Samuel Ingram

ingram23

https://github.com/VoxelrSam/LoadBalancer

CS 422 Project Report

Project Definition

I created a server load balancer that properly distributes requests to various compute servers depending on the current workload and balancing algorithm. This load balancer acts as the single point of entry for many requests that are then distributed to different servers based on a multitude of different algorithms. These algorithms have then been analyzed under multiple scenarios to determine which is best for different applications and which is most practical in the real world. Primarily, these algorithms are analyzed by timing them in terms of total time to complete a batch of requests and the average time per request, however, other factors such as data structure upkeep and overhead are taken into account.

Project Motivation

Distributed computing and wide networks of servers are becoming quite commonplace in the modern world, so it is ever so pertinent that we understand how to handle such networks. The project motivation consists of two main components: my own self education on this topic, and attempting to discover important details of how load balancers should be implemented. If we can get a better grasp on the best implementation for such load balancers, we can more quickly fulfill requests to people on the internet.

Related Work

When doing research, I found a number of good resources describing load balancers and algorithms on websites like jscape, ngix, and serverfault, but none of these websites showed any benchmark numbers for how fast or slow different implementations were. Some would describe when to use one approach over the other, but I never really got an idea for how much better one approach was. I hope that my numbers and findings can help answer that.

My Work

My main two software components were the load balancer and the server. After creating these, I went in and wrote a testing script to run through various instances of my load balancer to gather data. Before we get into the exact data that I gathered, I will first detail below the software components so that we are on the same page as to how everything worked. Everything in my project was programmed in JavaScript using Nodejs since that was a workflow that I was familiar with in terms of server frameworks. Please refer to the README for information on running or viewing the code.

The Load Balancer

- It is initialized with a port, server list, algorithm name
    - The server list should be an array of servers that act as the processing servers that the balancer should route requests to
    - The algorithm name passed in specifies which load balancing algorithm to use
- It is built using an Express server
- Upon receiving a request, it applies the algorithm specified on initialization and then forwards the request that it received on to whatever server the algorithm has chosen

- Upon receiving a response back from the server, it forwards the data back to the client
- If an algorithm needs data on the current load of the servers, the balancer has a post route that servers can send to, and it also gathers data from server response headers to update its information on each server's estimated execution time in the queue

The Servers

- They are initialized with a port and a specified delay time in milliseconds
    - This delay time is meant to simulate execution of a compute task on the server at the route /users. When querying this route, the server should wait for its specified delay time before responding. This route is what was used for gathering my metrics.
- They are also built with a similar Express server pattern
- These servers should serve static files in the folder named "public" with no delay. This feature was not extensively tested since it was not the purpose of this project, but it serves as a proof of concept that a normal server still works through my balancer. That being said, a test index file should be viewable from a browser.
- These servers were designed such that only one compute task is to be run at a time per server. The server will allow an arbitrary number of connections at one time, but all the connections will be put into a queue and handled in order one after another. In other words, it is similar to that of a single threaded server even though the exact code details are a bit different.
    - This approach was chosen since it is easier to simulate server load this way instead of having to actually flood the server and run complex computations. In the real world, most servers won't follow this design, but it was still a valid approach for the purposes of what I am trying to test

