DOCUMENTATION

ASSIGNMENT *2*

STUDENT NAME: EMINOVICI ANDREI-CORNELIU

GROUP: 30423

# CONTENTS

[1. Assignment Objective 3](#_Toc128043139)

[2. Problem Analysis, Modeling, Scenarios, Use Cases 3](#_Toc128043140)

[3. Design 3](#_Toc128043141)

[4. Implementation 3](#_Toc128043142)

[5. Results 3](#_Toc128043143)

[6. Conclusions 3](#_Toc128043144)

[7. Bibliography 3](#_Toc128043145)

# Assignment Objective

The main objective of this work is to propose, design and implement a queue management app simulation. More specifically, an app which can read input from the user using an interface, process it and launch multiple threads representing each a queue all managed by a manager which is another thread which takes decisions to optimize load distribution across all available servers(queues).

In real use scenarios, servers are expensive to create and maintain, so it is desired to make the most out of as little as possible that there is available. A queue management app will take a number of clients and process them using parallelized threads, thus greatly improving the waiting time for each individual client.

The secondary objectives are:

* Use of basic principles of Object Oriented Programming, namely abstractization, encapsulation, inheritance and polymorphism
* Use of the “ArrayBlockingQueue” Collection for storing the clients waiting to be served by a server
* Use of Atomic data types for synchronization
* Use of locks with “synchronized”, wait() and notify() calls for inter-thread communication
* Creation of a user interface using Java Swing

# Problem Analysis, Modeling, Scenarios, Use Cases

A queue is modelled by a server object which will serve clients one by one. The scheduler will assign clients to each queue until there are no more left to assign. The user input must be integer values for the required parameters of the simulation. As the assignment requirements ask, this is synchronized via use of BlockingQueue collection and locks.

The input data will be used to generate random clients and assign them to each running server available. The simulation results are displayed both in the logs text file and in the simulation window frame.

# DesignDiagram Description automatically generated

*Diagram

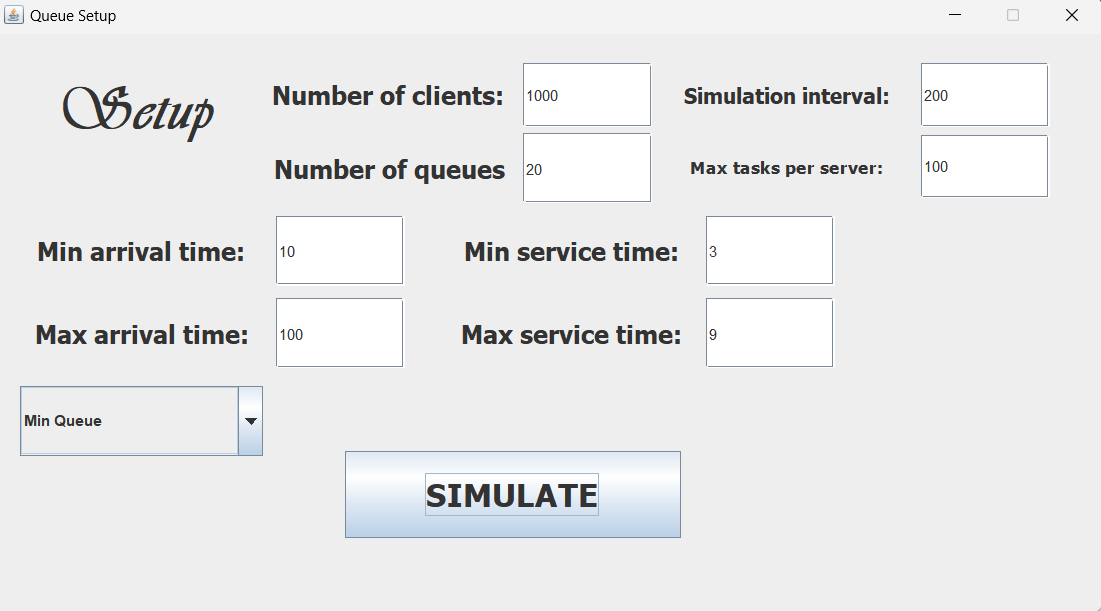
Description automatically generated*

# Implementation

A queue is represented by a server object using the BlockingQueue collection. This is necessary as there will be multiple accesses from multiple other threads and there needs to be synchronization in order to avoid data corruption and incorrect results. Each server is launched by a scheduler, an object which decides based on a selection strategy such as minimum length or minimum waiting time where to distribute each client to. This scheduler is launched by a simulation manager, the main object which stores all of the simulation parameters read from the user interface.

After the input data was successfully read, the simulation manager will generate a list of random clients represented by the tuple(ID, arrivalTime, serviceTime) according to the specified parameters by the user. Immediately after, the scheduler is launched on a parallel thread which will create all of the servers, initialize and start their threads as well. After that is done, the servers will receive the clients via the distributeClient() method implemented in the scheduler, which is further going to call another method addClient() from the Strategy type object which will then finally add the client into a server. Note that Strategy is an interface and it must be initialized as either shortestQueue or shortestTime because those classes are where the actual implementation of client distribution is made.

After this is done, the results will be updated and displayed in real time in the user interface “Simulation” frame, as well as in the standard console and written into a output\_logs.txt file. The simulation manager has an internal clock generating 1 second pulses and transmitting them to each running server via thread.notifyAll() in order to enable such execution.



Graphical user interface, text, application

Description automatically generated

# Results

The results of the asked simulation cases were uploaded along with the rest of the project files on the GitHub repository of this project. The values make sense, for example lowering the number of available servers will have a noticeable effect of increasing the average waiting time for a client.

# Conclusions

Regarding what I have learned, there is a lot to say. Firstly, the ability to take some code and execute it in parallel is exceptionally useful, especially when that code consists of lengthy or even infinite loops which would otherwise hault the entire execution, making any lines after the said loop unreachable or delayed.

Also, being able to speed up the average waiting time for a client via the use of multiple servers and queue distribution is incredibly efficient compared to what I knew up until this point. Substitute the sleep() calls for actual work a program needs to do and you have successfully parallelized a computationally intensive problem and now “attack” it with multiple processing cores as opposed to a single one, harvesting more power from the available hardware of your machine.

# Bibliography

The references for the project were Google and sites such as GeeksForGeeks, Baeldung and StackOverflow for help with many issues that I encountered along the way.