# CS231 Project 5: Modeling a Server Farm

October 16th, 2023

## Abstract

The main purpose of this project is to use our Queue structure to model job division in a multi-server (or processor) setting. Typically, the model has jobs arriving over time and upon arrival must immediately be assigned to a server for processing. Each server maintains its own queue of jobs for processing and operates under the rule that it must process jobs in the order of their arrival.

---

## Results

### Average Waiting Time of running jobs with varying parameters between 30 Servers

| Average waiting time (time units) | A Job arrives every 7 time units and requires 30 total units of processing time | A Job arrives every 10 time units and requires 100 total units of processing time | A Job arrives every 3 time units and requires 100 total units of processing time |
|---|---|---|---|
| RandomDispatcher | **35.0000701** | **149.93420478684095** | **1683354.976256** |
| RoundRobinDispatcher | **30.0533724** | **107.42964794296479** | **1682826.2547456** |
| ShortestQueueDispatcher | **29.9966049** | **99.97491759749175** | **1682283.0497792** |
| LeastWorkDispatcher | **29.996606** | **99.974939997494** | **1682189.4578176** |

The average waiting time is determined by how long a job waits within a queue to be completed, including the time spent waiting for itself to be on the job of the queue, and the time it takes to be completed. For the first 2 simulations, where the amount of work that comes in per unit of time is less than the servers available, jobs are basically completed as soon as they enter, with some error caused by the Dispatcher methods' assignments like in the Random and RoundRobin. For example, in the scenario where a job arrives every 10 time units and requires 100 total units of processing time, it takes 10 jobs to be imported by the time the very first is completed. If there are at least 10 servers to complete each job that comes in, then there wouldn't be any reason to stall. In algorithms such as ShortestQueue and LeastWork, this helps increase work efficiency ten fold as havi

ng 30 servers ensures that there will always be someone to work on a job when it comes in. Conversely, in the scenario where a job arrives every 3 time units and requires 100 total units of processing time, there will be too much work added to every queue no matter how well it is optimized, leading to an eventual rise in waiting time as shown.

---

## Exploration

### Which of the four dispatch methods seems best for this objective?

Given the provided scenarios with varying arrival rates and job processing times, the LeastWorkDispatcher seems to be the most promising choice. This dispatcher optimizes the usage of servers by prioritizing jobs based on their remaining processing time. In situations where job sizes and processing times vary widely, this method can lead to efficient use of resources, potentially reducing the average waiting time for jobs.

However, it's crucial to note that the best dispatcher can vary based on the specific characteristics of the workload, the nature of the tasks, and the goals of the system. For a definitive answer, it's important to conduct detailed experiments and analyze the results under various conditions to determine the most suitable dispatcher for your particular use case.

---

## Reflection

### What are the runtimes of the Queue methods within your LinkedList? Specifically, what are the runtimes of the peek, poll, and offer methods?

peek(): This method retrieves, but does not remove, the head of the queue.

Runtime: Constant time, O(1). It always takes the same amount of time to retrieve the first element of a linked list regardless of its size.

poll(): This method retrieves and removes the head of the queue.

Runtime: Constant time, O(1). Similar to peek(), removing the first element from a linked list takes constant time.

offer(E element): This method adds the specified element to the end of the queue.

Runtime: Constant time, O(1). Adding an element to the end of a linked list (assuming you keep a reference to the tail) is a constant-time operation.

## Some of the required dispatch methods pick the Server asymptotically faster in the worst case than others: specifically, two of the methods take constant time, and two of them take longer. How do the contant time methods compare to those that take longer?

When it comes to dispatch methods that pick the server asymptotically faster in the worst case:

**Constant Time Methods** (RandomDispatcher and RoundRobinDispatcher):

Runtime: Constant time, O(1). Both RandomDispatcher and RoundRobinDispatcher select servers in constant time regardless of the number of servers.

**Linear Time Methods** (ShortestQueueDispatcher and LeastWorkDispatcher):

Runtime: Linear time, O(n). Both ShortestQueueDispatcher and LeastWorkDispatcher iterate through the list of servers to find the appropriate server, making their selection time proportional to the number of servers in the list (n).

Comparing the constant time methods to those that take longer (linear time methods):

Constant Time Methods:

- Advantages: These methods are highly efficient and provide quick server selection, ensuring fast job dispatching regardless of the number of servers.
- Limitations: They might lack sophistication in selecting servers based on server workload or job characteristics, potentially leading to suboptimal job assignments in certain scenarios.

Linear Time Methods:

- Advantages: These methods consider server queue length or amount of work, allowing for more intelligent server selection. They can optimize server usage and potentially reduce overall system waiting times.
- Limitations: The runtime increases linearly with the number of servers, which could impact performance in systems with a large number of servers.

