Load Balancing with Random Choices

CSC 446: Research in Simulations Operations

Instructor: Dr. Kui Wu

By: Payton Murdoch and Sebastian Booth

Date: Tuesday, April 11th, 2023

Table of Contents

Table of Figures and Tables

Project Description

Load balancing is a methodology which exists in the practical application of queueing models. In general terms load balancers can be denoted as proprietary servers which distribute incoming traffic, packets or service requests efficiently among a network of servers. These methods are utilized in various high-traffic networking environments such as service requests being piped into DNS servers, cloud computing services and web server database farms. Within our project, we are examining three specific methods of load balancing and directly comparing and contrasting their efficiency in terms of the average queue length over the runtime. With this in mind, we beg the question as to whether or not adding pseudo-random elements to a load balancer can further optimize the system's efficiency.

To further elaborate, our test load balancing method is denoted as a Random Minimum load balancer. This takes a random selection of the servers within the series and chooses to send the oncoming traffic to the one with a minimum queue length. The Random Minimum method will be tested directly against two baselines, Pure Random load balancing and Round Robin load balancing. Pure Random as the name implies distributes traffic at random between all the servers with no consideration of queue length. Whereas Round Robin sequentially offloads the traffic to each server as it loops. When considering our question, Random Minimum can be compared to a hybrid methodology between the two baselines, it is not solely sequential and not solely random. It holds both calculated aspects similar to that within Round Robin and pseudorandom aspects akin to Pure Random.

Simulation Model

The model has three primary subsystems to simulate; the *sender*, using a Poisson arrival system, the *dispatcher* using one of three selected scheduling methods (RandMin, PureRand or RoundRobin), and the *server-array*, with exponentially distributed service times.

The *sender* produces traffic using a Poisson arrival system. Random exponentially distributed interarrival times are produced based on the mean interarrival time, λ.

The *dispatcher* is responsible for selecting a server to which each customer is sent. It uses one of three scheduling methods:

1. Random Minimum (RandMin): A subset of servers of size *d* is randomly selected, and the dispatcher finds the server with the shortest queue out of this subset. The random selection is uniformly distributed across all the servers.
2. Purely Random (PureRand): Each customer is sent to a server chosen based on a purely random choice. The random selection is uniformly distributed across all the servers.
3. Round Robin (RR): Servers are cycled through one at a time. Each client is dispatched to a new server, ie: S_1, S_2, …. S_m, S_1, S_2, …. $S_{m.}$

The goal of the *dispatcher* is to distribute customers to servers such that server workloads are minimized and customers pass through the system as quickly as possible. The theoretical ideal case would be for the *dispatcher* to calculate exactly how long it would take a customer to wait in each queue, and then dispatch the customer to the fastest. In a real-life system, a dispatcher is unlikely to know how long the service times will be for each customer, so picking the shortest queue in terms of the number of customers is the realistic ideal case. Note that RandMin, if $d = m$, is equivalent to this realistic ideal case, as it will check for minimum queue length across all servers. RR also approaches the ideal case as the variance for service time approaches zero.

The *server-array* contains all of the servers working in parallel. Each server has a separate queue, and service time is randomly generated along an exponential distribution according to the mean service time, μ. When a dispatcher sends a customer to a specific server if the server is occupied, the customer enters the server's queue, otherwise, it is immediately served by the server.

The system diagram (Figure 1) also shows a receiver into which all servers direct traffic. This simply represents the end of our simulation, where data for departure times is collected. Customers effectively exit the system upon departure from the server array.

