Name: Darshit Bimal Gandhi

Roll No: 22M0824

CS744: Design and Engineering of Computing System

Project Title: Multi-threaded Web Server and Closed-Loop Load Tester

Instructor: Prof. Mythili Vutukuru

Overview

Developed a multi-threaded web server using thread-pooling that will handle concurrent HTTP requests effectively. To access this server's capability and throughput, I also created a closed-loop load generator to simulate multiple concurrent clients.

Components and their features

- Multi-threaded Web Server:
  - The web server uses the `master-worker thread pool architecture` to handle multiple clients concurrently. It uses a pool of reusable worker threads.

  - The main server creates a `pool of worker threads` at the start. Whenever a new client connection is accepted, the server places the accepted client file descriptor in a queue/array shared with the workers. Each worker thread fetches a client from this queue and serves it as long as the client is connected. Once the client finishes and terminates, the worker thread returns to the queue to get the next client to handle. This way, the same pool of worker threads can serve multiple clients.

  - The main server thread and the worker threads use `mutex locks` to access this queue of accepted client file descriptors without race conditions.

- Closed Loop Load-generator:
  - This will rapidly fire requests at the server to `measure the capacity of the web server`.

  - Used `Valgrind` to fix any memory leaks from the web server before running the load test to avoid server crashes due to memory errors.

  - This will act as a `closed-loop load generator`, i.e., the load is generated from a certain number of concurrent emulated users.

  - Each thread of the load generator will emulate an HTTP user/client by sending an HTTP request to the server, waiting for a response from the server, and firing the next request after the think time.

  - The number of concurrent users/threads, think-time between requests, and the duration of the load test can be changed accordingly.

  - After all the load generator threads run for the specified duration, the load generator must compute (across all its threads) and display the following performance metrics before terminating:

    - `Average throughput of the server`: The average number of HTTP requests per second successfully processed by the server for the duration of the load test.
    - `Average response time of the server`: The average amount of time taken to get a response from the server for any request, as measured at the load generator.

- The python script then generates plots of the average throughput and response time of the server as a function of the load level.

Source Code Structure

- `server/http_server_with_thread_pool.c` : Main source file containing the code to start the web server.

- `server/serverHelperFunctions.h` : Contains the various helper functions that the main source file of the web server will call.

- `server/interuptHandler.h` : Contains code to handle interupts.

- `server/serverVariables.h` : Contains the web server's global variables.

- `server/htmlFiles/` : Contains dummy HTML files to demonstrate how the multi-threaded web server works.

- `load_generator/load_generator.c` : Main source file containing the code for the load testing of the web server.

- `load_generator/load_generator_script.py` : Python script to plot the desired graph by using the load_generator executable.

- `load_generator/loadGenHelperFunctions.h` : Contains the various helper functions that the main source file of the load-tester will call.

- `load_generator/loadGenVariables.h` : Contains the load tester's global variables.

Compilation and Usage

The recommended operating system for running this program is Ubuntu.

1. To start the web server at some port number (say 8001), go to the `server/` path and run the following commands in order:

   ```
   make
   ```

   ```
   ./server 8001
   ```

   After starting this, you can go to `127.0.0.1:8000` to check if your web server is running correctly or not.

2. To run the load tester, go to the `load_generator/` path and run the following commands in order:

   ```
   make
   ```

   ```
   python3 load_generator_script.py 8001
   ```

   (8001 is the port number on which the web server is running.)

This will run the load_generator.c and will plot the desired graph.

3. Clean Object files:

   ```
   make clean
   ```

Load Testing Results

My System Configuration :

CPU Cores : 4

Number of threads threads per core : 2 Total Processors : 8

Frequency of Processor 0 : 1800.000 Mhz Frequency of Processor 1 : 1800.000 Mhz
Frequency of Processor 2 : 1800.000 Mhz Frequency of Processor 3 : 1800.000 Mhz
Frequency of Processor 4 : 1800.000 Mhz Frequency of Processor 5 : 1800.000 Mhz
Frequency of Processor 6 : 800.016 Mhz Frequency of Processor 7 : 800.020 Mhz

Architecture of the CPU : x86_64 CPU op-mode : 32 bits, 64 bits Byte Order : Little Endian

