REPORT SOFTWARE PERFORMANCE AND SCALABILITY

Davide Bresaola 897180
Francesco Vinci 868164

2022/2023

Contents

1 Introduction

In this report we present the design of a web application and the study of its performance through benchmarks performed with the Tsung test tool.

1.1 Hardware

The server is hosted on a Lenovo C365 with the following characteristics:

- **CPU**: AMD E1-2500 (1.4 GHz, 2-core)
- **RAM**: 4GB
- **HDD**: 500GB
- **Network**: Wired Ethernet

1.2 Software

The host runs **Ubuntu Server** 22.04.03 LTS as native OS (no VM) with a minimal version of ubuntu-desktop as GUI to avoid degrading memory/CPU usage performance, considering the low specs of the machine, and consequently polluting tests.

2 Implementation

We are required to create a web application that allows a user to search for a movie in the IMDB database and provide information about the movie, main actors and directors, average rating, running time, etc.

We have implemented a web page that allows the user to enter the title he intends to search for and an API with a single endpoint that takes care of querying the database. The API returns a JSON-encoded response.

2.1 Nextjs

Our web application was implemented using **Next.js**.
Next.js is an open-source web development framework for building full-stack web applications providing React-based web applications with server-side rendering and static website generation.

Furthermore, Nextjs **Route Handlers** allow us to create custom request handlers for a given route allowing us to create REST APIs.

Next.js requires **Node.js** to work. Nodejs is a JavaScript runtime environment using an event-driven, non-blocking I/O model built on Chrome's V8 javascript engine, designed to build scalable network applications. It is used as backend service.

Node.js uses the concept of **Libuv**. Libuv is an open-source library built-in C. It has a strong focus on asynchronous and I/O, this gives node access to the underlying computer operating system, file system, and networking.
Libuv implements extremely important features of Node.js:

- **Event loop**: contains a single thread and is responsible for handling easy tasks like executing callbacks and network I/O. EventLoop is the heart of node.js.
- **Event queue**: As soon as the request is sent the thread places the request into a **FCFS** queue (Event queue).The process will emit event as soon as they are done with the work and event loop will pick up these events and call the callback functions that are associated with each event and response is sent to the client.
- **Thread pool**: When non-blocking requests are accepted there are processed in an event loop, but while accepting blocking requests it checks for available threads in a thread pool, assigns a thread to the client's request which is

then processed and send back to the event loop, and response is sent to the respective client.

All of this is summarized in Figure 1.

