DOCUMENTATION

Assignment 2

Jucan Rares Tudor
30422 – 1

Table of contents

1. Assignment objective 3
2. Problem analysis, modelling and use cases 4
3. Design 6
4. Implementation 7
5. Results 13
6. Conclusions 14
7. Bibliography 15

1. Assignment Objective

Design and implement a simulation application aiming to analyze queuing-based systems for determining and minimizing clients' waiting time.

Queues are used in many places, including in real-world problems and in software related ones also. The purpose of a queue is to provide place for a task/client while it is waiting for a given service, to be processed. This waiting time can be minimized if more queues are added, so we can part the clients to multiple servers where they can wait.

The software simulates in real time, the processing of clients, which are later put in queues. It always takes the clients according to their arriving time. We do not know at simulation time N, that who will arrive at simulation time N + 1. Each client should be put in a queue, which finishes in the fastest time possible compared to other queues. By doing this, we aim to achieve the best average waiting/ processing time in the end.

For simulating the queues, the application must manage to run them at the same time but meanwhile also allowing to set/ get data between the top-level module and the queues. This is necessary because we need to decide somehow where to put the new clients and to be able to add them in the queue.

2. Problem analysis, modelling and use case

Queues are commonly used to shape real-world domains. The main purpose of a queue is to provide a place for a "customer" to wait before receiving a "service". Queue-based management systems are interested in minimizing the time that "customers" wait in line before serving. A way to minimize the waiting time is the existence of several servers, meaning several queues in the system (each queue is considered to have an associated processor), but this approach increases the costs of the service provider.

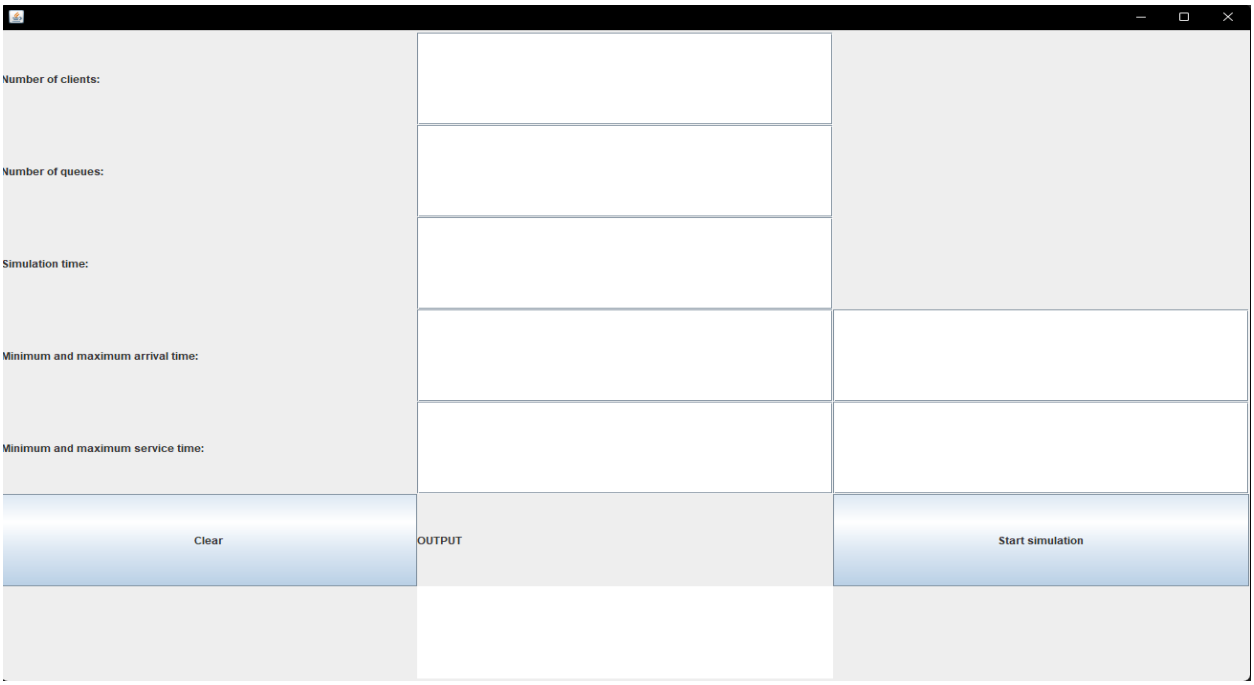
Functional requirements

I) The program must give the user the ability to enter the number of clients
II) The program must give the user the ability to enter the number of queues
III) The program must give the user the ability to enter the simulation time
IV) The program must give the user the ability to enter the arriving interval in which a client can show up to a queue by specifying the minimum and maximum arrival time (the minimum arrival time must be greater than one and the maximum arrival time must be lower than Tsimulation-1)
V) The program must give the user the ability to enter the service interval which represents how much time a clients must wait at a queue before leaving by specifying the minimum and maximum service time (the minimum service time must be greater than one and the maximum service time must be lower than Tsimulation-1)
VI) The program must give the user the ability to start the simulation
VII) The program must give the user the ability to clear the output window
VIII) The program must give the user the ability to observe the evolution of the queues over time

