Queue simulator

1. The assignment's objective

The main objective of the assignment is to develop a queue simulator meant to optimize the waiting time for each client. This is done by following a particular pattern, pattern that ensures that certain immediate steps (secondary objectives) are performed: we have N clients and Q queues; the clients all have an arrival time and a service time(the last one meaning how long they are going to stay in front of the queue) so they need to be ordered by these parameters, in order for them to be efficiently distributed in the Q queues, in that way minimizing the time that a client has to wait at a queue. The clients are generated randomly, depending on different parameters such as: minimum arrival time, maximum arrival time, minimum service time, maximum service time. Each of these parameters is read from a file at the launch of the application, as well as others such as maximum simulation time (the maximum number of seconds for which the queue simulator must work) or the number of clients/queues presented before. Clients are distinguished by one another through their ID (These aspects will be elaborated in Chapter 4). Another secondary objective, also for ensuring the efficiency of distributing the clients into queues, is computing, for each queue, the waiting time and the number of clients waiting. In that way, depending on the strategy chosen (i.e. the clients will be distributed to the queue with the shortest waiting time or to the one with the smallest number of clients) it is assured that clients will be correctly distributed. Also, computing the average time of the waiting at the queues is another secondary objective. This is done in the following manner: for each queue, after a client is processed, the number of total clients served will be incremented, and also at the waiting time the client's total time at the queue will be added (this will be elaborated in chapter 4). Last, but not least, the display of each of the queues' status will be done at each second, in a file, meaning that the person that is at the front of the queue will have the service time decremented every second, until it is processed entirely.

2. Problem analysis, modelling, scenarios, use cases

The queue simulator will run during a simulation time provided at the run of the application(it will run in real time). In terms of problem analysis, we have to consider the fact that each queue should be open as long as there are clients waiting to be served. Taking that into account, if there are no clients waiting at a queue, it should be closed, it's opening depending on another client being distributed to it. To get a better understanding of this, below will be presented the life cycle of a queue.

- The simulation time will run in the background
- At first the queue will be closed, waiting for a client to be distributed to it.
- When a client is distributed to the queue, the queue opens and starts processing the client.
- The client will be processed for service time seconds, after that he will leave the queue.
- After the client will be processed, if the queue has another client, he will be processed next, if not it will go back to step 2 (it will be closed, waiting for another client).

Also, at each second, in a file, the queue's status will be presented. If it has clients, they will be present at the queue and the first one's service time will be updated (it should decrement to 0 each second), if not the queue will display "closed".

A use case is presented below:

Use case: Queue Simulator

- After the application is run, N clients will be randomly generated. We also have Q queues.
- They are placed in the waiting list.
- In the moment that one's client arrival time equals the simulation time, he is distributed to a queue.
- Queues will process its clients, computing total number of clients and total time.
- Each second, at each queue the status of it will be displayed (it will be written in a file).
- This will be done in a loop until the simulation time equals the maximum simulation time or until all the queues are empty and the waiting clients are empty.

