

# CMPT 371-Fall 2022

## Mini Project

Riya Roopesh 301360795

Tushrima Kelshikar 301357928

### Single Threaded Web Server:

For this part of our mini project we used simple socket programming to create a **single threaded web server**. In our implementation of the web server we open a single connection and it handles a single HTTP request at a time. This connection is completely closed before a new connection is formed and this is executed in a single sequence.

Our implementation of the web server is created to handle various requests that result in the following response status codes.

Code	Message
200	OK
304	NOT MODIFIED
400	BAD REQUEST
404	NOT FOUND
408	REQUEST TIMED OUT

We tested our web server for each of these status codes with the help of a test.html and test2.html file respectively that can be found in the same directory as our code. In the web browser we entered the following, [http://IP\\_ADDRESS:PORT/test.html](http://IP_ADDRESS:PORT/test.html) . Here, the IP address is the same as the machine on which the code is being run and the port number is the same as the one in our code.

## 200 OK:

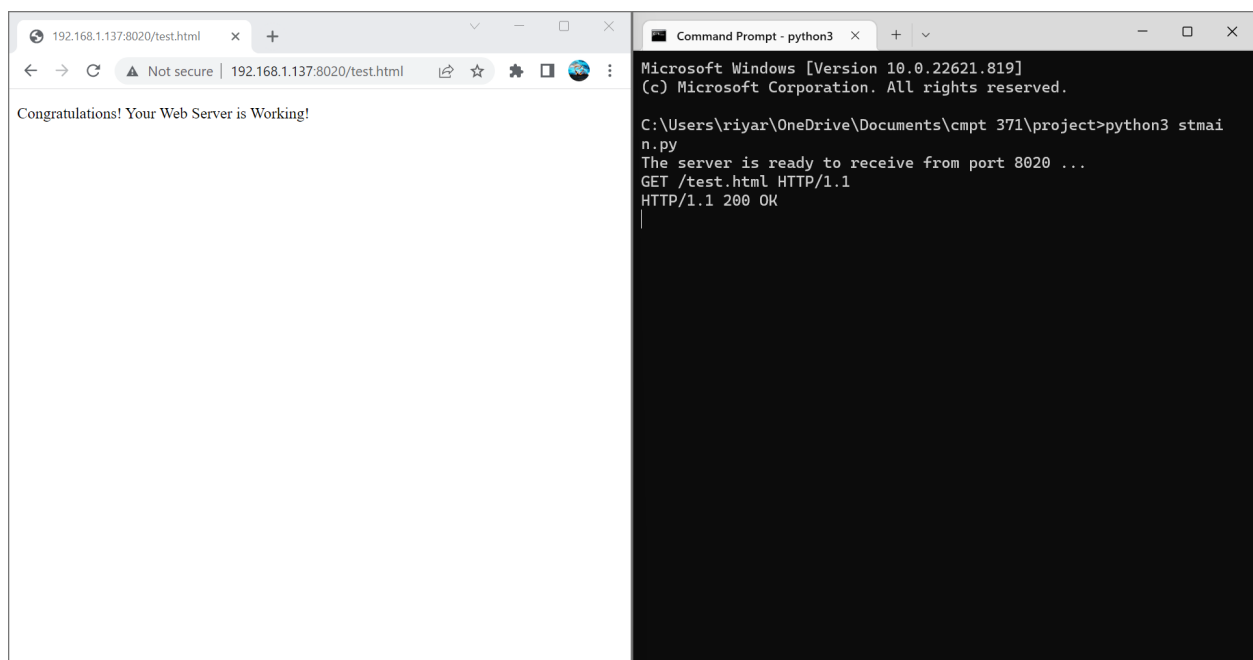
200 ok is a successful response when the server fetches the appropriate file as mentioned in the GET request sent in by the client (web browser). This also results in the printing of the file content in our browser.

In order to test our implementation, we ran our code on the terminal. Once a connection was established we used a web browser and ran the following,

<http://192.168.1.137:8020/test.html> .

This prompted the web browser to send a GET request for the test.html file from the server.

As the file was successfully retrieved from the server, it printed out the HTTP response on our terminal, and the file content on the web browser respectively as shown in the image below.

