Ca' Foscari University of Venice


Department of Environmental Sciences,
Informatics and Statistics

MCS Computer Science and Information Technology CM-90

[CM0481] Software Performance and Scalability

---

# IMDb: Performance and Scalability Analysis

---

*Authors:*
Michele Lotto - 875922
Daniel Jader Pellattiero - 903837

Rev. 26/06/2024

# Contents

# List of Tables

# List of Figures

# Listings

# Abstract

This report presents a comprehensive account of the implementation and performance testing process conducted on a web application that emulates the IMDb website[1].

The report begins with an overview of the system implementation, in which the technologies utilized are outlined. Subsequently, a theoretical analysis of the queueing system network is presented, in which the implemented system is characterised. The load test conducted on the system will be subjected to a thorough examination, including an analysis of its theoretical and empirical findings.

In conclusion, a potential architectural solution is evaluated for its potential to enhance the scalability of the tested system.

It should be noted that the implementation of the system, along with all automated programs and scripts referenced in this report, is fully documented and accessible on the GitHub repository[2].

[1] https://www.imdb.com/

[2] https://github.com/danieljaderpellattiero/unive-imdb

# Chapter 1

# System implementation

The project entails the development of a three-tiered web application, comprising a database, a RESTful API and a website. The following sections will present an analysis of each of these components, with a focus on the technologies and design choices implemented.

## 1.1 Database – MongoDB

MongoDB is a popular NoSQL document-oriented database that is designed to store and manage large volumes of structured and unstructured data.

The database was selected as the optimal choice for managing the IMDb Non-Commercial Datasets[1] due to its adoption of a schemaless data modelling, which enables the management of any non-normalised records or fields within the data. Furthermore, the usage of JSON format for storing the records allows for enhanced efficiency in interactions with the backend.

### 1.1.1 Data analysis

Each dataset, comprising a UTF-8-encoded tab-separated values (TSV) file, was subjected to analysis using the Python Pandas library. The primary steps of the data analysis process were as follows:

1. The replacement of missing values (denoted by `'\N'`) with default values of a type consistent with the column domain to which they belong.

2. The removal of records that lack some data fields.

3. The normalisation of fields containing arrays of elements.

4. The potential replacement of table indexes.

Further modifications were implemented to the tables, however, as these affect the database schema, they will be addressed subsequently.

Once the preliminary phase of preparing the datasets was complete, it was decided that the files should be exported in the JSON format, allowing them to be imported into the database. Additionally, the files were exported also in the Parquet format, enabling

[1] https://developer.imdb.com/non-commercial-datasets/

them to be uploaded to the GitHub repository via the GitHub Large File Storage (LFS) facility[2].

### 1.1.2 Collections' schema

MongoDB represents objects using BSON (Binary JSON) types, which are binary-encoded serialisations of documents that adhere to the JSON format. As each table in MongoDB is translated into the concept of a document collection, the diagrams of the respective collections imported from the datasets obtained post data analysis are presented below. Furthermore, supplementary notes on the refactoring of the tables are provided.

Table 1.1: '*title.basics*' collection schema

| Field | BSON type | Index | Index type |
|---|---|---|---|
| _id | String | yes | unique, ascending |
| titleType | String | no | - |
| name | String | no | - |
| nameEng | String | no | - |
| isAdult | Boolean | no | - |
| genres | Array[String] | no | - |
| startYear | 32-bit integer | no | - |
| endYear | 32-bit integer | no | - |
| runtime | 32-bit integer | no | - |
| rating | Double | no | - |
| votes | 32-bit integer | no | - |

*N.d.R.*: A collection was created ad hoc for records with the field *'titleType'* equal to *'tvEpisode'*.

Table 1.2: '*title.akas*' collection schema

| Field | BSON type | Index | Index type |
|---|---|---|---|
| _id | ObjectId | yes | unique, ascending |
| titleId | String | no | - |
| ordering | 32-bit integer | no | - |
| region | String | no | - |
| name | String | no | - |
| nameLower | String | yes | ascending |

*N.d.R.*: *'language'*, *'types'*, *'attributes'*, *'isOriginalTitle'* fields dropped.

[2]https://git-lfs.com/

Table 1.3: '*title.crew*' collection schema

| Field | BSON type | Index | Index type |
|---|---|---|---|
| _id | String | yes | unique, ascending |
| writers | Array[String] | no | - |
| directors | Array[String] | no | - |

Table 1.4: '*title.episodes*' collection schema

| Field | BSON type | Index | Index type |
|---|---|---|---|
| _id | String | yes | unique, ascending |
| titleId | String | yes | ascending |
| name | String | no | - |
| nameEng | String | no | - |
| season | 32-bit integer | no | - |
| episode | 32-bit integer | no | - |
| isAdult | Boolean | no | - |
| genres | Array[String] | no | - |
| startYear | 32-bit integer | no | - |
| endYear | 32-bit integer | no | - |
| runtime | 32-bit integer | no | - |
| rating | Double | no | - |
| votes | 32-bit integer | no | - |

Table 1.5: '*name.basics*' collection schema

| Field | BSON type | Index | Index type |
|---|---|---|---|
| _id | String | yes | unique, ascending |
| fullName | String | no | - |
| birth | 32-bit integer | no | - |
| death | 32-bit integer | no | - |
| professions | Array[String] | no | - |

*N.d.R.*: *'knownForTitles'* field dropped.

Table 1.6: '*title.principals*' collection schema

| Field | BSON type | Index | Index type |
|---|---|---|---|
| _id | ObjectId | yes | unique, ascending |
| titleId | String | yes | ascending |
| ordering | 32-bit integer | no | - |
| personId | String | no | - |
| job | String | no | - |
| characters | Array[String] | no | - |

*N.d.R.*: *'category'* field used to fill empty *'job'* fields; then dropped.

### 1.1.3 Schema indexes and bulk data insert

As can be seen from the tables above, in addition to the standard unique indexes, it was decided to add additional ones to optimise the queries requested by the backend. Given the significant use of wildcard searches based on regular prefix expressions, it was decided to implement ascending indexes on several *'String'* type fields to drastically reduce search times without having to resort to a text index, which would take up much more space due to the tokenisation and stemming of the fields.

In order to perform a bulk insert of the approximately 42.5 million records contained in $\sim$ 1.6 GB of JSON files, it was decided to write a script in Python that would take advantage of a multithreaded execution of the `mongoimport` command-line tool provided by MongoDB. The overall script is responsible for generating the database's collections, populating them and finally creating indexes.

### 1.1.4 Technical details

The database was configured as a locally managed instance of MongoDB, thus avoiding the utilisation of the cloud version of the service (Atlas). The service's local deployment facilitated the monitoring and control of its resources, as well as the measurement of its performance. These factors collectively contributed to the generation of more reliable results during the load tests that were conducted.

Technical insights into the environment are presented below.

- MongoDB 7.0.11 (Community Server service)
- MongoDB Compass 1.43.3 (database UI)
- MongoDB Shell 2.2.9 (database CLI)

## 1.2 RESTful API – Express

The web application's backend was developed using the Express framework, a Node.js web application framework that was identified as the most widely utilised in the "State of JavaScript 2023" survey[3].

[3]https://2023.stateofjs.com/en-US/other-tools/#backend_frameworks

A variety of endpoints were incorporated into the API, enabling a constrained yet still functional utilisation of the web interface. The user-accessible API calls are enumerated below, along with a concise description of each.

1. `localhost:3000/search/preview/:title` – Returns the top 4 most voted titles that match the search query prefix.
2. `localhost:3000/search/:title` – Returns the most voted titles that match the search query prefix; the results are paginated in groups of 8.
3. `localhost:3000/search/episodes/:title` – Returns the episodes of a specific title that matches the *'titleId'* parameter.
4. `localhost:3000/title/:id` – Returns the details of a specific title that matches the *'_id'* parameter.
5. `localhost:3000/episode/:id` – Returns the details of a specific episode that matches the *'_id'* parameter.

The official MongoDB driver for Node.js (Typescript) `mongodb@6.8` is responsible for handling queries from the backend to the database. We deliberately decided not to utilise any form of Object-Relational/Document Mapping (e.g. `Mongoose` or `Prisma`) due to the suitability of the JSON record format to the application domain model and potential performance overheads.

Aggregation pipelines are used to facilitate the execution of queries to the database. These pipelines allow the definition of queries through the use of a sequential list of stages, thereby simplifying the grouping and sorting of data and providing control over the execution times of the individual stages within the pipeline.

## 1.3 Website – Vue and Tailwind CSS

The web interface of the system was realised using Vue.js "The Progressive JavaScript Framework", a well-known framework used for building SPAs (Single Page Applications) renowned for its component-based architecture and reactive data binding system. The aesthetic component was addressed through the utilisation of Tailwind CSS, a utility-first CSS framework that can be suitably integrated with Vue. A bundle of the website, optimised for deployment on a static hosting service, was created at the end of the development process using Vite[4]. This design choice ensures that the frontend is efficiently packaged into static files, which can be served directly to users without requiring server-side processing. Consequently, during load testing with JMeter, the focus has been exclusively on the backend API endpoints, as the frontend's static nature does not impose additional load on the server.

Thumbnails of the web interface are available on GitHub[5].

[4]A modern front-end build tool that facilitates optimized production builds through the utilization of ES hot module replacement (HMR).

[5]https://github.com/danieljaderpellattiero/unive-imdb/tree/frontend/thumbnails

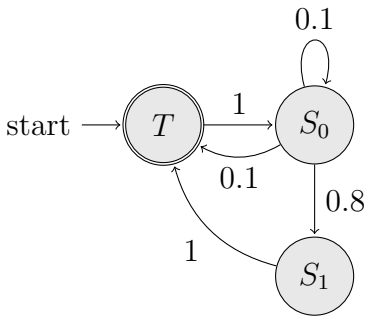
# Chapter 2

# Test case identification

## 2.1 High-level test case design

This brief chapter aims to present the test case designed to perform the load test of the system. In order to illustrate the user behaviour modelled by the test, we define a trivial finite-state automata with reference to the endpoints implemented in the RESTful API.

It should be noted that, although this aspect will be discussed in greater detail in the following chapters, the two endpoints that will be used in the load test result in two different classes of jobs, which must consequently be analysed individually.




The above automata has to be interpreted in the following manner:

- The user initiates a title search by name match, regardless of the customer language locale. (`/search/:title` – node $S_0$)
  - In 80% of cases, the search continues with a precise request for the title information. (`/title/:id` *or* `/episode/:id` – node $S_1$)
  - In 20% of cases, either the user terminates the search process pre-emptively, without finding the title, thus entering a thinking state – (node $T$) or it continues the search by hitting the same endpoint though the use of pagination. (`/search/:title?page=n` – node $S_0$)
- Upon receipt of the searched title information, the user is satisfied and returns to a thinking state before repeating a new search.

