Unicorn!

October 9, 2009 defunkt Engineering

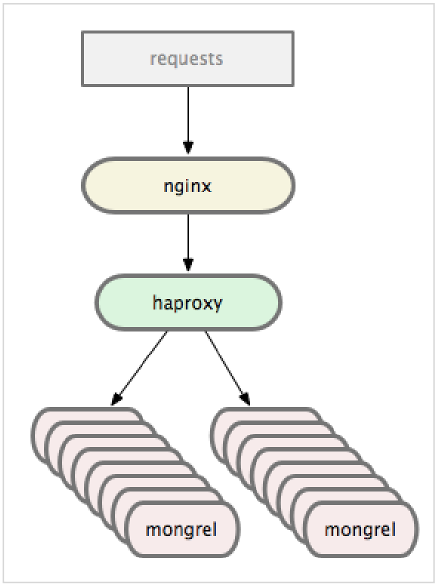
We've been running Unicorn for more than a month. Time to talk about it.

What is it?

Unicorn is an HTTP server for Ruby, similar to Mongrel or Thin. It uses Mongrel's Ragel HTTP parser but has a dramatically different architecture and philosophy.

In the classic setup you have nginx sending requests to a pool of mongrels using a smart balancer or a simple round robin.

Eventually you want better visibility and reliability out of your load balancing situation, so you throw haproxy into the mix:



Which works great. We ran this setup for a long time and were very happy with it. However, there are a few problems.

Slow Actions

When actions take longer than 60s to complete, Mongrel will try to kill the thread. This has proven unreliable due to Ruby's threading. Mongrels will often get into a "stuck" stage and need to be killed by some external process (e.g. god or monit).

Yes, this is a problem with our application. No action should ever take 60s. But we have a complicated application with many moving parts and things go wrong. Our production environment needs to handle errors and failures gracefully.

Memory Growth

We restart mongrels that hit a certain memory threshhold. This is often a problem with parts of our application. Engine Yard has a great post on memory bloat and how to deal with it.

Like slow actions, however, it happens. You need to be prepared for things to not always be perfect, and so does your production environment. We don't kill app servers often due to memory bloat, but it happens.

Slow Deploys

When your server's CPU is pegged, restarting 9 mongrels hurts. Each one has to load all of Rails, all your gems, all your libraries, and your app into memory before it can start serving requests. They're all doing the exact same thing but fighting each other for resources.

During that time, you've killed your old mongrels so any users hitting your site have to wait for the mongrels to be fully started. If you're really overloaded, this can result in 10s+ waits. Ouch.

There are some complicated solutions that automate "rolling restarts" with multiple haproxy setups and restarting mongrels in different pools. But, as I said, they're complicated and not foolproof.

Slow Restarts

As with the deploys, any time a mongrel is killed due to memory growth or timeout problems it will take multiple seconds until it's ready to serve requests again. During peak load this can have a noticeable impact on the site's responsiveness.

Push Balancing

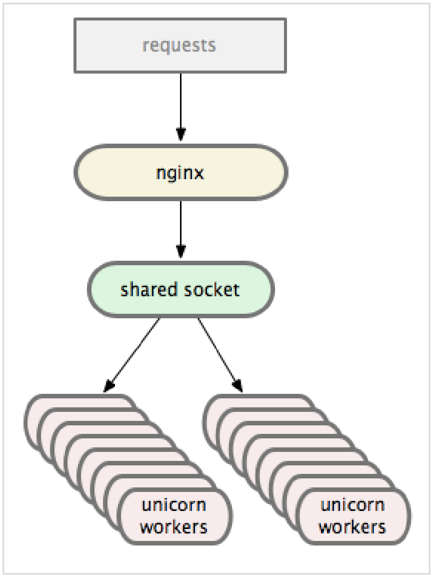
With most popular load balancing solutions, requests are handed to a load balancer who decides which mongrel will service it. The better the load balancer, the smarter it is about knowing who is ready.

This is typically why you'd graduate from an nginx-based load balancing solution to haproxy: haproxy is better at queueing up requests and handing them to mongrels who can actually serve them.

At the end of the day, though, the load balancer is still pushing requests to the mongrels. You run the risk of pushing a request to a mongrel who may not be the best candidate for serving a request at that time.

Unicorn

Unicorn has a slightly different architecture. Instead of the nginx => haproxy => mongrel cluster setup you end up with something like:



nginx sends requests directly to the Unicorn worker pool over a Unix Domain Socket (or TCP, if you prefer). The Unicorn master manages the workers while the OS handles balancing, which we'll talk about in a second. The master itself never sees any requests.

Here's the only difference between our nginx => haproxy and nginx => unicorn configs:

```
# port 3000 is haproxy
upstream github {
    server 127.0.0.1:3000;
}

# unicorn master opens a unix domain socket
upstream github {
    server unix:/data/github/current/tmp/sockets/unicorn.sock;
}
```

When the Unicorn master starts, it loads our app into memory. As soon as it's ready to serve requests it forks 16 workers. Those workers then select() on the socket, only serving requests they're capable of handling. In this way the kernel handles the load balancing for us.

Slow Actions

The Unicorn master process knows exactly how long each worker has been processing a request. If a worker takes longer than 30s (we lowered it from mongrel's default of 60s) to respond, the master immediately kills the worker and forks a new one. The new worker is instantly able to serve a new request – no multi-second startup penalty.

