

Queues Simulation

UTCN CTI ENGLISH

Muresan Daniel, 30422

Contents

1. Goal of the application
2. Problem analysis
 - 2.1 Assumptions
 - 2.2 Modeling
 - 2.3 Use cases
3. Projecting
 - 3.1 UML diagrams
 - 3.2 Classes projecting
 - 3.3 Relationships
 - 3.4 Packages
4. Implementation
5. Testing
6. Results
7. Conclusions
8. Bibliography

1. Goal of the application

Timing is one of the most important concern in these days. Everybody tries to obtain the most of every second during a day. From traffic jams to shop queues all these types of events are equivalent with losses of time. And the main objective of the society is to handle better this situations in order to save as much time as possible.

The goal of this application is to design and implement a system which simulates the evolution of some queues, containing tasks which require a certain amount of time to be solved and some servers which should solve those tasks, like the queues that form in shops at checkpoints with clients waiting to pay for what they bought.

The main operation i will implement is the way one task is distributed to a certain server, more precisely every time a server arrives and needs to be distributed to a server, it should be chosen for it the server with the smallest waiting time.

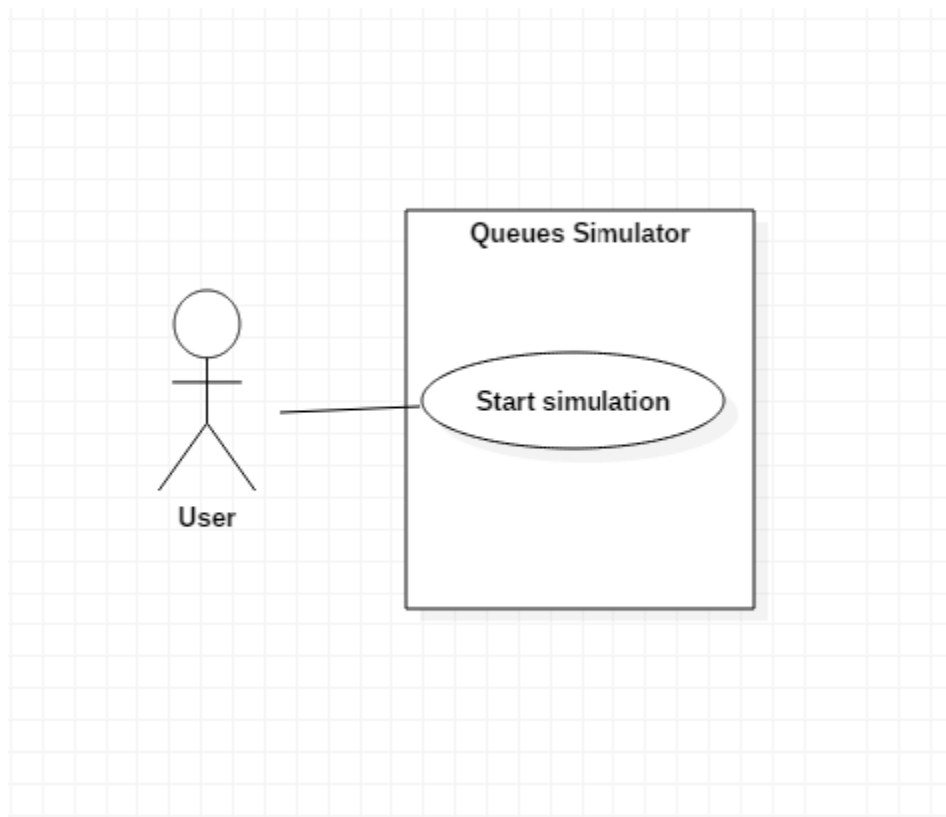
This application will also have an easy-understandable interface with intuitive buttons, text areas and labels. It will also have a section in which it would be able to display the real-time evolution of the queues including images of tasks and servers.

2. Problem analysis

This application should be able to receive some basic information about the simulation environment like: time limit, processing time interval for task, arriving time interval for tasks, number of servers, number of tasks, maximum number of tasks per server; and it should simulate based on the input provided by the specific case including: generating task , sending tasks to servers, “process” the task, and then remove it from that servers queue until all generated tasks are processed and then also give as output some statistics, like average waiting time or peak hour.

