DOCUMENTATION

ASSIGNMENT *ASSIGNMENT_2*

STUDENT NAME: TANASE LUISA ELENA
GROUP: 30422_2

CONTENTS

1. Assignment Objective

Main objective:

- The main objective is to design and implement an application aiming to analyze queuing-based systems by (1) simulating a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues, and (2) computing the average waiting time, average service time and peak hour .

Sub-objectives:

- Design an intuitive and user-friendly interface for users to input parameters such as number of clients (N), number of queues (Q), arrival times interval, service times interval, simulation time and selection policy.
- Develop a simulation manager that generates random clients with arrival times and service times based on specified interval limits.
- Simulate the queuing process including clients arriving, entering queues, waiting, being served, and leaving queues.
- Calculate average waiting time across all clients in the system.
- Calculate average service time for clients.
- Identify the peak hour(s) based on queue lengths or arrival rates.

2. Problem Analysis, Modeling, Scenarios, Use Cases

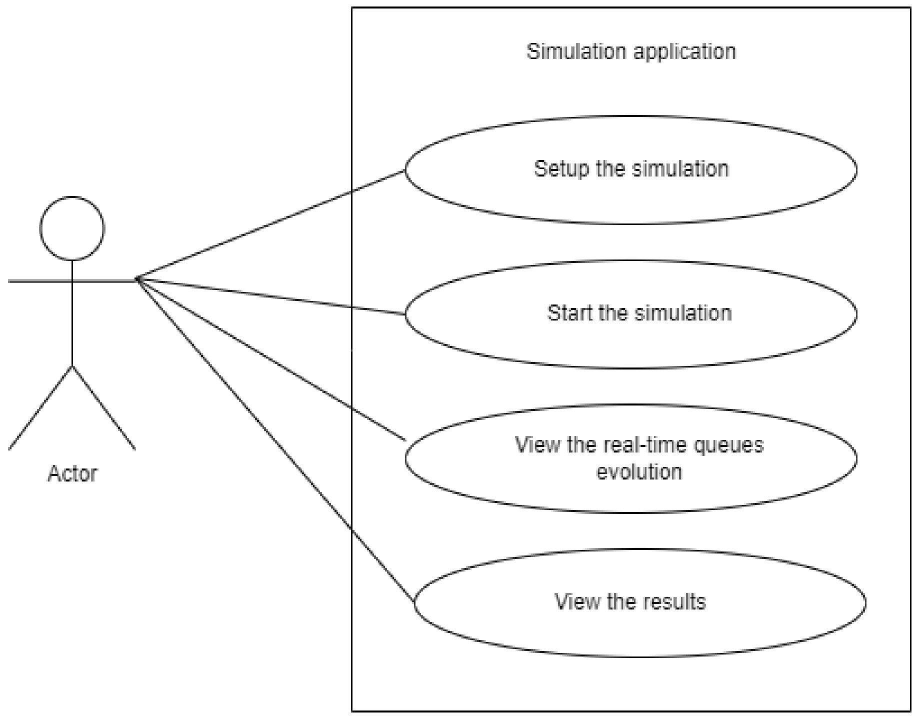
Functional Requirements:

- The simulation application should allow users to setup the simulation.
- The simulation application should allow users to start the simulation.
- The simulation application should display the real-time queues evolution
- Generate a specified number of clients (N) with random arrival times and service time requirements.
- Track the state of each client in the system (e.g., waiting in queue, being served, completed service).
- Display the queues and the current time during the simulation time interval.

- Display the average waiting period, peak hour and average service time.

Non-Functional Requirements:

- User Interface Design: The application should ensure a clean, intuitive, and responsive user interface (UI) with well-organized controls, clear labels, and meaningful feedback messages.
- Responsiveness: The application should respond quickly to user interactions such as input parameter changes, simulation control actions, and result displays.
- Scalability: Design the application to handle large simulation scenarios efficiently, considering factors such as memory usage, processing speed, and scalability of data structures.
- Code Quality: Follow coding standards, modular design principles, and documentation practices to ensure code readability, maintainability, and ease of future enhancements.



