MARCH 24, 2017 / #NGINX

NGINX rate-limiting in a nutshell

by Sébastien Portebois

NGINX is awesome… but I found its documentation on rate limiting to be somewhat… limited. So I've written this guide to rate-liming and traffic shaping with NGINX.

We're going to:

- describe the NGINX directives
- explain NGINX's accept/reject logic
- help you visualize how a real burst of traffic is processed using various settings: rate-limiting, traffic policy, and allowing small bursts

As a Bonus, I've included a a GitHub repo and the resulting Docker image so you can experiment and reproduce the tests. It's always easier to learn by doing!

NGINX rate-limit directives and their roles

This post focuses on the ngx_http_limit_req_module, which provides you with the `limit_req_zone` and `limit_req` directives. It also provides the `limit_req_status` and `limit_req_level`. Together these allow you to control the HTTP response status code for rejected requests, and how these rejections are logged.

Most confusion stems from the rejection logic.

First, you need to understand the `limit_req` directive, which needs a `zone` parameter, and also provides ***optional*** `burst` and `nodelay` parameters.

count the incoming requests. All requests coming into the same bucket will be counted in the same rate limit. This is what allows you to limit per URL, per IP, or anything fancy.

- `burst` is optional. If set, it defines how many exceeding requests you can accept over the base rate. One important thing to note here: *burst is an absolute value, it is not a rate.*
- `nodelay` is also optional and is only useful when you also set a `burst` value, and we'll see why below.

How does NGINX decide if a request is accepted or rejected?

When you set a zone, you define a rate, like `300r/m` to allow 300 requests per minute, or `5r/s` to allow 5 requests each second.

For instance:

- `limit_req_zone $request_uri zone=zone1:10m rate=300r/m;`
- `limit_req_zone $request_uri zone=zone2:10m rate=5r/s;`

It's important to understand that these 2 zones have the same limits. The `rate` setting is used by NGINX to compute a frequency: what is the time interval before to accept a new request? NGINX will apply the leaky bucket algorithm with this token refresh rate.

For NGINX, `300r/m` and `5r/s` are treated the same way: allow one request every 0.2 seconds for this zone. Every 0.2 second, in this case, NGINX will set a flag to remember it can accept a request. When a request comes in that fits in this zone, NGINX sets the flag to false and

the flag was already set to accept a request, nothing changes.

Do you need rate-limiting or traffic-shaping?

Enter the `burst` parameter. In order to understand it, imagine the flag we explained above is no longer a boolean, but an integer: the max number of requests NGINX can allow in a burst.

This is no longer a leaky bucket algorithm, but a token bucket. The `rate` controls how fast the timer ticks, but it's no longer a true/false token, but a counter going from `0` to `1+burst value`. Every time the timer ticks, the counter is incremented, unless it is already at its maximum value of `b+1`. Now you should understand why the `burst` setting is a value, and not a rate.

When a new request comes in, NGINX checks if a token is available (i.e. the counter is > 0), if not, the request is rejected. If there is a token, then the request is accepted and will be treated, and that token will be consumed (the counter is decremented).

Ok, so NGINX will accept the request if a burst token is available. *But when will NGINX process this request?*

You asked NGINX to apply a maximum rate of `5r/s`, NGINX accepts the exceeding requests if burst tokens are available, but will wait for some room to process them within that max rate limit. Hence **these burst requests will be processed with some delay**, or they will time out.

In other words, NGINX will not go over the rate limit set in the zone

received.

To use a simple example, let's say you have a rate of `1r/s`, and a burst of `3`. NGINX receives 5 requests at the same time:

- The first one is accepted and processed
- Because you allow 1+3, there's 1 request which is immediately rejected, with a 503 status code
- The 3 other ones will be treated, one by one, but not immediately. They will be treated at the rate of `1r/s` to stay within the limit your set. If no other request comes in, already consuming this quota. Once the queue is empty, the burst counter will start to be incremented again (the token bucket starts to be filled again)

If you use NGINX as a proxy, the upstream will get the request at a maximum rate of `1r/s`, and it won't be aware of any burst of incoming requests, everything will be capped at that rate.

You just did some traffic shaping, introducing some delay to regulate bursts and produce a more regular stream outside of NGINX.

