How to move a mountain (of data)

My Mountain

“Can you find out, exactly, how many domains SoftLayer hosts?” is the question that started my day a brief time ago. On the surface it seems like a fairly straight forward problem, much like being tasked with moving a mountain. Mountain is at point A, it needs to be at point B, seems pretty simple, at least until you realize how much you actually have to start moving, and how far away points A and B actually are.

The number of domains we host is an interesting statistic that our marketing department was curious about, the problem is that SoftLayer has way of tracking this metric in a meaningful way.
There are a lot of metrics SoftLayer does keep track of, but none of them can be used to say, exactly, how many domains are hosted in our datacenters.

The only solution that would really work, is get a list of all registered zones, go through each and every one, figure out what IP it resolves to, and see if that is owned by SoftLayer. Luckily I had just stumbled across an article (http://jordan-wright.com/blog/2015/09/30/how-to-download-a-list-of-all-registered-domain-names/) explaining how to get a list of registered zones from the registrars. Without that information, I would of had to brute force every possible domain name, which would of taken literally forever, as number of possible domain names in any give zone approaches 60,653,279,000,000,000,000,000,000,000,000,000,000,000,000,000,000 (63^26) possible combinations.

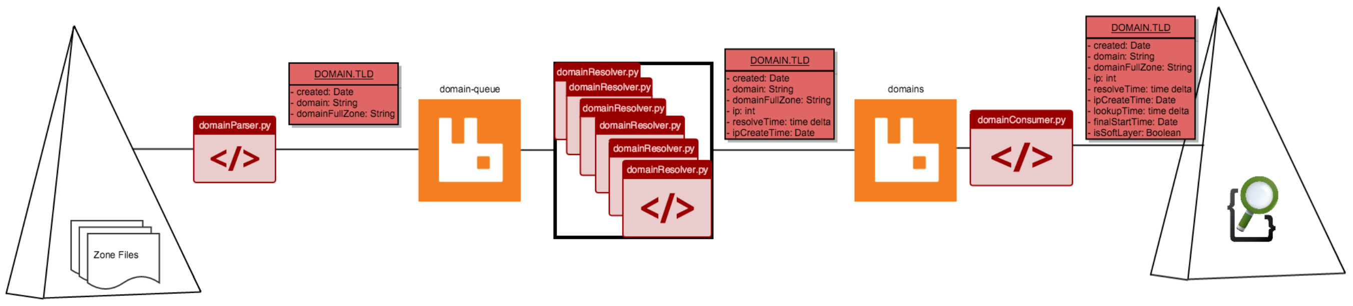
However, it turns out actually getting the records from the registrar is a bit of a hassle as each zone has a different registrar, and each registrar has a very different format for getting the proper access, if they give it out at all. The zones I was able to get access to account for about 70% of all registered domains (most of them in .COM), so I felt that was a big enough mountain to try to move for one day. There are services (http://viewdns.info/data/) that offer paid access to all zones, but that might be a project for another day.

Initial Planning

You can't just move a mountain by grabbing a shovel and going at it, and you can't sort through every domain on the Internet that way either. My plan boils down to the following:

1. **Get a server! (this is the easy part)**
2. **Get a list of SoftLayer IPs.**
This was pretty easy, we keep a database of all our owned subnets. So I just had to export that database, convert each subnet to its collection of ips, convert those into a binary number (for easy searching) and store those in my own local database.
3. **Setup a queue service**
I choose rabbitMQ (https://www.rabbitmq.com/) since I have heard of it before, and it was pretty easy to get setup. I just went with a single node implementation, so I might be missing out on some magic, but it all worked fine without much effort from me, which was nice.
4. **Setup a database service**
Just a simple MySQL database for the IP lookups, but I went with ElasticSearch (https://www.elastic.co/) to store the end domains because BIGDATA! Also it works really well with Kibana (https://www.elastic.co/products/kibana) to make pretty graphs and it was really easy to get up and running without actually knowing what I was doing. Storing documents is as simple as a POST request, super easy.
5. **Setup a recursive dns service**
I choose to run my own recursive service instead of using a public one or a SoftLayer one since I figured I wouldn't really be able to take advantage of much caching those public services would have to offer, and I wanted to go as fast as possible, so taking out any middle men is really important. I choose Unbound (https://www.unbound.net/) because it has a neat name.
6. **Script to parse zone files**
Use a regular expression to find out what the zone is (each zone needs its own regular expression because life would be too easy if they were all the same format), group them into stacks of 25 domains, and put them into a queue.
7. **Script to resolve zones**
Take 1 packet of 25 domains out of the queue, resolve each one, put them back into a packet and put them in a different queue. If the zone isn't resolvable, try www.zone.tld (http://www.zone.tld), if that doesn't work, give up.
8. **Script to check if the zone is hosted by SoftLayer**
Take last packet of 25 domains, check if they are SoftLayer or not, save the result.
9. **Some pretty graphs**