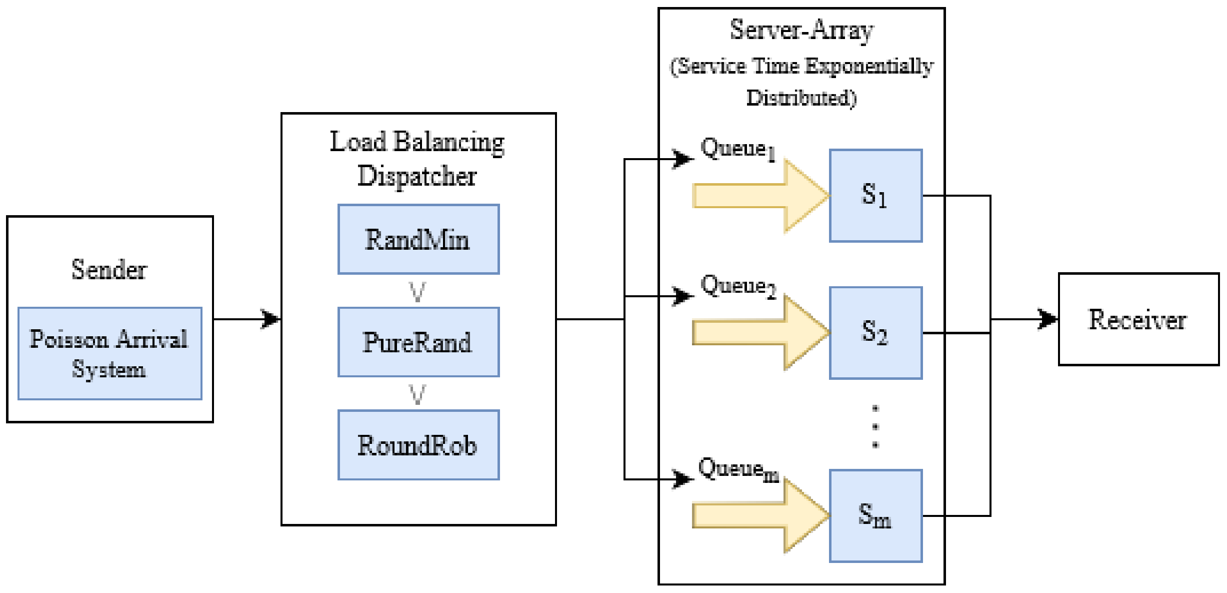

Figure 1. System Diagram

Simulation Parameters

Table 1 shows all the configurable parameters for the simulation model. The primary goal in testing will be to compare the results of the three scheduling methods, as described in the Simulation Model section. We also aim to conduct tests across each method, varying each of the additional parameters to observe changes in system behaviour across each of the different scheduling strategies. The additional parameters include the number of servers (*m*), server selection (*d*), arrival rate (λ) and service rate (μ).

Parameter	**Symbol**	**Description**
Scheduling Method	**RandMin**, **PureRand**, or **RR**	Scheduling method used by load-balancing dispatcher. Determine a strategy for selecting servers to which customers are sent.
Number of Servers	***m***	Number of servers working in parallel in the server array
Server Selection	***d***	Size of selection used by random minimum strategy. For each client, RandMin randomly selects *d* servers and finds the smallest queue
Arrival Rate	**λ**	Mean interarrival time used to produce random arrivals in the Poisson arrival system
Service Rate	**μ**	Mean service time used to produce random exponentially distributed service times

Table 1.

Table 2 shows the system outputs we will be using as performance metrics to compare the system's behaviour using different input parameters. They include metrics for server workloads and time spent in the system. Server workloads will be measured by maximum and average queue lengths across all servers and system time is a measure of the average time between arrival and departure for customers.

Output	**Description**
Maximum Workload	Longest queue length among all servers
Average Workload	Average queue length across all servers
Average System Time	Average time spent in the system across all customers

Table 2.

Methodology

All code for the simulation was written and executed in Python 3.9.13 [1]. The code splits the simulation into several sub-tasks, with the primary tasks being generating clients (*generateClients*), and processing events (*processEvents*). Additional functions were made for processing and extracting the data returned by *processEvents*, but the *generateClients* and *processEvents* functions encompass the simulation model described in this report.

Setup of Simulation

generateClients creates a list of clients/customers, each with a randomly generated inter-arrival time and service time. Both of these random values are exponentially distributed and are generated using Numpy's random.exponential function [2]. The randomly generated numbers are based on the system parameters λ and μ, denoting mean interarrival time and mean service time, respectively.

processEvents uses the list of clients created by generate clients and generates a list of events containing every arrival event. For each arrival event, the relevant customer is sent to a server based on the scheduling method, and a departure event is created based on the queue length and service times for the server. For each departure event, the queue lengths are updated and data is logged for future analysis. *processEvents* continues going through the event list in chronological order until the list is empty, creating departures based on arrivals, and logging data based on departures.

