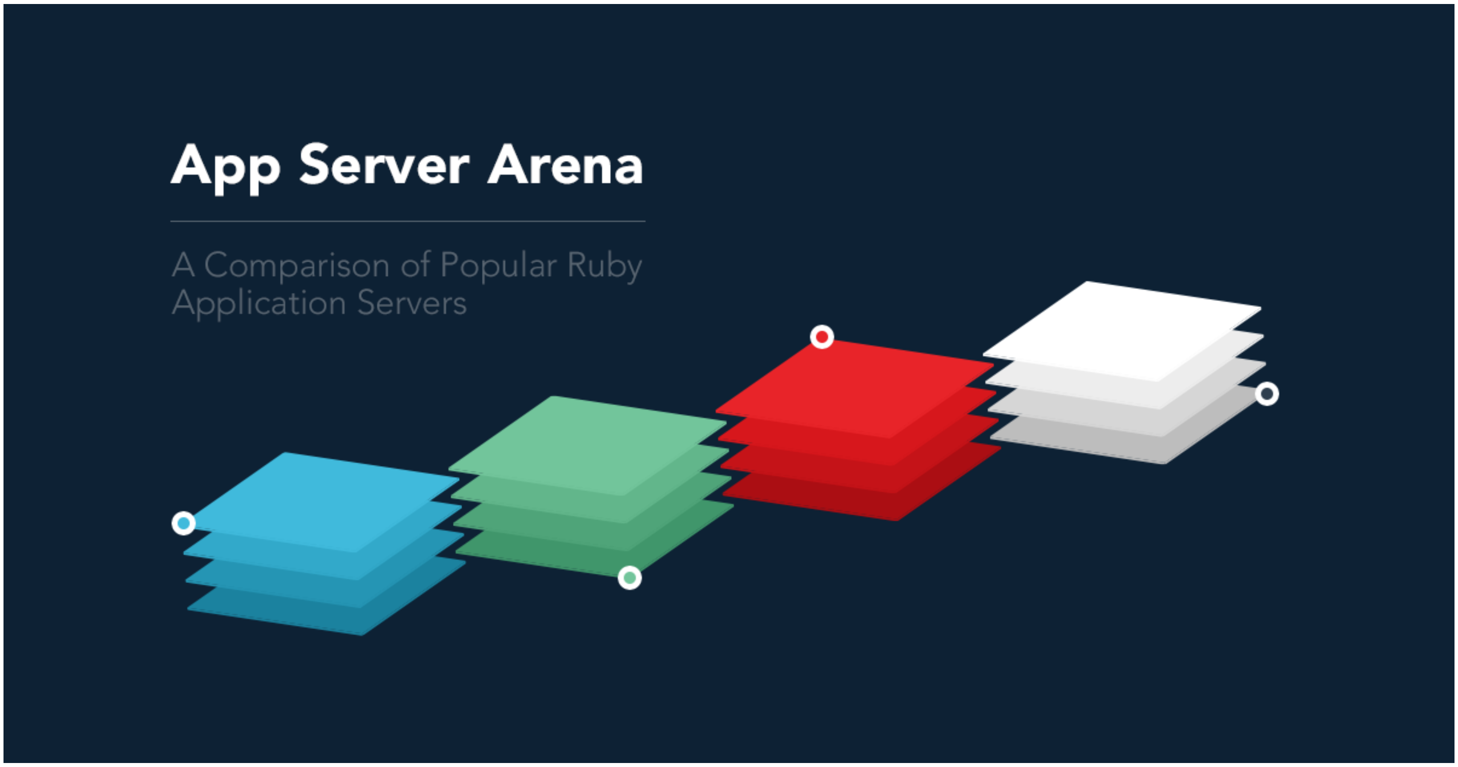
App Server Arena: Part 1, A Comparison of Popular Ruby Application Servers

June 18, 2014 | By J. Austin Hughey



In July 2013, I spoke at Lonestar Ruby Conference in Austin, TX about application servers in

Ruby. My talk included a performance comparison of multiple application servers in Ruby and how they do in various situations. This post details some of the findings from my research for that talk.

If you're interested in seeing the conference video, you can view it at Confreaks. Code for the simple test application I built is available at GitHub and slides are at Slideshare.

With the exception of performance assessment with Passenger 4, this research was performed in July 2013 and as such, may be a little bit dated. It's doubtful that massive differences have occurred in that time, but readers are encouraged to conduct their own tests should they find the results unusual.

