MINISTERUL EDUCAȚIEI NAȚIONALE
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Queues Simulator

Fundamental Programming Techniques

Horvath Andrea Anett

Faculty of Automation and Computer Science
Computer Science Department
2'nd year, Group 30424

Contents

I. Objectives of the proposed laboratory work

The main objective of the proposed laboratory work was to design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients' waiting time.

As secondary objectives, the following can be considered:

- developing the application in accordance with the object oriented programming principles
- getting familiar with random generation of data
- drawing the UML diagrams and writing the code based on them (forward engineering)
- determining all the use-cases
- designing the simulation according to real life principles
- working with multithreading (one thread per queue)
- working with files for reading and writing data
- running the project as a .jar file
- choosing the appropriate synchronized data structures to assure thread safety
- developing the queues as dynamical structures

II. Analysis of the laboratory work, modeling, scenarios and use-cases

Analysis

The Queues Simulator will be implemented to simulate a real life scenario of a service center, i.e. a market. The simulation will be scaled down to seconds, and the state of the system will be recorded each second.

The application must be developed to have a .jar file containing the project configured to run from the terminal and read 2 parameters: the name of the input file and the name of the output file. The input file must exist and must have a readable structure for the project in order to simulate successfully the queue and output the results gradually in the output file which if does not exist, it will be created.

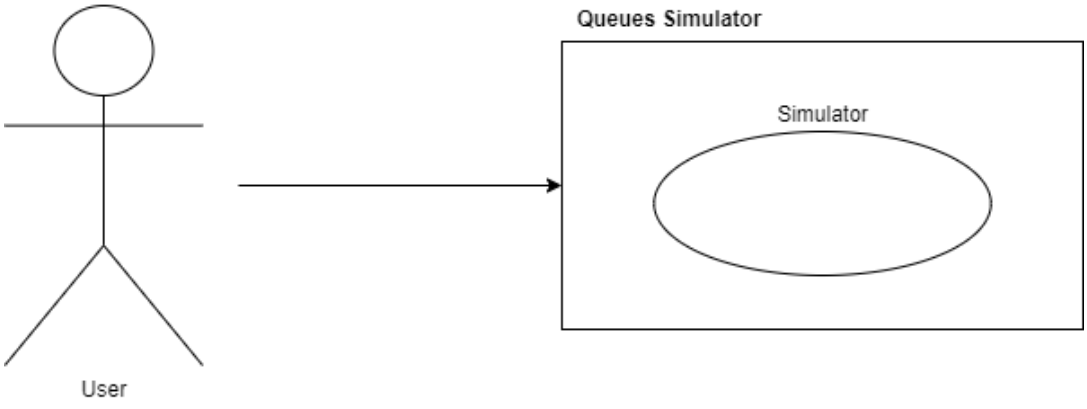
Queues simulation is done to model real world domains. The purpose of this structure in the case of this project is to provide a place for a "client" to wait before receiving a "service". The management of the queue-based system is interested in minimizing the time amount their "clients" are waiting in queues before they are served.

Modeling

The application simulates, according to the input file, for a period of time *t-simulation*, a series of *N* clients arriving for service, entering *Q* queues, waiting, being served and finally leaving the queue. All clients are generated when the simulation is started, and are characterized by three parameters: *ID* (a number between 1 and *N*), *t-arrival* (simulation time when they are ready to go to the queue, i.e. time when the client finished shopping) and *t-service* (time interval or duration needed to serve the client by the cashier, i.e. waiting time when the client is in front of the queue). The application tracks the total time spend by every customer in the queues and computes the average waiting time. Each client is added to the queue with minimum waiting time when its arrival time is equal to the simulation time (*t-arrival* = *t-simulation*).

Scenarios and use-cases

The only use-cases is sketched in the next diagram and detailed below.



In the following use-case the primary actor is considered to be the user.

Use-case: *Queue simulation*

Main success scenario:

1. The user opens Command Prompt.
2. The user changes the current directory with the directory in which the project is located.
3. The user introduces the appropriate command with the right arguments (existent .txt files) to run the application.
4. The application verifies the existence of the files.
5. The output file displays a log of the execution of the simulation and the average waiting time of the clients.

