Cloud Computing HW 1 - Simple Web Server

32190984 Isu Kim

April 14, 2024

1 Index

2 Introduction

The main goal of this assignment was to implement a simple web server that runs an HTTP server. However, the assignment has a restriction not to use high-level HTTP API frameworks (e.g. `Flask` `NodeJS`), but rather stick to the original socket programming instead.

In this document, we introduce a simple web server named `engineX`. `engineX` supports following features:

- **Multi-clients**: Multiple clients can access the web server concurrently
- **Basic HTTP Operations**: Supports standard `GET`, `POST`, `UPDATE` and `DELETE`.
- **Dynamic Contents**: Supports `HTML` and `text` content parsing.
- **HTTP Response Codes**: Returns ISO HTTP response code.

`engineX` was implemented with C language and was confirmed to work with multiple scenarios under Ubuntu 22.04 environment. The full source code can be found at https://github.com/isu-kim/kloud-computing-dku/tree/main/hw1.

In section 3, the document will deal with simple design and implementation for `engineX`. In section 4, the user guide and actual demonstrations of the software will be introduced. Finally, in section 5, we will be discussing about performance numbers and methods to improve performance as well as the conclusion.

3 Design & Implementation

3.1 Multi-Threading

The most important consideration of `engineX` is that it should support multiple concurrent clients at once. Meaning that the software must support multi-threading or multi-processing. `engineX` utilizes multi-threading instead of multi-processing. The main reason for this is the performance overhead of the forking process. It is a well-known fact that forking a new child process usually demands more computing resource compared to creating a simple thread (i.e. lightweight process).

The following figure describes a sequence of system calls involved in basic socket programming. In our case, this will be an HTTP client connecting to `engineX`.

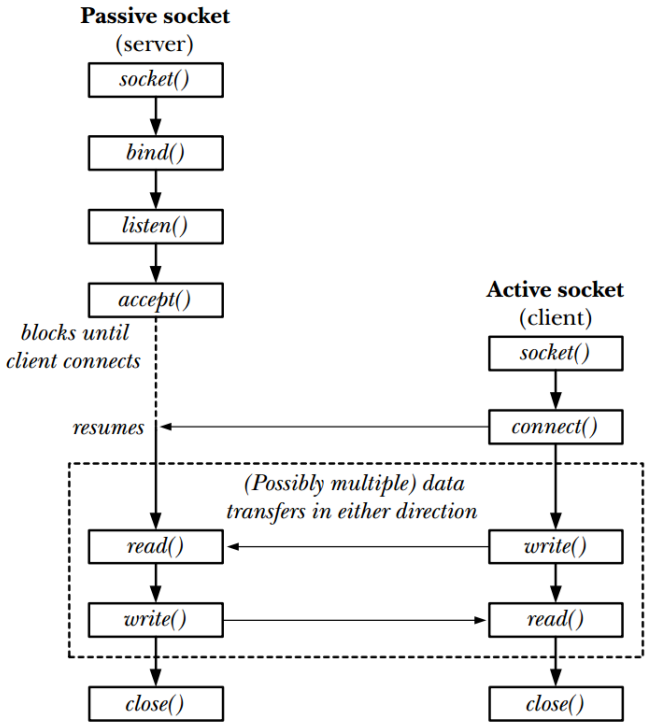

Figure 56-1: Overview of system calls used with stream sockets

Figure 1: Socket Workflow

For us to achieve multiple clients at once, each client connection should be processed in a separate worker thread. As Figure 2 shows, once a new connection has been established from a HTTP client, the sequences after `accept` which is `recv` and `write` should be processed by a worker thread.

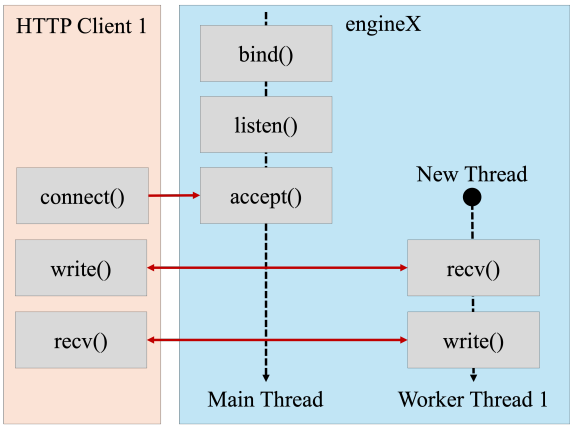

Figure 2: Single Client Workflow

Once another HTTP client arrives, the new client will be assigned to a new worker thread. The web server is not involved in that much of a race conditions. Each thread will just take care of one remote HTTP client, therefore there will be almost no synchronization required.

