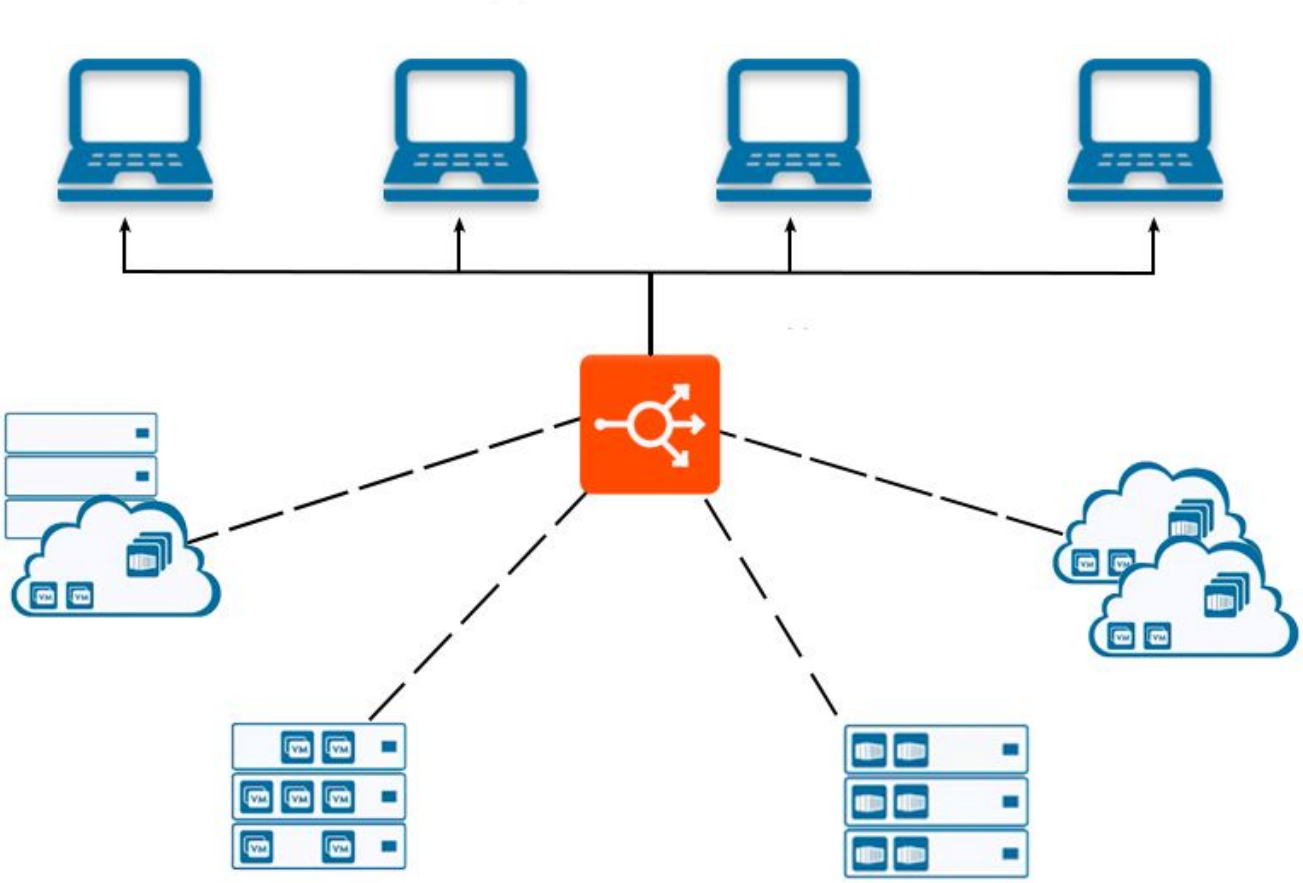
Load Balancing Strategies, Algorithms and Comparison

CSCI-ECEN 5273: Network Systems
Alex Costinescu and Suman Hosmane



I. Introduction

In the age of smartphones, laptops, and IOT, the number of internet-connected devices has skyrocketed. As internet devices become increasingly accessible and user-friendly, the number of users is bound to continue increasing. These users also span the entire globe and increasingly travel to new destinations, taking their connected devices with them.

Given the heavy and geographically diverse loads that modern services face as a result, it has become infeasible to run them on a single server or even in a single location. However, having many IP addresses for a single service is also not practical. This is where load balancers come into the picture. By obfuscating many servers behind a single IP address, load balancers enable services to scale to serve thousands or even millions of users. Additionally, given their dynamic nature, load balancers also allow for individual servers to be taken offline for maintenance or replacement transparent to users and other services that are accessing the system.

