Keith Cirkel
JavaScript Consultant

Twitter GitHub

20 Mar 2014 in Node.js, JavaScript, ServerSide, DevOps

Load balancing Node.js

This month I started the second professional Node.js contract of my career - building a multinational news website. I love Node.js and I consider myself lucky to be working on huge websites that use it as a technology. Node.js, however, is still considered a “new” technology, and is an emerging technology to use in “enterprise” situations. The biggest problem large providers face is how to properly scale out their servers to multiple, load-balanced Node.js processes.

Clusters

Luckily Node.js has got your back. It comes with an awesome module - Cluster - which lets you spin up a bunch of new processes and connect the sockets to your master process, so it acts like one big multithreaded Node.js server. You can even send messages back and forth between a master and its workers. It's also super easy to set up your app to handle Clusters, check it out:

There's also a really great management tool for multi-process Node.js setups called PM2 which uses Clusters under the hood, and offers loads of other features, such as monitoring, 0s downtime restarts and loads more!

The test setup

Of course, putting Clusters into production without any kind of testing is a bad idea, so I set up this simple test to see how well it performs on a VM (4 CPUs, 4gb RAM). This test isn't to test how many requests per second it can possibly acheive, but more to see how the CPU is utilised between the processes - to make sure that the setup is actually balancing the load.

I have a simple VagrantFile combined with some Chef scripts to load up a VM in VirtualBox with the required setup to run the app. This will fully automate the server setup so that we can test in a controlled environment which can eventually be ported to our production servers. The Chef scripts provision the server by installing Node.js, then copying the source files to `/home/mysite` followed by copying an upstart script to `/etc/init` (which should work in Ubuntu, RHEL and CentOS). The Upstrart script simply starts the Node.js and monitors it, if it quits unexpectedly then Upstart will automatically restart it. All of the code is available on this article's Github Repo - load-testing-node.

After a simple `vagrant up` (ok, a bit of a lie - because I don't know of a sensible, simple, way to copy the `src` directory using Chef, I have to first run `cp -rf src cookbooks/mysite/files/default/src`), the VM starts up and Chef does its magic, and lo', the app running on port 3000

which is also forwarded to localhost:3000. Now we have a working server with a fixed CPU power, we can load test our server configuration to ensure that everything works smoothly.

To test the load-balancing features on this set up I used wrk - which is a nice and simple load testing tool. Once again, we're not interested in ramping up to see how much this setup can handle, especially as they're not representative, just to see how well they balance the load. The test command used was `wrk -d30s -t12 -c5000 http://localhost:3000` which means "use 12 threads to send of 5000 concurrent requests a second for 30 seconds". With this simple test running, a quick check of `top` gives us a good indication of how well the load balancing is working (I'm expecting to see all processes using an even amount of CPU).

After checking `top` to see if Node.js was evenly distributing requests, the result was pretty strange; one core was seeing high CPU utilisation, but the others were not really working as hard as they could. As you can see below from the `top` output, this was not an even distribution of requests:

```
12997 mysite    20   0  657m  39m 5556 R  109  1.0
0:22.44 nodejs
12998 mysite    20   0  656m  38m 5556 S   55  1.0
0:09.85 nodejs
12995 mysite    20   0  656m  37m 5564 R   53  1.0
0:09.84 nodejs
12994 mysite    20   0  656m  38m 5556 R   51  1.0
0:09.40 nodejs
```

The numbers 109, 55, 53 and 51 represent the CPU usage here. These numbers indicate that Clusters in Node.js really fall down at the first hurdle: they don't properly load balance, and only favour one process. The StrongLoop blog summarises the problem with Clusters in Node 0.10 nicely - the gist being that Node.js 0.10 decided to defer the routing of these processes using the (Linux) Kernels Scheduler, but the kernel has a different idea of how scheduling should work, and doesn't properly "load balance" the processes, instead it tries to run the request on the process that last blocked the socket. This will supposedly be fixed in 0.12, but until

then the Node.js Cluster module is not an appropriate solution for this.