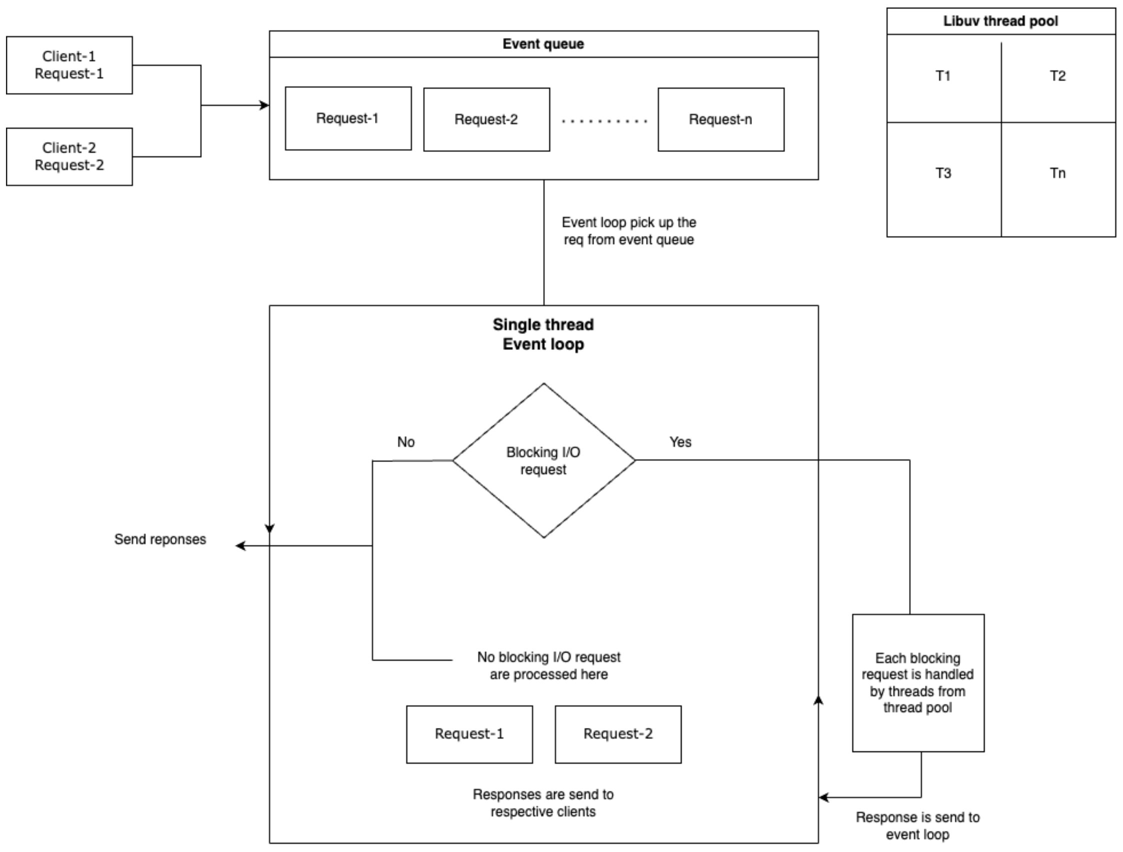

Client-1 Request-1
Client-2 Request-2
Event queue
Request-1
Request-2
Request-n
Libuv thread pool
T1
T2
T3
Tn
Event loop pick up the req from event queue
Single thread Event loop
No
Blocking I/O request
Yes
Send reponses
No blocking I/O request are processed here
Request-1
Request-2
Responses are send to respective clients
Each blocking request is handled by threads from thread pool
Response is send to event loop

Figure 1: Working of Node.js

2.2 PostgreSQL

As database we have chosen **PostgreSQL** as it is a powerful object-relational database system. It is also recommended by nextjs developers as well. The power of postgres lies in its architecture, reliability, data integrity, robust feature set, extensibility as well as running on all major OS.

PostgreSQL efficient performance greatly depends on CPU usage. Complex operations such as comparison, table joins, hashing, data grouping and sorting require more processor time.

As a ”bridge” between our application and the database we use **Prisma** ORM (Object Relational Mapping). Prisma is an open source next-generation ORM that can be used to access a database in Node.js and TypeScript applications. In our case we communicate with the database in TypeScript as the app is written with this language.

During the various tests we noticed some errors due to timeouts. These were due to the short time that the prisma query engine gave to queries to be processed, in fact the default value was 10sec. This value would essentially be sufficient if the server was powerful enough and the pool size was large enough. In fact, the default pool size is 5. The pool size allows the query engine to process a larger number of queries in parallel. So we decided to disable the pool timeout prevents the query engine from throwing an exception after x seconds of waiting for a connection and allows the queue to build up and also to increase the pool size to 40 parallel queries.

This essentially means that we will see at most 40 forks of the postgres process when we look at usage statistics (e.g. with the top command), obviously only in case of a high amount of requests.

```
datasource db {
  provider = "postgresql"
  url      =
   ↪  "postgresql://postgres:password@localhost:5432/mydb?connection_limit=40
   ↪  &pool_timeout=0"
}
```

2.3 Application

Our application consists of a web page for the user and a REST API with a single endpoint that is used by the web page or can be used independently.

As mentioned earlier the app was developed with Next.js 13. This new version of Next.js allows to create APIs easily. Anything inside the `app/api` folder will be an externally reachable endpoint. So in our case we created a `route.ts` file inside the `api/movie` folder.

```
//app/api/movie/route.ts

import { NextResponse } from "next/server"
import { PrismaClient } from "@prisma/client"
const prisma = new PrismaClient();

export async function GET(request: Request) {
    const { searchParams } = new URL(request.url);
    const titleName = searchParams.get("title");

    if (!titleName) {
        return NextResponse.json({
            response_code: 400,
            error: true,
            message: "Bad request",
            data: []
        });
    }

    const title = await prisma.title_basics.findFirst({
        where: {
            primarytitle: { contains: titleName },
            title_type: "movie",

        },
        select: {
            primarytitle: true,
            start_year: true,
            runtime_minutes: true,
            genres: true,
            title_crew: {
                select: {
                    directors: true,
                    writers: true,
                }
            },
            title_principals: {
                where: {
                    OR: [
                        { category: "actor" },
                        { category: "actress" },
                    ]
                },
                select: {
                    name_basics: {
                        select: {
                            primary_name: true,
                        }
                    }
                },
            },
            title_ratings: {
                select: {
                    average_rating: true,
                    num_votes: true,
                }
            }
        },
    });

    if (!title) {
        return NextResponse.json({
            response_code: 404,
            error: true,
            message: "Title not found",
            data: []
        });
    }

    if (title.title_crew?.directors) {
        const directorsSet = title.title_crew.directors.split(",");

        const directorsData = await prisma.name_basics.findMany({
            where: {
                nconst: { in: directorsSet }
            },
            select: {
                primary_name: true,
            }
        });
        title.title_crew.directors = directorsData as any;
    }

    if (title.title_crew?.writers) {
        const writersSet = title.title_crew.writers.split(",");

        const writersData = await prisma.name_basics.findMany({
            where: {
                nconst: { in: writersSet }
            },
            select: {
                primary_name: true,
            }
        });
        title.title_crew.writers = writersData as any;
    }

    return NextResponse.json({
        response_code: 200,
        error: false,
        message: "Title found",
        data: title
    });
}
```

