Design and Analysis for Layer 7 Load Balancing Algorithms

Jingwu Xu, Rongrong Miao, Saurabh Goyal
{*jix010, romiao, s2goyal*}*@eng.ucsd.edu*

***Abstract*—The primary goal of any load balancing algorithm is to distribute the jobs among servers to maximize throughput, minimize total turnaround of requests, to match the application need with the available computing resources, maintain stability, and good resource utilization. We wanted to address the problem of dealing with the high percentile requests as they are the ones which matter for large service providers. In our paper, we analyze the performance of some common load balancing algorithms on CPU-intensive tasks, IO-intensive tasks, and mixed tasks and we propose a new algorithm that can outperform existing algorithms on requests that are beyond 90 percentiles and have fewer time outs.**

***Index Terms*—networking, communication, load balancing**

I. Introduction

With the growing demand for Internet, the amount of requests and computation at each server is rising significantly. This problem can be solved if we distribute the applications across different servers. This is done in such a manner that it reduces the request response time and the load on a single computer. It also ensures high throughput of the system as a whole.

The proper distribution of any request across different available resources is known as load balancing. To balance the load a load balancer can use different algorithms. Generally these algorithms are classified into two categories, static and dynamic. In static algorithms,the assignment of requests to the computing nodes is carried out before the execution of the program. Information regarding processing resources is known at compile time. Random algorithm and Round Robin algorithm are examples of static Load Balancing algorithms. Dynamic load balancing algorithm is based on the distribution of processes/requests among the servers during execution time. Dynamic load balancing algorithm does not take into consideration of any prior knowledge about request behavior. Load balancing decisions is solely based on the active status of the system. The development of an effective dynamic load balancing algorithm involves many important issues: load estimation, load level comparison,the stability of the system, the characterstics of the load, the estimation of the request time, inter-node information exchange, and more. Least connection based algorithm is an example of dynamic load balancing algorithm which will be explained in detail in the later section.

In our research project we did some experiments with the various load balancing algorithms. In addition, in our work we define the traffic pattern, which specifies how clients will send the request over a period of time and the request pattern, which determines the benchmark of the time it will take to execute a single request. These setups will be discussed in detail in the setup section. We also design our own algorithm which stands out for the tail percentile of the requests which matter to the huge service providers.

The rest of the paper is organized as follows. Section II talks about the related work. We talk about the technical background for load balancing algorithms in section III. We state the design goals for a load balancer in Section IV. We discuss in detail about our experimental setup in section VI and provide our results and discussions in section VII.

II. Related Work

Very early papers like [1] introduced a distributed load-balancing policy for a multi computer system. The system had a cluster of independent computers connected via a LAN. They introduced three algorithms for load balancing in this system: (1) the local load algorithm, used by each server to monitor its own load; (2) the exchange algorithm, for exchanging load information between the processors, and (3) the process migration algorithm that uses this information to dynamically migrate processes from the overloaded to not so loaded processors. In [2] stability issues for distributed scheduling algorithms are discussed. Two very different distributed scheduling algorithms which contain explicit mechanisms for stability were then presented and evaluated with respect to individual specific stability issues. [3] proposed a load balancing algorithm which is adaptive and decentralized. It used broadcast messages to read about consensus on average load in the network. In [4], they explored the use of system state information in adaptive load balancing policies. They suggested that simple load sharing policies using small amounts of information perform quite well than when no load sharing is performed and nearly as well as more complex policies. We tried to explore this same fact in our experiments. We tried to keep our own algorithm as simple as possible and we eventually got it perform as good as other algorithms and better in some cases.

III. Technical Background

In this section, we will talk about the details of various common load balancing algorithms. Our own design will be discussed in detail in section IV.

(1) **Random Algorithm** A Random load balancing algorithm just selects any of the server at Random whenever a request come. It has nothing to do with the status of the server or how the request looks.

(2) **Round Robin Algorithm** A Round Robin load balancing algorithm chooses the server in a Round Robin fashion. It doesn't distinguish on the server status . It just follows a Round Robin mechanism whenever a request dives in.

(3) **Chained Round Robin Algorithm** A Chained Round Robin algorithm is similar to Round Robin. But the only difference is in a Chained Round Robin the load is distributed periodically but in a certain ratio. This ratio is decided by the system capabilities. In the case where each system is nearly identical this ratio reduces down to 1 request for each server i.e same as Round Robin.