In summary, constant time methods provide quick server selection but might lack optimization based on server workload. On the other hand, linear time methods offer more intelligent server selection but could experience performance challenges in systems with a high number of servers. The choice between them depends on the specific requirements and constraints of the application.

---

## Extensions

### Given a job sequence where on average jobs arrive every x seconds and take y total units of processing time, what seems like a minimal number of Servers necessary to handle those jobs without falling uncontrollably far behind? Does your best dispatcher support this intuition?

Determining the minimal number of servers necessary to handle a job sequence without falling uncontrollably far behind depends on the specific characteristics of the job sequence, including the average arrival rate (x) and the total processing time per job (y). Here are some key considerations and factors that can influence the minimum number of servers needed:

1. Job Arrival Rate (x):

A higher arrival rate (x) means jobs are arriving more frequently, requiring servers to process jobs quickly to keep up with the demand. A low arrival rate allows servers more time to process each job, reducing the urgency for immediate processing.

2. Total Processing Time per Job (y):

Longer processing times (y) mean each job takes more time to complete. If jobs have high processing times, it may require more servers to handle the incoming workload efficiently.

3. Dispatcher Efficiency:

The efficiency of the dispatcher algorithm in assigning jobs to servers can significantly impact the number of servers needed. An intelligent dispatcher, such as the LeastWorkDispatcher or an optimized version, can make better decisions and utilize server resources more effectively, potentially reducing the number of servers required.

In general, the intuition suggests that if jobs arrive frequently (high x) and have short processing times (low y), a higher number of servers might be necessary to prevent falling significantly behind. On the other hand, if jobs arrive infrequently (low x) or have long processing times (high y), fewer servers might be sufficient to handle the workload without falling far behind.

The LeastWorkDispatcher supports this intuition to some extent, as it aims to distribute jobs based on the remaining processing time, helping balance the workload among servers. However, it's important to note that even the best dispatcher can only optimize job assignments based on the available server resources and job characteristics. If the incoming workload is exceptionally high or the processing times are very short, additional servers might still be necessary to prevent falling uncontrollably far behind. Additionally, considering burstiness in job arrivals and real-time requirements is crucial in determining the optimal server configuration for a given workload.

## In our project so far, all Servers have been the same. If some of my servers were faster than others, how should I use them?

If one of your servers, for example works twice as fast as the others, you can certainly leverage this speed difference to optimize your job processing using the LeastWorkDispatcher.

Here's how you can adapt the LeastWorkDispatcher to incorporate the speed difference:

Modify the calculation of remaining processing time for each job based on the server's speed. If a server is twice as fast, the remaining processing time for jobs in its queue would be halved compared to other servers. (getTimeRemaining())

In your LeastWorkDispatcher, when selecting the server with the least remaining processing time, consider the server's speed as well. The LeastWorkDispatcher should compare the adjusted remaining processing times across servers. Servers that are faster will have shorter adjusted remaining processing times for their jobs, making them eligible for job assignments even if their actual remaining time might be longer than other servers.

Previously, ties between leastwork were decided arbitrarily, but you can now consider the tiebreaker being the server speed. The server with the higher speed will be selected. If the server speeds are the same, the tie will be broken arbitrarily, but if there are differences in speeds, the faster server will be chosen.

This approach ensures that when multiple servers have the same adjusted remaining processing time, the dispatcher selects the fastest server, optimizing the job assignment strategy based on both workload and server speed considerations.

By considering the adjusted remaining processing time based on the server's speed, the LeastWorkDispatcher will effectively balance the workload, ensuring that faster servers handle proportionally more jobs due to their higher processing speed. This adaptation allows you to make the most out of the faster server while still ensuring efficient job processing across all servers in the system.

## Include 1 paragraph in the extension section of your report describing how the lab helped you complete the project, or how it could be improved to be more helpful.

Engaging in the lab sessions proved immensely beneficial in the completion of this project. The opportunity to collaborate closely with Teaching Assistants (TAs) and the professor within the same room greatly facilitated troubleshooting various challenges encountered during the project. Having immediate access to expert guidance not only saved valuable time but also provided valuable insights and alternative perspectives that significantly enhanced the project's quality. The interactive environment fostered a sense of collaboration, enabling us to discuss intricate problems, exchange ideas, and explore innovative solutions collectively. However, there is always room for improvement. A more structured approach, such as designated Q&A sessions or workshops focusing on specific project components, could further enhance the lab's effectiveness, ensuring that students receive targeted guidance in areas where they encounter the most challenges.

---

## References/Acknowledgements

A lot of troubleshooting was done in order to make this work, Professor Lage and Professor Bender were both consulted as there were many, many errors.