The assumptions that I made for the use of this application is that the user will introduce integer numbers which would be valid and correspond to a good case of simulation, not bad input like negative time intervals, negative number of servers or tasks or other possible inputs that would not make sense and made the simulation impossible or to behave wrong.

The application has only one use case in which the user introduces a set of data for the simulation parameters and then starts the simulation.

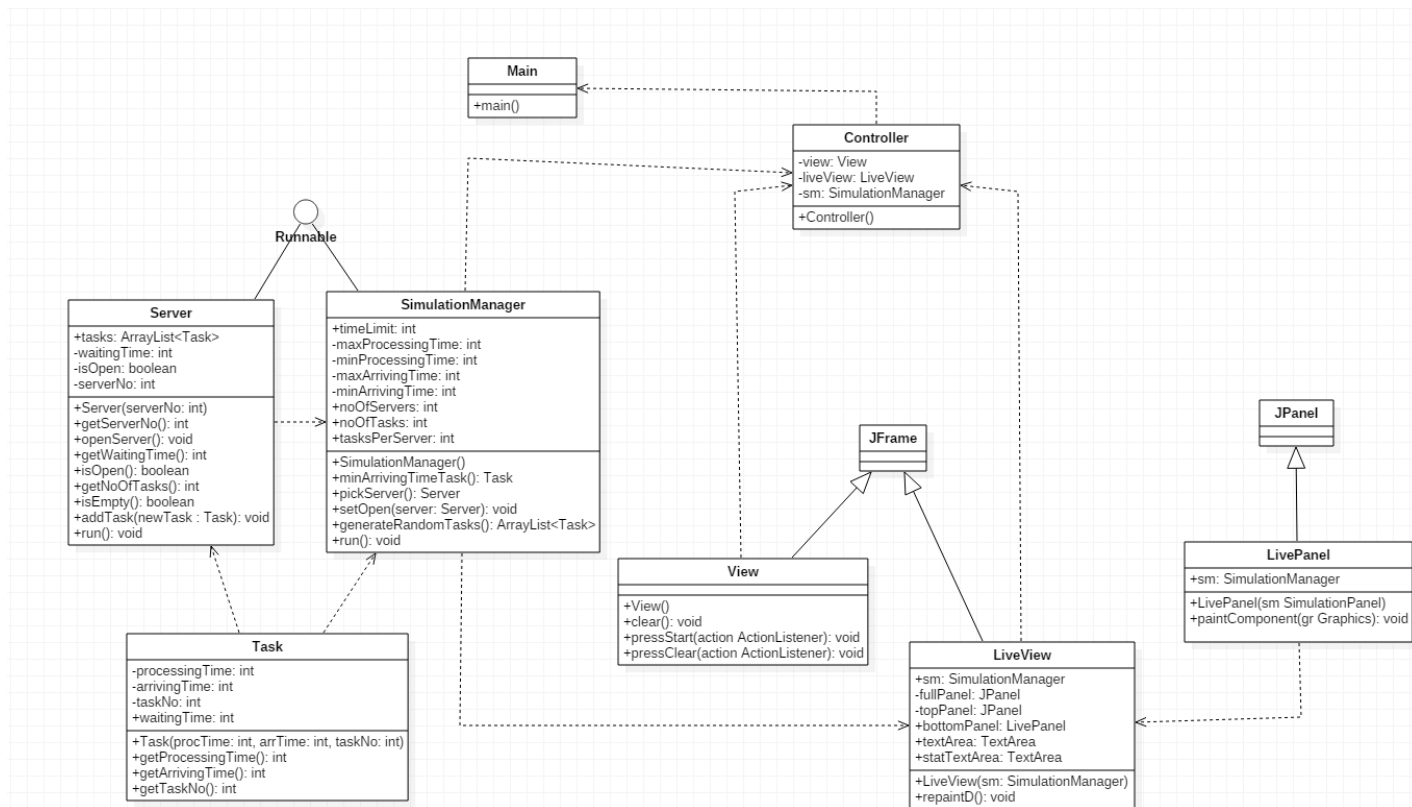


- **Use case : Start simulation**
- **Primary Actor: User**
- **Main success scenario:**
 1. The user introduce simulation parameters in the corresponding fields on the user interface.
 2. The user starts the simulation by pressing the Start button form the interface.
 3. The application reads the data form the fields of the user interface.