(4) **Least Connection Algorithm** In a Least Connection based load balancer, the load is divided to the server which has least number of active connections. Here by connection it means requests. Whenever a load balancer sends a request to any server, it increases the count of active requests to that server by one. Whenever the request is received back it decreases the number of active request by one. Whenever a new request comes from the client to the load balancer it selects the server which has least number of active connections.

Other than this, there are some other algorithms such as IP Hash based algorithm, least bandwidth algorithm, least Packet transferred algorithm, etc that we did not implement and evaluate in our project.

IV. Design Goals

The goal for load-balancing is to achieve the best allocation of the resources. Specifically, a good load balancer should have the following characteristics:

- To minimize the response time: rather than having requests queued up in a single server, using load balancers should distribute the work load and each request should be handled faster;
- To maximize the throughput: by balancing the load we aim to increase the number of completed requests per second;
- To achieve high availability: to be able to detect server fail-over and redirect requests to the servers which are alive;
- To have reliability: reduce the number of request failures such as timeouts due to high server load.

V. Statistical Latency Based Algorithm

Considering the design goals, we have designed a simple statistical Latency Based algorithm.

The high level idea is as follows: we are inspired by the TCP congestion control scheme that it estimates the RTT per packet. We also want to estimate the latency per request of each server. Our theory is that if one server has the lowest latency, this should correlate to the fact that it has the lowest load, and in order to maximize our chance for a successful response, we should forward our request to this server. Considering the time effect, we keep track of a window for latency data per server and takes the exponential average over the latency to be the base latency for one request on this server. However, we also want to include the factor of the success rate per server. We do also want to penalize for unsuccessful request for a high latency, and in our case the time out is 4 seconds. This indicates that our penalty is set to 4 seconds. To include the effect of retry, we simulate the out come of the trials as geometric Random variable X, which represents a success response. Therefore, we also keep track of the success rate of each server at the load balancer.
Define the probability of a success to be

$$Pr(X) = s = \frac{\text{number of successful requests}}{\text{total number of requests}}$$

. Since

$$E(X) = \frac{1}{Pr(X)} = \frac{1}{s}$$

, approximately after $\frac{1}{s}$ number of retries, we will get one correct response [5] [6]. Combining with the previous arguments, we can formulate the latency per request as follows:

$$l = base + p * (\frac{1}{s} - 1)$$

where base is the moving exponential average of the window, and p is the penalty, here we set it to be our time out, which is 4 seconds.

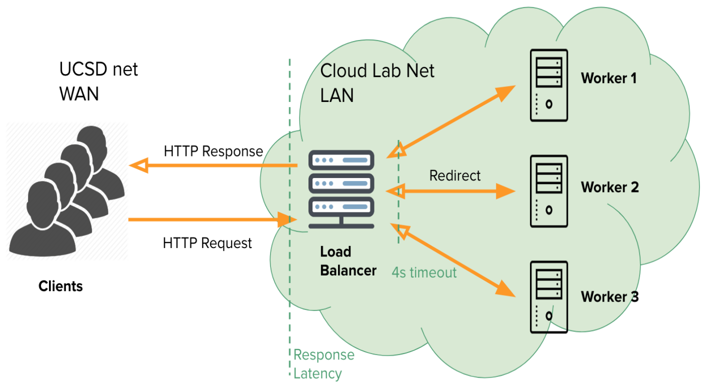
Then we want to assign weight to each server based on l computed above. When deciding which server to send for each request, a weighted Random drawing is performed for all servers, with each server weight computed as:

$$w_i = \frac{\frac{1}{l_i}}{\sum_j^n \frac{1}{l_j}}$$

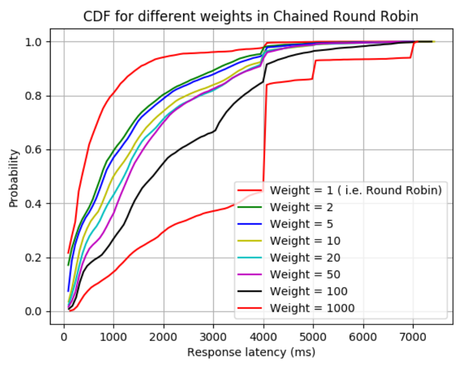
A design trade off we considered during our implementation was that whether we should reweight the server periodically or after each request or not. We were worried that reweighting the servers too often would cause too much overhead on the load balancer. We tried reweighting the servers at different time intervals, however, they do not perform as well as reweighting after each request possibly due to the fact that when server weight does not change and as a result we are bombarding one single server for a time interval. In our experiment, the reweighting did not cause too much overhead on the load balancer, and we show in our results that this scheme achieves reliability as we're improving the tail latency due to the effect of exponential averaging for latency, which our estimate makes more sense when we have more history data, and the penalty estimates, which reduces the chance of selecting a high-latency server.

