Simulating a server farm and facilitating assignment of new jobs to servers using various criteria

Abstract:

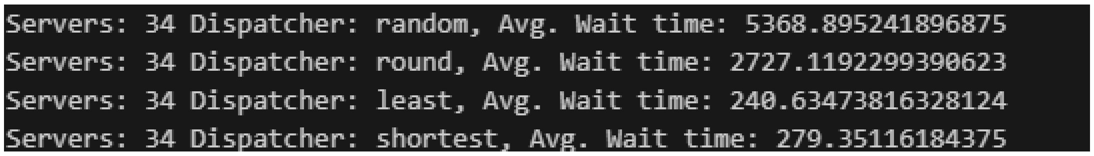
Servers or simply processors need to be utilized effectively to reduce their idle time in order to maximize their power. In this project, I implemented various techniques to enter jobs into a server to study their effectiveness. With the help of a server farm simulation, I was able to visualize my methods and implement variety of server numbers, number of jobs, etc. to further understand the simulation. Core CS concepts used in this project were implementing a queue interface using linked list, handling queues of jobs to enter into a server, implementing an abstract class of job dispatchers and simple data visualization. The key findings were that dispatchers that assign jobs to servers with least processing time remaining result with least average waiting time for jobs. In other words, giving low idle time to servers minimize the waiting time for jobs to process.

Results:

Req Result 1:

I ran my simulation to print the average waiting time under the given conditions provided i.e. 34 servers, 10 million jobs, arriving at 3 and each needing on average 100ms processing time. I implemented 4 job dispatcher types: random, round-robin, one with least work, and one with shortest job queue. Intuition tells the third one will perform best and the random will work worst.

The following table shows the results of the simulations. [see serverfarmsimulation.java line 32]



```
Servers: 34 Dispatcher: random, Avg. Wait time: 5368.895241896875
Servers: 34 Dispatcher: round, Avg. Wait time: 2727.1192299390623
Servers: 34 Dispatcher: least, Avg. Wait time: 240.63473816328124
Servers: 34 Dispatcher: shortest, Avg. Wait time: 279.35116184375
```

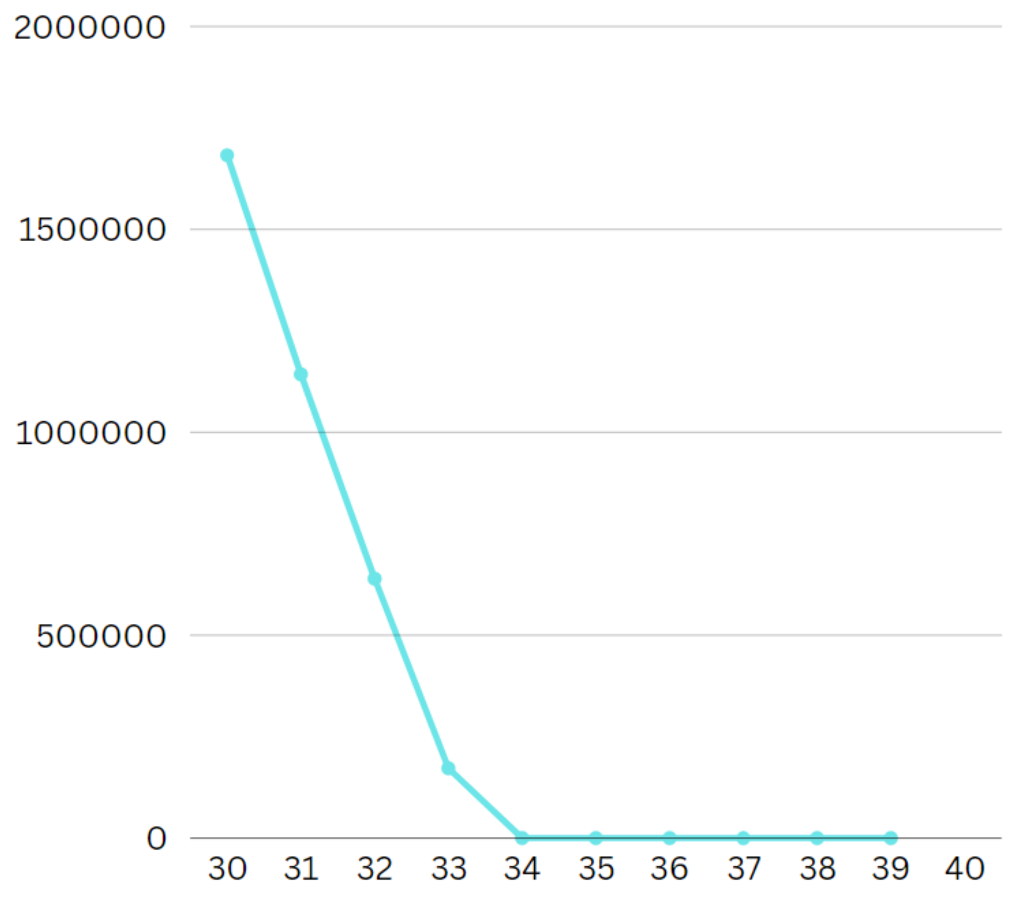
The results match our expectation.