Use Case: setup the simulation
Primary Actor: user
Main Success Scenario:
1. The user inserts the number of clients, the number of queues, the arrival time interval, the service time interval, the simulation time and selects the selection policy
2. The arrival time interval and the service time interval must be inserted with the interval limits separated by comma, like writing an interval on paper.
3. If no selection policy is selected, the default one is the shortest time strategy

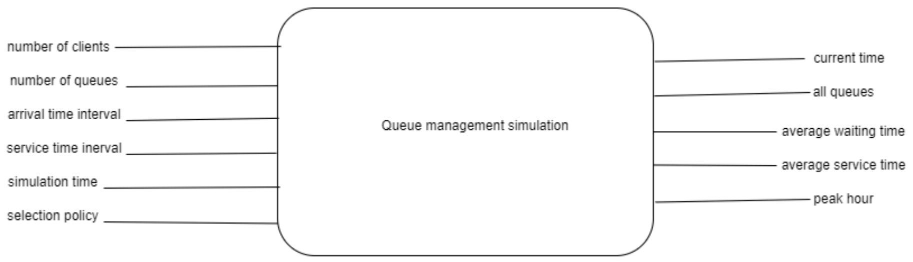
Use Case: start the simulation
Primary Actor: user
Main Success Scenario:
1. The user clicks the "start simulation" button.
2. The application will generate the clients randomly according to the simulation setup

Use Case: view the real time queues evolution
Primary Actor: user
Main Success Scenario:
1. The user can view the generated tasks and the current time and the queues evolution in each second

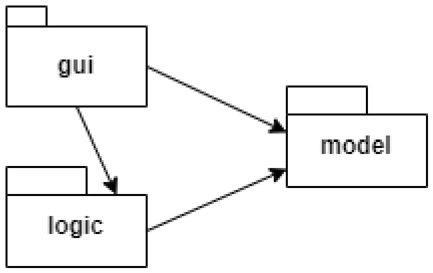
Use Case: view the results
Primary Actor: user
Main Success Scenario:
1. The user can view the average waiting time
2. The user can view the average service time
3. The user can view the peak hour