VI. Experimental Setup

We set up our experiment on cloud lab machines, with a cluster of 3 workers residing on 3 different virtual machines(VMs) and 1 load balancer on separate VM, see Figure 1 (a). All virtual machines have the same configuration (4 CPU, 4 GiB memory, network connection, etc). Load Balancer communicates with workers through local-area network(LAN). There isn't much network communication overhead between workers and load balancer in our experiment. Workers and load balancer are implemented with Python-based CGI HTTP Server, which only provides the ability to process HTTP requests and to generate responses using a relatively primitive



(a) Experiment Architecture



(b) CDF of Different Weights of Round Robin

Fig. 1: Experiment Setup

API. Our server implementation is designed with the purpose of prototyping and comparing different algorithms and not for the server scalability or robustness. The Python-based CGI HTTP Server instantiates concurrent threads to handle received requests, and all the threads need to share the global computation, memory and network resources that available on the VM. We implemented five load balancing algorithms, namely, Random, Round Robin, Chained Round Robin, Least Connection, and latency-based. An efficient load balancing algorithm should be able to the achieve the best allocation of available resources.

Request latency is computed as the round trip time (RTT) per request that goes through the load balancer. More specifically, the request latency starts once the load balancer takes in an client request, ends until it sends back the response, see Figure 1. During that period, load balancer needs to determine which server to redirect the request based on current load balancing algorithm, redirect that request and wait for worker's response, copy the response and send back to the client. We manually set a 4-sec http request timeout between load balancer and worker.

We use our own machine as client to flood http requests to the load balancer via wide-area network(WAN). The requests to load balancer follows a simplified version of real-world traffic to web servers [7] – sine wave. The base request speed on our client is 40 requests per second (rps), and the peak request speed is 100 rps. Requests attack load balancer continuously for 1 minute with a total number of 4032. The majority of requests can be processed in a short amount of time, while the minority requests takes longer. We implemented this request pattern by attaching a type for each request, where the type is drawn from a log-normal distribution with $\mu = 3.7$ and $\sigma = 1$. Typically in our experiment the workload of a request ranges from 0.01s to 0.2s. We simulate the request workload by adding dummy computation (conditional loops) and IO (files reading) task on the worker. Because our fake requests do not have text body in response, generally speaking, our requests could not congest the network bandwidth.

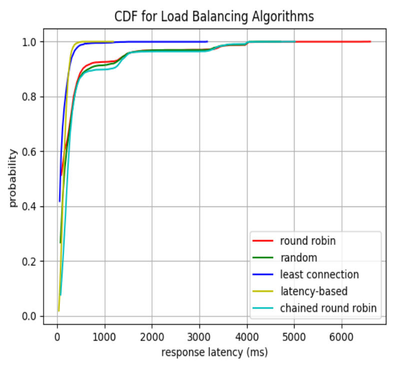
In our setup, we used uniform weight in Chained Round Robin algorithm for the following reasons. In the Chained Round Robin algorithm weights are assigned proportional to the server capacity. This means that the server which responds faster to requests in an ideal situation is given more weight. For our system setup, we figured out the ratio of requests to be 0.96:1.09:0.81 which clearly shows a weight of 1:1:1 (Round Robin) would achieve the best result. In Figure 1 (b) it shows that just keeping the same ratio and increasing that in proportion decreases the performance. This is due to the fact when we have a ratio we would want to reduce it to the smallest possible because we would not want to flood our server with a large batch of requests. This explains in our result section why Chained Round Robin did not outperform Round Robin in terms of request latency.