Enter nodelay

`nodelay` tells NGINX that the requests it accepts in the burst window should be processed immediately, like regular requests.

As a consequence, the spikes will propagate to NGINX upstreams, but with some limit, defined by the `burst` value.

a hands-on fashion, I set up a small Docker image with a NGINX config exposing various rate-limit settings to see the responses to a basic rate limited location, to a `burst`-enabled rate-limited location, and to `burst` with `nodelay` rate-limited location, let's play with that.

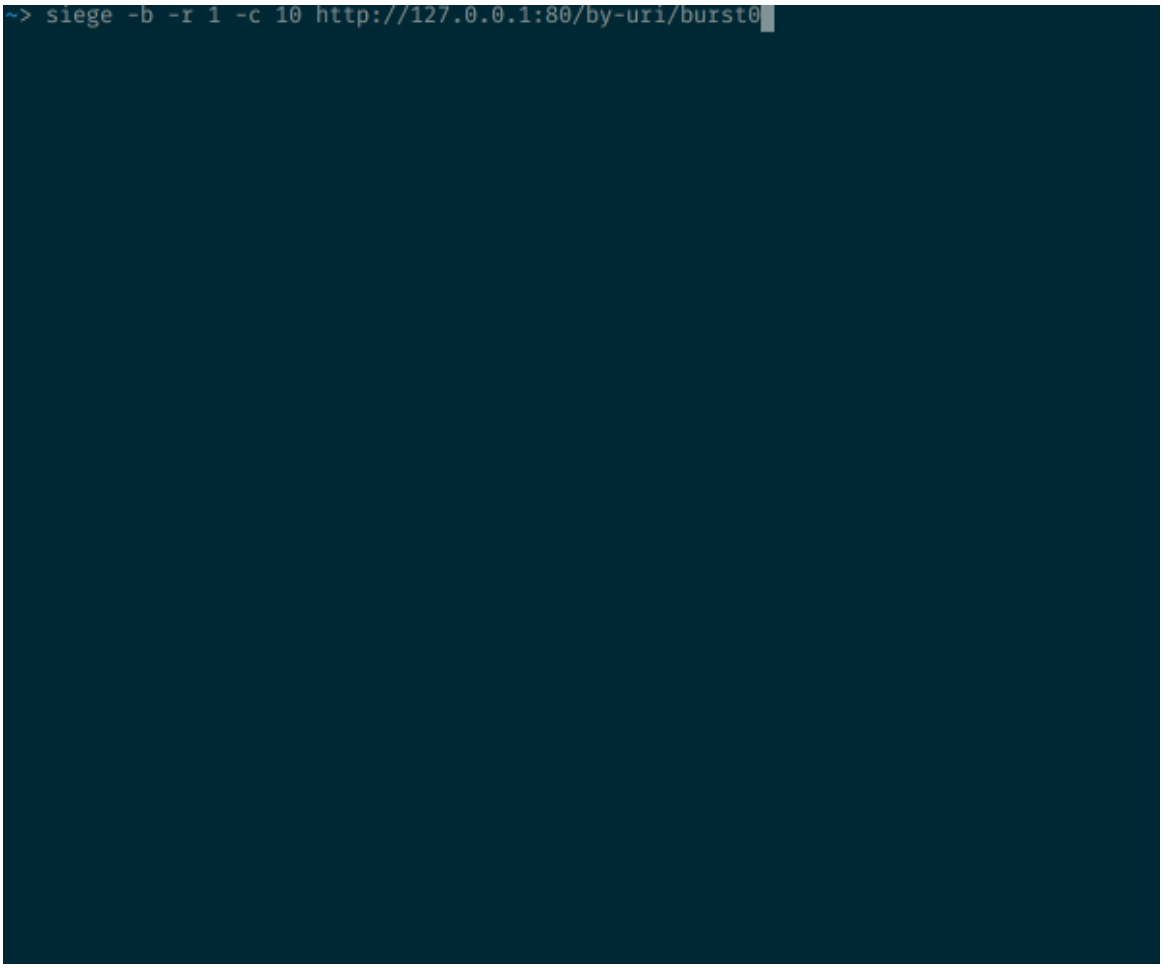
These samples use this simple NGINX config (which we'll provide a Docker image for at the end of this post so you can more easily test this):

```
limit_req_zone $request_uri zone=by_uri:10m rate=30r/m;

server {
    listen 80;

    location /by-uri/burst0 {
        limit_req zone=by_uri;
        try_files $uri /index.html;
    }

    location /by-uri/burst5 {
        limit_req zone=by_uri burst=5;
        try_files $uri /index.html;
    }

    location /by-uri/burst5_nodelay {
        limit_req zone=by_uri burst=5 nodelay;
        try_files $uri /index.html;
    }
}
```