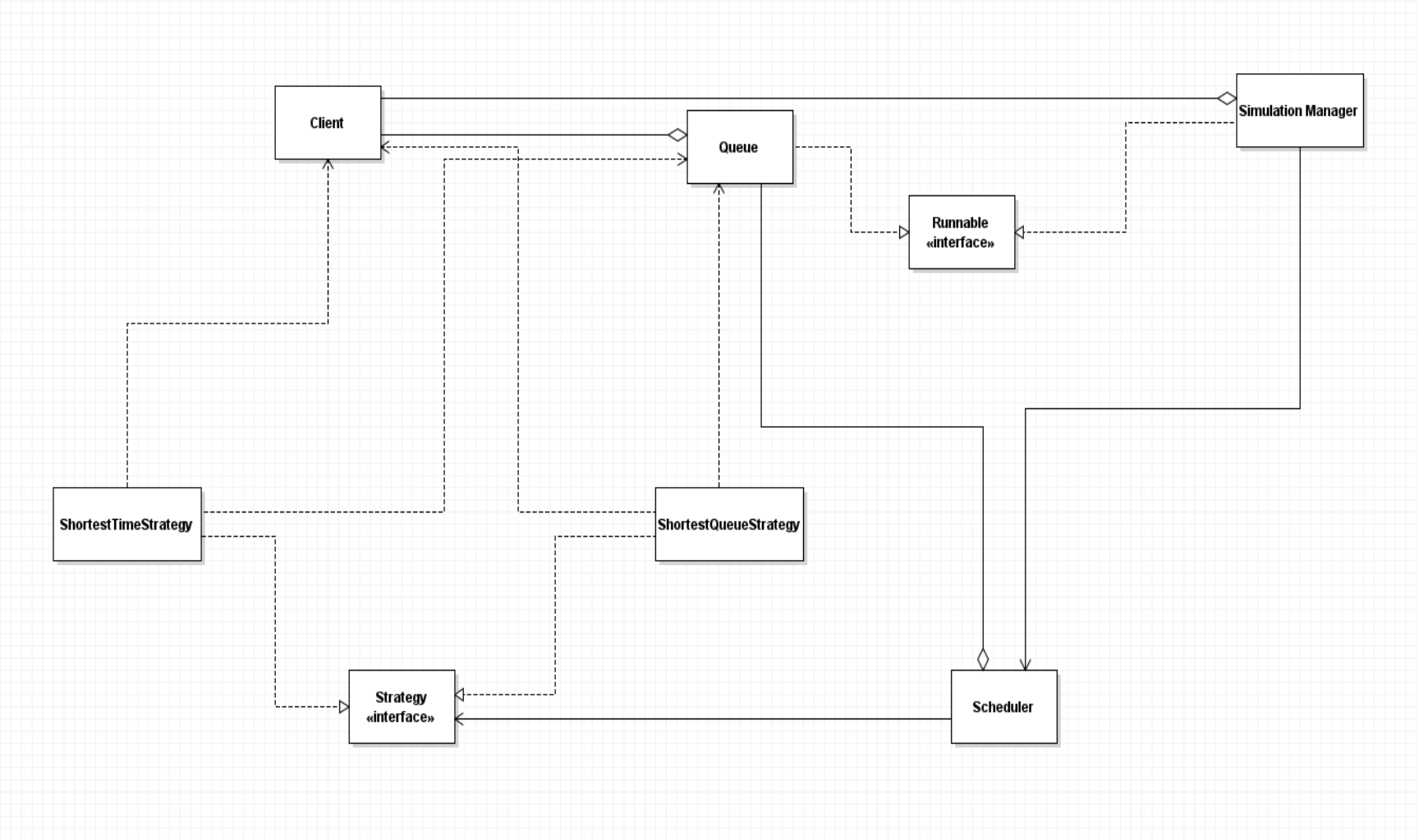
3. Design (design decisions, UML diagrams, data structures, design classes, interfaces, relationships, packages, algorithms, user interface)

Talking about the decision in the OOP design, we have multiple packages: Client, Queue, Schedule, Strategy, Simulation. Also, we have classes that serve different meanings: the Client class is responsible for replicating the client's attributes(it has ID, arrivalTime and serviceTime), the Queue class is the OOP equivalent to a real life queue(it contains a queue of people, and always the first one in the queue is processed), the Schedule class is used for correctly distributing the clients into queues. Also, we have the SimulationManager class that is responsible with always keeping track of the simulation time and with displaying the status of each queue in a file each second. In terms of interfaces, we have the Strategy interface that defines a method to add a client to a Queue that is later implemented by the classes ShortestTimeStrategy and ShortestQueueStrategy that add a client to a queue with smallest waiting time or smallest number of people respectively. What's more, the Queue class implements the Runnable interface, in that way achieving multithreading (we have one thread for one queue => Q threads). SimulationManager also implements Runnable (the Q+1th thread).

In terms of other relationships, between Queue and Client classes there is an aggregation, as it is between Scheduler and Queue (Queue has a BlockingQueue of Clients and Scheduler has a list of Queues). Between the SimulationManager class and Client class there is also an aggregation (the simulation Manager has a List of Waiting clients). What's more, between SimuationManager and Scheduler classes there is an association, being one-to-one. An association relationship is also between Scheduler and Strategy classes. Both ShortestTimeStrategy and ShortestQueueStrategy depend on Client and Queue classes.