Data Collection

In order to capture data for analysis, each customer is stored in a Python dictionary object, with a sub-dictionary containing all relevant data points:

```
Customer #: {'IAT': interArrivalTime, 'AT': arrivalTime, 'ST':
serviceTime, 'SN': serverNumber, 'QL': queueLength, 'DT': departureTime,
'TT': totalTime, 'QT': queueTime}
```

To record data pertaining to server workloads, each server corresponds to a list containing 2-tuples for every event that affects the server. Each tuple is of the form `(queueLength, time)`, allowing for easy analysis of queue lengths over the course of the simulation period. The data observed in the results is recorded from running the code 3 separate times with the same system parameters to obtain the results for each of the 3 load balancing methods on the same seed and then replicated with 4 other runs with different seeds. After 5 runs we repeat the process with a new scenario. For each run, we export the resulting average queue length based on total runtime to an Excel spreadsheet in order to construct the graphs and tables.

Testing Results

As described in the Simulation Parameters section, we wanted to conduct an extensive collection of tests in which we observed the efficiency of the methods by way of changes in each parameter in independent tests. These tests were all conducted with regard to control, our baseline system variables are the midpoints shown in the resulting graphs and tables by which the number of customers is 100, $m = 10$, $d = 5$ or $\frac{1}{2}m$, $\lambda = 1\frac{minutes}{arrival}$ and $\mu = 10\frac{minutes}{service}$. Each test parameter was run with three scenarios. The first is the control, the second is a scenario with lower numbered parameters and the third with higher parameters. The analysis will be split into four sections with each section reporting the collected statistics, the cross-replication averages and the resulting graphs and inferred trends.

Testing Number of Servers

The first test parameter to be analyzed is the Number of Servers, denoted m readily available within the system. As such, we set the other test variables to $d = \frac{1}{2}m, \lambda = 1\frac{minutes}{arrival}$ and $\mu = 10\frac{minutes}{service}$. We should note that d is changing as well in this case. Since d must be strictly less than m and greater than 1 since $d = 1$ is the exact same as the Pure Random method. The closest approximation we can do to independently test m is to maintain $d = \frac{1}{2}m$. In these tests $m = 4, 10, 14$. For collected statistics, abbreviations *RM, PR* and *RR* are utilized to denote Random Minimum, Pure Random and Round Robin respectively and the vertical breaks in Table 3 and the others represent the changes in system parameters.

Seed		RM	PR	RR		RM	PR	RR		RM	PR	RR
1		7.237661	6.387279	6.387279		0.683474	1.252677	0.918457		0.652217	1.27074	1.203682
2	$m = 4$	7.493469	7.863866	7.655564	$m = 10$	0.798601	1.10162	1.034907	$m = 14$	0.785651	1.11743	1.063667
3		5.674716	6.999925	6.572635		0.865506	1.307951	1.251288		0.394947	0.803966	0.784956
4		7.327747	6.095225	6.784667		1.695511	2.442509	1.928533		0.6843	1.114462	0.946059
5		5.427234	5.625642	5.625642		1.429531	2.09668	1.827243		0.590143	0.73859	0.911111

Table 3.

Average Statistics from these runs were recorded and culminated as cross-replication means, variances and the 95% confidence interval measures in Table 4. Furthermore, the information from the table was then implemented into the Figure 2 graph. It must be noted that the background colour in the following table corresponds to the colour of the points and trendlines. Although the error bars are not immensely apparent in this graph, it should be noted that the denoted error corresponds directly to the confidence measure that was recorded.

m Size	Random Minimum	σ^2 RandMin	Confidence Measure	Pure Random	σ^2 PureRand	Confidence Measure	Round Robin	σ^2 RR	Confidence Measure
4	6.632165	0.990218	0.195035	6.594387	0.751641	0.169923	6.605157	0.535889	0.143478
10	1.094525	0.195599	0.086683	1.640287	0.350676	0.116065	1.392086	0.212216	0.090289
14	0.621451	0.021036	0.028427	1.009037	0.051637	0.044538	0.981895	0.025239	0.031137

Table 4.