Summing up the role of Load Balancers: they are the network infrastructure enabling traffic division based on programmed aspects like response times, state, geographical location, etc. Additionally, in conjunction with load balancers, the recent advent of cloud technology allows servers to be commissioned and decommissioned on-the-go, making infrastructure more elastic and efficient. This architecture is hard to implement using a hard-wired approach, while a programmable load balancer is the right way to facilitate elasticity in the network.

With the role of load balancers specified, we can move on to the technical aspects, implementations and possible load balancing strategies used. Load balancers can be implemented on independent hardware, or as software running on another switch/server instance. Since server clusters and IPs within an organisation are open to restructuring, it makes sense to have programmable load balancers, rather than an ASIC which needs to be replaced/reprogrammed every time a server rack is shuffled around. There are many possible load balancing strategies that can be used to route connections between clients and servers. Some of the common methods used in load balancers are:

- Round Robin
- Weighted Round Robin
- Least Connections
- Chained Failover
- Weighted Response Time
- Least Response Time

These strategies can be classified into static load balancing and dynamic load balancing. A static architecture is a feedback-less approach, where requests are forwarded to servers irrespective of the server's state and performance. Round-Robin is a static load balancing strategy. On the other hand, dynamic load balancing may consider several aspects, such as

whether the server is online, number of requests already being served by a server, it's performance, geographical closeness to the requester, etc. There is feedback involved in the load balancing, hence the name dynamic load balancing. Least Connection and Chained Failover are examples of the dynamic approach.

We implemented a basic network architecture on GCP (Google Cloud Platform) consisting of a single VM instance running the load balancer application and three Debian Linux VM instances serving the requests directed by the load balancer. We implemented 3 load balancing strategies, namely Round-Robin, Least Connections and Chained Failover. This approach helped us cover both static and dynamic load balancing strategies. We then tested each strategy, recording response times and failure rates, and compared their performance to each other as well as to a single server without any load balancing.

II. Related Work

F5 Networks[1] load balancers can be found in various industry setups, including leading corporates. Their BIG IP Local Traffic Manager enables users to control network traffic, selecting the right destination based on server performance, security, and availability. This dynamic approach increases uniform use of hardware and increased efficiency, while giving the flexibility of "sticky" sessions. F5 have extended their load balancing to F5 BIG IP DNS architecture for establishments with data centres spread across the globe. This architecture employs topology-based load balancing to inspect a user's IP and determine the most efficient data center. F5 also offers a great deal of flexibility, in terms of the platform of implementation. The services are spread across Cloud, Virtual Editions on hypervisors and any industry certified proprietary hardware fine-tuned for efficient performance.

Nginx[2] is one of the big names in the load balancing arena. Nginx is used as a very efficient HTTP load balancer to distribute traffic to several application servers and to improve performance, scalability and reliability of web applications with nginx. The product provides 3 major load balancing strategies, round-robin, least connections and ip-hash. While the first 2 strategies do not guarantee the requests from a single user to always be handled by a particular server, ip-hashing can implement this setup by using "sticky" or "persistent" sessions. With the help of a reverse proxy implementation, nginx includes in-band server health checks. If the response from a particular server fails with an error, nginx will mark this server as failed, and will try to avoid selecting this server for subsequent inbound requests for a while, until the server can self-diagnose or is reset by a system administrator.

[1]F5 Networks: https://www.f5.com/services/resources/glossary/load-balancer
[2] Nginx: http://nginx.org/en/docs/http/load_balancing.html

III. System Design

Below is a very high level architecture overview of the position and role of load balancers in the networking environment.