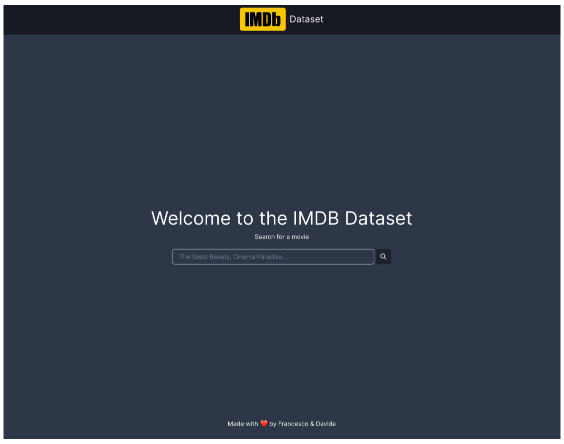
The whole request is expensive in fact we do a query with three joins and then two more queries. As we can see in line 20 the first query is executed.

We use prisma's `findFirst()` function which returns the first result among those found. We only search among `title_types` that are `"movie"` and we search among titles that **contain** user input, so we can get more than one match, for this reason we use `findFirst()`.

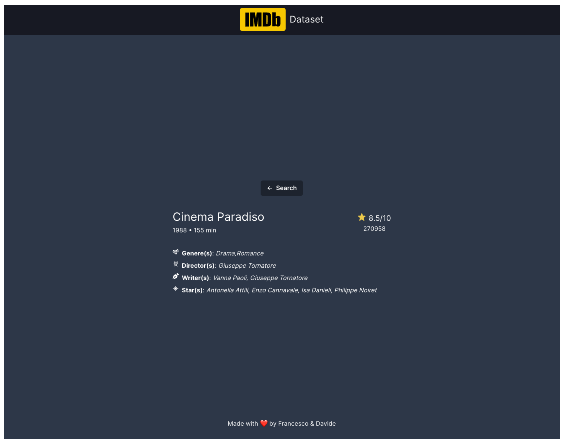
Then, when the query completes, we take the result and run two more queries to get the information about the director(s) and the writer(s). Unfortunately it was not possible to do everything in one query as the csv files provided by IMDB did not allow it.

We can then query the databases as follows:

`http://localhost:3000/api/movie?title=Cinema Paradiso`


IMDb Dataset
Welcome to the IMDB Dataset
Search for a movie
The Great Beauty, Cinema Paradiso...
Made with ❤️ by Francesco & Davide

(a) Search page



Cinema Paradiso
1988 • 155 min
★ 8.5/10
270958

Genere(s): *Drama,Romance*
Director(s): *Giuseppe Tornatore*
Writer(s): *Vanna Paoli, Giuseppe Tornatore*
Star(s): *Antonella Attili, Enzo Cannavale, Isa Danieli, Philippe Noiret*

(b) Movie page

Figure 2: Webapp

3 Queueing Network

After we implemented our application we wanted to test its scalability and performance. In order to do so we used Tsung, an open-source multi-protocol distributed load testing tool that simulates users interactions with our application.