There might be some cases when aclient is performing `POST` or `UPDATE` request and another client is performing `GET` request, which can potentially be a problem. However, this can be easily solved by using OS tricks such as locking files, and this is can be considered as an out of scope since `engineX` is mainly aimed to provide simple web server.

3.2 Workflow

The following Figure 3 shows a simplified behavior of `engineX` for a new connection.

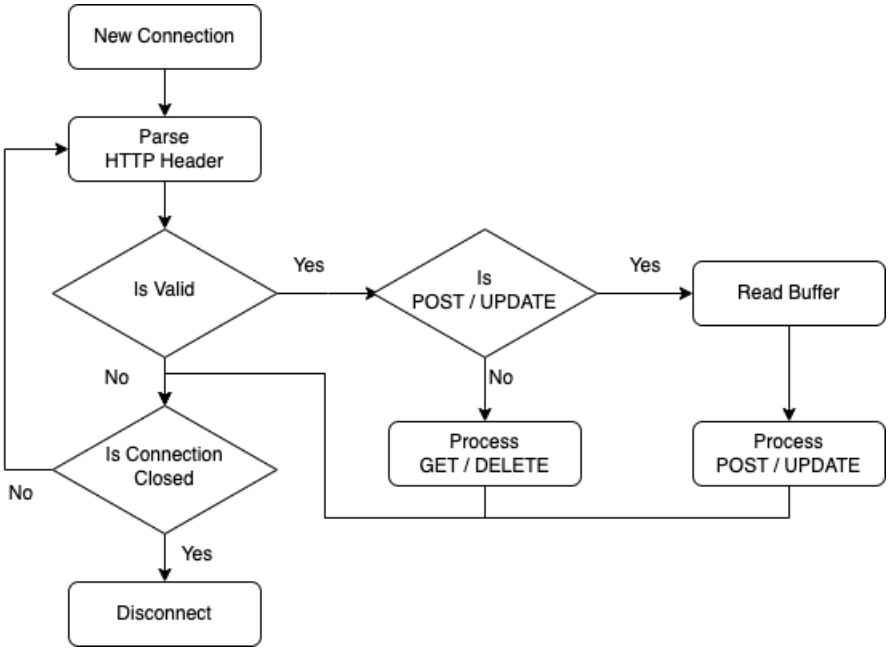

Figure 3: Connection Flow Chart

3.2.1 Parsing HTTP Header

Once a new connection is established, the server will first read the buffer using `read`. Then it will try to parse the basic HTTP header. In this step, the server will look for the following elements:

- **Method**: The HTTP request method. Currently supports `GET`, `POST`, `UPDATE`, `DELETE`.
- **Endpoint**: The URI that contains the file's name.
- **HTTP Version**: The string that contains the HTTP version such as `HTTP/1.1`

This was implemented using the basic C library's string utilities (i.e. `string.h`). There might be some cases when the client yet did not send the request information, in that case the server will try to read the buffer again unless the connection is closed.

3.2.2 Processing Request

Once the program has determined the method of the HTTP request, the server will now check if the request was `POST` or `UPDATE`. For those two cases, the client will send files. Therefore, the server needs to read the buffer with the content size provided in the HTTP header to receive the payloads. Once the payload is read, the server will create a new file or update an existing file using the payload received.

If the request was `GET` or `DELETE`, which does not require additional payloads, the server will immediately retrieve or delete the designated file.

