Heterogeneous Server Farm Simulation

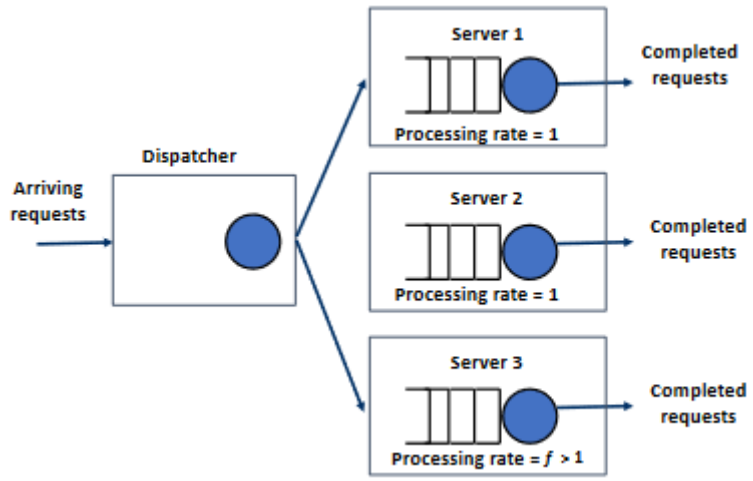
A. Introduction

Data services, websites, and applications often rely on vast networks of computers known as server farms. These server farms handle incoming user requests, process them, and return results—ideally as quickly and efficiently as possible. However, server farms are rarely uniform. As organisations expand their infrastructure over time, new servers with better specifications will be added while older machines remain in operation. This leads to heterogeneous server farms, where different machines have different processing capabilities.

A critical challenge arises: How should we assign incoming tasks to servers of varying speeds so the whole system runs most efficiently? This project explores this question using simulation. We utilise a 'hypothetical' server farm of three servers (two slow, one fast) and a dispatcher, which assigns jobs to the servers based on their current load and a chosen load balancing algorithm.

The allocation of jobs to queues occurs by one of two load balancing algorithms. In the first, a job is sent to a slower server if and only if one of the slow servers have at least d fewer jobs than the fast server. In the second, the number of jobs in the fast server is divided by f to account for the quicker service time. Observe both push jobs towards the fast server. Both d and f are specified by the user. This server design is shown below.

Figure 1: The server farm simulated for this project



Arrival and service times are simulated using two different modes: A random mode, where job arrivals and processing times are randomly generated using specific distributions; and a trace mode, where the simulation uses pre-supplied files for job arrival and service times.

The goal is to compare two load balancing algorithms and examine how prioritisation of faster servers and idle servers affects the overall performance. The key quantity of interest in this simulation study is the mean service time (hereafter, MST) – the amount of time the average job spends in the system. That is, the sum of the time the job spends waiting in the queue (which may be zero) and the service time. Statistical inference procedures are applied to examine the relationship between the MST and d (a parameter which controls how likely a job will be sent to a fast server).

B. Simulation methodology

The `server_farm_simulation.py` module contains the simulation program. To run it, parameter/s must first be defined. These are defined in the second cell of the script (in lines 27-38). If the script is being run in random or trace mode (as opposed to 'custom mode' where custom parameters are defined within the script), two parameters must be specified. The first parameter is `_ascii` (line 27), which must be set to `True`. The second is test number, specified in the subsequent line (an integer from 1 to 6 inclusive, since there are six possible tests, located in the config folder). Each test consist of four files, specifying the mode (random or trace), and the relevant parameters. Once the script is run, the required output files will be written to the 'output' folder (details in Section B.1). If `_ascii` is set to `False`, custom parameters are defined in lines 33-38. This 'mode' is used for exploring the relationship between MST and d. When the script runs, the relevant MSTs and confidence intervals are printed and returned (details in Section B.2).[1]

B.1. Random and trace mode

If running in trace or random mode, line 31 of the script retrieves the test number. The `get_inputs` function opens and reads the relevant ASCII files in the 'config' folder, retrieves the relevant parameters, and then feeds the parameters into the `compute_mean_service_time` function. Once this function has returned results, the relevant output of this function is written to the 'output' folder.

When running in random mode, the probability density function (pdf) of the arrival time is the product of the exponential distribution (with rate λ) and the continuous uniform distribution (with parameters a_{2l} and a_{2u}). The probability density function of the service time follows a Pareto distribution with scale parameter α and shape parameter $\beta + 1$. Python's `random` library was used to simulate from these three parametric distributions (exponential, uniform, and Pareto). The entire process takes up only two lines within the `generate_arrival_and_service_times` function. I also

[1] Note by default, the `_ascii` parameter is set to `True`, which means the module can be run using only the bash file.

used this library to ensure reproducibility—by initialising the random number generator used to generate arrival and service times using the `random.seed` function.[2]

However, the primary function introduced in this module is `compute_mean_service_time` (lines 133 to 323), which returns the mean service time along with other required output. A while loop is used to continue simulating until a stopping criterion is reached – either when the 'master clock' (column 't' in the script) reaches a certain maximum value (in random mode), all jobs have left (trace mode), or when a certain number of events have occurred (for the 'custom mode').[3] Further details on how this function simulates events to retrieve the mean service time are provided in the Python script, particularly the function docstrings.

