Modeling Multi-Server Job Dispatching

Abstract

This project models how computing jobs are distributed and processed across multiple servers, simulating the dynamics of real-world data centers and cloud systems. Using a Queue data structure implemented with a LinkedList, I developed and compared Random, Round Robin, Shortest Queue, and Least Work job dispatchers to study how different strategies affect average job waiting time. The simulation showed that load-aware dispatchers, which consider current queue lengths or remaining work, vastly outperform simple or cyclic assignment methods, reducing average wait time by more than twentyfold. Moreover, scaling experiments revealed a critical capacity threshold: at around 34 servers, system performance transitions sharply from overloaded to stable, after which additional servers bring only marginal gains. These results highlight how intelligent scheduling and optimal provisioning can dramatically improve efficiency in multi-server computing systems.

Results

- **Experiment**

To evaluate multi-server scheduling in a simulated server farm, I ran two experiments using the same job generation process: exponentially distributed interarrival times with meanArrivalTime = 3 and exponentially distributed processing times with meanProcessingTime = 100. In Experiment 1, I fixed the number of servers at 34 and processed 10,000,000 jobs, comparing four dispatchers: Random, Round Robin, Shortest Queue, and Least Work. In Experiment 2, I fixed the dispatcher to Shortest Queue, kept jobs at 10000000, and swept the number of servers from 30 to 40. The primary metric in all runs was average waiting time per job (total time in system from arrival to completion).

- **Hypothesis**

I hypothesized that Shortest Queue and Least Work dispatchers, which are load-aware, would substantially reduce waiting time relative to Random and Round Robin dispatchers, which are load-agnostic, because they react to instantaneous system imbalance. I also expected a tipping point in server count. Severe congestion for undersized farms will be followed by a sharp drop in delay once capacity crosses demand, and then diminishing returns beyond that point.

- **Table of results**

Table 1. Average waiting time vs. dispatcher (34 servers, 10,000,000 jobs).

Dispatcher	Average Waiting Time	Servers	Jobs
Random	5084.0653	34	10000000
Round Robin	2837.0008	34	10000000
Shortest Queue	274.4699	34	10000000
Least Work	233.8240	34	10000000

Shortest Queue and Least Work dispatchers cut average waiting time by ~10–20 times compared to Random and Round Robin. We can see the metric drops from thousands of time units (Random: 5,084; Round Robin: 2,837) to a few hundred (Shortest: 274; Least: 234). Thus, Shortest Queue and Least Work dispatchers work the best, and Random and Round Robin work worst, which is what I expected. This implies that dispatch policies that measure queue length or remaining work dramatically mitigate congestion under heavy load, mirroring real load balancers in data centers. The table above can be found in results.txt in the src folder.

Figure 1. Average waiting time vs. number of servers (Shortest Queue Dispatcher, 10,000,000 jobs).

Servers	Average Waiting Time	Average Waiting Time in ASCII Bar Graph
30	1,695,944.47	######################## ######################## ######################## ########
31	1,152,545.21	######################## ######################## ######################## ######
32	649,407.55	######################## ######################## ######################## ###
33	174,895.99	######################## ######################## ###################
34	273.51	######################## #######
35	166.75	######################## #####
36	139.54	######################## ####
37	125.49	######################## ###
38	116.85	######################## ###
39	112.04	######################## ##
40	108.49	######################## ##

Waiting time plummets from over 1.6 million (30 servers) to just ≈274 (34 servers), then decreases slowly toward 108 at 40 servers. This figure implies the system experiences a sharp capacity threshold near 34 servers, beyond which the benefit of adding more servers becomes marginal, which is what I expected and typical of saturation behavior in queueing systems. The original graph data can be found in shortest_queue_results.txt in the src folder.

- **Synthesis**

The two experiments validate both hypotheses. In Experiment 1, the Least Work Dispatcher achieved the lowest average waiting time, closely followed by Shortest Queue. In contrast, Random and Round Robin dispatchers produced waits more than ten times longer, confirming that awareness of system state is critical for throughput. Experiment 2 revealed a sharp non-linear threshold in performance when the system had fewer than 34 servers; average waiting times exploded, which is evidence of overload. At 34 servers, waiting time collapsed by several orders of magnitude, marking the transition to stable operation. Beyond 35 servers, improvements tapered off, indicating diminishing returns as utilization dropped below capacity. These outcomes mirror that smarter scheduling policies can substitute for costly hardware expansion, and understanding where utilization crosses the overload cliff allows system architects to provision just enough resources for efficiency. Intelligent dispatching, not merely raw capacity, is the decisive factor in scalable computing.

Acknowledgements

I would like to thank Professor Harper and Professor Al Madi for providing the project template and guidance site, as well as the TAs for guiding me through the debugging process. Thanks to Oracle and W3Schools Java Tutorial for the detailed documentation, and I learned to use other methods (like java.io.Map) when exporting data to txt files.

Extensions:

What

In the project, the four standard dispatchers (Random, Round Robin, Shortest Queue, and Least Work) determined how incoming jobs were assigned to servers in the simulated server farm. Each focused on simple metrics such as queue length or total remaining work, but ignored how job size affects waiting time. In real systems, long jobs can block many short ones, creating head-of-line delays that simple least-work balancing cannot prevent.

For my extension, I implemented a new class called SlowdownDispatcher, which routes each incoming job to the server that minimizes its expected slowdown rather than raw queue length or remaining work. The slowdown score is computed as score = 1+workAhead/jobSize + a*queueLength, where a is a is a small penalty (set to 0.1) to discourage joining long queues, workAhead is the server's remaining workload, and jobSize is the job's required processing time. This formula favors servers that let small jobs complete quickly while still balancing overall utilization. The design is inspired by queueing-theory results showing that minimizing expected slowdown can improve fairness and average response time in heterogeneous workloads.

Outcome

I ran the same large-scale simulation framework used for the original experiments, testing the new Slowdown dispatcher alongside the Least Work and Shortest Queue baselines. Each configuration processed 1,000,000 jobs with meanArrivalTime = 3 and meanProcessingTime = 100, varying the number of servers from 30 to 41.

Table 2. Average waiting times for three dispatchers across different server counts.

Servers	LeastWork	Slowdown	ShortestQueue
30	168746.06	154985.56	168769.47
31	115076.61	105691.84	115098.68
32	64763.69	59495.36	64811.53
33	17721.98	16306.74	17767.32
34	246.99	237.96	277.75
35	144.95	142.60	167.78
36	121.05	120.13	138.44
37	111.77	111.34	124.85
38	107.14	106.91	116.86
39	104.46	104.33	111.71
40	102.82	102.75	108.26
41	101.78	101.75	105.78

The SlowdownDispatcher consistently achieved the lowest waiting time across all configurations, especially near the system's congestion threshold (30–33 servers), where waiting times dropped by roughly 8–10% compared to the Least Work dispatcher. At higher server counts, all methods converged because the system had ample capacity. These results confirm that slowdown-based routing better prioritizes small jobs during periods of high load, smoothing out the queueing imbalance that occurs under purely workload-based dispatch. Therefore, the SlowdownDispatcher beat the LeastWork dispatcher. The table data is stored in the txt file under the ext folder.

How to run

Compile and run ExperimentRunner.java under the ext folder. A new experiment_results txt file will be generated. If you want to adjust the value a (slowdown penalty), change the parameter in the constructor SlowdownDispatcher(int k, boolean showViz) of SlowdownDispatcher.java. Increasing α gives more weight to queue length; decreasing it makes the dispatcher behave more like Least Work.