Starting with this config, all the samples below will send 10 concurrent requests at once. Let's see:

- how many get rejected by the rate-limit?

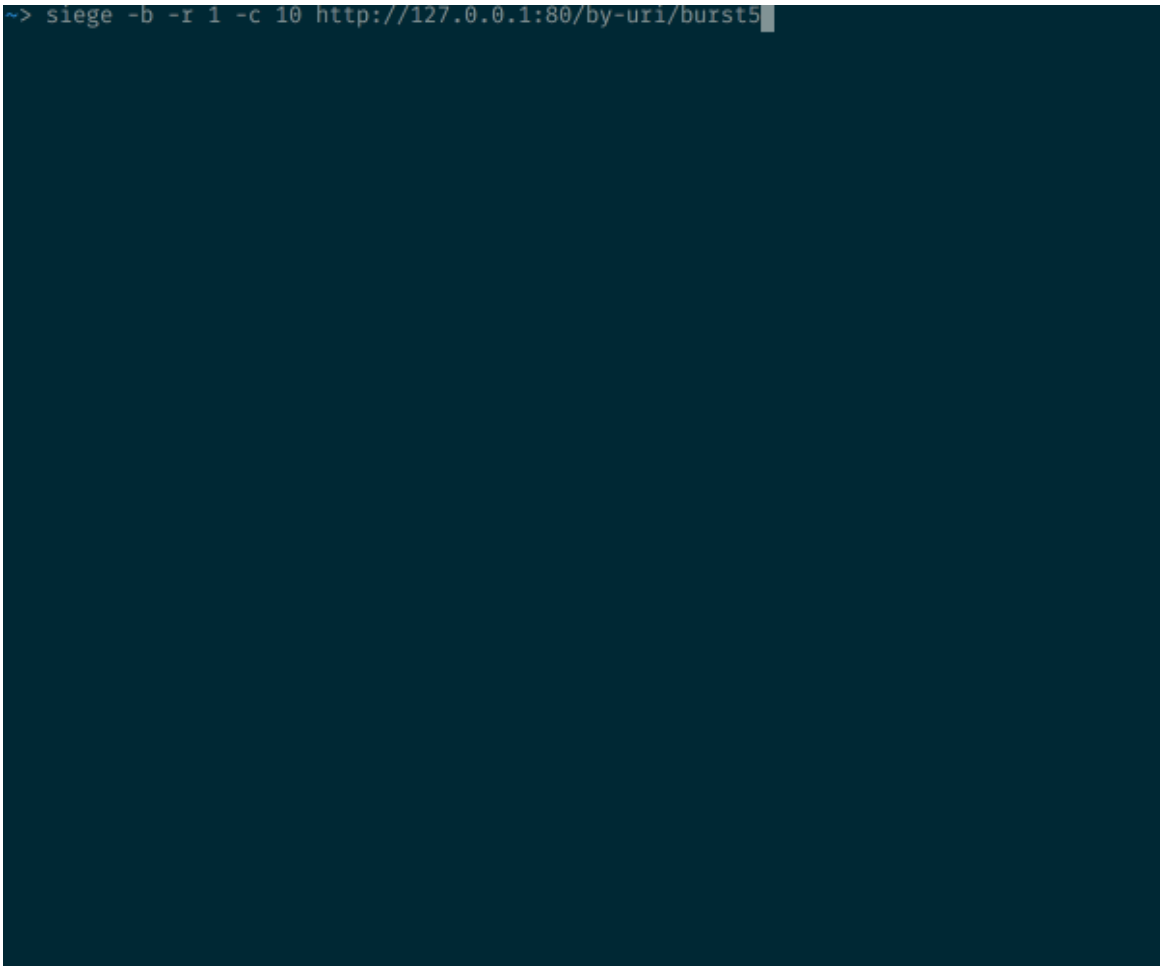
Sending 10 parallel requests to a rate-limited endpoint