All requests will be performed and the results will be sent back to the client using HTTP response. If the user's request was successfully performed, this will return 200. The current HTTP response codes that were implemented are as it follows:

- `200 OK`: User request was successfully performed
- `403 Forbidden`: The user did not have enough permission. This will be triggered when the designated file was not able to be read or modified by `engineX` process.
- `404 Not Found`: The file was not found under web directory.
- `413 Content Too Large`: The payload exceeds the maximum request payload content size. This is set as 4MB by default.
- `503 Internal Server Error`: `engineX` was unable to perform request.

If the request failed, the server will return the reason why the request failed. This will be returned as a string that represents the reason for `errno` defined in Linux.

3.2.3 Checking Connection

Once the server returns the response back to the HTTP client, the client might have closed the connection or kept the connection open for the next request. Therefore, the server checks if the socket connection is opened or not.

If the connection was identified as closed, the thread that is taking care of this connection will terminate. If not, the thread will keep servicing the requests.

4 User Guide & Demonstration

4.1 Environment

The environment that this program was checked running is as it follows:

- **OS**: Ubuntu 22.04.01 LTS **Little Endian**
- **CPU**: Intel(R) Core(TM) i9-7940X CPU @ 3.10GHz
- **RAM**: 64GB
- **Compiler**: GCC 11.3.0, Make 4.3
- **C**: C99

4.2 Building `engineX`

`engineX` offers `Makefile`, it provides following recipes:

1. `clean`: Cleans up the whole build, this also cleans up the web directory.
2. `build`: Builds `engineX`.
3. `debug`: Builds `engineX` in debug mode. This version of `engineX` will provide extra information on how the program is running.

For the end-user case, it is suggested to use `build`. An example terminal output can be like below:

```
$ git clone https://github.com/isu-kim/kloud-computing-dku.git
$ cd kloud-computing-dku/hw1/
$ make build
make engineX
gcc -O2 -Wall -std=gnu99 -c main.c -o main.o
gcc -O2 -Wall -std=gnu99 -c common/context.c -o common/context.o
gcc -O2 -Wall -std=gnu99 -c common/messages.c -o common/messages.o
gcc -O2 -Wall -std=gnu99 -c http/server.c -o http/server.o
...
gcc -o engineX main.o common/context.o ...
```

Code Snippet 1: Example Output of Building `engineX`

This will create an executable named `engineX`. `engineX` also provides command-line interfaces to run the software. The list shows available choices:

- `-help`, `-h`: Print out help message.
- `-port`, `-p`: Specify an port to listen HTTP server, defaults to `5000`.
- `-listen`, `-l`: Specify an IPv4 address to listen HTTP server, defaults to `0.0.0.0`.
- `-files`, `-f`: Specify directory of web files, defaults to `web/`.

An example execution command will be as follows:

```
$./engineX --port 8123 --listen 0.0.0.0
...
engineX - Simple Web Server by isu-kim (https://github.com/isu-kim/)
[2024-04-14 21:16:16] [INFO] Initializing commandline arguments
[2024-04-14 21:16:16] [INFO] No web files directory was specified, using ...
[2024-04-14 21:16:16] [INFO] Initialized arguments
- port=8123
- listen=0.0.0.0
- files_directory=web/
[2024-04-14 21:16:16] [INFO] Starting server...
[2024-04-14 21:16:16] [INFO] Checking web files directory
[2024-04-14 21:16:16] [INFO] Web files directory does not exist, creating one...
[2024-04-14 21:16:16] [INFO] Listening on 0.0.0.0:8123
```

Code Snippet 2: Example Output of Executing `engineX`

This will start up the web server in `0.0.0.0:8123`. The web server's file directory will be designated to `web/`. `engineX` provides default endpoint which is `/`. You can use your browser to check the server is up and running. Figure 4 demonstrates an example screenshot of visiting the website using Chrome browser.