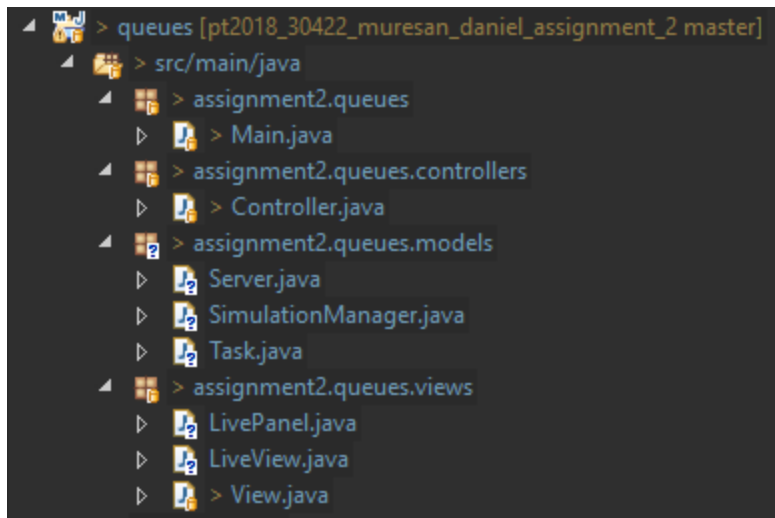
4. The application creates a simulation environment corresponding to the input values and starts the simulation.
 5. The application modifies in real-time the user interface displaying the evolution of the queues.
 6. The application displays the statistics of the simulation after all tasks were processed and all the queues are freed.
- **Alternative sequences:**
 - a) Invalid data input
 - when the application tries to process the parameters given by the user if they are not integer numbers or they are not specified a message is transmitted
 - the scenario returns to step one

3. Projecting

The classes I used for developing this application are the following ones: Task, Server, SimulationManager, as models, View, LiveView, LivePanel as views, Controller as a controller and Main for starting the application. I also used the predefined interface Runnable for working with threads and the classes JFrame and JPanel provided by Java Swing. All those classes are presented with the relationships between them in the following class diagram.



I used the Model-View-Controller pattern to organize my classes in packages. I used the packages as follows : assignment2.queues containing the class Main, assignment2.queues.controller containing the class Controller, assignment2.queues.models containing the classes Task, Server and SimulationManager, assignment2.queues.views containing the classes View, LiveView and LivePanel.



4. Implementation

Task class

Is the class representing the client which waits in a queue. It has some specific attributes: `processingTime` which represents the time needed for that task to be done, `arriving Time` which represents the time when that task is dispatched to a server and `taskNo` which is an integer which uniquely identifies one object of this class. It has a constructor with three parameters corresponding to those attributes and also getters for those attributes because they are private. It has an extra attribute which is public and corresponds to the time a task has to wait from the moment it is dispatched to a server until is "done".

Server class

This class represents the checkpoint from a store where a queue is formed. It has as attributes an `ArrayList` of tasks containing the tasks which are in the queue for that specific server. A boolean attribute which specifies if the server is open and an integer attribute which uniquely identifies a specific object from this class. It has a constructor with a parameter for that server number attribute and also sets the open attribute as false and the `ArrayList` with tasks

empty. It has a method for setting the boolean attribute open true, some getters for the private attributes and a method which calculates the waiting time for that server as the sum of all processing times of the tasks in the queue of that server. It also has a method for adding new tasks to the queue. And the final method run() which operates on the thread by putting it to sleep with the time corresponding to every task processing time. This method overrides the run() method from the Runnable interface.

@Override

```
public void run() {  
    int cnt ;  
    try {  
        while ( !tasks.isEmpty() ) {  
            cnt = 0 ;  
            Task t = tasks.get(0) ;  
            int wt = t.getProcessingTime() ;  
            waitingTime += wt ;  
            Thread.sleep(wt) ;  
            LiveView.textArea.append( "Task " + t.getTaskNo() + "  
was finished. Processing time: " + t.getProcessingTime() + "\n" ) ;  
            tasks.remove(t) ;  
            if (this.isEmpty())  
                this.isOpen = false ;  
            LiveView.repaintD() ;  
            for (Task t1 : tasks )
```



```

        cnt++;

        LiveView.textArea.append(( "Server " +
this.getServerNo() + " : " + cnt + " tasks remained" + "\n" ));

    }

    } catch (InterruptedException e) {

        System.out.println( " Thread wasn't done " );

    }

}

```

SimulationManager class

It is the core of the application because there tasks are generated, servers are created and for each of them a thread is started, log messages are sent from that class to the graphic interface and basically all of the program's functionality is stored here .

