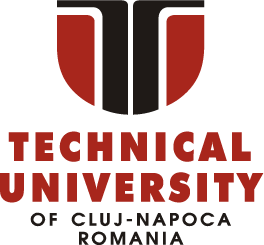
**TECHNICAL UNIVERSITY OF CLUJ-NAPOCA**

2nd year of study, Computer Science



Laboratory Work – Assignment 3

**Multithreaded simulation of Task execution on multiple servers**

Lecturer: Prof. Dr. Eng. Ioan Salomie

Teaching Assistant: Delia Balaj

Student: Diana-Gabriela Popescu

Group: 30425

**Table of contents**

**1. Introduction**

1.1 Task objectives . . . . . . . . . . . . . . . . . . . . . . . . . .page 3

1.2 Personal approach . . . . . . . . . . . . . . . . . . . . . . . .page 3

**2. Problem description**

2.1 Problem analysis . . . . . . . . . . . . . . . . . . . . . . . . .page 3

2.2 Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .page 4

2.3 Scenarios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .page 4

2.4 Use cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .page 5

**3. Projection**

3.1 UML diagrams . . . . . . . . . . . . . . . . . . . . . . . . . . page 7

3.2 Data structures . . . . . . . . . . . . . . . . . . . . . . . . . . page 8

3.3 Class projections . . . . . . . . . . . . . . . . . . . . . . . . page 9

3.4 Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 12

3.5 Relationships . . . . . . . . . . . . . . . . . . . . . . . . . . . page12

3.6 Packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 12

3.7 Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 13

3.8 Graphical User Interface . . . . . . . . . . . . . . . . . . page13

**4. Implementation and testing** . . . . . . . . . . . . . . . . . . . . . . . .page 13

**5. Results** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 13

**6. Conclusions** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 14

**7. Bibliography** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 15

**1. Introduction**

1.1 Task objectives

The task of the assignment is defined as it follows: “Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients’ waiting time”. In other words, one should create a system (by GUI interface) that is able to manipulate tasks and to perform basic operations on them.

1.2 Personal approach

This documentation paper aims to present a way of solving the problem of task processing. As for user interface, there will be developed algorithms for user to interact with the Graphical User Interface (GUI) and, for some methods, the output will also be displayed in the console. The solution is obtained by means of implementing several operations specific for tasks handling such as computing the average waiting time. This type of operations were chosen due to the fact that they are the most used and the most important operations regarding task processing.

**2. Problem description**

2.1 Problem analysis

The analysis of a problem starts from examining the real mathematical model of polynomials or the model we confront with in the real world and passing the problem through a laborious process of abstractization. Hence we identify our problem domain and we try to decompose it in modules easy to implement. Always, having a good model will ease the way the operations are performed and will make more complex programs be clear to read and easy to maintain. My implementation of the solution started with trying to find a suitable and clear model for the tasks and servers. Also, due to the fact that the user has an important role in running the application, the Graphical User Interface was also been given a high importance in the process of problem analysis.

Queues are commonly seen both in real world and in the models. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount its "clients" are waiting in queues.

One way to minimize the waiting time is to add more servers, i.e. more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the supplier. When a new server is added the waiting clients will be evenly distributed to all current available queues.

The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the clients spend waiting in queues and outputs the average waiting time.

To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time depend on the individual clients – when they show up and how much service they need. The finish time depends on the number of queues, the number of other clients in the queue and their service needs. This idea will also be developed when implementing the algorithms.

2.2 Modeling

Based on the information presented above, I thought on the actual implementation of the solution. I started to think about what classes are required and how classes should be organized in packages. Also, I realized the need of implementing ‘Runnable’ interface for some classes.

2.3 Scenarios

The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the clients spend waiting in queues and outputs the average waiting time.

