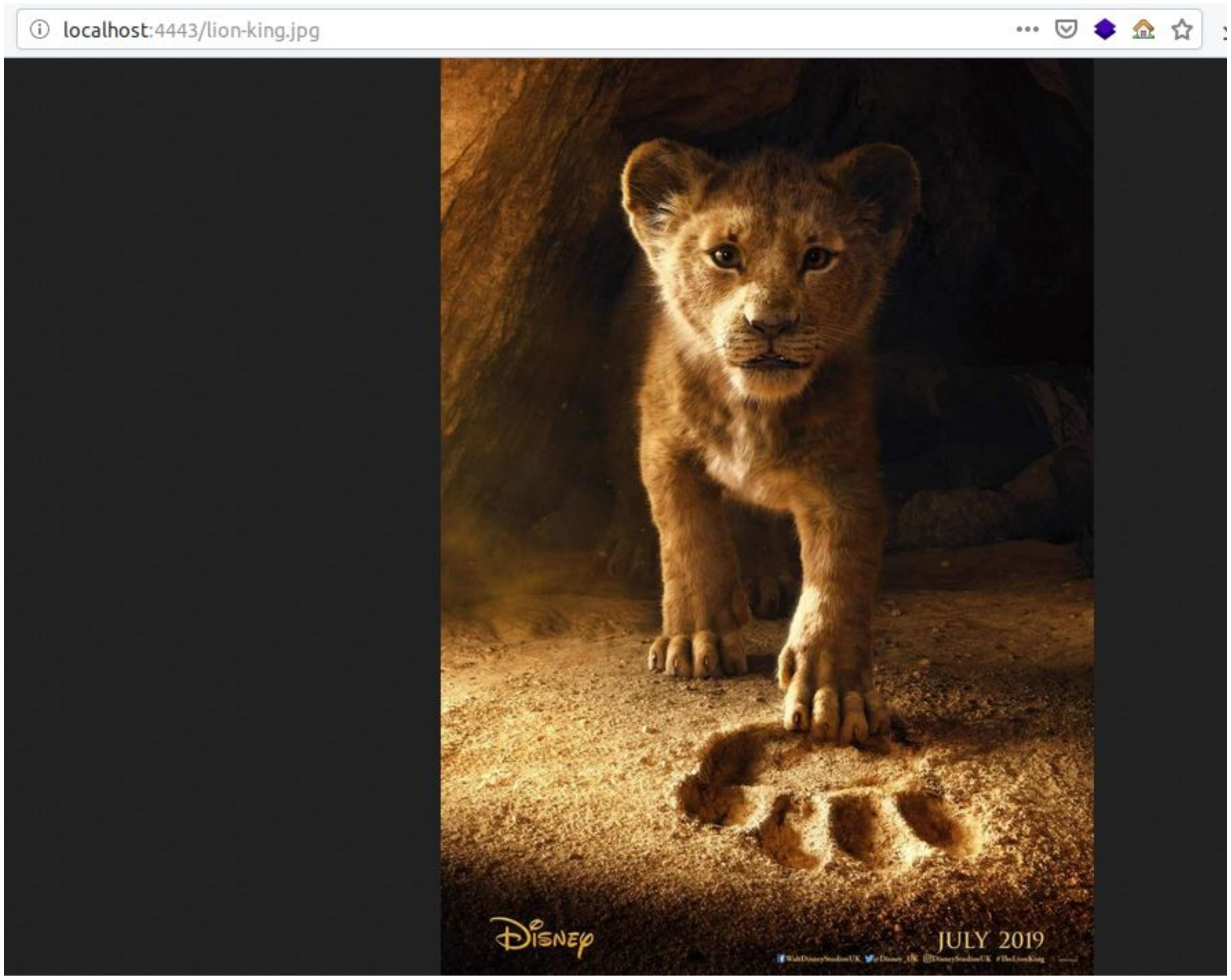
- *Testing with Real Browser*

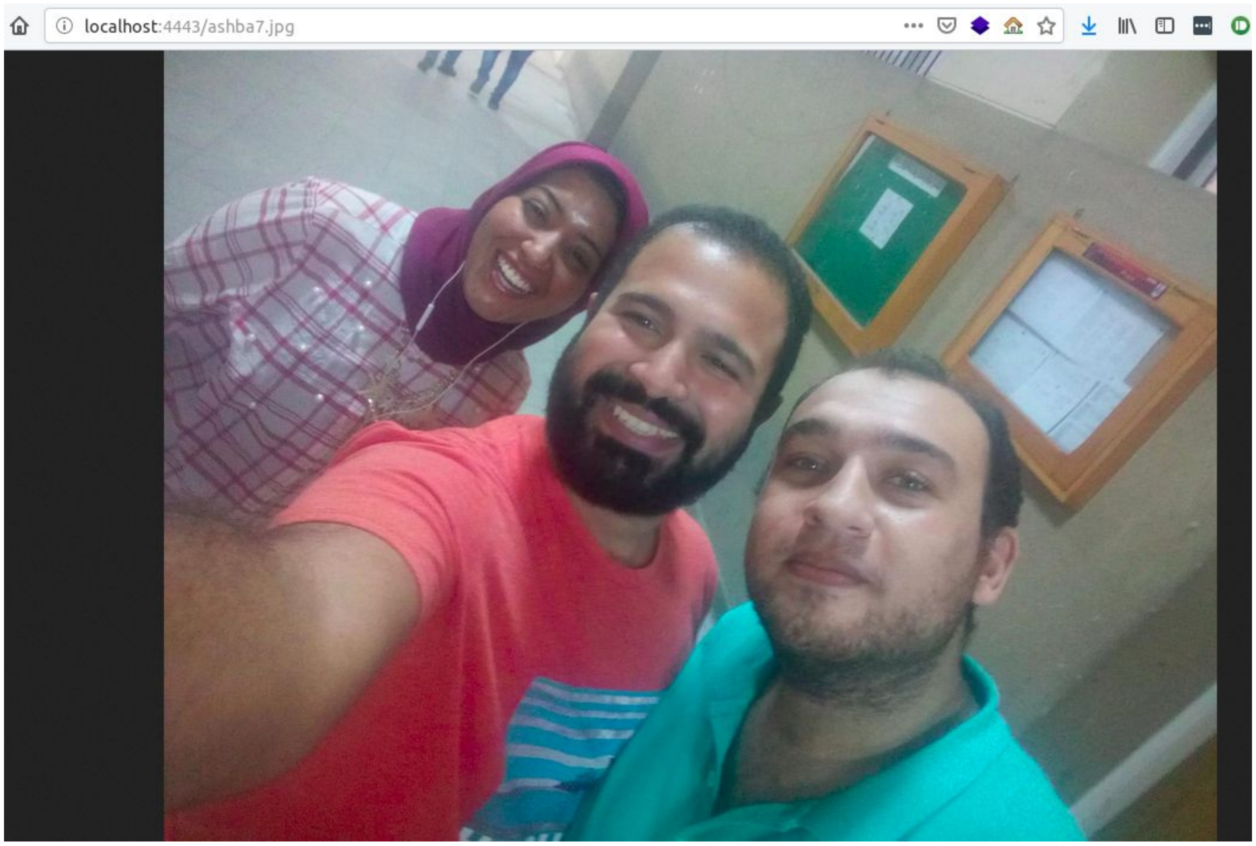
  Client was tested with Henry's Post Test Server V2 which is a service for developers testing clients that POST and GET things over HTTP.