It has multiple attributes for the simulation parameters like time limit , minimum processing time , maximum processing time , minimum arrival time , maximum arrival time , the number of servers , the number of tasks , the number of tasks per server , the peak hour which will be calculated based on the other parameters . And the most important attributes are an array of servers and an ArrayList of tasks .

As methods, firstly is has a constructor with parameters corresponding to all the intervals of the simulation which will be provided by the user on the graphic interface. In the constructor also generates the required number of servers. It has a method for taking the task with the minimum arriving time form the ArrayList . Another method is used for picking the server with the minimum waiting time in order to send a task to that server. It also has a method which opens the server and starts a new thread for that server. Another method for generating random tasks which will be returned like an ArrayList.

The most important method for this class is also the method `run()` which overrides the method from the interface `Runnable`. In that method all generated tasks are dispatched to servers corresponding to the minimum waiting time the thread is put to sleep for one second between the tasks and some information are sent to the text area from the graphic interface to display the queues evolution.

@Override

```
public void run() {  
    Server s = null ;  
  
    int currentTime = 0 ;  
  
    int cnt = this.noOfTasks ;  
  
    try {  
        while ( !tasks.isEmpty() && currentTime < timeLimit ) {  
            Task t = minArrivingTimeTask() ;  
  
            if ( t == null )  
                System.out.println( "There are no tasks" ) ;  
  
            int wt = 0 ;  
  
            Thread.sleep( 1000 ) ;  
  
            s = pickServer() ;  
  
            if ( s == null ) {  
                Thread.sleep( maxProcessingTime ) ;  
  
                s = pickServer() ;  
  
                peakHour = t.getArrivingTime() ;  
            }  
        }  
    }  
}
```

```

    }

    for (Task t1 : s.tasks )

        wt += t1.getProcessingTime() ;

    totalWaitingTime += wt ;

    t.waitingTime = wt ;

    s.addTask(t) ;

    if ( !s.isOpen() )

        setOpen(s) ;

    LiveView.textArea.append( "Task " + t.getTaskNo() + "
arrived at server " + s.getServerNo()

                                + " with arriving time " +
t.getArrivingTime() + "\n" ) ;

    LiveView.repaintD() ;

    this.tasks.remove(t) ;

    this.noOfTasks-- ;

    currentTime++ ;

}

LiveView.statTextArea.append( "\n" ) ;

LiveView.statTextArea.append( "Average of waiting time: " +
totalWaitingTime / cnt + "\n" );

LiveView.statTextArea.append( "Peak hour: " + peakHour +
"\n" );

```

```

        LiveView.statTextArea.append( "\n" );

        LiveView.statTextArea.setVisible(true) ;

    } catch ( InterruptedException e) {

        System.out.println( "Thread was interrupted" ) ;

    }

}

```

View class

This class is the first component of the graphic user interface. It is the first frame that the user will see and it is used to receive the parameters of a specific simulation given by the user. It has labels and text fields for all the parameters required and two buttons for starting the simulation and clearing the fields if the user wants to change some of those parameters.

LiveView class

It is the second component of the graphic user interface where the user will actually see the queue evolution and the log of events. It is a frame separated into two panels one where the log of events and the final statistics will be displayed on two text areas and another panels where the real-time queue evolution will be simulated with some pictures.

LivePanel class

It is used for the panel where the queues are represented with images. It extends the JPanel class and overrides the paintComponent method which places some images on a white background corresponding to the information provided by the simulation manager at that time.

Controller class

It is the class where the graphic user interface is connected with the functionality from the simulation manager.

It has as attributes a SimulationManager object, a View object and a LiveView object. In the constructor it creates a new View object corresponding to the attribute.

