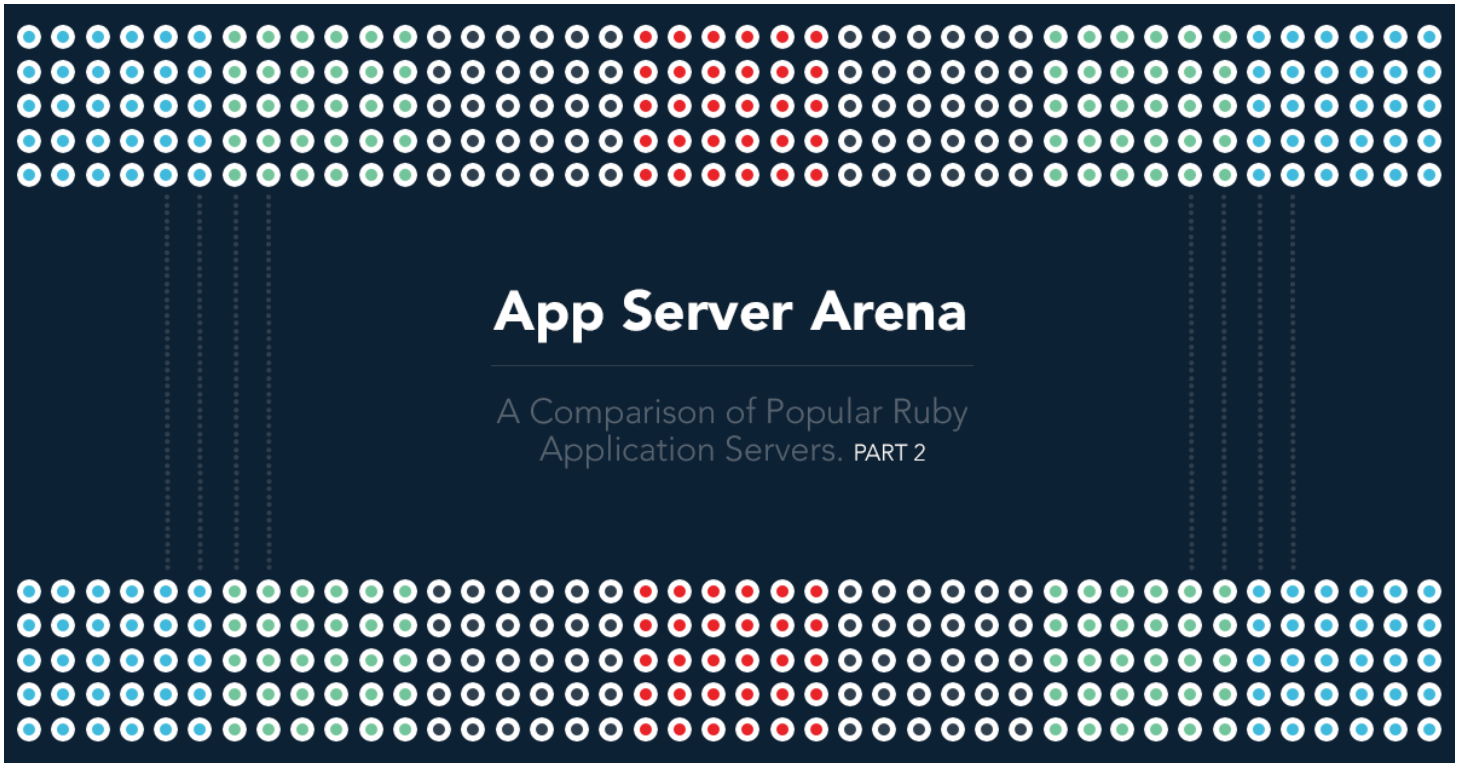
App Server Arena: Part 2, A Comparison of Popular Ruby Application Servers

June 25, 2014 | By J. Austin Hughey



In the first post in this series, I compared popular Ruby application servers Passenger, Thin, Unicorn and Puma across multiple categories: mode of operation, use cases and configuration. This post details a simplistic performance test and analysis.

The Arena

The arena in which all these application servers will do battle is a basic test application I built that is available on GitHub. It's a Sinatra app with some purely academic examples, not designed to do real, meaningful work, but instead to attempt a fair (or "fair enough") set of tasks for each application server to process and repeat multiple times.

`/server`

This action spits out information about the application server itself (pid of the worker, name of app server processing the request), and some information about the request and response environment variables. It's designed to be as close to the "metal" as reasonably possible, and does not reference any external data store.

`/pi`

Computes pi to 5,000 decimal places. Initially, I was doing 20k, but that turned out to cause all application servers to fall over very quickly into my tests, leaving me without enough sample size to be able to consider tests even borderline legitimate.

This task is designed to simulate mildly-computationally expensive work and see how each does under that particular type of strain.

`/borat`

I am so excite! The `/borat` resource fetches the last 10 tweets from @devops_borat. The intent here was to generate a test that would give a fair estimate of how various servers performed with real-world network latency, but unfortunately, Twitter's API rate limitations prevented me from testing this as thoroughly as I would have liked. I also was unable to include it in my `/random` test for that very reason. I've since removed this feature from the application's source code, though you should be able to see what it originally looked like by browsing the repo history.