To perform the benchmarks we need two different machines: the System Under Test (SUT), that in our case is the machine with Ubuntu server, and a testing machine (Client) used to stress the SUT.
The Client is hosted on a MacBook Air with macOS Ventura 13.5 with the following hardware:

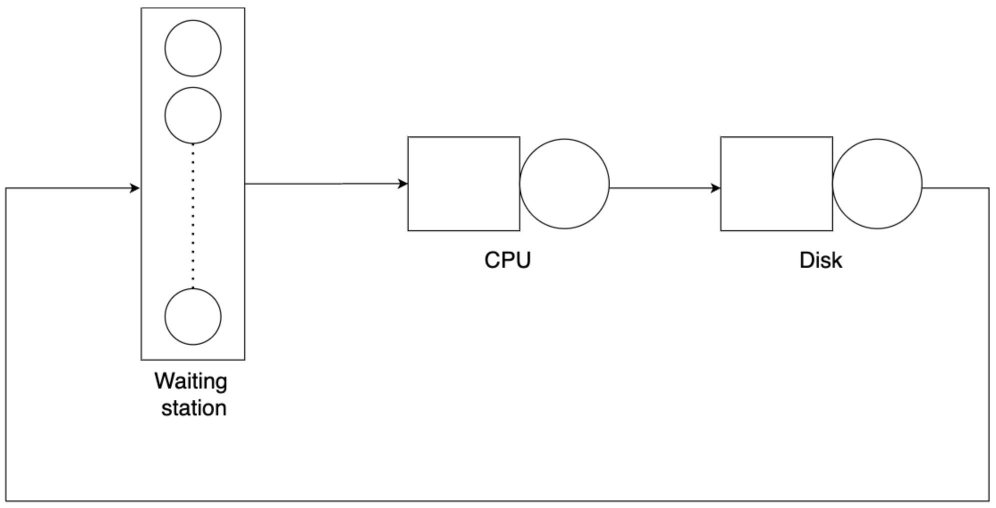
- **CPU**: Apple M1
- **RAM**: 8GB
- **HDD**: 256GB
- **Network**: Wired Ethernet

3.1 Components

The main components of our system are the **CPU** and the **Disk**. The system behaves as follows:

- The request is received and the CPU takes care of handling it. It is inserted into the Nodejs Event queue (FCFS) and the Event loop will take care of taking it and executing it. Being a blocking event, the request is placed in another queue and will be assigned a thread from the thread pool.
- The data is retrieved from the database, then a disk read occurs. Since each request is blocking it will be assigned a thread and thanks to the changes made previously postgres can handle 40 queries simultaneously. Once the request is complete, the response is sent to the client.

We decided to model the system like this:



3.2 Service Time

The SUT and the Client are connected via Ethernet to the same LAN.
To perform the load test we followed the following steps:

- Use tsung-recorder to record one user session
- Configure the file XML file that describes our benchmarking experiment
- Start the load test and wait its finish
- Analyse the results

To carry out tests with a large number of users it was necessary to add the `maxusers` parameter in the `client tag`, this is because in the final report by tsung many requests encountered the error `error_connect_emfile`.

This happens usually when we set a high value for `maxusers`. The errors means that we are running out of file descriptors; we must check that `maxusers` is less than the maximum number of file descriptors per process in our system (with `ulimit -n`, 256 by default in macOS).
We can either raise the limit of our operating system or decrease `maxusers`.

In fact the `maxusers` parameter is used to bypass the limit of maximum number of sockets opened by a single process. When the number of users is higher than the limit, a new erlang virtual machine will be started to handle new users. The default value of `maxusers` is 800.
So we changed the maximum number of file descriptors per process in our system, with `ulinit -n` and set `maxusers` to 30000 as recommended in Tsung's

documentation.

But first, to find the service time of the system, we performed a closed-loop test with only one user in the system so that the time spent in the waiting station will be zero.

We have a test duration of 30 minutes. During this period, 1 user per second is generated in the first 10 minutes, but since we want only one user we set `maxnumber` to 1.

For testing we created a csv file containing 10000 of the most popular movies. So every request made by the user will be different and every movie searched will be taken from the file.

The XML configuration file used by Tsung is the following:

```
...
    <load duration="30" unit="minute">
        <arrivalphase phase="1" duration="10" unit="minute">
            <users interarrival="1" unit="second" maxnumber="1"/>
        </arrivalphase>
    </load>
    <options>
        <option name="file_server" id="filmlist" value="./film.csv"/>
    </options>
    <sessions>
        <session name="query_session" probability="100" type="ts_http">
            <for from="1" to="10000" var="i">
                <setdynvars sourcetype="file" fileid="filmlist" delimiter=";"
                ↪  order="iter">
                    <var name="title" />
                </setdynvars>

                <request subst="true">
                    <http url="/api/movie?title=%%_title%%" version="1.1"
                    ↪  method="GET">
                        <http_header name='Accept' value='*/*' />
                        <http_header name='Accept-Encoding' value='gzip,
                        ↪  deflate, br' />
                        <http_header name='Accept-Language'
                        ↪  value='en-US,en;q=0.5' />
                    </http>
                </request>
            </for>
        </session>
    </sessions>
...
```

So with a closed loop test with one user in the system, we have that the service time is 2.87 seconds.