## 2.2 Computational settings

The following table provides an overview of the hardware specifications of the computers utilized in the load test.

Table 2.1: Hardware specifications

| Machine type | Role | O.S. | CPU | RAM |
|---|---|---|---|---|
| Desktop | Benchmark executor | Ubuntu 22.04 | Intel i5-13600K (20) | 64 GB |
| Laptop | System under test | Ubuntu 22.04 | Intel i5-8265U (8) | 8 GB |

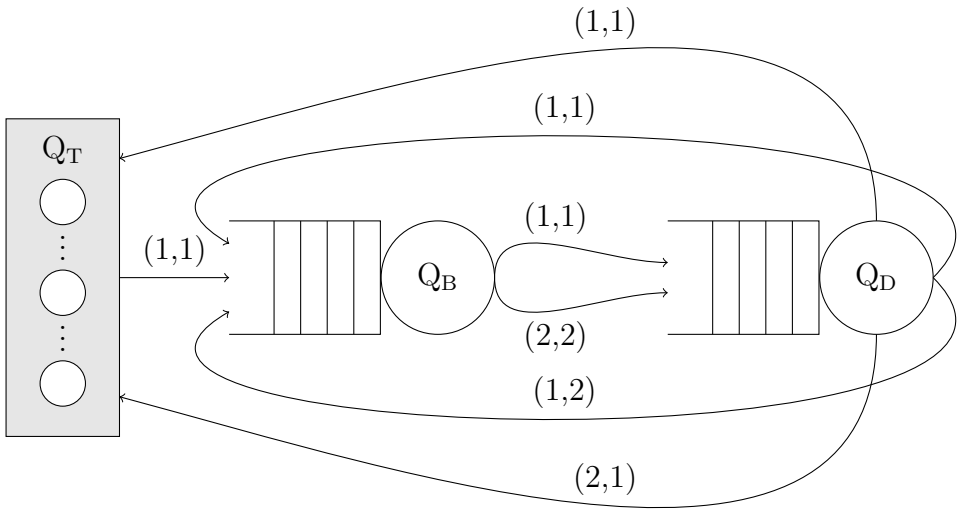
# Chapter 3

# Queueing network model

## 3.1 Introduction

This chapter presents a comprehensive theoretical analysis of the implemented model. The following illustration depicts the system components involved in the load test in the context of queuing theory.

For the purposes of this illustration, let $Q_T$ represent the thinking station, $Q_B$ represent the backend, and $Q_D$ represent the database.




It is possible to derive several characteristics of the system under consideration from the illustration above.

The system is characterised by a closed and interactive nature, free of stability conditions, and a constant number of users that dictate the nature of the workload. A further aspect that emerges relates to the multiclass nature of the jobs within the system, as indicated by the labels placed at the station entry and exit arcs.

Subsequently, an in-depth examination will be conducted on the distinct service time/demand and probabilistic routing for the two classes of jobs.

In conclusion, the network under analysis can be classified as a BCMP, exhibiting a closed topology and utilising "Processor Sharing" (PS) as its scheduling discipline.

Prior to examining the nature of the queues that characterise our network and their associated scheduling disciplines, we proceed with the formalisation and resolution of the network's traffic equations. These equations will then be essential in producing the theoretical analysis of the system.

## 3.2 Traffic equations

The system's traffic equations are presented below.

$$\begin{bmatrix} p_{T1B1} \\ p_{B1D1} \\ p_{B2D2} \\ p_{D1B1} \\ p_{D1B2} \\ p_{D1T1} \\ p_{D2T1} \end{bmatrix} = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 0.1 \\ 0.8 \\ 0.1 \\ 1.0 \end{pmatrix} \quad : \quad \begin{cases} e_{B1} = e_{T1} + (p_{D1B1} \times e_{D1}) \\ e_{B2} = p_{D1B2} \times e_{D1} \\ e_{D1} = e_{B1} \\ e_{D2} = e_{B2} \\ e_{T1} = e_{D2} + (p_{D1T1} \times e_{D1}) \end{cases} \Rightarrow \begin{cases} e_{B1} = 1.1111 \times e_{T1} \\ e_{B2} = 0.8889 \times e_{T1} \\ e_{D1} = 1.1111 \times e_{T1} \\ e_{D2} = 0.8889 \times e_{T1} \end{cases}$$

The system of equations above demonstrates that our routing is irreducible, which implies that any queue can be accessed from any other queue within the network. From an analytical standpoint, our system is under-determined, indicating the existence of an infinite number of solutions. The non-trivial solutions can be derived from the non-zero multiplicative coefficients.

Moreover, it is essential to highlight that within this context, where the system under consideration is an interactive queueing network, it is imperative to impose an additional constraint on the previous system, specifically $e_{T1} = 1$, given that this is the thinking station $\mathrm{Q_T}$.

## 3.3 Queueing network components

This section will provide a brief overview of the components of our network, with reference to Kendall's notation.

### 3.3.1 Thinking station

The thinking station can be represented by a queuing system described with the "G/D/$\infty$/IS" notation. The distribution of inter-arrival times for jobs at the thinking station is unknown, but the distribution of service times is known, as the jobs within it remain in service (*thinking state*) for a fixed constant time, known as thinking time. By definition, the thinking station is equipped with an infinite number of servers, indeed an undefined number of jobs can access it. Consequently, the service discipline of this system is of the "Delay center" or "Infinite Servers" type, as all the jobs remain in an idle state in parallel.

### 3.3.2 Backend station

With regard to the JavaScript runtime environment Node.js, despite its reputation as a "purely single-threaded" execution environment due to its Event Loop, it is in fact capable of offloading I/O blocking operations to separate threads or the operating system, thus

allowing the main thread to continue processing other tasks. The library that permits the dispatching of blocking operations to a thread pool is called `libuv` and it belongs to the Chrome V8 Engine. Furthermore, Node.js has also implemented the `worker_threads` module, which allows for further enhancements of performance by the utilisation of multi-core processors for CPU-bound operations.

In light of the aforementioned considerations, it can be concluded that the RESTful API functions as a "G/G/1/PS" queueing system. This is due to the fact that no manual instantiation of multiple worker threads was performed for the API, and thus a unique main thread is responsible for handling all incoming requests through a "Processor Sharing" scheduling discipline.

### 3.3.3 Database station

With regard to MongoDB, an examination of the "Production Notes" section reveals that its WiredTiger storage engine is inherently multithreaded. In particular, the total number of active threads (i.e. concurrent operations) relative to the number of available CPUs can impact performance as follows:

- The throughput increases in direct proportion to the number of concurrent active operations, up to the overall number of CPUs.
- The throughput decreases as the number of concurrent active operations exceeds the number of CPUs by a certain threshold amount.

The optimal number of concurrent active operations for a given application can be determined through the measurement of throughput.

In order to ascertain the exactness of the information provided in the documentation, an empirical test was conducted to monitor the thread pool instantiated by the MongoDB service in both idle and stressed states.

The results obtained from the Linux command-line utility `htop` are presented below.

In conclusion, the database can be identified as a "G/G/#C/PS" queueing system, where "#C" represents the number of cores. The inter-arrival times and service times are both unknown, however it can be stated that the database leverages all CPU cores with a "Processor Sharing" service discipline.