Req Result 2:

(More on this on the extension)

We can guess that since with 34 servers and average time close to the actuall processing time needed, adding just 1 more server should be enough to bring it very close. But the part with number of servers <34 is uncertain. [see serverfarmsimulation.java line 40]

The following graphs shows the result for “shortestQueue” dispatcher from 30 to 39 servers:



We can see that the average time started from millions and very quickly approached the value close to the processing time for an average job. With 34 servers, we are good enough to not fall too far behind.

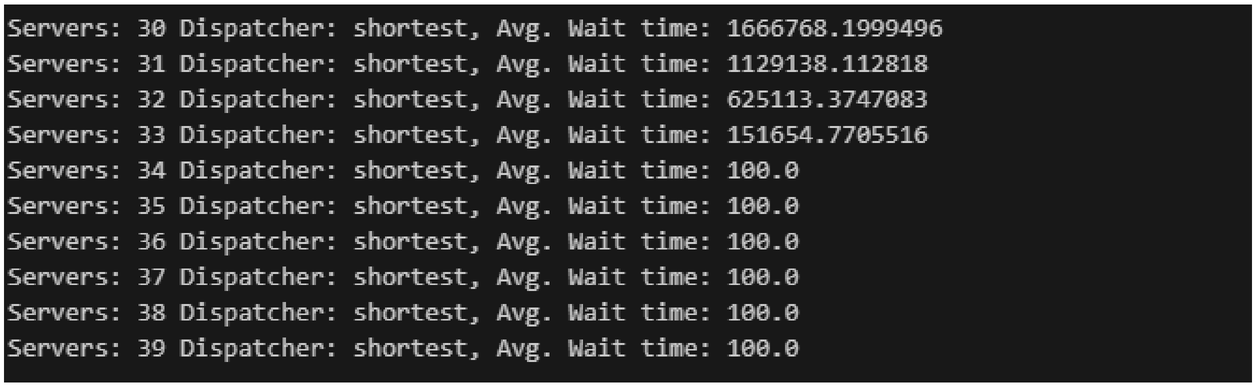
Extensions:

For the extension, I decided to calculate why the drop was certain at 34 servers.

For this, I added a new method the jobmaker class to hypothesize an average situation. I call it getMyJob(). Analyzing the next Exponential () function, I could see that the average value returned is just the mean value entered into it. Specifically, the mean (expected value) of the random variable generated by