In terms of data structures used, in order to assure that the Queue class is thread safe, we have a BlockingQueue, in that way, all the queues can work properly, even though we are dealing with multithreading. Also, we use a LinkedList in SimulationManager in order to store the waiting clients and one in Scheduler class in order to store the queues.

Below will be presented the UML diagram:



4. Implementation

a) The Client class- has as fields the identifier of a client (the ID), the arrival time, service time, waiting time and the total waiting time. The waiting time is computed when a Client is distributed to a queue and the total waiting time is calculated as the sum of the service time plus the waiting time. This last field will be used when calculating the average waiting time: when the client leaves the queue, this total waiting time will be added to the total time of functioning of the queue and the number of clients processed by the queue will increment. This class contains getters and setters for these fields and also contains a method called toPrint() that is used by the SimulationManager class in order to print the clients in the waiting list or present at one of the queues.

b) The Queue class- contains fields such as waitingPeriod or nrOfWaitingClients, used by the Scheduler class in order to know where to distribute a new Client. Also contains fields such as nrOfTotalClients and nrOfTotalTime used for computing the number of clients and the time for which the queue has been open (and as a consequence, are helpful for computing the average waiting time). All of these fields are stored using AtomicInteger, in that way we are able to avoid problems which may appear due to multithreading and we are able to compute for each queue the waiting time and the nrOfTotalClients, nrOfTotalTime separately. We also use a LinkedBlockingQueue in order to store the clients, also for thread safety reasons. The Class also contains a field stored as an AtomicBoolean used to control the start and stop of a thread(isRunning). In terms of methods, the class contains getters and setter for some of the fields but it also contains other methods which are very important. One method is the addClient() method, used for adding a client to a queue, updating its fields(the waiting time and the total waiting time). Since the Queue class implements the Runnable interface, we have the run() method which is the most important method of this class. In the run method(whose while loop is controlled by the Boolean isRunning) it is defined the mechanism of functioning of a Queue: if the LinkedBlockingQueue used to store clients is empty, the thread will be interrupted, otherwise the thread will sleep for approx. 1 second, and will computed the Service time for the person in front(it will decrement it). If the service time of the client is 0, it means that he has been processed and therefore, to the nrOfTotalTime it is added the totalWaitingTime of the client that has been processed, the nrOfTotalClients is incremented and also the client is eliminated from the Queue. The stopThread() method is used for setting the AtomicBoolean isRunning to false. In that way, we are able to stop the Thread. The startThread() method is also crucial because it is used after a Queue has been interrupted (i.e. it has been "paused") and a new client is distributed to it. In that case, the thread should start and this method does exactly so.

c) The Scheduler class' main purpose is to dispatch a Client to the correct Queue, i.e. to the Queue that best fulfills the needs defined by the strategy (smallest waiting time or smallest number ff clients). The Scheduler contains a LinkedList used to store the queues and also a strategy field of type Strategy. The Scheduler contains a defineStrategy() method used to define the strategy by which the clients will be added to the queues(it initializes the object of type Strategy with one of the classes that implement this method). In this project the implementation will be done by the ShortestTimeStrategy() class, meaning that the client will be distributed to the queue with the minimum waiting time. The other strategy is ShortestQueueStrartegy(). The

Scheduler contains another method, dispatchClient(), which calls the strategy's addClient() method. Also, another very important task for the Scheduler is to define a thread for each queue and start it. This will be done in the constructor, where the strategy will also be defined.

d) In the Strategy package, we have the Strategy interface that defines the addClient methd(). The implementations of this method can be found in the ShortestTimeStrategy() and ShortestQueueStrartegy() classes. The methods addClient() of these two classes add clients to the queue with minimum waiting time or with minimum number of clients waiting, respectively.

