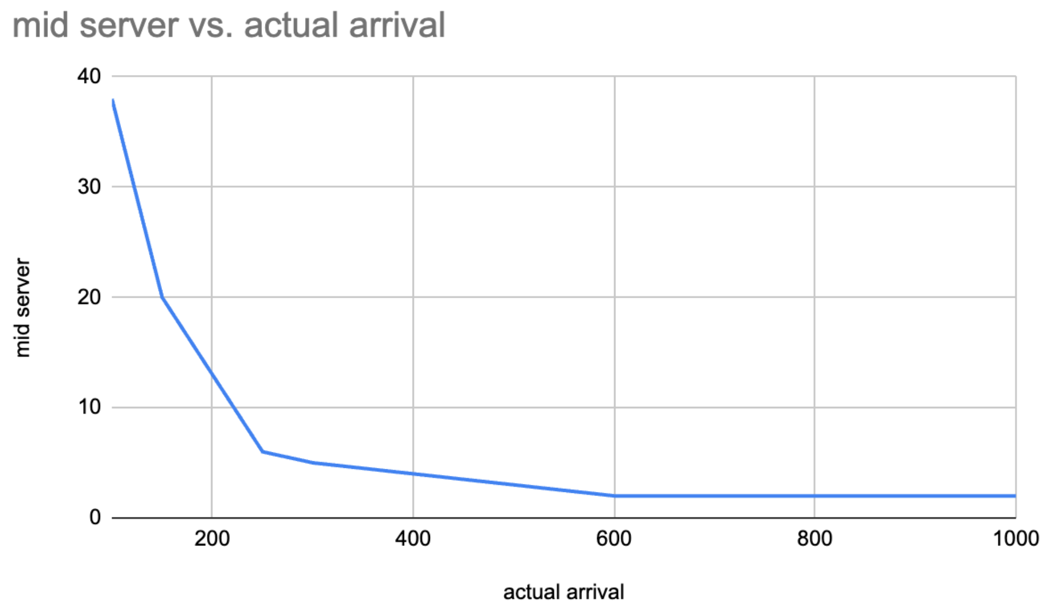
I use the initial VM as a master VM which plays the role of a coordinator for all the front end and middle end servers. Master servers will use a concurrent hash map to store the roles of all the booted servers and use their vm id as the key. And when a VM starts running, the server will call RMI on the master server to obtain the corresponding role value in the hash map. Then after getting the role value of the VM ID, that specific server will then prompted to execute the front end or middle server routine. And the master will keep track of the current number of opened front end and middle servers in two global variables for future dynamic scaling. When the master server is booted, it'll create a binding for the remote object with the name "master" that other front end and middle servers will communicate with by looking up the remote object master binded to. To coordinate the process request phase with the parse request between the two tiers of servers, I use a centralized queue in the master server. And all parsed requests in the front end will be enqued into the centralized queue, and all middle servers will deque the parse requests from the same queue and then process them. After the initial booting finishes, the master VM will act as a front end in addition to monitoring for scale outs.

I will boot a middle server before any operation happens as we will need at least one middle server and this can reduce the time wasted waiting for the middle servers to boot. The initial arrival rate is measured by timing the call to the second getNextRequest because the first request might be noisy due to the processing that took up time after the first client arrives to try to accept the first connection. And the master VM will process the first two requests to minimize timeouts during the booting period. Then after the server gets an estimation of current arrival rate, by fixing the number of front end servers then finding the optimal number of middle servers, and fixing the number of middle server then finding the optimal number of front end servers for each arrival rate. A general trend between the number of optimal middle servers and arrival rate is shown in the plot. The graph is then plotted using desmos to get a equation to calculate the optimal mid server given the arrival rate. And to avoid out of memory issues on the stack due to opening too much servers, the number of mid servers I open is capped at 6 initially.



Then as more requests come into the server, I will monitor the current number of awaiting front and middle tasks and compare it with the number of servers available to decide if scale out needs to happen. By trial and error, I have found that the best scale out factor s such that an indication of resize is needed when size of queue/number of mid server > 1.5. And from my measurement, the time needed to process one request is approximately 60ms. While the time to process a midtier task takes approximately 300-400ms, this means the ratio of front end and middle end should be approximately 1:5 to ensure no bottleneck in either of the two tiers. Therefore, everytime i scale out the midtier, I will also check if the ratio is satisfied and scale out front if it is no longer satisfied. I also tested on the amount of servers we should boot each time we scale out and found by repeated trials that booting one extra server at a time produces the best performance. And to avoid thrashing, I impose a cooldown period of 5200 ms after scale out is done, and during this period, no more scaling out can be done. This is because server booting takes 5000ms and we should only consider scaling out more after changes of the booted servers can be reflected.

And in the case where the load decreases and scale in is needed. I monitor the time each individual server has been consecutively idle and scale in if it reaches a constant amount of time. By benchmarking the different time values to use as an indicator of scaling in, I found that 500 seems to work best for front end and 2500 works best for back end. In the case we do a scale in for the front end, we first unregister the VM as a front end so it stops receiving more requests, then parse all the requests left in the queue, then end the VM.

To avoid wasting time on processing requests that can’t be completed on time, everytime I try to poll from the centralized queue. I check if the middle tasks queue have more tasks than completable, and if it does, I will drop the earlier requests until the size is under 1.5*number of mid server. The 1.5 is decided similarly to the scale out factor, because a mid server can process approximately 2 requests in 1000ms from experimentation and benchmarking, that means each server is able to handle 2 requests before they timeout. Therefore, the queue should not have requests that are more than twice the number of mid servers, and by testing a few numbers in the proximity of 2, 1.5 seems to have the best performance in terms of dropping and scaling out.