Metrics Gathering

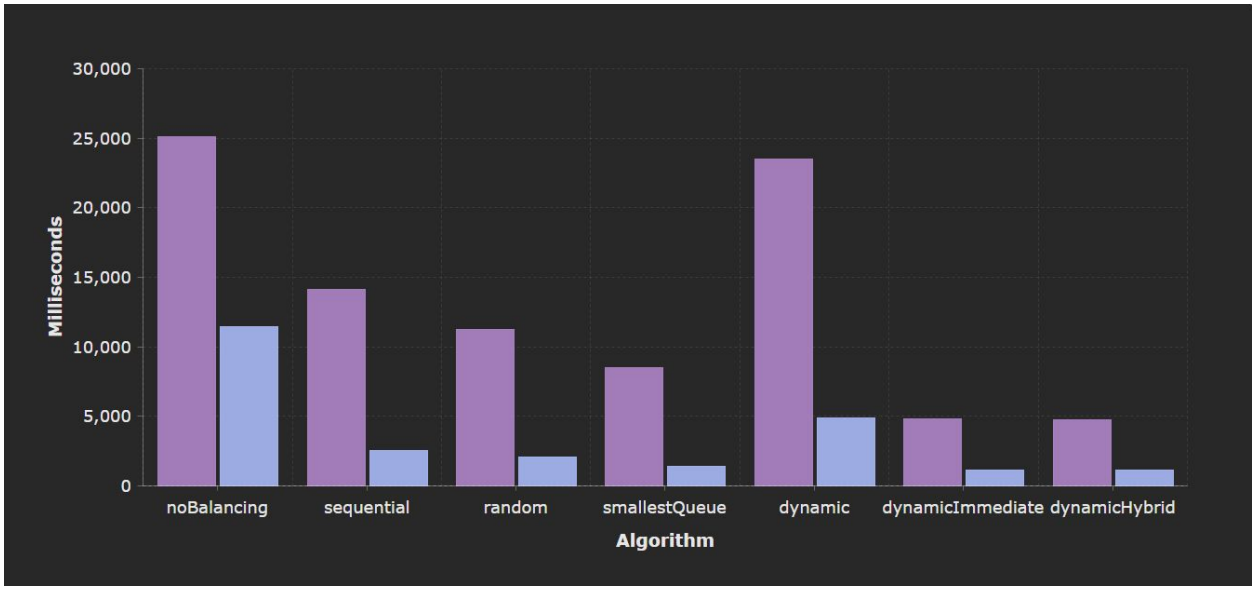
- A suite of tests were created to run through all of the different load balancer algorithms
- By default, 100 messages are sent per algorithm and the averages and total request times are then recorded to metrics.json
- The tests should wait 25 milliseconds between requests. This allows for a good balance between sending too many at once and not sending them fast enough to flood the servers a bit.

Balancer Algorithms

- Seven different algorithms were written and work as detailed below
    - noBalancing: The balancer always chooses the first server in the server list. This should be similar to only a single server with no balancer. This will act as our control so that we can better understand the benefit of having a load balancer.
    - sequential: The balancer sequentially steps through the server list as the next server is requested. This results in an even distribution of requests across all servers
    - random: The next server is chosen completely at random from the list
    - smallestQueue: The server keeps track of how many requests are still pending for each server. The next server chosen is the one with the least amount of pending requests.
    - dynamic: When a server is sending its completed request back to the balancer, it attaches the estimated amount of time that it will need to complete computation of the current process queue. When the balancer chooses a server, it chooses the one with the smallest estimated computation time.

- dynamicImmediate: Instead of attaching the process time to the response, the server sends its estimated time immediately upon receiving a request. The server does this via a post request to the balancer. This allows the balancer to update its computation estimates in a more timely manner.
- dynamicHybrid: This combines the two previous algorithms wherein the server immediately sends a computation estimate, and it attaches a new estimate to the response that it sends back. This allows even more frequent updating of computation times for the balancer.