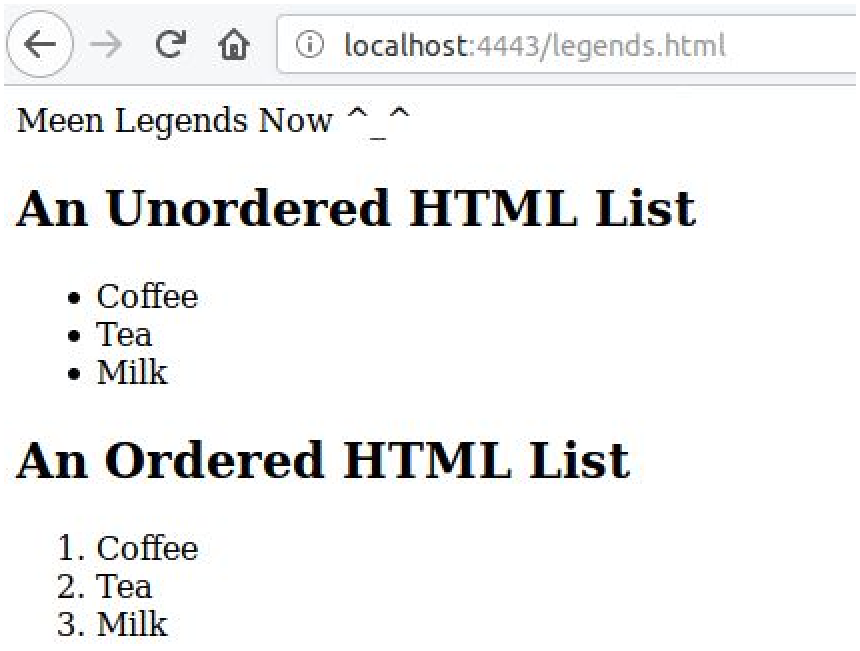
  Using input file:

```
GET /t/jtioq-1542455122/post ptsv2.com
GET /t/jtioq-1542455122/post ptsv2.com
GET /t/jtioq-1542455122/post ptsv2.com
```

