Queue Simulator

Pop Roxana

Group 30424

Contents

[Objective 3](#_Toc450083421)

[Description 3](#_Toc450083422)

[Problem Analysis 3](#_Toc450083423)

[Data Modeling 3](#_Toc450083424)

[Use cases 3](#_Toc450083425)

[Design 4](#_Toc450083426)

[Packages 4](#_Toc450083427)

[Classes 5](#_Toc450083428)

[UML Diagram 10](#_Toc450083429)

[Results 11](#_Toc450083430)

[Conclusion 14](#_Toc450083431)

[Further Developments 14](#_Toc450083433)

[Bibliograpy 14](#_Toc450083436)

# Objective

The objective of this project is to implement a program that simulates a queues-based system. The simulation is made so that the waiting time is minimized, or at least the number of items that are in a queue at one moment.

# Description

The queue problem is a real-life one. Actually, in programming, the meaning of the word was inherited from real-life. It means the same thing: a structure that allows new elements to enter on one way, and older elements to exit on the other. Often, the example of people waiting in a queue at the shop in order to be served is given.

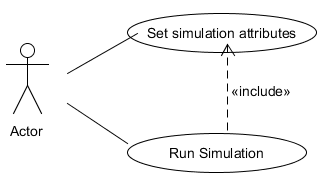
The application goal is to simulate such a system. In order to do that, tasks (people in the store) will enter the queue at random times, one or more at the time, and new servers (queues) can open if there are too many people in the other queues. The number of servers though will have an upper bound, just like a store cannot have an infinite number of pay points.

# Problem Analysis

## Data Modeling

Mainly, the models I Identified were the simulator (that actually simulates the whole process), the tasks (correspondents of clients) and the servers (correspondents of pay points). I also needed a class to generate tasks randomly, say, a task generator that would map the behavior of clients going randomly to the pay points. Now, in real life, the servers would be all active at the same time, each processing its task, and in the same time, new clients (tasks) would arrive. From the programmatic point of view, the simulator, the servers and the task generator should not run successively, but concurrently. In order to achieve this concurrency, threads must be used.

# Use cases



# Design

## Packages

I designed my application on, again, the MVC pattern. The project is divided into four main packages: model, view, controller and utitlities.

**Model**

The model contains the classes which perform the actual simulation. They are the following:

* *Scheduler*;
* *Server* ;
* *Simulator*;
* *Task* ;
* *TaskGenerator* .

The simulator performs, well, the simulation in a given interval. It retrieves at random period of times some tasks from the task generator, it sends them to the scheduler and the schedular sends each to the appropriate server, which then processes it.

**View**

The view package actually contains one class only, a frame, that is *Frame*. This frame will represent the interface, that will be updated by the classes in the controller in relation to the changes in the model.

**Controller**

The classes in this package make the connection between what is happening in the simulator and what is outputed on the frame. They are as follows:

* *Controler;*
* *ButtonListener* ;
* *Main*;

**Utilities**

This package contains two utility classes:

* *Constants;*
* *RandomGenerator.*

## Classes

In this section I will explain in detail the contents and purpose of each class individually.

#### The model package

I will take the classes in a sort of bottom up approach, meaning I will start with the simplest one and as I explain them I will convey the way they are all linked.

##### Task

This class has two fields, one constructor and five methods, as seen bellow:

private long arrivalTime;

private long processTime;

public Task(long processTime);

public void setArrivalTime(long arrivalTime);

public void setProcessTime(long processTime);

public long getArrivalTime();

public long getProcessTime();

public String toString();

The two fields refer to the time at which the task goes to a server (arrival time), and to how long it takes for that server to process it (process time).