Alternative sequence A:

1. The user opens Command Prompt.
2. The user changes the current directory with the directory in which the project is located.
3. The user introduces the appropriate command with an existing input .txt file and an inexistent output .txt file as arguments to run the application.
4. The application verifies the existence of the input file and creates an output file with the name given in the argument in the project's directory.
5. The output file displays a log of the execution of the simulation and the average waiting time of the clients.

Alternative sequence B:

1. The user opens Command Prompt.
2. The user changes the current directory with the directory in which the project is located.
3. The user introduces the appropriate command with an non-existing input .txt file and output .txt file as arguments to run the application.
4. An exception is thrown and the program exits.

Alternative sequence C:

1. The user opens Command Prompt.
2. The user changes the current directory with the directory in which the project is located.
3. The user introduces the appropriate command with the right arguments (existent .txt files) to run the application.
4. A system error occurs and the files cannot be opened.
5. An exception is thrown and the program exits.

III. Design

1. Development approach

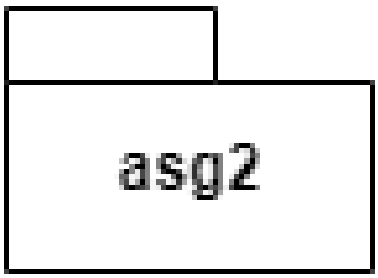
The queues simulator application is all included in one package, the asg2 package. Since the object oriented paradigm imposed in case of this task only four classes which did not needed a further classification into packages, only the asg2 package was used.

Thus, the only existing package includes the following classes:

- Client: contains the model of a person in the queues simulation process;
- Server: contains the model of the queue with the clients waiting in it;
- Scheduler: coordinates all the elements in the simulation (the clients and the servers);
- SimulationManager: initiates and coordinates the process of the simulation, also generates the clients randomly.