`/sleep`

Sleeps for 1 second then returns a response. Spoofs unusually high request queuing and shows how each processes requests while stuck in a "wait-like" situation.

`/random`

Does one of the above at random with a 50% chance to run `/server`. The idea would be to simulate load based on several users swarming the application at once, all doing different things.

LET THE GAMES BEGIN!

In no case can any of the performance tests here be taken as "gospel", or in any way interpreted that these simplistic use cases are in any way all-encompassing. These are really straight-forward and simple tests that show different servers under different situations, but are not generalizable and applicable to even a majority of applications out there, especially those in complex problem domains.

In other words, your mileage may vary.

Testing hardware

This app was deployed to four different environments on Engine Yard Cloud backed by Amazon EC2 in US East 1. All VMs are High CPU Mediums. They come with 2 VCPU cores each and 1.7GB of memory. No database was used and no external data store (with the exception of the Twitter example, since a REST API could be considered a data store) was used in the app because I didn't want to artificially slow down tests because the app server was waiting on, for example, the database driver for whatever reason - that would just skew the tests in a weird way.

Testing methodology

I used a tool called Siege by Jeff Fulmer. I'd had major problems getting apache bench to work right and stumbled on Siege, which has a lot of very interesting options.

(Tip: you can install this crazy easy on OS X via brew install siege. If you don't have Homebrew, you really should get it.)

I used the following command/syntax to perform these tests, then put the results in flat files you can read under the "performance" subdirectory of the aforementioned repository:

```
siege -r 1000 -c 100 -b -q http://asa-puma/sleep > puma.txt 2>&1
```

I'm not concerned with doing anything too fancy here, I just want a straight up, "how fast can you do it?" test. With the `/borat` example hitting the Twitter API, I left off the `-b` (benchmark) option to allow siege to internally throttle requests so as not to get blacklisted from Twitter.

The arguments used:

- `-r` repetitions. Do this test N number of times, in this case, 1000.
- `-c` concurrency. How many simultaneous requests are we doing? In this case, 100 simultaneous requests, a thousand times over.
- `-b` benchmark. Removes internal throttling from siege. Really tries to hit the server crazy hard.
- `-q` quiet. Suppresses output that otherwise shows every. single. get. request. ever. made. throughout the entire test.

Other than that, I used the default settings for siege:

```
jaustinhughey:~/ $ siege -C

                                                                  [14:45:02]
CURRENT  SIEGE  CONFIGURATION
JoeDog/1.00 [en] (X11; I; Siege 3.0.0)
Edit the resource file to change the settings.
```

```
----------------------------------------------
version:                        3.0.0
verbose:                        true
quiet:                          false
debug:                          false
protocol:                       HTTP/1.1
get method:                     HEAD
connection:                     close
concurrent users:               15
time to run:                    n/a
repetitions:                    n/a
socket timeout:                 30
delay:                          1 sec
internet simulation:            false
benchmark mode:                 false
failures until abort:           1024
named URL:                      none
URLs file:                      /usr/local/Cellar/siege/3.0.0/etc/urls.txt
logging:                        true
log file:                       /usr/local/var/siege.log
resource file:                  /usr/local/Cellar/siege/3.0.0/etc/siegerc
timestamped output:             false
comma separated output:         false
allow redirects:                true
allow zero byte data:           true
allow chunked encoding:         true
upload unique files:            true
```

A quick note about the “concurrency” rating you’ll see in these tests: the higher it is, the worse the app server performed. This is because the number in the concurrency rating is how many requests are being processed at the same time, on average. If performance is high, very few things will be processed “at the same time” because formerly issued instructions (requests) will have already been processed. So, higher concurrency, lower performance according to siege output.

through `/sleep`. Passenger appears roughly on-par with the other gladiators when doing simple tasks under mild to moderate load. It had the highest rate of concurrency on `/borat` test.

Passenger outperformed Puma in the `/random` test yet maintained a very high degree of concurrency during the same. It failed to complete the `/random` test however, and had the longest "shortest duration" metric. Overall, it appears to be a solid contender for generic use cases and will likely outperform the others here in a low-to-moderate traffic, multi-application deployment due to its elastic loading of applications into memory, thus making your dollar go further. I would recommend this for development shops and digital agencies producing simple, low-traffic web applications for small businesses.

Raw data: sleep | borat | pi | random | server

Passenger 4

Passenger 4 was tested using the same methodology listed above in March 2014, some time after the original tests here were performed. I'm pleased to say that it has certainly improved in many areas versus its predecessor. However, I did not repeat the `/borat` test on Passenger 4 due to the same Twitter API rate limitations mentioned above.

Compared to Passenger 3, version 4 dramatically improved on the `/pi` and `/random` tests, completing all transactions. It took over three hours to complete the `/pi` test, and had a transaction rate of 8.8 transactions per second - roughly on par with all other application servers tested though significantly behind Unicorn and Thin. Passenger 4 performs on par or better on the `/server` test than any other application server tested here, was roughly equal to Unicorn in the `/sleep` and `/random` tests.