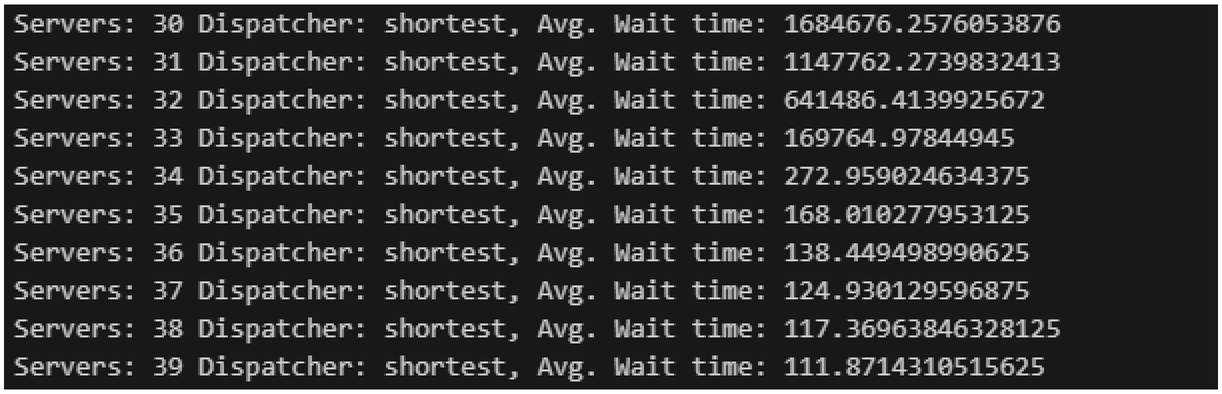
-mean*log(random(0,1)), where random(0,1) is a uniform random variable between 0 and 1,
is simple equal to mean.

Thus, I implemented the getMyJob method simply using this idea to see the values.
I present them below: [To replicate my output see line 66 on serverfarmsimulation.java in extension folder]



```
Servers: 30 Dispatcher: shortest, Avg. Wait time: 1666768.1999496
Servers: 31 Dispatcher: shortest, Avg. Wait time: 1129138.112818
Servers: 32 Dispatcher: shortest, Avg. Wait time: 625113.3747083
Servers: 33 Dispatcher: shortest, Avg. Wait time: 151654.7705516
Servers: 34 Dispatcher: shortest, Avg. Wait time: 100.0
Servers: 35 Dispatcher: shortest, Avg. Wait time: 100.0
Servers: 36 Dispatcher: shortest, Avg. Wait time: 100.0
Servers: 37 Dispatcher: shortest, Avg. Wait time: 100.0
Servers: 38 Dispatcher: shortest, Avg. Wait time: 100.0
Servers: 39 Dispatcher: shortest, Avg. Wait time: 100.0
```

The one implemented normally (getNextJob()) provides a similar output:



```
Servers: 30 Dispatcher: shortest, Avg. Wait time: 1684676.2576053876
Servers: 31 Dispatcher: shortest, Avg. Wait time: 1147762.2739832413
Servers: 32 Dispatcher: shortest, Avg. Wait time: 641486.4139925672
Servers: 33 Dispatcher: shortest, Avg. Wait time: 169764.97844945
Servers: 34 Dispatcher: shortest, Avg. Wait time: 272.959024634375
Servers: 35 Dispatcher: shortest, Avg. Wait time: 168.010277953125
Servers: 36 Dispatcher: shortest, Avg. Wait time: 138.449498990625
Servers: 37 Dispatcher: shortest, Avg. Wait time: 124.930129596875
Servers: 38 Dispatcher: shortest, Avg. Wait time: 117.36963846328125
Servers: 39 Dispatcher: shortest, Avg. Wait time: 111.8714310515625
```

The idle waiting time goes to 0 for ideal case at server number 34. But in the realistic implementation, it hovers close to 100.

This is simply because in the ideal case, each job arrives only after the previous job is finished (when num of servers >34, more on this later). So only the processingTimeNeeded is printed.

In the realistic case, whenever jobs arrive after previous ones finish, it takes 100, but the cases where the previous job took longer it will count as >100 thus increasing the average waiting time.

Calculating the minumum number of servers required for given condition:

The above cases all had num of jobs 10 million, each with mean processing time 100ms and arrival time 3ms+current time. And for this condition, the required servers number was 34.

Lets calculate the average idle first.

Let number of jobs be J, number of servers be S, processing time be y and arrival time be x, then

Average idle Time = Σaverage idle time for each job / J

We can find an expression for average time for each job.

For a small number of servers, the shortest queue dispatcher is similar to the round robin, most will have the same number of jobs, so it will loop around.

Define a batch to be a complete iteration over the list of servers. After 1 batch all the jobs will process by number of servers (time to loop around). So when new one arrives it will see (y-x*S) time remaining for the previous job.

