Modeling a Server Farm

Muneeb Azfar Nafees

March 2025

Contents

1 Abstract

Efficient job scheduling is crucial for modern computing systems, such as large-scale server farms and microprocessor environments that handle vast amounts of data. In this project, I simulated a multi-server environment in Java using a *linked-list-based FIFO queue* to model how jobs are processed and dispatched through four different strategies: **random, round-robin, shortest queue, and least work**. This simulation employed the core data structure *"queue"* to ensure that jobs were processed in the order they arrived, allowing for an accurate evaluation of each dispatch strategy's performance. The results indicated that the **LeastWorkDispatcher** achieved the lowest average waiting times. Additionally, a critical threshold was observed with the **ShortestQueueDispatcher** at around 34 servers for $10,000,000$ jobs, beyond which adding more servers resulted in only marginal improvements.

2 Results

2.1 Modifications and Commands

To streamline experimentation, I updated the `ServerFarmSimulation` file so that it automatically runs all required experiments in a single execution and prints the results in a nicely formatted way. Rather than requiring multiple runs, the simulation is now set up to internally execute two experiments:

1. **Experiment 1 (Dispatcher Comparison):** The simulation runs for four dispatcher types (random, round-robin, least work, and shortest queue) using a fixed configuration of 34 servers and 10,000,000 jobs.
2. **Experiment 2 (Server Count Impact - ShortestQueueDispatcher):** The simulation runs with the ShortestQueueDispatcher while varying the number of servers from 30 to 40, each with 10,000,000 jobs.

Each simulation run automatically resets the state (job maker and dispatcher) to ensure that the results are independent and reproducible, and the outcomes are printed out in a clear, formatted manner in one execution.

2.2 Experiment 1: Dispatcher Comparison

This experiment compared the performance of four different job dispatchers under the following settings:

- **Number of Servers:** 34
- **Number of Jobs:** 10,000,000
- **Mean Arrival Time:** 3
- **Mean Processing Time:** 100

My hypothesis was that the **LeastWorkDispatcher** would yield the best performance (i.e., the lowest average waiting time) since it assigns jobs to the server with the least remaining work, thereby balancing the load more effectively. In contrast, the **RandomDispatcher** was expected to perform the worst due to its arbitrary assignment of jobs.

After running the simulation 10 times for each dispatcher, the average waiting times obtained were as follows:

Dispatcher	Average Waiting Time (units)
RandomDispatcher	5496.22
RoundRobinDispatcher	2810.14
ShortestQueueDispatcher	277.55
LeastWorkDispatcher	235.29

Table 1: Average waiting times for each dispatcher using 34 servers and 10,000,000 jobs.

The results confirm the hypothesis: the **LeastWorkDispatcher** performs best, while the **RandomDispatcher** performs worst.

2.3 Experiment 2: Impact of Server Count on Waiting Time (ShortestQueueDispatcher)

The second experiment focused on the **ShortestQueueDispatcher** to study the effect of varying the number of servers on the average waiting time. The simulation was run with 10,000,000 jobs, a mean arrival time of 3, and a mean processing time of 100, while the number of servers was varied from 30 to 40.

The average waiting times recorded for each server count are summarized in the table below:

Number of Servers	Average Waiting Time (units)
30	1,686,780.48
31	1,140,171.88
32	645,373.58
33	173,014.27
34	279.54
35	166.22
36	138.72
37	125.36
38	117.24
39	111.98
40	108.41

Table 2: Average waiting times for the ShortestQueueDispatcher with varying number of servers.

The graph below displays the trend of average waiting time versus the number of servers. Notice the dramatic drop between 33 and 34 servers, suggesting that at least 34 servers are required to handle the load effectively. Additional servers beyond 34 continue to reduce the waiting time, but with diminishing returns.

