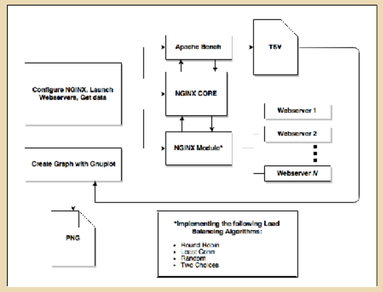
Bringing Innovative Load Balancing to NGINX

Adam Schwartz, Computer Science Capstone 2018. https://github.com/anschwa/capstone

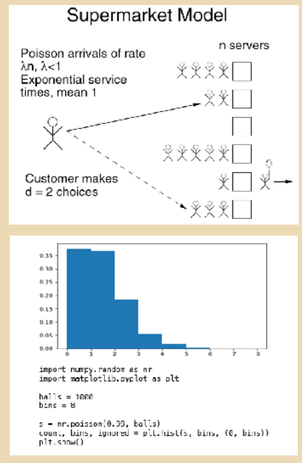
ABSTRACT

Load balancing remains an important area of study in computer science largely due to the increasing demand on data centers and webservers. However, it is rare to see improvements in load balancing algorithms implemented outside of expensive specialized hardware. This research project is an attempt to bring these innovative techniques to NGINX, the industry leading open source load balancer and webserver.

In addition to implementing a new, native NGINX module, I have developed a simple work ow to benchmark and compare the performance of available load balancing algorithms in any given production environment. My benchmarks indicate that it is possible to take advantage of more sophisticated load distribution techniques without paying a significant performance cost in additional overhead.



ABOVE: My Software architecture for running simulations with NGINX.



TOP RIGHT: The two-choices load balancing algorithm as described in Mitzenmacher's paper.

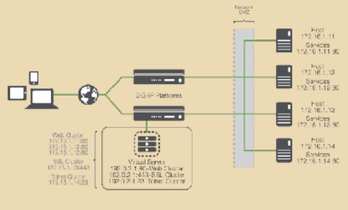
BOTTOM RIGHT: What a Poisson distribution looks like

METHODS AND PROCESS

I developed two load balancing modules for NGINX: random and two-choices. Both modules are compiled and dynamically linked into the system installation of NGINX because it makes development much easier. However, both modules can can be statically linked if desired. Although NGINX provides an API for writing modules in perl, I chose to implement them directly in C to eliminate any potential overhead that may skew the results. I also consider native NGINX module implementations more useful to the open source community.

In order to test the effectiveness of the load balancing algorithms, I created a simple webapp in Go that will simulate my production webserver environment. Go is an excellent language to use for this task because it has an extensive HTTP package in the standard library, compiles to native machine code, and does not need any additional dependencies to host a webserver.

```
import numpy as np
import numpy.random as nr

rate = nr.poisson(0.99, requests)
DIST = [(w * 0.1) + 1 for w in rate]

def uniform_random(requests, servers):
    rand = nr.randint(low=0, high=servers, size=requests)
    load = [0] * servers
    for i in range(requests):
        weight = DIST[i]
        chosen_server = rand[i]
        load[chosen_server] += weight
    return np.std(load)

def round_robin(requests, servers):
    load = [0] * servers
    for i in range(requests):
        weight = DIST[i]
        chosen_server = i % servers
        load[chosen_server] += weight
    return np.std(load)

def two_choices(requests, servers):
    load = [0] * servers
    for i in range(requests):
        choice_one = nr.randint(0, servers)
        choice_two = nr.randint(0, servers)

        if load[choice_one] < load[choice_two]:
            best_choice = choice_one
        else:
            best_choice = choice_two

        weight = DIST[i]
        load[best_choice] += weight
    return np.std(load)
```



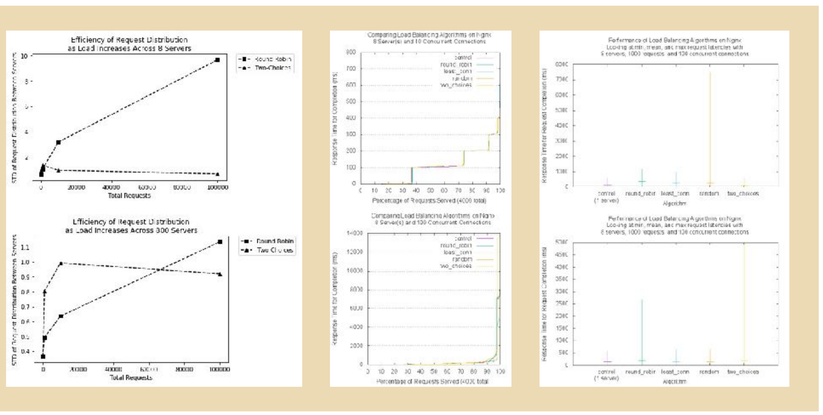
LEFT: Code Segments from the Load Balancer Implementations in Python

ABOVE: Example of high performance load balancing system architecture

RESULTS AND DISCUSSION

My extensive benchmarking revealed no obvious distinction between load balancing algorithms running in NGINX. Regardless of the active module, performance remained about the same. However, there were some general trends regarding concurrent and total requests that were anticipated. Namely, when you flood your webserver with requests, it takes longer to respond.

What these results do indicate, is that the overhead of a load balancer may become negligible when taking into account the total overhead associated with completing an HTTP request. In the earlier simulations with python, I was concerned that the increased latency of two-choices would make it an inconvenient load balancer in a production environment. However, my results show that we may be able to take advantage of two-choice's uniform load distribution abilities without paying much performance penalty.



When the number of concurrent connections are kept relatively low, each load balancing module behaves nearly identical. However, as we increase the concurrent connections, we see that the vast majority of requests are completed under 500 ms, but approximately 5% of requests take thousands of milliseconds longer to complete. This behavior is a known issue with using Apache Bench, but it also addresses the problem load balancing attempts to solve. That is, once a webserver becomes overloaded, it is very hard for it to recover.

REFERENCES: K Salchow. 2007. Load balancing 101: Nuts and bolts. White Paper, F5 Networks, Inc (2007). Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R Larus, and Albert Greenberg. 2011. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. Performance Evaluation 68, 11 (2011), 1056–1071. Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. IEEE Transactions on Parallel and Distributed Systems 12, 10 (2001), 1094–1104. Tyler McMullen. 2016. Load Balancing is Impossible. (November 2016). https: //www.youtube.com/watch?v=gas2v1emubU

ACKNOWLEDGEMENTS: I want to give a special thanks to David Barbella for being my Capstone Adviser and Charlie Peck for offering additional guidance during my project. I would also like to thank Maxim Dounin for answering my questions on the NGINX mailing list.