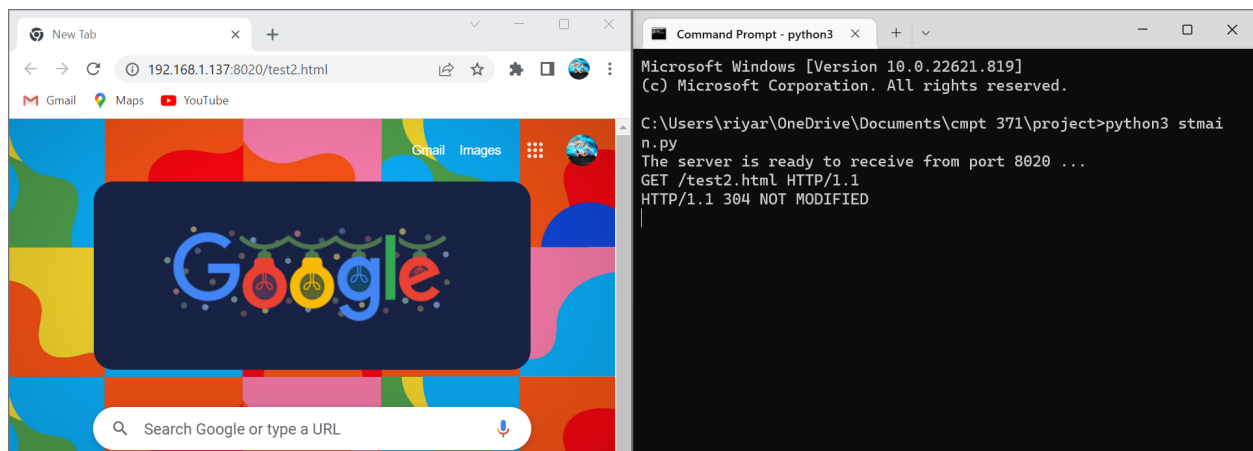


## 304 NOT MODIFIED:

304 Not Modified is an indication that the file has not been modified since the last time it was accessed. In our implementation we have created a new file where the created time and the last modified time are considered to be equal. If they are equal then we print 304 NOT MODIFIED and if they are not equal we print 200 OK.

In order to test our implementation, we ran our code on the terminal. Once a connection was established we used a web browser and ran the following, <http://192.168.1.137:8020/test2.html> .

This prompted the web browser to send a GET request for the test2.html file from the server. Since our requests were being sent by the web browser the only way for us to test 304 Not Modified was by creating a new file test2.html that had the same created\_at and last\_modified time, as we were asked to create a simple web server and there was no requirement for a cache.



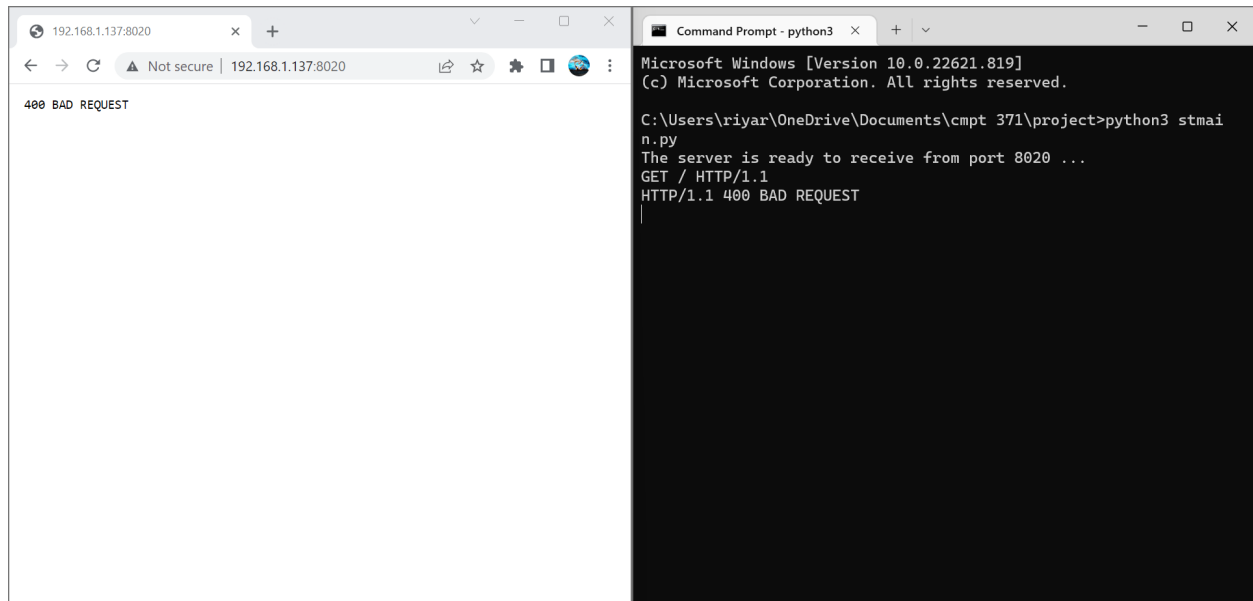
## 400 BAD REQUEST:

400 Bad Request is a client error response when the server cannot or does not process the request sent in by the client (web browser) due to an error from the client side. This also results in the printing of the file content in our browser.

In order to test our implementation, we ran our code on the terminal. Once a connection was established we used a web browser and ran the following, <http://192.168.1.137:8020> . We implement this error from the client. Client based errors can be of the following types, malformed request syntax or invalid request message framing.

This prompted the web browser to send a GET request file from the server. Since this is a wrong input the file cannot be implemented. As the file was unable to be retrieved from the server, it

went to the last condition in our code, which is to print out the HTTP response on our terminal, and the filecontent on the web browser respectively as shown in the image below.



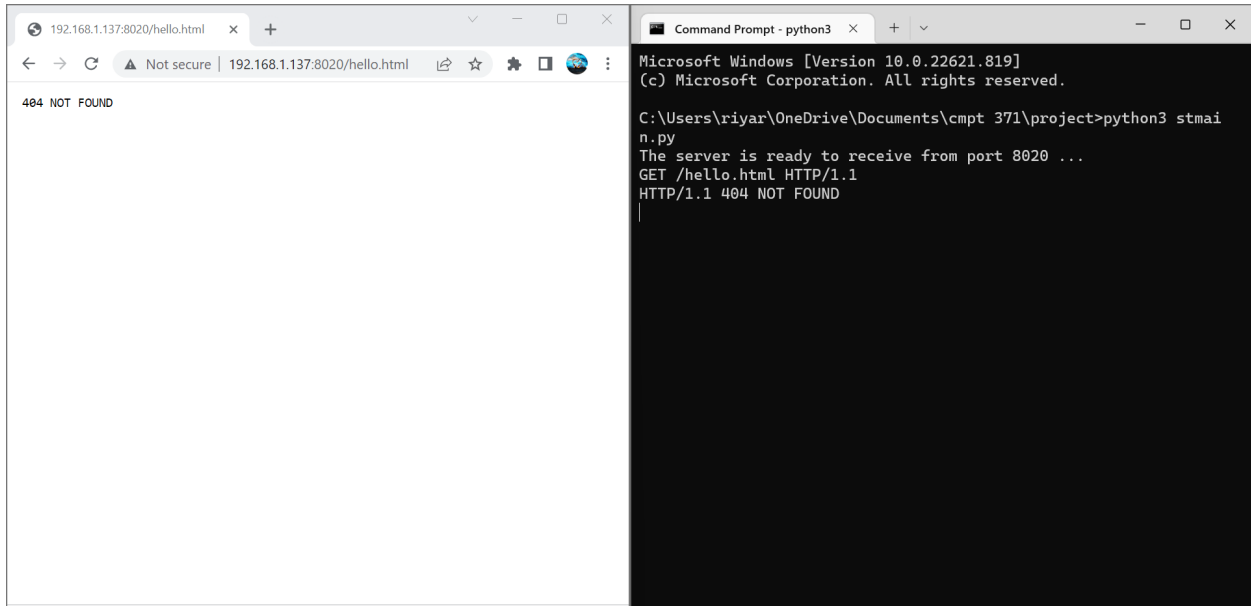
#### 404 NOT FOUND:

404 Not Found is a client error response when the server cannot or does not find the request sent in by the client (web browser) since the file does not exist or from the client side. This also results in the printing of the file content in our browser.

In order to test our implementation, we ran our code on the terminal. Once a connection was established we used a web browser and ran the following,

<http://192.168.1.137:8020/hello.html>.