When this happens the client is sent a 502 error page. You may have seen ours and wondered what it meant. Usually it means your request was killed before it completed.

Memory Growth

When a worker is using too much memory, god or monit can send it a QUIT signal. This tells the worker to die after finishing the current request. As soon as the worker dies, the master forks a new one which is instantly able to serve requests. In this way we don't have to kill your connection mid-request or take a startup penalty.

Slow Deploys

Our deploys are ridiculous now. Combined with our custom Capistrano recipes, they're very fast. Here's what we do.

First we send the existing Unicorn master a USR2 signal. This tells it to begin starting a new master process, reloading all our app code. When the new master is fully loaded it forks all the workers it needs. The first worker forked notices there is still an old master and sends it a QUIT signal.

When the old master receives the QUIT, it starts gracefully shutting down its workers. Once all the workers have finished serving requests, it dies. We now have a fresh version of our app, fully loaded and ready to receive requests, without any downtime: the old and new workers all share the Unix Domain Socket so nginx doesn't have to even care about the transition.

We can also use this process to upgrade Unicorn itself.

What about migrations? Simple: just throw up a "The site is temporarily down for maintenance" page, run the migration, restart Unicorn, then remove the downtime page. Same as it ever was.

Slow Restarts

As mentioned above, restarts are only slow when the master has to start. Workers can be killed and re-fork() incredibly fast.

When we are doing a full restart, only one process is ever loading all the app code: the master. There are no wasted cycles.

Push Balancing

Instead of being pushed requests, workers pull requests. Ryan Tomayko has a great article on the nitty gritties of this process titled I like Unicorn because it's Unix.

Basically, a worker asks for a request when it's ready to serve one. Simple.

Migration Strategy

So, you want to migrate from thin or mongrel cluster to Unicorn? If you're running an nginx => haproxy => cluster setup it's pretty easy. Instead of changing any settings, you can simply tell the Unicorn workers to listen on a TCP port when they are forked. These ports can match the ports of your current mongrels.

Check out the Configurator documentation for an example of this method. Specifically this part:

```
after_fork do |server, worker|
  # per-process listener ports for debugging/admin/migrations
  addr = "127.0.0.1:#{9293 + worker.nr}"
  server.listen(addr, :tries => -1, :delay => 5, :tcp_nopush => true)
end
```

This tells each worker to start listening on a port equal to their worker # + 9293 forever – they'll keep trying to bind until the port is available.

Using this trick you can start up a pool of Unicorn workers, then shut down your existing pool of mongrel or thin app servers when the Unicorns are ready. The workers will bind to the ports as soon as possible and start serving requests.

It's a good way to get familiar with Unicorn without touching your haproxy or nginx configs.

(For fun, try running "kill -9" on a worker then doing a "ps aux". You probably won't even notice it was gone.)

Once you're comfortable with Unicorn and have your deploy scripts ready, you can modify nginx's upstream to use Unix Domain Sockets then stop opening ports in the Unicorn workers. Also, no more haproxy.

GitHub's Setup

Here's our Unicorn config in all its glory.

I recommend making the SIGNALS documentation your new home page and reading all the other pages available at the Unicorn site. It's very well documented and Eric is focusing on improving it every day.

Speed

Honestly, I don't care. I want a production environment that can gracefully handle chaos more than I want something that's screaming fast. I want stability and reliability over raw speed.

Luckily, Unicorn seems to offer both.

Here are Tom's benchmarks on our Rackspace bare metal hardware. We ran GitHub on one machine and the benchmarks on a separate machine. The servers are 8 core 16GB boxes connected via gigabit ethernet.

What we're testing is a single Rails action rendering a simple string. This means each requeust goes through the entire Rails routing process and all that jazz.

Mongrel has haproxy in front of it. unicorn-tcp is using a port opened by the master, unicorn-unix with a 1024 backlog is the master opening a unix domain socket with the default "listen" backlog, and the 2048 backlog is the same setup with an increased "listen" backlog.

These benchmarks examine as many requests as we were able to push through before getting any 502 or 500 errors. Each test uses 8 workers.

```
mongrel
 8: Reply rate [replies/s]:
          min 1270.4 avg 1301.7 max 1359.7 stddev 50.3 (3 samples)
unicorn-tcp
 8: Reply rate [replies/s]:
          min 1341.7 avg 1351.0 max 1360.7 stddev 7.8 (4 samples)
unicorn-unix (1024 backlog)
 8: Reply rate [replies/s]:
          min 1148.2 avg 1149.7 max 1152.1 stddev 1.8 (4 samples)
unicorn-unix (2048 backlog)
 8: Reply rate [replies/s]:
          min 1462.2 avg 1502.3 max 1538.7 stddev 39.6 (4 samples)
```

Conclusion

Passenger is awesome. Mongrel is awesome. Thin is awesome.

Use what works best for you. Decide what you need and evaluate the available options based on those needs. Don't pick a tool because GitHub uses it, pick a tool because it solves the problems you have.

We use Thin to serve the GitHub Services and I use Passenger for many of my side projects. Unicorn isn't for every app.

But it's working great for us.

Edit: Tweaked a diagram and clarified the Unicorn master's role based on feedback from Eric.

Have feedback on this post? Let @github know on Twitter.

Need help or found a bug? Contact us.