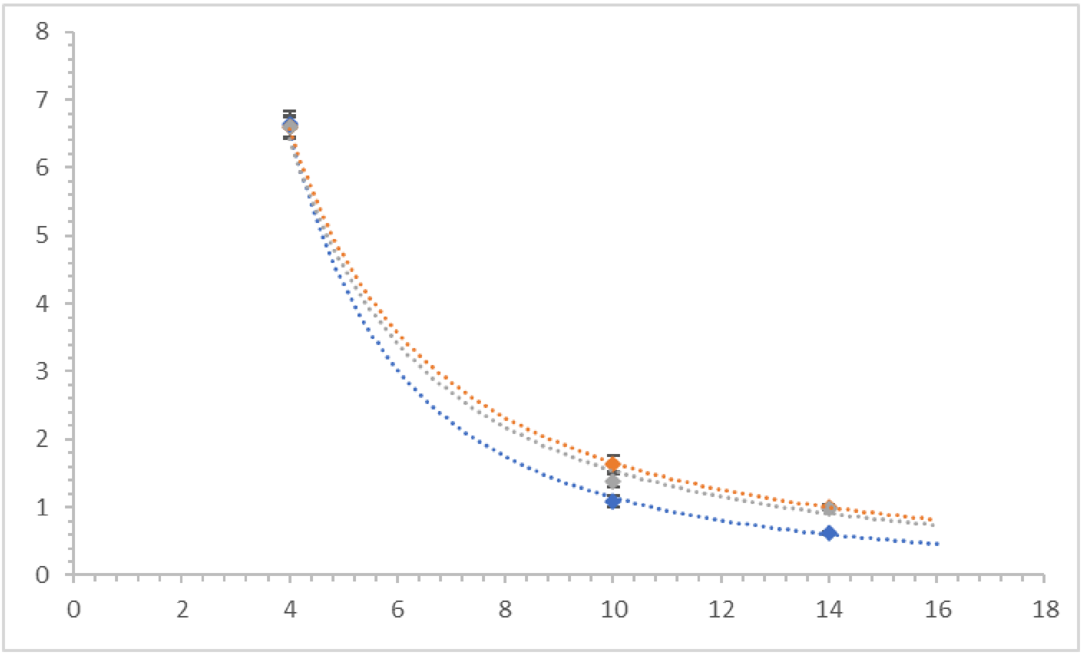

Figure 2. The measure of efficiency between the number of Servers in a system represented on the X axis and the average length of the queue during runtime represented on Y.

Testing Selection Size

Selection size is the next test and unlike m, there are no variables dependent upon it. Thus it can be measured independently. As such, we set the other system parameters to $m = 10$, $\lambda = 1\frac{minute}{arrival}$ and $\mu = 10\frac{minutes}{service}$. We wanted to keep d bounded so that it does not replicate the behaviours of Pure Random or Round Robin techniques as such we want $1 < d < m$. Therefore we determined that our test variables are $d = 3, 5, 9$. These test variables give us an idea as to how Random Minimum behaves when the selected size is $< \frac{1}{2}m$, $\frac{1}{2}m$ and $> \frac{1}{2}m$. The culminated replications are recorded in Table 5.

Seed		RM	PR	RR		RM	PR	RR		RM	PR	RR
1		1.251914	1.546738	1.221716		0.683474	1.252677	1.252677		1.337848	2.09668	1.827243
2		1.252467	1.586142	1.462911		0.798601	1.10162	1.10162		1.387753	1.818087	1.966237
3	$d = 3$	1.143979	1.346622	1.590156	$d = 5$	0.865506	1.307951	1.307951	$d = 9$	0.858811	1.243708	1.25665
4		1.051674	1.439334	1.547736		1.695511	2.442509	2.442509		1.293688	1.568415	1.555817
5		0.941034	1.252677	0.918457		1.429531	2.09668	2.09668		0.967001	1.405515	1.315557

Table 5.

Similarly to the previous test, all the cross-replication results are compounded and recorded in Table 6 with means, variance and confidence measures. Then they are expressed in Figure 3.

d Size	Random Minimum	σ^2 RandMin	Confidence Measure	Pure Random	σ^2 PureRand	Confidence Measure	Round Robin	σ^2 RR	Confidence Measure
<½	1.128214	0.017971	0.026274	1.434302	0.019099	0.027087	1.348195	0.078048	0.054756
½	1.094525	0.195599	0.086683	1.640287	0.350676	0.116065	1.640287	0.350676	0.116065
>½	1.16902	0.057233	0.046889	1.626481	0.114128	0.066213	1.584301	0.096321	0.060829

Table 6.