This prompted the web browser to send a GET request for the hello.html file from the server. As the file did not exist in our folder it could not be retrieved from the server, hence it printed out the HTTP response on our terminal, and the filecontent on the web browser respectively as shown in the image below.

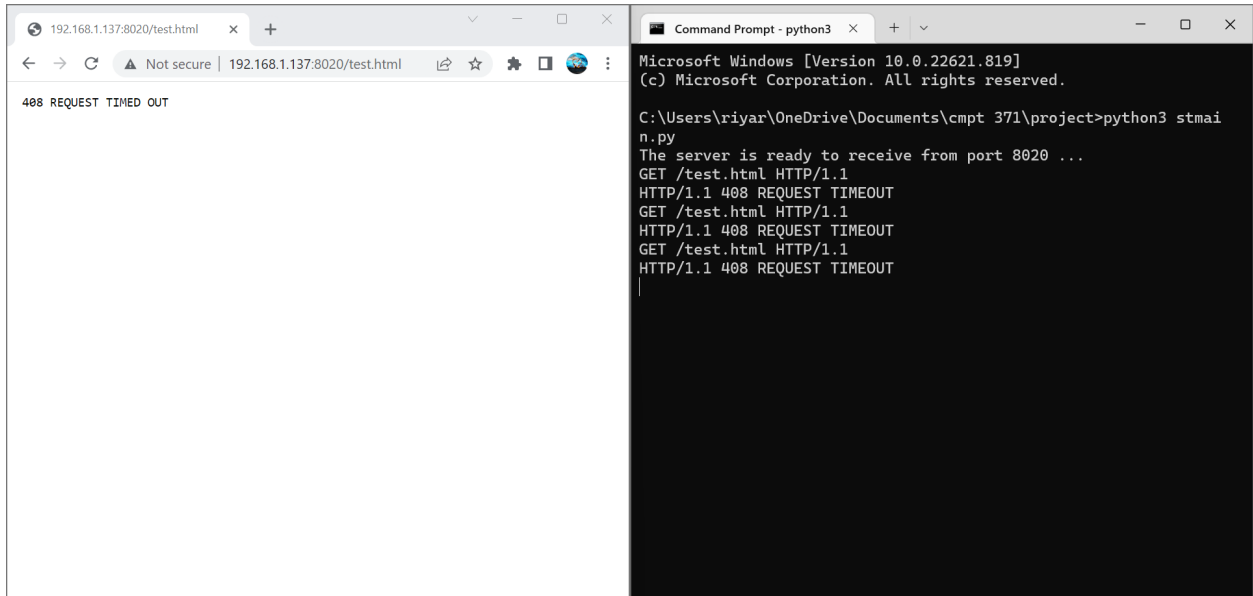


#### 408 REQUEST TIMED OUT:

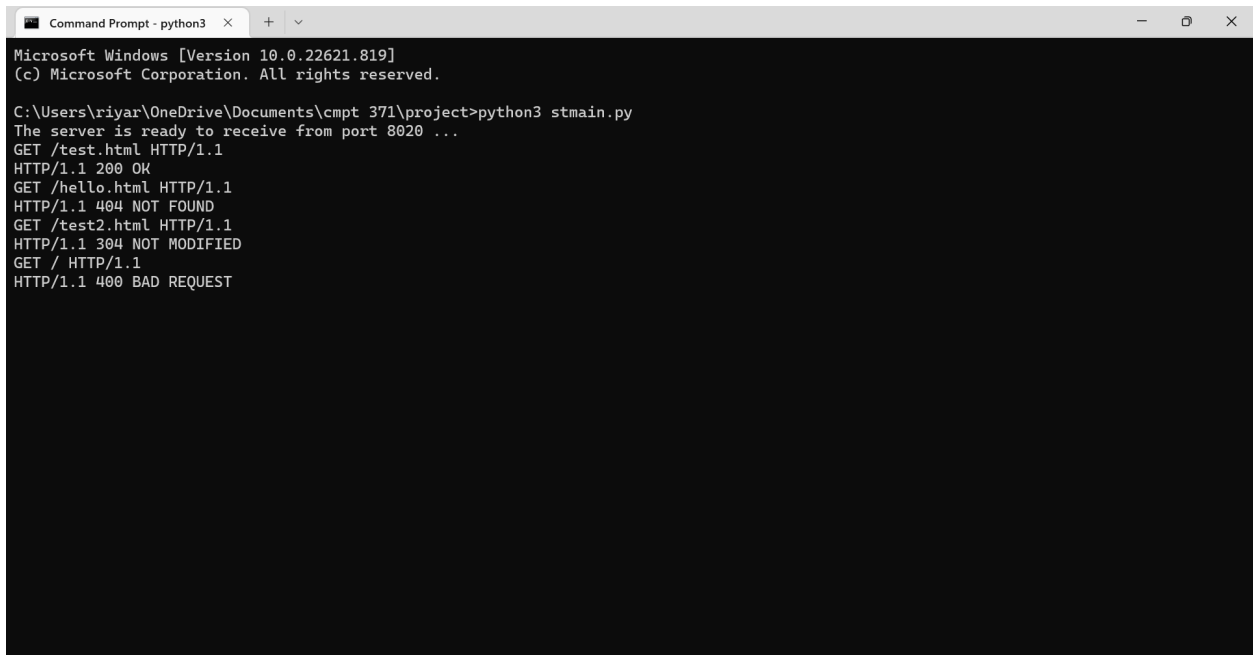
408 Request timed out is a client error response when the response is sent on an idle connection when the request to the server takes longer than the server's allocated timeout window. This also results in the printing of the file content in our browser.