3. Design

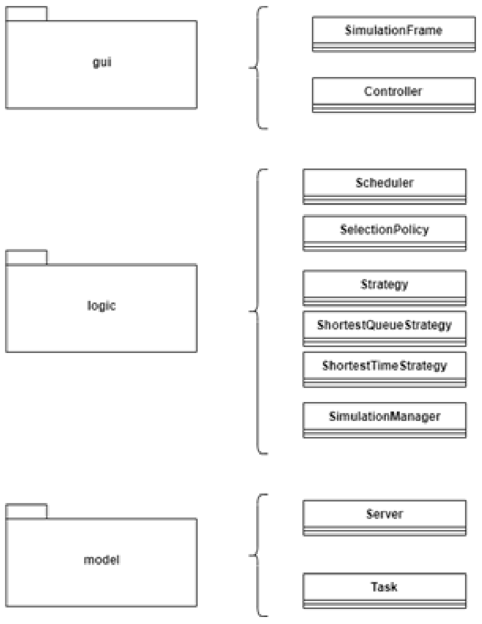
Overall System Design

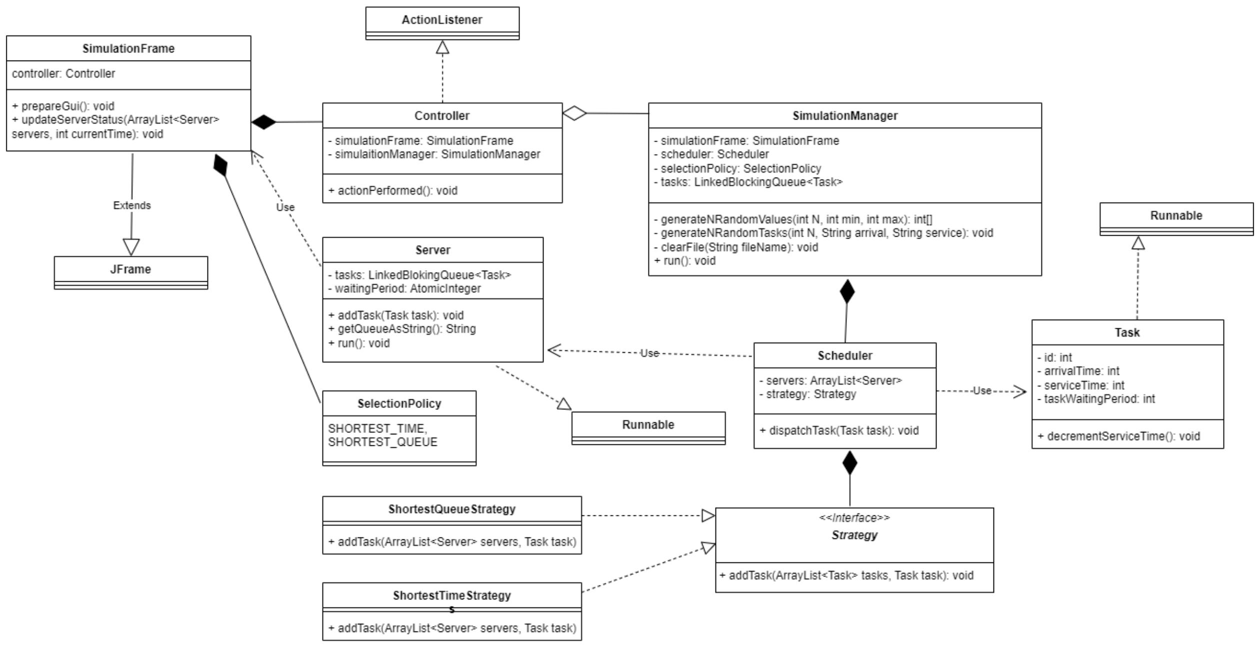


Package Diagram



Division into classes




ActionListener
SimulationFrame
controller: Controller
+ prepareGui(): void
+ updateServerStatus(ArrayList<Server> servers, int currentTime): void
Controller
- simulationFrame: SimulationFrame
- simulaitionManager: SimulationManager
+ actionPerformed(): void
SimulationManager
- simulationFrame: SimulationFrame
- scheduler: Scheduler
- selectionPolicy: SelectionPolicy
- tasks: LinkedBlockingQueue<Task>
- generateNRandomValues(int N, int min, int max): int[]
- generateNRandomTasks(int N, String arrival, String service): void
- clearFile(String fileName): void
+ run(): void
Extends
JFrame
Use
Server
- tasks: LinkedBlokingQueue<Task>
- waitingPeriod: AtomicInteger
+ addTask(Task task): void
+ getQueueAsString(): String
+ run(): void
Runnable
Task
- id: int
- arrivalTime: int
- serviceTime: int
- taskWaitingPeriod: int
+ decrementServiceTime(): void
Use
Scheduler
- servers: ArrayList<Server>
- strategy: Strategy
+ dispatchTask(Task task): void
Use
SelectionPolicy
SHORTEST_TIME,
SHORTEST_QUEUE
Runnable
ShortestQueueStrategy
+ addTask(ArrayList<Server> servers, Task task)
<<Interface>>
Strategy
+ addTask(ArrayList<Task> tasks, Task task): void
ShortestTimeStrategy
+ addTask(ArrayList<Server> servers, Task task)

Used data structures:

- In order to ensure thread safety:
  - o The tasks are stored in a LinkedBlockingQueue structure, both in the server and the simulation manager
  - o The waiting period is stored as an AtomicInteger

4. Implementation

SimulationFrame

Description:
The **SimulationFrame** class represents a GUI window for simulating queuing systems. It provides user interface components for inputting simulation parameters, starting simulations, displaying server status, and showing simulation results.

Inherits From: JFrame

Constructor:
SimulationFrame(String name):

- Constructs a new **SimulationFrame** object with the specified window name.
- Parameters:

  -name (String): Name/title of the simulation window.