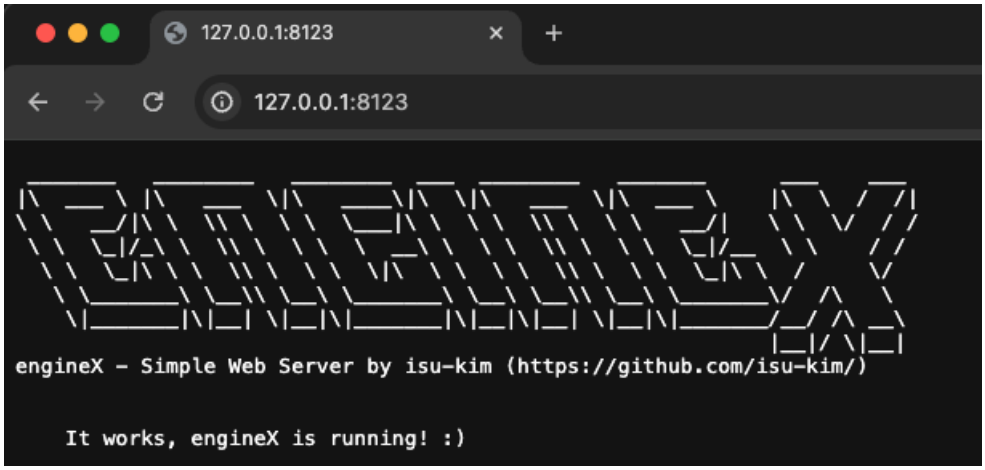

Figure 4: Screenshot of Web Browser

Meanwhile, you can check the website is running using commands such as `curl`.

```
$ curl http://127.0.0.1:8123
...
engineX - Simple Web Server by isu-kim (https://github.com/isu-kim/)


    It works, engineX is running! :)
```

Code Snippet 3: Example Output of Executing `curl`

4.3 POST Method

engineX supports POST request which sends a text file to server. This will create the file under the designated endpoint. An example command is as follows:

```
$ curl -X POST --upload-file "isu.html" http://127.0.0.1:8123/isu.html
Successfully written file as web//isu.html: 232 bytes
```

Code Snippet 4: Example Output of POST using curl

This will POST the file isu.html in the local directory as endpoint isu.html. In the server, the log will show that the request has been successfully processed like below:

```
... [INFO] [Worker][127.0.0.1:17104] Successfully written file web//isu.html
... [INFO] [Worker][127.0.0.1:17104][200] POST /isu.html HTTP/1.1
```

Code Snippet 5: Example Output of engineX after POST

Also, in the web directory, there will be a new file isu.html.

```
$ ls web/
isu.html
```

Code Snippet 6: Example Output of POST Result

- The maximum file size is defined as 2MB. If you wish to post larger file than 2MB, modify the size defined as HTTP_MAX_REQUEST_LEN in common/defines.h
- POST method only supports text files. Posting binary files or using form-based requests might not work. Meaning that the curl -X POST –upload-file "path/to/your/file.txt" was tested only.
- POST method only supports creating new files. If you would like to change the file contents, use UPDATE.

4.4 GET Method

Figure 5 shows the screenshot that that the HTML file uploaded in 4.3 was rendered.

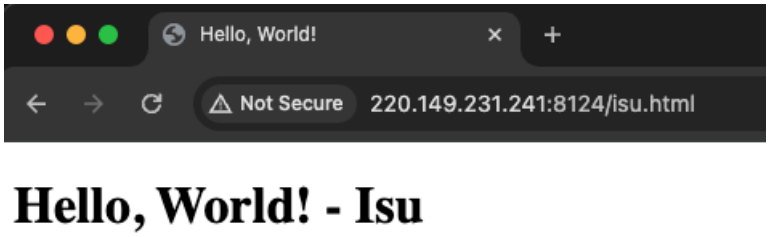

Figure 5: Screenshot of Web Browser

Now, if we change the permission of `web/isu.html` to a value that is not accessible by `engineX` process, the following page will be shown.

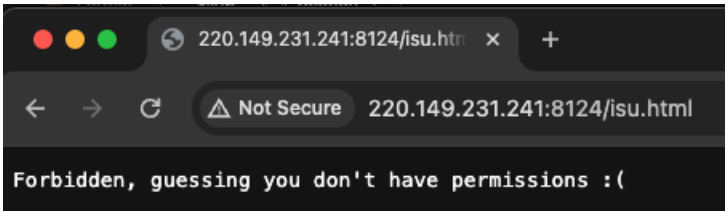

Figure 6: Screenshot of Web Browser

This will show that the user did not have enough permission to read the file. Also, the response code will come as 403 forbidden. Once we delete the file, the page will show that there is no such file like the following figure.