2.4 Use cases

The use cases are strongly related to the user running the application. Therefore, the user should choose:

- minimum and maximum interval of arriving time between clients ( tasks )

- minimum and maximum service time

- number of queues

- simulation interval

- some other information you may consider necessary

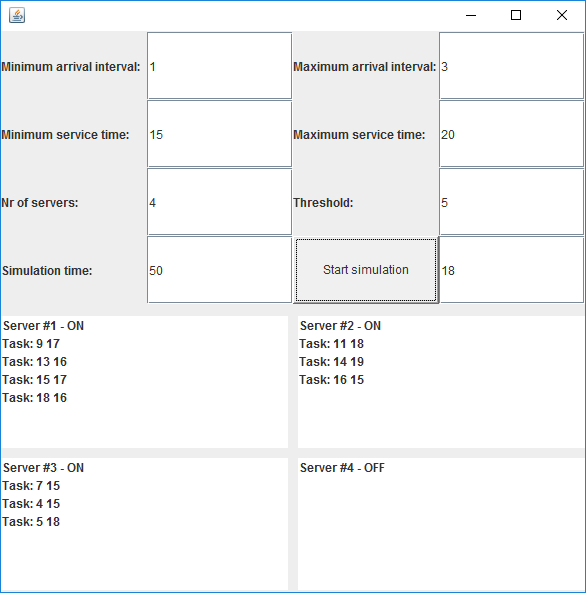


Illustration 1 : Graphical User Interface

Then, the user will get this minimal output, at the final of the simulation:

- average of waiting time, service time and empty queue time for 1, 2 and 3 queues for the simulation interval and for a specified interval

- log of events and main system data

- queue evolution

- peak hour for the simulation interval

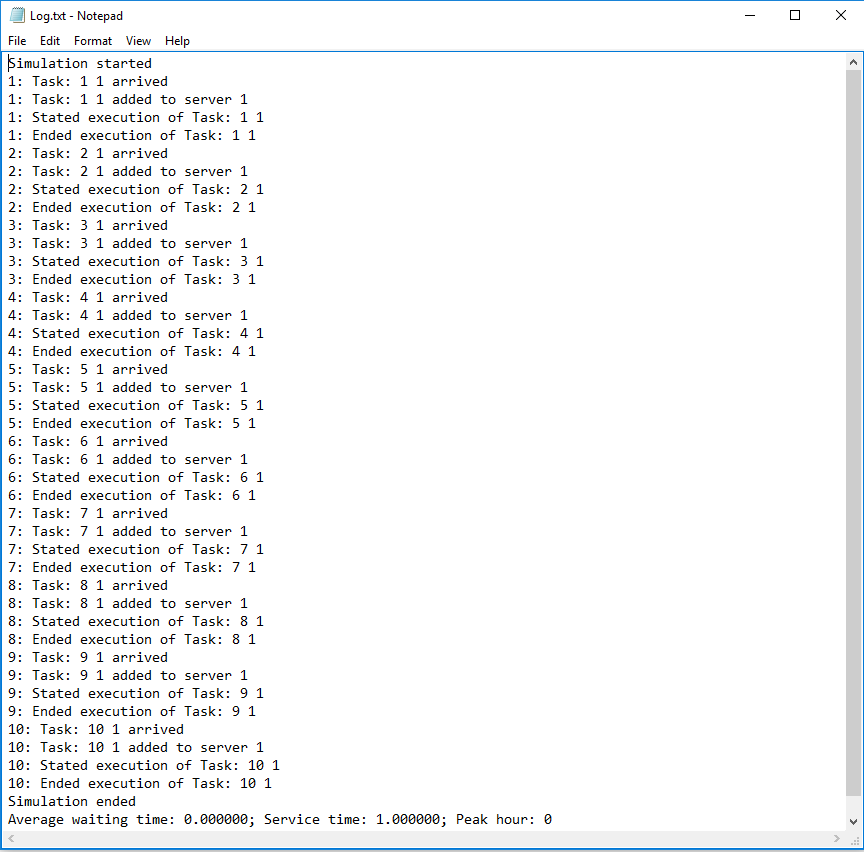


Illustration 2 : Graphical User Interface ( output )