Kibana (https://www.elastic.co/products/kibana) worked really well for this. Basically magic. All that together would look something like this:



v1

Flash forward an indeterminable amount of time, I've setup the server, got all the services configured and running, gotten all the data pilled up in a big mountain and written the needed scripts. Its time to get this show on the road!

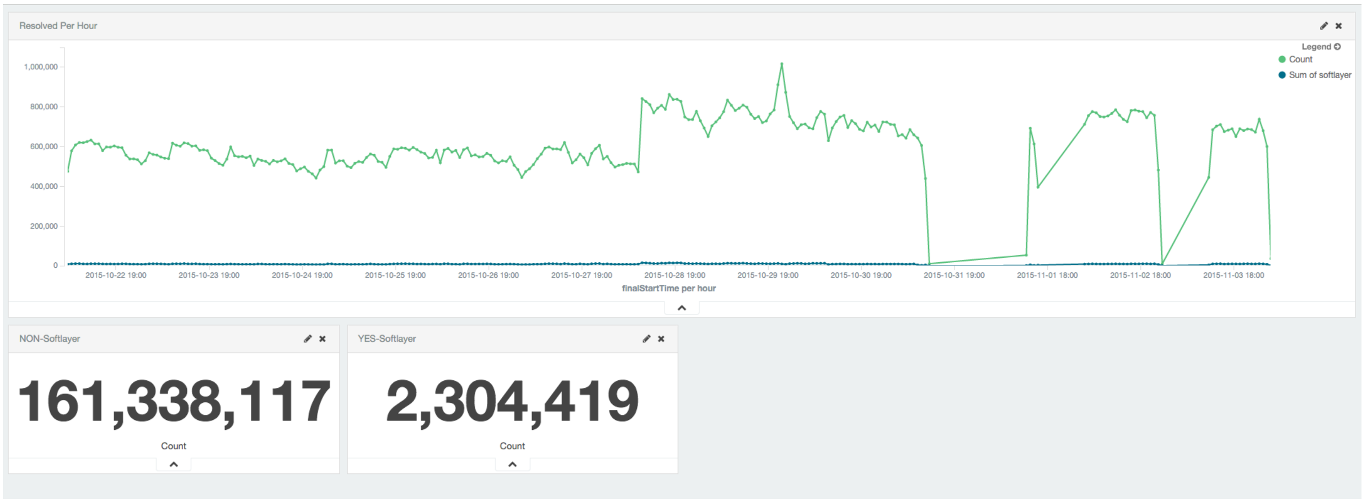
The first problem I ran into was that I was going WAY too fast parsing the domains. I ran the server out of its 8G of RAM in about 60 seconds, whoops. There is about 15G of domain data, so for me to keep it all in RAM I would need at LEAST 15G of RAM for just RabbitMQ, not to mention all the other services on my server. Instead, I choose to rate limit the parser script to add data at a rate that was somewhat comparable to how fast I was removing them from the queue.

Problem number two has that some unresolvable domains take a LOT of time to fail, and there were a lot more unresolvable domains than I thought there would be. Since I was resolving the domains sequentially, this meant a packet of 25 unresolvable domains took up to a minute to complete. Since I had about 250,000,000 unresolvable domains, I needed to find a better solution. So to fix this I googled "python multi processing" and copy pasted whatever was on the first link (http://www.ibm.com/developerworks/aix/library/au-multiprocessing/). I also dove into the Unbound configuration and did my best to limit how long it would wait on unresolvable domains, getting the penalty for a miss down to 2s. So now I can resolve 25 domains at once, taking the time from a max of 1 minute, to a max on 4s for each packet of domains.