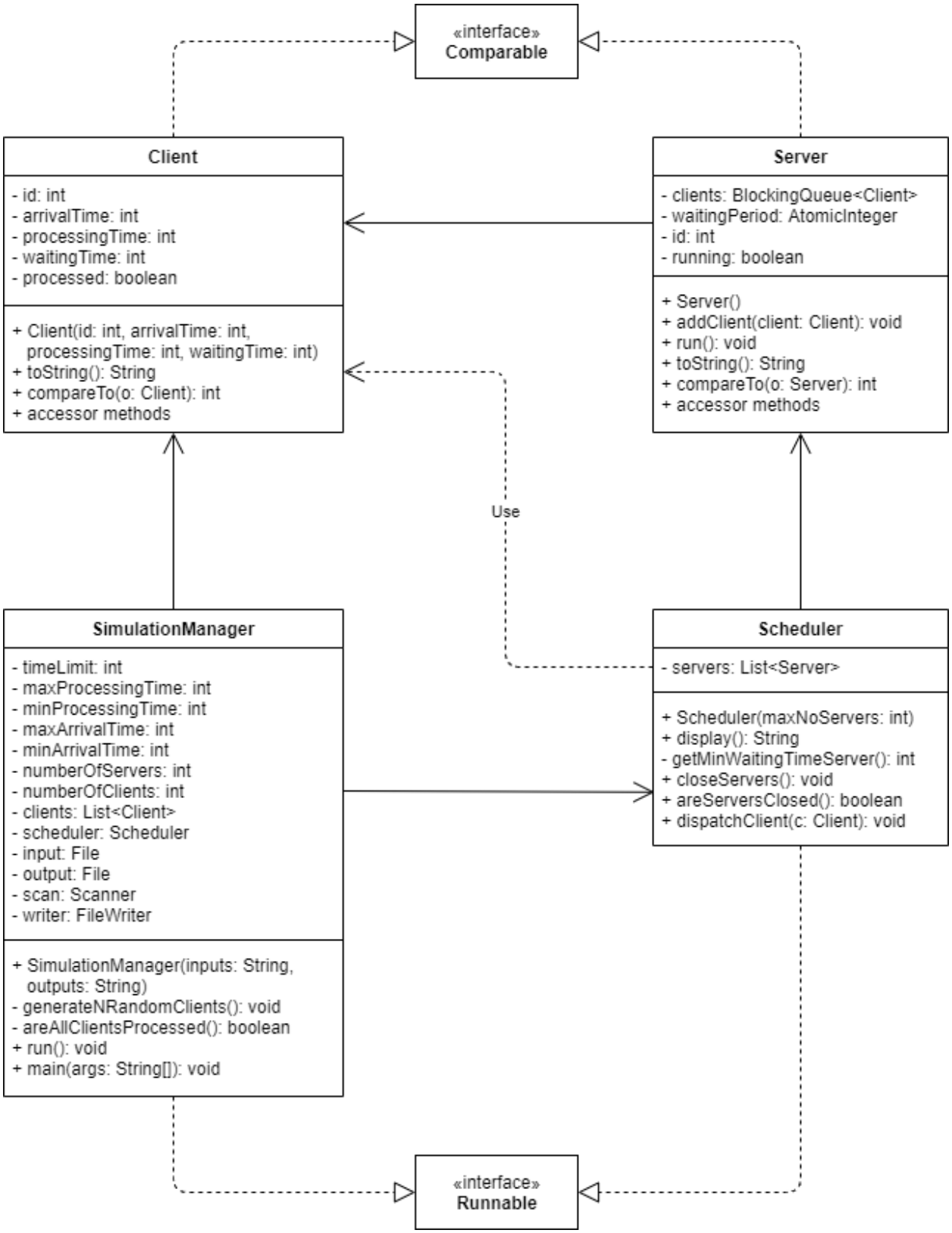
2. Structure and UML Diagrams

2.a. Package Diagram



The content of the package is shown in the class diagram.

2.b. Class Diagram


«interface»
Comparable
Client
- id: int
- arrivalTime: int
- processingTime: int
- waitingTime: int
- processed: boolean
+ Client(id: int, arrivalTime: int, processingTime: int, waitingTime: int)
+ toString(): String
+ compareTo(o: Client): int
+ accessor methods
Server
- clients: BlockingQueue<Client>
- waitingPeriod: AtomicInteger
- id: int
- running: boolean
+ Server()
+ addClient(client: Client): void
+ run(): void
+ toString(): String
+ compareTo(o: Server): int
+ accessor methods
Use
SimulationManager
- timeLimit: int
- maxProcessingTime: int
- minProcessingTime: int
- maxArrivalTime: int
- minArrivalTime: int
- numberOfServers: int
- numberOfClients: int
- clients: List<Client>
- scheduler: Scheduler
- input: File
- output: File
- scan: Scanner
- writer: FileWriter
+ SimulationManager(inputs: String, outputs: String)
- generateNRandomClients(): void
- areAllClientsProcessed(): boolean
+ run(): void
+ main(args: String[]): void
Scheduler
- servers: List<Server>
+ Scheduler(maxNoServers: int)
+ display(): String
- getMinWaitingTimeServer(): int
+ closeServers(): void
+ areServersClosed(): boolean
+ dispatchClient(c: Client): void
«interface»
Runnable

IV. Implementation

Default package

a. Class Client

Instance variables: *id*: int, *arrivalTime*: int, *processingTime*: int, *waitingTime*: int, *processed*: boolean.

The variables represent the attributes of a client which have either fixed values got from the randomly generated data (*id*, *arrivalTime*, *processingTime*) or variable values which are to be computed during the simulation (*waitingTime*, *processed*).

The constructor `public Client(int id, int arrivalTime, int processingTime) {...}` initializes the *id*, the *arrivalTime* and the *processingTime* with the values given as parameters and the *waitingTime* and *processed* with 0 and *false* respectively because initially a client will always has this values.

The class also contains getters and setters in order to have access to the private instance variables.

The method `public String toString() {...}` is used to display a client according to his attributes in the next form: (id, arrivalTime, processingTime).

The class implements the Comparable interface, and contains the next method which has the purpose of ordering the clients in ascending order of their *arrivalTime*s.

```
@Override
    public int compareTo(Client o) {
        return this.arrivalTime - o.getArrivalTime();
    }
```

b. Class Server

Instance variables: *clients*: BlockingQueue<Client>, *waitingPeriod*: AtomicInteger, *id*: int, *running*: boolean.

The variable *clients* represents the list of clients waiting at the server, *waitingPeriod* is the sum of the processing times of the clients in the queue, the *id* is only an identifier, and *running* represents a flag indicating whether the server is opened or closed.