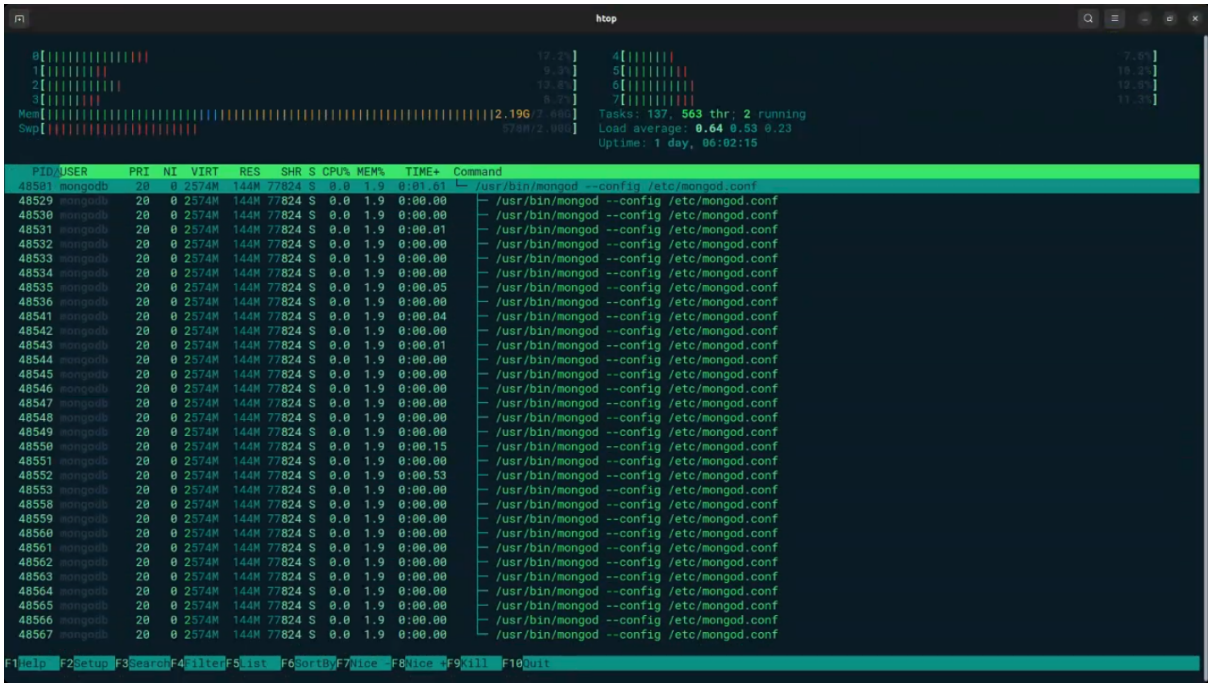



Figure 3.1: MongoDB thread pool in idle state

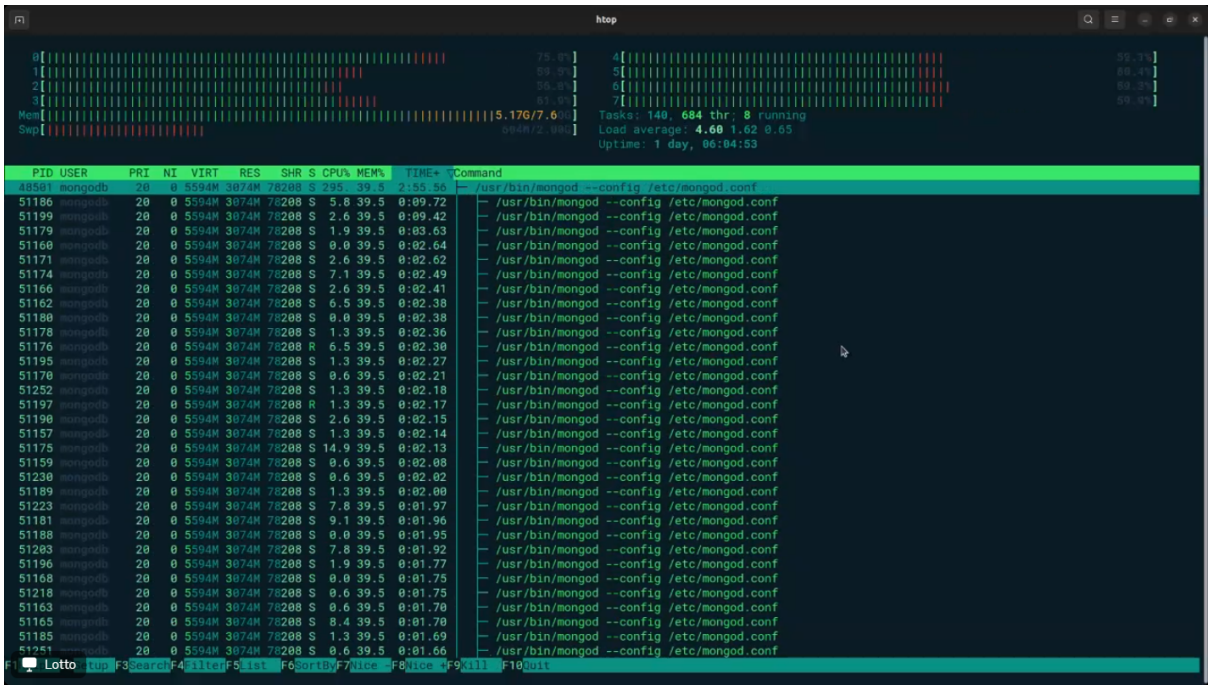



Figure 3.2: MongoDB thread pool under load conditions

# Chapter 4

# Q.N. Theoretical analysis

To fulfil the requirements of the assignment, it is necessary to undertake a comprehensive theoretical analysis of the queueing network.

In order to apply the theoretical notions learned during the course to the estimation of the degree of scalability of the system, the optimal number of users and the identification of the bottleneck of the system, it is first necessary to measure the service times for the queueing network components with regard to the two classes of jobs previously defined.

Although the queries to the endpoint `'/search/:title?page=n'` may vary depending on the parameterization of the URL, we assume that they belong to the same job class. This assumption will be subsequently confirmed by the empirical results.

## 4.1 Service times measurements

### 4.1.1 Overview

Given that we are operating in a "Processor Sharing" regime, it is possible for us to exploit the fact that, in the event that there is only one job within the system, its service time is in fact equivalent to the response time. In light of these considerations, we employed JMeter, a load testing software that will be discussed in greater detail later on, to perform two tests (one for each class of jobs) with the objective of measuring the various service times of our stations.

Both tests comprise a user (a thread) executing a request to the RESTful API at a constant interval of 1 second. In both cases, the user makes a total of 100 queries, a number determined by the size of the query set imposed by the task, which allows a good normal approximation.

Details on the tests carried out are provided in the following sections.

### 4.1.2 Test query set

A Python script was employed to perform a random sampling with reinsertion of 10,000 movie titles based on our MongoDB collections.

It was assumed that the probability of a movie being selected in the extraction is proportional to the number of votes it received. In order to obtain the most comprehensive query set possible, it was decided to include all movies, all episodes, and all the various linguistic declinations of these in the extraction. The entire process was carried out using the `pandas.DataFrame.sample` function.

### 4.1.3 RESTful API Profiling

To obtain the most precise measurement of service times for the two classes of jobs, two different versions of the backend were implemented. One was employed for load testing and the other was used for the current employment; the latter version contained mechanisms to perform profiling on its execution.

Indeed, within the API, it was necessary to distinguish between the processing times of code in Node.js and those of database aggregation pipelines. To perform profiling on the Express execution, it was decided to utilise the `PerformanceObserver` Node interface, which forms part of the Performance Timing API. Indeed, this interface enabled the effective measurement of execution times without the addition of overheads through the utilisation of the `Marks`, namely high-resolution timestamps.

Conversely, regarding MongoDB profiling, the situation was more intricate. In order to facilitate the process, we proceeded to activate the `system.profile` collection within our database. This is a *capped collection*[1] that is created by default by the MongoDB Profiler for each database and hidden from the user. Its purpose is to measure precise metrics regarding CRUD operations and other administrative and/or configuration commands executed on a running instance of MongoDB. Consequently, the utilisation of the profiler enabled the accurate determination of the execution times inherent to the aggregation pipelines.

Upon completion of the whole process, all metrics evaluated by the API execution are transcribed asynchronously, thus maintaining optimal performance, by `Pino`.

`Pino.js` is a fast and lightweight Node.js logging library: it is a robust and scalable solution due to its comprehensive asynchronous logging of JSON-formatted files that provides customizable log levels and serializers while maintaining low processing costs.

### 4.1.4 Empirical service times

Following the execution of the two tests with JMeter, the service times of our system components with respect to the job classes identified in our network are as follows:

$$\mu_{B1}^{-1} = 0.001\,69\,\text{s} \qquad \mu_{D1}^{-1} = 0.002\,58\,\text{s} \qquad \mu_{B2}^{-1} = 0.001\,71\,\text{s} \qquad \mu_{D2}^{-1} = 0.001\,46\,\text{s}$$

It should be noted that, although MongoDB employs a scheduling discipline of the "Processor Sharing" type, it is not a single-server system analogous to the M/G/1/PS queues analysed during the course. However, since we must assume that we are dealing with a single server station, in order to be able to compute, for instance, the asymptotic bounds for the throughput and the expected response time (part of the theoretical analysis carried out in the following chapters), it is necessary to treat the database as if it were a single server, and therefore to calculate its service time in the following way:

$$\mu_{Dx}^{-1} = \frac{\overline{R}}{\text{number of CPUs}} \quad \text{where} \quad x \in \{1, 2\}$$

## 4.2 Bottleneck identification

Thus far, we have calculated the service times of each station per job class and the relative visit ratios ($\overline{V}_i$). These $\overline{V}_i$ are derived from the solutions of the traffic equations

[1] A fixed-size collection that performs document insertion and retrieval based on the order in which they were added, exhibiting behaviour analogous to that of circular buffers.

and indicate the expected number of visits to each station with respect to each visit made to the reference station ($Q_T$).

$$\overline{V}_{B1} = 1.111\,11 \quad \overline{V}_{D1} = 1.111\,11 \quad \overline{V}_{B2} = 0.888\,89 \quad \overline{V}_{D2} = 0.888\,89 \quad \overline{V}_{T} = 1.000\,00$$

The data enables the calculation of the service demands ($\overline{D}_i$), which represent the quantity of service requested by a user from each station for each visit to the reference station. This is achieved through the application of the following formula:

$$\overline{D}_i = \overline{V}_i \times \mu_i^{-1} \quad \text{where} \quad i \in \{B1, D1, B2, D2\} \tag{4.1}$$

$$\overline{D}_{B1} = 0.001\,88 \quad \overline{D}_{D1} = 0.002\,86 \quad \overline{D}_{B2} = 0.001\,52 \quad \overline{D}_{D2} = 0.001\,30$$

The results indicate that the database is the bottleneck station. A further calculation of the utilisation of each station can be made using the Bottleneck Law:

$$\rho_i = X_1 \times \overline{D}_i \quad \text{where} \quad i \in \{B1, D1, B2, D2\} \tag{4.2}$$

A graph is provided below, which illustrates the varying levels of station utilisation over the number of users.

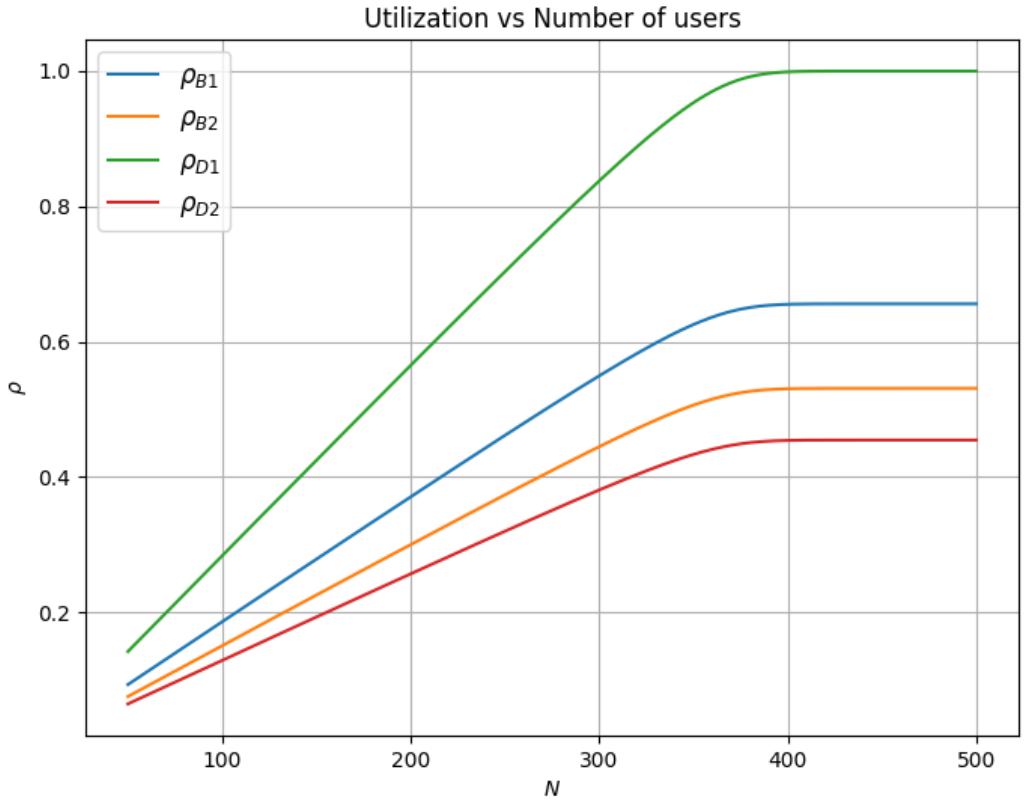



Figure 4.1: Stations utilisation over the number of users

It can be concluded that the database is the component with the highest utilisation rate.