The next roadblock was that resolving zones turns out to be VERY CPU intensive, which greatly limited the number of domain resolving scripts I could run. Overall, Unbound took ~50% of my 8 cores, with the domain resolver itself taking most of the rest. This turned into a hugely limiting factor that caught me a bit by surprise as I had assumed resolving domains was a pretty quick process. At the end I had about 40 domain resolver scripts running, burning through 200-400 domains a second.

My domain checker script actually worked a LOT better than I expected. Checking and saving the results of a domain was almost instant, even when doing the domains in sequence, so I didn't even have to bother with multi processing here, and only had to have 1 of these scripts running.

At the end, it took a little over 300 hours of computer time to completely move my little mountain of data. It turns out that out of 160 million domains, SoftLayer hosts about 2.3 million, or a little under 1.5% of the Internet.



v1 COST

So, how much did it cost to move my little mountain?
I used the following hourly server.

```
Intel Xeon E3-1270  4 Cores 3.40GHz Up to 2 drives  8 GB @ 0.368$/hour
```

I started at October 22, 6am and the last domain got parsed at November 4, 3am, which gives me about 309 hours, as I'm subtracting about 30 hours where the project wasn't running due to me breaking things. For accounting purposes, I'm assuming those little accidents didn't happen.

price: 309hours * .368$/hour = 102.672$
Which I think is pretty good, since a comparable (https://www.softlayer.com/Store/configureOrder/257/49515) server ordered on a monthly

basis would run at least 235$.

v2

Now that we have v1 has a benchmark, lets see how fast we can go! The key to scaling up any application is making each piece as independent as possible and design them to talk over the network. If I had started v1 with just a giant script that did everything in memory, scaling up would involve a complete re-write, but luckily it only needs some minor improvements, so here we go!

Improvements

1. **Configuration Management**
   I went with SaltStack (http://saltstack.com/) because I'm fairly familiar with it already, and because it has a salt-cloud feature which lets me define and provision my entire infrastructure with one command. I plan on having a LOT of virtual machines working away so being able to easily spin them all up at the same time is really important to me. Mostly because this project has quite a few packages that need to be installed, and I don't want to have to bother with doing all that work.
2. **Threads vs Processes**
   multiprocessing is nice, but multi-threading is better for what I'm doing, and all my scripts will need to use threading to get as much throughput as possible. So I went and slightly changed around the domain parser, domain resolver, and domain checker to all use threads. the domain checker and resolver ALSO use processes too, but that is mostly so I can run the script as a service and have it spin up the needed number of processes so I can better control how much CPU power is used.
3. **Config Files**
   Hard coding in IP addresses is fine when everything is one the same server, but if I'm going to be distributing everything, I need to be able to easily change where the data goes on the fly.

Servers

Domain Master - Hourly Bare Metal - 4 Cores 3.50GHz 32 GB

This guy is responsible for controlling all the other via Salt. It will also host the ElasticSearch service and will run the domain parser script, since all the zone files will reside here. In hindsight, I used way too much ram for this server, I could have gotten away with 8G I think.

Rabbit Node - Virtual Server - 4 cores 48GB RAM - 1Gbps Network

Just runs the rabbitMQ service and that is it. In the future I need to figure out what all this clustering rabbitMQ can do is about, but for now I am just using the one node. With clustering and possibly splitting up the 2 queues I think I could have saved a decent amount of money by needing less RAM.

Resolver Node - Virtual Server - 2 cores 1G RAM

Runs Unbound and the resolver script. These rely mostly on CPU power and very little ram. 2 cores because most of the time these scripts are just waiting for results from the network, so I have plenty of free time to spend switching between processes. Each node runs about 40 processes of the resolver script (each with 25 threads, each thread resolving 1 domain)