The constructor in this class initializes the *waitingPeriod* and the *running* with 0 and false, this being the initial conditions of the server.

The class contains getters and setters in order to have access to the private instance variables.

The method `public void addClient(Client client) {...}` is used to add a new client in the queue of the server and computes the *waitingTime* of the client and the new *waitingPeriod* for the server.

The class implements the Runnable interface so it overrides the method `public void run() {...}`. The method is executed as long as the *running* variable is *true* in a try - catch block (used in case an exception for thread interruption is thrown). The method verifies each second if the server's list of clients is empty or not. If yes, the

thread will go to sleep for another second, meaning the server will be closed until a client appears. If there is at least one client in the queue, the client to be processed (the head of the queue) is taken and its *processingTime* together with the *waitingPeriod* of the server will be decremented and the thread is set to sleep for another second. This is done until the *processingTime* reaches 0, when the *clients* is dequeued and the thread sleeps for one more second. Then the next client is taken for being processed or the thread waits (the server is closed) until a new client appears.

The method `public String toString() {...}` is used to display the list of clients waiting at the server. It uses the method with the same name from the class Client.

The class implements the Comparable interface, and contains the next method which has the purpose of ordering the servers in ascending order of their *id*.

```
@Override
    public int compareTo(Server o) {
        return this.id - o.getId();
    }
```

c. Class Scheduler

Instance variables: *servers*: List<Server>.

The variable *servers* represents the list of servers included in the simulation.

The constructor `public Scheduler(int maxNoServers) {...}` gets that integer as argument as the number of servers to be created. Each created server gets an *id* (numbers in ascending order in the interval [1, *maxNoServers*]), and is added in the list. Also a thread is started for each of the servers inserted in *servers*.

The method `public String display() {...}` has the purpose of bringing the model of the server into a printable form. It returns the *id* of the queue and the list of the clients or the key word "Closed" in case the list is empty. An example of output could be: " Queue 1: (3,14,2) / Queue 2: Closed".

The method `private int getMinWaitingTimeServer() {...}` is used to get the server with the minimum *waitingPeriod*, in order to put the newly arrived client there. The method returns the id of that specific server. In case there are two or more servers with the same minimum *waitingPeriod*, the method returns the first one in order of their *id*.

The method `public void closeServers() {...}` sets all the servers *running* instance variable to *false*; is used for the moment when the simulation has to stop.

The method `public boolean areServersClosed() {...}` returns a boolean value indicating whether the servers are all closed or not.

The last method in this class is `public void dispatchClient(Client c) {...}` and places the newly arrived client to the server with the minimum *waitingPeriod*.

d. Class SimulationManager

Instance variables: *timeLimit*: int, *minProcessingTime*: int, *maxProcessingTime*: int, *minArrivalTime*: int, *maxArrivalTime*: int, *numberOfServers*: int, *numberOfClients*: int, *clients*: List<Client>, *scheduler*: Scheduler, *input*: File, *output*: File, *scan*: Scanner, *writer*: FileWriter.

The first 6 instance variables are the input data which serves as parameters for the randomly generated data, the *clients* and the *scheduler* represent the generated clients and the servers. The *input* and the *output* are the files for reading and displaying the results, and the *scan* and the *writer* are the tools for reading form and writing in the files.

The constructor in this class gets as arguments two strings corresponding to the I/O files. The two files are opened and if the output file does not exist then it is created. The scanner is initialized with the *input* file and the data from it is read in the integer variables. The scanner is than closed and the *scheduler*'s constructor is called for the read *numberOfServer.* Then the clients are generated with *generateNRandomClients* method.

The method `private void generateNRandomClients() {...}` is used to generate a number of *numberOfClients* clients with the *id*s taken in order and the *arrivalTime* and the *processingTime* random numbers in the ranges [*minArrivalTime*, *maxArrivalTime*] and [*minProcessingTime*, *maxProcessingTime*] respectively. The clients are then sorted in increasing order of their *arrivalTime*.

The method `private boolean areAllClientsProcessed() {...}` returns an answer to the question if all the clients have been processed or not. It is used to determine if the simulation can end. It checks the instance variable *processed* of each client in the list.

