

QUEUES SIMULATOR

STUDENT: Iulia-Maria Farcas

Group: 30421

1. Main Objective

The purpose of this assignment is to design and implement a simulation application aiming to analyse queuing based systems for determining and minimizing clients' waiting time.

The application simulates the way the queues work by implementing a multithreading strategy in order to minimize the waiting time for clients. The application takes the input data from the in-text.txt file and will write the output values in out-text.txt file. The output file will show the evolution of the queues and clients and an average waiting time.

2. Analysis, assumptions and data modeling

This application should simulate customers waiting to receive a service, just like in the real world, they have to wait in queues, each queue processing clients simultaneously. The purpose is to analyse how many clients can be served in a certain simulation interval.

All the input data will be provided in the in-text.txt file and based on the parameters provided the application will generate randomly clients in order to simulate the behavior of our application. All customers will be served, regardless of their service time, with respect to maximum duration of the simulation.

The application will create and initialize the queues, by instantiating them and starting their corresponding threads. Each queue will have their own thread which will keep track of the clients that are waiting in that queue.

The application also incorporates a main thread that is used to simulate the app. The main thread will be the one will be the one that will keep track of the time and will stop the simulation when the time limit is met.

The output file, out-text.txt, is being updated after each second of the simulation passes, it will show the status of the waiting clients and of the queue and at the end of the simulation it will append the average waiting time.

Assumptions:

We assume that the input file respects a given pattern and some logical requirements such as minimum is smaller than maximum and that if the simulation time is over but there are clients left in the queue, those won't be served anymore.

The following data should be considered as input data read from a text file for the application:

- Number of clients (N);
- Number of queues (Q);
- Simulation interval (*tsimulationMAX*);

- Minimum and maximum arrival time ($t_{arrivalMIN} \leq t_{arrival} \leq t_{arrivalMAX}$);

- Minimum and maximum service time ($t_{serviceMIN} \leq t_{service} \leq t_{serviceMAX}$);

USE CASES

Use Cases: Start the application

Primary actor: User

Main Success Scenario:

1. The user gives in the command line an input file that respects the required pattern
2. The application reads the input file
3. The application creates the necessary objects for performing its tasks
4. The application simulates the queues properly
5. The application updates the output file
6. The application simulation ends successfully and it appends the average waiting time to the output file

Alternative scenario: the input file provided by the user is wrong, the simulation will not work properly

3. Design

The main class that controls the simulation of the application is called Scheduler. It controls the overall data flow, so it can be viewed as the controller of the application.

The classes that are using threads are implementing the Runnable interface and are defining a method called run(). In most cases, the Runnable interface should be used if the only overridden method is run() and no any other Thread methods.

If the interface Runnable is used in order to create the thread, when we start the thread it causes the object to call the run method, which will separately execute the thread.

This application has two classes that implement a Runnable interface, Server class and Scheduler class. The synchronization of the threads is made by making each thread wait 1 second before continuing its task, this way the threads are not overlapping when the method for writing in the file are called.

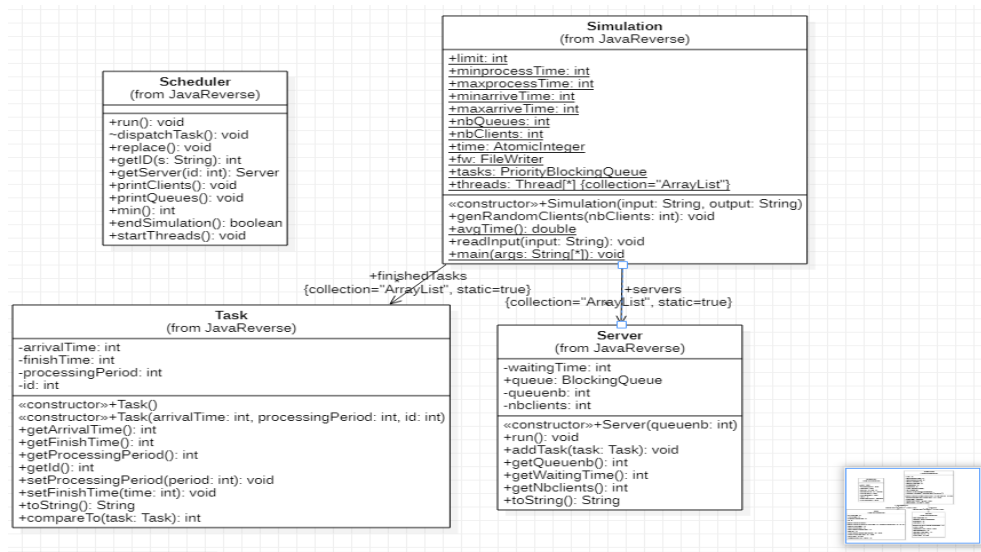
The Server class represents a queue. It has a thread in which the data corresponding to the first client in the queue are updated. The Server class implements a BlockingQueue in which the clients are stored. It uses a FIFO approach, so the head of the queue is always the customer that has been waiting the longest in the queue, the client that has to be served.