In order to test our implementation, we ran our code on the terminal. Once a connection was established we used a web browser and ran the following, <http://192.168.1.137:8020/test.html>. We also need to uncomment the following line, `time.sleep(3)`, which enforces the execution to be suspended for 3 seconds. We check if the time is greater than 1 the code returns the 408 Request Timeout message.

This prompted the web browser to send a GET request for the test.html file from the server. As the file took too long, it could not be retrieved from the server, hence it printed out the HTTP response on our terminal, and the filecontent on the web browser respectively as shown in the image below.



The following image is our output when sending in multiple requests, it is accessed in a single sequence one request is fulfilled and then the other request is obtained.

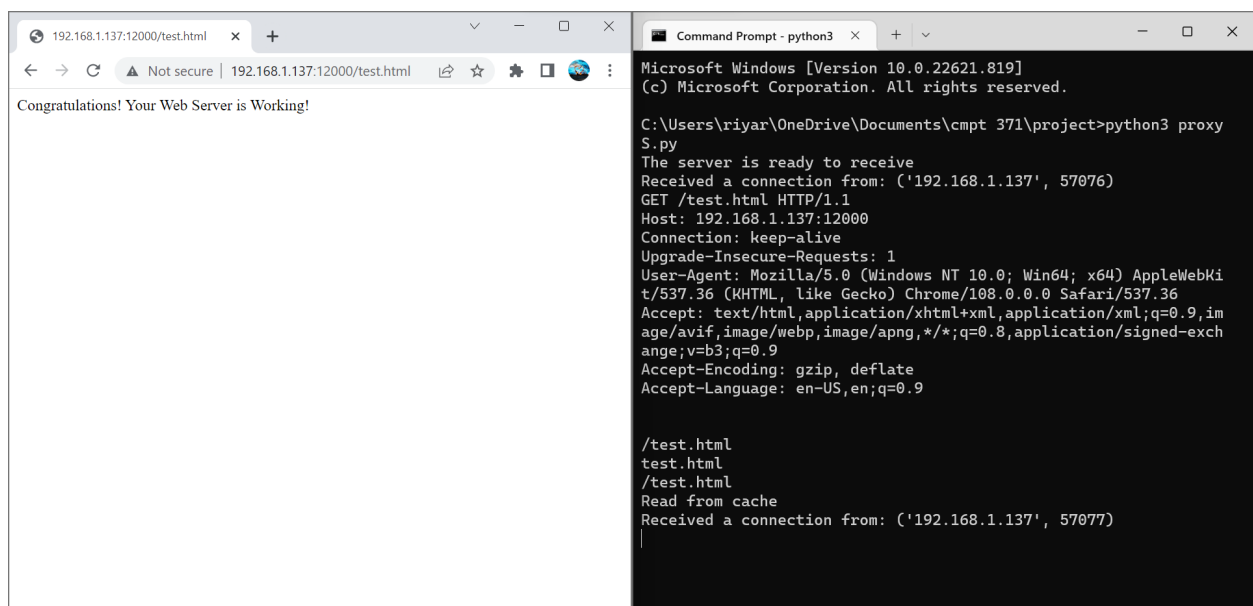


# Web Proxy Server:

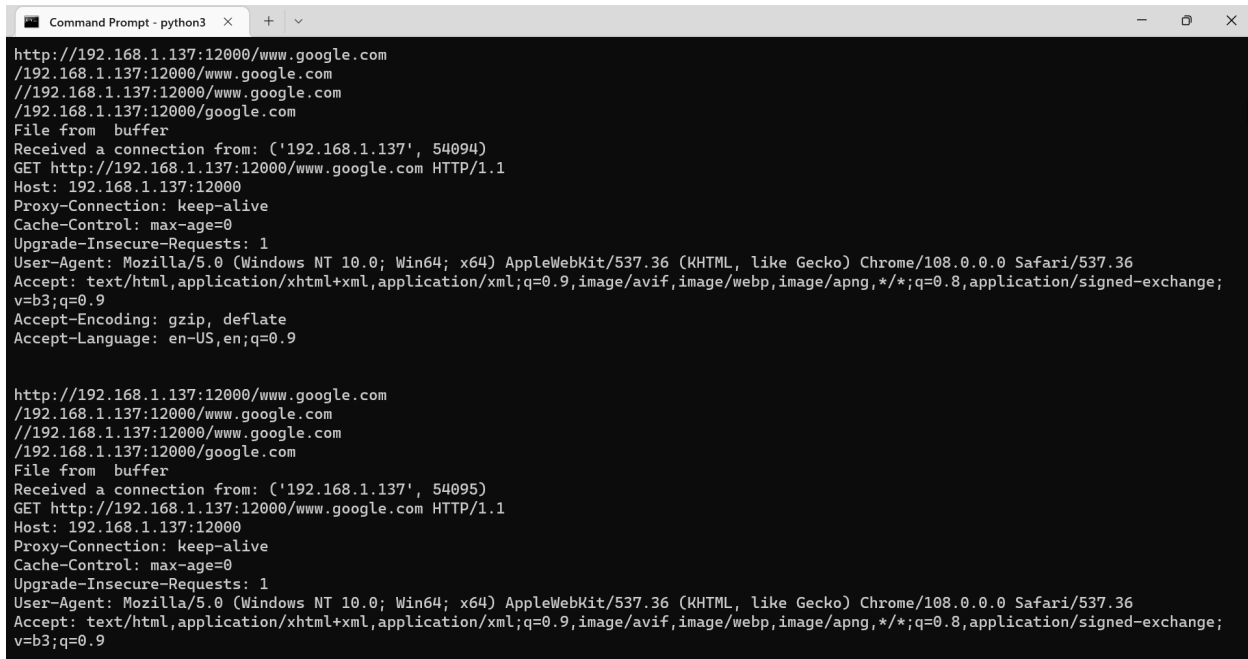
For the part of the mini project we used simple socket programming to implement a web proxy server, a web proxy server is a web server that acts as a gateway/ middleman between the client server(web browser) and the real server. The main role of the proxy server is to satisfy client requests without involving the real server. As a result, the proxy server can be implemented as the client and server. A proxy server intercepts a request sent in by the client to the real server for a particular file. If the file has been accessed previously, then the proxy fulfills the client's request, or else it sends the request to the real server which fulfills the request and sends the response.

Whereas, in a web server, there is no intermediate to intercept the request sent in by the client. Therefore, the real server fulfills the request and sends in the response.

In our implementation, when the client sends the request for the first time, it reads from the original server, but when we implement the request again, the request is read from cache. In order to test this we will run the proxyS.py in the terminal. We will then run the following on the web browser <http://192.168.1.137:12000/test.html>. This will result in the file being read from cache as shown in the image below.