An introduction to the gladiators

My talk and research looked at four Ruby application servers:

- Passenger, one of the easiest to configure and very common
- Thin, an EventMachine-based server
- Unicorn, a very straight-forward app server for fast clients and responses
- Puma, a truly concurrent application server

The application servers in question here were compared in several categories:

- Mode of Operation (Fighting Style)
- Use Cases (Strategy)
- Configuration (Training)
- Performance (Combat)

Passenger

Phusion's Passenger is an application server that is compiled either as an Apache module or compiled directly along with the nginx source code (because nginx doesn't contain a plugin architecture like Apache does). It excels in situations that need multiple low traffic applications running on the same machine.

Mode of Operation (Fighting Style)

Phusion Passenger operates by embedding itself into nginx or Apache. In this example, I only examined nginx simply because Engine Yard does not use Apache in our stack.

There's one interesting side effect of using nginx with Passenger, however: because nginx doesn't have a "plugin-like" architecture, similar to the way Apache does, nginx has to be recompiled from scratch using Phusion's nginx source code with their modifications. This is usually not a big deal, and to Phusion's credit they generally do a pretty good job with it, but it should be noted that, when running Passenger, instead of being able to get nginx straight from its maintainers, you'll have to get it through Phusion for their updates.

Once compiled and installed, Passenger is basically a part of nginx. You then configure Passenger by modifying nginx configuration - usually somewhere like `/etc/nginx/` or `/opt/nginx`.

Passenger has a huge array of configuration options to choose from to suit nearly any environment. However, it should be noted that under its default configuration, Passenger will use an elastic worker spawning method that waits until a worker is necessary before starting one. This has numerous benefits and unintended side effects, however, that we'll get into later.

When Passenger is configured, it's set as a "location" in nginx configuration. Further configuration instructs nginx to forward requests to that location, which is how Passenger then takes over. It takes a given request, finds an available worker, and then forwards the request to it. There are two ways to configure this request routing internally:

- per-worker
- pool-wide (global queue)

With the per-worker feature, Passenger maintains a separate queue for each worker. This can be problematic if a request comes in that takes a while and other requests are queued up behind it in the same worker.

The second option, a global queue, allows Passenger to put all requests on the same queue "stack". Workers then are given whatever is next on this stack, thus making the aforementioned long request situation a little less problematic.

Use Cases (Strategy)

Passenger is best used in situations where its unique spawning method can really shine. Passenger has the ability to kill off workers that don't have enough traffic to justify their existence (so, zero traffic) to conserve memory, and dynamically respawn them as needed.

This has several benefits and side effects. For an always-on, dedicated application, this behavior can be a real pain. While it can be changed, the problem with this method for large, dedicated applications is that, during periods of low traffic (during the night, for example), workers will be killed off, and then when traffic picks up in the morning, requests will hang inside Passenger while it tries to launch new worker processes in memory. This also happens after nginx is restarted.

There is a significant benefit for machines with multiple applications that have low to moderate traffic, however: with this spawning method, you can literally get more bang for your buck by allowing an "elastic" and automatic management of small applications by Passenger. This means you can deploy more applications to the same hardware, so long as their traffic is relatively minimal and the application is small enough to launch relatively quickly when Passenger needs to spin up another worker.

There are several features that other application servers have that Passenger doesn't - unless you pay for them. For example, Passenger 3 can't run multi-threaded. With Passenger 4, you can do this with the Enterprise version (which is a paid annual license to Phusion). Additionally, Passenger does not offer zero-downtime deploys (restarting one worker after another seamlessly) in its standard edition product, but that feature is available under the Enterprise license, as are several others.

In short, Passenger can be great for digital agencies looking to host multiple applications on medium-sized virtual machines.

Configuration (Training)

Passenger has a dizzying array of configuration options detailed at http://www.modrails.com/documentation/Users%20guide%20Nginx.html. Some of the more salient ones are listed here.

- `passenger_ruby` - tell Passenger which Ruby interpreter to use
- `passenger_spawn_method` - smart or direct; smart caches code on spawn, direct doesn't; direct is more compatible with some applications in some cases, but is slower

than smart

- `passenger_max_pool_size` - max number of application workers that can exist across ALL applications
- `passenger_min_instances` - the minimum number of application workers per application
- `passenger_pool_idle_time` - how long, in seconds, a worker can remain idle before being shut down; set this to zero to disable worker shutdown except for extenuating circumstances
- `passenger_pre_start_url` - by default, Passenger won't start a worker until it has requests for it, this option lets you pre-start workers based on the URL the app is going to be accessed at during nginx startup, Passenger spoofs a dummy request to the URI given during startup to "jump start" itself

All Passenger configuration belongs in nginx configuration. Various directives can exist in http, server, or location blocks.

