LineRate: Excessive HTTP 404 Throttling

Brian Talley, 2015-24-02

> Fusker thwarting using the LineRate Node.js datapath scripting engine

Fuskering is so fun to say, I couldn't resisting writing article about it. But, aside from just raising eyebrows when you use the term, fuskering is a real problem for some site maintainers. And having been in this position myself, I can verify that it's a difficult problem to solve. A flexible, programmable data-path, like the LineRate load balancer, makes light work of solving these kinds of problems.

Background

So, what exactly is fuskering? Simply stated, fuskering is requesting successive URL paths using a known pattern. For example, if you knew example.com had images stored at `http://example.com/img01.jpg` and `http://example.com/img02.jpg`. You might venture to guess that there is also an image at `http://example.com/img03.jpg`. And if I find that `img03.jpg` was there, I might as well try `img04.jpg`. Utilities, like curl, make automating this process extremely easy.

Photo sites are a typical target for fuskering because image filenames are usually pretty predictable. Think about a URL like `http://example.com/shard1/user/jane/springbreak14/DSCN5029.jpg` and you start to see where this could be a problem. Not only is this a potential privacy concern, but it's also a huge burden on the datacenter assets serving those files. In some multi-tier architectures, serving a 404 is actually **more** burdensome than serving an asset that exists. When you request something that doesn't exist, it's possible that all of the following could happen:

1. Cache miss on CDN, CDN requests from origin
2. Front end load balancer receives requests, makes balancing decision, forwards request
3. Web tier receives request, processes and sends to caching tier
4. Caching tier receives request and consults memory cache and then disk cache. Another cache miss
5. Services API tier receives request for URL to file system mapping

At this point, either your well written object storage API will signal to you that the file really doesn't exist or you're not using object storage and you have to actually make a request to the local disk or NAS. In either case, that's a lot of work just to find out that something doesn't exist.

Assets that *do* exist, end up in one of the caching tiers and are found and served much earlier in this process, typically right from CDN and the request never even touches your infrastructure.

If your site is handling a lot of requests and they are spread across many servers - and possibly many data centers - correlating all the data to try and mitigate this problem can be tedious and time consuming. You have to log each request, aggregate it somewhere, perform analytics and then take action. All the while, your infrastructure is suffering.

Options and limitations

Requiring authentication and filename scrambling are two ways to reduce the likelihood that your site will attract fuskers in the first place. Of course, these methods do not actually make fuskering impossible, but by requiring the user to enter identifying information or by making the filenames extremely difficult to guess, the potential consequences and level of effort become too great, and the user will likely move on.

The technique detailed in this article is just one of many ways to combat fuskers. Some other possible solutions are user-agent string checking, using CAPTCHA, using services like CloudFlare, traffic scrubbing facilities, etc. None of these is a silver bullet (or cheap, in some cases), but you could evaluate them all and figure out what works best for your environment and your wallet.

There are also ways for a determined user to subvert a lot of these protective measures: Tor, X-Forwarded-For spoofing, using multiple source IPs, and adaptive scripts to minimize the effect of the block time window (i.e. Send `max_404` requests, wait `time_window`, repeat), etc.

[One] Solution

Having a programmable data path makes solving (or at least mitigating) this issue easy. We can track each HTTP session, analyze the request and response, and have all the details we need to detect excessive 404's and throttle them. I provide a complete script to do this at the end; I'll describe the solution and how the script works next.

This article uses fuskering as motivation for this solution, but realize that the source of excessive 404's could come from a variety of sources, such as: people that automate data collection from your site (and forget to update request paths when they change), a misbehaving application, resources that moved without a proper 301/302 redirect, or a true 404 DoS attack (which is meant to exploit all the things I mention in the Background section), just to name a few.

Script overview

We're going to use the local LineRate Redis instance for tracking the 404 info. We're storing a key-value pair, where the key is the client's IP address and the value is the number of 404 responses that client has received in a configurable time window. This "time window" is handled by setting an expiration on the key-value pair and then extending it if necessary. If no 404's are detected during the grace period, the entry expires and the client is not subject to any request throttling.

When a new request is received, the client's IP is determined (see next section on Source IP) and checked against the Redis database. If a db entry is found and the corresponding value exceeds the allowed number of 404's, we intercept the request and respond directly from the load balancer with an HTTP 403.

On the response side, when we detect a 404 is being returned to a client, we increment the counter for the client IP in the Redis db. If the client's IP doesn't exist, we add it and init the value to '1'. In either case, the time window is also set.

The time window and the maximum number of 404's are configurable via the `config` object.

Source IP

To accurately analyze the 404's, you need to know the client's true source IP. Remember that any connection coming through a proxy is not going to have the actual client's source IP. Enter the X-Forwarded-For (XFF) header. If present, the XFF header will contain an ordered, comma-separated list of IP addresses. Each IP identifies another proxy, load balancer or forwarding device that the request passed through before it got to you. IPs are ***appended*** to this list, so the first IP is that of the actual client. In our script logic, we can check for the XFF header and if it's present, use the first IP in this list as the client IP. In the absence of a XFF header, we'll simply use the 'remoteAddress' of the connection object.

redis

There's a couple important things to point out in regards to using the included Redis server.

First, the LineRate load balancer runs multiple instances of the Node.js engine and variables are unique to that instance. If you were to store 404 tracking info in local variables, you might get results that you don't expect. See here for more info.

Second, using redis lends itself especially well to this example because you can run this script on all the virtual-servers for your site and get instant aggregated analysis and action.