Now let's calculate the **mean service time** of the Disk and the CPU. To do this we need to know the **percentage of time** the Disk and CPU are used. For this purpose we performed a test with tsung and observed the behavior of the system with `top` and `iotop` and obtained approximately the following results.

Disk	**CPU**
97%	3%

As might be expected from this type of service, most of the job's service time is spent on disk. So now we can calculate the mean service time for both in components.

$$\text{(total service time)} \cdot (\% \text{ of time in station } i)$$

	Mean service time
CPU	86.1 ms
Disk	2783.9 ms

So the service rate are:

$$\mu_{CPU} = \frac{1}{86.1} = 0.01161\ ms$$
$$\mu_{Disk} = \frac{1}{2783.9} = 0.0003592\ ms$$

3.3 Service demand and bottleneck

Since we are in a closed system to find the relative visit ratio we have to solve the following system of traffic equations:

$$\begin{cases} e_0 = e_2 \\ e_1 = e_0 \\ e_2 = e_1 \end{cases} \tag{1}$$

We set the waiting station as the reference station and suppose $e_0 = 1$. Thus we obtain the relative visit ratio for each stations.

$$\begin{cases} e_0 = 1 \\ e_1 = 1 \\ e_2 = 1 \end{cases} \tag{2}$$

Now let's calculate the service demand ($D_i = \frac{V_i}{\mu_i}$).

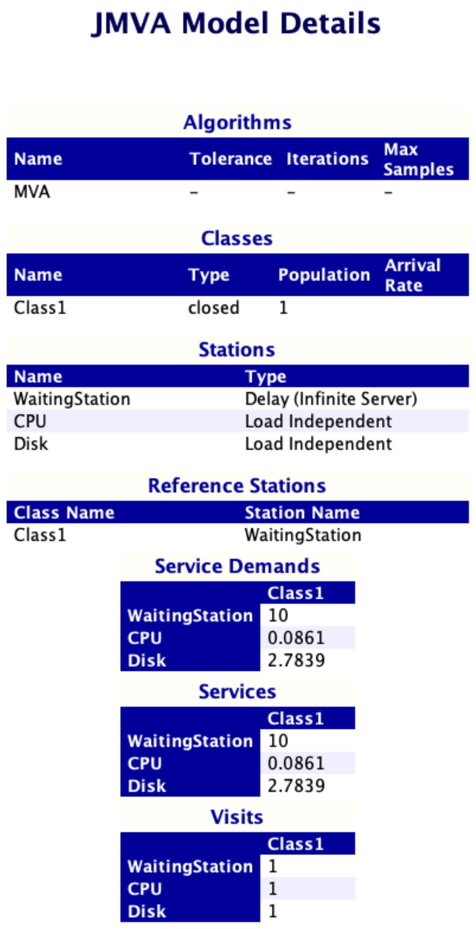
$$D_{CPU} = \frac{e_1}{\mu_{CPU}} = \frac{1}{0.01161} = 86.132$$
$$D_{Disk} = \frac{e_2}{\mu_{Disk}} = \frac{1}{0.0003592} = 2783.96$$

The station with the highest service demand is the **bottleneck**, so in this case is the **Disk**.

4 Model validation

Now that we have all the data we need we can start analyzing our model with the MVA (Mean Value Analysis) using the JMT software. We are going to calculate what is the expected response time according to the number of users and what is the optimal number of users.

The JMT model was configured in this way:



JMVA Model Details

Algorithms

Name	Tolerance	Iterations	Max Samples
MVA	-	-	-

Classes

Name	Type	Population	Arrival Rate
Class1	closed	1	

Stations

Name	Type
WaitingStation	Delay (Infinite Server)
CPU	Load Independent
Disk	Load Independent

Reference Stations

Class Name	Station Name
Class1	WaitingStation

Service Demands

	Class1
WaitingStation	10
CPU	0.0861
Disk	2.7839

Services

	Class1
WaitingStation	10
CPU	0.0861
Disk	2.7839

Visits

	Class1
WaitingStation	1
CPU	1
Disk	1

Figure 3: JMT model

We chose 10 seconds of service time for the waiting station and for the what-if analysis we set from 1 to 50 customers, with a step of 20.

