Understanding New Relic Queuing

January 10, 2013 | By J. Austin Hughey

Most of us have all seen that dreaded seafoam green graph when using New Relic. You know the one -__request queueing__. Sending a cold shiver up your spine, it depicts a slow application struggling to serve your users and an increasing probability that those users will "bounce" away from your app and into the arms of another.

...or does it?

Unfamiliar with New Relic? They're one of our partners, and you should most certainly check them out.

Request Queuing in New Relic is possibly one of the most misunderstood metrics in modern web application performance monitoring, and at Engine Yard, we get a lot of questions about it. Most developers understand this as a measurement of how long requests sit around, on average, before being worked by an application worker process (e.g. Passenger or Unicorn worker).

Unfortunately, this is a complete misunderstanding of the metric, how it's measured, and what it means. This article aims to better explain what Request Queuing actually is, differentiate between queuing and latency, and help developers better understand when an application really has issues, and when request queuing figures are simply an unavoidable fact of life.

There are two figures here that need to be paid attention to - how many requests are in queue to be worked at a given time, and how long it takes a request to be worked from receipt by nginx to processing by an application worker. This is a subtle, yet critical difference in the way that things work, and understanding these differences will greatly help you in diagnosing potential performance issues with your application and/or configuration.

So let's start this article by differentiating between request queuing, and request latency.

Request Latency

Most of the time, when people see New Relic graphs like the below and think about request queuing, what they're really seeing is request latency.

In reality, what the dreaded seafoam-green representation on this graph depicts is a simple mathematical subtraction of the time between when nginx inserts a time tracking header, and when the New Relic Agent parses that header - nothing more.

To illustrate, take a look at this example nginx configuration. This is part of our standard configuration on Engine Yard Cloud:File: /etc/nginx/common/proxy.conf - included from /etc/nginx/servers/appname.conf

```
 index index.html index.htm;

# needed to forward user's IP address to rails
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header Host $http_host;
proxy_set_header X-Forwarded-Proto $scheme;
proxy_set_header X-Queue-Start 't=$start_time';

proxy_redirect off;
proxy_max_temp_file_size 0;
```

As this configuration shows, nginx adds a header called X-Queue-Start to each HTTP request. Later, as the request is processed by your application worker, New Relic looks for that header:

```
 module NewRelic
module Agent
module Instrumentation
module QueueTime
unless defined?(MAIN_HEADER)
MAIN_HEADER = 'HTTP_X_REQUEST_START'
MIDDLEWARE_HEADER = 'HTTP_X_MIDDLEWARE_START'
QUEUE_HEADER = 'HTTP_X_QUEUE_START'
ALT_QUEUE_HEADER = 'HTTP_X_QUEUE_TIME'
HEROKU_QUEUE_HEADER = 'HTTP_X_HEROKU_QUEUE_WAIT_TIME'
APP_HEADER = 'HTTP_X_APPLICATION_START'
```

https://github.com/newrelic/rpm/blob/master/lib/new_relic/agent/instrumentation/queue_time.rb#L8

And then parses it:

```
 # Checks that the time is not negative, and does the actual # data recording
def record_time_stat(name, start_time, end_time) # (String, Time, Time) -> nil
total_time = end_time - start_time
if total_time < 0
raise "should not provide an end time less than start time: #{end_time.strftime(
'%s.%N')} is less than #{start_time.strftime('%s.%N')}. total time is #{total_ti
me}."
else
NewRelic::Agent.get_stats(name).trace_call(total_time)
end
end
```

https://github.com/newrelic/rpm/blob/master/lib/new_relic/agent/instrumentation/queue_time.rb#L173

So how did we get to that specific point in the code? In NewRelic::Agent::Instrumentation::Queutime#parse_queue_time_from, the object 'matches' is derived from get_matches_from_header, which specifies QUEUE_HEADER, in this case 'HTTP_X_QUEUE_START' which was added by nginx upstream. That 'matches' object is then passed to record_rollup_queue_stat with the variable ‘end_time’, which will ultimately be Time.now. The record_rollup_queue_stat method calls record_rollup_stat_of_type with the end time and matches objects previously defined. That method then calls record_time_stat, which we see above calculates the “total_time” value as simple subtraction and reports it to New Relic.

According to our own code review, and even according to New Relic’s engineers in private discussions we’ve had with them, at no point does the New Relic Agent actually make an attempt to look at a Unicorn socket or Passenger master process to determine how many requests are actually queued inside of it. Ergo, this measurement is far different than most developers expect.

The mere presence of “request latency” (what New Relic calls “queuing”) isn’t necessarily indicative of a problem, though may be under certain circumstances.

So how is this useful?