VII. Results

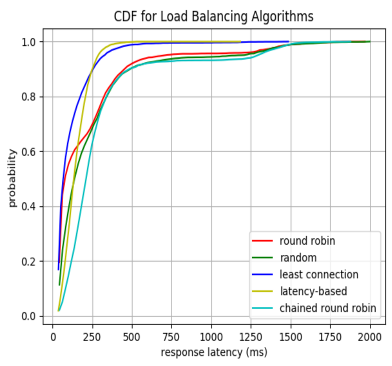
The goal of this project is to find out what is the best load balancing algorithm given a specific request pattern and the request type. We test the performance of different load balancing algorithms from different aspects, namely, request latency, request latency percentiles, connection timeouts, and connection failures. We come up with mainly three different real-world request scenarios, computation intensive, IO intensive, or mixed tasks. We have conducted separate experiment for each scenario and compared the performance of five load balancing algorithms as mentioned above. The algorithm we proposed, namely the statistical latency-based algorithm, has achieved comparable good overall performance, but significantly reduces the latency for the higher percentile of requests, and total number of request timeouts.

A. *Computation Intensive*

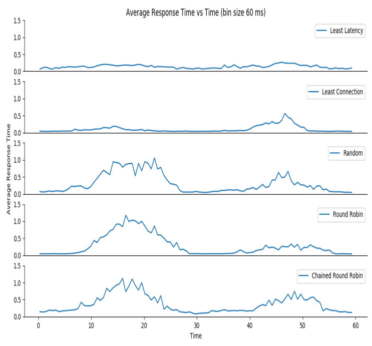
For computation intensive scenario, we adjust our requests to be pure computational task, the conditional loops. As explained before, we use parameter *type* to specify the task complexity. Here the type is drawn from a log-normal distribution with $\mu = 3.7$ and $\sigma = 1$ and then get translated into number of conditional loops by taking a customized sigmoid function derived from $\frac{x}{1+|x|}$. The computation time for each request varies from 10ms to 200ms. In this scenario, requests are competing for the computation resources that available on



(a) CDF

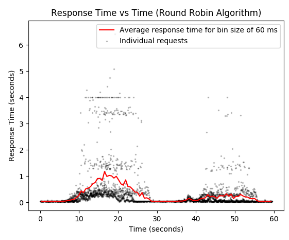


(b) CDF Expanded

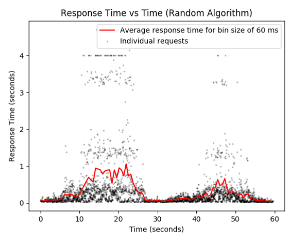


(c) Avg. Response Delay

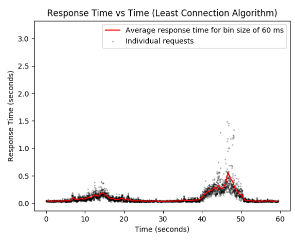
Fig. 2: Performance of Load Balancing Algorithms on Computation Intensive Tasks



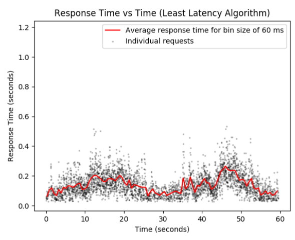
(a) Round Robin



(b) Random



(c) Least Connection



(d) Latency-Based

Fig. 3: Response Time per Request across Time

single Python-based CGI HTTP Server process, which is 4 CPU in our VM setting. The main cause of worker delay is the shared computation resources, and is linearly increased regarding to the number of concurrent requests that being processed at worker server.

In the computation intensive setting, we first compare the load balancing algorithm regarding to the response latency, and we show the results as an cumulative distribution function(CDF) across time, see Figure 2. As can see from Figure 2 (a), Round Robin, Random, and Chained Round Robin all have similar request latency, where Chained Round Robin performs slightly worse than Random and Round Robin. On the other hand, Least Connection and Latency-Based load balancing algorithms have an obvious advantage over other three algorithms as they can process most of the requests in less than 500ms, while the other three algorithms could only guarantee 88% of the requests being handled in less than 500ms. More interestingly, we can see that Latency-Based algorithm outperforms the Least Connection algorithm when reaching around 90 percentile of the total requests, from where Latency-Based has higher percentage of requests get processed compared to Least Connection in the same amount of delay. Figure 2 (b) provides a zoomed in view of the requests latency between 0 to 2000ms based on (a). From that we can see Latency-Based algorithm does not win on the earlier percentile of request latency, while it wins on reducing the latency for requests in later percentile.