After 1 batch, when the next job arrives, it will go to the first server and wait for (y-x*S)*1
After 2 batches, when the next job arrives, it will wait for (y-x*S)*2
After n batches, next job will wait for (y-x*S)*n

The total number of batches is roughly J/S
(We can think of it as distributing the number of jobs equally between S servers, so each one will take 1/S time)
Thus, for the final batches, final waiting time = (y-x*S)*J/S
Since it is a linear function of S, taking the sum will result in

Σaverage waiting time for each job ~ 0.5*(y-x*S)*J^2/S

(This can be shown by taking an integral for with respect to J, it only approximates since our data is discrete)

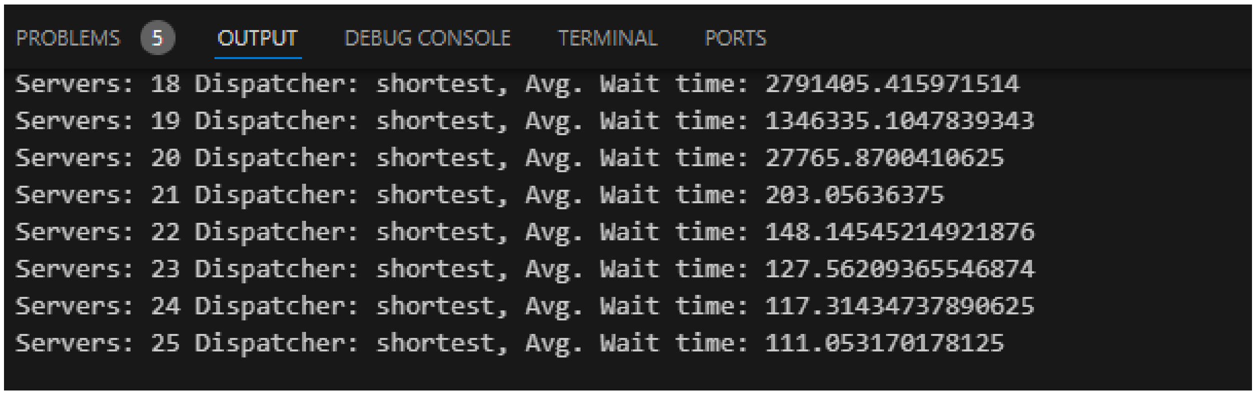
Thus, **Average idle time = 0.5*(y-x*S)*J/S**
And Average wait time = y + average idle time

So the required number of Servers can be taken simply when this expression equals 0. This is bound to happen as by the way of my modelling, negative values are possible when servers are idle.

This 0.5*(y-x*S)*J/S=0
implies y-x*S=0
or S = y/x

Thus, when number of servers > y/x, the servers wont fall too behind.
And we can take the least value as in our case it was 100/3~33.3<34. Thus 34 was optimal choice.

We can verify with experiments:
Here we are taking process time 100 and arrival time 5, from our calculations, it should take 21 servers: (see serverfarmsimulation.java line 35)
Calculating from formula of average wait time for 18, gives 2.77 million which is close enough as shown from actual data:



```
Servers: 18 Dispatcher: shortest, Avg. Wait time: 2791405.415971514
Servers: 19 Dispatcher: shortest, Avg. Wait time: 1346335.1047839343
Servers: 20 Dispatcher: shortest, Avg. Wait time: 27765.8700410625
Servers: 21 Dispatcher: shortest, Avg. Wait time: 203.05636375
Servers: 22 Dispatcher: shortest, Avg. Wait time: 148.14545214921876
Servers: 23 Dispatcher: shortest, Avg. Wait time: 127.56209365546874
Servers: 24 Dispatcher: shortest, Avg. Wait time: 117.31434737890625
Servers: 25 Dispatcher: shortest, Avg. Wait time: 111.053170178125
```

Thus, this matches our prediction for number of servers needed.

Acknowledgements:

In completing this project, I consulted with Max and Allen, but did not look at any code. I attended the evening TA sessions on Sunday and Monday. I also consulted the wikipedia page on probability distibution and uniform distribution.