# Distributed systems job allocator for cost cautions resource efficiency

## Introduction

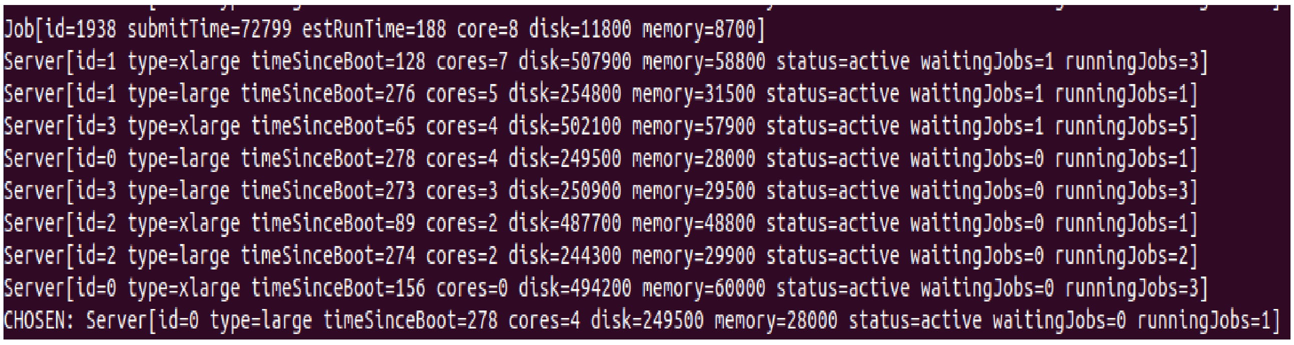
This job allocator aims to be as cost-efficient as possible while maintaining a high resource percentage. This job allocator does not just give up after using a bunch of different metrics and not finding a match. This algorithm will keep searching for a server by slowly lowering the criteria one by one. This allows it to be better allocated while not faulting to just the first server available. This algorithm borrows a lot from a best-fit algorithm in regards to matching criteria, but from tests receives better cost and resource utilization.

## Problem Definition

Keep costs low, resource utilization high without overloading the largest server, for assigning distributed jobs. The reasoning for developing such an algorithm was to not peg a server 100% server utilization, but still have a high amount and low overall cost. This is because keeping computer hardware at such high amounts of load can damage the systems over time, therefore not keeping them at 100% but still having low costs and low turnaround time is perfect.

## Algorithm Description

To start, the basis of the algorithm is it receives a job and then it asks the server for all the capable servers for that job. After receiving all the capable servers it will sort them by core count. It will then go through and try and find a server that has more or equal the specs that the job requires (cores, disk, memory) and a server that does not have any waiting job. The reason for checking the waiting jobs of a server is because we don't want to overload one server even if it has the best matching specs. After all, it will greatly reduce our turnaround time. If it still does not find a server after this first search it will go through all the servers again using the same criteria as the first search but without caring about core count. This happens again if it still has not found a server with just memory and waiting jobs. If it still has not chosen a server, it will then only search through the servers based on waiting jobs, since we do not want to overload one server and split the load evenly as possible it will try and find a server with only 1 waiting job, then less than 4 waiting jobs. If after all this a server has not to be chosen it will default to the last server (which will have the largest core count, since we sorted it smallest to largest).



```
Job[id=1938 submitTime=72799 estRunTime=188 core=8 disk=11800 memory=8700]
Server[id=1 type=xlarge timeSinceBoot=128 cores=7 disk=507900 memory=58800 status=active waitingJobs=1 runningJobs=3]
Server[id=1 type=large timeSinceBoot=276 cores=5 disk=254800 memory=31500 status=active waitingJobs=1 runningJobs=1]
Server[id=3 type=xlarge timeSinceBoot=65 cores=4 disk=502100 memory=57900 status=active waitingJobs=1 runningJobs=5]
Server[id=0 type=large timeSinceBoot=278 cores=4 disk=249500 memory=28000 status=active waitingJobs=0 runningJobs=1]
Server[id=3 type=large timeSinceBoot=273 cores=3 disk=250900 memory=29500 status=active waitingJobs=0 runningJobs=3]
Server[id=2 type=xlarge timeSinceBoot=89 cores=2 disk=487700 memory=48800 status=active waitingJobs=0 runningJobs=1]
Server[id=2 type=large timeSinceBoot=274 cores=2 disk=244300 memory=29900 status=active waitingJobs=0 runningJobs=2]
Server[id=0 type=xlarge timeSinceBoot=156 cores=0 disk=494200 memory=60000 status=active waitingJobs=0 runningJobs=3]
CHOSEN: Server[id=0 type=large timeSinceBoot=278 cores=4 disk=249500 memory=28000 status=active waitingJobs=0 runningJobs=1]
```

**One job assignment  using config20-short-med**

Take the image above, the job requires 8 cores, 11800 disk and 8700 memory. The algorithm goes through the first 3 servers and they have sufficient computer specifications but because they are already full of jobs and has a job waiting, it decides to choose server 0 of type large. This is because it has required specifications and does not have any waiting jobs.

## Implementation Details

For this algorithm, it is required to have a Server class that overrides the compareTo method to make it easy to sort an array of servers by core count. Secondly, the job and server must be put into a class that allows simple calling of their specifications, i.e core count, memory, etc. Lastly, you need to make sure you can have a server collection that contains all your available servers to choose from that is sorted.

## Evaluation

To get started with using this algorithm follow these steps:

1. Clone the Github linked below under references.
2. Install OpenJDK.
3. Once the Github has been cloned open a terminal inside the ‘src’ folder of the cloned Github repository.

4. Once your in the src folder, type in the terminal ‘javac Client’, to compile the Client code.
5. Run the test script with the following command `./test_results "java Client -a fff" -o ru -n -c ../configs/other/` (without `)

This algorithm is not a revolutionary algorithm that is the best at cost, resource and turnaround time. This algorithm focuses on high resource utilisation without compromising on cost and high turnaround times. If you want a job allocator algorithm that provides the fastest then Best Fit is still the best algorithm you can choose, it will always choose the best server available for the job it is given at that certain time. Even First Fit has a better turnaround time. This is the main downside of the algorithm, therefore if you are processing jobs that are time conscious and require high speed then you should stick with a Best Fit algorithm.

On the other hand, quick turnaround times are not the only thing that is considered in a job allocation algorithm I believe it is the least important time if it is not exponentially large like the ATL algorithms turnaround times are. This is why I created this algorithm to be a hybrid of cheap costs and high resource efficiency. The algorithm does just that, in the tests, it beats FF, BF and WF by a minimum of 6% in resource utilisation. Cost is a different outcome, it is cheaper than FF, BF and WF but only over the average of all the tests, which means that it depends on your job workload and available servers. Even though the cost is not always better with this algorithm it is always close by a few dollars with BF and FF. This algorithm should be used if you want high resource utilisation without compromising on very high turnaround times.

## Conclusion

To conclude I think this algorithm is a good alternative to algorithm ATL and BF since it's a combination of both these algorithms combined in one. At the end of the day if you don't care about turnaround time and care more about costs and resource utilisation then ALT is a better algorithm job allocation algorithm, but if you care about turnaround time but still want better resource utilisation and costs than BF then this is an algorithm you should consider.

## References

https://github.com/claudesortwell/COMP3100