##### TaskGenerator

This class has the sole purpose of generating, at random times, some tasks. It stores the tasks in a BlockingQueue. The reason for that is that this class implements Runnable, therefore it runs in its own thread, and the tasks it creates are accessible to other threads (through getTask), so some form of synchronization must be imposed.

private static final int MIN\_SLEEP\_TIME = 200;

private static final int MAX\_SLEEP\_TIME = 2000;

private BlockingQueue<Task> generatedTasks;

public TaskGenerator();

public void run();

public Task getTask();

##### Server

This class has the main purpose of processing the tasks, which is done in the run method. As I emphasized before, there was a also a need for this class to be a thread, so it implements Runnable and in its run method it handles the tasks coming in. For synchronization purposes, the collection of tasks it handles is again, a blocking queue, and the waiting time (the time in which all tasks could be performed) is an atomic long. There is some extra logic in this class, so that at the end of simulation more data can be given as output, like: average waiting time, service time, empty time and peak hour. The servers also have access to the frame, so that their status is synchronized with what is being displayed in the interface. Even this is a bit against the MVC pattern, I found no better solution for the desired result.

private int id;

private BlockingQueue<Task> queue;

private AtomicLong waitingTime;

private AtomicInteger noOfTasks;

private long totalServiceTime =0;

private long totalWaitingTime=0;

private long totalEmptyTime=0;

private int totalNoOfTasks=0;

private boolean wasEmpty;

private LocalTime timeOfQueueEmpty;

Frame frame;

public Server(int id,Frame frame);

public AtomicLong getWaitingTime();

public void setWaitingTime(AtomicLong waitingTime);

public void run();

public void addTask(Task t);

public Task[] getTasks();

public int getNoOfTasks();

public int getId();

public BlockingQueue<Task> getQueue();

public void setQueue(BlockingQueue<Task> queue);

public boolean isEmpty();

public long getAverageWaitingTime();

public long getAverageServiceTime();

public long getEmptyQueueTime();

##### Scheduler

This class has the purpose of sending the tasks to the appropriate server. It is actually the class that assures minimization of the waiting time. Actually, in its implementation I simply assure that the task will be sent to the server whit the minimum number of tasks being currently processed. It has the following fields and methods:

private Frame frame;

private ArrayList<Server> servers;

private int maxNoOfTasks;

private long peekTime;

public Scheduler(Frame frame);

public ArrayList<Server> getServers();

public void setServer(ArrayList<Server> servers);

public ArrayList<Task[]> getTasks();

public void dispatchTasksOnServer(ArrayList<Task> tasks;

private boolean updateMaxNoOfTasks();

private Server getServerWithMinNoOfClients();

public boolean isDone();

##### Simulator

The simulator runs in its own thread as well. It’s the class that glues together the whole model. It must simulate the process in a given interval, so in its runs method, it will finally sleep for that interval. But, it doesn’t do it at once. It gets some tasks from the generator, sends them to the scheduler, then goes to sleep from a random period of time. That is, until it reached the end of the interval. This assures that tasks arrive at random period of times, of course, real life like.

The fields and methods can be observed bellow:

private DateTime currTime;

private TaskGenerator taskGenerator;

private Interval interval;

private Scheduler scheduler;

private Frame frame;

public Simulator(TaskGenerator taskGenerator, Frame frame);

private ArrayList<Task> getTasks();

public void run();

#### The View package

The View package contains one class only, called *Frame*. Here, the display of the data is done. Each server updates the frame when receiving a new task or finishing an old one, and it does that through the *updatePanel* method, where the sub-panel corresponding to the server is rewritten with the updated data. The main panel is used during simulation for visualization of the servers’ activity, but once the simulation is done, a report will be shown there. In order to load that report, the displayOutputData method is used from the simulator class.

private JPanel upperPanel;

private JPanel mainPanel;

private ArrayList<JPanel> contentPanels;

private JTextField startSim;

private JTextField endSim;

private JTextField minProc;

private JTextField maxProc;

private JTextField minTaskInt;

private JTextField maxTaskInt;

private JTextField minNoServers;

private JTextField maxNoServers;

private JTextField minNoTasksPerTime;

private JTextField maxNoTasksPerTime;

private JTextField maxNoTasksPerServer;

private JButton start;

private JButton addAccButton;

public Frame(String title);

public void updatePanel(int panelId, Task[] tasks );

private void addInputComponents(JPanel panel);

public void addButtonListener(ActionListener a);

public ArrayList<String> getFieldValues();

public void displayOutputData(String outputData)();