## 4.3 Asymptotic bounds

In the next section, we will perform a mean value analysis to compute the expected performance indices of our system. This will be done through Little's Law and the recursive procedure on which this technique is based, without the need to compute the normalization constant $G_K$ required by Gordon Newell's Theorem.

Prior to conducting the previously announced analysis, a preliminary calculation is conducted using a Python script to determine the upper and lower bounds of response time ($\overline{R}$) and throughput ($X$) in relation to the number of customers in the network.

This is done according to the following formulas:

$$\overline{R} \geq max\left(\overline{D}, N\overline{D}_b - \overline{Z}\right) \tag{4.3}$$

$$X \leq min\left(\frac{N}{\overline{D}+\overline{Z}}, \frac{1}{\overline{D}_b}\right) \quad \text{where} \tag{4.4}$$

$$\overline{Z} \quad \text{exp. thinking time,} \quad \overline{D}_b \quad \text{bottleneck service demand,} \quad \overline{D} = \sum_{j=2}^{K} \overline{D}_j$$

The parametric values of the calculated limits are provided below:

$$R \geq max\left(0.007\,56,\ N \times 0.002\,86 - 1\right)$$

$$X \leq min\left(\frac{N}{0.007\,56 + 1},\ \frac{1}{0.002\,86}\right)$$

## 4.4 Mean Value Analysis

### 4.4.1 JMVA emulation setup

In order to estimate the performance indices of our queueing network, we employed the "Java Modelling Tools (JMT)" suite of applications developed by the "Politecnico di Milano". In particular, we utilised the JMVA ("Mean Value Analysis and Approximate solution algorithms for queueing network models") application.

The subsequent steps are those required to prepare the emulation within the software according to our queueing network configuration.

1. The number of customers (50) and the number of classes of jobs were entered. As can be seen, only one class was indicated, as only class 1 jobs were output from the thinking station.

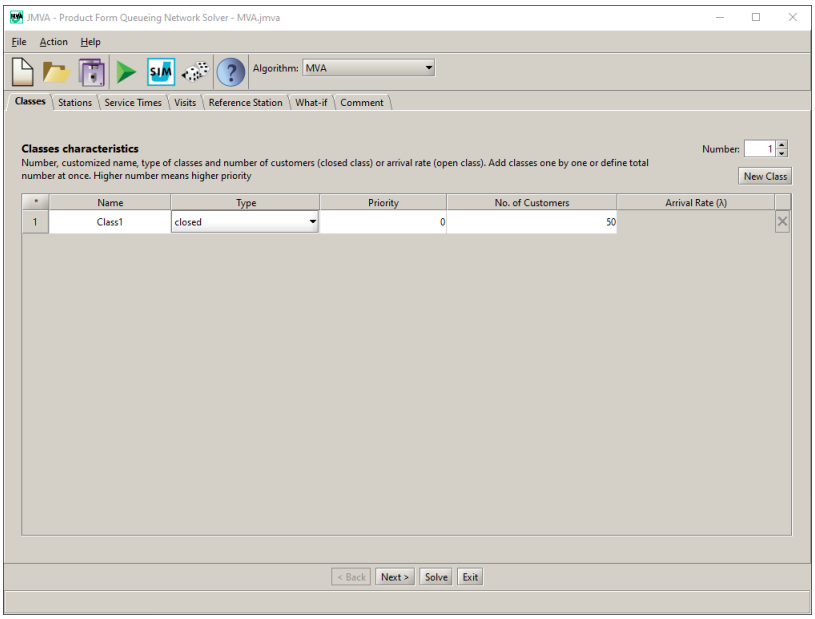



Figure 4.2: JMVA setup – classes of jobs

2. The stations in the queueing network are entered according to their respective load type. The thinking station is designated as a "delay type", whereas other stations are classified as "load-independent", as their service time remains consistent regardless of the number of customers.

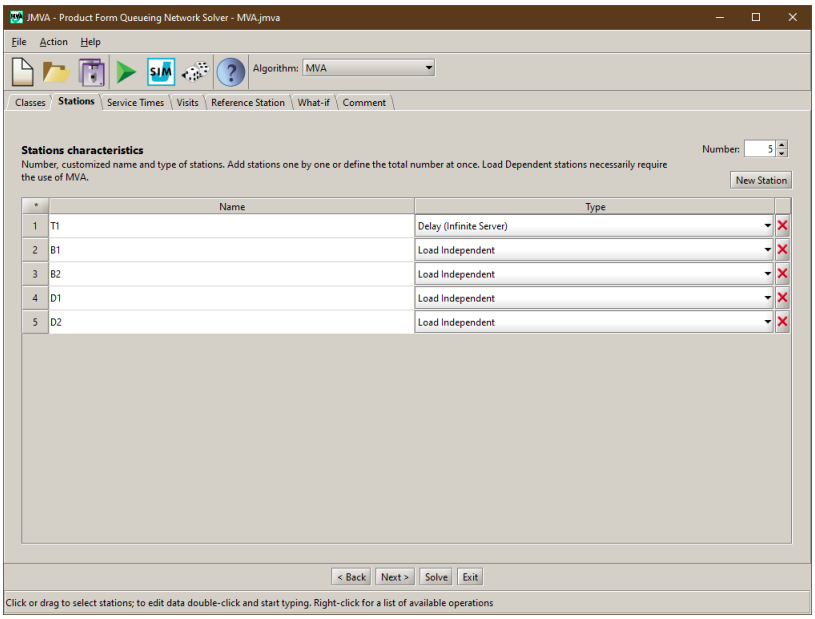



Figure 4.3: JMVA setup – stations

3. The service times previously calculated for each station are entered.

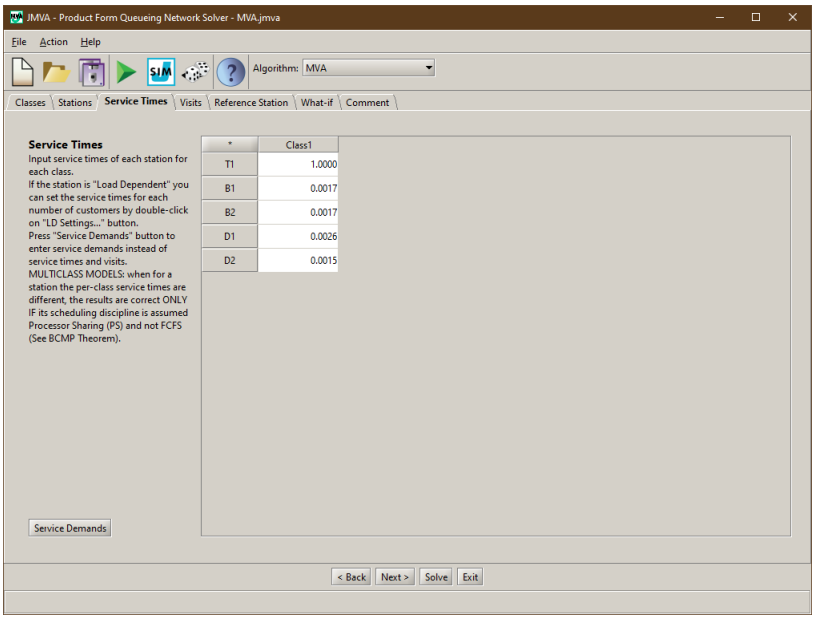



Figure 4.4: JMVA setup – service times

4. The relative visit ratios previously calculated for each station are entered.

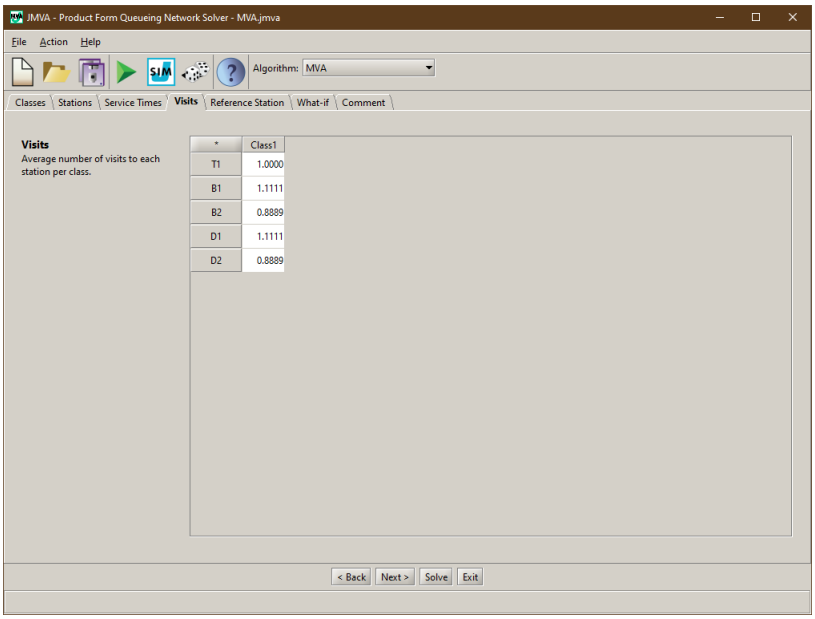



Figure 4.5: JMVA setup – relative visit ratios

5. The reference station of the queueing network is indicated.

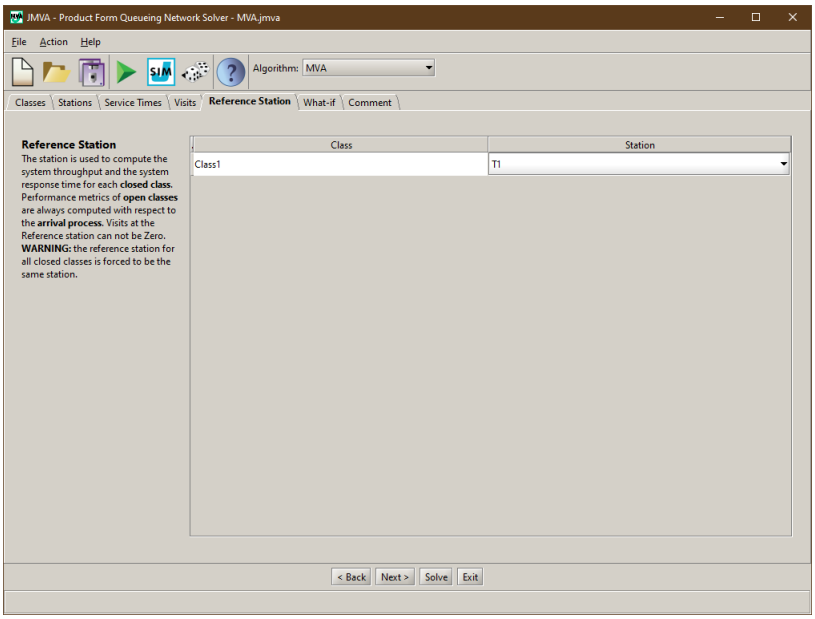



Figure 4.6: JMVA setup – reference station

6. The initial number of users is indicated as 50, with a subsequent increment of up to 500 users per increment of 1 user.