If you’re seeing request latency show up in New Relic, it’s generally an indication that something is getting “in front of” New Relic’s ability to parse that header. It’s usually not a major concern unless you start seeing those numbers fly wildly out of control.

In our experience, large spikes and steep valleys generally reflect a few controller actions or series of actions or application flow that has something jumping in front of New Relic in the order of processing a request. This could be a before_filter called on only certain actions for certain reasons in certain conditions, for example. Other causes could include application server restarts (Passenger, Unicorn) due to memory constraints, resource contention, slow database queries, or various forms of timeouts.

Conversely, situations with a large amount of request latency that are constant and stable usually reflect a series of over-used before_filters, slow Rack middleware, resource contention (usually being CPU bound), slow connections to frequently-called external services, or constantly slow database performance.

We’ve seen before_filters come into play because they’re executed before New Relic calculates its request “queuing” time. In those cases, whatever slows down the before filter contributes to request latency. This can give an application the appearance of having a problem, when in fact, there may not actually be one. It really depends on what the actual time (in milliseconds) is in request latency, whether or not it’s necessary for most or all requests, and whether or not that number is reasonable. In most cases, ~500ms of latency is totally unreasonable, but ~25ms may be reasonable if the Rack middleware or before_filters in use are indeed necessary. However, you may have to be more flexible on those numbers depending on what dependent elements of your application are responsible for execution. For example, if you're retrieving information from a REST API, and a large amount of geographic distance exists between your servers and the location of the API, network latency could end up being a major contributor. (The same could apply to DBaaS services, depending on where their endpoints are.)

Additionally, a resource contention issue could be at play in these situations as well. When I say "resource", I have a very broad meaning in mind intentionally. If your application depends on external web services that are frequently called and they're slow, that can be a potential cause. In other cases, your database may be getting hammered (perhaps another application is experiencing high traffic, some one ran a report on the same machine you're doing your reads from, or a cache expired and is being rebuilt). Perhaps your application is simply under a great deal of traffic and you're getting CPU bound. On Amazon EC2, you could even be experiencing slow EBS I/O (since EBS is a network-based storage system). Regardless, when you see long periods of sustained request queuing, it's often useful to rule out that *any* external resource on which your application depends is the bottleneck. Generally speaking, you'll usually find that something is indeed slowing things down if you look hard enough.

If you see large "spikes" in request queuing, I recommend checking the following items, at a bare minimum: Examine CPU usage on all instances. Are the CPUs busy? Examine memory usage on all instances. Are you utilizing swap? (ssh in, issue 'free -m' to see) Check /var/log/syslog for indications that Passenger is being restarted, or /data/appname/log/unicorn*.log for indications that Unicorn is being restarted. If you're using Passenger, has nginx been recently restarted or tmp/restart.txt been touched? Have you just pushed out a deploy by chance? This causes a reload of application workers, which can cause requests to queue while new workers are spun up. Check your application logs for repeated stack traces. Look at any databases that your application is using and check for a slow query log. For example, on a MySQL database master or slave, you can look at /db/mysql/log/slow_query.log. Are there slow queries listed here? Are they executed in the requests that are slow? Has your cache recently expired and needed to be regenerated? If you have a front-end cache (e.g. CDN, CloudFlare), did that recently "tap" your server/application for fresh data?

While the above is not an exhaustive list, it's a place to start that should give you a good idea of the kinds of things to explore.

Request Queuing

By contrast, request queuing is the actual measure of how many HTTP requests are sitting inside of a Unix socket or Passenger master process and are waiting to be picked up by an application worker process, and how long, on average, those requests sit idle before being worked.

We're going to look at two different application servers in this article: Passenger (v3) and Unicorn. WIthout getting terribly deep into either, the basic difference on request routing is that with our default Passenger configuration, requests are taken by nginx, the header added as seen above, and then forwarded to the Passenger master process for "balancing" among the workers in its pool. With Unicorn, the requests are put directly into a Unix socket on the local machine, where Unicorn workers go and pull the request straight out of the socket of their own volition.

But what happens if requests aren't being processed as fast as they pile up? This is what real request queuing actually is. In that case, requests will "queue up" inside a Passenger master process or Unix socket while waiting for a worker to become available.

So the question then becomes, if you suspect actual request queuing (and not latency), how can you examine your worker processes and their various queuing mechanisms to see whether or not this is going on?

