Documentation

Simulation of Queuing Based Systems

Student: Rusu Daniel
Group: 30422

Table of Contents:

1. Objective

Main objective of the assignment:

Design and implement a simulation application aiming to analyse queuing based systems for determining and minimising clients' waiting time.

Secondary objectives:

- Creating a good looking, fully-functional graphical user interface.
- Implementing algorithms to run on multiple threads, in order to simulate the behaviour of real-life queues.
- Linking the user interface to the algorithms, in order to display the output correctly to the user.

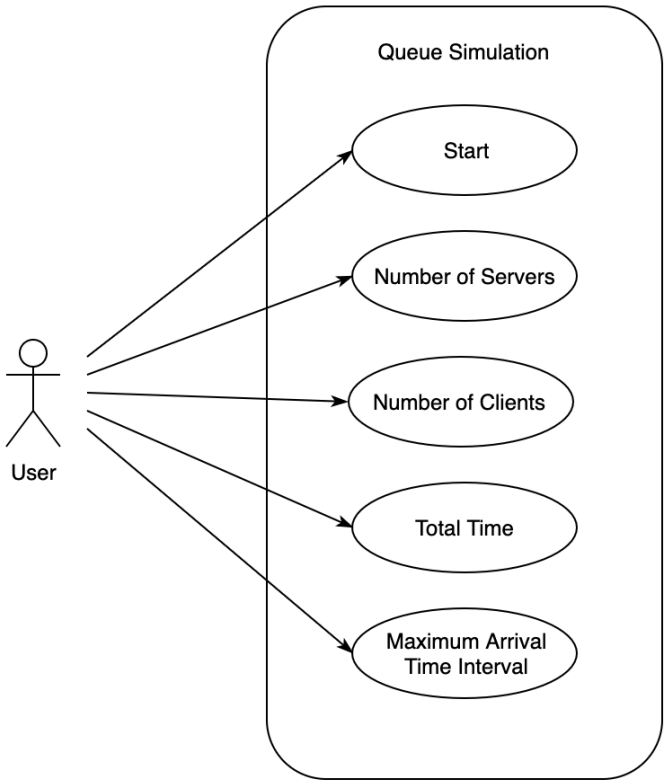
All the secondary objectives presented above will be detailed in the fourth chapter of this documentation.

2. Problem analysis, modelling, use-cases

The application is built to simulate a real-world queue system (like a supermarket).
It shows to the user the way that different clients could join different queues, based on the queue total waiting time or total number of clients, and simulates the evolution of each queue, based on arrival and processing times (calculated randomly for each client).

This application's main goal is to determine and minimise clients' waiting times during an event that happens daily (e.g. buying groceries or coffee).

To better illustrate how the application can be used, I have built a use-case diagram:



Use-cases description:

- **Number of Servers**: The input is used to provide the number of queues in the simulation. The clients will be distributed to all the queues based on the type of strategy the user wants.
- **Number of Clients**: This input is used by the user to set the size of the client (task) pool that will be distributed to all the queues (servers), in order to be processed.
- **Total time:** The user can set this time in order to make the simulation take more or less time. It represents the number of seconds during which clients are arriving at the queues, not the total running time of the application. Total running time is determined by the number of remaining clients and their processing times, after the arriving interval is over.
- **Maximum arrival time interval**: The user can set this as a time (in seconds) that tells the application the maximum number of seconds in which the next client will position at a queue. This is useful when the user wants to simulate a heavier or lighter “traffic” at the queues, in order to test the behaviour of the system.
- **Start**: In order to visualise the simulation, the user must first start the application. Once it starts running, the program will execute the whole process without any further actions performed by the user.

The user can run the application with the predefined number of servers, number of clients, total time and maximum arrival time interval, or change any of the parameters, so that the simulation will present the desired real-world situation.