Unicorn

Unicorn is an application server for fast clients and applications that has a really great operational infrastructure under the hood, relying on in-memory forking to recover crashed workers and pulling requests from a unix socket instead of a primary router process or thread. It can be configured to call Ruby blocks `before_fork` and `after_fork` during its operation, and can do zero downtime deploys when the master receives a HUP signal or a USR2+QUIT.

Mode of Operation (Fighting Style)

Unicorn, by contrast to Passenger, doesn't have an internal "tie-in" with nginx, nor does it have a single router process. Instead, Unicorn launches a master process that contains one single copy of your application in memory, and then forks itself into worker processes. The number of worker processes depends on how Unicorn is configured; it could be one to as many as the machine can reasonably hold.

Unicorn is then configured to bind to a unix socket on the local machine. Requests from nginx are then placed in this socket. Each worker then, of its own volition, dips into the socket, finds the next request to handle, works it, then returns a response to nginx.

The Unicorn master process stands by and observes each of its workers. If any worker becomes unresponsive, the master kills the worker and simply forks itself again. In this way, Unicorn's overall architecture is rather stable, and allows for hot restarts - restarting only one worker at a time after having code deployed.

Use Cases (Strategy)

ThinkGeek.com used to sell this awesome shirt that fans of the Matrix trilogy will get a good chuckle out of. You can think of the Unicorn master process as 'Agent Smith', cloning himself if one of the 'clones' dies.

When Unicorn starts up with an appropriate configuration, it will first read your application into memory as one master process. Then it will read that configuration and run its `before_fork` block. After that, Unicorn will fork itself into N number of workers as defined in its configuration, and then each worker will execute its own `after_fork` block. Unicorn's master process then sits back, grabs a cup of coffee and a newspaper, and plays "manager", just watching the other processes, ready to kill one off via SIGKILL if it takes too long to execute a request. If that happens, or if the process crashes, the Unicorn master simply forks itself to replace it.

Unicorn is built to cut off long running worker processes after a default of 30 seconds via SIGKILL. This is why Unicorn says it's built for "fast clients" - because anything beyond that cut-off is subject to termination. This can be changed in Unicorn's configuration, however.

Unlike Passenger (even version 4 - except enterprise), Unicorn is capable of a zero downtime deploy. By sending the Unicorn master a HUP signal, it will reload itself and its workers based off your most recent code deploy. Note that if using the `preload_app` feature, a USR2 + QUIT has to be sent to the master process instead of HUP. This tells the master to load up another copy of itself, and once it's verified that copy is running, kills off its old workers. Either case can affect a zero-downtime deploy, though the second of these can cause a temporary memory usage spike depending on the size of the application.

Unicorn is best used in situations with one specific application on a host, as it will spawn several workers as configured and maintain them at all times. Ergo, memory consumption could be a problem if you launch more workers than you have memory to reasonably support.

At Engine Yard, we've generally found that for dedicated applications, Unicorn is a better choice over Passenger for several reasons. First, even though Passenger is, for all intents and purposes, a quality product, we've seen several cases with version 3 where Passenger has crashed, failed, had a process become unresponsive or taken on some other extremely strange behavior that isn't logged anywhere and makes absolutely no sense. The term "gremlins" truly applies here. We see far less of this with Unicorn, and from our experience, we find that Unicorn tends to be quite a bit more stable. However, it should be noted again that this is with Passenger 3. We have yet to see this same degree of comparison versus Passenger 4, especially at high load, and I'm hopeful that this kind of comparison will no longer be valid with Passenger 4.

Unicorn is not well suited to applications that have long-running requests, large uploads, or long-polling/websockets, etc. as it doesn't run a threaded or evented architecture. It's very simple and straight-forward, finding any request it can, working it as fast as it can, and repeating the loop. Any request that takes longer than the configured cut-off will just be terminated.

All around, we find that Unicorn's best use case is with a single, dedicated application running at all times behind a reverse proxy (nginx) and load balancer (haproxy or ELB on Engine Yard) that stick to a regular, request/response application flow cycle. It generally processes requests very fast and has a highly stable architecture.

Configuration (Training)

Unicorn can use a configuration file that details, in Ruby, what to do `before_fork` , `after_fork` , and so on. For example, `before_forking` the master into a worker process, you want to disconnect ActiveRecord if it's got an open database connection; otherwise that could get pretty funky when you have several worker processes using the same database connection later on down the road. You can then re-establish a new connection inside the worker process by calling the appropriate method in an `after_fork` block in the Unicorn configuration.