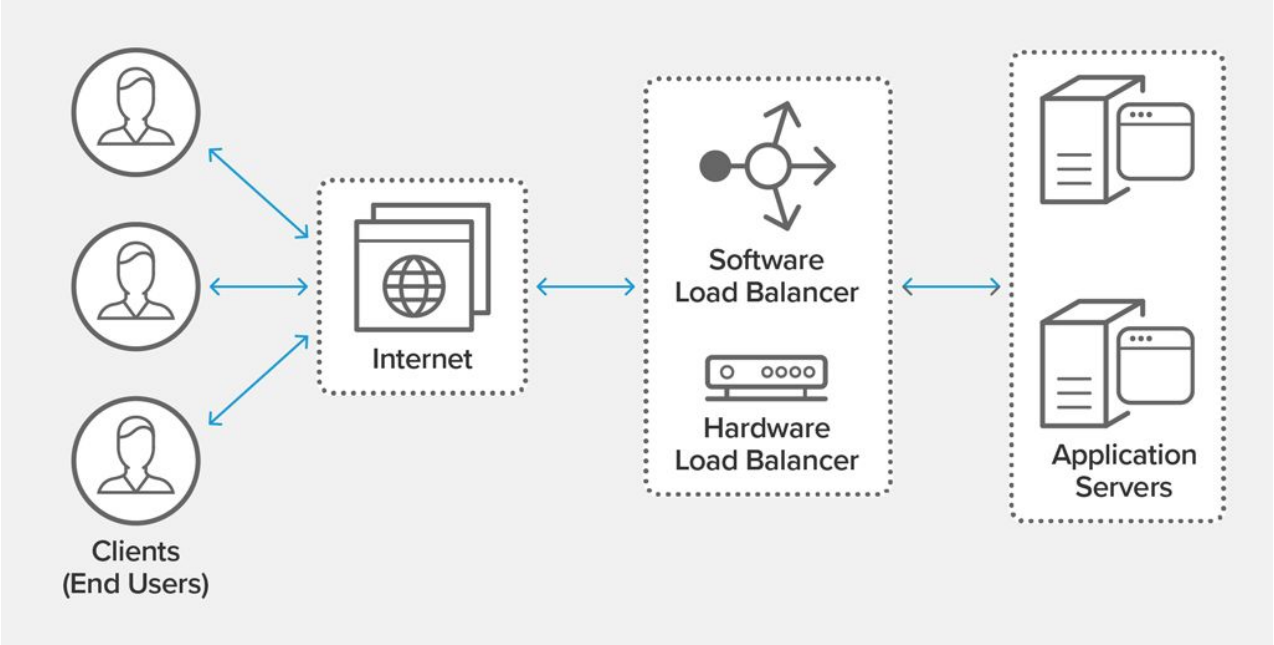


Figure 1: Load Balancing System Architecture

There are, in general, two types of load balancers. First is a proxy load balancer, also known as a “one-armed” load balancer. This type of balancer receives a request from a client, selects a server from its pool, and then finally forwards the request to that server. The server then processes the request and sends a response directly back to the client. This minimizes load on the load balancer since it only needs to handle traffic in one direction (thus “one-armed”), but also requires that every server is publicly accessible. Proxy load balancers also require IP/Port spoofing, since the client is expecting a response from the IP/Port of the load balancer, but the response is being sent from a different IP/Port.

The second type of load balancer is called an inline load balancer. This type of balancer also handles delivering the responses from servers back to the client. This means that the balancer must now handle effectively twice as much traffic, but also reduces complexity by removing the need for IP/Port spoofing. This type of balancer can also be used as a NAT gateway, which means that individual servers do not need to be assigned public IPs.

For our project, we decided to build an inline load balancer using Python. We chose this because of its relative simplicity, as we are both relatively new to networking and wanted to gain

a better understanding of load balancers before attempting something more complex. Python was chosen because it is a language that both of us were comfortable with and also provides socket libraries which are sufficiently low-level for a simple load balancer.

To receive requests, our balancer implements a basic multithreaded TCP server using Python's built-in TCP sockets. We chose to use TCP sockets instead of an HTTP server because this allows us access to the transport layer, which includes the client's IP address. This would potentially allow us to add IP hashing as a balancing strategy if we would like to add it in the future. This also means that there is less processing occurring between the request being received and our load balancing handling it, which improves response time.

The main thread listens for requests and, upon receiving one from a client, spawns a child thread which is responsible for handling the rest of the interaction between client and server. This child thread parses the request received from the client, builds a new request to send to the server, and then awaits a response from the server. Once the server has finished processing the request, it sends a response back to the child thread which then repackages the response to be sent back to the client.

To choose which server should be selected to process a response, we have built 3 basic strategies into our load balancer. These are:

1. **Round Robin:** Rotates through servers on each request.
2. **Least Connection:** Selects the server which is currently processing the fewest requests.
3. **Chained Failover:** Rotates through servers until it finds one that provides a successful response (or until it has tried every server).

Users provide the load balancer with a list of servers to select from as well as which strategy (or mode) the load balancer should use before starting the balancer.