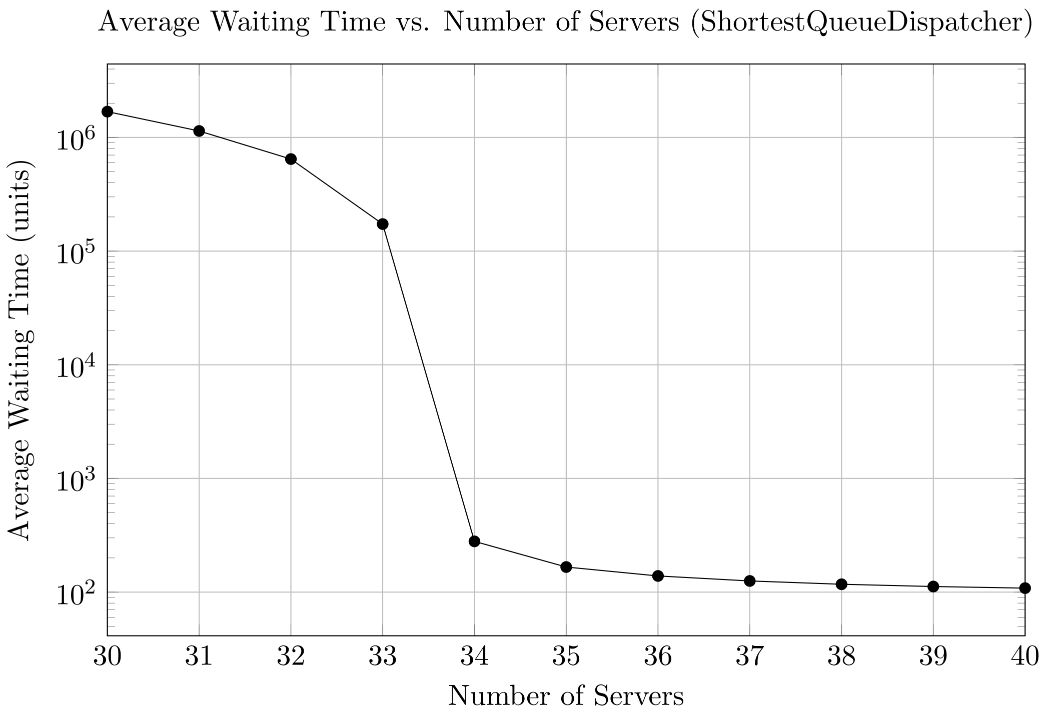

Figure 1: Average waiting time versus number of servers for the ShortestQueueDispatcher (logarithmic scale on the y-axis).

2.4 Conclusions

The experiments yield several important insights:

- **Dispatcher Comparison:** The **LeastWorkDispatcher** achieved the lowest average waiting time, confirming that allocating jobs to the server with the least remaining work optimizes load distribution. Conversely, the **RandomDispatcher** performed the worst due to its arbitrary assignment of jobs.
- **Server Count Impact:** There is a critical threshold at around 34 servers for the **ShortestQueueDispatcher** when processing 10,000,000 jobs; the waiting time drops significantly. Below 34 servers, the system falls far behind in processing jobs, while increasing the number of servers beyond this point results in smaller incremental improvements. Note that this threshold is specific to the scenario with 10,000,000 jobs, and may shift when processing a different number of jobs.

3 Extensions

3.1 Extension 1: Priority Queue

3.1.1 What did you do and why?

In this extension, I modified the server's job queue from a standard FIFO (first-in, first-out) structure to a priority queue. The objective was to explore how changing the job processing order, by prioritizing jobs with higher processing times, affects overall system performance and fairness.

The goal was to determine if processing jobs with higher processing times first would improve efficiency or lead to trade-offs in fairness. I altered the underlying queue structure in the `PriorityServer` class to insert jobs in order based on their required processing time. This approach contrasts with the FIFO method, as it ensures that the job with the highest processing time is processed first.

3.1.2 What was the outcome?

The performance of the priority queue implementation was compared against the baseline FIFO-based simulation. The average waiting times over 10 simulations for 34 servers and 100,000 jobs are summarized in Table below:

Dispatcher	**FIFO Avg. Wait Time (units)**	**Priority Queue Avg. Wait Time (units)**
Random	2910.12	16157.80
Round-Robin	1793.49	9975.24
Least Work	221.09	1158.82
Shortest Queue	287.67	1305.67

Table 3: Average waiting times for 34 servers and 100,000 jobs over 10 simulations.