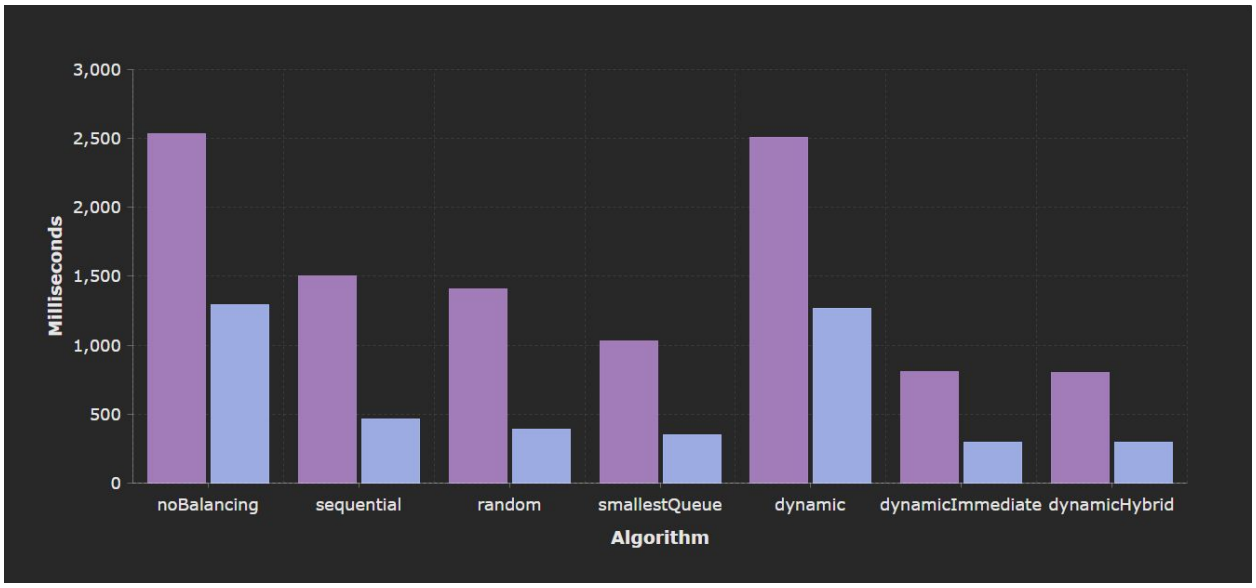
Now that we have a relative grasp on how the code all works together, let's take a look at the results that I found. I tested all the algorithms with 10, 100, and 500 messages each. For these tests, I used five servers with delays of 250, 500, 100, 200, and 700 milliseconds respectively. First, let's look at the 100 message test



category	total	average
noBalancing	25106	11439
sequential	14113	2527.8
random	11266	2095
smallestQueue	8511	1418
dynamic	23536	4908
dynamicImmediate	4792	1152
dynamicHybrid	4763	1142

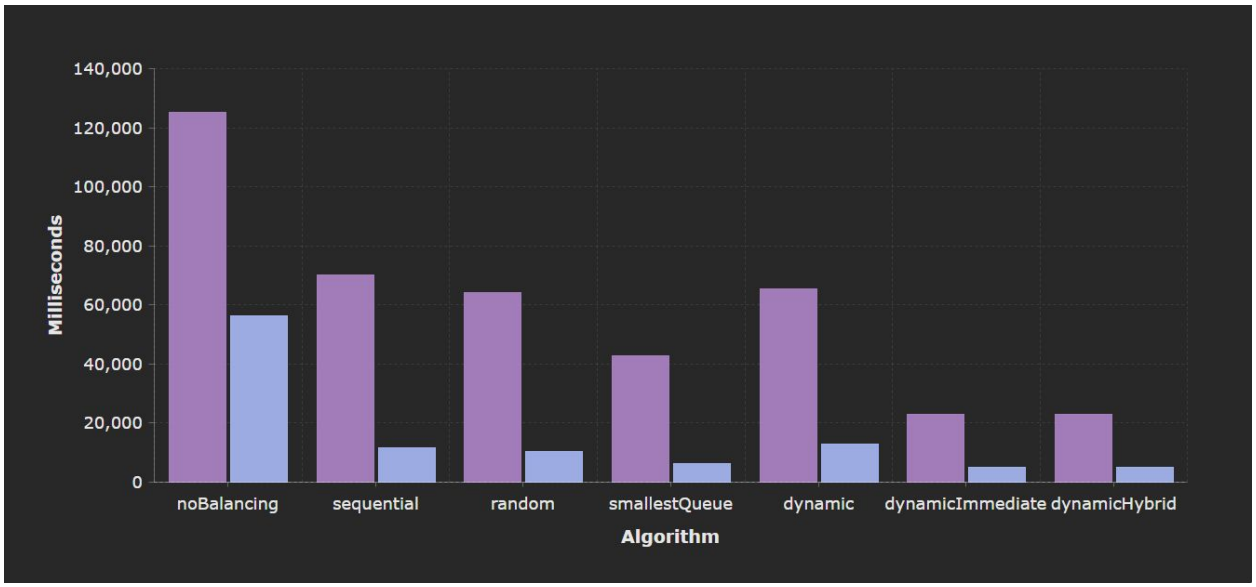
In the above graph, the “total” bar is used to measure the time from the first request sent to the last request response received, and the “average” bar is used to measure the average response time for each request sent. From here, we can already see some interesting points. Namely, we get progressively better from left to right until we start to make the move to dynamic. At first, this can seem really confusing since we would think that having the approximate pending execution time of each server would result in major improvements, but the key to understanding what happened is in the details of the algorithm. For the default dynamic algorithm, its time estimates are only sent when responding, which may be quite some time from the initial request. While the balancer has yet to receive a new estimated time from the server, it continues to pile on requests for the same server until finally the server completes its execution and sends a response. By the time the server sends the response, the estimated time for the query is huge since it has gained so many requests. This was exactly the case in my testing. As I watched the servers that my balancer picked, it kept piling on requests to the same servers until it received a new time estimate update. Unfortunately, this means that an idle server with a huge delay time may receive a ton of requests and then get backed up and slow down the entire testing time.