Methods:

1. **prepareGui(): void**
    - Description: Initializes and configures the GUI components within the simulation window.
2. **updateServerStatus(ArrayList<Server> servers, int currentTime): void**
    - Description: Updates the server status text area with the current server queue information.
    - Parameters:
        - servers (ArrayList<Server>): List of server objects representing queues.
        - currentTime (int): Current simulation time.
3. **setGeneartedTasksLabel(String geneartedTasksLabel): void**
    - Description: Sets the text for the generated tasks label.
    - Parameters:
        - geneartedTasksLabel (String): Text to display for generated tasks information.
4. **getN(): String**
    - Description: Retrieves the inputted number of clients as a string.
    - Returns: String representation of the number of clients.
5. **getQ(): String**
    - Description: Retrieves the inputted number of queues as a string.
    - Returns: String representation of the number of queues.
6. **getArrivalTimeInterval(): String**
    - Description: Retrieves the inputted arrival time interval as a string.
    - Returns: String representation of the arrival time interval.
7. **getServiceTimeInterval(): String**
    - Description: Retrieves the inputted service time interval as a string.
    - Returns: String representation of the service time interval.
8. **getSimulationTime(): String**
    - Description: Retrieves the inputted simulation time as a string.
    - Returns: String representation of the simulation time.
9. **getSelectionPolicy(): SelectionPolicy**
    - Description: Retrieves the selected queuing selection policy from the combo box.
    - Returns: Selected **SelectionPolicy** enum value.
10. **setAverageWaitingTimeLabel(String averageWaitingTime): void**
    - Description: Sets the text for the average waiting time label.
    - Parameters:
        - averageWaitingTime (String): Text to display for average waiting time information.
11. **setPeakHourLabel(String peakHour): void**
    - Description: Sets the text for the peak hour label.
    - Parameters:
        - peakHour (String): Text to display for peak hour information.

Controller

Description:
The **Controller** class acts as an event listener for user actions in the GUI and manages the simulation process based on user inputs.

Implements: ActionListener

Constructor:

Controller(SimulationFrame simulationFrame):

- Description: Constructs a new **Controller** object with a reference to the **SimulationFrame** object.
- Parameters:
  - simulationFrame (SimulationFrame): Reference to the associated **SimulationFrame** object.

Methods:

actionPerformed(ActionEvent e): void

- Description: Handles user actions/events triggered in the GUI components.
- Parameters:
  - e (ActionEvent): Event object representing the user action.
- Behavior:
  - Retrieves the action command from the event (**e.getActionCommand()**) to determine the user action.
  - Processes the "START" action command:
    - Extracts simulation parameters (N, Q, arrival time interval, service time interval, simulation time, selection policy) from the **SimulationFrame** GUI components.
    - Initializes the **SimulationManager** with the extracted parameters and starts it in a new thread.

Scheduler

Description:
The **Scheduler** class manages the scheduling of tasks to servers based on a specified strategy, such as shortest queue or shortest time.

Constructor:

1. **Scheduler(int Q, SelectionPolicy policy, CyclicBarrier barrier):**

- Description: Constructs a new **Scheduler** object with the specified number of servers, selection policy and a CyclicBarrier to synchronize the servers.

- Parameters:
    - Q (int): Number of servers/queues to create.
    - policy (SelectionPolicy): Selection policy for task dispatching.

Methods:

1. **dispatchTask(Task task): void**
    - Description: Dispatches a task to a server based on the scheduling strategy.
    - Parameters:
        - task (Task): Task object to be dispatched.
    - Behavior:
        - Calls the appropriate strategy method (**addTask**) to dispatch the task to a server.
2. **getServers(): ArrayList<Server>**
    - Description: Retrieves the list of servers managed by the scheduler.
    - Returns: ArrayList<Server> containing the server objects.

<u>SimulationManager</u>

Description:
The **SimulationManager** class orchestrates the simulation process by generating random tasks, managing task dispatching to servers, updating simulation status, and computing simulation metrics such as average waiting time and peak hour.