**3. Projection**

3.1 UML diagrams

a) Use-case diagram

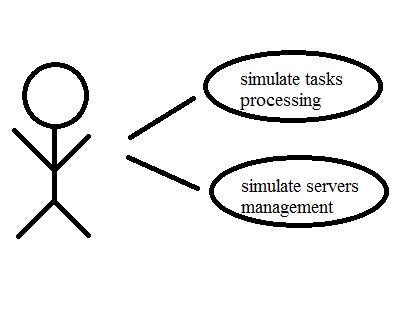


Illustration 3 : Use case diagram

The use case diagram presents the actor, which is the user that interacts with the application. He can perform several actions such as addition, subtraction, multiplication, division, integration and differentiation. The user can also set degrees and coefficients by means of the Graphical User Interface.

b) Class Diagram

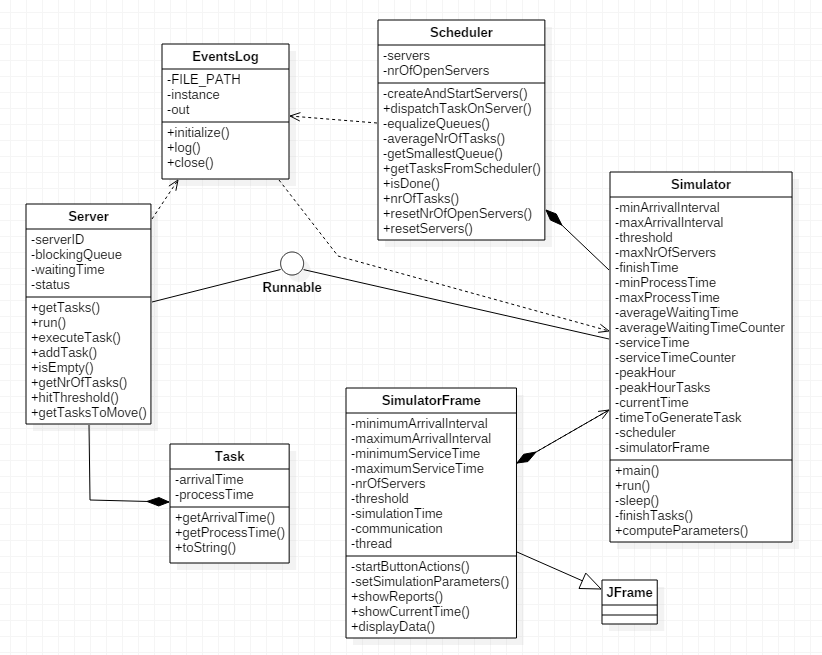


Illustration 4 : Class diagram

As presented in the class diagram, there are various relationships between the classes and interfaces of the system.

3.2 Data structures

The data structures used in the application are either primitive data types ( int or double ) or new types that are defined by the designed classes and interfaces. Furthermore, there were used arrays of the types presented above. Not to be forgotten is the use of GUI-like types.

3.3 Class projections

Class projection refers mainly to how the model was thought, how the problem was divided in sub-problems, each sub-problem representing more or less the introduction of a new class. For the beginning, it has to be mentioned that, in my design, I used four one single package for organizing the classes and the interfaces. This is: “threadsAssign3”. The mentioned package is to be described next, along with the classes and interfaces belonging to it.

The application will need a simulator class. This class will have a run method which will handle the task creation and will update the UI every second. The created tasks will then need to be handed over to a scheduler, which will handle their distribution on servers. The scheduler needs to create the number of servers requested by the user, and then start them when the need arises. The scheduler will decide which server the task should be executed on, and then send the task to the appropriate server, which will put it in its queue and execute it at the appropriate time.

All the actions will also need to be logged. For that, the application will need a logger class which will write every relevant action in a .txt file.

Package “threadsAssign3” – contains all the design, related to the user interface (GUI) and the logic of the application.

* class **SimulatorFrame**- handles the graphical user interface. When initialized it creates two panels: one for the user input, start button and status field, and one for the visual representation of the queues. The user input area is composed of seven input text fields along with their corresponding labels: the minimum and maximum arrival interval, the minimum and maximum service time, the number of servers, the threshold for the servers and the simulation time. The start button is the one which starts the simulation. In case another simulation is already started, it stops it, resets all the state and statistics variables to their default values and starts a new simulation. The simulation parameters are either read from the text fields, or, if the text fields are empty, set to some default values. Before the parameters are set, the text fields are checked, and in case either of them contains anything other than an integer the output text field will be set to an error message and the simulation will not be started. The output text field will also display the simulation time, with the help of a function called by the simulator, and the simulation results.

