# Multithreaded Quiz System

Jaimin Sanjay Gajjar (B20AI014) and Chakshu Anup Dhannawat (B20AI006)
Github Repo: *PCS2-Project*

**Abstract**

This paper reports our experience with building a Multi-threaded Quiz System, using TCP protocol, and evaluating the performance of the overall system. Our work includes analyzing the bottlenecks, and the parameters which lower the system's performance. Thus, making pathways towards making a efficient and optimized system. The Covid Times has made us more closer to the online systems. All the education was shifted to online mode and the current systems were not sufficient to withstand that. That is why we have made a Multi-threaded Quiz system that can serve huge number of clients at the same time.

## I. Introduction

WE understood that in COVID times, there was a urgent need of a better platform that does not crash on receiving huge number of clients. For that we have made a multi-threaded quiz system, that is able to withstand huge number of clients at the same time. We chose a Multi-threaded system as a Multi-threaded systems help us to make system Responsive, Resource optimized and more scalable and economic.

## II. Implementation and Evaluation of the System

### A. Basic Performance of server

To implement the final product, we initially ran a basic multi-threaded server, and then later on came up with the idea of using locust.io to test the performance of our server. After evaluation, we can see that we are not able to serve many clients, thus we have come up with the following solution to get maximum efficiency: We will initially get the maximum number of clients that our server can handle, then we will set a threshold on the number of clients, and when the number of clients reach that threshold, the new clients will be scheduled in Shortest-Job First(SFS) scheduling scheme based on the number of questions in the quiz which the client wants to give.

### B. Using Django for web deployment

As Django uses a multi-threaded server on its backend, we have used it, as it eases the process of web deployment and handling multiple web pages of the online quiz system.

In the website, we have 3 instances, 'Admin', 'Teacher' and 'Student'. 'Admin' is a superuser of the server. 'Teacher' will add the question and make a quiz out of those questions.'Student' will login from their portal, and attempt the quiz.(Possible multiple times). Every data generated, whether it be login credentials, quiz questions, or results, will be stored in SQLite3 Database. The server then fetches data from the database, and performs the necessary evaluation, and returns the marks of the student.

Some screenshots of the web interface are shown below:

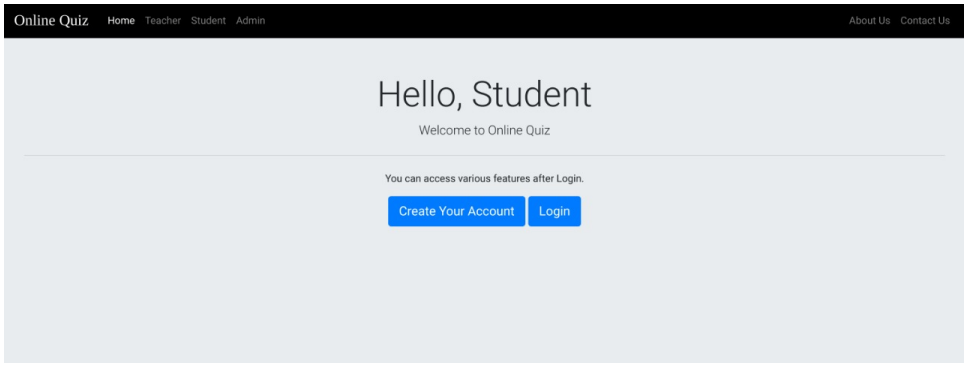



Fig. 1: Student Login Page

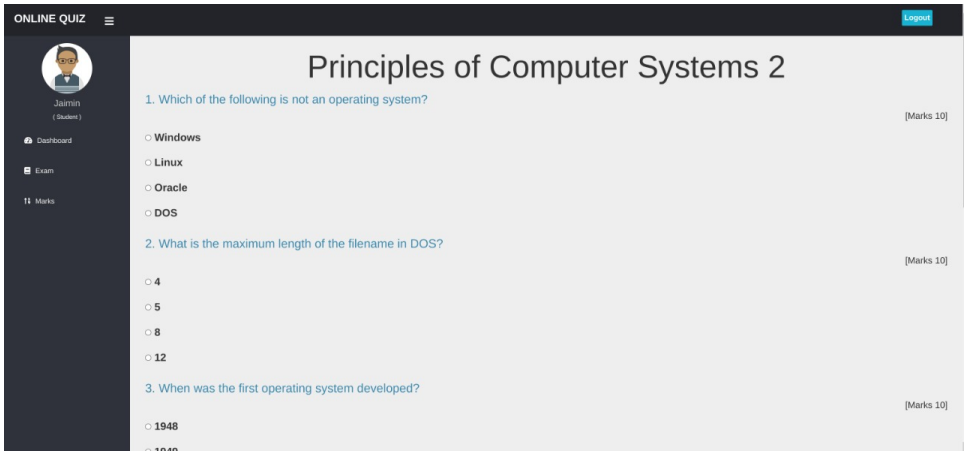



Fig. 2: Student Quiz Page

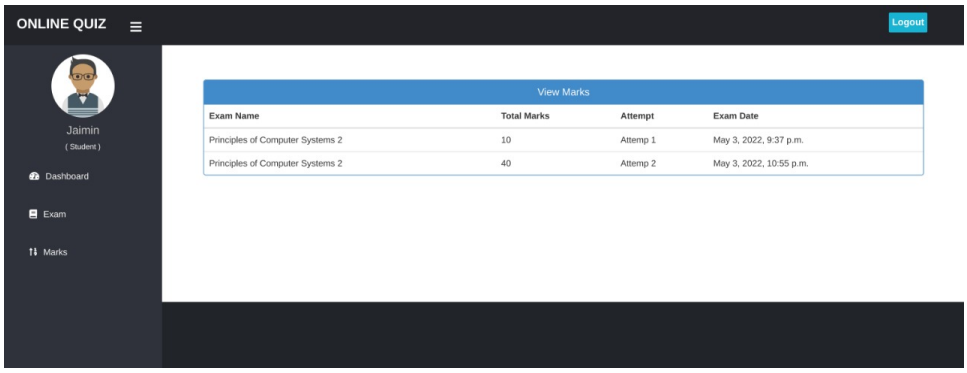


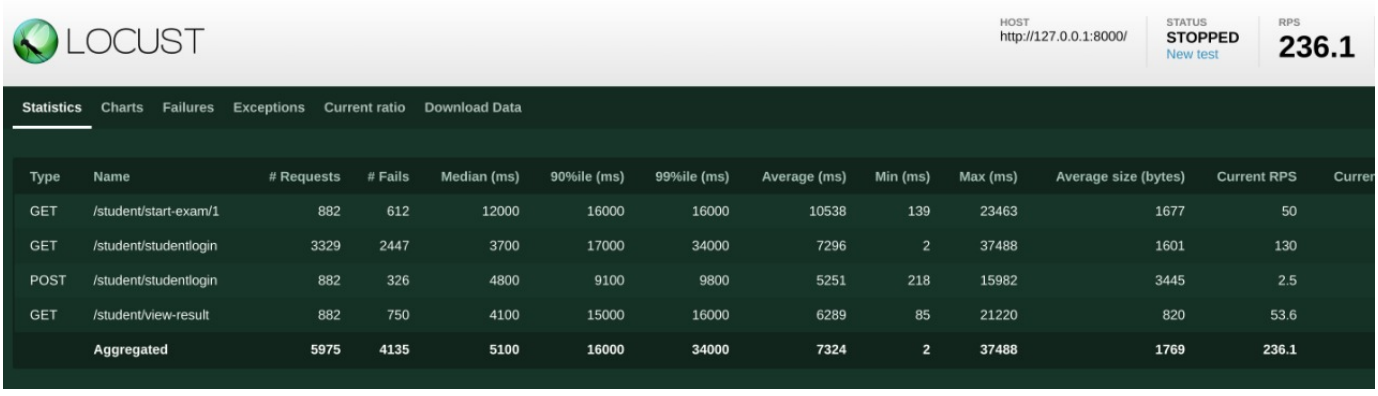

Fig. 3: Student View Marks Page

## III. Evaluation Metrics Used

For evaluating the performance of the server, we used the python library, 'locust'. Locust is an open source load testing tool in which we can define the client's behaviour, and can swarm the system with huge number of clients.