Address Size : 39 bits physical, 48 bits virtual Total Memory : 8023152 kB

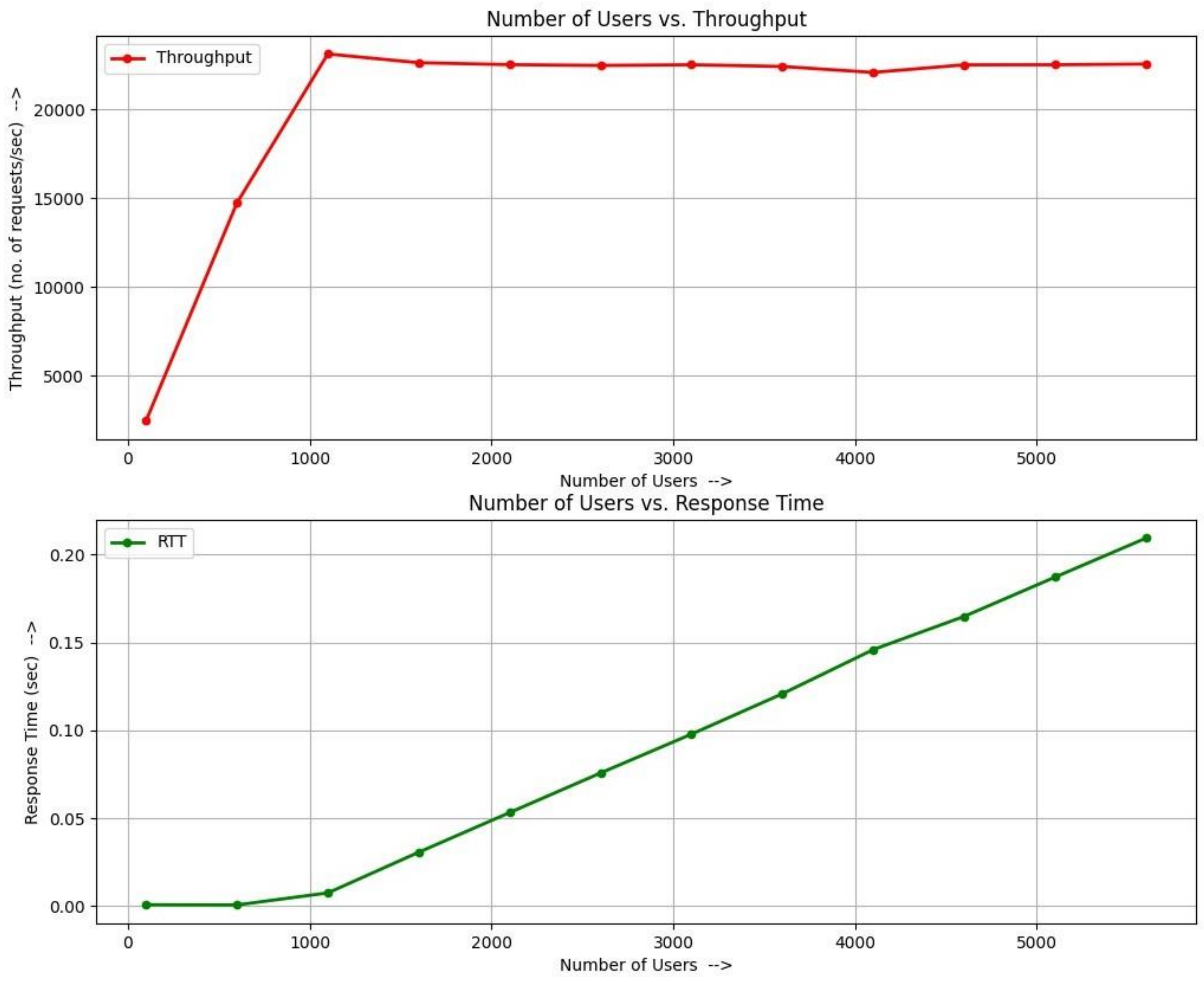
Load Testing Results :

Server is running on core 0.

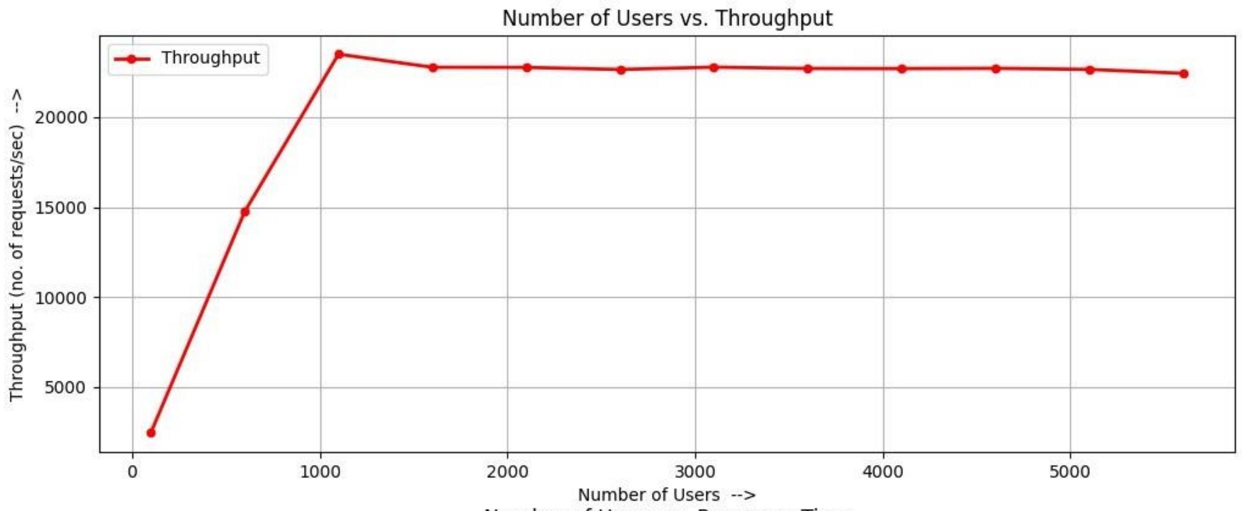
Load Generator is running on core 1,2,3. Below are the graphs snapshots :