The area used for the visual representation of the queues is created at the call of the displayData function, which will create n JLists, n being the number of servers, and then display them in the panel. The function is called by the simulator every second, sending as parameter a list of string arrays, representing the tasks and names of the servers.

* class **EventsLog** - is a singleton which handles the logging of the events and their output to the “Log.txt” file. It has an initialize method, which will create an instance of the class, and open a PrintWriter for writing in the log file. It will then print a “Simulation started” message in the file. The logging of the events is performed with the help of the log function, which takes as parameter a string, and prints it in the file, preceded by the current time of simulation.

When the simulation is over, the simulator will call the close method, taking as parameters the average waiting time, service time and peak hour. This will print the statistics in the file, close the PrintWriter and then try to open the file with the default .txt app.

* class **Simulator** - is the one containing the run method started when the user presses the “Start simulation” button. The filled fields are parsed to integers and set as simulation parameters. If any field is left empty, the default values will be loaded.

The run method will generate tasks according to the specified or default parameters. Each generated task is then sent to the scheduler, which distributes it to the servers. The run method will then refresh the graphical user interface, compute the peak hour, sleep for one second, and then repeat the process until the final time is reached.

The Simulator class also contains the statistics variables, such as the average waiting time, simulation time and peak hour, along with the methods necessary for their computation.

* class **Task** - contains two integers: the arrival time, and the process time, the time necessary for the task to be processed. It has a constructor which takes as parameter both of these variables, and a toString method which is used when composing the array of strings for the graphical user interface representation of the servers.
* class **Scheduler** - is responsible for the management of the servers and for the distribution of tasks. On instantiation it creates a list of n servers, n being the maximum number of servers specified in the input field or the default value for the maximum number of servers.

When the dispatch function is called, the scheduler checks if the last opened server hit the threshold, and if it did, and the number of open servers is smaller than the maximum number of servers of the simulation, it starts the thread for the next server which has not yet been and then equalizes the queues. After the scheduler checks if it needs to open a new queue, it places the new task on the smallest queue available.

* class **Server** - The Servers are the objects which handle the tasks’ execution. Their main component is a blocking queue, which holds the tasks in the queue. The server is also assigned an ID, useful for the display on the Graphical User Interface, an atomic integer which holds the waiting time in the queue, and a status in String format, which has the “OFF” value at creation, and which gets the value “ON” at the first insertion of a task on the server. The addTask method simply receives a task from the scheduler and adds it to the blockingQueue, writing the event in the logger. The Server class implements runnable, since its main functionality relies on threads. The run function contains a while loop which runs infinitely and calls a function which executes each task. The execution of a task means putting the thread to sleep for the number of seconds equal to that task’s processing time. The function will log the start of the thread, call the sleep function, and then log the end of the thread.

The class also contains one method for removing n elements from the queue, method which is needed when equalizing (rebalancing) the queues. The Scheduler calls this function and removes the elements exceeding the average of the queues. For each element that is removed, the app will log it as a task which moved out of that server’s queue.package.

3.4 Interface

This section being already developed in detail until now, I will remember briefly some important facts. The user interface is mainly realized by means of “java.swing” and “java.awt” packages. I used instances of JFrame, JPanel, Button, JLabel. Also, I used GridLayout for organizing components inside the frame and panels. The button and the text fields are the main way of the user interacting with the application.

3.5 Relationships

As presented in the class diagram, there are various relationships between the classes and interfaces of the system. There are also some other dependencies and associations due to the fact that some methods use as parameter types of a different class.

3.6 Packages

The program is divided in one package, as mentioned before. The package “threadAssign3” contains classes: “Server”, “Task”, “Scheduler”, “Simulator”, “SimulatorFrame” and “EventsLog”.

3.7 Algorithms

The main algorithms regarding tasks are the following: handling tasks and computing different various parameters regarding them, like average waiting time or peak hour.