We further compare the average response delay for different load balancing algorithms versus the time we flood requests onto load balancer, see Figure 2 (c). The general shape of the average response time reflects the traffic pattern we used to send requests. We can clearly see from the Chained Round Robin plot that the average response delay increases as the number of requests increases, and that pattern is exactly the sine wave we used to generate client traffic pattern. Our assumption is that the load balancer can analyze the average response time graph and predicts what the incoming traffic would be, and dynamically allocates more resources to reduce request latency.

Figure 2 (c) also provides explanation on how each algorithm performs when dealing with the same requests. Our proposed algorithm, Latency-Based, and Least Connection algorithm both have stable average response delay during the period we flood the requests. However, Latency-Based is able to distribute requests more decently than Least Connection does, and hence there is not an sudden increase on the average response time in Latency-Based plot compared to Least Connection plot around 45s, which is the second peak of our traffic pattern. As one may notice, Round Robin unexpectedly has much lower average response delay around 45s compared with Random and Chained Round Robin, and that explains why Round Robin handles more requests within 250ms over Random and Chained Round Robin, see Figure 2 (b). This is probably because the requests we generated at client by chance in favor of the Round Robin fashion near the second peak. As a result, Round Robin has lower average response delay than

TABLE I: Computation Intensive Requests Statistics

Request Response Time (ms) Percentile Table

	50 percentile	90 percentile	95 percentile	99 percentile
Latency Based	133.94	**250.93**	**287.45**	**1183.64**
Least Connection	**67.35**	255.57	327.62	3166.11
Round Robin	84.15	539.28	1421.21	6606.22
Random	145.78	658.45	1432.10	4710.17
Chained Round Robin	216.54	1096.86	1469.96	5109.34

Request Timeouts

	Number of Time Outs
Latency Based	**1**
Least Connection	**0**
Round Robin	40
Random	43
Chained Round Robin	30

Chained Round Robin and Random at that period.

We next compare the reponse time per request during that 1 min traffic period for all algorithms except Chained Round Robin, see Figure 3. Each black dot in the plot represents a request, where x-value is the time it arrives at load balancer, and y-value is its reponse delay. From plot (a)(b) in Figure 3, we can see four visible clusters, the bottom cluster has the majority of the requests, which can be processed in less than 1 sec. Another two clusters above it are in latency range from 1s to 2s and from 3s to 3.5s. These clusters are caused by the requests retrasmission from load balancer to worker when some server reaches its maximum connection limmits. The top cluster forms a strait line with 4 sec latency, and that is caused by the hard timeout we set from load balancer to each worker.

Recall in the experiment set up, request latency is computed as the time difference between the load balancer takes in a request and sends it back, and it includes the delay from load balancer to worker, which is the major delay in our setting, but also includes selecting next server and sends back response to the client. During the peak time of request traffic, load balancer may fall into the situation it receives huge amount of response back from worker instantly and costs another delay on sends those response back to each client. That explains why there are few requests have higher response delay above 4 second timeout limit, see Figure 3 (a). Furthermore, there is an interesting pattern in 3 that we can see several bands on graph (a) and (b). We have included further discussion about this phenomenon in the appendix.

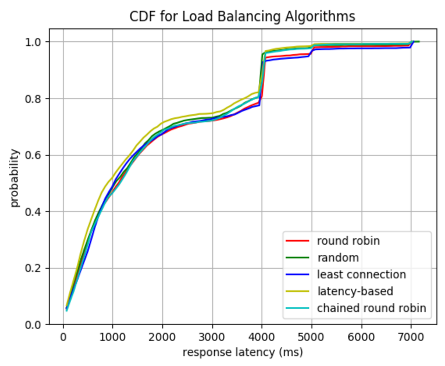
The response time per request plot for Latency-Based and Least Connection algorithm again explains why our Latency-Based algorithm outperform Least Connection on dealing with higher percentile requests. All the requests using Latency-Based algorithm can be handled within 0.5 sec, while in Least Connection there are few requests that need to be processed longer than 0.5 sec, which we called high response delay requests.