The Task class represents the customer. It contains the arrival time of the client, the processing time and an ID generated when the application starts. The class also implements Comparable interface, in this way, when the tasks are added to the waiting list, they'll be added in order with respect to their arrival time, the first element of the list being the client that arrives the earliest.

The Scheduler class is the controller of the application. It keeps track of the simulation time, prints the status of the clients and the queues, it replaces the dead threads with new ones and it also dispatches clients to the queue with the minimum waiting time. It uses an object of type AtomicInteger in order to count the time passed, this way the

thread's safety being assured, because of the operations, such as addAndGet, that work on atomical level. Atomicity deals with actions that have indivisible effects, and is a key concept in multithreading.

Next I will present the UML diagram:



I chose to implement the clients list with BlockingQueue and PriorityBlockingQueue because the classes are capable of blocking the threads that are trying to add and remove elements from the queue.

The server and threads lists are implemented with ArrayList because it's easy to store and access the data as it uses dynamic array to store the elements.

Task Class:

- Equivalent of a client in real life
- Implements Comparable Interface in order to store the clients from the smallest arrival time to the biggest
- Has getProcessingPeriod() and setProcessingPeriod() which are used to update for long the first client in the queue will be processed

Server Class

- Has a list of clients, implemented as a FIFO with BlockingQueue
- Implements Runnable
- The thread is the processing the queue, more exactly the first member in the queue, after each second the processing time of the client that's being served is decreased and when the processing time reaches 0 the client finished it's business and it's removed from the queue
- Also has a method that adds new clients to the queue, addTask(), and computes the waiting and finish time for each customer from the list

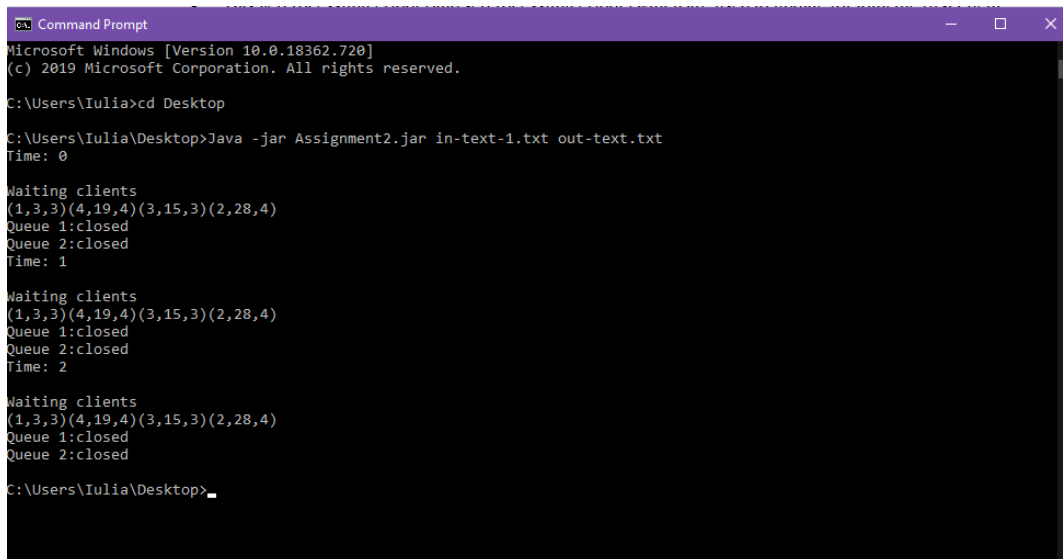
Scheduler Class

- Also implements Runnable
- Uses a thread to keep track of the simulation time
- It's responsible for assigning a client to the right queue, by finding the queue with the minimum waiting time, dispatch()
- It's also the class in which the server threads are started and replaced if they die, queues are opening and closing dynamically based on if they have clients or not, startThreads() and replace()
- Only the queues that have clients are started and only threads that died are replaced
- This class is the one that updates the output file after each simulation second passes, printClients() and printQueues()
- The class doesn't have its own attributes, it uses Simulation's class attributes in order to compute all the methods