Our load balancer also has the ability to run in a verbose mode for debugging. This prints every received response as well as the interaction between load balancer/server for those responses. For actual use, this mode would be disabled, as printing to the console increases CPU usage.

If a server is not found using the selected strategy or the selected server times out, the load balancer builds a generic 404 response and delivers that back to the client.

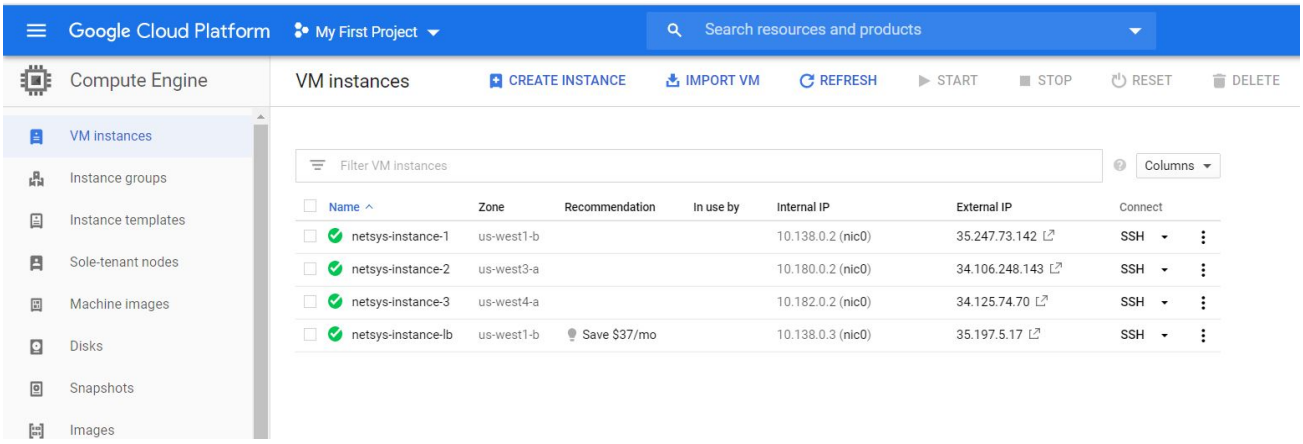


Figure 2: GCP infrastructure for the project

For our testing implementation, we spun up 4 Debian Linux instances on the Google Cloud Platform. The three 1-vCPU & 600MB memory instances emulated servers, while the single, 2-vCPU & 3.75 GB memory machine hosted our load balancer. While in a real-world scenario the servers would likely be far more powerful than the load balancing hardware, our setup was aimed at making the implementation more cost-effective given our limited GCP budget. Since the servers would not be handling any intensive requests or connecting to back-end application servers, we could test the performance with relatively low-specification hardware.

IV. Evaluation/Findings

To evaluate our load balancer, we built a tool in Python which can send N requests every X seconds for Y number of tests. This tool is also multithreaded and times how long it takes for every thread to receive a response from the load balancer (or time out). Fails and successes are aggregated, with fails consisting of timeouts and 404 responses and successes consisting of 200 responses. The tool then prints the total fails, successes, and time.

We tested each method of load balancing as well as a single server running without a load balancer. Behind the load balancer we had three servers running Python HTTP servers which received requests, ran some intensive code which calculated prime numbers, and then sent a response with the total calculation time. While this does not reflect a real-world scenario, it allows us to achieve an exaggerated view of how each strategy performs. The load balancer and servers all ran on GCP instances as detailed above.

We tested three different scenarios for each strategy. First, a short burst of traffic was simulated using 10 requests sent 0.1s apart. Next, to simulate a slightly longer burst of traffic, we sent 20 requests 0.1s apart. Finally, to simulate consistent traffic over a longer period of time, we sent 100 requests 1s apart. We simulated the 10 request burst 9 times, while the other two forms of traffic were simulated 3 times each. For each simulation, we recorded the successes, fails, and total time.

