Tudor Todea-Stefan
Group 30424

Programming Techniques
March 2020

Queues Simulator

1. Purpose of the laboratory work

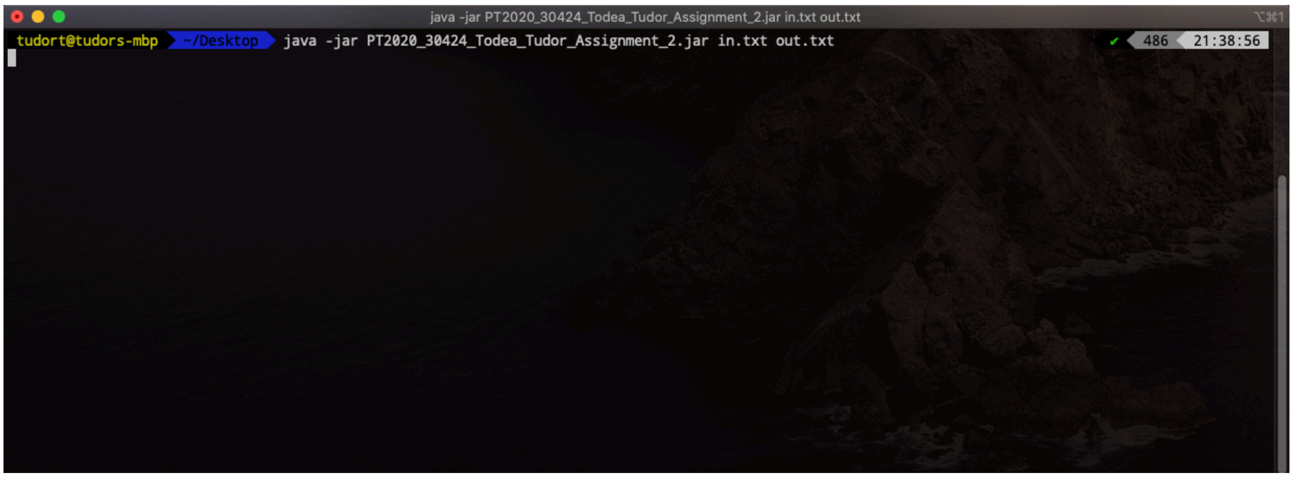
We have designed an application that simulates the behaviour of a certain number of queues and a number of clients, characterised by certain attributes, that will utilise those queues during our simulation. To achieve this purpose we utilise the concept of concurrency and make use of the Java Threads to handle the whole simulation and the individual queues.

In order to build the application we created four classes: **Task**, **Server**, **Scheduler** and **SimulationManager**. The decisions taken regarding the behaviour of these classes will be presented in the 3rd chapter of this documentation, while their implementation will be discussed in the 4th chapter.

2. Analysis, modelling and use-case scenarios

The project must simulate the behaviour of N number of clients that will pass through Q possible queues during a time interval tLimit. We will, as a result, use two files for our simulation: an input file containing the constraints, and an output file in which we will print the state of the queues and the clients at every timestamp and at the end compute and print the average waiting time of our clients. These average waiting time is computed for all the clients that are placed in a queue, regardless of wether they have been processed or not during those tLimit seconds of simulation. To simplify the work that a user must do in order to get the program running, we placed our code in a jar file.

Our application is supposed to be run from the command line using a .jar file. An example of this can be seen in the picture below:



```
java -jar PT2020_30424_Todea_Tudor_Assignment_2.jar in.txt out.txt
tudort@tudors-mbp  ~/Desktop  java -jar PT2020_30424_Todea_Tudor_Assignment_2.jar in.txt out.txt     ✔  486  21:38:56
```

The main syntax is: "java -jar projectJarName,jar inputFile.txt outputFile.txt". The input file must exist and obey the following format:

- N -the number of clients
- Q -the number of queues
- tLimit -the time limit of the simulation
- tArrivalMin, tArrivalMax - the minimum and maximum possible arrival times for our generated clients
- tProcessingMin, tProcessingMax - the minimum and maximum possible processing times for our generated clients

An example for this input format would be the following:

```
4
2
60
2,30
2,4
```

Use Case: Queues Simulator
Main Success Scenario:

1. The user writes the desired input parameters in the input text file
2. The user inputs the correct path and name for the desired .jar file and a valid path to an existing input file
3. The application starts running
4. The end of the application is marked by the apparition of a new line in the command line, meaning that the previous command had been executed successfully
5. An output file with the specified name has been created at the desired path, or, if the file already exists, its contents have been replaced with the ones that resulted from our simulation

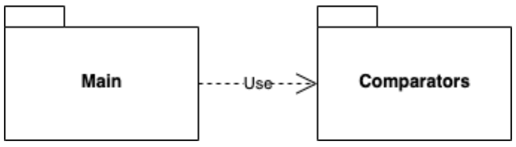
Use Case: Queues Simulator
Alternative Sequences:

1. Incorrect syntax.
2. Invalid path for the .jar file or the input file (the path is taken relative to the selected directory in the terminal)
   -these two scenarios will lead to the 2nd step of the successful scenario
3. Invalid format of the input file, which will lead to undesired simulation parameters
   -this will lead to the first step of the successful scenario

3. Development

1. Package Diagram

Through the development of this assignment we had to split it into two packages: the **Main** package that contains the four main classes of our project and the **Comparators** package that contains two comparator classes that are used to sort our queues either by waiting time, which is useful when we dispatch a new task to the queue with the lowest waiting time, and by ID, which is useful when keeping the order of the queues when printing our result in the output file.



2. Class Diagram

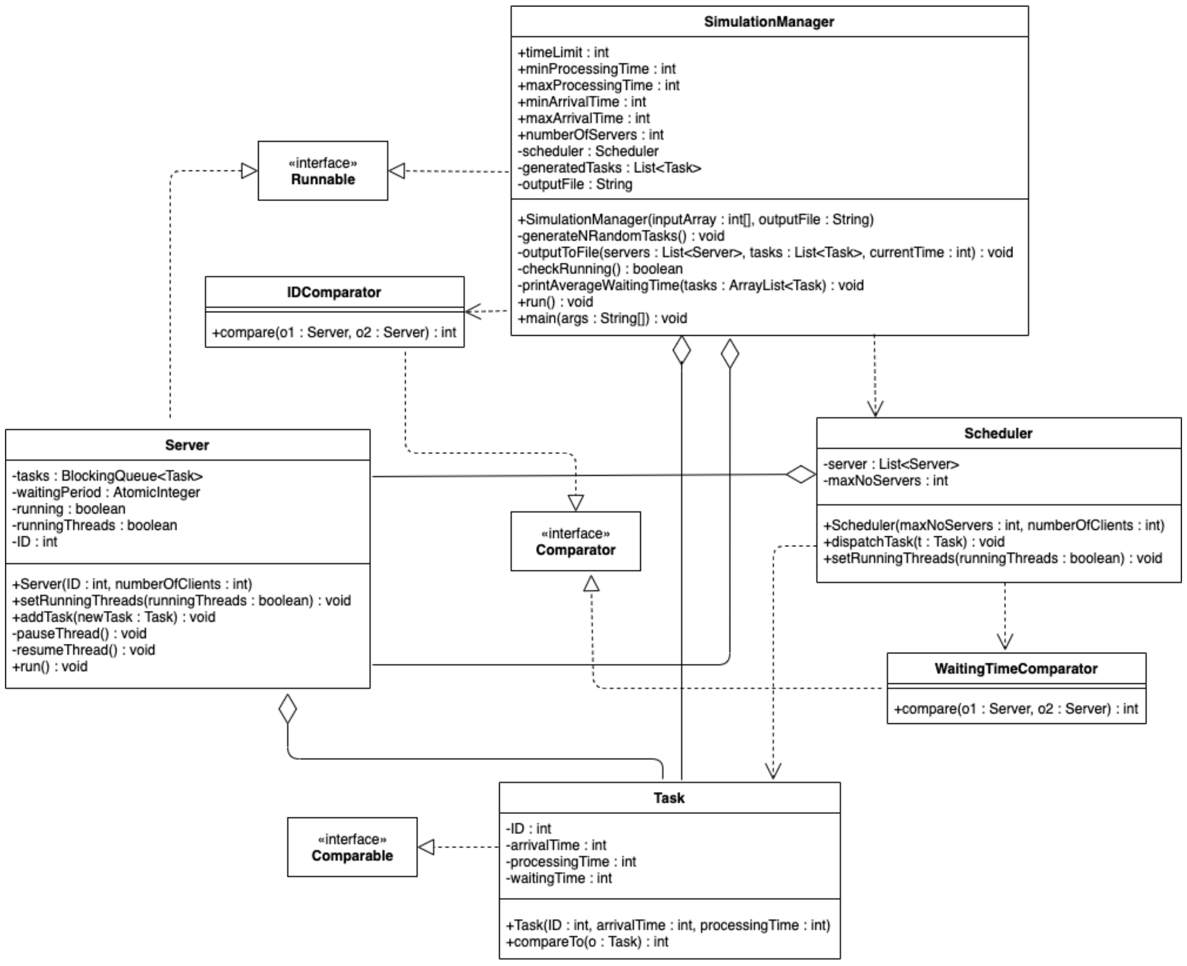
During the development the main project has been split into four classes:

Task - represents the client entity and contains the attributes necessary for processing a client.

Server - represents the queue entity and contains its necessary attributes as well as methods necessary for the processing of the received tasks.