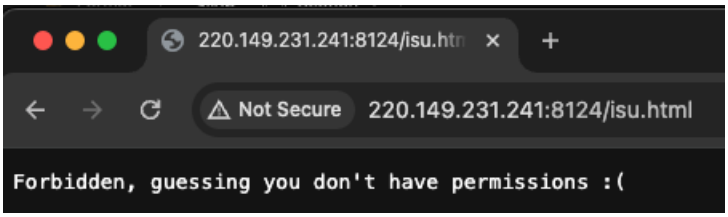

Figure 7: Screenshot of Web Browser

- The server will serve its type as `text/text` or `text/html` according to the file's extension. If the file ends with `.html`, the file will be served as `text/html`. Other content types were not defined.
- The server can only serve contents up to 4MB. You can adjust this value by modifying `HTTP_MAX_RESPONSE_LEN` under `common/defines.h`.

4.5 UPDATE Method

engineX supports UPDATE method, which can overwrite the content of an existing file. This only works if the file already exists. An example command is as follows:

```
$ curl -X UPDATE --upload-file "isu-mod.html" http://127.0.0.1:8124/isu.html
Successfully updated file web//isu.html 232->267 bytes
```

Code Snippet 7: Example Output of UPDATE using curl

This will upload the file isu-mod.html in the local directory and overwrite the content of web/isu.html. Meanwhile, in the server, the log will show that the request has been successfully processed like below:

```
... [INFO] [Worker][127.0.0.1:45225] Successfully written file web//isu.html
... [INFO] [Worker][127.0.0.1:45225][200] UPDATE /isu.html HTTP/1.1
```

Code Snippet 8: Example Output of engineX after UPDATE

As we have updated the file, you can see the changes in using the browser. As compared to Figure 5, Figure 8 demonstrates that the web page has changed.

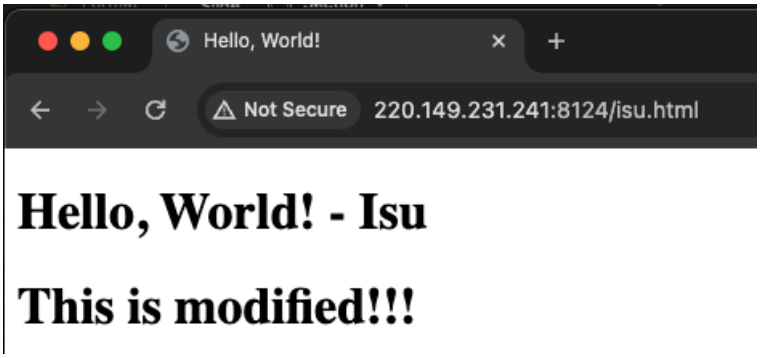

Figure 8: Screenshot of Web Browser

- The maximum file size is defined as 2MB. If you wish to upload larger file than 2MB, modify the size defined as HTTP_MAX_REQUEST_LEN in common/defines.h
- UPDATE method only supports text files. Posting binary files or using form-based requests might not work. Meaning that the curl -X UPDATE –upload-file "path/to/your/file.txt" was tested only.

4.6 DELETE Method

engineX supports DELETE method. This deletes an existing file. An example command is as below:

```
$ curl -X DELETE http://127.0.0.1:8124/isu.html
Successfully deleted file web//isu.html
```

Code Snippet 9: Example Output of engineX after DELETE

This will delete the file web/isu.html. Meanwhile, in the server, the log will show that the request has been successfully processed like below:

```
... [INFO] [Worker][127.0.0.1:3308][200] DELETE /isu.html HTTP/1.1
```

Code Snippet 10: Example Output of engineX after UPDATE

Since the file was deleted, the endpoint will no longer be able to be accessed. The following figure shows that the server returns 404 not found.