To get a better understanding of the latency improvements on higher percentile of requests, we conclude the request latency statistics table, see Table I. In the computation intensive setting, we can see Least Latency has the smallest reponse time 67.35ms for 50 percentile of the requests, while our Latency Based algorithm is not winning at this percentile. However, the Latency Based algorithm has achieved slightly better response time for 90 percentile and 95 percentile compared to Least Connection, and an almost 3x reduction on 99 percentile latency compared to the second best algorithm, 1183.64ms compared to 3166.11ms. This makes us strongly believe in computation intensive setting, the Latency Based Load Balancing algorithm could significantly helps reduce the response latency at higher percentile of the requests. Another advantage we have over other common load balancing algorithms is that, because we reduce the latency for higher tier of requests, we have much fewer request timeouts, see Table I. We manually set a 4s timeout from load balancer to worker, as can see both Least Connection and Latency Based algorithms have 99 percentile latency fewer than the timeout limit. Since the computation task in this setting is not heavy (10ms to 200ms) enough to show the timeouts difference between Least Connection and Latency Based algorithms, both of them have close to zero timeouts. We have also conducted experiments with heavier computation task and the results have shown our algorithm is able to achieve fewer timeouts than other four load balancing algorithms. Due to the page limit, we do not show the detailed results here.

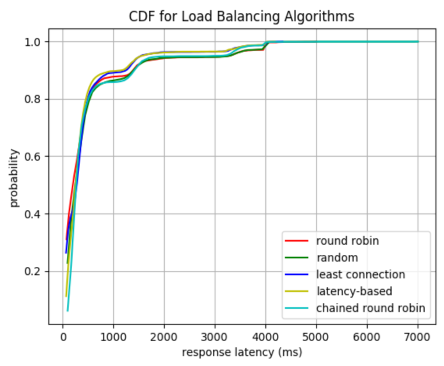
B. IO Intensive

For IO intensive scenario, we adjust our requests to be pure IO task, file reading. Using the *type* parameter generated from log-normal distribution, the request needs to read a certain fraction size of the file from arbitrary offset, based on value of *type*. The IO time for each request varies from 100ms to 200ms. In such scenario, each request is competing for the memory resources that available on single Python-based CGI HTTP Server process, which is 4 GiB in our VM setting. The main cause of worker delay is the shared IO bandwidth, L3 cache, and file lock. However, the work delay is not linearly related to the number of concurrent requests on worker server. Because our simulated IO task is reading fraction of file from an arbitrary offset, if two requests happen to retrieve data that share large overlaps, it will be significantly faster than two requests with no overlap on their target data due to the efficiency of reading from L3 cache compared to hard-disk. Because of that, the number of concurrent requests that being processed at worker server may not be the correct reflection on how many workload that a worker owns.

In the CDF plot for IO intensive task, as shown in Figure 4 (a), all five algorithms have almost similar performance regrading to the response latency probability. One reason is due to the less variation on the request complexity, 100ms to 200ms. Another reason is most of the time, requests are waiting for the file lock to access that portion of the file it wants to read. With a closely look at the IO CDF plot, we can see that the Latency-Based algorithm slightly outperforms other four algorithms but in an in-negligible way. Least Connection algorithms failed in this scenario because it mainly explains the share of computation resources, so it degrades to a normal Random or Round Robin like algorithm. However, our proposed algorithm, not limited to any specific type of



(a) IO Task CDF



(b) Mixed Task CDF

Fig. 4: CDF for IO Intensive and Mixed Tasks

TABLE II: CPU-IO Mixed Requests Statistics

Request Response Time (ms) Percentile Table

	50 percentile	90 percentile	95 percentile	99 percentile
Latency Based	254.21	1161.81	1555.71	4189.78
Least Connection	277.59	1255.77	1544.45	4341.26
Round Robin	202.49	1392.70	3265.54	5004.92
Random	249.78	1398.15	3274.87	7009.23
Chained Round Robin	263.44	1407.81	2916.09	7011.2

Request Time Outs

	Number of Time Outs
Latency Based	**47**
Least Connection	44
Round Robin	108
Random	105
Chained Round Robin	51

TABLE III: IO Intensive Requests Statistics

Request Response Time (ms) Percentile Table

	50 percentile	90 percentile	95 percentile	99 percentile
Latency Based	**898.37**	4007.66	**4011.78**	7025.17
Least Connection	1021.83	4008.21	5002.77	7054.68
Round Robin	1116.33	4007.689	4460.59	7020.3
Random	1058.68	4006.86	**4010.54**	7161.73
Chained Round Robin	1156.62	4006.99	**4011.62**	7023.15

Request Time Outs

	Number of Time Outs
Latency Based	**615**
Least Connection	784
Round Robin	766
Random	690
Chained Round Robin	672

task, learns the server status from its previous experience and dynamically adjust the weight of each server according its statistical latency. In this manner it is able to make an more dynamic balance of the workload for each server.

