Fundamental Programming Techniques

Assignment 2

Queues Simulator

Pașc Andreea-Mădălina

2nd Year, Computer Science

Group 30421

1. Assignment Objective

The main purpose of this assignment is to implement a simulation application, aiming to analyze queuing based systems for determining and minimizing clients’ waiting time.

Queues are commonly used to model real world domains. 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 system is interested in minimizing the time amount their "clients" are waiting in queues before they are served. 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 service supplier. When a new server is added the waiting customers will be evenly distributed to all current available queues.

The application should simulate (by defining a simulation time 𝑡S𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛) a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues. All clients are generated when the simulation is started, and are characterized by three parameters: ID (a number between 1 and N), 𝑡A𝑟𝑟𝑖𝑣𝑎𝑙 (simulation time when they are ready to go to the queue; i.e. time when the client finished shopping) and 𝑡S𝑒𝑟𝑣𝑖𝑐𝑒 (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 𝑡A𝑟𝑟𝑖𝑣𝑎𝑙 time is greater than or equal to the simulation time (𝑡A𝑟𝑟𝑖𝑣𝑎𝑙 ≥ 𝑡S𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛).

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 (𝑡A𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁 ≤ 𝑡A𝑟𝑟𝑖𝑣𝑎𝑙 ≤ 𝑡A𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋 );
* Minimum and maximum service time (𝑡S𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁 ≤ 𝑡S𝑒𝑟𝑣𝑖𝑐𝑒 ≤ 𝑡S𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋 );

Output:

* The output file should contain the status of the pool of waiting clients and the queues as the simulation time *tSimulation* goes from 0 to *tSimulationMAX.* The simulation is finished when there are no more clients in the waiting queue or at the service queues or *tSimulation > tSimulationMax.* The average waiting time is computed and appended to the file.

1. Problem analysis, scenarios, use cases

This program is a simulation of different clients waiting in line to receive a service. Just like in the real world, they have to form different queues, each queue processing clients simultaneously. The main focus is to determine and minimize the clients’ waiting times and to compute their average waiting time.

The input of this assignment is read from a .txt file using the command *java -jar PT2020\_Group\_FirstName\_LastName\_Assignment2.jar in.txt out.txt.* The input file will be formatted according to the problem’s requirements mentioned above. The clients will be generated randomly within the bounds given in the file, meaning that the number of servers, clients and the time will be taken from the file and for the service time and arrival time a random value within the bounds in the file will be taken.

After the user inputs the command and if the file contains the specified format, the output will be put in a .txt file. In the output file, the user should be able to see all of the necessary steps for tracing the clients within the queues:

* The waiting clients: the clients remaining to enter a queue
* The moment of time
* The separate queues, whether they are closed or not and if they have a client
* The id, arrival time and service time of the client and their progression

Problem scenarios:

* Preconditions: the user has to introduce the command *java -jar PT2020\_Group\_FirstName\_LastName\_Assignment2.jar in.txt out.txt*. The data in the input text file should have the same format as mentioned above. If it has a different format, there will be an incorrect output or no output at all.
* Normal scenario: The user introduces the command and the input file is formatted correctly. After pressing enter, and the program is finished, an output file will appear. In this output file, the user should be able to visualize all the steps that were necessary for simulating the queues.

1. Implementation

O imagine care conține captură de ecran, computer

Descriere generată automatUML Diagram

* Data structures

The data structures used for the implementation of the problem are:

* ArrayList: a part of collection framework and is present in java.util.packages. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.
* Packages

A package in Java is used to group related classes. Think of it as a folder in a file directory. They are used to avoid name conflicts, and to write a better maintainable code.

Three different packages were used for this code:

* components: in this package is the main logic behind the implementation of the data that we collect from the input files. Here we have the classes Task, Scheduler, Server, Simulation. All of these classes re going to be presented in the next section.
* strategy: in this package are the classes which allow us to choose the priority in which the clients are put in the queues. We have 2 strategies: ConcreteStrategyQueue and ConcreteStrategyTime, a SelectionPolicy and a Strategy Interface. All of these classes are going to be presented in the next section.
* main: in this package are the classes used to extract the input from the input file and to add the output to the output file and the Main class. The classes are going to be presented in the next section.

All of these 3 different packages are under a main package: com.quessimulator.

* Classes

Java is an object-oriented programming language, so everything in Java is associated with classes and objects. A class is the blueprint from which individual objects are created.

1. Package *components*
2. Class *Task*

* As mentioned in the requirements, in the input file we are given a set of parameters: the number of clients, the number of servers, the time, the bounds for the service and arrival time
* Each client is being defined by the tuple (ID, serviceTime, arrivalTime). The elements from the tuple are generated randomly, within the bounds presented in the input.
* This class is used in order to set the elements from the tuple for each client. For this, we have a constructor Task with the parameters represented by the elements in the tuple and we have a getter for each element. At the same time, we have a method which helps to decrease the processingPeriod after the client enters the queue.