Raw data: pi | random | server | sleep

Unicorn

Unicorn was faster than Passenger or Thin with `/borat` and had the second highest transaction rate for the same. It had the shortest duration of both longest and shortest transaction with `/borat` as well. It was one of only two that actually finished the `/pi` test, and did so faster than Thin. Unicorn had the highest concurrency and transaction rates on `/pi` of all app servers tested and the shortest "longest transaction" on `/pi`. It completed the entire `/server` test; second in total duration only to Thin. Unicorn was the only contestant to complete all `/sleep` tests and still managed to have the shortest "longest transaction" duration.

It had zero failed transactions on `/random` with highest concurrency rating.

Raw data: sleep | borat | pi | random | server

Thin

Thin was one of only two to complete the `/pi` test, though it was slightly slower than Unicorn. It had the shortest transaction under `/pi` and quickest "longest transaction" under `/server`. Thin had the highest concurrency rating and transaction rate under `/server` and unfortunately failed miserably with `/sleep`, only completing 66 requests!

It was one of only two to finish the `/random` test, though it took much longer than Unicorn and had the lowest transaction rate. Thin appears to be an excellent choice for any application that doesn't rely on lots of wait-like scenarios or that don't have much request queueing.

Raw data: sleep | borat | pi | random | server

Puma

Puma was the fastest to finish `/borat` but failed miserably with the `/pi` test. It was on-par with Unicorn's speed for finishing the `/server` test, and had the second highest transaction rate for the same. It failed the `/sleep` test horribly, finishing less than 25% of the test.

Puma also failed the `/random` test, finishing less than 25%. I found this extremely odd so I went for a second round for Puma on the `/random` test, thinking that perhaps over my planned 100,000 requests, it was thrown an inordinate number of "pi" tests (because it is random, after all). The results were repeated the second time around. While it's still possible that it was hit with an unusually large number of "pi" tests, it's not as likely as it just plain doesn't work well when presented with computationally expensive tasks while processing multiple other incoming requests.

In all fairness, due to Ruby's GVL, I should have at least run more Puma workers. Instead, in the interest of research, I ran only as many as there are CPU cores on the machine (2) because I wanted to see, even with a 1:1 worker:core ratio, how Puma performed compared to its contemporaries. In most cases, the `/pi` and `/random` tests being notable exceptions, it performs on-par with the others tested here in spite of having far fewer worker processes.

Raw data: sleep | borat | pi | random | server

Key Takeaways

After performing these tests, there are a few basic generalizations we can make that should give developers wondering which application server to use a fair idea of where to start. While these generalizations won't apply to every situation, they should provide a fair overview of strengths and weaknesses based on the data above.

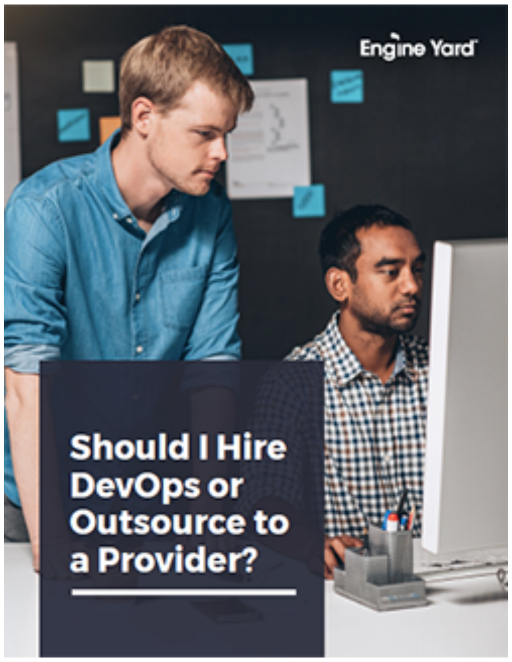
Use Case	Recommended Application Server
Multiple apps, low traffic, limited hardware	Phusion Passenger
Live Streaming in Rails 4, Long Polling	Thin, Puma (on JRuby or Rubinius for threads)
Single application fast request/response (API for example)	Unicorn

If you're unsure of which application server to use for your project, consider starting with Passenger 4 on your local development machine to test for unusual behavior. If you find none, deploy to a staging environment with Passenger 4 and try throwing it a lot of traffic from your favorite load testing application. See how well it holds up. If you find that it's not performing well, consider the guidelines here and attempt deploying under a different application server.

Deploy your Ruby app for free on Engine Yard.
Start Trial

Other Notes

This test wasn't as fair as it should have been to Puma because of the GVL in MRI. The test should be repeated under JRuby and/or Rubinius to see how it performs when it has all the tools it's supposed to.



FIRST NAME *

First Name*

EMAIL *

Email*

NUMBER OF SERVERS *

- Please Select -

DOES YOUR APP RUN ON RUBY? *

○ Yes
○ No

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

GIMME THE EBOOK

J. Austin Hughey

Comments

‹ Prev Post

Next Post ›

Subscribe Here!

*Email**

Subscribe

System Status: **All Systems Operational**

Privacy Policy **Leadership** **Contact**

We Hang Out Here Too