The input should consist of these factors

I) N = number of clients (which will be randomly generated)
II) Q = number of queues
III) Tsimulation = the maximum simulation interval
IV) TArrivalMin and TArrivalMax = minimum and maximum arrival time of a client

TServiceMin and TServiceMax = minimum and maximum service time of a client



The output should consist of these factors

I) Current time
II) A list of the waiting clients sorted by arrival time
III) A list of all the queues
IV) Average waiting time
V) Average service time
VI) Peak hour

The user can read from this report the client's status in each time interval/ second, and their waiting time. The queues current load is also presented. The last lines of the file contains the average waiting time of those clients that have been fully processed (entered a queue, but also exited it), the average service time of all clients, and the peak hour, meaning when the queues had the biggest number of clients.

3. Design

For implementing the problem described above, the best approach is to use Threads. This helps us to run simultaneously the queues and access their data, by using synchronized functions and volatile variables. It helps us, so that we can easily use one another methods or variables from different threads and at the same time without any problem. In java, one may easily implement a Thread, a class must implement the Runnable interface or extend the Thread class and afterwards implement the run() method.

To make the threads safe, a synchronized collection is used in the Queue ("LinkedBlockingQueue"). It is a queue that is optimized for concurrent programming in Java (working with multiple threads). The variables are also pre-defined "Atomic" variables, which are optimized for the same reason. The volatile variables are stored in the main memory, not in the CPU cache, to maintain thread safety. Besides that, the "Atomicity" also solves the problem of bad timing between threads. By using this, we will never happen to update a value, but get the old result back.

The Blocking Queue also helps us with the blocking of the thread. In case there are no more elements left in the list, or the maximum limit is exceeded, the current thread will be blocked. It can be un-blocked by adding a new element or taking out one.

I used the MCV architecture in this project so here are the packages I created:

I) View: The View package contains one class: "MainMenu". This class basically implements the front end of this program and is the place where the user can input the simulation parameters;
II) Controller: The Controller package contains one class: "MainController". This class is what connects the front-end, which in our case should be the output file and GUI and the model;
III) Model:
IV) Exception: I created this package to handle any exceptions that might get thrown out

UML Diagram

4. Implementation

Task class:

This class has 4 parameters: the ID, arrival time, service time and waiting time for a task, one constructor with parameters, the corresponding getters and setters and a method for determining the task with the smaller arrival time.

```
public class Task implements Comparable<Task> {

    private int ID;
    private int arrivalTime;
    private int serviceTime;
    private int waitingTime = 0;

    //constructor with parameters
    public Task(int ID, int arrivalTime, int serviceTime) {...}

    //getters
    public int getID() { return ID; }

    public int getArrivalTime() { return arrivalTime; }

    public int getServiceTime() { return serviceTime; }

    public int getWaitingTime() { return waitingTime; }

    //setters
    public void setServiceTime(int serviceTime) { this.serviceTime = serviceTime; }

    public void setWaitingTime(int waitingTime) { this.waitingTime = waitingTime; }

    //method
    @Override
    public int compareTo(Task task) {...}
}
```

Queue class:

This class also has 4 parameters, one constructor without parameters, corresponding getters and setters and a method called “addTask” which basically adds a task to a queue but also takes in account the service time of the task and adds it to the waiting time of the queue

```java
public class Queue implements Runnable {

    private volatile BlockingQueue<Task> queue;
    private AtomicInteger waitingTime;

    private Task currentTask = null;
    private boolean stop = false;

    //constructor without parameters
    public Queue() {...}

    //getters
    public BlockingQueue<Task> getQueue() { return queue; }

    public AtomicInteger getWaitingTime() { return waitingTime; }

    public Task getCurrentTask() { return currentTask; }

    //setter
    public void setStop(boolean stop) { this.stop = stop; }

    //method
    public void addTask(Task task) {...}

    @Override
    public void run() {...}
}
```

Strategy interface:

I implemented this interface to add a task according to a policy

```java
public interface Strategy { //interfata pentru a adauga un task in functie de o strategie

    void addTask(List<Queue> queues, Task task);
}
```

ConcreteStrategyQueue class:

This class implements the Strategy interface and it takes as inputs a task and a list of queues and adds the task to the queue with the smallest number of tasks.

```
public class ConcreteStrategyQueue implements Strategy {

    @Override
    public void addTask(List<Queue> queues, Task task) { //adaugam task-ul la coada cu cele mai putine task-uri

        int index = 0, min = Integer.MAX_VALUE;

        for (Queue q : queues) { //parcurgem  cozile

            if (q.getQueue().size() < min) { //si cautam cea cu cele mai putine task-uri

                min = q.getQueue().size();
                index = queues.indexOf(q);
            }
        }

        queues.get(index).addTask(task); //si adaugam task-ul in coada gasita
    }
}
```