1. Class *Server*

* This class represents the Queue in which the clients are put.
* We have a constructor Server which is used in order to set the tasks, the waitingPeriod for each task and the CycleBarrier synchronizer
* waitingPeriod is an AtomicInteger. An AtomicInteger is used in applications such as atomically incremented counters, and cannot be used as a replacement for an Integer. However, this class does extend Number to allow uniform access by tools and utilities that deal with numerically-based classes.
* The tasks are put in an ArrayBlockingQueue. ArrayBlockingQueue class is a bounded blocking queue backed by an array. By bounded it means that the size of the Queue is fixed. Once created, the capacity cannot be changed. Attempts to put an element into a full queue will result in the operation blocking. Similarly attempts to take an element from an empty queue will also be blocked. Boundness of the ArrayBlockingQueue can be achieved initially by passing capacity as the parameter in the constructor of ArrayBlockingQueue. This queue orders elements FIFO (first-in-first-out).
* A CyclicBarrier is a synchronizer that allows a set of threads to wait for each other to reach a common execution point, also called a barrier. CyclicBarriers are used in programs in which we have a fixed number of threads that must wait for each other to reach a common point before continuing execution. The barrier is called cyclic because it can be re-used after the waiting threads are released.
* We have getters for each parameter and we have an addTask method which adds a new task to the list of tasks and its processinfPeriod
* It is a Runnable class which means that we are going to have a method run(). In this method we take the next task from the queue and we stop the thread for a time equal with the task’s processing time. At the same time we decrement the waitingPeriod for the task. The waitingPeriod is decremented by the current thread once a task is completed.

1. Class *Scheduler*

* As the name suggests, it sends the tasks to the server according to the established strategy.
* It has a constructor which is used in order to iterate in the server list and create the servers and the thread with the objects
* We have a changeStrategy method which is used in order to apply the strategy pattern to instantiate the strategy with the concrete strategy corresponding to the policy
* The dispatchTask methid is used in order to call the strategy addTask method and to terminate or start new threads according to the number of tasks there is in a server/queue. If the thread is TERMINATED it will create a new one and if it is NEW is will start it
* We also have a getter for the list of servers

1. Class *Simulation*

* As the name suggests, it is used to gather all the data and to implement the simulation of the entire system
* Here, the timeLimit, min and max arrivalTime, min and max serviceTime, numberOfServers, numberOfClients and strategy are instantiated according to the input values.
* We have a scheduler parameter which will be responsible with the queue management and client distribution and a pool of tasks and servers
* The constructor is used in order to set the input for the file input and the output to the file output
* Method generateNRandomTasks and generateRandom are used to generate randomly the arrival and service times according to the limits given in the input file. At the same time, it sorts the list of tasks with respect to the arrivalTime
* Method ccheckEnd is used in order to verify whether there are still tasksk in the servers or not
* The lass is a Runnable class, so it implements a run() method. Here, all the parameters are instantiated with the values from the input file, a new scheduler is created and a strategy is fixed. After the instantiation, the method starts counting the time and tasks are being put in the server if their arrivalTime equals to the currentTime. At the same time, the task is removed from the waiting list
* The output file and the time are being updated

1. Package *strategy*
2. Interface *Strategy*

* An interface is a completely "abstract class" that is used to group related methods with empty bodies.
* This interface is used for the addTask method

1. Enum *SelectionPolicy*

* An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).
* In this enum we put the policy for the selection: selection based on the shortest queue or on the shortest time

1. Class *ConcreteStrategyQueue*

* In this class we add the task with the shortest queue

1. Class *ConcreteStrategyTime*

* In this class we add the task with the shortest waitingPeriod

1. Package *main*
2. Class *FileInput*

* This class is used to get the input values form the input file
* It has a constructor with a file reader which reads line by line the elements from the file and assigns them to the parameters
* We also have here getters for every parameters in order to access them from other classes

1. Class *FileOutput*

* This class is used in order to write and format the output file
* We have setters in order to assign the values which are going to be written in the file, such as the time, the list of tasks and the servers
* The constructor is used in order to format the output in a readable way and to put the values in the file

1. Class *Main*

* This class is used to create a new Simulation object and to start the Thread process

1. Input File/Output File Model

An example of input/output files is the following:

In-Test.txt

4

2

60

2,30

2,4

Out-Test.txt

Time 0

Waiting clients: (1,2,2); (2,3,3); (3,4,3); (4,10,2)

Queue 1: closed

Queue 2: closed

Time 1

Waiting clients: (1,2,2); (2,3,3); (3,4,3); (4,10,2)

Queue 1: closed

Queue 2: closed

Time 2

Waiting clients: (2,3,3); (3,4,3); (4,10,2)

Queue 1: (1,2,2);

Queue 2: closed

Time 3

Waiting clients: (3,4,3); (4,10,2)

Queue 1: (1,2,1);

Queue 2: (2,3,3);

Time 4

Waiting clients: (4,10,2)

Queue 1: (3,4,3);

Queue 2: (2,3,2);

…

Average waiting time: 7

1. Results

After running the program, with 3 different input files: “in-test-1.txt”, “in-test-2.txt”, “in-test-3.txt”, three separate output files were created “out-test-1.txt”, “out-test-2.txt”, “out-test3.txt”. The input files were tested with the compiled code as a .jar file and within the program by setting the fi les as arguments.

1. Conclusions

As a conclusion, this program helped with realizing the importance of time management. Small values like the input ones may be insignificant, but when it comes to very large values, a lot of servers and a lot of tasks per server, it is very important to construct a model in which to assign the tasks in a optimal way, such that the time costs and the memory costs are as small as possible. It also provided a very good exercise regarding the thread management.

1. Bibliography

* [www.w3schools.com](http://www.w3schools.com)
* [www.tutorialspoint.com](http://www.tutorialspoint.com)
* [www.geekforgeeks.com](http://www.geekforgeeks.com)
* [www.baeldung.com](http://www.baeldung.com)
* [www.docs.oracle.com](http://www.docs.oracle.com)
* Java Concurrency pdf