Scheduler - provides a link between the SimulationManager and the Server class. It is used for manipulating the Server objects as it is required by the SimulationManager.

SimulationManager - the main class of the project. It handles the main thread of the application, deals with the input and the output as well as with the management of the Servers and Tasks through the Scheduler attribute object.


SimulationManager
+timeLimit : int
+minProcessingTime : int
+maxProcessingTime : int
+minArrivalTime : int
+maxArrivalTime : int
+numberOfServers : int
-scheduler : Scheduler
-generatedTasks : List<Task>
-outputFile : String
+SimulationManager(inputArray : int[], outputFile : String)
-generateNRandomTasks() : void
-outputToFile(servers : List<Server>, tasks : List<Task>, currentTime : int) : void
-checkRunning() : boolean
-printAverageWaitingTime(tasks : ArrayList<Task) : void
+run() : void
+main(args : String[]) : void
«interface» Runnable
IDComparator
+compare(o1 : Server, o2 : Server) : int
Server
-tasks : BlockingQueue<Task>
-waitingPeriod : AtomicInteger
-running : boolean
-runningThreads : boolean
-ID : int
+Server(ID : int, numberOfClients : int)
+setRunningThreads(runningThreads : boolean) : void
+addTask(newTask : Task) : void
-pauseThread() : void
-resumeThread() : void
+run() : void
«interface» Comparator
Scheduler
-server : List<Server>
-maxNoServers : int
+Scheduler(maxNoServers : int, numberOfClients : int)
+dispatchTask(t : Task) : void
+setRunningThreads(runningThreads : boolean) : void
WaitingTimeComparator
+compare(o1 : Server, o2 : Server) : int
«interface» Comparable
Task
-ID : int
-arrivalTime : int
-processingTime : int
-waitingTime : int
+Task(ID : int, arrivalTime : int, processingTime : int)
+compareTo(o : Task) : int

4. Implementation

1. Task Class

The **Task** class has the following main attributes: *ID, arrivalTime, processingTime* and *waitingTime*. The *ID* is used as a unique key, and unique identifier for each client. The *arrivalTime* and the *processingTime* are generated randomly from an interval provided in the input file. The former represent the time at which the customer is supposed to arrive, while the latter represents the time that our customer will spend at the front of the queue. The *waitingTime* attribute represents the amount of time our client spend from the moment he arrives until he is processed and exits the queue. It can also be written as *processingTime* + the *waitingPeriod* of the queue before our client enters it.

2. Server Class

The **Server** class represents the queue manager. It handles the processing of the tasks that are dispatched to the queue by the **Scheduler**. The attributes of this class are:
- *ID,* which is a unique identifier for each queue
- *tasks*, the list of clients that are being processed by our Server

- *waitingPeriod*, which is used to store the sum of the processing times of all the *tasks.* It is an AtomicInteger since its value is modified inside the run() method of the thread. By defining it as an AtomicInteger we prevent possible problems that may arise from the use of concurrency
- *running*, a boolean variable that pauses the Server when there are no clients in the queue and restarts it when someone enters it. Initially, the queue is closed, this is why the variable is initialised as false.
- *runningThreads*, is a flag that, when triggered, stops the thread (the queue) permanently.

The main methods of this class are:

addTask(Task *newTask):*

This method adds a new Task object to the *tasks* of the Server. It then increments the *waitingPeriod* by the *processingTime* of the newly added task. Finally, it sets the newly incremented *waitingPeriod* as the *waitingTime* of the added client.

run():

This method is the one that is executed by the Server thread created in the Scheduler. This thread is "alive" as long as the *runningThreads* flag is true. While there are no clients in the queue, it is paused until a new task is provided for our Server object. As long as there are tasks for the queue to process we get the task at the head of the queue and decrement its *processingTime* by 1. If the remaining *processingTime* is already 1, the client is removed from the queue since he is supposed to be gone by the next iteration. The total *waitingPeriod* of the queue is decremented by 1 and the thread sleeps for one second. This process continues until there are no clients in the queue (the *running* flag is set to false) or the *runningThreads* flag is triggered and the whole thread stops altogether.

3. Scheduler Class

The **Scheduler** class is used as a manager of the servers, and the link between the ScheduleManager and Server classes. It contains *servers* as the main attribute, which is the list composed of all the queues. Inside its constructor there are *maxNoServers* objects initialised and their respective threads are started. The threads are started individually. A ThreadPool Executor could also have been used to handle a fixed number number of threads since we already know their number. Aside from that we also have a couple of important methods here:

dispatchTask(Task t):

This method sorts the list of queues in ascending order by their *waitingPeriod.* After that it sends the task *t* to the one with the lowest *waitingPeriod.*

setRunningThreads(boolean runningThreads):

This method is used to trigger the *runningThreads* flag of each Server object. In reality it is used only two times: one when initialising the Server objects in the constructor of Scheduler, when we set the flag to **true** and the second time after the main thread in the SimulationManager stops and we also stop the queues by setting the flag to **false**.