To analyze the latency performance of different algorithms in IO settings, we generate the latency statistics table, see Table III. In table (a), We can see that in this setting, our proposed Latency-Based algorithm is actually outperforming all other load balancing algorithms in the lower percentile of the requests, 898.37ms for 50 percentile of requests compared to $>$ 1000ms using other algorithms. And all algorithms have similar request response time for the higher percentile of requests. Nonetheless, our Latency-Based algorithm has fewer timeouts compared to other algorithms, nearly 10% less than second best algorithm Chained Round Robin.

C. *CPU Intensive and IO Intensive*

In this mixed scenario, each request can be either CPU intensive or IO intensive. We have the same experiment variables as the previous two scenarios. Unsurprisingly, the result has the characteristics for pure CPU and IO intensive task. The CDF is shown in Figure 4 (b). As can be seen from the graph, all algorithms perform similarly, which is due to the IO tasks involved. However, Least Connection and Latency Based algorithm also outperform the rest, which is the characteristic for CPU-intensive tasks. The overall latency for each algorithm is shorter than pure IO task, but Latency Based algorithm still gives reliability by reducing tail-latency in classes of requests that are beyond 90 percentile, see Table II.

Interestingly, the number of timeouts for Chained Round Robin is approximately half of Random and Round Robin. We have come up with a hypothesis to explain this situation. The main idea is that in Chained Round Robin scheduling, each server's load changes periodically while the server load for Round Robin algorithm increases as the number of requests increases.

Our argument is that Chained Round Robin is like a greedy algorithm that bombards the servers alternatively until one server cannot take more requests, then we move on to the next server. However, in reality we do not flood the first server until it is unresponsive, and in our case we limit the batch size to be the weight. In this case, if server i has to deal with a request batch, the server load increases as the competition for resources becomes more intense, and as a consequence the latency for individual request increases. However the timeouts do not increase because when we bombard server i, its status was most likely idle. Perhaps in this scenario the load for each server changes alternatively between high load and idle. In contrast, for Round Robin algorithm, the server load increases as the number of requests queued up at the server. Therefore, when the server is at high load, even if we send a request that requires minimum computation time, the server might not be responsive and this request will likely suffer from a time out. The possible evidence to back up this theory is that the reduced number of time outs for Chained Round Robin is more obvious in the case of IO-intensive tasks rather than CPU-intensive tasks, as IO-intensive tasks take longer to complete. Therefore, in Round Robin scheduling, with the increase of server load, the likelihood of a timeout also increases non-trivially as the marginal increase of request time is higher.

VIII. Future Work

Here we list several problems we run into when doing this project and leave them for future improvement.

(1) We were not able to measure the instant CPU load using the built-in library in Python. Since we're using Python, the programming language gives a large delay, therefore, the CPU statistic we got might experience non-trivial delay. Secondly, there's no such thing as instant CPU load, as the number provides by the system has been averaged and smoothed. Furthermore, since Python cannot make use of multiple cores in a CPU because of GIL, but the CPU statistics we got was averaged over all cores in a CPU. Thus, the statistics was inaccurate and not useful. If we want to include CPU metric in our load balancing scheme, we need to divert to another measure.

(2) In the future, we can take advantage of parallel computation for the requests by implementing in a parallel language or include multiprocessing modules in Python.

(3) We also have many theories that we come up with discussed in previous sections as well as the appendix that we did not have time to design experiment to verify them. For instance, why Chained Round Robin has higher lantency but less timeouts compared to Round Robin algorithm.

(4) We can also extend our experiments to different traffic and workload pattern.

IX. Conclusion

In conclusion, the choice of load balancing algorithms truly depend on the type of request. For example, if the requests are mostly short and sparse, perhaps the provider can get away with naive Round Robin or Random scheduling algorithm. On the other hand, if the provider already knows the hardware differences between the servers and are able to quantify them, it makes more sense to use the Chained Round Robin by setting an optimal weight. Our statistical latency-based algorithm do not provide outstanding performance in terms of short requests ($<$ 50 percentile) but provides reliability as we reduce the tail latency and reduce the number of time outs. Therefore, it is suitable for service providers who aim to provide high reliability service.