Server was tested with Firefox







Meen Legends Now ^_^

An Unordered HTML List

- Coffee
- Tea
- Milk

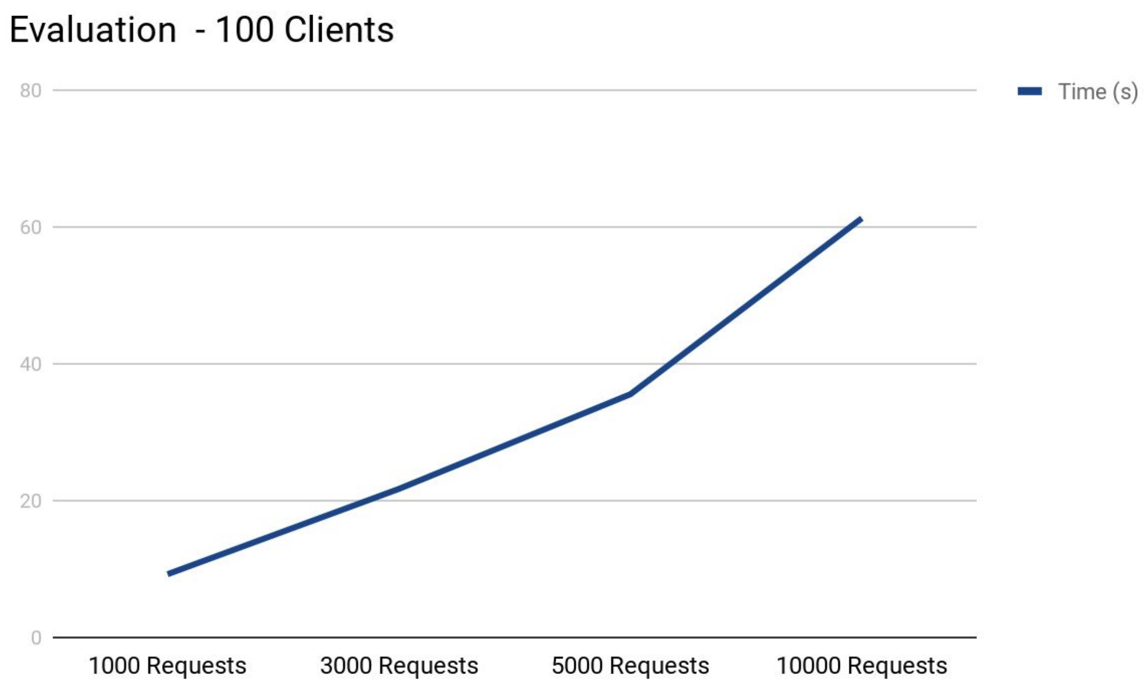
An Ordered HTML List

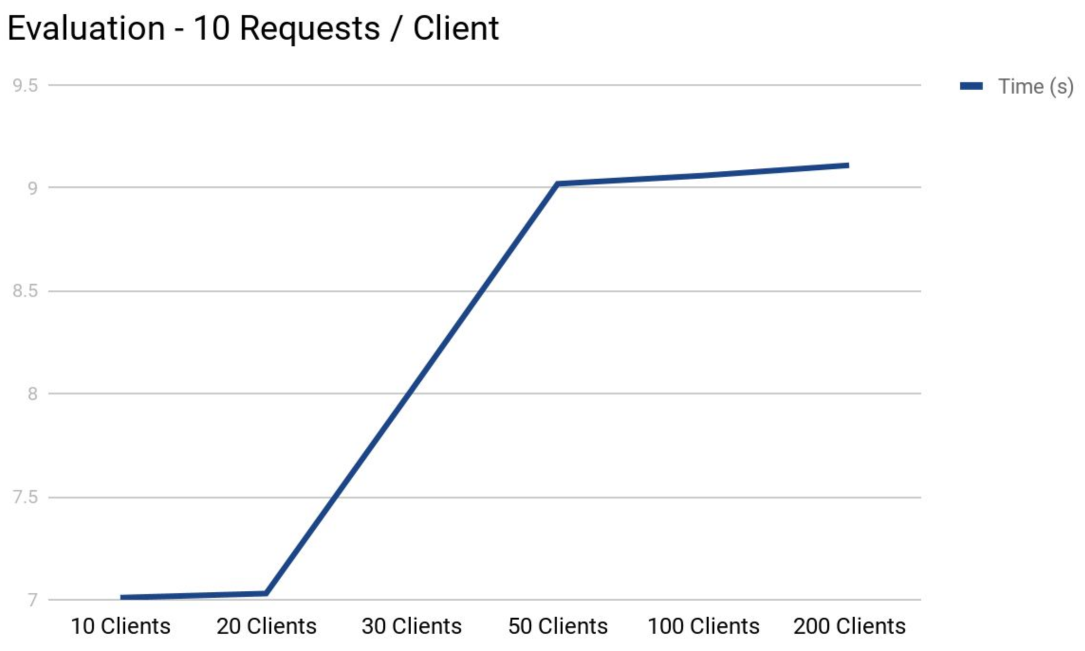
1. Coffee
2. Tea
3. Milk

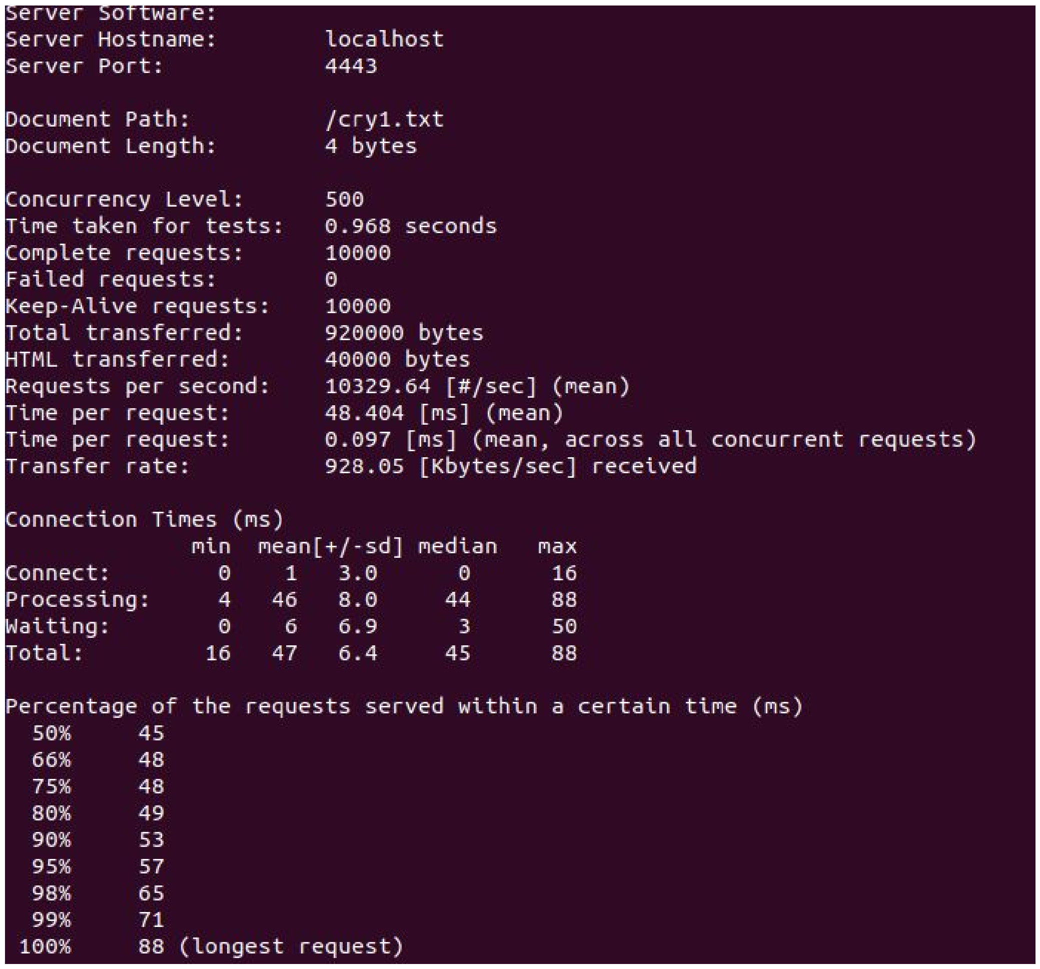
- *Performance Evaluation*

  - Using Apache benchmark tool:

```
siege -u 127.0.0.1:4444/legends.html -d1 -r100 -c10
```







```
Server Software:
Server Hostname:        localhost
Server Port:            4443

Document Path:          /cry1.txt
Document Length:        4 bytes

Concurrency Level:      500
Time taken for tests:   0.968 seconds
Complete requests:      10000
Failed requests:        0
Keep-Alive requests:    10000
Total transferred:      920000 bytes
HTML transferred:       40000 bytes
Requests per second:    10329.64 [#/sec] (mean)
Time per request:       48.404 [ms] (mean)
Time per request:       0.097 [ms] (mean, across all concurrent requests)
Transfer rate:          928.05 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   3.0      0      16
Processing:     4   46   8.0     44      88
Waiting:        0    6   6.9      3      50
Total:         16   47   6.4     45      88

Percentage of the requests served within a certain time (ms)
  50%     45
  66%     48
  75%     48
  80%     49
  90%     53
  95%     57
  98%     65
  99%     71
 100%     88 (longest request)
```