ConcreteStrategyTime class:

This class implements the Strategy interface and it takes as inputs a task and a list of queues and adds the task to the queue with the smallest waiting time.

```
public class ConcreteStrategyTime implements Strategy {

    @Override
    public void addTask(List<Queue> queues, Task task) { //adaugam task-ul la coada cu timpul de asteptare cel mai scurt

        int index = 0, min = Integer.MAX_VALUE;

        for (Queue q : queues) { //parcurgem  cozile

            if (q.getWaitingTime().get() < min) { //si cautam cea cu cel mai mic timp de asteptare

                min = q.getWaitingTime().get();
                index = queues.indexOf(q);
            }
        }

        queues.get(index).addTask(task); //si adaugam task-ul in coada gasita
    }
}
```

Scheduler class:

This class is responsible for creating and starting Q (the number of queues) threads and setting up the output text file (LogOfEvents.txt). Also, it has a method for changing the current strategy from the time to queue strategy or vice versa

```
public class Scheduler {

    private List<Queue> queues;
    private int maxClientsPerQueue;
    private int maxNoQueues;
    private Strategy strategy;

    public FileWriter writer;

    //constructor with parameters
    public Scheduler(int maxClientsPerQueue, int maxNoQueues) {...}

    //getter
    public List<Queue> getQueues() { return queues; }

    //methods
    public void dispatchTask(Task task) { strategy.addTask(queues, task); }

    public void changeStrategy(SelectionPolicy policy) {...}
}
```

SimultionManager class:

This class is the one that handles the writing of the output of the simulation. It has as parameters the ones from the GUI, a list of tasks that are going to be processed (created randomly), a selection policy which indicates what strategy we are going to use and a stop boolean variable which will be our simulation's on/off switch. There is also a constructor with parameters and a method which creates N (the number of total tasks) random tasks that are placed in the "generatedTasks" list and are also sorted by their arrival time so the user can visualize the output clearer. During the simulation, we are going to display the output both in a text file and in the GUI. First of all, we are going to go through all the tasks that are waiting and check if there are any that are ready to be processed (if their arrival time is equal to the current time of the simulation) and if there are, then we are going to add them to queue according to the strategy that we chose. Secondly, we are going to traverse the queues and show what is happening in each one. If a queue does not have any active tasks then it is going to be "closed".

```java
private int minArrival;
private int maxArrival;
private int minService;
private int maxService;

private SelectionPolicy selectionPolicy = SelectionPolicy.SHORTEST_TIME;
private Scheduler scheduler;
private MainController controller;

private List<Task> generatedTasks; //in lista asta vom stoca task-urile ce le vom genera aleatoriu

private int peakHour;
private int totalTasks = 0;

private boolean stop = false; //switch-ul ce dicteaza daca simularea continua

public static int sumServiceTime = 0;
public static int completedTasks = 0;
public static int sumWaitingTime = 0;

//constructor with parameters
public SimulationManager(int N, int Q, int simulationTime, int minArrival, int maxArrival, int minService, int maxService, MainController controller) {...}

//method
private void generateRandomTasks() {...}

@Override
public void run() {...}
```

MainMenu class:

This class is the one that manages the GUI. I used a grid layout with three columns to make the design more user friendly (especially for the minimum and maximum combos). Also, I used a ”JTextArea” to display the output directly in the GUI every second.

```java
private JTextArea output = new JTextArea();

public MainMenu() {...}

//getters
public int getN() { return Integer.parseInt(N.getText()); }

public int getQ() { return Integer.parseInt(Q.getText()); }

public int getSimulationTime() { return Integer.parseInt(simulation.getText()); }

public int getMinService() { return Integer.parseInt(minService.getText()); }

public int getMaxService() { return Integer.parseInt(maxService.getText()); }

public int getMinArrival() { return Integer.parseInt(minArrival.getText()); }

public int getMaxArrival() { return Integer.parseInt(maxArrival.getText()); }

public JButton getStartBtn() { return startBtn; }

public JButton getClearBtn() { return clearBtn; }

public JTextArea getOutput() { return output; }

//setter
public void setTextArea(String s) { this.output.setText(s); }
```

5. Results

For testing we used three data sets for simulation parameters. The user of the application can get a very beautifully detailed simulation on steps for queueing up N random clients. It is easy to also calculate the average and to estimate a real-life situation also with that.

6. Conclusions

As a conclusion, I would say that this program would come in handy for some smaller restaurants, hotels, motels where the owner can not decide if there's a worth of upgrading his/ her business.

The project for myself as the developer of it, was a good exercise and helped me understand the concept of concurrency in Java, by that I mean the use of threads and their advantages. I learnt that time management is an important aspect of anything and even a small error could cause a big failure in bigger projects. Therefore, paying attention to thread safety is important.

Although I did not know much about the subject of threads, I learned a lot. Understanding and making it work was a lot easier when I had to do the research, try out the given examples, and make the connections between each element.

7. Bibliography

I) YouTube
II) Stack Overflow