Simulation Class

- It's responsible for reading the input data and instantiating the necessary objects needed for the simulation
- The average waiting time is computed in this class, the sum of the waiting time of all served clients over how many clients were served. Clients that were still in the queue when the simulation ended are not counted in the average waiting time, as their processing time was not over
- The attributes from this class are used by Scheduler class for controlling the flow of the simulation and by Server class in order to add the finished tasks to a new list which will later be used for computing the average waiting time
- The clients are generated randomly in this class based on the input data
- In the constructor method each thread is assigned to a server, thread which is later replaced in Scheduler if it dies

4. Running the application



```
Microsoft Windows [Version 10.0.18362.720]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Iulia>cd Desktop

C:\Users\Iulia\Desktop>Java -jar Assignment2.jar in-text-1.txt out-text.txt
Time: 0

Waiting clients
(1,3,3)(4,19,4)(3,15,3)(2,28,4)
Queue 1:closed
Queue 2:closed
Time: 1

Waiting clients
(1,3,3)(4,19,4)(3,15,3)(2,28,4)
Queue 1:closed
Queue 2:closed
Time: 2

Waiting clients
(1,3,3)(4,19,4)(3,15,3)(2,28,4)
Queue 1:closed
Queue 2:closed

C:\Users\Iulia\Desktop>
```

The application is a jar file that can be run from the command line as presented in the above picture.

The user has to provide the full path for the input and output path if they are not in the same directory as the application.

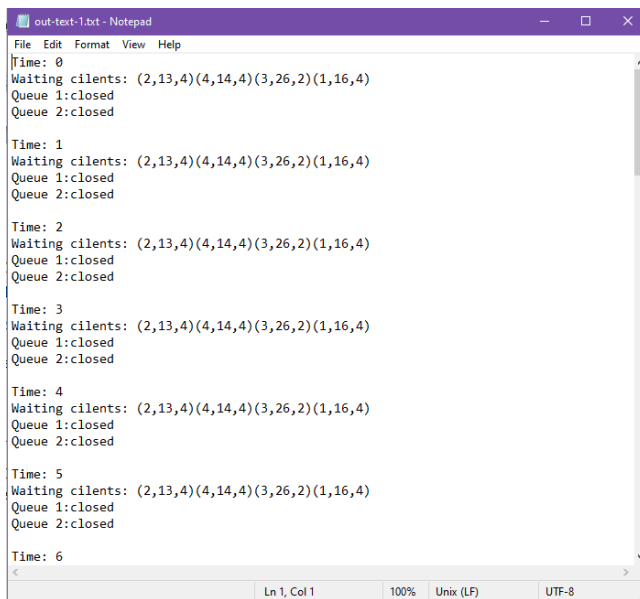
Command for running the application (*):

Java -jar Assigment2.jar input_file_name output_file_name

(*) nu am reusit sa salvez jar-ul cu denumirea ceruta

5. Results

The application is able to compute the average waiting time and after each second it will update both the terminal, as seen in the picture above, and the output file that's given as a parameter in the command line, by using multithreading and exception handlers.



```
File Edit Format View Help
Time: 0
Waiting cilents: (2,13,4)(4,14,4)(3,26,2)(1,16,4)
Queue 1:closed
Queue 2:closed

Time: 1
Waiting cilents: (2,13,4)(4,14,4)(3,26,2)(1,16,4)
Queue 1:closed
Queue 2:closed

Time: 2
Waiting cilents: (2,13,4)(4,14,4)(3,26,2)(1,16,4)
Queue 1:closed
Queue 2:closed

Time: 3
Waiting cilents: (2,13,4)(4,14,4)(3,26,2)(1,16,4)
Queue 1:closed
Queue 2:closed

Time: 4
Waiting cilents: (2,13,4)(4,14,4)(3,26,2)(1,16,4)
Queue 1:closed
Queue 2:closed

Time: 5
Waiting cilents: (2,13,4)(4,14,4)(3,26,2)(1,16,4)
Queue 1:closed
Queue 2:closed

Time: 6
```

6. Conclusions and Possible Improvements

This assignment was a good way to familiarize with threads and synchronization and understand them better, as it was my first application in which I had to use asynchronous programming and multithreading in order to achieve the desired results. Through this assignment I also learned how to generate a .jar file.

Related to the improvements that can be brought to this application, one would be a better strategy so that the application uses encapsulation in all classes and also based on the input parameters to generate more statistics that could be useful to someone who might use this type of application.

7. Bibliography

<https://www.geeksforgeeks.org/>

<https://www.javacodegeeks.com/2014/08/java-concurrency-tutorial-atomicity-and-race-conditions.html>

<http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html>
<https://stackoverflow.com/>