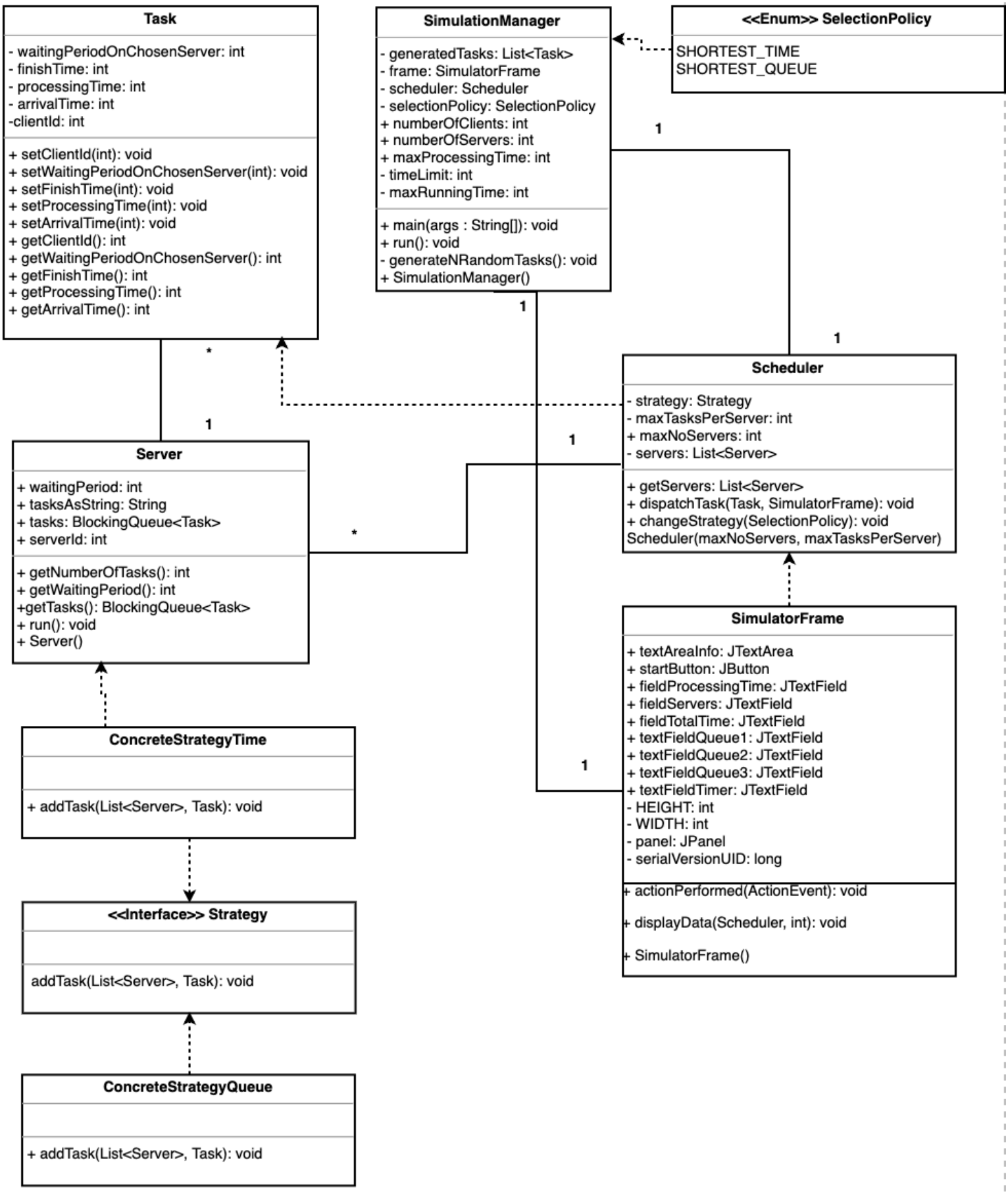
3. Design (Decisions, UML Diagrams, Class Design, User Interface)

The software design approach is OOP style. The application uses a system of interacting objects for the purpose of solving the proposed problem.

The classes are illustrated as rectangles divided into 3 sections: the first one (up) contains the name of the class, the second (middle) contains the class fields and the third (down) contains the implemented class methods.

The relationships between different classes are represented using different types of links. There are multiple ‘dependency’ relationships drawn on the diagram. For example, the Scheduler class has a dependency type relationship with the Task class because, in order for the scheduler to place the task in one of the servers, it has to know some of task’s fields.
There are also multiple ‘association’ relationships present in the diagram. For example, there is such a relationship between Scheduler and Server classes, since a scheduler can manage and run multiple servers, but because there is only one scheduler in the whole system, all servers must be run by that one scheduler. This is called a one-to-many relationship.

