Technical University of Cluj-Napoca

Queues simulation

-project documentation-

Pop Andrei

Group 30424

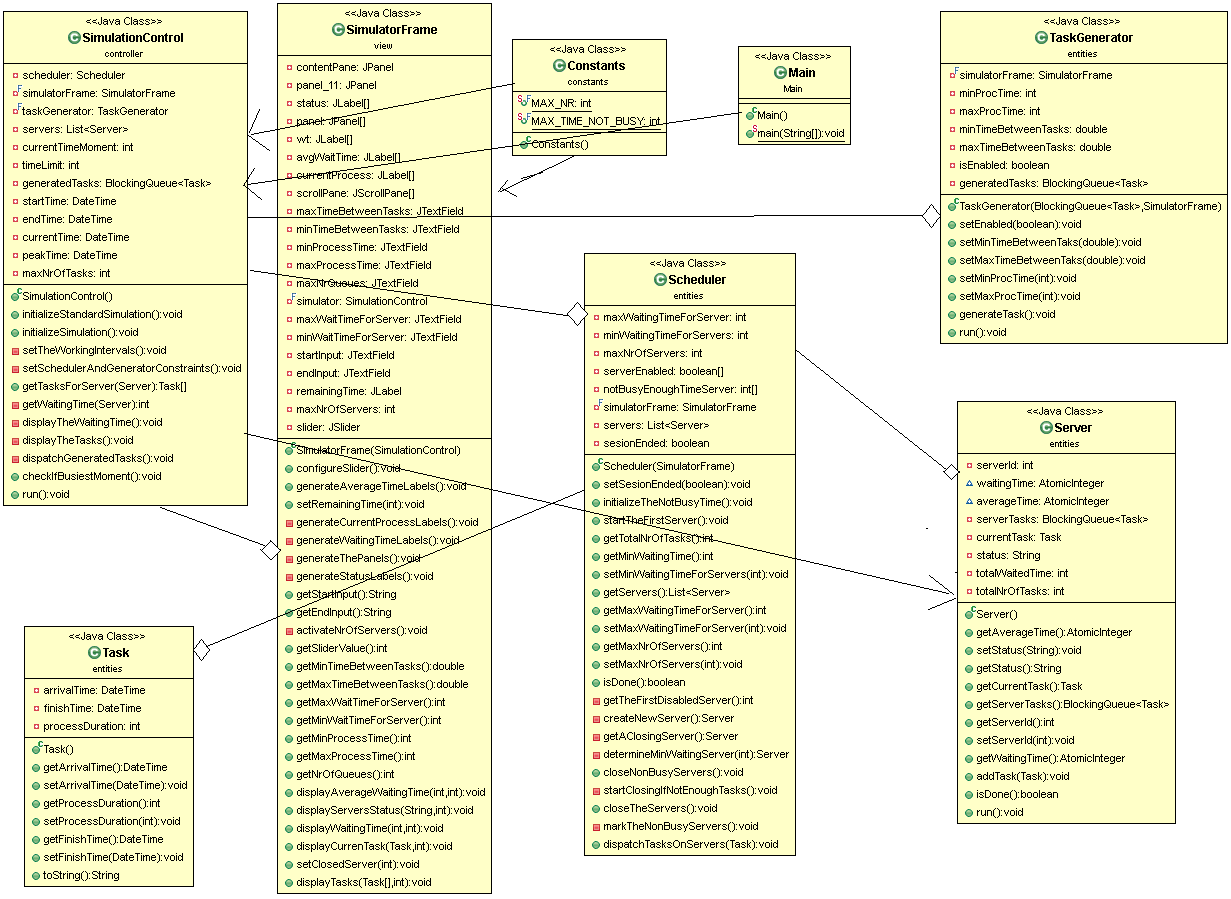
1. Objective

The objective of this project was to implement a queue simulation. The application tries to simulate the multiple queues from a megastore and also deal with closing and open them. The queues can also be seen as servers which communicate between them in order to complete a big task. Each of them is given smaller tasks to complete. In order to simulate the customers or the tasks in a queue, a task generator was created. Each task can be seen as a customer who’s waiting in order to pay for what he/she bought. In order to do this simulation, some constraints and some constants were needed. The application can also be fed with real word data, which mean that it can represent the image of multiple queues and how they’re managed during time. The user can modify the number of tasks for each queue and also the duration of the tasks.