We then activated the proxy server on our PC and ran the following, and ran the following on the web browser <http://192.168.1.137:12000/www.google.com> and this resulted in the following output as shown below.



```
Command Prompt - python3
http://192.168.1.137:12000/www.google.com
/192.168.1.137:12000/www.google.com
//192.168.1.137:12000/www.google.com
/192.168.1.137:12000/google.com
File from buffer
Received a connection from: ('192.168.1.137', 54094)
GET http://192.168.1.137:12000/www.google.com HTTP/1.1
Host: 192.168.1.137:12000
Proxy-Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9

http://192.168.1.137:12000/www.google.com
/192.168.1.137:12000/www.google.com
//192.168.1.137:12000/www.google.com
/192.168.1.137:12000/google.com
File from buffer
Received a connection from: ('192.168.1.137', 54095)
GET http://192.168.1.137:12000/www.google.com HTTP/1.1
Host: 192.168.1.137:12000
Proxy-Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
```

## Multi-threaded Web Server:

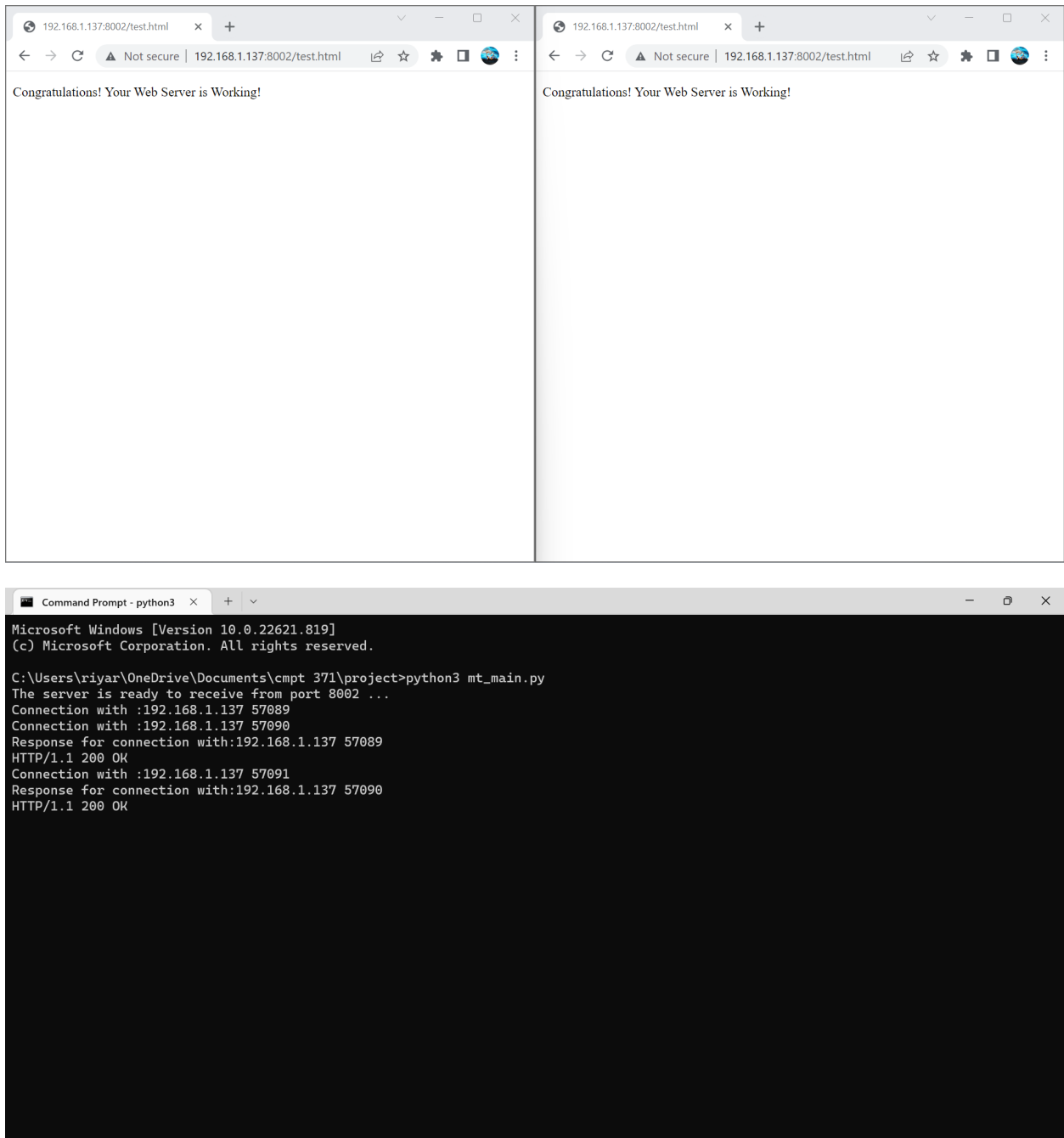
For this part of our mini project we extended our web server to its multi-threaded version, which is capable of handling multiple requests simultaneously. A multi-threaded web server has multiple connections at the same time. This connection is still open once a new connection is sent and this is executed using an in-built python library called threading.