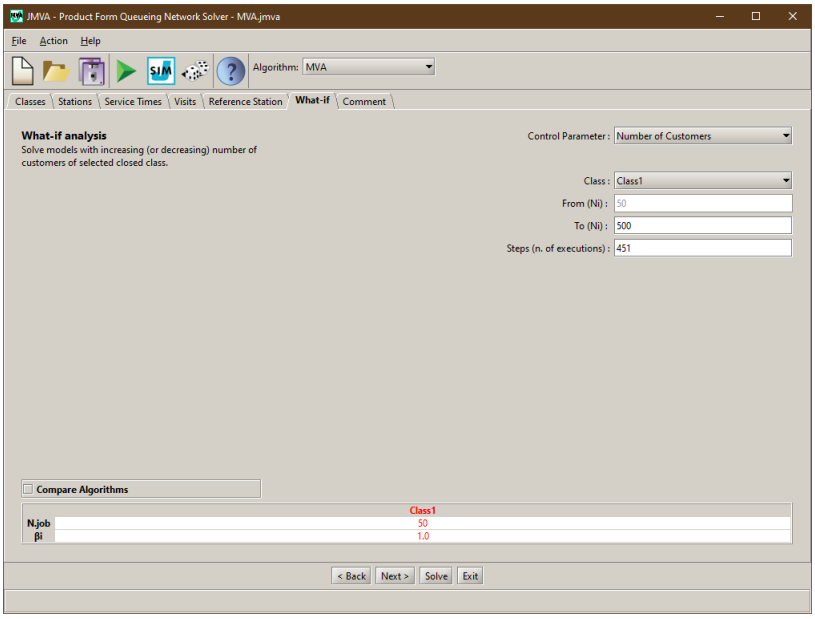



Figure 4.7: JMVA setup – emulation settings

### 4.4.2 JMVA emulation results

The graphs generated by the emulation of JMVA, in conjunction with the previously calculated theoretical bounds, are presented below for the reader's convenience. As can be observed in the graphs, the optimal number of users has already been determined; this will be calculated in the following section.

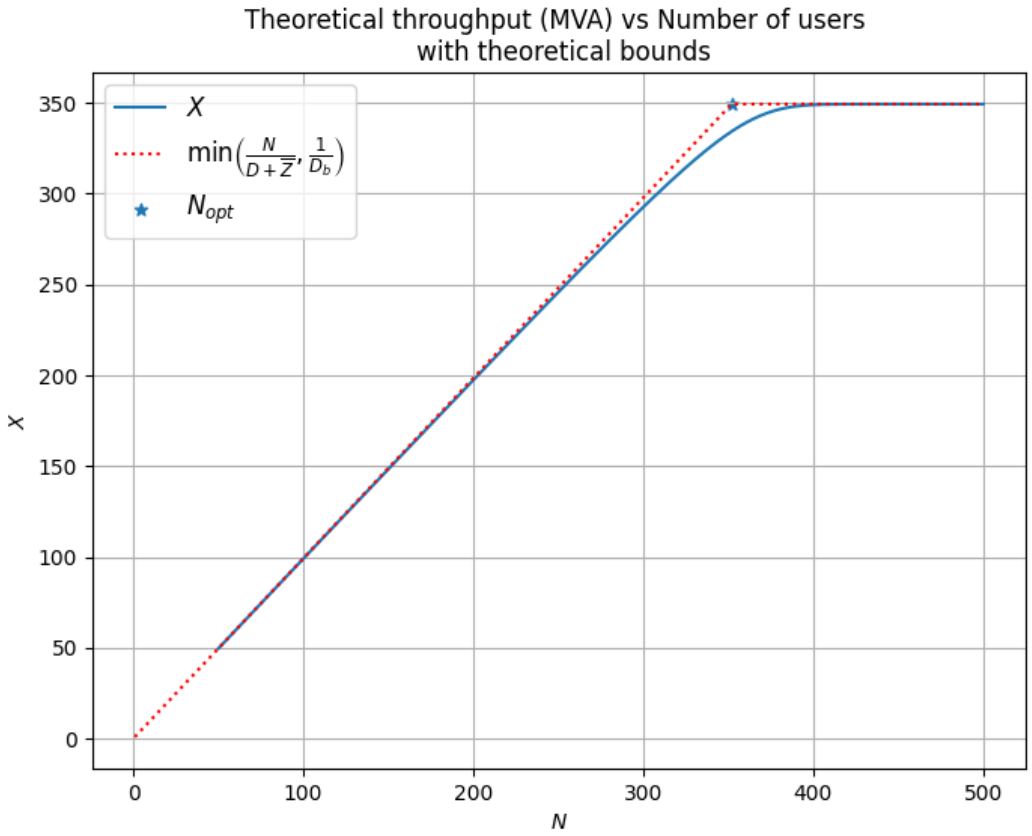



Figure 4.8: MVA – Throughput over the number of users

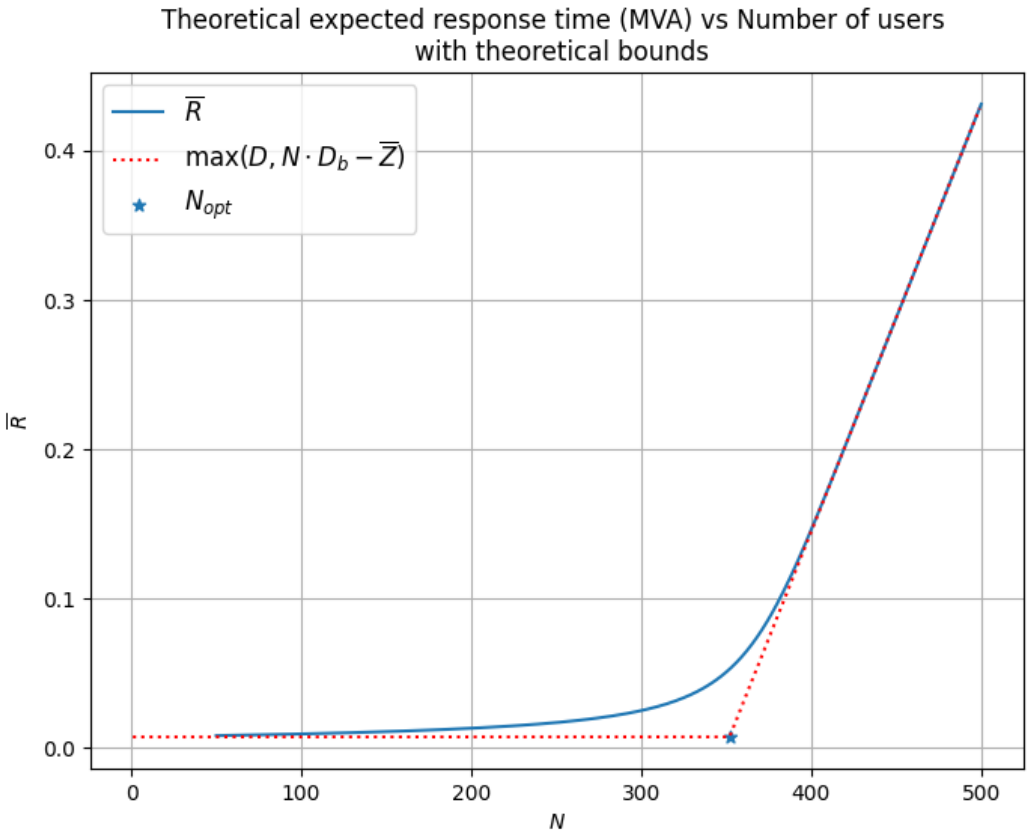



Figure 4.9: MVA – Response time over the number of users

## 4.5 Optimal – theoretical – number of users

In conclusion to the theoretical analysis of the queuing network, we proceed with the estimation of the optimal number of users. This can be obtained from the intersection of the previously calculated asymptotes for throughput and expected response time and with the following formula:

$$N_{opt} = \frac{\overline{D} + \overline{Z}}{\overline{D}_b} = 351.986\,51 \tag{4.5}$$

The outcome is fully reflected in the graphs illustrated above.

# Chapter 5

# Q.N. Empirical analysis

This chapter will address the core experimental aspect of the assignment: the configuration of the load test and the procedure by which it is launched. Subsequently, the final experimental outcomes will be presented.

## 5.1 JMeter – load test configuration

In the "User Defined Variables" module, the address of the computer running the RESTful API was entered. All other components of the load test are contained in a "Transaction controller", which allows the system's throughput and expected response time to be measured, rather than the user's.

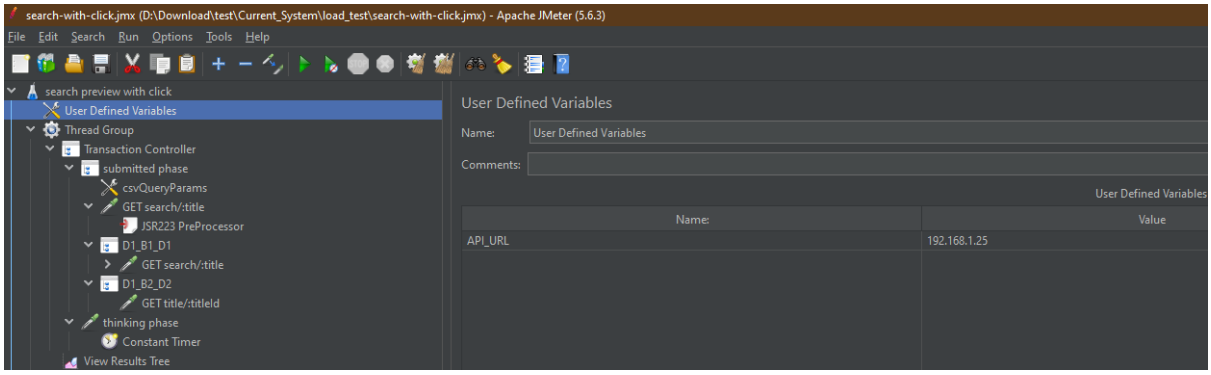



Figure 5.1: JMT – User Defined Variables

In the “Thread Group” module, a population of 500 users was specified (this value will be updated automatically). The *ramp-up* period is 0, which means that when the test is initiated, all threads are simultaneously spawned in the thinking station. The *loop count* is infinite, which means that requests to the API will continue indefinitely, based on the query set imported via CSV. The thread lifetime (*duration*) is 10 minutes, after which it will eventually die.