1. Problem analysis, modeling, scenarios, use cases

In order to implement the simulation, it was needed a task generator. The purpose of this task generator was to create different tasks, at different moments of time, with different durations. Each of the task is placed into a waiting queue, from which it is popped when the servers becomes available. The application starts running with a single server. After that, more servers are added as needed in order to obey the requirements that were imposed related to the maximum waiting time per server. Each of the server will have a variable number of tasks, such that it would not get over a waiting time. If the waiting time of the server would be greater if adding a new task on the respective server, a new server is created so the new tasks are added a new server. In order to save money and other resources, the servers are closed after a certain moment of time. If they have less customers than they should, for a long period of time which will be defined by the user, then they will start closing. This means that when they start closing, the tasks that would usually be accomplished by them are passed to other servers which are opened. If the other servers become full, then the closing server is re-opened and new tasks are added on it. Each of the task has a specific duration, so the number of tasks on a server isn’t relevant because there can be many tasks with a short time needed in order to be completed and also on other servers can exist tasks which are easy to solve and do not need too much time for this. There are also a limited number of servers which are available. When the maximum number of servers is reached and there are more tasks to be added, then a message is passed to the user, informing him or her that all the servers are full and the current task can’t be placed to a certain server. If there are very few tasks to solve, then only one server will be kept open, all the others will be closed in order to save resources. The user has the possibility to modify the time interval between the tasks. So if there’s a smaller time interval between the tasks, then more tasks will be generated and placed on the servers. If there’s a bigger time interval between the tasks, then fewer tasks will be generated and added to the processing servers. Each of the servers can be seen as a different object who’s lifetime depends on the number of tasks that it has to process. All the servers are controlled and managed by a scheduler who’s in charge of opening, closing and dispatching tasks on servers. When a new task arrives into the scheduler, the scheduler will find out which is the server with the minimum waiting time and the task from the scheduler will be added on the server with the minimum waiting time. If there’s no available servers for the new task, the scheduler will create a new server on which the task will be placed. Also, in case of a server who didn’t have enough tasks during the last time units, the server will start closing, which means that the tasks will be distributed to other servers which are still active and not to the server that is currently closing. The currently closing server will receive new tasks only if the other servers are already full, because it isn’t worth opening a new server. In order to use the application, the user has to provide the minimum waiting time for a server, the maximum waiting time for the server, the number of servers, the minimum time between the tasks and the maximum time between the tasks. In order to simulate the arriving of new clients, the user can user a slider to modify dynamically the number of tasks generated. The value of the slider is inverse proportional to the number of the clients which will be generated in a time interval. Also the user can start the simulation by pressing a button which can be found on the user interface and is called “Start simulation”. The most important thing for a good and relevant simulation is to provide a valid input. This means that in order to see what’s happening, the user should provide an input which will allow him or her to watch on the screen the different steps of the processes. Each server will be implemented on it’s own thread. This means that one server won’t know about the execution phase or the state of another server. All the server will be coordinated by the scheduler. The scheduler needs also some control, so the top unit, or the control unit is called “Simulation control”. This unit is in charge of all the action that need to be performed in order to provide a good simulation. It also controls the graphical user interface. In order to communicate with all the other components, the communication is done via getters an setters. The unit which interacts with the user is called “Simulation Frame”. The role of this window is to provide the user a graphical representation in real time of what’s going on with the servers along the time. There are also statistic date which is displayed on the screen in order to provide more information.

1. Design

3.1 UML Diagram

3.1 Class design

The application is composed of 8 classes. Each of these classes has a certain functionality. Also they’re grouped by their functionalities in five packages. I have also used the MVC pattern. They’re only one controller, one view class and multiple models. The models were created such that would communicate with the controller and the controller will update the relevant information on the screen. In the following lines, each package will be presented in details:

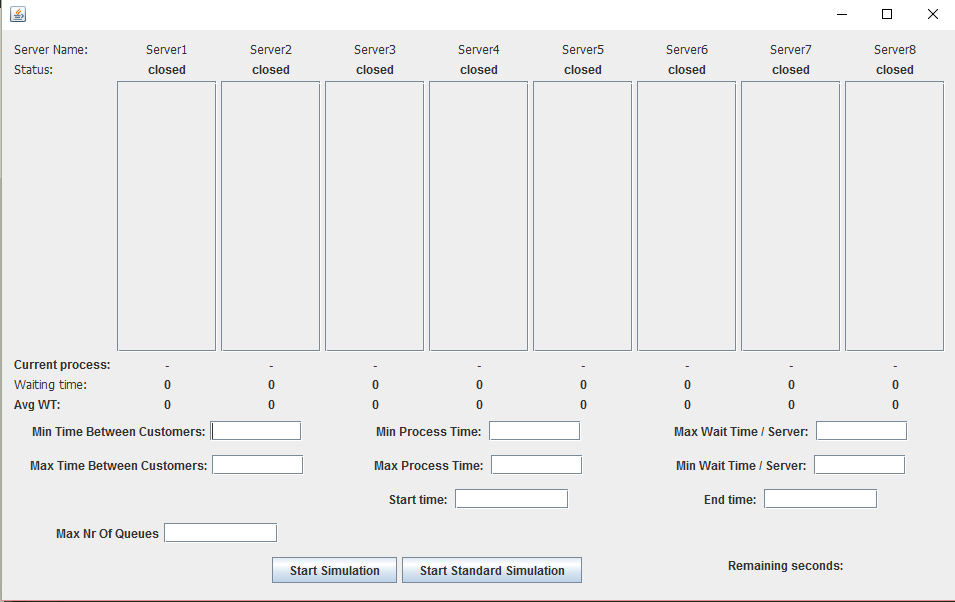
* **Constants package**: this package was designed in order to contain all the required constants. I have designed this class because otherwise it is necessary to pass the constants from a class to another which is messy. Another way to keep the constants would be to declare them each time we need them, but working this way, a constant which represent something would be declared multiple times, which is not a desired thing. This package is composed only by one class called “Constants” which is organized such that the constants related to certain fields are grouped into smaller inner classes.
* **Controller package**: this package contains the controller of the application. It is composed by only one class which controller the whole flow, of course, communicating with other classes. The controller class is called “SimulationControl”.
* **SimulationControl:** has attributes and capabilities in order to manage the whole simulation. The Simulation Control class communicates with all the other classes. At the beginning it sets all the constraints for all the other classes. For example, for the TaskGenerator, it sets maximum and minimum time for a task, minimum and maximum time between tasks etc. Also, it creates the Scheduler, which creates and starts the first server. Also, the SimulationControl class communicates with the SimulationFrame class which is the view of the whole application. It receives the user input from this class and passes it to the other associated classes.
* **Models package:** this package contains all the models of the application. Each of the models has a specific task, providing data which will be then processed and sent again to the other classes by the SimulatorControl class. The models communicate between them, mainly the communication being done with the help of the controller. The package contains the following classes:
* **Scheduler:** this class has the role of the task manager. It has access to all the servers, so each time a new task is generated by the Task generator class, the Scheduler will find the server with the minimum waiting time and pass the task to it. If all the opened servers are full, then the Scheduler will look for a server which is currently closing. A server is closing after a certain time interval if it doesn’t have enough tasks to process. This means that it is not worth keeping it opened. So in case that all the other servers are full, and a closing server is found, then the closing server is kept opened and the tasks will be passes to its queue. If all the servers are full and there are no other closing servers which can be opened, then a new server will be created and opened in order to process the current tasks. Another requirement that should be fulfilled by the Scheduler is that it should close all the servers which are done with processing their tasks. So after each moment of time, the Scheduler checks in order to find out if there are servers which need to be closed.
* **Server:** the goal of the Server is to process all the tasks assigned by the scheduler in a short time interval. The Server will take the tasks one by one from its queue which was filled by the Scheduler, analyze it, process it and then repeat these steps for the next task from its queue. If there are no more tasks in the queue, the server will be blocked and it will wait for a new task to be passes by the Scheduler. Usually, when a server is done processing all its tasks, if it isn’t the last one, it will be closed in order to reduce the costs.
* **Task:** this class represents the objects which are processed by the servers. In real world, an equivalent to the Task would be a customer which is waiting into a line in order to get the desired services which are provided by some specialized people, in our case, servers. Each task has a starting time, and an ending time. After the task was completed, it will be removed from the server’s queue, and by computing the difference between the ending time and the starting time it will be determined how long did it take for the server to process the task.
* **TaskGenerator:** the goal of this class is to generate tasks which will be processed by the servers. What it actually does, is simulating the arrival of clients into a megastore. At certain moments of time there will be customers coming, one or more which then will choose a line to wait in order to checkout. The task generator can be configured to generate tasks which will take a certain amount of time to be processed. The bound for the processing time of a task can be set by the user on the graphical user interface. Another thing that can be configured and controlled by the user is the time between the generated tasks. Between each task/ each set of tasks, there will be a certain amount of time who’s bounds can be set also by the user. The user will have the possibility to modify dynamically the time between the tasks.
* **View package:** this package contains the view part of the application. It displays on the screen all the relevant information which is needed by the user. It is composed by a single class which will be presented in details below:
* **SimulatorFrame**: the objective of this class is to display on the screen the relevant information about the server and the tasks which are currently processes. The SimulatorFrame will also help interacting with the user. At the beginning, the user will give some initial input in order to configure some classes from the models package, and then press a start button. After the start button was pressed, the simulation will start and the user will see on the screen the effects of its provided inputs.