The UML Diagram of the system is presented on the next page.


Task
- waitingPeriodOnChosenServer: int
- finishTime: int
- processingTime: int
- arrivalTime: int
-clientId: int
+ setClientId(int): void
+ setWaitingPeriodOnChosenServer(int): void
+ setFinishTime(int): void
+ setProcessingTime(int): void
+ setArrivalTime(int): void
+ getClientId(): int
+ getWaitingPeriodOnChosenServer(): int
+ getFinishTime(): int
+ getProcessingTime(): int
+ getArrivalTime(): int
SimulationManager
- generatedTasks: List<Task>
- frame: SimulatorFrame
- scheduler: Scheduler
- selectionPolicy: SelectionPolicy
+ numberOfClients: int
+ numberOfServers: int
+ maxProcessingTime: int
- timeLimit: int
- maxRunningTime: int
+ main(args : String[]): void
+ run(): void
- generateNRandomTasks(): void
+ SimulationManager()
<<Enum>> SelectionPolicy
SHORTEST_TIME
SHORTEST_QUEUE
Scheduler
- strategy: Strategy
- maxTasksPerServer: int
+ maxNoServers: int
- servers: List<Server>
+ getServers: List<Server>
+ dispatchTask(Task, SimulatorFrame): void
+ changeStrategy(SelectionPolicy): void
Scheduler(maxNoServers, maxTasksPerServer)
Server
+ waitingPeriod: int
+ tasksAsString: String
+ tasks: BlockingQueue<Task>
+ serverId: int
+ getNumberOfTasks(): int
+ getWaitingPeriod(): int
+getTasks(): BlockingQueue<Task>
+ run(): void
+ Server()
SimulatorFrame
+ textAreaInfo: JTextArea
+ startButton: JButton
+ fieldProcessingTime: JTextField
+ fieldServers: JTextField
+ fieldTotalTime: JTextField
+ textFieldQueue1: JTextField
+ textFieldQueue2: JTextField
+ textFieldQueue3: JTextField
+ textFieldTimer: JTextField
- HEIGHT: int
- WIDTH: int
- panel: JPanel
- serialVersionUID: long
+ actionPerformed(ActionEvent): void
+ displayData(Scheduler, int): void
+ SimulatorFrame()
ConcreteStrategyTime
+ addTask(List<Server>, Task): void
<<Interface>> Strategy
addTask(List<Server>, Task): void
ConcreteStrategyQueue
+ addTask(List<Server>, Task): void
1
*
1
1
1
1
*
1

4. Implementation

The application consists of ten classes, that are used in order to reach all the secondary objectives mentioned in the first chapter of the documentation.
These classes will be presented and detailed below.

• The Task class

This is the class used to represent the client. It has fields such as arrival time, processing time and client id that are used by other classes to simulate a real client's arrival and processing time. Client id is used to represent a specific task, and only one.

Important methods implemented in this class:

- getters and setters for some of the fields, so that methods from other classes can access and modify or update the data of the client. For example, when a client is generated in the SimulationManager's generateNRandomTasks method, the arrival time and processing time need to be set into the Task class. Also, when the information about a client must be displayed in the log of events section of the user interface, some fields need to be read from classes other than Task, and for that we use the getters.

• The Server class

The multi-threading aspect of the application consists in multiple queues of clients 'running' at the same time. These queues are instances of the Server class. The most important fields are waitingPeriod, tasksAsString and tasks.

- 'waitingPeriod' represents the total waiting time of that queue (i.e. how long a client will have to wait before being processed in that queue). This is important because, when simulating a scenario where clients choose the queue with the lowest waiting time, this field is the one that gets compared with other queues' waiting times.
- 'tasksAsString' is a representation form of the clients that are currently in the queue. It is used to display the server status in the user interface, showing the client id for every task.
- 'tasks' represents the list of clients assigned to the queue. Each element of the list will be processed and then removed in the server's 'run' method.

Important methods implemented in this class:

- run: method implemented from the Runnable interface, in order to describe the behaviour of the server thread. Inside the 'run' method, all the clients in the 'tasks' BlockingQueue are processed: it takes the first one out, stops the queue for a number of seconds equal with the client's processing time, then it removes the client id from the string representation displayed in the user interface.
- Server: constructor method used to initialise waitingPeriod and tasks fields of the queue.
- getters for the class fields.