B.2. Relationship between MST and d

The `simulated_mst_inference` function applies standard frequentist statistical inference procedures to explore the relationship between (a) MST and d, and (b) MST and the choice of load balancing algorithm. The first step is defining the 'server farm designs' that were used to estimate the MST by simulation. A grid of different values for d were used such that $d \in \{0, 0.3, 0.7, 1, 1.5, 2, \infty\}$. A value of 0 means jobs will be sent to the faster server when it is busier than both slow servers (algorithm 2 also considers how the fast server less likely to be busy in the future due to its lower average service time, further pushing jobs toward the fast server). A value of ∞ means jobs will only be sent to the fast server when it is idle. Each intermediate value is also examined (no point making the grid too fine since differences will be less likely to be statistically significant). In addition to the grid of d values, both algorithms 1 and 2 were used to assign jobs to the queues. With 7 values of d and 2 algorithms, this results in 14 different 'server farm configurations'.

The second step is defining all relevant parameters. Recall this was done toward the start of the script (lines 33-38). These parameters are fed into the `simulated_mst_inference` function which calls the `compute_mean_service_time` function. A description of these parameters is provided below.

These are $S = 10$ MSTs that are simulated for each server farm configuration (resulting in a total of 14×10 MSTs being simulated). $N = 20000$ events are simulated to estimate each MST; and $m = 500$ events in each simulation are discarded as the burn-in sample (i.e. the transient part). These parameters provide a good balance between having enough data to identify true differences when they exist (i.e. rejecting false null hypotheses) and ensuring a reasonable runtime. Moreover, the size

[2] I chose the integer 93342021, a custom of mine for all university projects so-far – an eight-digit integer where the first four digits correspond the course code and the final four to the calendar year. This ensures no trial-and-error process is ever used to choose an initial integer that produces the most 'favourable' result. This function uses the Mersenne Twister algorithm to generate random numbers, which only repeats after $2^{19937} - 1$ numbers have been generated, so we can be sure that draws from each distribution are indeed independent and identically distributed – necessary for the validity of this study.

[3] For the custom mode (used for the MST vs. d exploration), I waited for a certain number of events to occur rather than waiting for all jobs to leave as the stopping criterion. This is because in the latter method, a new job may arrive and depart after the period where no more jobs are introduced into the system, positively biasing the MST. Of course, the difference will be negligible for a large number of events, but is incorporated to ensure the output of the function is valid even for a small number of events.

of each hypothesis test, α, is set to 0.05, following standard practice. The rest of the parameters ($f = 1.5$ and the probability density function parameters) are provided by project requirements.

Denote the sth ($s = 1, \ldots, 10$) simulated MST of the ith ($i = 1, \ldots, 14$) server farm configuration as $\widehat{MST}_{i,s}$. The sample average MST of the ith server farm configuration, as well as the (asymptotic approximation of the) standard error of the estimated MST of the ith server farm configuration, are computed as follows:

$$\overline{\widehat{MST}_i} \triangleq \frac{\sum_{s=1}^{10} \widehat{MST}_{i,s}}{10}; \quad \text{se}\big(\widehat{MST}_i\big) \triangleq \sqrt{\frac{\sum_{s=1}^{10} \big(\widehat{MST}_{i,s} - \overline{\widehat{MST}_i}\big)^2}{9}}$$