e) The SimulationManager class- it has many fields that are related to different methods. This Class is used to read the parameter inputs from a file (such as the time limit, minserviceTime, maxServiceTime, minArrivalTime, maxArrivalTime, numberOfQueues, numberOfClients) and they each stored in their respective fields. The reading from a file is done by the method readFile(). It also has a field for its selectionPolicy, which is SHORTEST_TIME in this assignment. Other fields are a scheduler of type Scheduler and a CopyOnWriteArrayList of clients that represents the waiting list. In the constructor of this class, the Scheduler will be defined with the ShortestTimeStrategy, through the SHORTEST_TIME selectionPolicy (when a scheduler is defined, the defineStrategy() method will be called and because of the selection policy parameter => the addClient() method of the ShortestTimeStrategy() class will be chosen). For writing into a file, a new field is declared as is used in the methods blankFile() that acts like an erases for a file (meaning that when we want to run the application we want to write into a blank file so the blankFile() method sort of acts like that) and toPrint() method which is used in the run() method for writing at each second(real time) the status of each queue and the waiting list. We have also other important methods such as generateRandomClients() that genereates N number of clients randomly, depending on certain parameters read from a file(minserviceTime,maxServiceTime, minArrivalTime, maxArrivalTime). It generates the clients and then it adds them to the list of waiting clients. After all the clients are generated, the list will be sorted by arrivalTime and then by serviceTime. This is another important step in improving the average waiting time (it goes by the principle of the priority of the client that comes first and spends the least time in front of the queue). The method computeAverageTime() actually computes the average time of all the queues in the following manner: it iterates through each queue, adding all the number of clients and the times for which each queue was open(had clients). After that, the total time will be divided by the total number of clients processed by the queues. This method will be called only after all the clients have been processed or the simulationTime has passed. Another very important method is stopAllThreads(). It will be called in the same way as the computeAverageTime() method will be called (at the end of the simulation or after all the clients have been processed). This method works the following way: it iterates through each queue, calling for the method stopThread() that will set the Boolean flag isRunning to false and thus, managing to stop every thread. The run() method of the SimulationManager class (it also implements Runnable) works as follows: We have a currentTime(initially 0) that is incremented each second and a Boolean ok that signals if there still are clients either in the waiting list or at one of the queues. While the current time is not the total simulation time(stored in the field timeLimit), the method

toPrint() is called in order to write into the file the contents of each of the queue and the waiting list. After that, the current time will be incremented and the thread will be suspended for a second using the sleep() method. After we exit the while loop, all threads will be stopped using the method described above and the averageWaitingTime will be computed and also written into the file. We also have a main() method that has as parameters the arguments(more specifically, they will be the file to read data from, in order to generate clients, or to determine how many clients and queues are at our disposal and also to find out the simulation time(timeLimit) and the file to write into, in order to display the results based on these parameters). The main method defines a simulationManager object through its constructor and creates and starts a new thread of this type.

5. Results

The assignment reads from a file and displays into another file the results. We have three files that have been tested, in-test-1.txt, in-test-2.txt, in-test-3.txt and the results are displayed into the files out-test-1.txt, out-test-2.txt, out-test-3.txt. The input files(the first three) all have different parameters, ex: the first test file describes that we have 4 clients and 2 queues available, 1 minute of simulation time and 2<arrivalTime<30 and 2<serviceTime<4 for each of the clients. It looks the following way:

4

2

60

2,30

2,4

The output file displays for each time the status of the queues and the waiting list => to demonstrate, provided that at the beginning in the file will be displayed :

Time 0

Waiting clients: (1 , 5 , 2); (3 , 9 , 5); (2 , 13 , 2); (4 , 24 , 6);

Queue number 1: closed

Queue number 2: closed

for the 13th second in the output file will be printed something like this:

Time 13

Waiting clients: (4 , 24 , 6);

Queue number 1:(3 , 9 , 1);

Queue number 2:(2 , 13 , 2);

In the end, at the bottom of the file, the average waiting file will be displayed:

Average waiting time: 3.75

This is just an example, and the principle is the same for all 3 input files.

There has also been a .jar file generated that can be run from the command line with these input and output files and also with others.

6. Conclusions

The assignment simulates, in real time, a situation in which N clients arrive at certain times to a waiting list and are distributed to Q queues in such a way in which the average waiting time is improved. As a later improvement, there could be developed a GUI for the display of the status of the queues at each second in order to be more convenient for the user to see the result of the simulation.

7. Bibliography

Java_Concurrency.pdf

https://www.caveofprogramming.com/categories/java-multithreading/