Constructor:

SimulationManager(SimulationFrame simulationFrame, int N, int Q, String arrivalTimeInterval, String serviceTimeInterval, int simulationTime, SelectionPolicy policy):

- Description: Constructs a new **SimulationManager** object with simulation parameters and initializes task generation.
- Parameters:
    - simulationFrame (SimulationFrame): Reference to the simulation GUI frame.
    - N (int): Number of tasks/clients.
    - Q (int): Number of servers/queues.
    - arrivalTimeInterval (String): Interval for task arrival time (format: "min,max").

        - serviceTimeInterval (String): Interval for task service time (format: "min,max").
        - simulationTime (int): Total simulation time in seconds.
        - policy (SelectionPolicy): Selection policy for task dispatching.

Methods:

1. **run(): void**
    - Description: Executes the simulation process, including task generation, dispatching using the Scheduler and a CyclicBarrier initialized with the number of servers, status updates, and metric computations.
    - Behavior:
        - Generates random tasks based on specified arrival and service time intervals.
        - Manages task dispatching to servers using the selected scheduling strategy.
        - Updates simulation status in the GUI and logs simulation information to an output file.
        - Computes and displays average waiting time and peak hour in the GUI and output file.
2. **generateNRandomValues(int N, int min, int max): int[]**
    - Description: Generates an array of N random values within the specified range.
    - Parameters:
        - N (int): Number of random values to generate.
        - min (int): Minimum value of the range (inclusive).
        - max (int): Maximum value of the range (inclusive).
    - Returns: Array of random values.
3. **generateRandomTasks(int N, String arrivalTimeInterval, String serviceTimeInterval): void**
    - Description: Generates random tasks based on arrival and service time intervals.
    - Parameters:
        - N (int): Number of tasks to generate.
        - arrivalTimeInterval (String): Interval for task arrival time (format: "min,max").
        - serviceTimeInterval (String): Interval for task service time (format: "min,max").
4. **clearFile(String fileName): void**
    - Description: Clears the content of the specified file.
    - Parameters:
        - fileName (String): Name or path of the file to clear.

<u>Strategy</u>

Description:
The **Strategy** interface defines a common contract for strategies used in task scheduling algorithms.

Methods:

1. **addTask(ArrayList<Server> servers, Task task): void**
    - Description: Adds a task to the appropriate server(s) based on the specific scheduling strategy.
    - Parameters:
        - servers (ArrayList<Server>): List of server objects available for task dispatching.
        - task (Task): Task object to be added to a server based on the strategy.

ShortestQueueStrategy

Description:
The **ShortestQueueStrategy** class implements the **Strategy** interface to add tasks to the server with the shortest queue length.

Implements: Strategy

Methods:

1. **addTask(ArrayList<Server> servers, Task task): void**
    - Description: Implements the strategy to add a task to the server with the shortest queue length.
    - Parameters:
        - servers (ArrayList<Server>): List of server objects available for task dispatching.
        - task (Task): Task object to be added to a server based on the strategy.
    - Behavior:
        - Iterates through the list of servers to find the server with the shortest queue length.
        - Sets the task's waiting period based on the server's current waiting period.
        - Adds the task to the selected server's queue.

ShortestTimeStrategy

The **ShortestTimeStrategy** class implements the **Strategy** interface to add tasks to the server with the shortest service time.

Implements: Strategy

Methods:

1. **addTask(ArrayList<Server> servers, Task task): void**
    - Description: Implements the strategy to add a task to the server with the shortest service time.
    - Parameters:
        - servers (ArrayList<Server>): List of server objects available for task dispatching.
        - task (Task): Task object to be added to a server based on the strategy.
    - Behavior:
        - Iterates through the list of servers to find the server with the shortest waiting time.
        - Sets the task's waiting period based on the selected server's current waiting period.
        - Adds the task to the selected server's queue.

SelectionPolicy

The **SelectionPolicy** enum provides two options:
SHORTEST_TIME and **SHORTEST_QUEUE**.