3.2 Data structures

For creating the simulation, I had to use multiple threads because each of the server needs to run independent of the others, so they don’t share anything. Each of the servers, will run on a different thread, and they have to communicate with the scheduler, which controls them and which is on another thread. In order to do this I had to use a synchronized date structure. I have used LinkedBlockingQueues. This data structure makes sure that only one thread can write at one moment of time and that each operation is done atomically, so when the next thread will access the fields, they will valid, because the operation performed by the previous thread was done atomically. When a new task was generated, it was passes to the Scheduler. All the tasks that were generated, were added by the TaskGenerator into a ListBlockingQueue, and then the Scheduler was distributing them among the available servers. Each of the servers also has a ListBlocking Queue in order to keep all the tasks in the order that they need to be processed. When a task is processed by a server, it is removed from the waiting queue and the size of the queue is decreased.

3.3 Graphical user interface

The graphical user interface is composed only by one window which contains all the information about the simulation. It can be seen during the simulation how the tasks are distributed to each of the servers and how they’re processed by each of the servers. It will also be shown the status of each server, the average waiting time, the current processed task, and the waiting time. Also, at the beginning, the user will have the possibility to set some constraints on how the data should be provided by the models. As it can be seen on the frame, the maximum number of server which van be displayed on the screen will be eight. Another thing that the user can set is the time interval between he/she wants to run the simulation. On the frame, it will be shown how many seconds are left until the end of the simulation. At the end of the simulation, the servers aren’t forced to close. They’re closed only after processing all the tasks that they have been assigned. At the end of the simulation, a window will pop-up and show the peak hour, the time at which the most customers were coming, in our case, the most tasks were generated.



1. Future improvements

For further improvements, it should be taken into consideration working with data which is fed from the real world. This application could be used as a task or customer coordinator. When a task occurs in the real world, the scheduler will be notified and the task will be passed to a server in order to take care of it. If the application is fed with data from the real world, than it will show how the thing should be organized in order for the tasks to be processed as fast as possible. Another thing which can be improved is working on multiple simulation. One simulation is not enough many times, because we many parallel such systems, so having the possibility to watch more simulations at one time would be very helpful. The information shown on the frame is enough for a specialized user. In order to make the application relevant for non-specialized users, it would be helpful to have graphs drawn on the screen which will show the evolution of the queues in time. I think that a very important improvement would be to integrate the application into a network, such that more simulations would share data about the servers and the information processed. In order to make the application optimal, the servers among several applications will be shared in order to complete the tasks. If a simulation doesn’t have any extra servers available, than it will send the task to another simulation which has servers that aren’t used.

1. Conclusions – what I have learned

During the time I have designed the application, I have learned especially how to work with threads. This is very important because in the real world we want everything to work as fast as possible, so there’s always a need for paralleling things. After designing this application, I’ve realized how important it is to make sure that the threads are communicating properly between them and that the shared variables are written back into the main memory after performing the specific operations on them. Another thing that I have learned is to avoid critical races, by using synchronized methods or by using semaphores.

As a conclusion, I think that the development of this application made me think more about how to do things in parallel and how to think about synchronization.

1. Bibliography

Books:

- Joshua Bloch, Effective Java (2nd Edition);

- Kathy Sierra, Bert Bates, Head First Java (2nd Edition), O'Reilly Media;

- Barry Burd, Java for Dummies (5th Edition), Wiley;

Websites:

- http:// stackoverflow.com/

- https:// www.oracle.com/java/

-http:// www.oracle.com/technetwork/articles/javase/index-142890.html

- http:// www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/

- http:// christoph-burmeister.eu/?p=1556

- http:// theopentutorials.com/tutorials/java/jdbc/jdbc-mysql-create-database-example/

- http:// tutorials.jenkov.com /