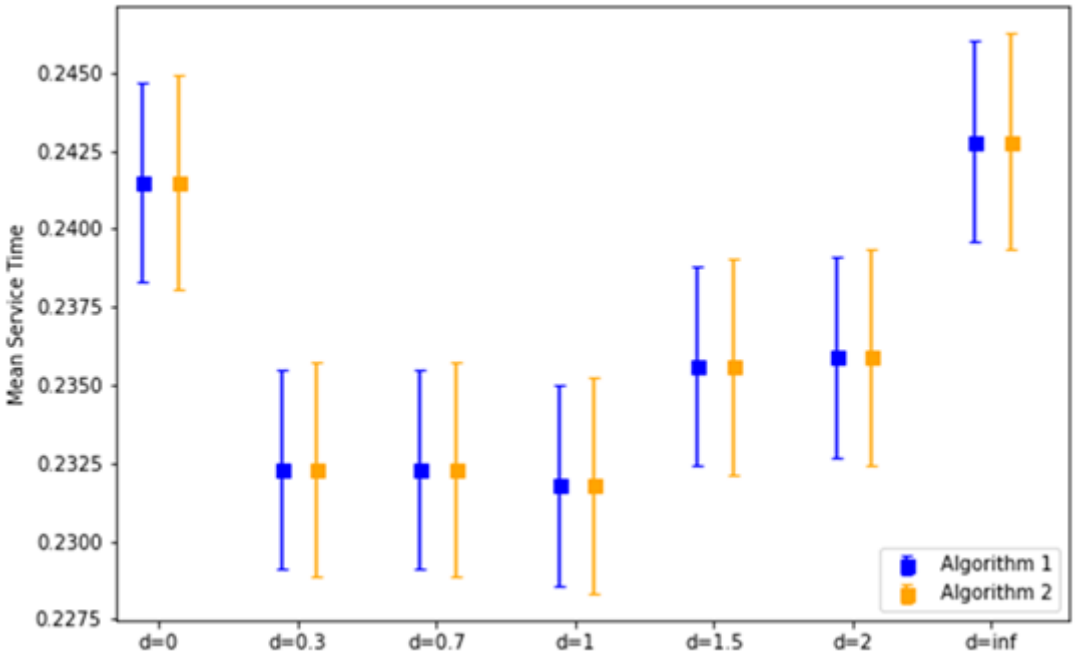
A 95% (asymptotic) confidence interval for $\widehat{MST}_i$, the estimated MST of the ith server farm configuration, is formed below (where $t_{0.975,9}$ is the 97.5th percentile of the t-distribution with 9 degrees of freedom):

$$95\% \text{ CI for } \widehat{MST}_i = \Big[\overline{\widehat{MST}_i} \pm t_{0.975,9} \times \text{se}\big(\widehat{MST}_i\big)\Big]$$

If the confidence interval of server farm design j does not overlap with the interval of server farm design k, one can conclude with 95% confidence that the true mean service times of the two server farm designs are different. However, if the sample average MST of server farm j is within the CI of server farm k, there still may be a significant difference (which can be examined using a t-test). Otherwise, we do not have evidence to conclude there exists a difference with 95% confidence. Refer to Figure 2 overleaf for the plots of each server farm design and details of the statistical inference procedure, which concludes the optimal value for d is 1 (or not statistically different from 1) but finds no difference with respect to the choice of algorithm.

Also note each comparison involves a 5% chance of making a type-I error – falsely inferring a difference in mean service times – when no true difference exists, which may be present since due to the use of simultaneous hypothesis testing. For further details on the computation, refer to the docstring and comments in the `simulated_mst_inference` function (lines 326-364).

Figure 2: Statistical inference procedures: MST vs. d and algorithm choice



The plot above shows the 95% confidence interval bars for each value of d, for both algorithms. Whilst there is no statistically significant difference in MSTs between algorithms 1 and 2 for the values of d considered above, there is strong evidence that the choice of d impacts the MST.

First, it is clear that MST is lowest for $d = \{0.3, 0.7, 1\}$. The MST of these values is clearly lower relative to $d = 0$ since the CIs don't even overlap. There is also very strong evidence that the MST with these values is less than the MST with $d = 1.5$. For example, there is strong evidence that the MSTs are different when using a Welsh t-test to test if there is a difference in (mean) MST for $d = 1$ vs $d = 1.5$ ($p < 4.8 \times 10^{-4}$). Note this test does not assume the variances of the MSTs are equal (unlike Student's t-test). However, comparing $d = 1$ to $d = 0.7$ yields a p-value of 0.213, implying the means are not statistically different. Similarly, against $d = 0.3$, the p-value is 0.4152.

So whilst $d = 1$ gives the lowest average MST for both algorithms, the MST is not statistically different from the MST with $d = 0.3$ and $d = 0.7$. Once d exceeds 1, there is a clear trend of increasing MSTs. For algorithm 1, we can therefore conclude $d = 1$ is the optimal value of d to minimise MST since the MST is less than $d = 0$ and $d = 2$ and this difference is statistically significant at the 5% level. This in turn implies the optimal value for algorithm 1 is any value within the interval (0,1] due since all other quantities used to determine queue assignment are integers. However, since n_3 is divided by f in algorithm 2, which may not be integral, the optimal value is likely to be floating-point value, albeit one that isn't statistically different from 1 at the 5% level given the results of this statistical inference procedure. Parameters S or N may be increased to increase the power of the test, and a finer grid of d values may be used, however this will increase runtime.

C. Conclusion

This report has described how a particular heterogeneous server farm can be simulated. There is evidence that the mean service time of this server farm is affected by the value of d (a parameter which controls how likely a job will be sent to the fast server), with a value of 1 minimising the mean service time, but no evidence the choice of load balancing algorithm affects the mean service time when controlling for d. Exploration of alternative server farm designs and quantities other than the mean service time and are left to further research.

Matthew Cord, 2021