Checking Node - Virtual Server - 2 cores 1G RAM

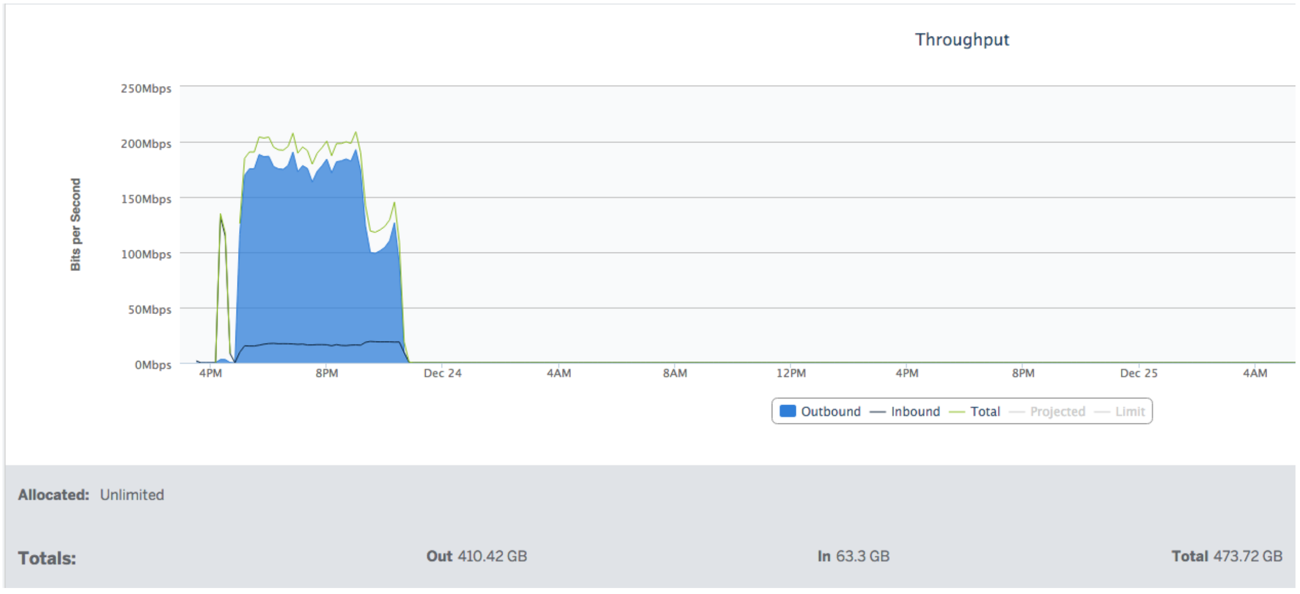
Just runs the checking script, not really much RAM is needed, just CPU power. Each node is running about 80 processes of the consumer, which gets the load on the system to about 80%

New Problems

When you go as fast as you can, new things always break where you thought everything was working fine before.

1. My domain parser actually was able to push out a lot more traffic than I thought it was, easily saturating the rabbit servers 100Mbps connection, so I had to upgrade it to 1Gbps which is a little more expensive. The domain parser ended up doing about 200Mbps, and got through all the zones in about 30 minutes which was a really good sign. It actually took me longer than I care to admit to notice this problem, it wasn't until

my ssh session started being really slow that I thought I might have a network saturation problem.