1. **SHORTEST_TIME**:
    - This policy selects the server with the shortest waitng time among all available servers when assigning a task. It aims to minimize the time a task spends in the system by prioritizing servers that can process tasks quickly.
2. **SHORTEST_QUEUE**:
    - This policy selects the server with the shortest queue length among all available servers when assigning a task. It aims to balance the workload across servers by directing tasks to servers with fewer tasks in their queues, potentially reducing overall waiting times for tasks.

Server

The **Server** class models a server in a queuing-based system, capable of processing tasks asynchronously.

Fields:

1. **tasks (LinkedBlockingQueue<Task>):**
    - Description: A queue representing tasks waiting to be processed by the server.
2. **waitingPeriod (AtomicInteger):**
    - Description: An atomic integer representing the cumulative waiting time of tasks in the server's queue.

Methods:

1. **Server():**
    - Description: Constructor to initialize the server with an empty task queue and zero waiting period and the CyclicBarrier from the Scheduler.
2. **addTask(Task task): void**
    - Description: Adds a task to the server's task queue and updates the waiting period accordingly.
    - Parameters:
        - task (Task): The task to be added to the server's queue.
3. **getQueueAsString(): String**
    - Description: Returns a string representation of the tasks in the server's queue.
    - Returns:
        - String: String representation of tasks in the format "(taskId, arrivalTime, serviceTime) ".
4. **run(): void**
    - Description: Implements the server's processing logic in a separate thread. Decrements service time for tasks in the queue (task.peek()) and removes completed tasks, meaning their service time reached zero. All servers await for the barrier.
5. **getTasks(): LinkedBlockingQueue<Task>**
    - Description: Getter method to access the server's task queue.
    - Returns:
        - LinkedBlockingQueue<Task>: The task queue of the server.
6. **getWaitingPeriod(): AtomicInteger**
    - Description: Getter method to access the server's cumulative waiting period.
    - Returns:
        - AtomicInteger: The atomic integer representing the waiting period.

<u>Task</u>

Description:
The **Task** class represents a task in a queuing-based system, containing information such as task ID, arrival time, service time, and task waiting period, simulating clients for the queues.

Methods:

1. **Task(int id, int arrivalTime, int serviceTime):**
   - Description: Constructor to initialize a task with specified ID, arrival time, and service time.
   - Parameters:
     - id (int): Unique identifier for the task.
     - arrivalTime (int): Time at which the task arrives in the system.
     - serviceTime (int): Time required to process the task.
2. **decrementServiceTime(): void**
   - Description: Decrements the remaining service time of the task by one unit.
   - Behavior: Synchronized method to ensure thread-safe access and modification of service time.
3. **getId(): int**
   - Description: Getter method to retrieve the task's ID.
   - Returns:
     - int: Task ID.
4. **getArrivalTime(): int**
   - Description: Getter method to retrieve the task's arrival time.
   - Returns:
     - int: Arrival time of the task.
5. **getServiceTime(): synchronized int**
   - Description: Getter method to retrieve the remaining service time of the task.
   - Returns:
     - int: Remaining service time of the task.
   - Notes: Method is synchronized for thread-safe access.
6. **getTaskWaitingPeriod(): int**
   - Description: Getter method to retrieve the task's waiting period.
   - Returns:
     - int: Task's waiting period in the system.
7. **setTaskWaitingPeriod(int taskWaitingPeriod): void**
   - Description: Setter method to update the task's waiting period.
   - Parameters:
     - taskWaitingPeriod (int): New waiting period value to set for the task.