Alternatives

After establishing that Node.js clusters don't actually work that well, it's time to look at alternatives. I'm going to look at 3 alternatives, each with distinct advantages and disadvantages: Nginx, HAProxy and Varnish.

Nginx

Nginx is an open source HTTP server, similar to Apache but much faster. It has built in load balancing, which makes it an ideal candidate for this project.

Nginx is actually a pretty darn solid choice for load balancing Node.js processes. It is pretty easy to configure in a load balanced configurtion, and has tonnes of features to sweeten the deal. You can weight particular processes so they get more requests than the rest, enforce sticky sessions (where users will always hit the same instance upon returning), add timeouts and lots more. It is also incredibly useful for horizontal scaling (adding many servers) as the upstream processes can be on different machines. You can also set it up with SSL Termination, meaning Nginx keeps the SSL certificates and talks to your Node.js processes over HTTP - simplifying their deployment process. Nginx can also even serve static files, meaning you don't need to bog down your Node.js processing serving up CSS files (but it does complicate the config, so it is perhaps best left to Node.js).

After setting up a basic load-balancing config (available in the "nginx" branch of this article's git repo) I did another simple load test, the results were much more positive:

```
 1018 mysite    20   0  659m  58m 5576 R   68  1.5
0:04.60 node
 1023 mysite    20   0  659m  58m 5576 R   68  1.5
0:05.00 node
 1024 mysite    20   0  659m  58m 5576 S   62  1.5
```

```
0:05.00 node
 1030 mysite    20   0  659m  58m 5576 S   56  1.5
0:04.28 node
 1700 www-data  20   0 63688 2736  900 R   33  0.1
0:01.88 nginx
 1701 www-data  20   0 63768 2776  900 R   30  0.1
0:02.11 nginx
 1699 www-data  20   0 63728 2848  912 R   29  0.1
0:02.64 nginx
 1702 www-data  20   0 63772 2692  900 R   29  0.1
0:01.42 nginx
```

The even distribution of CPU utilisation across each `node` process proves that Nginx is successfully load balancing equally among all cores. By default nginx span up 4 “worker processes” which I’ve also included in the top output. In total all of these processes took 375% CPU (or, on average, 93.75% per CPU core). It could certainly be tweaked to ensure it tries to use as much of the CPU as available, but that’s another topic.

There aren’t many downsides to nginx - the config can be a bit alien at times (but then so can the others), and the configuration options for load balancing options are somewhat limited. In fact, it is pretty much the defacto choice in load balancing. But, first and foremost, nginx is a server not a load balancer. While it works great, it has its limitations and it’s worth testing other tools to see if they perform better or offer more features.

HAProxy

HAProxy or “High Availability Proxy” is an open source Proxy server. It is pretty much built solely to load balance lots of machines. Unlike Nginx it cannot serve up static files. It comes with all the great load balancing features of Nginx - weighted processes, failover, horizontal scaling, plus loads more settings and features! It offers very detailed configuration for how processes are load balanced, as well as throttling the req/s to prevent the underlying processes being overloaded. It also offers some great monitoring options.

Because HAProxy is literally only a load balancer in its basic config, it

happily just distributes the load across all given Node.js process, so, based on the setup (available in the "haproxy" branch of this article's git repo) the `top` output is as follows:

```
17318 root      20   0 23712 4064  428 R  100  0.1
0:18.22
haproxy
15050 mysite    20   0  658m  55m 5572 S   38  1.4
1:31.88 node
15063 mysite    20   0  659m  56m 5568 S   38  1.4
1:33.23
node
15058 mysite    20   0  657m  55m 5568 R   37  1.4
1:30.06 node
15071 mysite    20   0  658m  55m 5568 S   37  1.4
1:30.41 node
```