4.1 Utilisation

In the graph we can see the CPU (blue) and Disk (cyan) station utilizations. When saturation is reached the CPU utilization becomes stable, but the one that reaches 1.00, therefore Disk, is the station that reaches saturation first, which is the system bottleneck.

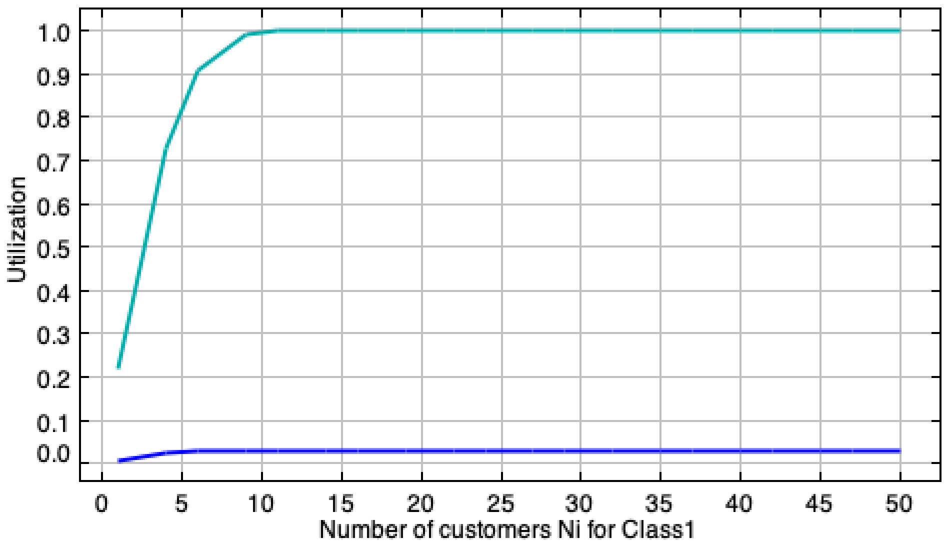


Figure 4: Utilization

4.2 Response Time

In the graph we can see the response time for each station, Waiting station (red), CPU (blue) and Disk (cyan). We can see that the waiting station is constant because it is the infinite server station. The CPU does not reach saturation because the system bottleneck prevents it from growing further, in fact it could scale more but from 9 customers it maintains a costant response time. While Disk, as we can see, has an exponential growth until the queue reaches about 5-10 clients and then grows linearly following an oblique asymptote, this is because it starts to become saturated.