The Script

If you're not already familiar with Node.js and the LineRate scripting engine, be sure to check out the LineRate Scripting Developer's Guide.

requires and config

Load the required modules and initialize the config object.

You might want to tune `config.time_window` and `config.max_404` to your environment. Continue reading to gain a better understanding of the implications of changing these values.

```
var vsm = require('lrs/virtualServerModule');
var async = require('async');
var redis = require('redis').createClient();

// Change config as needed.
var config = {
    vs: 'vs_http',     // name of virtual-server
    time_window: 10,  // window in seconds
    max_404: 10       // max 404's per time window
};
```

redis

Pretty basic stuff here, but note that we're loading the module and creating a client object all in one line.

```
var redis = require('redis').createClient();

redis.on('error', function (err) {
    console.log('Error' + err);
});

redis.on('ready', function () {
    console.log('Connected to redis');
});
```

onRequest() - async waterfall

The async module is used to provide some structure to the code and to ensure that things happen in the proper order.

When we receive a new request from a client, we get the client's IP, check it against the database and then handle the the rest of the request/response processes. Each of the functions are detailed next.

```
function onRequest(servReq, servResp, cliReq) {
    async.waterfall([
        function(callback) {
            get_client_ip(servReq, callback);
        },
        function(client_ip, callback) {
            check_client(client_ip, callback);
        },
        function(throttle, client_ip, callback) {
            doRequest(servResp, cliReq, throttle, client_ip, callback);
        },
    ], function (err, result) {
        if (err) {
            throw new Error(err); // blow up
        }
    });
}
```

get_client_ip()

Check for the presence of the XFF header. If present, get the client IP from the header value. If not, use `remoteAddress` from the servReq connection object.

```
function get_client_ip(servReq, callback) {
    var client_ip;

    // check xff header for client ip first
    if ('x-forwarded-for' in servReq.headers) {
        client_ip = servReq.headers['x-forwarded-for'].split(',').shift();
    }
    else {
        client_ip = servReq.connection.remoteAddress;
    }

    return callback(null, client_ip);
}
```

check_client()

check_client() is where we determine whether to block the request. If the client's IP is in redis and the corresponding value exceeds that of `config.max_404`, we set `throttle` to `true`. Else, `throttle` remains `false`.

`throttle` is used in the next function to either allow or block the request.

```
function check_client(client_ip, callback) {
    var throttle = false;
    redis.get(client_ip, function (err, reply) {
        if (err) {
            return callback(err);
        }

        if (reply >= config.max_404) {
            throttle = true;
        }

        return callback(null, throttle, client_ip);
    });
}
```

doRequest()

In `doRequest()`, the first thing we do is check to see if `throttle` is `true`. If it is, we simply return a 403 and close the connection. If you wanted more aggressive throttling, you could also update the expiration time of the redis key associated with this client's IP here.

If there is no throttle, we register a listener for the 'response' to the cliReq() and send the request on to the client. When we receive the response, we check the status code. If it's a 404, we increment the redis 404 counter.

For any client that requests more than `config.max_404` in a rolling window of `config.time_window` will start to get blocked. Once the time window passes, the requests will be allowed again.

```
function doRequest(servResp, cliReq, throttle, client_ip, callback) {

    if (throttle) {
        servResp.writeHead(403);
        servResp.end('404 throttle.  Your IP has been recorded.\n');

        // note you could choose to bump the redis key timeout here
        // and effectively lock out the user completely (even for good requests)
        // until they stop ALL requests for 'time_window'

        return callback(null, 'done');
    }
    else {
        cliReq.on('response', function(cliResp) {

            var status_code = cliResp.statusCode;

            if (status_code === 404) {
                redis.multi()
                    .incr(client_ip)
                    .expire(client_ip, config.time_window)
                    .exec(function (err, replies) {
                        if (err) {
                            return callback(err);
                        }
                    })
            }

            // Fastpipe response
            cliResp.bindHeaders(servResp);
            cliResp.fastPipe(servResp);
        });

        cliReq();

        return callback(null, 'done');
    }
}
```

Testing

This bash one-liner will use curl to send 15 consecutive requests for an image that doesn't exist and results in a 404 response. Note the change from '404' to '403' from request #10 to request #11. This is the throttling in action.

```
> for i in $(seq -w 1 15);do echo -n "${i}: `date` :: "; curl -w "%{http_code}\n" -o /dev/null -s htt
01: Fri Feb 13 09:59:44 MST 2015 :: 404
02: Fri Feb 13 09:59:44 MST 2015 :: 404
03: Fri Feb 13 09:59:44 MST 2015 :: 404
04: Fri Feb 13 09:59:44 MST 2015 :: 404
05: Fri Feb 13 09:59:44 MST 2015 :: 404
06: Fri Feb 13 09:59:44 MST 2015 :: 404
07: Fri Feb 13 09:59:44 MST 2015 :: 404
08: Fri Feb 13 09:59:44 MST 2015 :: 404
09: Fri Feb 13 09:59:44 MST 2015 :: 404
10: Fri Feb 13 09:59:44 MST 2015 :: 404
11: Fri Feb 13 09:59:44 MST 2015 :: 403
12: Fri Feb 13 09:59:44 MST 2015 :: 403
13: Fri Feb 13 09:59:44 MST 2015 :: 403
14: Fri Feb 13 09:59:44 MST 2015 :: 403
15: Fri Feb 13 09:59:44 MST 2015 :: 403
```

Pulling it all together, here's the full script. Happy cloning!

Please leave a comment or reach out to us with any questions or suggestions and if you're not a LineRate user yet, remember you can try it out for free.