It also has two inner classes which are used to handle the events from pressing the two buttons from the View frame. When the button “Start” is pressed, inside the controller a new object of SimulationManager is created with the parameters from the View frame and with that SimulationManager object a new LiveView object is created for that simulations. After those are created a new thread is started for that simulation and on the LiveView frame the log of events and the real-time evolution of the queues is displayed . When the button “Clear” from the View frame is pressed the second inner class calls the clear method from the View class and all fields are reset .

Main class

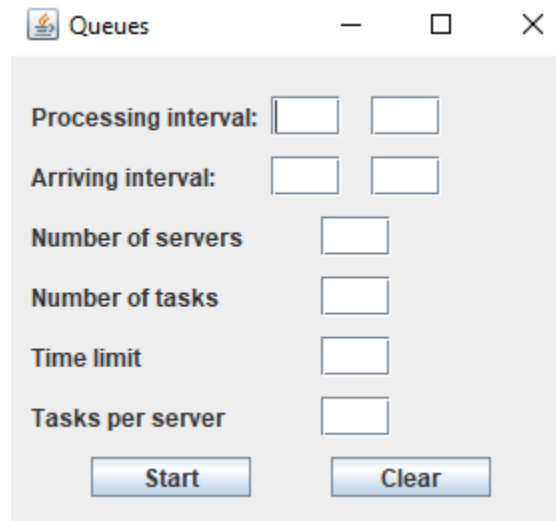
It has only one method called main() where an object of the Controller class is created and so the application starts.

5. Testing

For this application I haven't run to many tests because the methods doesn't have independent functionality and they must work together to obtain the final result.

6. Results

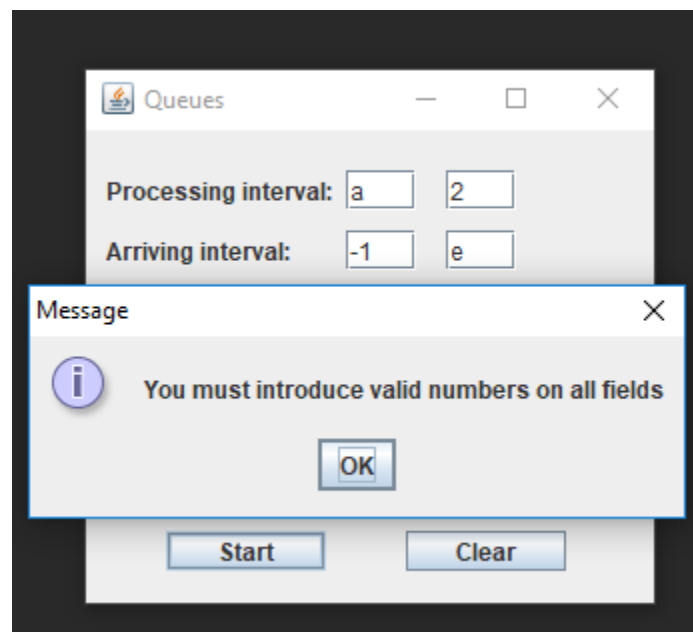
In this section I will give few screenshots with the application during some run examples. In the first image it is presented the first frame that the user will see when the application is launched.



The screenshot shows a window titled "Queues" with a standard Windows title bar (minimize, maximize, close buttons). Inside the window, there are several input fields and two buttons:

- Processing interval:** Two empty text boxes.
- Arriving interval:** Two empty text boxes.
- Number of servers:** One empty text box.
- Number of tasks:** One empty text box.
- Time limit:** One empty text box.
- Tasks per server:** One empty text box.
- Buttons:** "Start" and "Clear" buttons at the bottom.

In the next image is presented the error message transmitted if some fields are not completed or other invalid input is introduced.



