

---

# **FINAL PROJECT - WEB SERVER**

---

**November 18, 2020**

By

Pranathi Bhuvanagiri - 20171135

Rupa Nuthalakanti - 20171111

Sushanth Reddy - 20171172

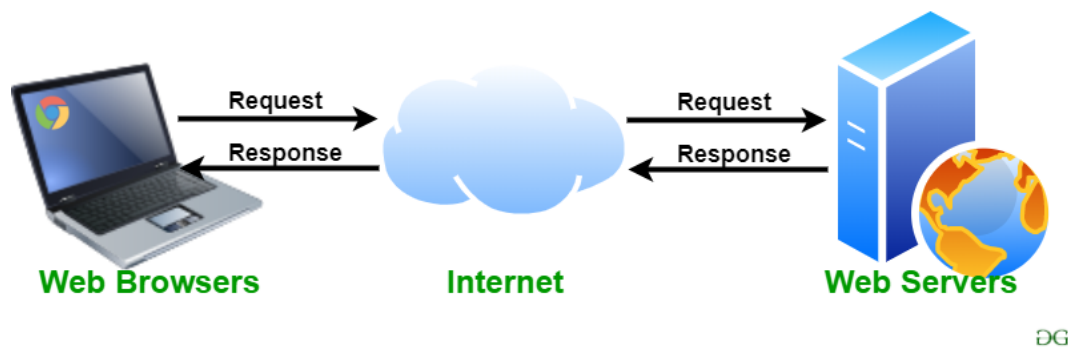
# Contents

1	Introduction . . . . .	2
1.1	What is a Web Server? . . . . .	2
2	Approach . . . . .	3
3	Implementation . . . . .	3
3.1	TCP Server . . . . .	3
3.2	Teaching HTTP Protocol . . . . .	4
3.3	HTTP Requests . . . . .	5
3.3.1	HTTP GET . . . . .	5
3.3.2	HTTP POST . . . . .	6
3.3.3	HTTP PUT . . . . .	6
3.3.4	HTTP DELETE . . . . .	7
3.4	Multi-Threading . . . . .	8
3.5	Handling Synchronisation Problems . . . . .	8
4	Challenges Faced . . . . .	9
5	Project Presentation . . . . .	9

# 1 INTRODUCTION

## 1.1 What is a Web Server?

A web server is a computer that runs websites. It's a computer program that distributes web pages as they are requisitioned. The basic objective of the web server is to store, process and deliver web pages to the users. This intercommunication is done using Hypertext Transfer Protocol (HTTP). These web pages are mostly static content that includes HTML documents, images, style sheets, test etc. Apart from HTTP, a web server also supports SMTP (Simple Mail transfer Protocol) and FTP (File Transfer Protocol) protocol for emailing and for file transfer and storage. The main job of a web server is to display the website content. The Web Server is requested to present the content website



**Figure 1:** A simple Web Server

to the user's browser. All websites on the Internet have a unique identifier in terms of an IP address. This Internet Protocol address is used to communicate between different servers across the Internet. These days, Apache server is the most common web server available in the market. Apache is an open source software that handles almost 70 percent of all websites available today. Most of the web-based applications use Apache as their default Web Server environment.

The project aim is to implement a simple web server using multi-threading, synchronization which can handle several requests such as GET, POST, UPDATE, DELETE ,i.e HTTP requests.

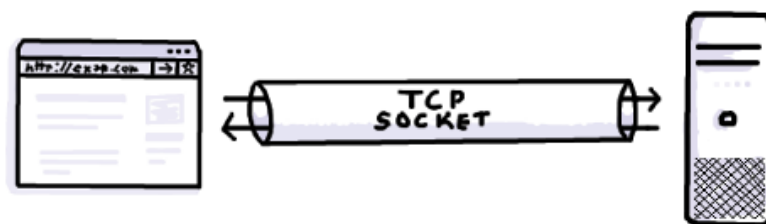
## 2 APPROACH

HTTP (Hypertext Transport Protocol) is built on top of TCP (Transmission Control Protocol), so we created a program capable of sending and receiving data through a TCP socket. At this point our TCP server doesn't understand HTTP. To convert it from a simple TCP server to an HTTP server all we need to do is teach it the HTTP protocol. After this, our server can take HTTP requests and respond to clients. And also, we used Multi-threading to handle multiple requests from multiple clients at the same time and semaphores to avoid synchronization.