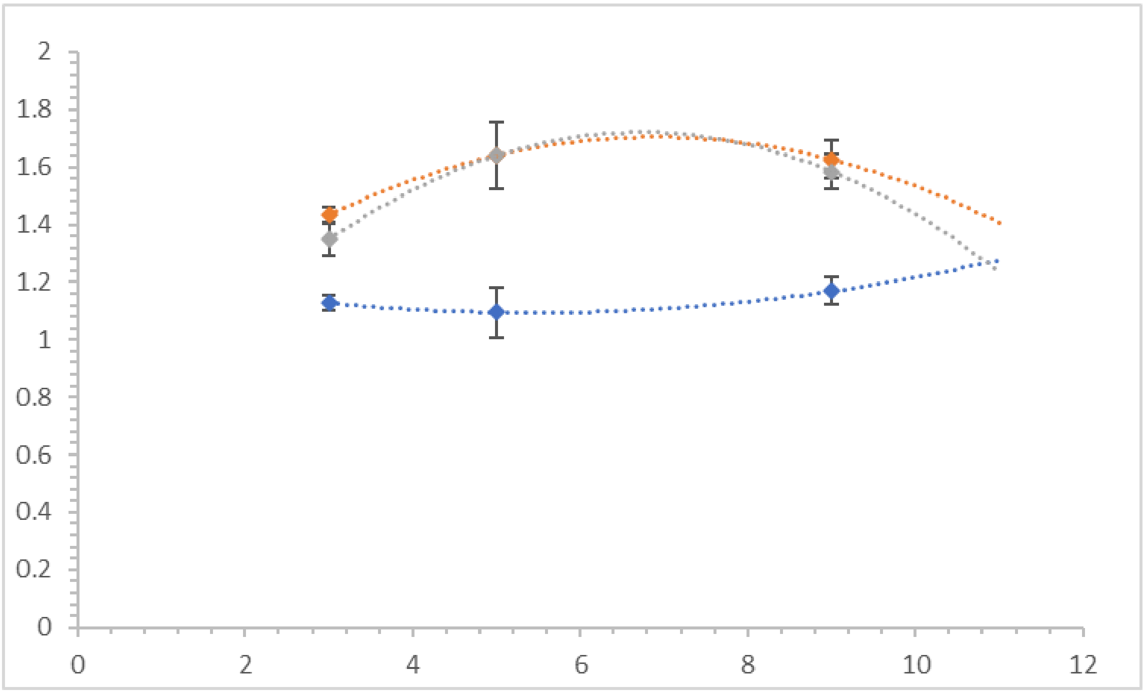

Figure 3. Graphical representation of random selection size on the X axis and average queue length on the Y.

Testing Arrival Rate

Arrival rates were measured in $\frac{minutes}{arrival}$. With regard to testing the variable independently we measured $\lambda = 0.5, 1, 1.5$ with the other system variables set to $m = 10$, $d = 5$ and $\mu = 10\frac{minutes}{service}$. These replicated tests yield Table 7, the cross-replication results yield Table 8 and Figure 4.

Seed		RM	PR	RR		RM	PR	RR		RM	PR	RR
1		2.119442	2.237018	2.578377		0.683474	1.252677	0.918457		0.743989	1.582428	1.254927
2		2.058069	2.313084	2.698904		0.798601	1.10162	0.918457		0.854998	1.33002	1.201109
3	$\lambda = \frac{1}{2}$	2.120234	2.035737	1.896097	$\lambda = 1$	0.865506	1.307951	1.251288	$\lambda = \frac{3}{2}$	0.556583	0.85632	0.804844
4		1.820727	2.300003	2.394502		1.695511	2.442509	1.928533		0.734479	1.178773	1.107325
5		1.621812	2.105064	1.689657		1.429531	2.09668	1.827243		0.616324	0.98749	0.970804

Table 7.

λ Size	Random Minimum	σ^2 RandMin	Confidence Measure	Pure Random	σ^2 PureRand	Confidence Measure	Round Robin	σ^2 RR	Confidence Measure
0.5	1.948057	0.048442	0.043138	2.198181	0.015034	0.024032	2.251507	0.192362	0.085962
1	1.094525	0.195599	0.086683	1.640287	0.350676	0.116065	1.368796	0.235724	0.095159
1.5	0.701275	0.013678	0.022922	1.187006	0.08151	0.055957	1.067802	0.033226	0.035726

Table 8.