4. SimulationManager Class

The main attributes of this class are:
- *timeLimit, minProcessingTime, maxProcessingTime, minArrivalTime, maxArrivalTime, numberOfServers, numberOfClients* - these attributes are read from the input file and are the simulation parameters by which we will generate the clients and run the application
-*scheduler* - the Scheduler object responsible for handling the instructions given by the SimulationManager and directing them towards the servers
- *generatedTasks* - the randomly generated clients using the parameters provided by the input file
- *outputFile* - the name of the file as a String

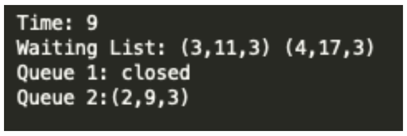
The SimulationManager is the main class of our project. We start off the main method by reading the information from the input file. We then initialise the attributes of our class inside the constructor, we call the **generateNRandomTasks**() method to generate the clients, then we initialise the scheduler object. In the main method, after the initialisation of the SimulationManager we start its respective thread. Before discussing the **run**() method of this thread, we will discuss about the other important methods of this class:

generateNRandomTasks():

This method will generate *numberOfClients* Task objects within the following parameters: their arrival time is selected at random between *minArrivalTime* and *maxArrivalTime*. The same is done for the processing time of each client. After that we sort the clients by their arrival time, and then assign each of them an *ID* in ascending order.

outputToFile():

This method is used to output the state of the simulation at each moment of time. A simulation snippet may look like this:



```
Time: 9
Waiting List: (3,11,3) (4,17,3)
Queue 1: closed
Queue 2:(2,9,3)
```

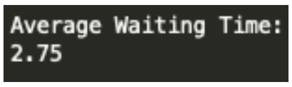
Firstly, it prints the current time of the simulation. Then the clients that have not yet arrived are displayed in the Waiting List. Each client is represented by a a collection of three numbers: their *ID, arrivalTime* and *processingTime*. The clients are separated from each other by a blank space. Underneath that, we have the state of every queue. In our example, the first queue is closed, while there is a client being processed at the head of the second queue. The first queue finished processing a client beforehand and now finds itself closed since there is no client that has yet arrived at the current time (we can see that the next client arrives only at time 11). There can be multiple queues and a multitude of clients present at each queue. These constraints are specified in the input file.

checkRunning():

We utilise this method to check if we can stop the simulation before reaching the time limit. That means that when all of our queues are closed and there are no clients in the Waiting List, which in turn means that there are no more clients that will be processed during the simulation, we can stop the simulation early.

printAverageWaitingTime():

Inside this method we will compute the average time in the following way: we take each client that has ever been removed from the Waiting List, add its *waitingTime* to a total and divide that total by the *numberOfClients*. We then print the result after our main thread has stopped.



```
Average Waiting Time:
2.75
```

run():

This is the method called by the start of the SimulationManager thread. It runs until the flag *running* is set to false. Firstly, it checks if we have a client arriving at the current simulation time. If it does, we call the **dispatchTask**() method of our Scheduler object and send that task to the proper Server. We then add the removed client to a list that will be used later to compute the average waiting time. We then output the current state of our simulation to the output file, and then increment the simulation time by 1 and have our thread sleep for one second. We then check if we have reached the time limit or if the **checkRunning**() method returns false. If this happens, we set the *running* flag to false and the thread stops. After this main thread is stopped, we again use the *schedule* attribute to call the **setRunningThreads**() method from the Scheduler and stop the Server threads. After that we print the average waiting time and our application ends entirely.

5. IDComparator and WaitingTimeComparator Classes:

These two classes implement the Comparator interface and are used to sort the Server objects by *ID* or by *waitingPeriod* depending on our needs.

5. Results

As stated in Chapter 2, to obtain a valid result the input file must be of the specified format and have valid data. That means that the arrival time or the processing time of the clients should not have the possibility of being higher than the simulation time since that would not make sense from a practical standpoint.

The results of the simulations can be seen in the output file specified by the user. The resulted file should contain information of every simulation step and should end with the computed average waiting time. The results can be observed in the output files provided with the application. Aside from the three required tests, we also added "in-test-4.txt" and "in-test-5.txt" that can be used to test the application. Their result is also attached to this project.

6. Conclusions

While completing this assignment we have gained knowledge of the Java Multithreading mechanism and its use, as well as the benefits of concurrent programming. As a potential further development we could possibly add more threads to deal with each individual subtask.

7. Bibliography

Assignment 2 Lab Presentation - application structure and code skeleton

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html

http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html