We tried to mimic the steps a student will take to attempt a quiz.The steps will be:

- GET the student login page.
- Enter his/her login credentials and generate a POST request.
- Attempt the quiz.
- GET the results.




| Type | Name | # Requests | # Fails | Median (ms) | 90%ile (ms) | 99%ile (ms) | Average (ms) | Min (ms) | Max (ms) | Average size (bytes) | Current RPS | Curren |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GET | /student/start-exam/1 | 882 | 612 | 12000 | 16000 | 16000 | 10538 | 139 | 23463 | 1677 | 50 | |
| GET | /student/studentlogin | 3329 | 2447 | 3700 | 17000 | 34000 | 7296 | 2 | 37488 | 1601 | 130 | |
| POST | /student/studentlogin | 882 | 326 | 4800 | 9100 | 9800 | 5251 | 218 | 15982 | 3445 | 2.5 | |
| GET | /student/view-result | 882 | 750 | 4100 | 15000 | 16000 | 6289 | 85 | 21220 | 820 | 53.6 | |
| | **Aggregated** | **5975** | **4135** | **5100** | **16000** | **34000** | **7324** | **2** | **37488** | **1769** | **236.1** | |

Fig. 4: Mimicking user's Behaviour

The evaluation metrics which we used are as follows:

### A. Requests per second (RPS)

A web server's primary function is to receive and process requests, but if your server becomes overloaded with requests, performance can suffer. RPS is a metric which calculates the number of requests received during a specified monitoring period.

### B. Average response time (ART)

Average response time measures the length of request/response cycles, so that you can assess the average amount of time the application takes to generate a response from the server.

### C. Number of Users

It shows the number of users using the web-page at a given time.

## IV. Results and Conclusion

When we are running the django server, the website is running smoothly for a single client, and the server is also returning the marks correctly.

Now we have used Locust to swarm the web server with multiple clients, which give quizzes concurrently(as the server is multi-threaded) We have used peak users=10000, and swarm rate=100 users/sec. Thus, a maximum of 10000 clients will exist at any given time, and 100 new users will be added every second

The results are as follows:

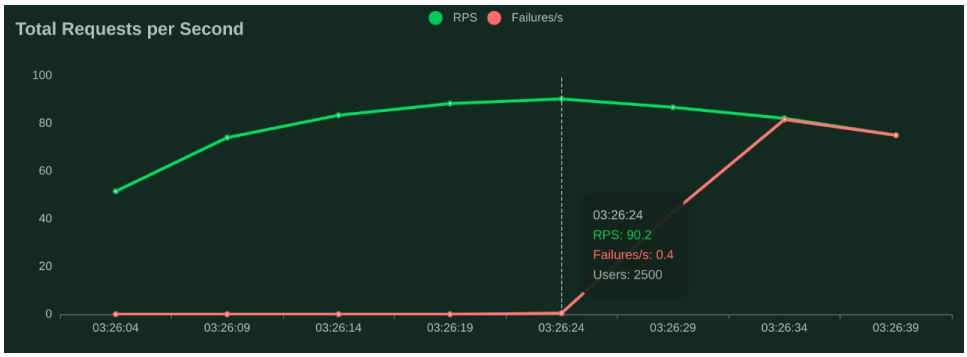



Fig. 5: Total Requests per seconds

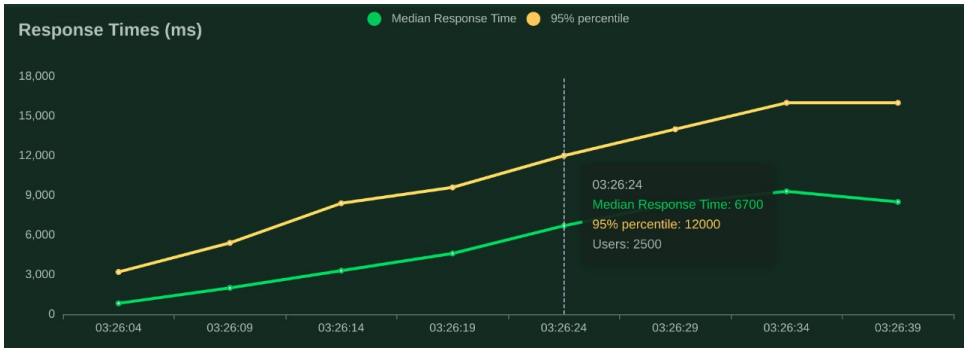



Fig. 6: Average response time

s

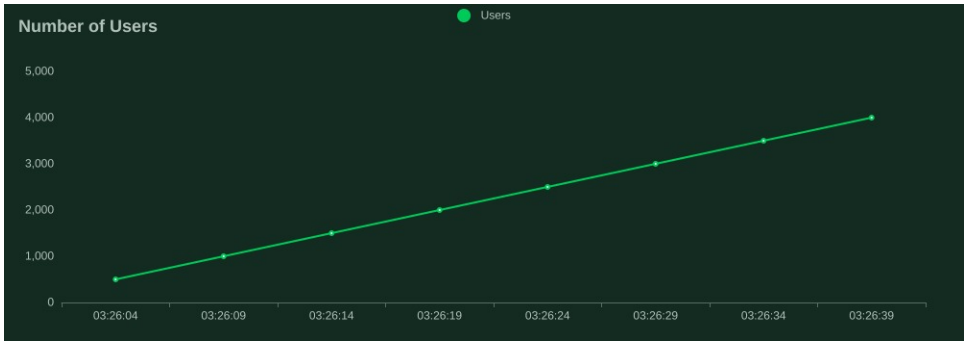



Fig. 7: Number of Users

After looking at the graphs, we can infer that, our server performs well till RPS is 90.2, and the number of users reach 2500. Beyond that, failure starts to occur, and increases linearly.

We can also see that the Median Response time is increasing in a linear fashion as the number of clients are increasing.

The server was running on our local machine which had RAM of 8 GB, Thus, we can see that the amount of parallel processing is highly dependent on the resource constraints of the server.

In our case, multiple threads exist and each of them needs access to the database to read/write operations, throughout its lifetime. Threads share an address space, having many active threads leads to diminishing returns due to lock contention. So in our case(having a database), many processes reading tables can proceed simultaneously, yet updates of dependant data need to be serialised to keep data consistent which may cause lock contention and limit parallel processing.

We have learnt that on memory intensive computations, cache sizes may become important, fewer threads whose data stays in cache and CPU pre-fetches minimise stalls, work better than threads competing to load their data from main memory.

One must select the optimal number of threads for the desired task, as either having too low number of threads or high number of threads will cause the system to perform poorly.

## REFERENCES

[1] https://github.com/muneeb50/Multithreaded-Online-Quiz-System
[2] https://locust.io/
[3] https://link.springer.com/chapter/10.1007/BFb0024207?noAccess=true
[4] https://dl.acm.org/doi/abs/10.1145/232974.232994

## CONTRIBUTION

The thought process and planning were done together as a team, the individual contributions are mentioned below:

- Jaimin Sanjay Gajjar (B20AI014) - He wrote the backend part(django webapp server side), helped in recognizing user behaviour and mimicking it. He also contributed in making the report and ppt.
- Chakshu Anup Dhannawat(B20AI006) -
  He coded the frontend part of the website of the multi-threaded quiz system. Coded the locust file which mimicks the client behaviour. Evaluated the performance of the server. He also contributed in making the report and ppt.