Unicorn is generally set to listen to one specific socket on the local machine. You'd then have nginx put all its requests there as a location block in its configuration. Unicorn workers then proactively, of their own volition, examine that socket and grab requests to work from it directly, instead of having a single thread or process push requests to each worker independently.

Some of Unicorn's more interesting configuration options:

- `listen` : tells Unicorn where to put the socket for requests
- `timeout` : number of seconds to wait on a worker before forcibly killing it with a SIGKILL
- `worker_processes` : how many worker processes to fork; in memory, you'll have N+1 - the master isn't counted among the total here (Example: `worker_processes: 6` means you have 7 Unicorn processes - the master and 6 workers)
- `stdout/stderr_path` : logs output from stdout/stderr directly to the log file locations for each option

I generally recommend running multiple Unicorn workers in production. I would take the number of CPU cores on a machine and multiply by 1.5, plus one or two depending on expected load and the application. This ensures you have enough processes to continue allowing workers to process requests if one happens to be stuck with a long running request, or is waiting on I/O or network response for an abnormal amount of time. This is because, unlike Thin or Puma, Unicorn is neither evented nor is it multi-threaded.

Thin

Thin is an EventMachine-based application server for Ruby that claims to be the "most secure, stable, fast and extensible Ruby web server". It's similar to Unicorn in that it launches multiple workers and listens to a socket, but different in that it has no master controlling process and has one specific socket per worker process.

Mode of Operation (Fighting Style)

Thin works much like Unicorn, except that when started in cluster mode, each Thin worker opens its own socket, or is bound to its own port. Nginx can then be configured to "round-robin" balance requests between as many Thin sockets/ports as you have configured to start. It doesn't have a master process (by default) that runs and monitors the workers like Unicorn does, but it is capable of a "hot restart" using the "onebyone" option in configuration, which restarts one worker at a time for zero-downtime deploys.

Under the hood, Thin relies on an EventMachine-based architecture. This is not a fully asynchronous architecture like Puma, launching new requests in threads, but in theory it should allow for significant speed improvements by taking action only when enough data has been received from the client (when an "event" fires as opposed to waiting for something to finish).

Use Cases (Strategy)

Thin isn't yet available on Engine Yard (let us know if you'd like to see this change), though it can be installed via custom chef recipes, which I've done for this article (and the source code can be seen in the cookbooks/ directory of the source code linked at the beginning).

Like Unicorn, Thin is best suited to a single application running at all times on a given host. It runs based on EventMachine under the hood, meaning that applications that can benefit from an evented architecture - for example, a long-polling application - may benefit significantly from Thin.

Configuration (Training)

Thin's overall configuration is similar to Unicorn, though instead of just one socket, Thin has a socket for each individual worker when launched in cluster mode. This means that your reverse proxy (nginx in our case) needs to balance requests between those sockets. If you have, for example, 5 Thin workers, you should have 5 sockets and 5 entries in an nginx upstream:

```
upstream thin_upstream {
  server unix:/var/run/engineyard/thin.example.1.sock fail_timeout=0;
  server unix:/var/run/engineyard/thin.example.2.sock fail_timeout=0;
  server unix:/var/run/engineyard/thin.example.3.sock fail_timeout=0;
  server unix:/var/run/engineyard/thin.example.4.sock fail_timeout=0;
  server unix:/var/run/engineyard/thin.example.5.sock fail_timeout=0;
}
```

Thin can be booted with a configuration file in YAML format. Some of the more interesting options:

- environment: “production”, “staging”, etc.
- servers: number of application workers you want to run
- onebyone: boolean; whether or not to do hot restarts, restarting each worker “one by one”
- tag: a special tag that will be seen in ps output on the machine, “thin-MyAppName” for example

Even though Thin is evented, I would still recommend the same basic configuration for Thin’s number of production workers as I mentioned for Unicorn: `(num_cores * 1.5) + 1 || 2` . This should provide you with more than enough workers in the unlikely event that one gets tied down somehow.

Puma

Puma is an Engine Yard sponsored project. Its primary strength is that it’s a truly concurrent application server, unlike any of the other examples mentioned here (Thin’s concurrency support is labeled “experimental” at the time of analysis).

Mode of Operation (Fighting Style)

Puma can be configured to bind to ports, or to pull from a socket, just like Thin and Unicorn. However, unlike Thin and Unicorn, Puma will open a new thread for each incoming request. This means that blocking actions that aren’t necessarily heavy on CPU usage should not be a problem for Puma.