• The Scheduler class

This is the class that sets up the queues, based on the number specified by the user and then dispatches the clients from the initial client pool to all the running servers. It contains a list of all the servers and the strategy used to place clients in queues (shortest waiting time or shortest queue)

Important methods implemented in this class:

- Scheduler: constructor method used to initialise all the server threads, based on the maximum number of servers provided by the user. It also sets the server id (used for identifying the running server when needed) and starts the initialised threads.
- changeStrategy: the purpose of this method is to provide the user with a way to select the desired strategy used for placing clients in queues. When simulating queue-based systems in order to improve the service (for example, the waiting time of clients), it is very important to be able to analyse as many scenarios as possible. Those scenarios could differ, for example, in the way clients arrive at the queues: sometimes they are interested in the shortest queue, other times in the shortest waiting time. The application is built to simulate both of these situations.
- dispatchTask: this method is essential for the application to run correctly. It is used when trying to place a client from the big client pool in one of the queues, in order to decide what queue is the best at the given time. The method calls another method, addTask, implemented in the ConcreteStrategyTime and ConcreteStrategyQueue classes (it will be detailed there). This method is also used to update the log of events area in the user interface: whenever a client is added to a queue, it will be stated for the user to see.

• The SimulationManager class

The main class of the application. It is used to start the main thread, to control the global timer and to accept input from the user: timeLimit, maxProcessingTime, numberOfServers, numberOfClients, etc. It also has a field used to store all the generated clients at the beginning of the simulation: generatedTasks.

Important methods implemented in this class:

- SimulationManager: constructor method of this class, it is called in the ‘main’ method to create a new scheduler (calls the constructor that starts all the server threads), to create a new frame for the graphical user interface and to generate the number of clients specified by the user, with random arrival and processing times.
- generateNRandomTasks: method used to generate the number of clients specified by the user, with the random arrival and processing times, also in an interval specified by the user, and store them in the generatedTasks list. After generating and storing the clients, the it calls a sort method on the list of clients, with respect to their arrival time.
- run: method implemented from the Runnable interface, in order to describe the behaviour of the simulation manager thread. It sets the global timer every second, uses an iterator on the list of generated clients to dispatch them to the right server, when the global time equals that client’s queue arrival time. Then it displays all servers relevant data in the graphical user interface, so that the user can see the progress of all clients in the queues.

• The ConcreteStrategyTime and ConcreteStrategyQueue classes

These classes implement the Strategy Interface that has only one method: addTask.
It has two parameters: servers and task, and is used to decide in which server to place the given task.
The only difference between these two classes is that ConcreteStrategyTime selects the server based on its waiting time, and ConcreteStrategyQueue selects the server based on its number of clients.

- **The SimulatorFrame class**

This is the class used to build the graphical user interface and update its fields. It contains two methods:

- SimulatorFrame: constructor method used to build the whole interface. All the text fields, labels and buttons are added here, and the frame is initialised. The graphical user interface is built to display the evolution of the system queues and also to accept input from the user (even though this is not yet implemented).
- displayData: method implemented to set the text of the fields corresponding to queue status. It does this by taking the 'tasksAsString' field from each server and putting it on the user interface text fields using the 'setText' method.

5. Results

To observe the results obtained by running the application, the user is encouraged to watch and follow the evolution of all the queues, along with the log of events area, displayed at the bottom of the graphical user interface.
Implementing functionality that displays on the screen the average waiting times and peak hours is a priority in the future developments, so that the user can have a better idea of how the simulated scenario evolved.

6. Conclusions

Developing this application was very useful for me, because it taught me how to design systems better in order to create a project that could be developed and maintained not only by me, but maybe a team of developers. I have learned how threads work and how to implement multi-threading, a skill that is very useful in many applications.

Future developments:

- Implement functionality that allows the user to see average waiting times at each queue, and peak hours for the simulated scenario.
- Implement a feature that allows the user to manipulate the parameters of the simulation directly from the user interface.

7. Bibliography

- Pdf presentation document received from the lab teacher.

- Oracle documentation on BlockingQueue:
https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/BlockingQueue.html

- JavaTPoint Swing tutorials:
https://www.javatpoint.com/java-swing

- Lecture slides about UML diagrams, use-case diagrams and general application design.