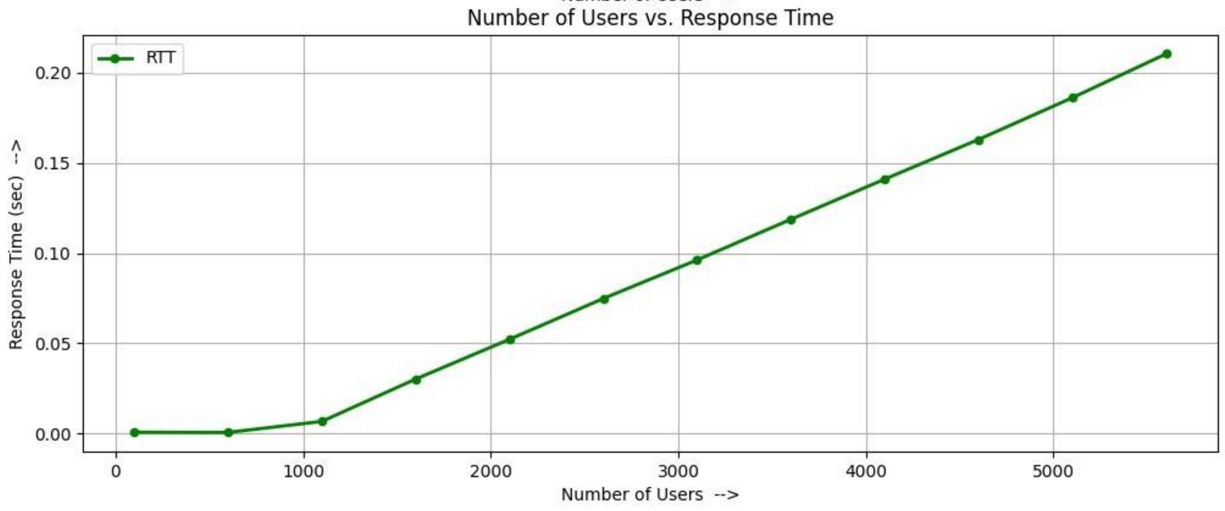
1. Think time: 0.04 sec, Test Duration : 60 sec