However, when discussing threading, we must constantly be aware of Ruby’s Global VM Lock (GVL). This is a limitation (possibly a feature?) of the language interpreter that ensures that even when launching new threads, except in specific cases, Ruby will only execute one Ruby code instruction at a time per process. MRI can still hand off async instructions to underlying C-based drivers, for example, but as for executing actual Ruby code, the GVL ensures that only one instruction is processed at a time per each Ruby process.

For this reason, Puma will run best under JRuby or Rubinius. Unfortunately, I didn't have time to profile Puma under either of these interpreters; instead performance benchmarking below is based on MRI 2.0. **This is, admittedly, *unfair* to Puma, but maintains comparison parity with the other application servers mentioned here.**

Use Cases (Strategy)

Unlike the above referenced app servers, Puma is a fully threaded application server. It doesn't rely on EventMachine like Thin, and like Unicorn it can run several workers pulling requests off of a centralized unix socket, or it can bind a worker to a port. Puma doesn't maintain a master process like Unicorn, however.

Because Puma is threaded, it can benefit more from JRuby or Rubinius than the other items here. Unfortunately there wasn't enough time to profile results under JRuby/Rubinius for Puma, but an application may see a performance gain even on MRI 2.0, depending on what's going on.

Puma is best suited to single applications running on a host, just like Thin and Unicorn. However, because it's multi-threaded, if you run a Ruby implementation without an internal GVL (JRuby/Rubinius), you can theoretically run a single Puma process on your machine instead of one worker per core (plus a few for good measure), as you generally have to for the other application servers mentioned here. This is because Puma can request a new thread from the operating system for each incoming request, drop execution of that thread while waiting on external events (disk I/O, database driver access, network I/O, etc.), pick up other threads while this is going on, and then pick up execution of the other thread after the "wait" event is finished.

Under MRI, because of the GVL, Puma can't quite shine as well as it would under a non-GVL-addled interpreter. However, even still, there are several cases where Puma may be a good fit. If you have an application that isn't particularly CPU bound, but does execute multiple external requests to databases, APIs, and disk I/O, Puma may provide a performance boost.

Configuration (Training)

Like Thin, Puma can take arguments on the command line or through a configuration file. However, since Puma is threaded, it has multiple other options designed to give you control over those threading capabilities.

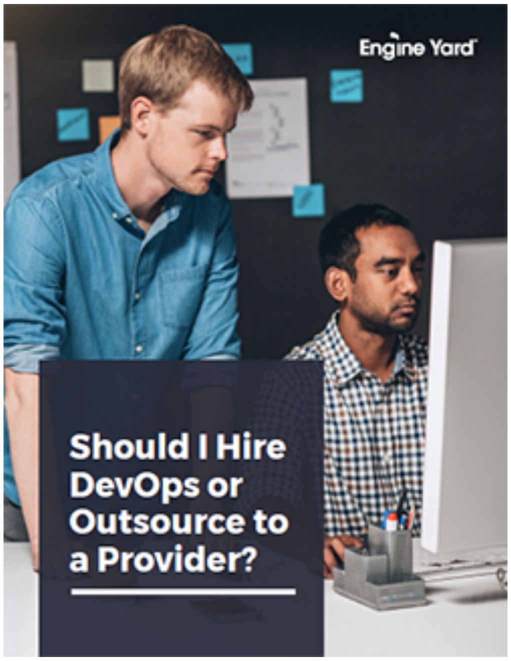
Like Unicorn, Puma maintains a single socket to pull requests from.

Some of Puma's more interesting options:

- `-e environment` - "production", "staging", etc.
- `--pidfile` - where to put the pidfile
- `-t X:Y` - minimum number of threads, maximum number of threads, per process (Example: -t 4:16)
- `-w X` - how many workers to spawn; theoretically not necessary on JRuby/Rubinius, but with MRI you should spawn the same number you would for Thin or Unicorn
- `-b:` - where to bind to; can be either a unix socket or a port

In the second post in this series, I detail the arena in which they do battle, results and analysis and key takeaways.

Comments


Engine Yard
Should I Hire DevOps or Outsource to a Provider?

- Please Select - ▾

□ Yes, subscribe me to Engine Yard blog post notification emails!

GIMME THE EBOOK

J. Austin Hughey

‹ Prev Post

Next Post ›

Subscribe Here!

Email*

Subscrib

System Status: All Systems Operational

Privacy Policy Leadership Contact

We Hang Out Here Too