Modeling a Server Farm Using Java

Abstract:

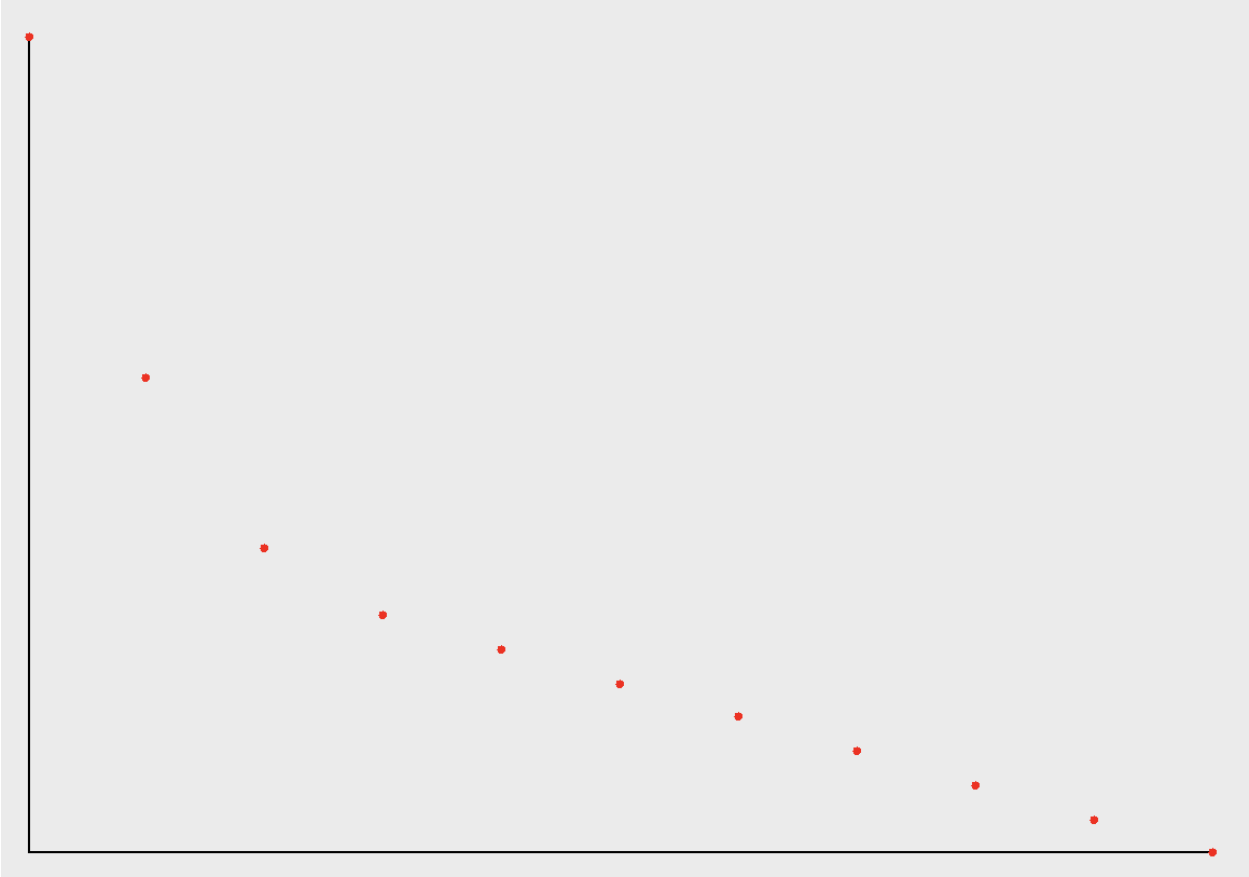
The real world problem I explored in this project was the optimization of server farms, efficient job dispatching, and optimizing resources. To better understand the problem in this project, I used different job dispatching strategies, like round robin, random, shortest queue, and least work. I also tested the different number of servers being used. The core CS concepts I used in the project were data structures like queues and linked lists, algorithms like different job dispatchers, and classes and inheritance which helped to optimize the server farm. My key findings from the project were that the parameters can have a large impact on the outcome of the problem even with small changes, and showed the potential for optimization in the project depending on different factors.

Results:

I ran experiments with different versions of the code including different job dispatchers like round robin, shortest queue, and least work. I also tried different server code, like the preemptive server. The metric that I reported in order to understand the outcomes of my experiments were the wait times of the jobs (on average). I also looked at the number of servers and the type of dispatcher used in the experiments to see how the wait times were affected.

Dispatcher	Avg wait time
random	-4.997240678491064E8
round	-5.000172792005124E8
shortest	-4.999922007748767E8
least	-4.999763130054177E8

Required result 1: The round dispatcher works the best because it has the shortest wait time out of all the dispatchers. The random dispatcher works the worst because it has a longer wait time than all the dispatchers, but it is barely significant, and there isn't much of a difference between the random, shortest, and least dispatchers. I expected the shortest dispatcher to work the best because the jobs being done by the servers with shorter wait times makes more sense to be completed faster to me.



Required result 2: This matches my expectations because as more servers got added, the job waiting times decreased. Having more servers means that there is more distribution in terms of the work there is to complete.

The outcome of my experiments showed that there were differences in the job waiting times based on the dispatcher type, although they weren't very significant. Out of these, the round robin dispatcher was the most efficient, with the lowest average waiting time. Increasing the number of servers also helped improve efficiency, and reduced the average wait times.

Extensions:

For this extension, I explored a different objective, which was to minimize the average ratio of job size to its wait time instead of minimizing average time in the system. To do this, I made changes to the JobDispatcher class and added a new method to get the average ratio of the job size to the wait time. The outcome was that the jobs would spend less time waiting now, which also means that smaller jobs were likely finished first. To replicate the outcome in my code base, you would have to add a new method to the JobDispatcher class that did the calculations and then made sure they were implemented correctly through the rest of the program.

Reflection:

1. They are the same because they both have the same impacts on the performance of the program. They both have the same efficiency because the Jobs are assigned based on the Server workload so it doesn't matter whether each Server has its own Queue or a singular Queue.
2. After using the PreemptiveServer class instead of the Server class, the wait time improved a lot. The results matched my expectations because I expected it to do better since it was meant to process the jobs remaining more quickly.

Acknowledgements:

I worked with Kamalani on parts of the project. I also got help from my mom and some of the TAs in office hours.