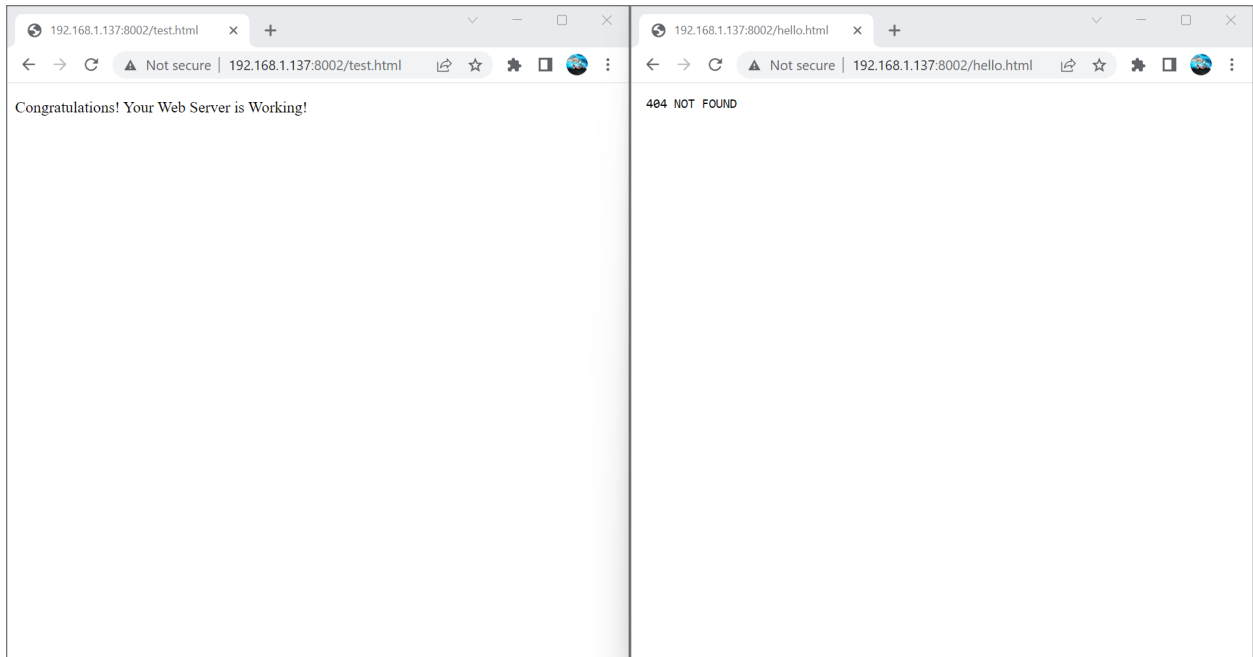
We tested our web server for each of these status codes with the help of a test.html and test2.html file respectively that can be found in the same directory as our code. In the web browser we entered the following, [http://IP\\_ADDRESS:PORT/test.html](http://IP_ADDRESS:PORT/test.html). Here, the IP address is the same as the machine on which the code is being run and the port number is the same as the one in our code.

In order to test our implementation, we will have two browsers open and send in requests to the server simultaneously. The response on the terminal will indicate parallelism in the handling of



the request. Our implementation will indicate the connection address for each request as well as response.





```
Command Prompt - python3
Microsoft Windows [Version 10.0.22621.819]
(c) Microsoft Corporation. All rights reserved.

C:\Users\riyar\OneDrive\Documents\cmpt 371\project>python3 mt_main.py
The server is ready to receive from port 8002 ...
Connection with :192.168.1.137 57101
Connection with :192.168.1.137 57102
Response for connection with:192.168.1.137 57101
HTTP/1.1 200 OK
Connection with :192.168.1.137 57103
Response for connection with:192.168.1.137 57102
HTTP/1.1 404 NOT FOUND
```