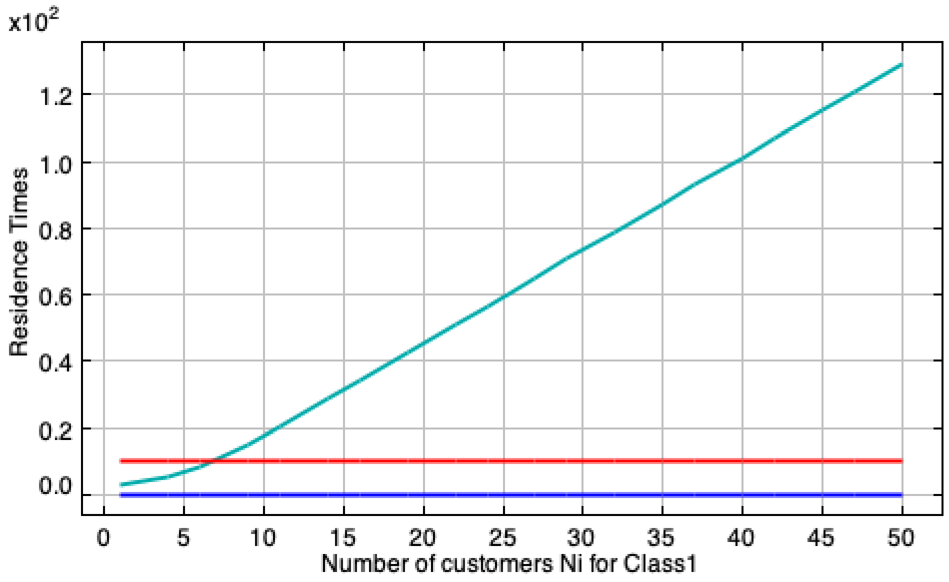


Figure 5: Response time

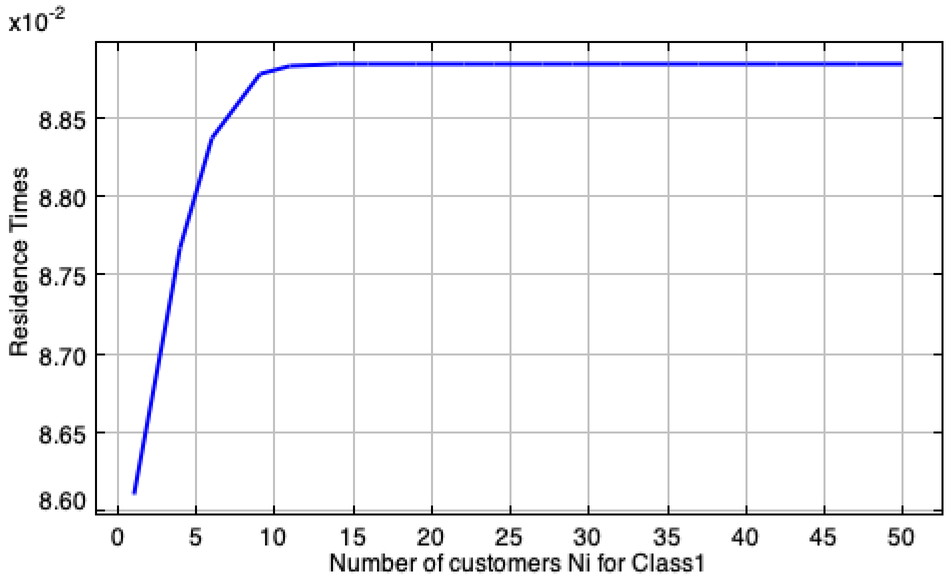


Figure 6: CPU response time

4.3 Throughput

As we have seen we are able to manage from 5 to 8 customers before the performance starts to degrade and reach saturation at 10 customers. This is well evidenced by the throughput. The graphs of the three components in this case coincide as the throughput is given by the bottleneck, i.e. the disk.

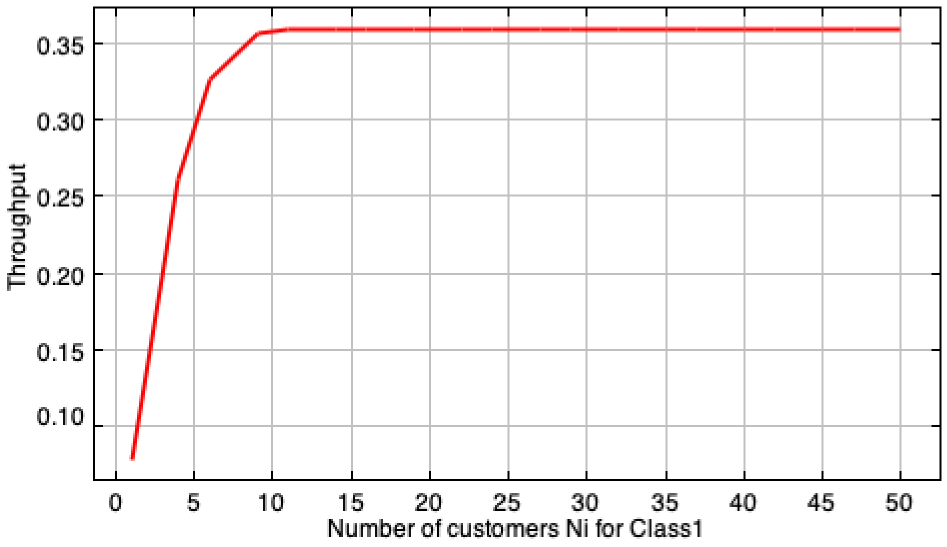


Figure 7: Throughput

4.4 Number of users

We observe that as the number of customers in the waiting station (red), which is the infinite server station, increases, there will be many jobs in the thinking phase but when the system reaches saturation the jobs tend to spend more time inside the component than in the thinking phase. Since they are there because they wait too long to be served they cannot be in the thinking phase. This is why when we increase the number of jobs in the system above a certain point (just before saturation) the number of jobs in the thinking phase increase very slowly.

The CPU which is not the bottleneck as we can see has the same problem, when the system is saturated we see that there are only 3.2 clients on average which is a relatively low number.

On the other hand, the Disk, which is the bottleneck, as the number of jobs increases, these are accumulated and therefore they will start spending more time in the Disk queue than in the CPU.