	Round Robin			Chained Failover			Least Connection			Single Server (No Load Balancer)		
	Success	Fail	Time (ms)	Success	Fail	Time (ms)	Success	Fail	Time (ms)	Success	Fail	Time (ms)
10/0.1s	10	0	19,846.013	10	0	72,863.298	10	0	21,626.221	10	0	138,801.741
	10	0	26,406.813	10	0	46,392.713	10	0	86,559.546	9	1	246,480.760
	10	0	30,653.830	10	0	36,748.126	10	0	99,429.197	9	1	276,613.528
	10	0	104,600.618	10	0	31,746.135	10	0	26,730.617	10	0	117,766.301
	10	0	32,213.435	10	0	74,727.364	10	0	149,212.554	9	1	343,706.692
	10	0	66,312.834	10	0	122,686.518	10	0	140,028.309	10	0	229,351.169
	10	0	25,221.648	10	0	23,453.476	10	0	27,976.706	10	0	178,357.867
	10	0	77,182.612	10	0	68,910.071	10	0	81,263.505	9	1	243,812.657
	10	0	28,448.789	10	0	106,587.926	10	0	145,883.292	10	0	162,269.259
	10.000	**0.000**	**45,654.066**	**10.000**	**0.000**	**64,901.736**	**10.000**	**0.000**	**86,523.327**	**9.556**	**0.444**	**215,239.997**
20/0.1s	20	0	291,607.179	20	0	198,498.021	20	0	186,483.931	11	9	284,905.522
	20	0	208,975.104	20	0	221,480.972	20	0	153,286.846	10	10	416,476.170
	20	0	287,895.402	20	0	248,243.986	20	0	236,927.262	9	11	415,224.311
	20.000	**0.000**	**262,825.895**	**20.000**	**0.000**	**222,740.993**	**20.000**	**0.000**	**192,232.680**	**10**	**10**	**372,202.001**
100/1s	39	61	398,264.624	63	37	766,614.927	44	56	405,200.867	9	91	317,444.068
	40	60	388,742.768	69	31	820,131.819	44	56	438,174.012	13	87	412,623.009
	35	65	363,483.891	61	39	770,991.959	41	59	340,166.042	9	91	333,597.478
	38.000	**62.000**	**383,497.094**	**64.333**	**35.667**	**785,912.902**	**43.000**	**57.000**	**394,513.640**	**10.333**	**89.667**	**354,554.852**

Figure 3: Testing results

The full document with our results can be found in the Github repo for our project. Our results showed that all three forms of load balancing provided significant performance improvements over a single server in every simulation. In particular, 20 request rapid bursts saw an improvement from a 50% fail rate to a 0% fail rate when the load balancer was in use. Additionally, 10 request bursts saw an average 70% improvement in response time and 20 request bursts saw a 40% improvement.

The sustained 100 requests also showed a significant 55% decrease in fails, although response times increased. However, this increase was due to the fact that the overloaded single server would immediately send errors for a large number of requests that it received.

All three balancing strategies were effectively able to handle the 10 and 20 request bursts without any fails. We found that round robin performed the best for 10 request bursts, with a 30% improvement in response time over chained failover and a 48% improvement over least connection. This reversed for the 20 request burst, where least connection fared the best while round robin was the slowest to respond.

Finally, chained failover had the lowest fail rate for the sustained 100 requests with a 35% fail rate. Least connection came second with a 57% fail rate while round robin performed the worst with a 62% fail rate. This is because the three servers were likely overloaded with the 100 requests even with load balancing, but chained failover allowed multiple tries for each request to be processed. As a result, chained failover took roughly double the time to completely respond to all requests, but had a much lower fail rate. Least connection performed slightly better than round robin because it makes decisions on which server to use based on which server has a lighter load, although it provides no failover so there were still many failed responses.

Looking at success rates, all three balancer strategies provided more than a 3x improvement as compared to a single server given 100 sustained requests. In the best case, failed chainover displayed a 6.22x improvement over a single server, although with the caveat of doubled response time.

V. Review of Team Member Work

Task	Completed By	Timeline
Build a basic handler which is able to forward requests and responses between a client and a server	Alex	03/20 - 04/02
Add the ability to configure the load balancer with many servers using Round Robin	Alex/Suman	04/03 - 04/07
Implement Least Connection	Suman	04/05 - 04/08
Implement Chained Failover	Alex	04/10 - 04/13
Develop script to randomize request serve times	Suman	04/12 - 04/15
Deploy on GCP	Alex/Suman	04/18 - 04/21
Build a Evaluation Tool	Alex	04/20 - 04/24
Run tests and gather results, prepare results for presentation	Alex	04/24 - 04/27
Write final report and develop slides	Alex/Suman	04/22 - 04/30