3.8 Graphical User Interface

An important fact to be mentioned is the use of “java.swing” package. The user interface is mainly realized by means of “java.swing” and “java.awt” packages. I used instances of JFrame, JPanel, Button, JLabel. Also, I used GridLayout for organizing components inside the frame and panels. The button and the text fields are the main way of the user interacting with the application.

**4. Implementation and testing**

The implementation was done in Eclipse and it was also tested in this environment. However the program should maintain its portability. Concerning the code implementation I did not make use of laborious algorithms, but I have rather stayed faithful to the classical algorithms. The personal touch in the implementation is also felt in the way the Graphical User Interface is thought.

Testing implies checking for any errors and warnings in the program or limitations of this program. When an exception might be thrown, it will be handled by the method. So, from this point of view, there are lots of errors that will be avoided this way. Other possible scenarios will be tried as future development.

**5. Results**

The application is user friendly and useful in performing basic task-related operations. As the application is developed on a Java platform, it is highly portable and allows it to run on several operating systems. The application is to be used by anyone who is interested in performing these types of operations and has a basic knowledge about the content.

**6. Conclusions**

The application can be further developed, by adding some other operations on polynomials. For me personally, the design of this system helped me think about the way classes should be organized in packages. Also, at first, I preferred using a strong and simple tool: ArrayLists. The app is, by no means, meant to be a perfect simulation of a set of servers receiving n tasks. The functionality of the application could be improved in a number of ways. Firstly, when rebalancing the queues upon the opening of a new queue, the scheduler will redistribute the tasks in servers which are above the average number of tasks waiting in all the servers’ queues. The method which performs this functionality in the Server class currently calls the method take on n elements and adds them to a list which is returned to the Scheduler. This means that the elements removed from the queue are taken from its front, rather than from its back. This is not the way things are supposed to happen, as this will lead to some of the tasks generated at an earlier time to be executed much later than tasks generated at a later time. The correct way to implement this would be to drain the queue to a collection, to remove and return the needed elements and to add the remaining elements back in the queue, in the same order.  
 Another known issue is the lack of synchronization on the critical regions in the app. The main problem here is with the regions in the server operating on the blocking queue. The reason such problems arise is because in the same object two threads will perform tasks concurrently, namely the server thread, which executes tasks, and the simulation thread, through the scheduler, which might be acting upon the blocking queue by adding, subtracting. This is not usually a big issue, but in case the threads get interleaved it can lead to some weird results.

On the subject of thread execution, another thing which might be useful would be to wait for the completion of the last tasks in the server. Currently the app ends the simulation once the queues are empty, but that means that at least one task will still be executing itself at the time the simulation has stopped. This will cause the report to show at least one task which was started but never ended.

From a data checking and validation standpoint, a useful feature would be for the parameters which expect minimum and maximum values to check that the value in the maximum field is not smaller than the one in the minimum field. Currently all the checking an app does refers to the value in the text field being an integer.

Another useful feature would be the ability of the user to specify a simulation interval in which the application to sample data for the reports, such as if, for example, the user is only interested in the average waiting time between 15 and 18, it will have the option of finding that out. Also, a useful feature would be the ability to switch between a “number of tasks” mode and a “time of tasks” mode, referring to the way the application decides the length of a queue. Currently, the app only implements the “number of tasks” approach, but a “time of tasks” approach could prove useful, in terms of waiting time optimization, in situations where the duration of a task is already known (such as in the case of this simulator).

This assignment provides a simplistic simulation of a network of servers, scheduling and executing different requests from users. It thus helps in providing a better understanding of the usefulness of parallelization and multithreading in app development, from an overall time of execution standpoint, as well as from a per-task waiting time standpoint, providing visible improvements compared to an iterative approach, in which only one thread would be responsible for handling all the tasks. It also helps in understanding the effects of using multiple threads by allowing for an easy comparison by means of running the same simulation with one or more servers.

**7. Bibliography**

* <http://staruml.io/>

-for downloading a tool for creating class diagrams.

* <http://bellekens.com/2010/12/20/uml-composition-vs-aggregation-vs-association/>

-for better understanding of the relationships between classes.