Passenger

This is really easy with Passenger: just type "passenger-status" on the command line:

```
 app i-xxxxxxxx ~ # passenger-status
----------- General information -----------
max = 16
count = 7
active = 0
inactive = 7
Waiting on global queue: 0

----------- Application groups -----------
/data/appname/current:
App root: /data/appname/current
* PID: 5660 Sessions: 0 Processed: 235 Uptime: 17m 18s
* PID: 17067 Sessions: 0 Processed: 2654 Uptime: 3h 46m 33s
* PID: 30527 Sessions: 0 Processed: 401 Uptime: 46m 56s
* PID: 17070 Sessions: 0 Processed: 2942 Uptime: 3h 46m 33s
* PID: 17088 Sessions: 0 Processed: 3854 Uptime: 3h 46m 25s
* PID: 17061 Sessions: 0 Processed: 1383 Uptime: 3h 46m 33s
* PID: 17073 Sessions: 0 Processed: 3243 Uptime: 3h 46m 33s
```

You'll see output that shows you a great deal of information about each Passenger worker - how long it's been up, how many requests it's processed and, yes, how many requests are actually in the queue. You can watch this for a few minutes automatically by issuing "watch passenger-status". The command will automatically be re-run every 2 seconds and the output shown on your screen. This is a much easier way to take a quick peek, but may be a little harder to deal with programmatically.

Now, there are two things to look at here. In the "General Information" section of passenger-status we can see that Global Queuing is indeed turned on. This is default with our platform. In a nutshell, global queuing basically makes Passenger pull all requests off the same queue. Otherwise, each Passenger worker has its own private queue, and the master would forward requests to whichever process has the least requests in its queue. We have seen situations where Passenger can get a little flaky, so in some cases a simple restart (/etc/init.d/nginx restart) can help. If that fails, however, ask yourself a few questions:

Do your requests rely on an external web service or API? Is that down or slow?

Is your database performing as fast as normal?

How is EBS I/O load on the instance? Did Amazon have another "event"?

Did anyone on your team just deploy code recently? Even if it wasn't announced?

Is that deploy still in progress by chance? (tail -f /home/deploy/appname-deploy.log)

What's load like on the instance? Are you memory or CPU bound at all?

The general idea is to find a bottleneck, eliminate it, and re-assess. And anything that your application depends on is a potential bottleneck - even general latency over the internet itself can cause cascading effects throughout your app.

Unicorn

By contrast, Unicorn requires you to write a little code to examine the status of its Unix socket. Using Raindrops, one of Unicorn's dependencies, we can open up the Unix socket that nginx puts all incoming requests into and look at what's inside of it. Here's an example script you could use, also available at github: https://github.com/jaustinhughey/unicorn-status

```
 require 'rubygems'
require 'unicorn'

# Usage for this program
def usage
puts "ruby unicorn_status.rb "
puts "Polls the given Unix socket every interval in seconds. Will not allow you
to drop below 3 second poll intervals."
puts "Example: ruby unicorn_status.rb /var/run/engineyard/unicorn_appname.sock 1
0"
end

# Look for required args. Throw usage and exit if they don't exist.
if ARGV.count < 2
usage
exit 1
end

# Get the socket and threshold values.
socket = ARGV[0]
threshold = (ARGV[1]).to_i

# Check threshold - is it less than 3? If so, set to 3 seconds. Safety first!
if threshold.to_i < 3
threshold = 3
end

# Check - does that socket exist?
unless File.exist?(socket)
puts "Socket file not found: #{socket}"
exit 1
end
```

```
# Poll the given socket every THRESHOLD seconds as specified above.
puts "Running infinite loop. Use CTRL+C to exit."
puts "------------------------------------------"
loop do
Raindrops::Linux.unix_listener_stats([socket]).each do |addr, stats|
header = "Active Requests Queued Requests"
puts header
puts stats.active.to_s + stats.queued.to_s.rjust(header.length - stats.active.to
_s.length)
puts "" # Break line between polling intervals, makes it easier to read
end
sleep threshold
end
```

This script looks at a given Unicron socket every N seconds as specified on the command line, and gives you a basic idea as to what's inside it. We recommend keeping the poll interval fairly low (definitely no faster than every 3 seconds) since the more often you poll the socket, the higher chance you'll seriously degrade performance on the machine.

Using this script would be fairly simple. You can use wget to put it on an instance as described in the readme in its GitHub repository, and then run it like so:

```
ruby unicorn_status.rb /path/to/your/unicorn/socket.sock 10
```

We generally store Unicorn sockets under /var/run/engineyard/unicorn_appname.sock, where appname is the name of your application. Another way to find this would be to look at the upstream definition in /etc/nginx/servers/appname.conf (again, appname is a stand-in for the name of your application). The first few lines should define an upstream that looks similar to the following:

```
upstream upstream_todo {
server unix:/var/run/engineyard/unicorn_todo.sock fail_timeout=0;
}
```

You can see here that in this case, nginx is going to proxy requests into /var/run/engineyard/unicorn_todo.sock. That's where you'd tell the above script to look:

```
ruby unicorn_status.rb /var/run/engineyard/unicorn_todo.sock 10
```

You'll see output similar to the following:

```
 app_master i-143b2f69 ~ # ruby unicorn_status.rb /var/run/engineyard/unicorn_tod
o.sock 10
Running infinite loop. Use CTRL+C to exit.
----------------------------------------
Active Requests            Queued Requests
47                                       0
```

If you suspect that actual request queuing is going on, and not just latency, use the above script to examine your socket (if you're on Unicorn; otherwise, just use 'passenger-status' as mentioned above). This should give you a much better idea of where the problem may be, if indeed a real problem exists in the first place. So what do I do if there's no actual queuing, but lots of latency?