10 requests hitting a rate-limited endpoint at the same time

That config allows 30 requests per minute. But 9 out of 10 requests are rejected in that case. If you followed the previous steps, this should make sense: The `30r/m` means that a new request is allowed every 2 seconds. Here 10 requests arrive at the same time, one is allowed, the 9 other ones are seen by NGINX before the token-timer ticks, and are therefore all rejected.

Ok, so let's add the `burst=5` argument to let NGINX handle small bursts for this endpoint of the rate-limited zone:



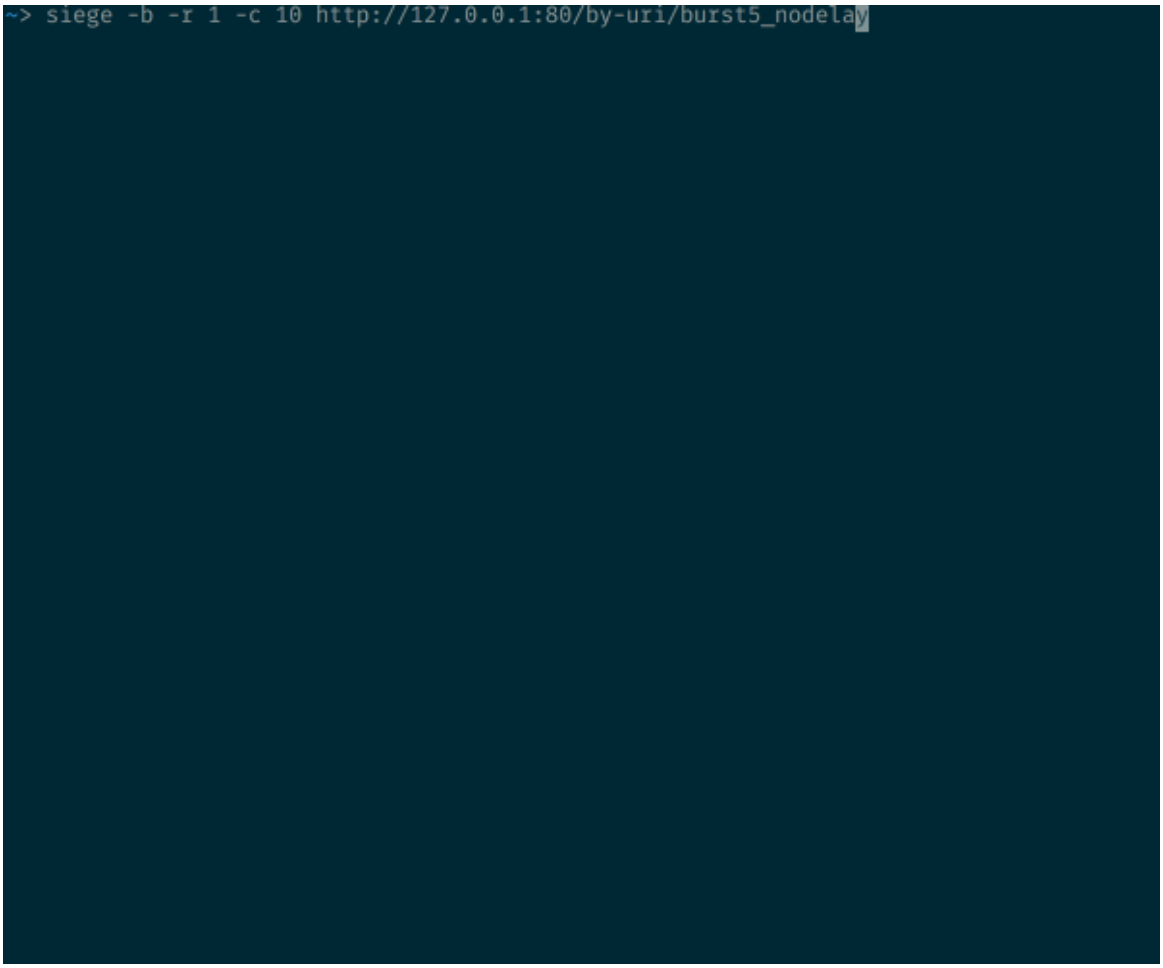
10 concurrent requests sent at once to an endpoint with a burst=5 argument