A bit strange, `haproxy` is using up 100% - or one whole cpu core - but the Node.js processes are under medium load. Total load of these processes is 250% (averaging about 62.5% of each of the 4 CPU cores). The caveat with HAProxy is that in it's default configuration it is single threaded, meaning at 5000req/s the single `haproxy` process was maxing out and ended up the bottleneck while the Node.js processes where barely stressed. A simple change in the config was to add `nbproc 2` which spins up two `haproxy` processes, which solved the issue (you can add `nbproc N` where N is as many as you want, but for this test 2 seemed to be the right amount):

```
17282 root      20   0 19416 2764  428 R   86  0.1
0:11.15 haproxy
15050 mysite    20   0  658m  56m 5572 R   67  1.4
1:17.95 node
15058 mysite    20   0  660m  57m 5568 R   64  1.5
1:16.30 node
15071 mysite    20   0  659m  56m 5568 R   63  1.4
1:17.40 node
15063 mysite    20   0  658m  56m 5568 R   62  1.4
1:19.60 node
17281 root      20   0 19972 2764  404 R   52  0.1
```

```
0:10.53 haproxy
```

Much healthier! Nice, evenly distributed load between all processes. Between 2 `haproxy` processes and 4 `node` processes the system was utilising 394% CPU (or 98.5% CPU on each core), while serving 5000req/s. Putting this into production will be a bit of alchemy: some time will need to be spent getting just the right amount of `haproxy` vs `node` processes vs CPU cores.

Varnish

Varnish was the quirkiest choice; Varnish is actually a caching proxy meaning it handles all of the requests when it can, and if it can't it'll talk to the Node.js servers and forward the request, caching the response for subsequent requests. It's childsplay to set up a basic configuration, which you can see on the ("varnish" branch of this article's git repo). The `top` output was a bit more interesting than the others:

```
15175 varnish   20   0 3150m  88m  81m S  186  2.2
2:16.46 varnishd
   11 root      20   0     0    0    0 R   46  0.0
0:20.50 kworker/0:1
    3 root      20   0     0    0    0 R   44  0.0
0:27.22 ksoftirqd/0
```

Because of Varnishes caching features, when this simple Node.js app gets load tested we can see in the ouput that `varnishd` actually does all the heavy lifting. The Node.js barely even registers on `top`. It goes without saying that varnish is going to be faster, as it serves cached up content rather than hitting the real servers. I was impressed with the results though - it handled the 5000req/s benchmark but only using 276% CPU (the kworker and ksoftirqd threads ramped up in CPU as a part of varnishd, I believe). In fact, it ended up doing about 7600req/s with similar CPU usage numbers. In reality though, these numbers will be wildly different as some pages will be dynamic (no cache) and `varnishd` will end up passing the request right through to node.

Conclusion

There's one definite take-away from this - don't use the Node Cluster module in production (even the docs say Cluster is "`Stability: 1 - Experimental`"). Unless, of course, if you're reading this in the future and using Node.js 0.12 or 1.0, where the issues with the Cluster module are supposed to be fixed, in which case it may be ready for use.

As for the other three tools, I don't think there is a definitive answer. Each one is a different class of load balancer.

Varnish seems like a great tool if your website relies heavily on caching - it certainly takes the load off of Node.js, but it would not be a viable candidate for heavily dynamic websites. If you don't already have caching at the front of your webserver then Varnish is a sensible choice. Varnish keeps the cache in memory, so if you're serving lots of pages you may need more RAM.

HAProxy on the other hand is an uber configurable load balancer, and feels like it could be optimized to be faster than Nginx in certain configurations. If you dont have the need for caching, and are using a CDN for static assets (you *are* using a CDN for static assets, right?) then HAProxy is a great candidate, especially if you're a "knob twiddler", HAProxy has plenty of them!

Nginx does a little bit of everything, and does it all really well. It has some good configuration options for load balancing - but not as good as HAProxy, it has good caching options - but not as good as Varnishes. Nginx also can serve static content out of the box, which is great if you can't afford to put them on a CDN - although you could just leave it to Node.js to do this.

Of course, for the most demanding of websites, you could use a combination of 2 or all 3; Varnish could happily sit in front of HAProxy infront of Nginx infront of Node.js - although that might be overkill.

I'd love to hear what other great options are out there to load balance

Node.js processes, as always you can conversate with me over on the Twitterverse - I'm @keithamus.

Share this post

Twitter Facebook Google+