## 3 IMPLEMENTATION

### 3.1 TCP Server

Since HTTP works through a TCP socket, we'll start by writing a simple TCP server (Echo server). An Echo server is a program that returns the data that it receives, nothing less, nothing more. Browser will send an HTTP request to our TCP server. Since our server is an Echo server, it will return the data that our browser sends it back to the browser. We will use Python's socket library for this.



HTTP requests and responses are sent through a TCP socket.

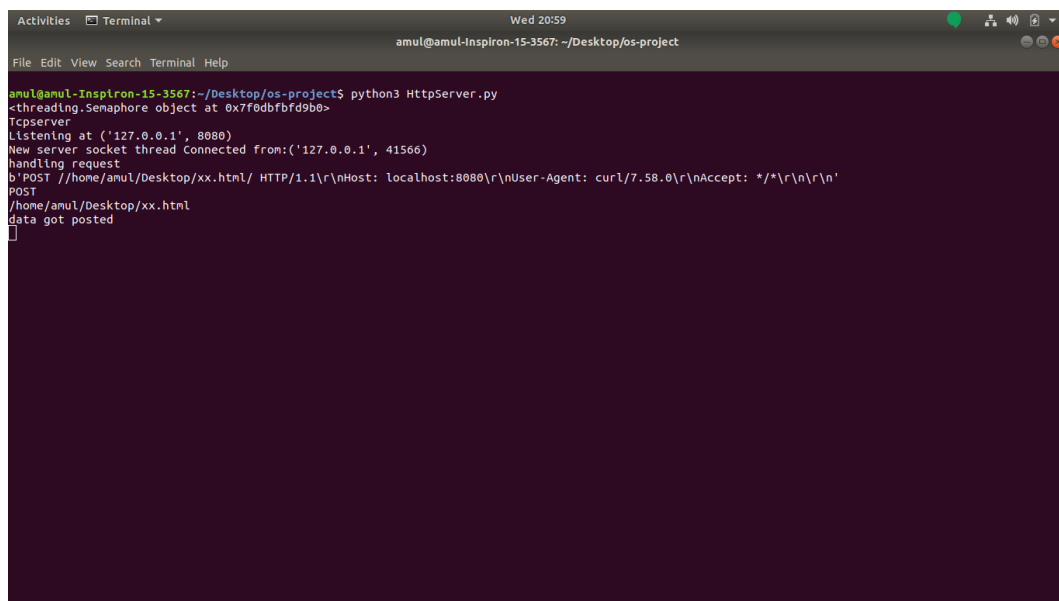
**Figure 2:** Working of HTTP and TCP protocol

## 3.2 Teaching HTTP Protocol

To make our server able to process HTTP requests, we store HTTP request string sent by browser in a data variable and we separate request type, version and resource name by parsing it. Now, we handle these requests and write a program which sends appropriate responses back to the browser.

An HTTP request is made up of the following parts:

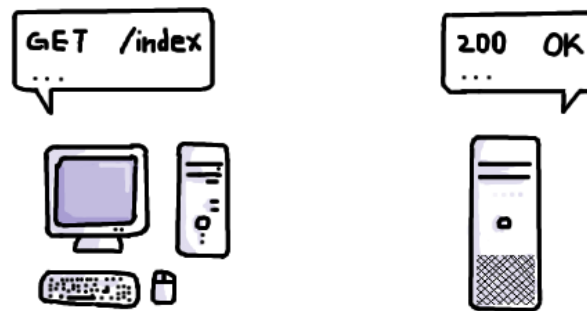
- The request line (it's the first line)
- Request headers (optional)
- A blank line
- Request body (optional)

A screenshot of a terminal window titled 'amul@amul-inspron-15-3567: ~/Desktop/os-project'. The terminal shows the execution of a Python script 'HttpServer.py'. The output includes: '<threading.Semaphore object at 0x7f0dbfbfd9b0>', 'Tcpserver', 'Listening at ('127.0.0.1', 8080)', 'New server socket thread Connected from:('127.0.0.1', 41566)', 'handling request', and a raw HTTP request string: 'b'POST //home/amul/Desktop/xx.html/ HTTP/1.1\r\nHost: localhost:8080\r\nUser-Agent: curl/7.58.0\r\nAccept: \*/\*\r\n\r\n'. Below this, the server responds with 'POST' and '/home/amul/Desktop/xx.html', followed by 'data got posted'.