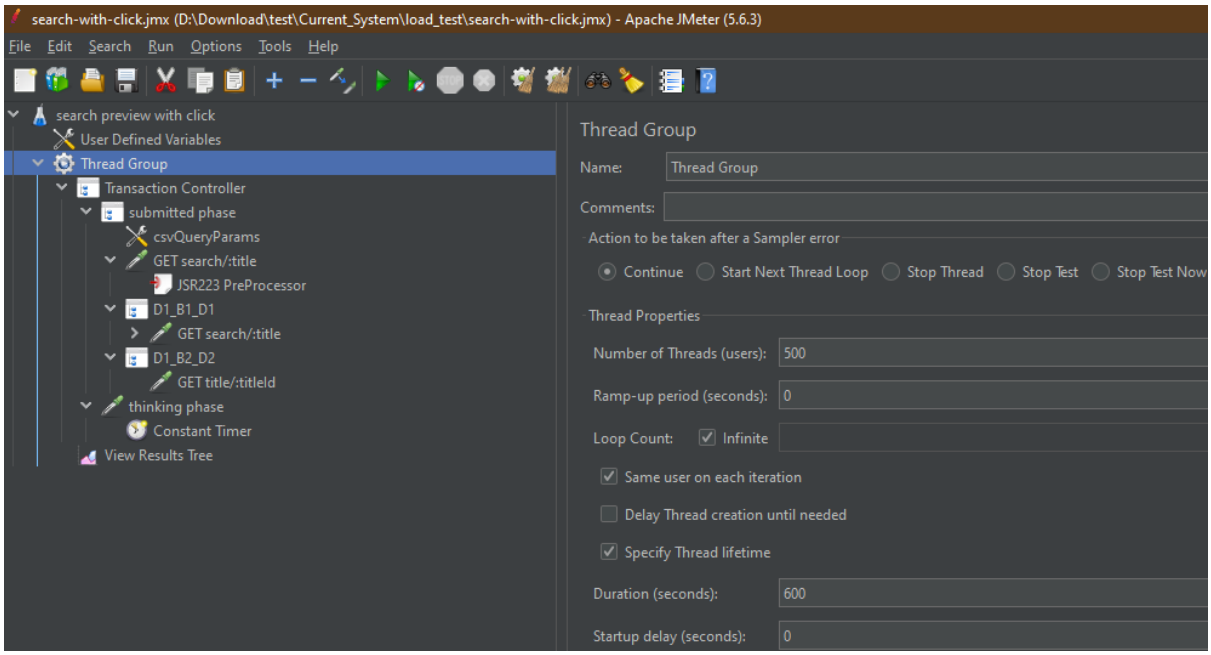



Figure 5.2: JMT – Thread Group

In the “csvQueryParams” module, the location of the CSV file containing the query set of 10,000 films was specified. The file will be read cyclically by each thread and will also be shared among them, which will therefore start reading it from different rows.

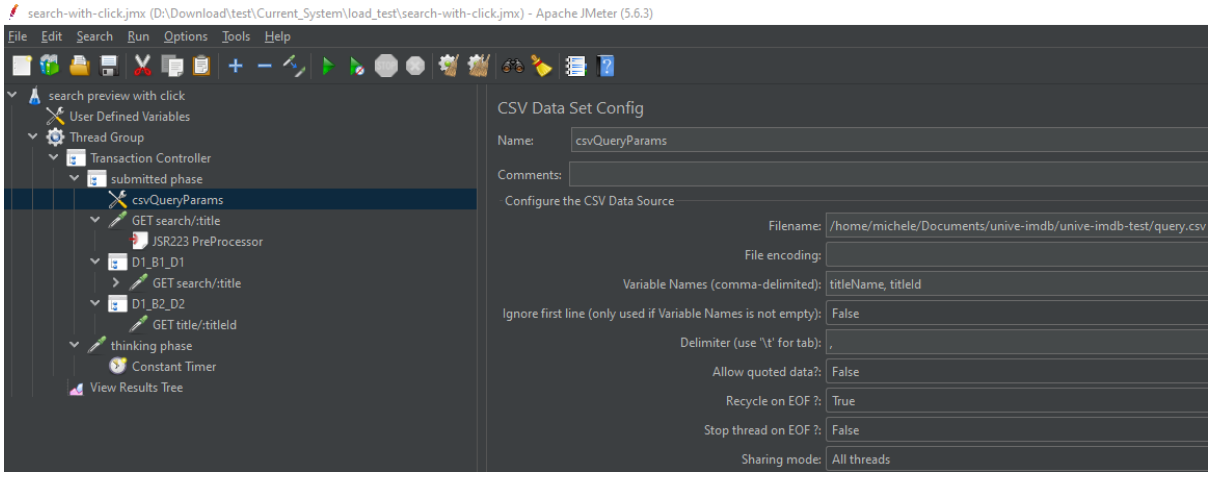



Figure 5.3: JMT – CSV Query Params

In this “JSR223 PreProcessor” module, the page number, which will subsequently be used to parameterize the API request “`/search/:title?page=n`”, is initialized to 1. Additionally, a variable responsible for probabilistic routing, which will take values between 1 and 10 inclusive, is also initialized.

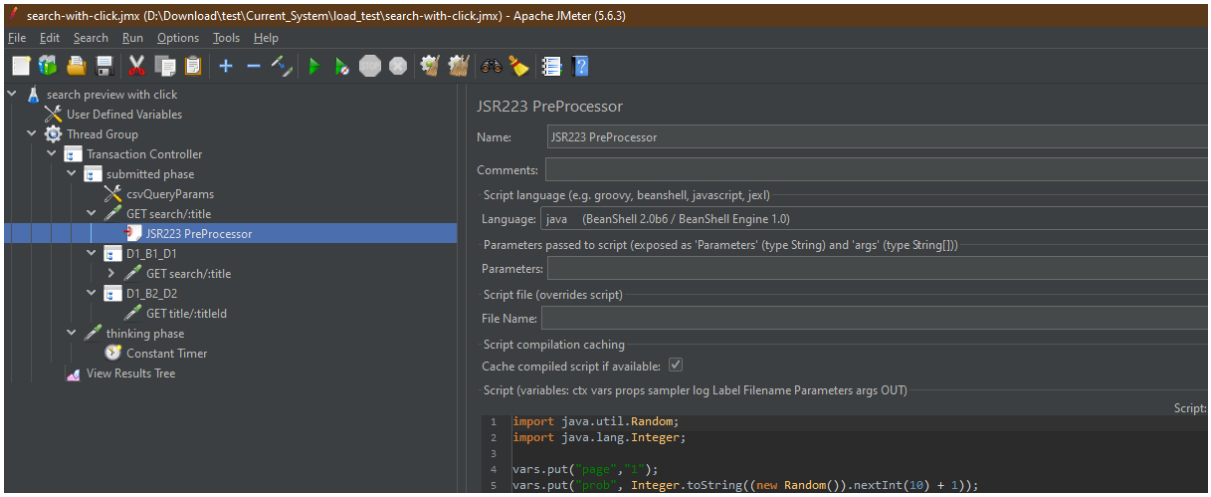



Figure 5.4: JMT – JSR223 PreProcessor (1)

In this “HTTP Request” module, a request is submitted to the “`/search/:title?page=1`” endpoint.

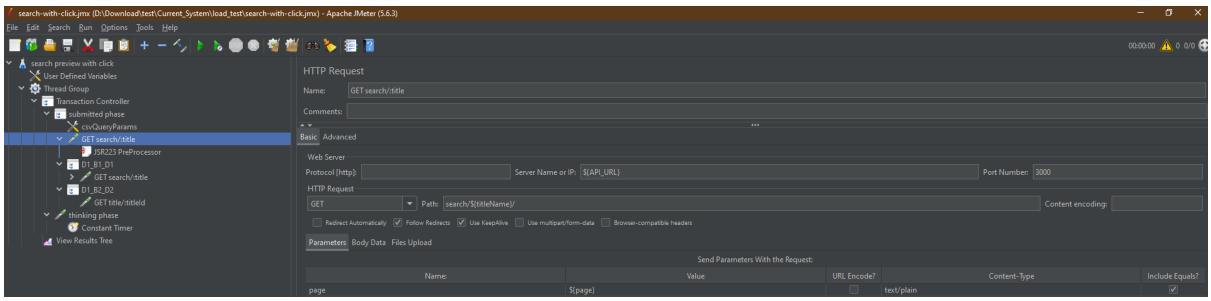



Figure 5.5: JMT – HTTP Request (1)

In the “While Controller” module, we test whether the user is within the 10% probability of repeating the title search with an updated pagination value.

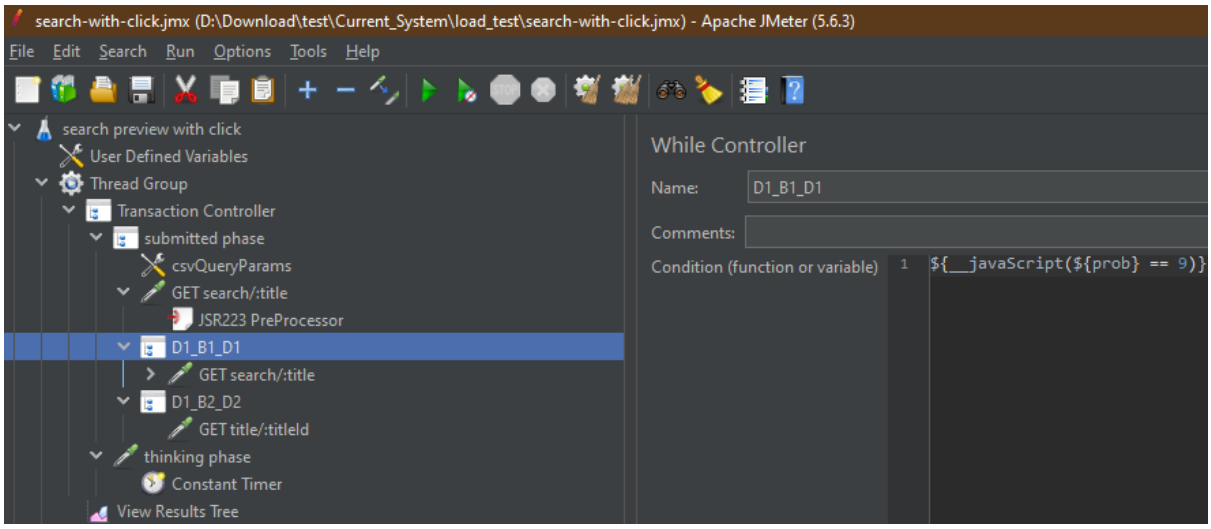



Figure 5.6: JMT – While Controller

In this “JSR223 PreProcessor” module, the page number is incremented in accordance with the occurrence of the event in which the search is reiterated.