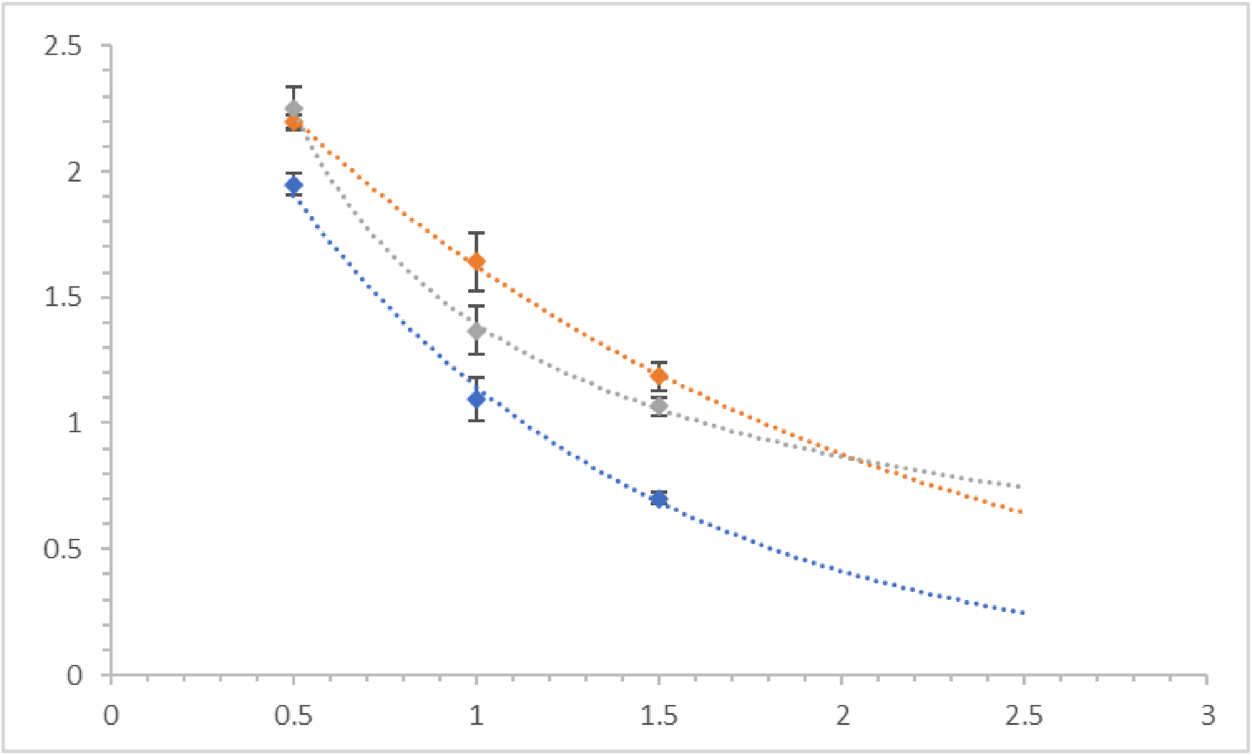

Figure 4. Trends and points relating to changes in arrival rates on the X axis and average queue length on the Y.

Testing Service Rate

Service Rates were also measured in $\frac{minutes}{service}$. In order to test this variable we will observe at different points $\mu = 5, 10, 15$ with $m = 10$, $d = 5$ and $\lambda = 1\frac{minute}{arrival}$. The initial results were stored in Table 9, with compounded means, variance and C.I. measures in Table 10 and Figure 5.

Seed		RM	PR	RR		RM	PR	RR		RM	PR	RR
1		0.535677	0.858575	0.766859		0.416203	1.252677	0.918457		2.302059	2.596332	2.09245
2		0.520149	0.810746	0.688497		0.798601	1.10162	1.034907		2.122418	2.237493	2.367539
3	μ = 5	0.520149	0.810746	0.688497	μ = 10	0.865506	1.307951	1.251288	μ = 15	1.706163	1.659041	2.039528
4		0.486137	0.664716	0.747572		1.695511	2.442509	1.928533		2.266745	1.95411	2.180842
5		0.416203	0.579157	0.626627		1.429531	2.09668	1.827243		1.584484	1.854024	1.57659

Table 9.

μ Size	Random Minimum	σ^2 RandMin	Confidence Measure	Pure Random	σ^2 PureRand	Confidence Measure	Round Robin	σ^2 RR	Confidence Measure
5	0.495663	0.002301	0.009402	0.744788	0.013873	0.023086	0.70361	0.003079	0.010876
10	1.04107	0.264817	0.10086	1.640287	0.350676	0.116065	1.392086	0.212216	0.090289
15	1.996374	0.109077	0.064731	2.0602	0.133391	0.071583	2.05139	0.085992	0.057475

Table 10.