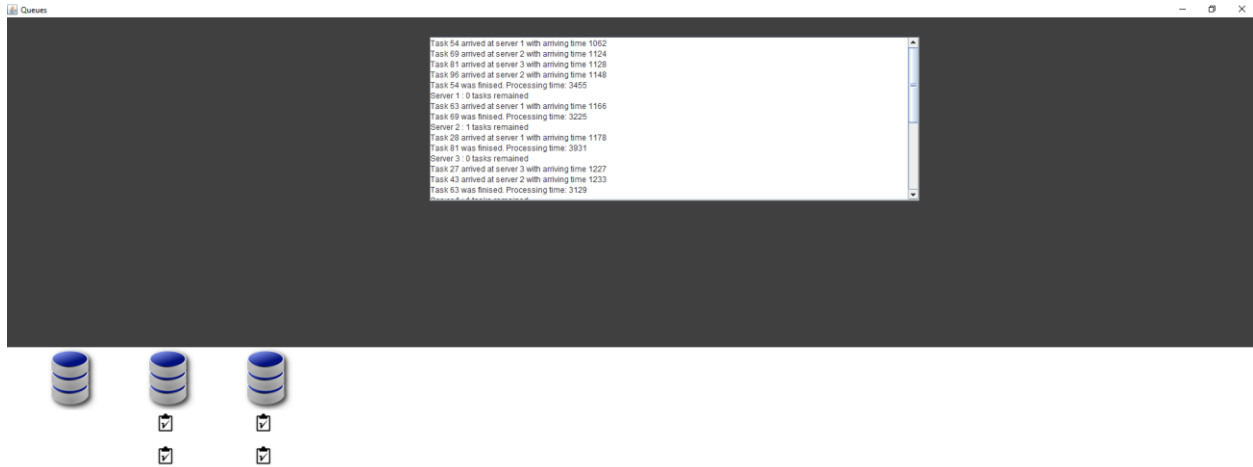
This screenshot shows the same "Queues" window as before, but with an error message dialog box overlaid on top. The dialog box is titled "Message" and contains the following information:

- Icon:** An information icon (a lowercase 'i' inside a circle).
- Text:** "You must introduce valid numbers on all fields".
- Buttons:** An "OK" button.

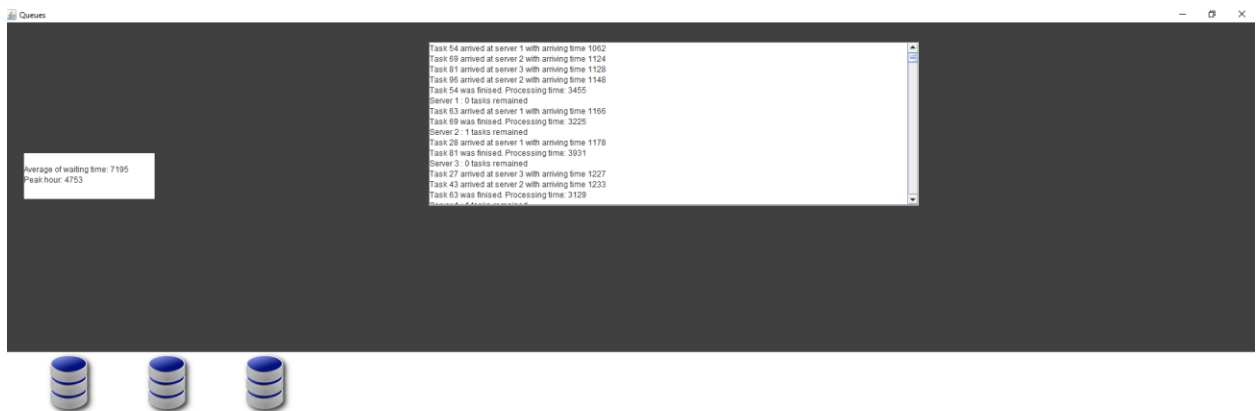
The "Queues" window behind the dialog shows the following input values:

- Processing interval:** "a" and "2".
- Arriving interval:** "-1" and "e".

And in the next image is presented the frame that appears if the input is valid and the simulation started



And next is the final result of the application when the statistics about the simulation are displayed on the second text area.



7. Conclusions

This application helped me to learn how to work with threads in order to obtain a simulation as close as possible to a real time situation. As further development for this application the possibility of closing the servers or

opening new ones when all of the existing ones are full would be some nice improvements.

8. Bibliography

<http://tutorials.jenkov.com/java-concurrency/index.html>

<https://stackoverflow.com/questions/5446396/concerns-about-the-function-of-jpanel-paintcomponent>

<https://openclipart.org/detail/215486/server-2>

<https://thenounproject.com/term/task/39626/>

<https://www.youtube.com/watch?v=UXW5a-iHjso>