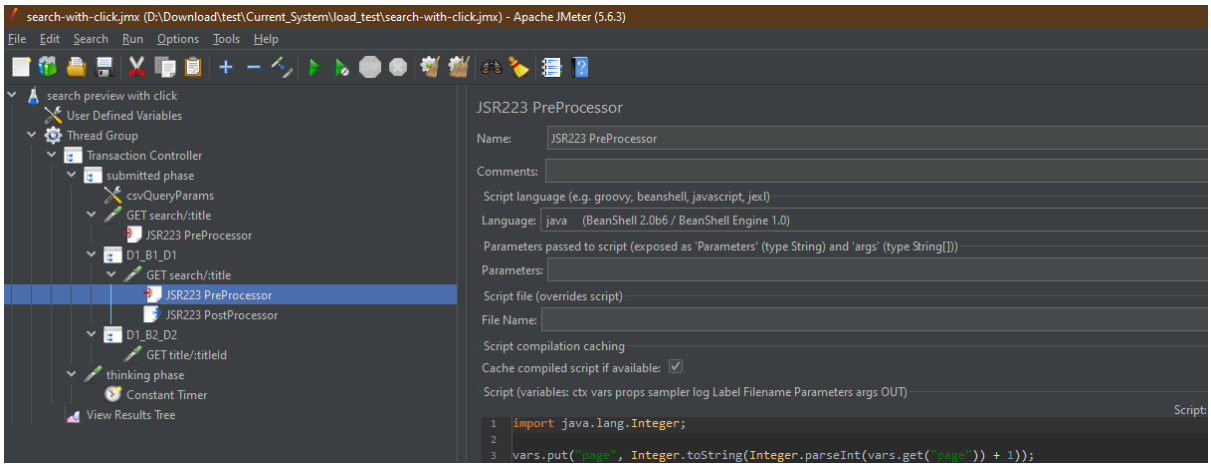



Figure 5.7: JMT – JSR223 PreProcessor (2)

In this “HTTP Request” module, a request is submitted to the “`/search/:title?page=n`” endpoint.

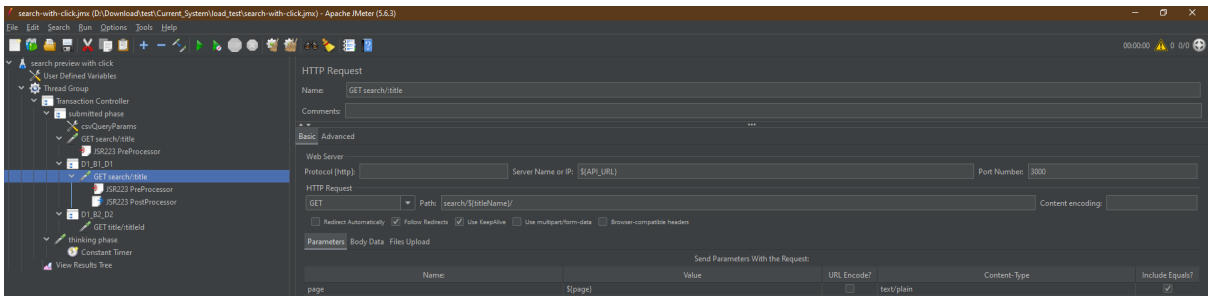


Figure 5.8: JMT – HTTP Request (2)

In the “JSR223 PostProcessor” module, the value is reassigned to the variable responsible for probabilistic routing.

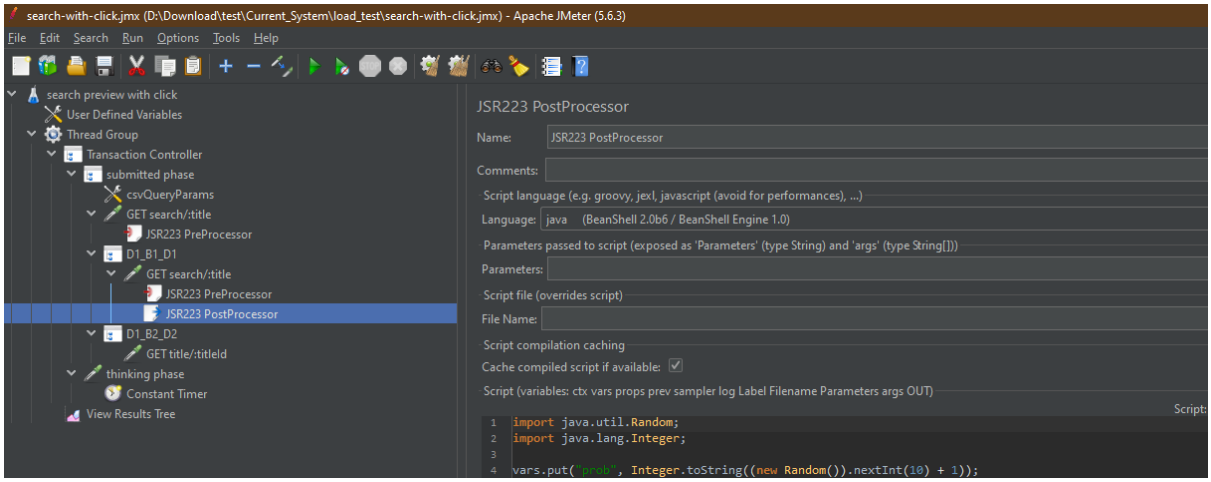



Figure 5.9: JMT – JSR223 PostProcessor

In the “If Controller” module, we assess whether the user is within the 80% probability of performing a precise search to retrieve the title or episode information.

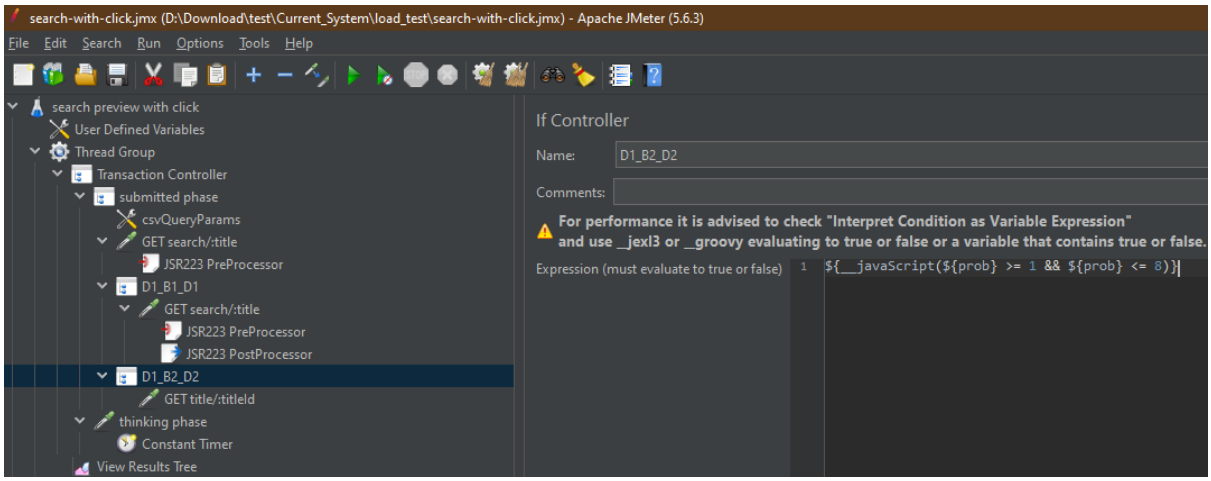



Figure 5.10: JMT – If Controller

In this “HTTP Request” module, a request is submitted to the “`/title/:id` *or* `/episode/:id`” endpoint.

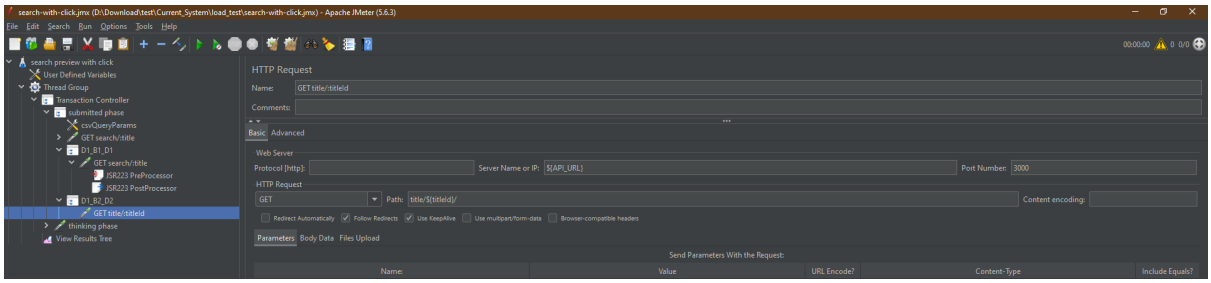



Figure 5.11: JMT – HTTP Request (3)

In the “Constant Timer” module, the thinking time of 1 second is defined as the period of time that a thread will wait at the conclusion of one of its cycles before initiating a new one.