These results indicate that while the priority queue strategy might offer improved processing for jobs with longer processing times, it leads to significantly increased overall waiting times. This outcome was expected, as prioritizing longer jobs delays the processing of shorter ones, thus compromising fairness.

Note that while this O(n) priority queue approach works for 100,000 jobs, its linear time complexity suggests that it may not scale well to very large numbers of operations.

3.1.3 How can you replicate my outcome in my code-base?

To replicate this extension in my code-base, ensure that the following files are compiled in the `extension` directory:

- `PriorityServer.java` – implements the priority-based queue.
- `PriorityJobDispatcher.java` – serves as the abstract dispatcher for priority-based servers.

- `PriorityRandomDispatcher.java`, `PriorityRoundRobinDispatcher.java`, `PriorityLeastWorkDispatcher.java`, `PriorityShortestQueueDispatcher.java` – implement the various dispatch strategies for the priority system.
- `PriorityServerFarmSimulation.java` – the main simulation class that runs the experiments without visualization.

After compiling, run the simulation using the command:

```
java PriorityServerFarmSimulation
```

To obtain comparable results for the FIFO simulation, update the `ServerFarmSimulation` file in the `src` directory to set the number of jobs to 100,000 (instead of 10,000,000). Then, run the file using the command:

```
java ServerFarmSimulation
```

We use the Experiment 1 results from the output as these provide the necessary comparison between the priority queue implementation and the baseline FIFO model.

3.2 Extension 2: Hybrid Dispatcher

3.2.1 What did you do and why?

Since the LeastWorkDispatcher and ShortestQueueDispatcher performed best during the exploration phase, I combined their approaches to create a weighted Hybrid Dispatcher. In this extension, I modified the `HybridQueueDispatcher` to compute a composite metric for each server by assigning weights to both the server's queue size (representing the ShortestQueue approach) and its remaining processing time (representing the LeastWork approach). I experimented with different weight combinations to determine which balance minimizes the average waiting time while maintaining fairness.

3.2.2 What was the outcome?

I ran 10 simulations for each weight configuration using 34 servers and 10,000,000 jobs. The results are summarized in Table 4 below.

Weights (Least, Shortest)	**Avg. Wait Time (units)**
(1.0, 0.0)	239.6112
(0.0, 1.0)	275.1964
(0.5, 0.5)	258.5256
(0.4, 0.6)	239.7531
(0.3, 0.7)	245.7589
(0.2, 0.8)	244.8476
(0.1, 0.9)	242.6356
(0.6, 0.4)	236.2114
(0.7, 0.3)	244.9774
(0.8, 0.2)	238.5436
(0.9, 0.1)	250.1555

Table 4: Average waiting times for the Hybrid Dispatcher with various weight configurations (34 servers, 10,000,000 jobs, averaged over 10 simulations).

The results indicate that the best performance was achieved with weights of 0.6 for the LeastWork component and 0.4 for the ShortestQueue component, yielding an average waiting time of approximately 236.21 units. In contrast, placing all the weight on either metric resulted in higher waiting times (239.61 units for (1.0, 0.0) and 275.20 units for (0.0, 1.0)). Overall, these outcomes suggest that while a balanced approach can improve performance, emphasizing the LeastWork metric slightly more than the ShortestQueue metric appears optimal in this scenario.

3.2.3 How can you replicate my outcome in my code-base?

To replicate this extension in your code-base, ensure that the following files are compiled in the `extension` directory:

- `HybridQueueDispatcher.java` – implements the composite metric and weighted selection logic.
- `ServerFarmSimulation.java` – which includes the branch for the "hybrid" dispatcher.

After compiling, run the simulation in the `extension` directory using the command:

```
java ServerFarmSimulation 34 10000000 hybrid
```

Then, adjust the weights inside the `HybridQueueDispatcher` class to test different combinations as shown in Table 4. The average waiting times can be obtained by running the simulation multiple times and averaging the results.

4 Acknowledgments

1. What is Priority Queue | Introduction to Priority Queue - GeekforGeeks
2. TA: Rishit Chatterjee and Aayan Shah