Acknowledgment

The authors would like to thank Alex Snoeren and Rajdeep Das for their key insights during every phase of this project.

References

[1] A. Barak and A. Shiloh, "A distributed load-balancing policy for a multicomputer," *Softw. Pract. Exper.*, vol. 15, no. 9, pp. 901–913, Sep. 1985, ISSN: 0038-0644. DOI: 10.1002/spe.4380150905. [Online]. Available: http://dx.doi.org/10.1002/spe.4380150905.

[2] J. A. Stankovic, "Stability and distributed scheduling algorithms," *IEEE Trans. Softw. Eng.*, vol. 11, no. 10, pp. 1141–1152, Oct. 1985, ISSN: 0098-5589. DOI: 10.1109/TSE.1985.231862. [Online]. Available: http://dx.doi.org/10.1109/TSE.1985.231862.

[3] P. Krueger and R. Finkel, "An adaptive load balancing algorithm for a multicomputer," Jan. 1984.

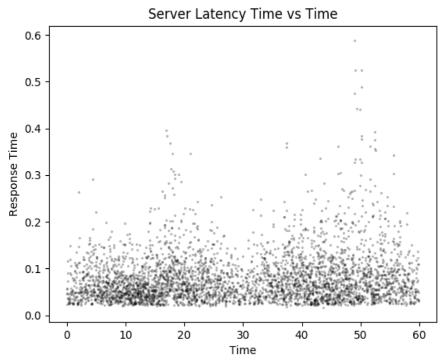
[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Softw. Eng.*, vol. 12, no. 5, pp. 662–675, May 1986, ISSN: 0098-5589. [Online]. Available: http://dl.acm.org/citation.cfm?id=5527.5535.

[5] M. D. Mitzenmacher, "The power of two choices in randomized load balancing," AAI9723118, PhD thesis, 1996, ISBN: 0-591-32091-6.

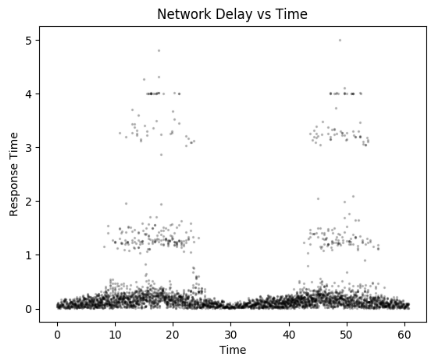
[6] *ELS: Latency based load balancer*, https://labs.spotify.com/2015/12/08/els-part-1/, Accessed: 2019-03-20.

[7] *Predicting trends*, https://www.oreilly.com/library/view/the-art-of/9780596518578/ch04.html, Accessed: 2019-02-21.

APPENDIX
FURTHER EXPLANATION OF CLUSTERS IN RESULTS



(a) Server Processing Latency



(b) Network Latency

Fig. 5: Server Processing Latency vs. Network Latency

It is noticeable that in Figure 3, we can see distinct bands on graph (a) and graph (b). Initially we found this pattern perplexing as there does not exist three modes in our server processing time distribution. We investigated the situation further as follows and formed a theory.

$$\text{total latency} = \text{network latency} + \text{server processing latency}$$

We measured the total latency at the load balancer, and we also measured the server processing latency on the server side, which was attached in the response to the load balancer. From these information and the equation above, we can deduce the network latency. We rerun the one minute experiment using the Chained Round Robin algorithm as this pattern was typical in this algorithm. We graphed the distribution of server processing time and the network latency for the same set of requests. As is shown in Figure 5, the majority of the latency attributes to the network, rather than the server. Furthermore, in graph (a), there's no visible cluster, but in contrast, the network latency graph clearly displays the three clusters together with the time out line as we have seen before. Therefore, we conclude that our set up and implementation did not cause the cluster but rather the network is to blame. We hypothesized that it could be due to the high latency of some requests, the network experienced TCP retransmissions.

One question the reader might ask is that why this pattern does not show up in the Least Connection and Latency Based algorithm. We make the argument that the Least Connection and latency-based algorithm were able to return almost all requests within 1 second, therefore, there latency was very short and they might not experience TCP retransmission problem. Further experiments and analysis are required to verify this hypothesis.