10 Requests:



category	total	average
noBalancing	2538	1292
sequential	1504	468
random	1407	393
smallestQueue	1030	350
dynamic	2510	1267
dynamicImmediate	807	298
dynamicHybrid	803	297

500 Requests:



category	total	average
noBalancing	125372	56413
sequential	70165	11657
random	64123	10534
smallestQueue	42854	6234
dynamic	65519	12850
dynamicImmediate	23113	4995
dynamicHybrid	23055	5000

One interesting point here is that the dynamic algorithm seems to improve with more requests when compared to the other algorithms. I believe that this happens since with so many requests, the balancer has time to collect the delayed computation estimates. In essence, we will receive the same penalties up front for bad assignments to idle servers, but our assignments will become a lot more improved in the later requests after having received initial responses from each server.

Before concluding our analysis, we should first make mention of algorithm upkeep. The first three algorithms (noBalancing, sequential, and random) do not require much of any external data structures or algorithms to get the next server. The last four (smallestQueue, dynamic, dynamicImmediate, and dynamicHybrid) all require finding the minimum of some array of values. For a small set of servers (5 in the case of our testing), the overhead for computing these hardly has any effect on algorithm execution. If, say, we were running a huge network with many servers, it may take a nontrivial amount of time to sort these values. Using something like a min heap with costs of log(n) will probably be our best solution here, but then we need extra space for keeping our data structure. Furthermore, dynamicImmediate and dynamicHybrid require sending post requests to the balancer from the servers over the network. If we have a bad network or too many servers, these could bog down the network and clog it up resulting in packet delays or losses. That being said, despite some of these algorithms appearing better than others, there may be good reason to use some of the simpler ones in practice. As an extension to this project, it would be interesting to test extreme cases with many servers and

bad networks to see if the overhead required would significantly affect the numbers I previously obtained.

Conclusion

Over the course of this project, I have learned a lot about load balancers, especially with regards to the various algorithms needed to support such a system. Through my data, it would appear that the best algorithms are built upon information. The dynamicHybrid algorithm is the clear winner in my test cases since it provided the balancer with the most data so that it could make the most well informed decisions. As discussed in my paragraph on overhead though, this may not always be the best algorithm in practice. If network usage is a concern, the smallestQueue approach could free up valuable bandwidth, but if sorting time is also a concern, it may make sense to take an even simpler approach with the sequential or random algorithms. In the end, any of these algorithms produced considerable benefits over using a single server. I like to think of it like a single threaded process versus a multithreaded one. The servers are similar to that of individual threads, and distributing your tasks over them can greatly increase performance in certain areas. My research and work on this project has helped me understand the different benefits and considerations with load balancers, and I hope that it can help other readers come to understand the same.