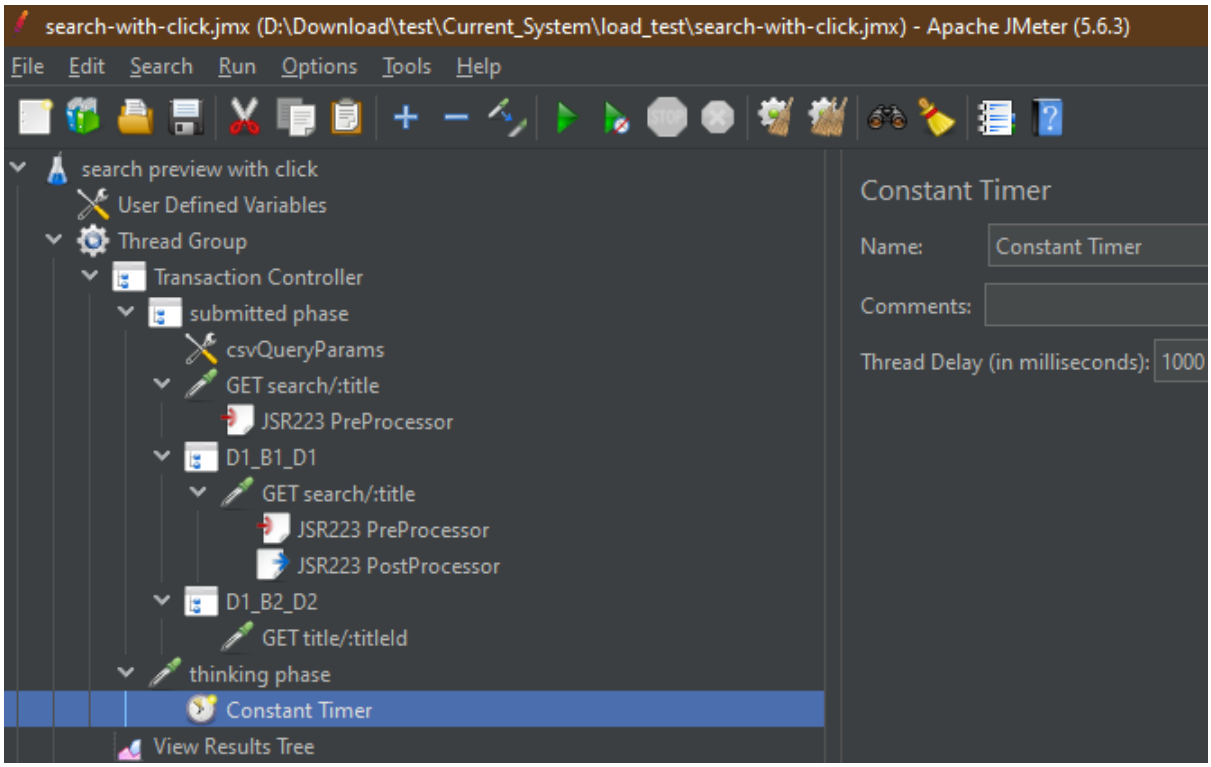



Figure 5.12: JMT – Constant Timer

## 5.2 Load test execution

### 5.2.1 Express and MongoDB configuration

Prior to executing the load test, it is of crucial importance to ensure that the RESTful API and the database have been properly configured.

With regard to Express, it is necessary to compile and subsequently run the testing version of the script, which is relieved of profiling mechanisms and is also controlled by `nodemon`[1], a Node.js npm package that among its various features automatically restarts the service when an error occurs.

With regard to MongoDB, it is essential to ensure that the Query Plan Cache is free of any residual data. This can be achieved by executing the command `db.collection.getPlanCache().clear()` within `mongosh`. Furthermore, it is necessary to disable the MongoDB Profiler by executing `db.setProfilingLevel(0, { <options> })`. Finally, if desired, the Profiler collection can be cleared with `db.system.profile.drop()`.

It should be noted that the testing version of the RESTful API has been equipped with a mechanism to automatically clear the database cache after each query. This is to prevent the distortion of test results due to caching mechanisms, which could otherwise occur if identical sequential queries were to be executed.

[1] https://nodemon.io/

### 5.2.2 Automated load test startup

To ensure an absolute minimum of potential error in the load testing process, it was deemed appropriate to employ a Python utility for this specific purpose. The implemented source code is responsible for importing the load test configuration file, which was exported from JMeter in the jmx format, using the `beautifulsoup`[2] library.

Subsequently, the actual load test is initiated through the `subprocess` library, which launches 10 sequential tests with a varying number of customers (threads), ranging from 50 to 500.

```
import subprocess
from tqdm import tqdm
from bs4 import BeautifulSoup

def test(name:str):
 jmx_name=f'{name}.jmx'

for n_users in tqdm(range(50, 550, 50)):
 with open(jmx_name) as f:
  Bs_data = BeautifulSoup(f.read(), "xml")
  tag=Bs_data.find('intProp', {"name":"ThreadGroup.num_threads"})
  tag.string = str(n_users)

with open(jmx_name, 'w') as f:
 f.write(str(Bs_data))

results_path = f'results_{n_users}'

subprocess.Popen(f'jmeter -n -t {jmx_name} -l
   {results_path+"/results.csv"} -e -o {results_path}', shell=True,
   stdout=subprocess.DEVNULL).wait()

if __name__ == "__main__":
 test('filename')
```

Listing 5.1: Python script for automated load test startup

## 5.3 Load test empirical results

The graphs of throughput and expected response time, calculated from the load test performed with JMeter, are presented below. The graphs were generated following a parsing step of the statistics exported in JSON format from the benchmark software.

[2]https://www.crummy.com/software/BeautifulSoup/

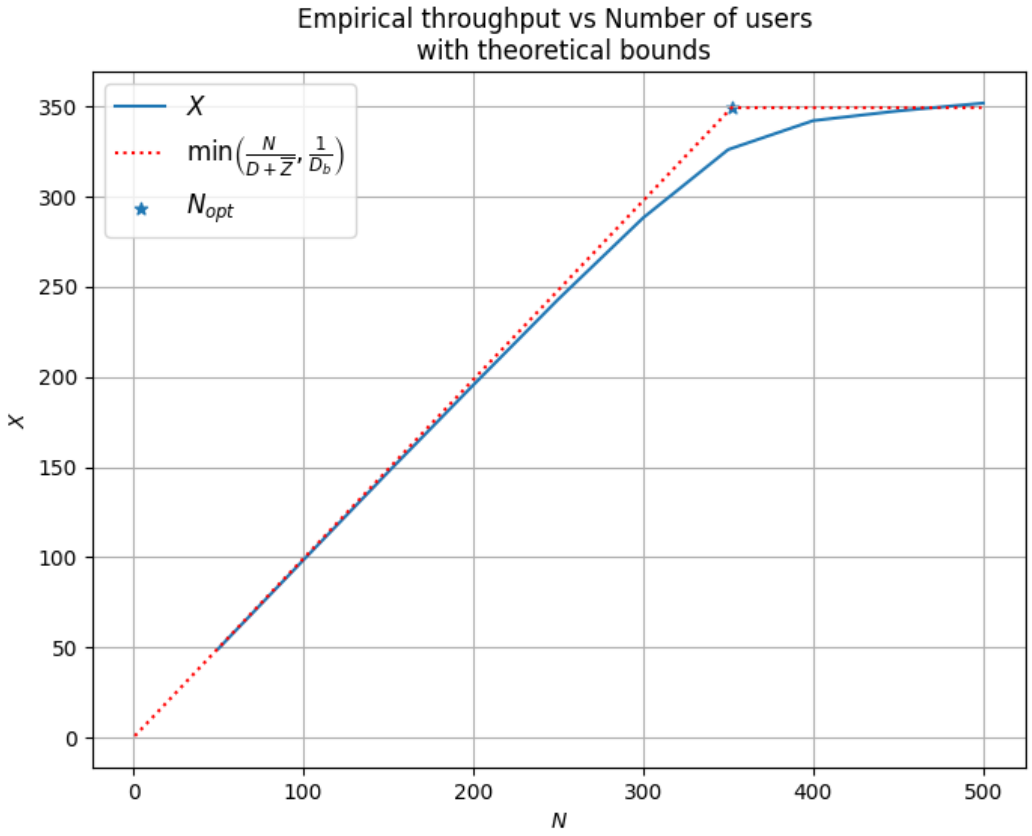



Figure 5.13: JMeter Load test – Throughput

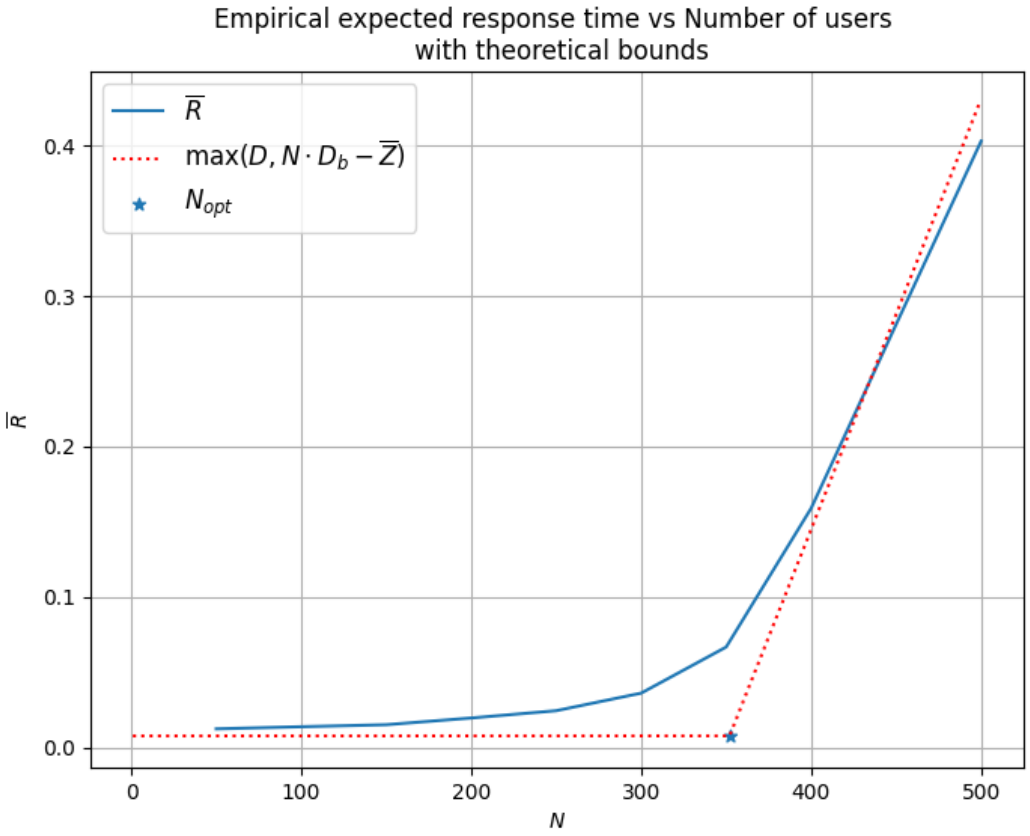



Figure 5.14: JMeter Load test – Expected Response Time

The graphs demonstrate a satisfactory fit with respect to the analytically calculated asymptotes. Additionally, they illustrate the precision of the estimated optimal number of users that the system can handle, which was previously determined through analysis.