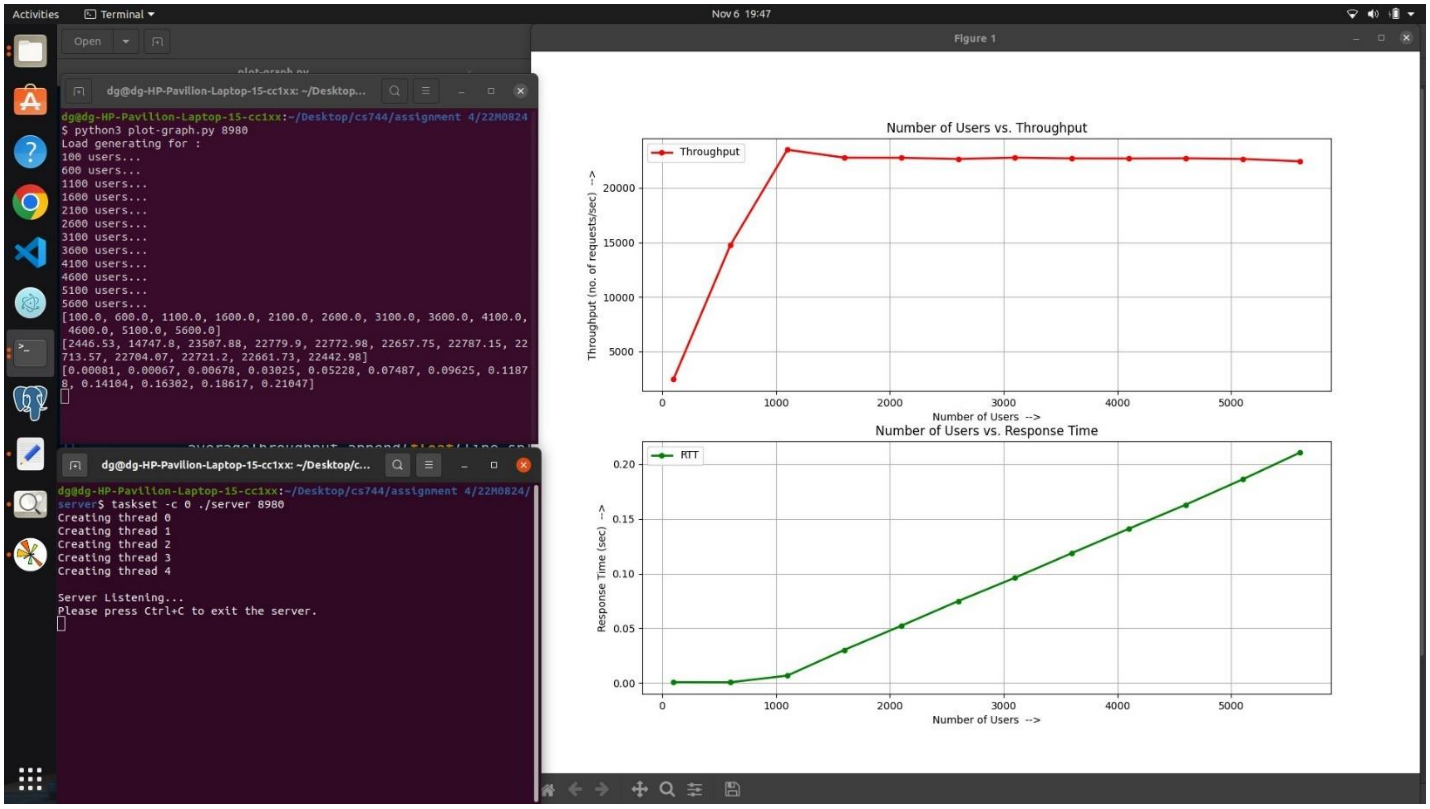
Snapshot 1:



Snapshot :


Number of Users vs. Throughput
Throughput
Throughput (no. of requests/sec) -->
20000
15000
10000
5000
0
1000
2000
3000
4000
5000
Number of Users -->


Number of Users vs. Response Time
RTT
Response Time (sec) -->
0.20
0.15
0.10
0.05
0.00
0
1000
2000
3000
4000
5000
Number of Users -->



Activities
Terminal
Nov 6 19:47
Open
plot-graph.py
dg@dg-HP-Pavilion-Laptop-15-cc1xx: ~/Desktop...
Figure 1
Number of Users vs. Throughput
Throughput
Throughput (no. of requests/sec) -->
Number of Users -->
Number of Users vs. Response Time
RTT
Response Time (sec) -->
Number of Users -->

```
dg@dg-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/cs744/assignment 4/22M0824
$ python3 plot-graph.py 8980
Load generating for :
100 users...
600 users...
1100 users...
1600 users...
2100 users...
2600 users...
3100 users...
3600 users...
4100 users...
4600 users...
5100 users...
5600 users...
[100.0, 600.0, 1100.0, 1600.0, 2100.0, 2600.0, 3100.0, 3600.0, 4100.0,
 4600.0, 5100.0, 5600.0]
[2446.53, 14747.8, 23507.88, 22779.9, 22772.98, 22657.75, 22787.15, 22
713.57, 22704.07, 22721.2, 22661.73, 22442.98]
[0.00081, 0.00067, 0.00678, 0.03025, 0.05228, 0.07487, 0.09625, 0.1187
8, 0.14104, 0.16302, 0.18617, 0.21047]
```

```
dg@dg-HP-Pavilion-Laptop-15-cc1xx:~/Desktop/cs744/assignment 4/22M0824/
server$ taskset -c 0 ./server 8980
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4

Server Listening...
Please press Ctrl+C to exit the server.
```

As this is Closed loop load testing, we can see that our average throughput is increasing along with the number of users and after some saturation point, it is staying nearly same/ throughput flattens (horizontal line in graph). Also, the response time is somewhat linearly increasing with increasing number of users.

Here we can see that the server saturation is happening at around 1100 users. Average throughput corresponding to 1100 users was approximately 23,507 requests/sec and Average Response time was approximately 0.00678 sec.

Also, by using htop, we are able to see that when we run our server on core 0, after a certain point, the core utilization is becoming 100% due to which the average throughput is staying nearly similar after that point and we are getting nearly a horizontal line in our graph. Because of this, we can conclude that CPU is becoming a bottleneck in this case.