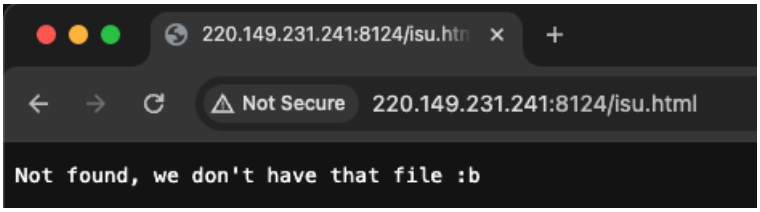

Figure 9: Screenshot of Web Browser

5 Evaluation & Conclusion

5.1 Evaluation

To check the performance of engineX, I have used wrk to generate workload and query a single endpoint. For the experiment, the wrk was running with 32 threads with 32 connections for 5 seconds. In order to minimize the parameters affecting the performance, the benchmark was executed in the same machine as engineX. This will avoid inaccuracy caused by network IO to an extent. An example output was as it follows:

```
$ wrk -t32 -c32 -d5s http://127.0.0.1:8124
Running 5s test http://127.0.0.1:8124
 32 threads and 32 connections
 Thread Stats  Avg   Stdev   Max  +/- Stdev
  Latency   2.98ms  4.12ms 48.04ms  83.45%
  Req/Sec  329.43  631.35   2.97k  93.69%
 14562 requests in 5.10s, 10.50MB read
 Non-2xx or 3xx responses: 1
Requests/sec:  2855.06
Transfer/sec:   2.06MB
```

Code Snippet 11: Example Output of wrk

The output shows that engineX is capable of serving 2855 requests with only one failure. In order to compare with production-grade application, I have tested the same parameters with Nginx. Nginx was set to have only one worker process. The results were as follows:

Type	Nginx	engineX
Requests/sec	21684	2855
Transfer/sec(MB)	17.76	2.06

Figure 10: Table of Performance against Nginx

The table shows that we have failed to beat Nginx. The performance was around 1/10 of the performance that Nginx offers. To improve engineX in the future, I have investigated potential bottlenecks as well as performance overheads that might have occurred. The following content shows the processes where the performance impact was considered to be high.

5.1.1 Receiving Packets

Currently, the server relies on read function which regularly checks if the client has sent a packet to the server. However, since this is done as an infinite loop, the server is running as if it is doing a busy waiting. This will require more resource usage. Therefore, to solve this issue, using epoll or select on the socket file descriptor and triggering the events once a TCP packet has arrived would be a better choice.

5.1.2 Single Request per Connection

Currently, each worker thread can only take care of a single request at once. The current implementation only processes a single received packet at once. In order to maximize concurrency, each worker thread that are dealing with a single connection shall be able to create another child thread that takes care of read and write operations. This is the main reason why `engineX` is struggling with performance issues.

5.1.3 HTTP Header Parsing

The current implementation of parsing HTTP headers heavily relies on string operations, which are highly considered a bottleneck. For example, `engineX` uses `strstr` or `strcmp` functions from `string.h` which are normally a high overhead due to its high time complexity. Therefore, optimizing string parsing part will be better choice in the future.

5.1.4 Heap Memory Issues

The current implementation of responding packets requires allocating and deallocating heap memory. Since the response body itself is around 4MB, utilizing heap memory instead of function's stack memory s required. However, since heap memory allocation takes more time, there is a potential performance drop.

5.1.5 No Caching Involved

The current implementation does not have any caching methods. If `engineX` performs caching for `GET` requests, this will avoid reading the file over and over again for a same content.

5.2 Conclusion

Implementing `engineX` was a lot of fun. However, the worst part was dealing with strings in C using `string.h`. Since C language utilizes string as `char[]`, this was especially difficult to be processed properly. Since offsets of strings and size of strings matter, there were lots of segmentation faults as well as buffer overflows.

Meanwhile, implementing socket network programming was a lot of fun. Since I was familiar with using HTTP or REST API frameworks such as `Flask`, `Gin` from `python` or `golang`, building `engineX` from scratch was a lot of fun. Also, it was interesting to see the contents served from my server was actually being rendered in the browser properly as if they were served by other web servers. Also, for performance optimization, I have looked up for `epoll` and `select` which can trigger TCP packet arrivals for higher performance and this was a nice experience.

In summary, this was a nice assignment to see how sockets and they act in the lower level. In the future, I would like to optimize the performance up to 50% of the performance that Nginx offers.