The class implements the Runnable interface so it overrides the method `public void run() {...}`. The method is written in a try - catch block in order to throw InterruptedException or IOException in case the thread is interrupted or the files cannot be opened. Two integer variables are declared here: *currentTime* and *totalWaitingTime*, the first for counting the time until it reaches *timeLimit* and the second for computing the average waiting time. The following are done in a while loop having as condition the *currentTime* being smaller or equal to the *timeLimit*. Thus, the program checks if all the servers are closed and if all the clients have been processed, and in a positive case, the program exits the while loop. Otherwise, the *currentTime* is displayed and the clients are checked if they are done shopping so they can be dispatched to a server and mark them as processed, also the *waitingPeriod* of that server is updated. Then, the clients in the waiting list are printed and also the queues with their clients. In the end, the *currentTime* is incremented and the thread is set to sleep for one second. After the loop exits, the servers are closed, the average waiting time is computed and printed, and the writer is closed.

The last method here is the main method `public static void main(String[] args) {...}` , its arguments being the input and the output file. An object of the SimulationManager class is initialized and a thread is started for it.

V. Results

For testing the correctness of the simulation, the program was tested on the next three input files, which are included in the project's directory.

in-test-1.txt	in-test-2.txt	in-test-3.txt
4	50	1000
2	5	20
60	60	200
2,30	2,40	10,100
2,4	1,7	3,9

An obtained output when running the code for the first input file was the following:

```
Time: 0
Waiting clients:
(2,4,3) (1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 1
Waiting clients:
(2,4,3) (1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 2
Waiting clients:
(2,4,3) (1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 3
Waiting clients:
(2,4,3) (1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 4
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: (2,4,3)
Queue 2: Closed

Time: 5
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: (2,4,2)
Queue 2: Closed

Time: 6
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: (2,4,1)
Queue 2: Closed

Time: 7
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 8
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 9
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 10
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 11
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 12
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 13
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed
```

```

Time: 14
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 15
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 16
Waiting clients:
(1,17,4) (3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 17
Waiting clients:
(3,22,4) (0,30,4)
Queue 1: (1,17,4)
Queue 2: Closed

Time: 18
Waiting clients:
(3,22,4) (0,30,4)
Queue 1: (1,17,3)
Queue 2: Closed

Time: 19
Waiting clients:
(3,22,4) (0,30,4)
Queue 1: (1,17,2)
Queue 2: Closed

Time: 20
Waiting clients:
(3,22,4) (0,30,4)
Queue 1: (1,17,1)
Queue 2: Closed

Time: 21
Waiting clients:
(3,22,4) (0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 22
Waiting clients:
(0,30,4)
Queue 1: (3,22,4)
Queue 2: Closed

Time: 23
Waiting clients:
(0,30,4)
Queue 1: (3,22,3)
Queue 2: Closed

Time: 24
Waiting clients:
(0,30,4)
Queue 1: (3,22,2)
Queue 2: Closed

Time: 25
Waiting clients:
(0,30,4)
Queue 1: (3,22,1)
Queue 2: Closed

Time: 26
Waiting clients:
(0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 27
Waiting clients:
(0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 28
Waiting clients:
(0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 29
Waiting clients:
(0,30,4)
Queue 1: Closed
Queue 2: Closed

Time: 30
Waiting clients:

Queue 1: (0,30,4)
Queue 2: Closed

Time: 31
Waiting clients:

Queue 1: (0,30,3)
Queue 2: Closed

Time: 32
Waiting clients:

Queue 1: (0,30,2)
Queue 2: Closed

Time: 33
Waiting clients:

Queue 1: (0,30,1)
Queue 2: Closed


Average waiting time: 3.75
```

V. Conclusions

The Queues simulator fulfils all the requirements of the laboratory work.

For future improvements the following can be considered:

- Graphic interface for entering the input data;
- Verification of the input data;
- Adding a new strategy for determining the shortest queue according to the number of clients.

VI. Bibliography

1. https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

2. https://www.tutorialspoint.com/java/util/timer_schedule_period.htm

3. https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html

4. http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/

5. https://www.w3schools.com/java/java_files_read.asp

6. https://www.w3schools.com/java/java_files_create.asp