What's going on here? As expected with the `burst` argument, 5 more requests are accepted, so we went from 1 /10 to 6/10 success (and the rest is rejected). But the way NGINX refreshed its token and processed the accepted requests is quite visible here: the outgoing rate is capped at `30r/m` which is equivalent to 1 request every 2 seconds.

ticks again, processing another pending request, which is returned with a total roundtrip time of 4.02 seconds. And so on and so forth… The `burst` argument just lets you turn NGINX rate-limit from some basic threshold filter to a traffic shaping policy gateway.

My server has some extra capacity. I want to use a rate-limit to prevent it from going over this capacity.

In this case, the `nodelay` argument will be helpful. Let's send the same 10 requests to a `burst=5 nodelay` endpoint:



As expected with the `burst=5` we still have the same number of status 200 and 503. But now the outgoing rate is no longer strictly constrained to the rate of 1 requests every 2 seconds. As long as some burst tokens are available, any incoming request is accepted and processed immediately. The timer tick rate is still as important as before to control the refresh/refill rate of these burst tokens, but accepted requests no longer suffer any additional delay.

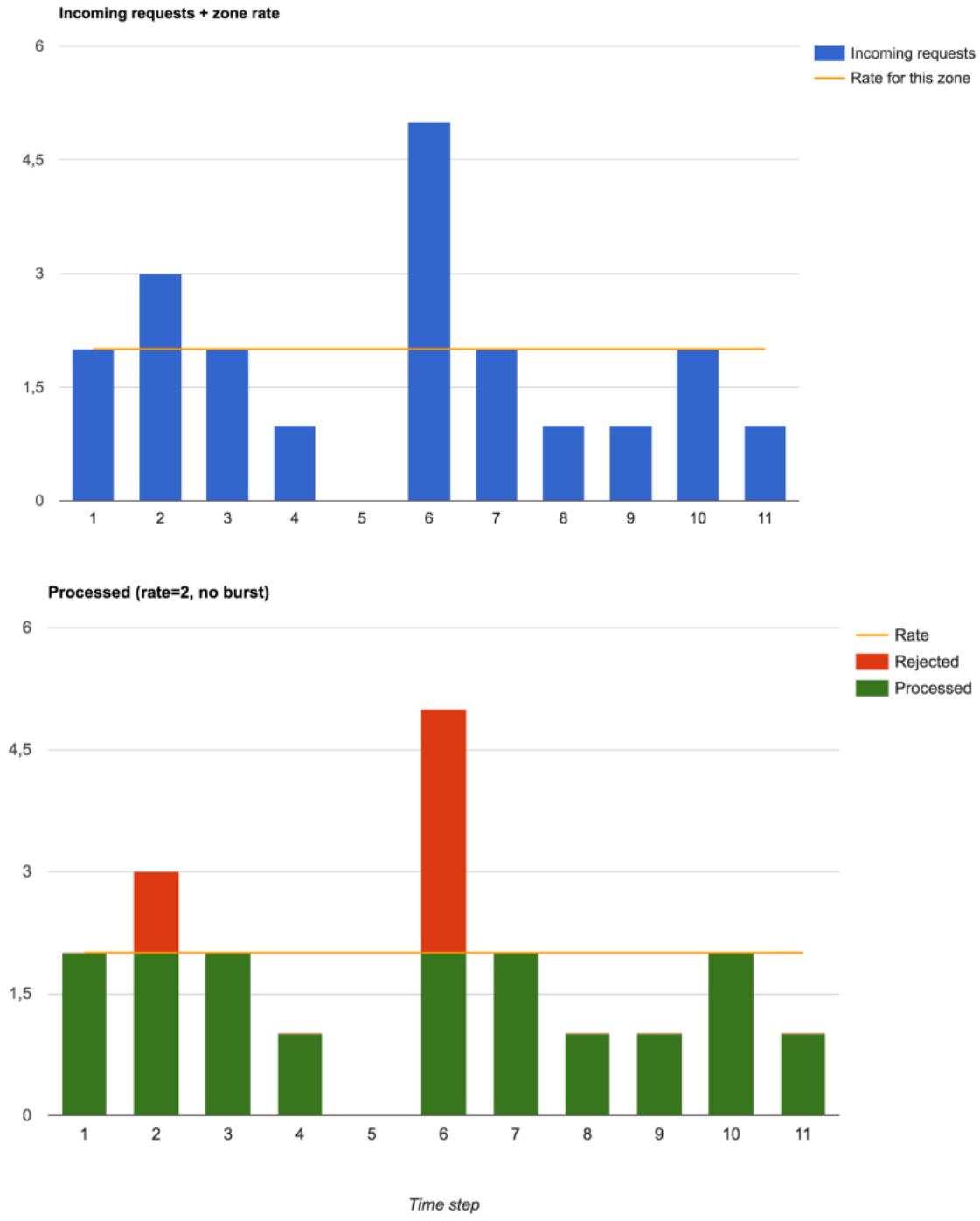
Note: in this case, the `zone` uses the `$request_uri` but all the following tests work exactly the same way for a `$binary_remote_addr` config which would rate-limit by client IP. You'll be able to play with this in the Docker image.