**Figure 3:** HTTP REQUEST LINE FORMAT

An HTTP response is made up of the following parts:

- The response line (it's the first line)
- Response headers (optional)
- A blank line
- Response body (optional)



**Figure 4:** Working of HTTP protocol

The response line and the headers contain information for the browser. They are not shown to the client. The response body contains the data which the browser displays on the screen for the client.

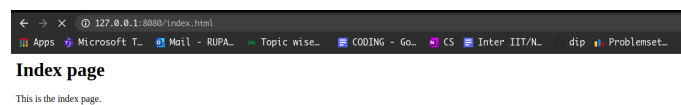
The blank line plays a very important role in HTTP and many other protocols. It helps to separate the headers and response body. Without it, there's no way for browsers to tell which part of the response should be shown to the user.

### 3.3 HTTP Requests

#### 3.3.1 HTTP GET

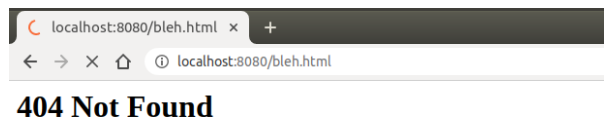
GET is used to request data from a specified resource.

If the resource we specified is present in the server, the response is displayed in the browser as:



**Figure 5:** Response generated for GET request when there is a file

If the resource we specified is not present in the server, the response is displayed in the browser as:



**Figure 6:** Response generated for GET request when there is no file

### 3.3.2 HTTP POST

POST is used to send data to a server to upload a resource. The user uses the CURL command for this request.

If the user requests to upload a new resource to the server, the response is displayed as:

```

Activities  Terminal  Wed 21:32
amul@amul-Inspiron-15-3567: ~/Desktop/os-project

File Edit View Search Terminal Help

amul@amul-Inspiron-15-3567:~/Desktop/os-project$ python3 HttpServer.py
<threading.Semaphore object at 0x7fb1a81839e8>
Tcpserver
Listening at ('127.0.0.1', 8080)
New server socket thread Connected from:('127.0.0.1', 41820)
handling request
b'POST /home/amul/Desktop/tt.txt/ HTTP/1.1\r\nHost: localhost:8080\r\nUser-Agent: curl/7.58.0\r\nAccept: */*\r\n\r\n'
POST
/home/amul/Desktop/tt.txt
data got posted
  
```

**Figure 7:** Response generated for POST request when there is no file

If the user requests to upload an already existing resource, the response is displayed as:

```

Activities  Terminal  Wed 21:34
amul@amul-Inspiron-15-3567: ~/Desktop/os-project

File Edit View Search Terminal Help

amul@amul-Inspiron-15-3567:~/Desktop/os-project$ python3 HttpServer.py
<threading.Semaphore object at 0x7ff015eab940>
Tcpserver
Listening at ('127.0.0.1', 8080)
New server socket thread Connected from:('127.0.0.1', 41900)
handling request
b'POST /home/amul/Desktop/tt.txt/ HTTP/1.1\r\nHost: localhost:8080\r\nUser-Agent: curl/7.58.0\r\nAccept: */*\r\n\r\n'
POST
/home/amul/Desktop/tt.txt
Resource already exists
  
```

**Figure 8:** Response generated for POST request when there is a file

### 3.3.3 HTTP PUT

PUT is used to update an existing resource in the server. The user uses the CURL command for this request.

If the user requests to update an already existing resource, the response is displayed as:

```
anul@anul-Inspiron-15-3567:~/Desktop/os-project$ python3 HttpServer.py
<threading.Semaphore object at 0x7fe0b1830a58>
Tcpserver
Listening at ('127.0.0.1', 8080)
New server socket thread Connected from:('127.0.0.1', 42048)
handling request
b'PUT //home/anul/Desktop/tt.txt/ HTTP/1.1\r\nHost: localhost:8080\r\nUser-Agent: curl/7.58.0\r\nAccept: */*\r\n\r\n'
PUT
home/anul/Desktop/tt.txt
tt.txt
data got updated
```

**Figure 9:** Response generated for PUT request when there is a file

If the user requests to update a non existing resource, the response is displayed as:

```
anul@anul-Inspiron-15-3567:~/Desktop/os-project$ python3 HttpServer.py
<threading.Semaphore object at 0x7f5b3d6c4a20>
cpserver
Listening at ('127.0.0.1', 8080)
New server socket thread Connected from:('127.0.0.1', 42172)
handling request
b'PUT //home/anul/Desktop/tt.txt/ HTTP/1.1\r\nHost: localhost:8080\r\nUser-Agent: curl/7.58.0\r\nAccept: */*\r\n\r\n'
PUT
one/anul/Desktop/tt.txt
t.txt
resource Not Found
```

**Figure 10:** Response generated for PUT request when there is no file

### 3.3.4 HTTP DELETE

DELETE is used to delete a resource in the server. The user uses the CURL command for this request.