Table 1: Task responsibility and timeline

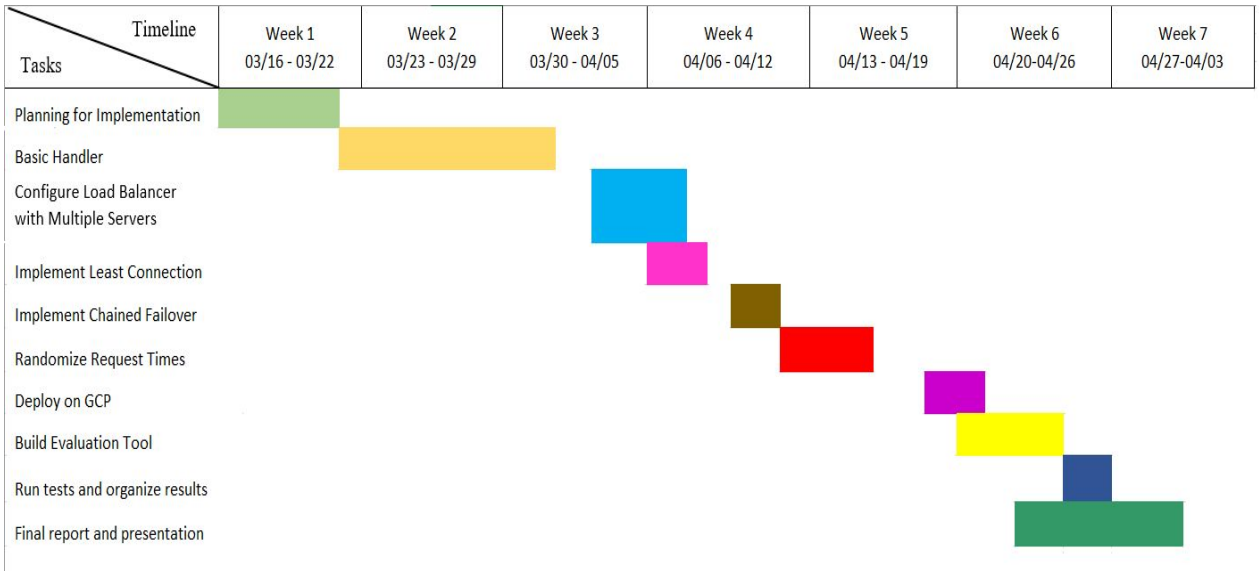


Figure 4: Gantt Chart of the project progress

Alex Costinescu

- Built the complete load balancing framework to handle client-server connections (requests/responses)

- Implemented the Round Robin and Chained Failover balancing strategies.
- Devised and implemented an evaluation tool to compare the performance of each strategy.
- Created startup scripts for running the load balancer and test server request handler as services.

Suman Hosmane
Having little experience in the Python network programming, I handled the simpler sub-systems of the project.

- During the initial run of the project development, I developed the script for spawning servers in Python using requested network ports rather than having a default self assigned port.
- Under load balancing strategies I implemented a Least connection algorithm over the network platform developed by Alex.
- As part of the final leg, I set up the GCP (Google Cloud Platform) virtual infrastructure with multiple servers and load balancer machines.

VI. Conclusion

We implemented a real-time, multi-threaded load balancer capable of using three different load balancing strategies: round robin, failed chainover, and least connection. This balancer was built using Python as well as its built-in libraries and is implemented using TCP sockets. Using our balancer as well as several Python servers running heavy calculations on independent Debian systems in Google's GCP environment, we tested our balancer. With this testing, we compared each balancing strategy with each other, as well as with a non-load balanced single server.

The goal of the project was to gain a better understanding of how a simple, transparent load balancer might be implemented and to better understand the related networking topics. We feel that we have achieved this goal, as both of us feel that we have a better understanding of the underlying technologies and concepts. We feel that we learned a lot about how and why tools like nginx are used to improve the capabilities of applications and websites that need to handle heavy traffic loads.

From our testing, we learned that the chained failover strategy provides the highest chance for a request to be successfully served, but sacrifices response time to do so. This was largely as expected, as this strategy makes multiple attempts to serve a request using different servers. Least connection performed significantly better, but failed to fulfill as many requests. Round robin performed the worst, largely because it does not make an informed decision about which server to select. Regardless of strategy, a load balanced system with multiple obfuscated servers performed better than a single server system. With three servers, we saw improvements of over 3x because the load balancer helped in avoiding completely overloading any given server.

Based on this testing, we feel that the best strategy for a load balancer would be a combination of failed chainover and least connection. This way, the server with the lightest load would be selected to serve a request and, if it failed, the request could be handed to the server with the next lightest load.