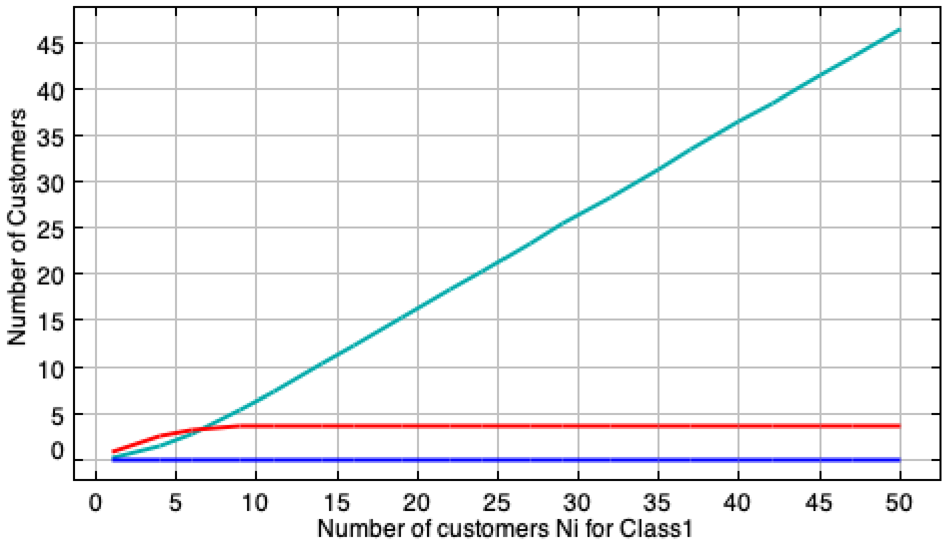


Figure 8: Number of customer

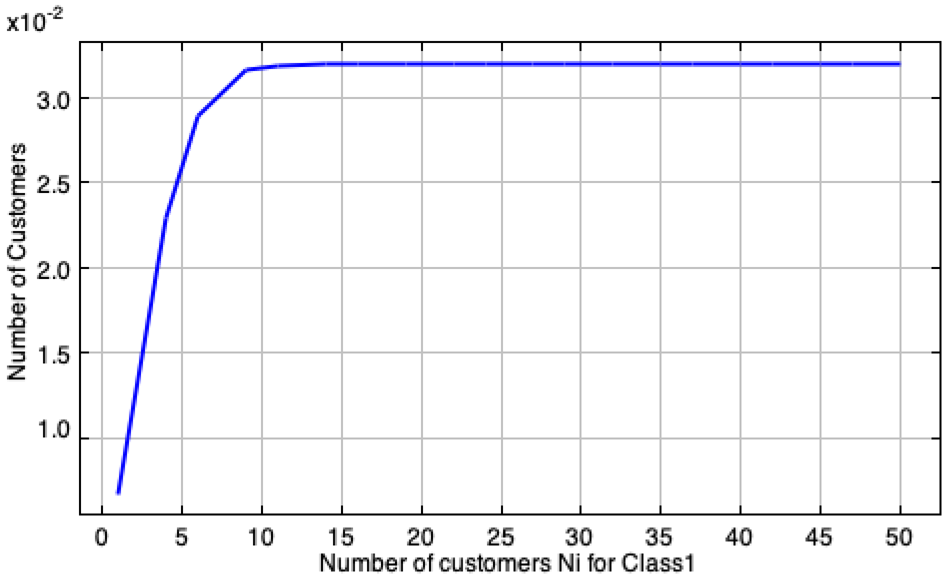


Figure 9: CPU Number of customer

5 Conclusion

While completing this assignment, we acquired valuable insights into systems modeling and performance analysis, that is crucial when deploying large systems that require efficient scaling:

- Benchmarking analysis is essential when aiming for scalability. It involves careful planning, considering the expected user load.
- Building a robust analytical model is crucial for comprehending performance patterns and interpreting benchmark results accurately.
- Inefficient web applications can consume more server resources and bandwidth than necessary, leading to increased hosting and operational costs. Performance analysis helps optimize resource usage, potentially reducing operational expenses.
- Performance analysis helps identify bottlenecks in web applications, whether they are related to server performance, database queries, frontend rendering, or third-party integrations. Addressing these bottlenecks can significantly improve overall performance.
- Load Testing: Load testing during performance analysis simulates various levels of user traffic to evaluate how an application behaves under different loads. This testing helps predict how the application will perform in real-world scenarios and allows for proactive adjustments to handle traffic spikes.

We discovered that even when modeling a seemingly simple system, numerous considerations and assumptions must be carefully evaluated. This necessitated a rigorous exploration of the theoretical concepts we had studied, along with testing various configurations. Ultimately, the trial-and-error approach proved instrumental in achieving a practical model that closely aligned with the theoretical findings.