If the user requests to delete an existing resource, the response is displayed as:

```
anul@anul-Inspiron-15-3567:~/Desktop/os-project$ python3 HttpServer.py
<threading.Semaphore object at 0x7f44b8ba6a20>
Tcpserver
Listening at ('127.0.0.1', 8080)
New server socket thread Connected from:('127.0.0.1', 42310)
handling request
b'DELETE //home/anul/Desktop/tt.txt/ HTTP/1.1\r\nHost: localhost:8080\r\nUser-Agent: curl/7.58.0\r\nAccept: */*\r\n\r\n'
DELETE
home/anul/Desktop/tt.txt
data got deleted
```

**Figure 11:** Response generated for DELETE request when there is a file

If the user requests to delete a non existing resource, the response is displayed as:



```

anul@anul-Inspiron-15-3567:~/Desktop/os-projects$ python3 HttpServer.py
<threading.Semaphore object at 0x7f139f1bc9e8>
Topservice
Listening at ('127.0.0.1', 8080)
New server socket thread Connected from: ('127.0.0.1', 42202)
handling request
b'DELETE //home/anul/Desktop/tt.txt/ HTTP/1.1\r\nHost: localhost:8080\r\nUser-Agent: curl/7.58.0\r\nAccept: */*\r\n\r\n'
DELETE
/home/anul/Desktop/tt.txt
data Not Found

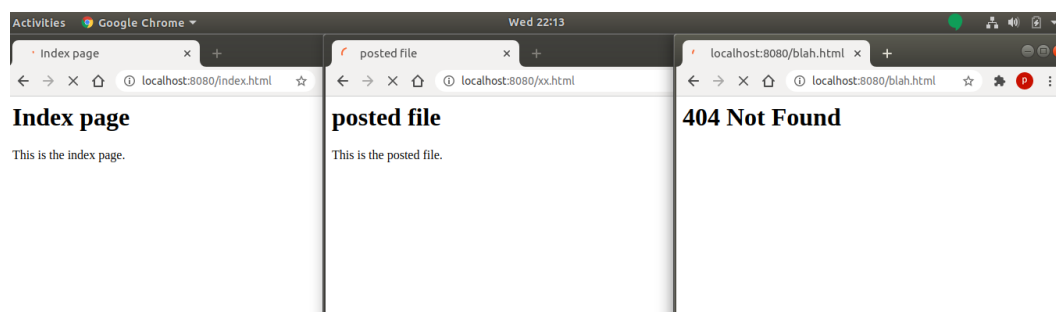
```

**Figure 12:** Response generated for DELETE request when there is no file

### 3.4 Multi-Threading

Multi-threading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system.

To Handle multiple requests from multiple users at the same time, we used Multi-threading. For this, we have used Thread from the threading module as a subclass to create and start a new thread and these threads are stored in a list.



**Figure 13:** Multi-threading

### 3.5 Handling Synchronisation Problems

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the Semaphore() method, which returns the new mutex.

The acquire(blocking) method of the new mutex object is used to force threads to run synchronously. The optional blocking parameter enables you to control whether the thread waits to acquire the mutex.

If blocking is set to 0, the thread returns immediately with a 0 value if the mutex cannot be acquired and with a 1 if the mutex was acquired. If blocking is set to 1, the thread blocks and wait for the mutex to be released.

The `release()` method of the new mutex object is used to release the mutex when it is no longer required.

## **4 CHALLENGES FACED**

We found that there will be a problem when multiple clients request to access the same resource, say one client wants to update a file at the same time another client wants to delete. In such cases, there will be a clash and it leads to critical section problem. So we used Semaphores for each request. But we also found that if two Clients want to read the same resource (HTTP GET Request), this situation is similar to Readers-Writers problem. So we followed the same technique as in Readers-Writers problem which is to use another mutex to maintain the count of number of readers who want to access the resource. If the readers count is greater than or equal to 1, we lock the main mutex and we release it when the readers count is 0.

## **5 PROJECT PRESENTATION**

The following drive link has our recorded presentation -> [Click Here](#)