#### The Controller package

##### Controler

In the controller the Frame class is instantiated (in the constructor), then the Simulator class is instantiated (in the start method) and its thread started. The frame instance will be passed to the simulator, from where it is passed on up to the servers. It contains the following fields and methods:

private Frame frame;

public Controler();

public void start();

##### ButtonListener

This class, as the name suggests it, is an action listener for the only button on the frame. So, when this is pressed, the start method of the controller is called, leading to the start of the simulation. Before that, all settable attributes in the simulation are retrieved from the frame (such as simulation interval, no of servers etc.). The attributes and methods are presented below:

private Frame frame;

private Controler controler;

public ButtonListener(Frame frame,Controler controler);

public void actionPerformed(ActionEvent arg0);

private void retrieveInputData();

##### Main

This class contains only the main method that each java app should have, and there the controller is instantiated.

#### The Utilities package

##### Constants

The utilities package contains the quite usual constants class, even if in this particular case they’re not exactly constants: they are set only once, in the beginning, as they are the simulation parameters. In the beginning, they have some default values. They are the following:

public static long MIN\_PROCESS\_TIME = 2000;

public static long MAX\_PROCESS\_TIME = 20000;

public static long SIMULATION\_START = 1000;

public static long SIMULATION\_END = 30000;

public static int MIN\_NO\_OF\_SERVERS = 3;

public static int MAX\_NO\_OF\_SERVERS = 5;

public static long MIN\_INTERVAL\_BETWEEN\_TASKS = 1000;

public static long MAX\_INTERVAL\_BETWEEN\_TASKS = 1000;

public static int MIN\_NO\_OF\_TASKS\_PER\_SECOND = 1;

public static int MAX\_NO\_OF\_TASKS\_PER\_SECOND = 1;

public static int MAX\_NO\_OF\_TASKS\_PER\_SERVER =5;

##### RandomGenerator

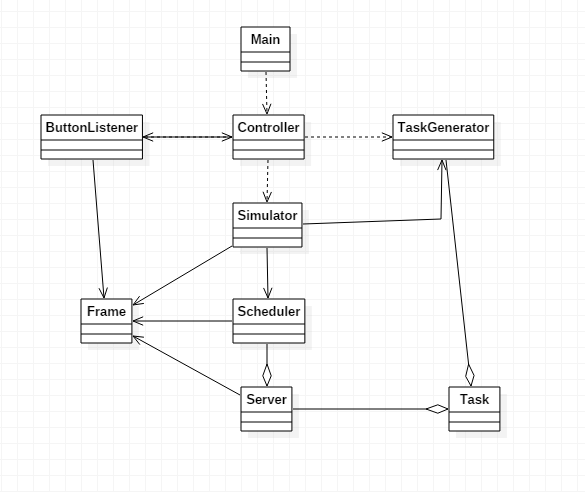
This project required a lot of random numbers being generated between a minimum and maximum, so I decided to implement this helpful class, containing methods that provide the required behavior. They are the following:

public static int getRandomIntInRange(int min, int max);

public static long getRandomLongInRange(long min, long max);

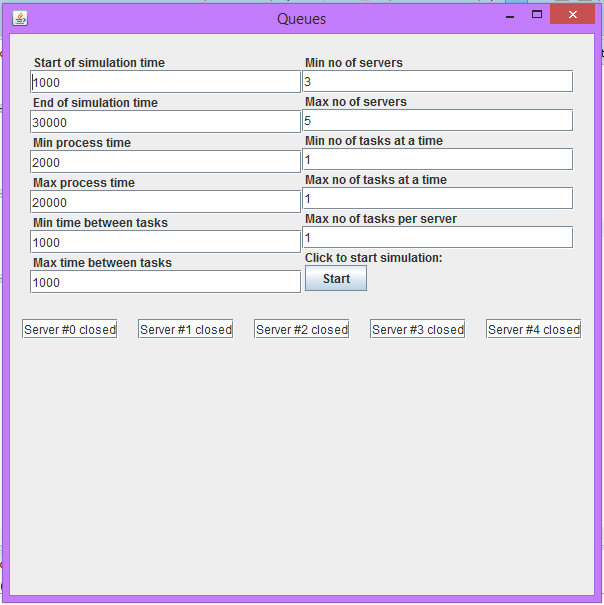
## UML Diagram

The UML diagram can be observed below:

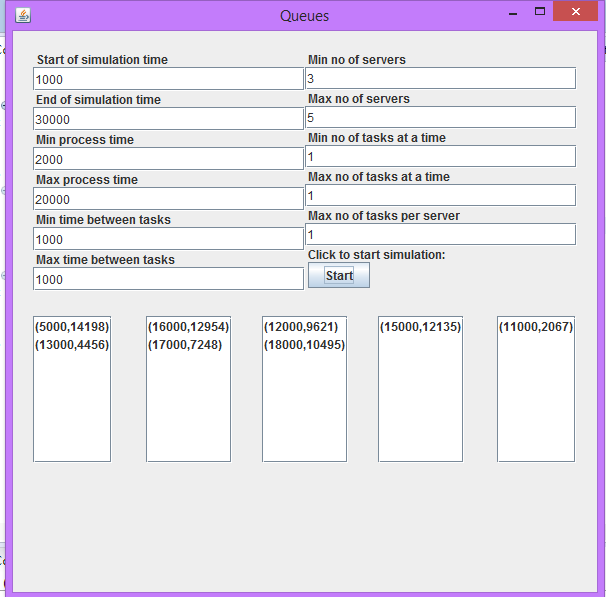


# Results

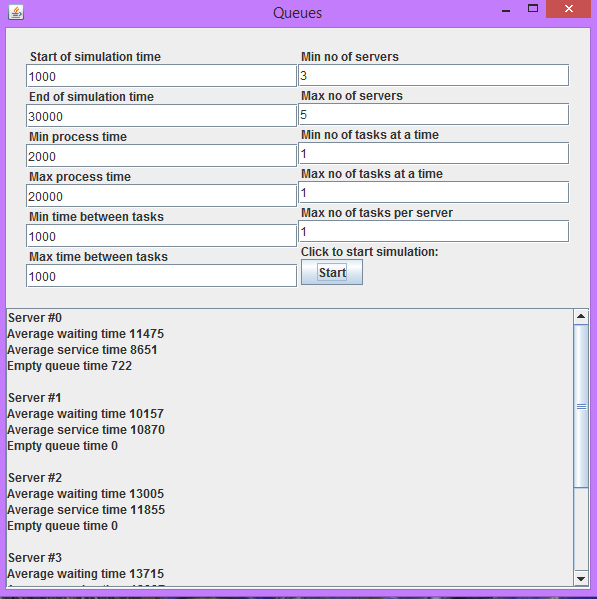
This is how the interface looks when the program is started:



During the simulation:



And when it ends:



# Conclusion

# This is a simple project meant to simulate a real-life behaviour: tasks in a queue. In order to do that, it uses threads that run concurently. From an academic point of view, that was actullay the purpose of it, to understand how threads work, how they can be used, and maybe, most important of all, why they should be used.

# Further Developments

# Further developments can be brought to the project itself. As it is, even though it may fulfill the basic functionality, it can be improved a lot. The interfce, as maybe noticed, is not estethic at all. The only used pattern is the MVC, and even that not that rigurously, so the design could be improved from the OOP point of view. Some threads remain in some infinite loops, even when their job is done (the servers), and that is uselless and resource consuming.

# Once the functionality of the project as it is is covered, there is a new behaviour that could be added. In its current state, when a new server is added, the tasks already scheduled to some servers remain as the are, and only new tasks are given to the newly-created server. Now, a better approach would be to redirect some of the tasks already waiting, and not the most recent ones, but based on their priority.

# Bibliograpy

1. <https://docs.oracle.com/en/java/>
2. <http://stackoverflow.com/>
3. <http://tutorials.jenkov.com/java-reflection/index.html>