5. Results

Test1

Inputs:

N = 4
Q = 2
Simulation time = 60
Arrival time interval = 2,30
Service time interval = 2,4

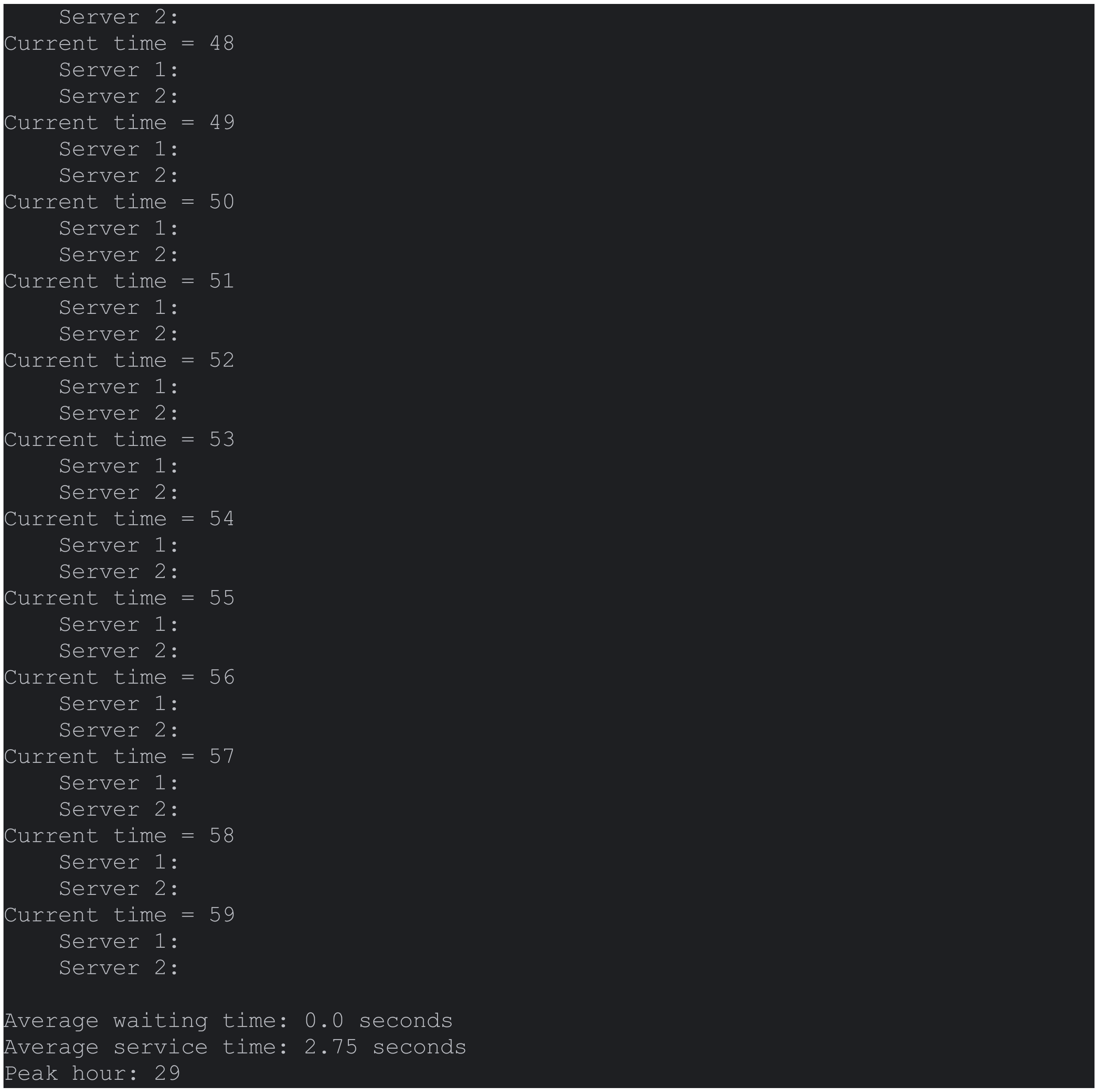
Results from the log file:

```
Generated tasks:
(1,13,3)
(2,19,4)
(3,24,2)
(4,29,2)

Current time = 0
    Server 1:
    Server 2:
Current time = 1
    Server 1:
    Server 2:
Current time = 2
    Server 1:
    Server 2:
Current time = 3
    Server 1:
    Server 2:
Current time = 4
    Server 1:
    Server 2:
Current time = 5
    Server 1:
    Server 2:
Current time = 6
    Server 1:
    Server 2:
Current time = 7
    Server 1:
    Server 2:
Current time = 8
    Server 1:
    Server 2:
Current time = 9
    Server 1:
    Server 2:
Current time = 10
    Server 1:
    Server 2:
```

```
Current time = 11
    Server 1:
    Server 2:
Current time = 12
    Server 1:
    Server 2:
Current time = 13
    Server 1: (1,13,3)
    Server 2:
Current time = 14
    Server 1: (1,13,2)
    Server 2:
Current time = 15
    Server 1: (1,13,1)
    Server 2:
Current time = 16
    Server 1:
    Server 2:
Current time = 17
    Server 1:
    Server 2:
Current time = 18
    Server 1:
    Server 2:
Current time = 19
    Server 1: (2,19,4)
    Server 2:
Current time = 20
    Server 1: (2,19,3)
    Server 2:
Current time = 21
    Server 1: (2,19,2)
    Server 2:
Current time = 22
    Server 1: (2,19,1)
    Server 2:
Current time = 23
    Server 1:
    Server 2:
Current time = 24
    Server 1: (3,24,2)
    Server 2:
Current time = 25
    Server 1: (3,24,1)
    Server 2:
Current time = 26
    Server 1:
    Server 2:
Current time = 27
    Server 1:
    Server 2:
Current time = 28
    Server 1:
    Server 2:
Current time = 29
```

```
    Server 1: (4,29,2)
    Server 2:
Current time = 30
    Server 1: (4,29,1)
    Server 2:
Current time = 31
    Server 1:
    Server 2:
Current time = 32
    Server 1:
    Server 2:
Current time = 33
    Server 1:
    Server 2:
Current time = 34
    Server 1:
    Server 2:
Current time = 35
    Server 1:
    Server 2:
Current time = 36
    Server 1:
    Server 2:
Current time = 37
    Server 1:
    Server 2:
Current time = 38
    Server 1:
    Server 2:
Current time = 39
    Server 1:
    Server 2:
Current time = 40
    Server 1:
    Server 2:
Current time = 41
    Server 1:
    Server 2:
Current time = 42
    Server 1:
    Server 2:
Current time = 43
    Server 1:
    Server 2:
Current time = 44
    Server 1:
    Server 2:
Current time = 45
    Server 1:
    Server 2:
Current time = 46
    Server 1:
    Server 2:
Current time = 47
    Server 1:
```



```
    Server 2:
Current time = 48
    Server 1:
    Server 2:
Current time = 49
    Server 1:
    Server 2:
Current time = 50
    Server 1:
    Server 2:
Current time = 51
    Server 1:
    Server 2:
Current time = 52
    Server 1:
    Server 2:
Current time = 53
    Server 1:
    Server 2:
Current time = 54
    Server 1:
    Server 2:
Current time = 55
    Server 1:
    Server 2:
Current time = 56
    Server 1:
    Server 2:
Current time = 57
    Server 1:
    Server 2:
Current time = 58
    Server 1:
    Server 2:
Current time = 59
    Server 1:
    Server 2:

Average waiting time: 0.0 seconds
Average service time: 2.75 seconds
Peak hour: 29
```

6. Conclusions

<u>What I've Learned</u>

GUI Design: Designing an intuitive and user-friendly GUI is crucial for simulation applications to allow users to input parameters and visualize simulation results effectively.

Queuing Systems: Understanding the basics of queuing theory, including concepts such as arrival time, service time, waiting time, and queue management strategies like shortest queue and shortest time, is essential for simulating queuing-based systems.

Concurrency: Handling concurrent tasks and threads, such as simulating servers processing tasks concurrently, requires careful synchronization and management to ensure thread safety and accurate simulation results.

<u>Future Developments</u>

Enhanced Simulation Features: Implementing more advanced queuing strategies.

Visualization Tools: Integrating graphical representations, charts, and real-time updates within the GUI can enhance user experience and provide interactive visualization of simulation results.

7. Bibliography

1. *https://dsrl.eu/courses/pt/materials/PT_2024_A2_S1.pdf*