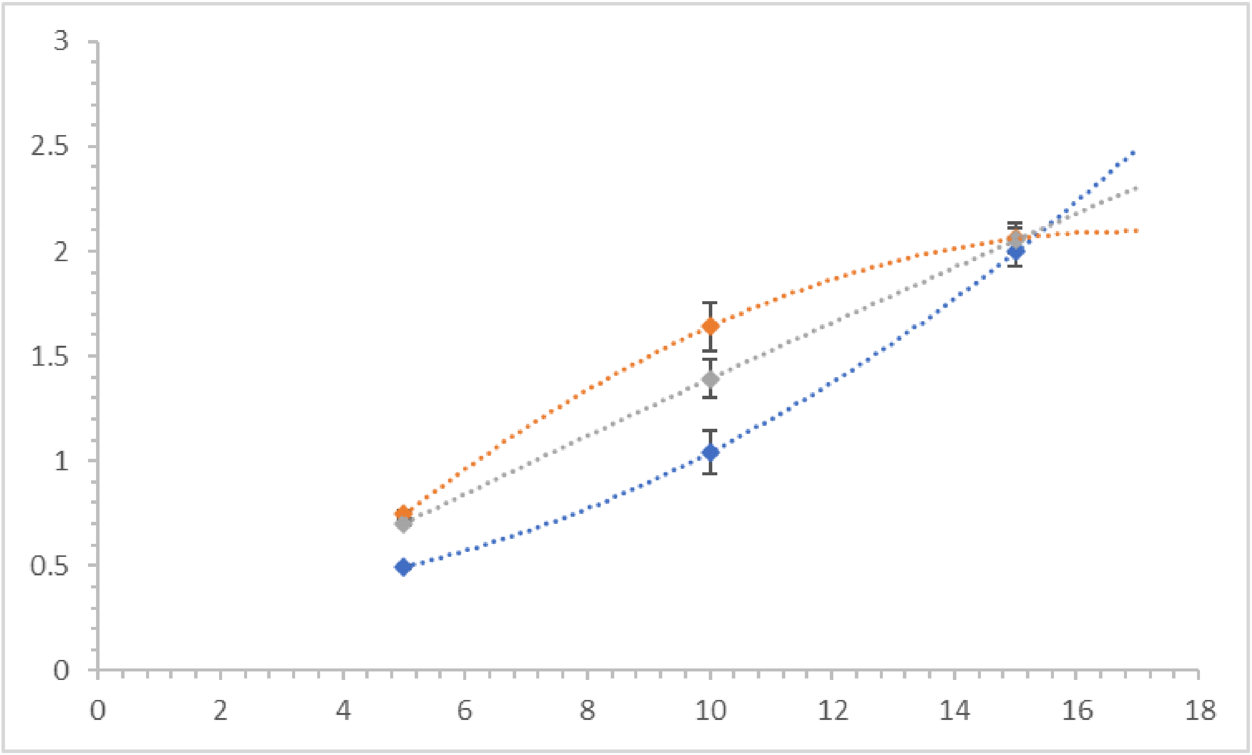

Figure 5. Predictive Graph depicting changes in Service Rates on the X axis and average queue lengths on the Y.

Analysis of Results

In the following section we will highlight some of the observations displayed within the tests. Prior to analyzing and addressing the observations, we must discuss the trend lines denoted in each other in the graphical figures. Given the limited data points, it can be difficult to properly evaluate with precision the trajectory of the variables. As such when compiling the overall data in Excel, lines of best fit were utilized so that we can estimate long-term results and further consider how our methods can be scaled and optimized further for a larger system. Based on our estimations within the corresponding figures we have the denoted trendlines, Figure 2 exhibits power function behaviours with $y = cx^b$, and Figure 3 shows polynomial behaviours with $y = a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0$. Furthermore, Figure 4 appears to follow exponential trends following the general equation $y = a^x$and Figure 5 also shows polynomial behaviours.