In cases where the New Relic graphs indicate lots of request queuing, but you can actually prove that there isn't any, there are generally a few things to look at.

First, are you running the latest version of the New Relic gem in your app? We've seen cases where upgrading does solve measurement issues, so we do recommend trying that as a "low-hanging fruit" first measure.

Do you have custom nginx configuration in place by chance? Some custom nginx configuration options can interfere with nginx's setting and timing of that header, depending on how they're implemented. Try disabling or refactoring any custom configuration you have in place and observe what effect it has on request queuing metrics.

Look at the request queuing metrics that have come through and see if you can find common patterns pertaining to your code. Are they all on the same controller or action? Are they all using a common before_filter? What does that before_filter look like? If you have Rack middleware, try temporarily disabling it to see if that makes a difference.

As an example of diagnosing your application for request queuing issues, consider the following story. We had one customer who encountered this issue. They had a before_filter executing on every request that looked up a UUID in a MySQL database. As you may (or may not) know, MySQL doesn't have a native UUID datatype like PostgreSQL does, so it was being stored as a string in their database. Furthermore, this customer had accidentally forgotten to place an index on that column, meaning that every lookup was taking several hundred milliseconds per request (and that would worsen dramatically as the company grew, since each lookup involved a full table scan). Once we pointed this out, they properly indexed the column and tweaked their before_filter to run only on actions that absolutely required it, and request “queuing” figures in New Relic dropped dramatically - from ~500ms per request to about ~25ms.

Debugging New Relic

New Relic comes with a special “debugging” mode that you can enable for your application.__Be really careful about doing this in production.__Wait for a low-traffic time before you do so, or you could be making things worse than they are. To use it, you’ll need to modify config/newrelic.yml on your server or servers in question and explicitly set the log_level to debug:

```
 # The newrelic agent generates its own log file to keep its logging
# information separate from that of your application. Specify its
# log level here.
log_level: debug
```

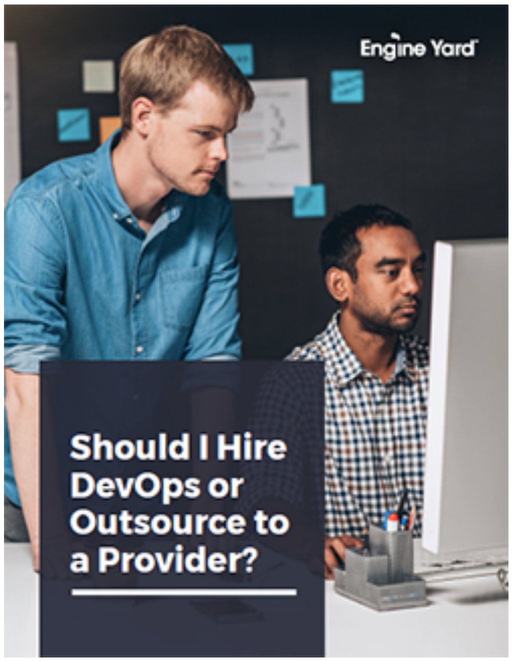
After you do this, you’ll need to restart your application server (Passenger/Unicorn). Then you should be able to tail -f /data/appname/current/log/newrelic_agent.log to see diagnostic information come through.

As we’ve seen, what New Relic calls “request queuing” really a measure of latency instead. The mere presence of this latency should not cause you alarm unless those numbers get really, really high during low to moderate traffic periods. Unless your site is down (and you should have other monitoring in place for that), request queuing is never a cause for alarm or emergency treatment, merely for study and adjustment as needed.

New Relic is an extremely useful tool for diagnosing application problems and monitoring overall performance. Understanding how its request queuing measurements work should help you use it more effectively, and know how to properly react to the data reported.

Comments

Free Ebook: Should I Hire DevOps or Outsource to a Provider?



You have to invest in your infrastructure: Do you hire DevOps for this critical function, assign it to your already overworked engineers, or outsource to a provider that offers full-stack capabilities?

FIRST NAME *

*First Name**

EMAIL *

*Email**

NUMBER OF SERVERS *

DOES YOUR APP RUN ON RUBY? *

○ Yes
○ No

- 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