Let's recap

If we try to visualize how NGINX accepts the incoming requests, then processes them depending on the `rate`, `burst`, and `nodelay` parameter, here's a synthetic view.

To keep things simple, we'll show the number of incoming requests (then accepted or rejected, and processed) per time step, the value of the time step depending on the zone-defined rate limit. But the actual duration of that step doesn't matter in the end. What is meaningful is the number of requests NGINX has to process within each of these steps.

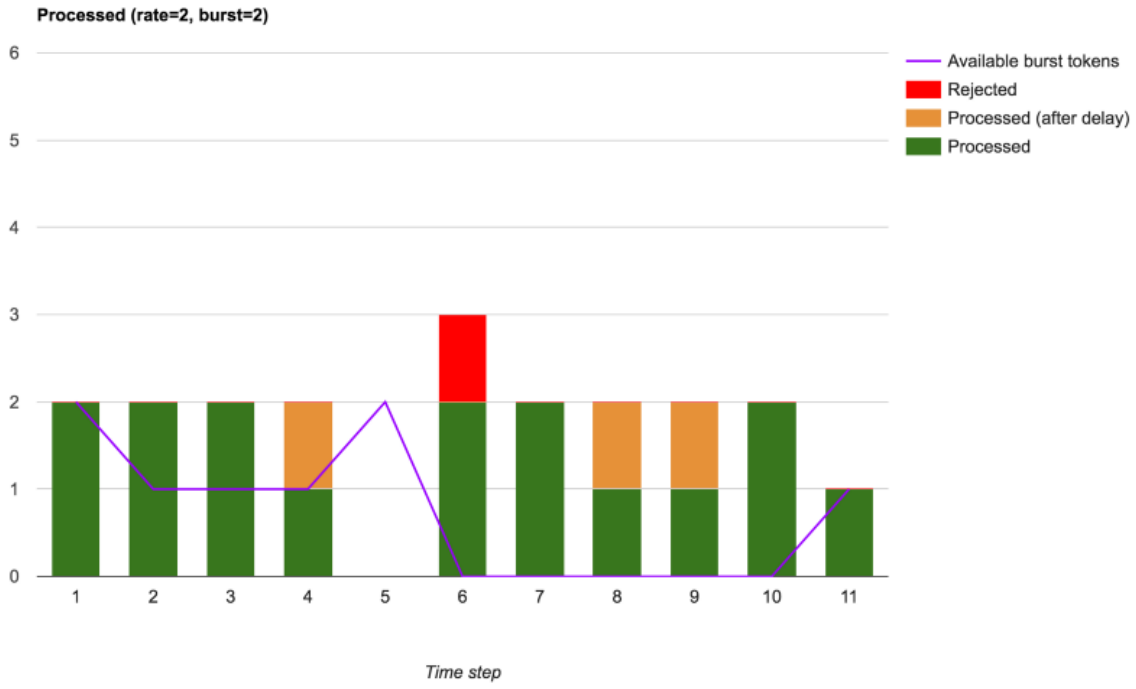
So here is the traffic we'll send through various rate limit settings:



Accepted and rejected requests when no burst setting is defined

Without using the burst (i.e. `burst=0`), we saw that NGINX acts as a pure rate-limit/traffic-policy actor. All requests are either immediately

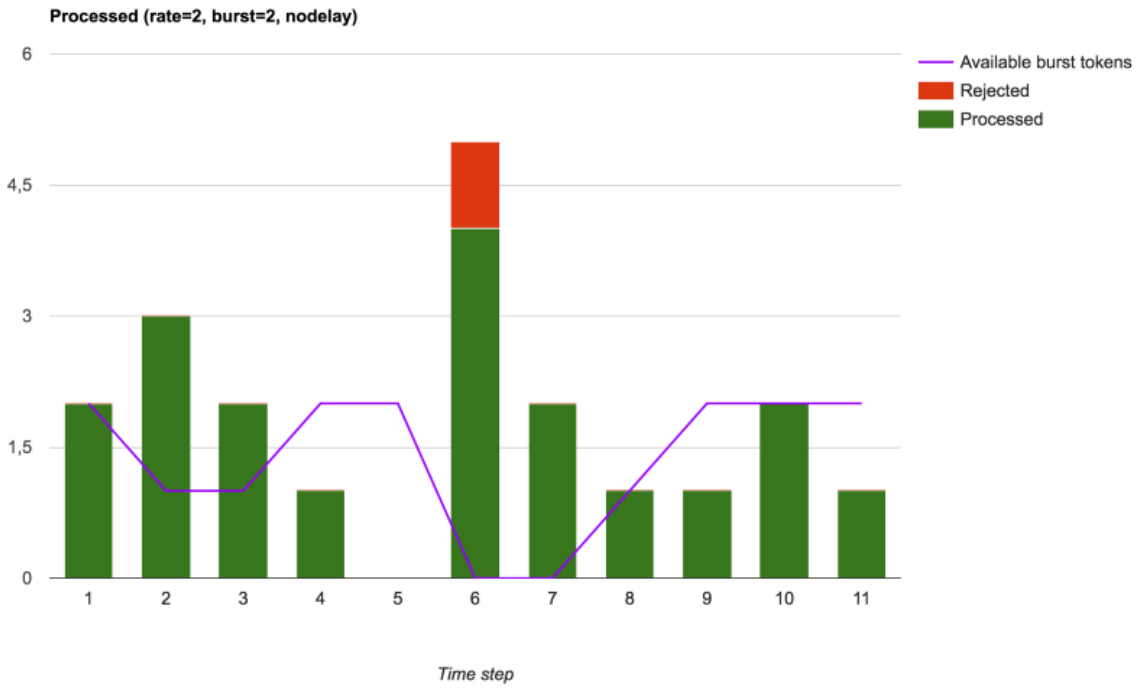
Now if we want to allow the small burst to use the unused capacity under the rate-limit, we saw that adding a `burst` argument lets use do that, which implies some additional delay in processing the requests consuming the burst tokens:



Accepted, accepted with delay and rejected requests when burst is used

We can see that the overall number of rejected requests is lower, and NGINX processes more requests. Only the extra requests when no burst tokens are available are rejected. In this setup, NGINX performs some real traffic-shaping.

Finally, we saw that NGINX can be used to either do some traffic-policy or to limit the size of the burst, but still propagates some of these bursts to the processing workers (upstreams or local), which, in the end, does generate less stable outgoing rate, but with a better



Accepted & processed requests and rejected requests when burst is used with nodelay

Playing with the rate limit sandbox yourself

Now you can go explore the code, clone the repo, play with the Docker image, and get your hands on it real quick to better solidify your understanding of these concepts.
https://github.com/sportebois/nginx-rate-limit-sandbox

Update (June 14th, 2017)

NGINX published a few days ago their own detailed explanation of their rate-limiting mechanism. You can now learn more about it in their Rate Limiting with NGINX and NGINX Plus blog post.