Server Numbers

When observing Figure 2 and the accompanying table of data, it is apparent that with a lower number of servers in play the behaviours of all methods seem to correlate to a specific degree. As a result, there is not a vast difference in the efficiency of the systems initially. Given that our sample size of customers was only 100 we tested the servers at 4, 10 and 14 percent of the total capacity so that we did not yield results with infinitely growing queues or with the majority of the queues remaining empty in order to obtain more quantitative results and trends. With an increase in the number of servers available, there is a massive change in the overall efficiency of each system with Random Minimum improving its queue lengths ever so slightly in comparison to the others. The denoted trend of the system seems to move in the same downward trajectory with regard to queue lengths. However, it tapers off with a greater number of servers as it will be more apparent that with a greater number of servers, the greater the probability that they will be empty. Of course, as stated in the accompanying paragraph, with the change in server numbers there is also a change in the number of randomly selected servers which could further contribute to the observed behaviour.

Random Selection Numbers

With regard to Figure 3, the selection of random numbers d displays a very minimal change in the difference with regard to average queue lengths since d only affects the Random Minimum method. We can observe at least in the calculated results Random Minimum displays marginally more efficient behaviours. It further provides us with a look into what happens as d approaches both 1 and the number of total servers. As observed in the graph each trend shows parabolic behaviour with both Round Robin and Pure Random methods showing a negative trend and Random Minimum showing a positive trend. Now, these trends do not infer much as to future behaviour as our tested system is limited with $m = 10$, but it seems apparent that there will be observable intersections between each of the trend lines. To further elaborate, when d is equal to 1 we know that it displays behaviour identical to the purely random method. Additionally, when d is equal to the server size at least for the first m iterations it displays identical behaviour to Round Robin (assuming relatively low variance for service times). As such we can infer that those behaviours are represented by the intersections.

Arrival and Service Rates

The behaviours measured in Figure 4 can provide us with decently sufficient inferences as to the trajectory of each method. It must be noted that Arrival Rates and Service rates have a direct correlation since the time it takes for a server to become idle again is a determinant of the selection factor in the Random Minimum method. In this regard, we wanted to give μ a sufficiently large time so that we can allow queues to grow. As such, given our graph and denoted trend lines we can observe that the efficiency of Random Minimum remains the most efficient method of utilization in comparison to the baselines. In our denoted predictive trends it continues to remain as such even as it approaches arrival rates that result in majorly empty queues. Additionally, when considering Service Rates we see that, unlike all other graphs, Figure 5 denotes an increasing trend in all regards. This is quite self-explanatory as we know that increasing service rates increases all queues as it allows for more arrivals while a single request is processed. An interesting observation is that all trendlines seem to come to the point of intersection over a greater Service Rate value. This could possibly signify that with greater and greater service times methods of distinct methods of load balancing could have minimal differences in the overall results.

Conclusion

While running the simulations it was consistently apparent that in direct comparison to the two baselines, a Random minimum method for load balancing was more efficient. In terms of Server numbers, its average queue length was 0.2544278333 less than the *RR* and *PR*. With regards to selection size, its average queue length was still 0.4150558333 less than the baselines. Based on arrival rates it was still 0.3709775 less long on average and similarly with service rates, it performed better with lines on average being 0.254357833 less long. Now, these all seem like marginal numbers, however, our test simulation was run with a consistent sample size of only 100 customers. Although our baselines within the code are sufficiently faster at computing with regard to run times, we can scale these inferences to consider the algorithmic processes in methods like load balancing over internet-accessible devices. We can consider the fact that having systematic and random elements can reduce the overall run time of specific algorithms as it does not need to scan every server available in order to perform their selection method and when we are considering internet traffic which is exponentially larger than 100 service requests every millisecond of saved computation time is needed. We can further denote that when considering factors like efficiency we can deduce that the greater the number of servers in a load balancer, the greater the efficiency and further there is an inverse relationship with regard to Service rates and Arrival rates as efficiency is optimized when service rates are lower. Arrival rates are higher and the same can be said for the inverse.

Suggestions

Our study is subject to various presumptions given that we tested with small sample size. An attempt to scale our results to higher degrees of tests can subject this study to increased error. If we were to scale this study to a vastly greater number of service requests, we would have to optimize the storage of information in order to navigate it at a much more efficient rate as loops through the data sets would experience immensely long runtimes. Multithreaded child processes of servers or Splay tree organization of the information instead of rudimentary lists would allow for such an experiment to be done at a much faster rate. Furthermore, running more tests would allow us to have created a more precise understanding of the system behaviour, especially with regard to the trendlines.

References/Resources

1. Python: https://www.python.org/
2. NumPy:

   https://numpy.org/doc/stable/reference/random/generated/numpy.random.exponential.htm

   l