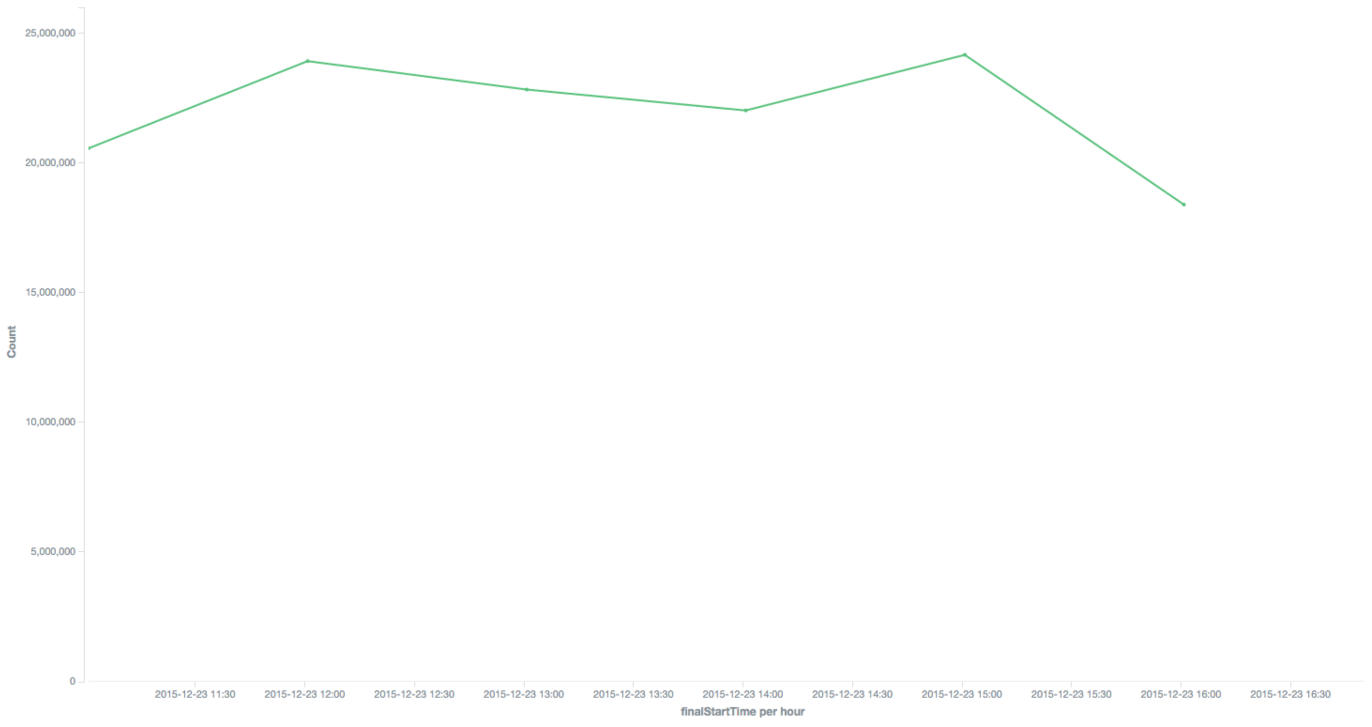
Allocated: Unlimited

Totals: Out 410.42 GB In 63.3 GB Total 473.72 GB

2. Versions matter! I tried upgrading Elastic Search to 2.0 because it sounded (https://www.elastic.co/blog/elasticsearch-2-0-0-released) like a good idea, but this tanked my performance of the checker script. Going from .001s to store a domain to .04s. I suspect some of the improvements that went into better indexing made saving results take a bit more time, but I didn't want to dig into documentation to figure out what I needed to change in the config file, so I just reverted back to 1.7 and everything went blindingly fast again. Besides, I don't need fast indexing, just fast saving. I can wait a few extra seconds for my graphs to be drawn.

3. There are a staggering amount of changes I do to these system. You don't really appreciate how much work it takes to get a system from the OS default to a working state for your application until you have to write down every change into configuration management. It was worth the effort though, being able to spin up about 40 servers with one command, have them configure themselves and get working automatically is the best.

With all those changes, I was able to get go through a staggering 6000-7000 domains/second, which is about a 30x speedup!



v2 COST

So, after I got everything spun up and running, it only took a smidgen over 6 hours to complete the project. However that is only half the story, lets add up how much the whole thing cost to see if using more servers might actually save us money.

```
Master Node x 1 @ .595$/hour
    - Intel Xeon E3-1270 v3  4 Cores 3.50GHz 32 GB
Rabbit Node x 1 @ .606$/hour
    - Virtual Server, 4 cores 48GB RAM
Resolver Nodes x 25 @ .060$/hour
    - Virtual Server, 2 cores 1G RAM
Consumer Nodes x 10 @ .060$/hour
    - Virtual Server, 2 cores 1G RAM
```

I pressed to GO! button at December 23, 11am and the queue was empty at December 23, 6:30pm, which we will call about 8 hours of billable time, and I will add 1 hour to the master for setup and cloud provisioning time.

Adding an hour to the master for initial setup times.
(.595 x 1 x 9) + (.606 x 1 x 8) + (.060 x 25 x 8) + (.060 x 10 x 8) =
5.355$ + 4.848$ + 12$ + 4.8$ = **26.998$**

So, not only is using more servers significantly faster, it is also about a quarter of the cost as well!

Conclusion

```
The man who moves a mountain begins by carrying away small stones.
--Confucius
```

To tackle any big project, you must break it into small, workable pieces. This is really what the whole “Cloud Computing” movement is really about. Taking huge, megalithic projects, breaking them down into small pieces for small servers to work on individually.

All the code and Salt configurations are available on GitHub (https://github.com/allmightyspiff/domainThing) for what they are worth. It is mostly a collection of code that almost barely works, but it might server as a decent example if you ever want to figure out how many domains you host, the hard way :)