# DOS Prevention for Node.js

Brandon Wang

## Abstract

The world of web development has been taken by storm by Node.js, an event driven server which takes advantage of event-driven programming for superior performance over traditional multithreaded/processed web servers. Its light weight, bare bones implementation comes at a cost, however. It does not include any functionality that well-established web servers like Apache. This includes protection against popular attacks such as DOS. In this project, a solution for preventing DOS over HTTP is presented through a suite of throttling classes.

## Problem Statement

Node.js implementations do not have a built in way of protecting against DOS.

## Introduction

For the last twenty years, two web servers have dominated the world of the web: Apache HTTPD and Microsoft IIS [1]. These two implementations have traditionally relied on two popular methods of concurrency to be able to serve large numbers of requests: multithreading and multiprocessing. Both of these approaches have drawbacks when serving large numbers of requests. For example, many servers have associated data stores which imply I/O. Many implementations which have I/O will block, preventing the thread from serving any other requests. This implies that in order to serve an additional request, we must spawn a new process or thread or wait for the blocking request to finish. This has considerable overhead which prevents efficient use of compute time.

One of the popular approaches to avoiding overhead associated with blocking implementations is the use of event-driven programming where each thread may serve many requests simultaneously. Example implementations include Ngnix, Tornado, and Node.js.

In addition, JavaScript is the de facto client side web scripting language and has been for some time. Code used for the client side, may very well be relevant for server side as well. For example, the data store is commonly abstracted away from its raw representation in the form of models. Models used on the server side can also be used on the client side. Therefore, if the same language were used on both client and server, some development overhead can be reduced. Node.js aims to fit this need and is widely gaining traction within the web development community.

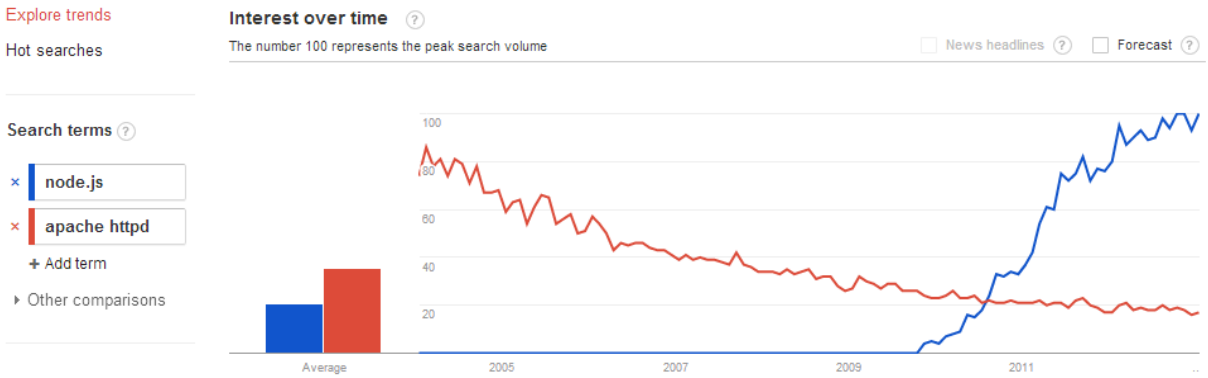


**Figure 1: Node.js vs. Apache HTTPD in Google Trends [2]**

Node.js is a layer between various I/O notification mechanisms such as epoll or select and the kernel. The kernel will notify Node.js when a request is being received and propagates those notifications to user code written in JavaScript. Since Node.js is so bare bones, many of the features found in more well established web server implementations are not found. Amongst them include DOS prevention. Within production, there are many schemes which aim at solving this problem.

Arguably, the most common implementation involves using the web server Ngnix as a load balancer/reverse proxy which acts as a frontend to requests, propagating them to one or more Node.js instances. Ngnix has facilities for preventing DOS. However, this does not come without a cost. Each additional component within a system implies a multitude of additional complexity. With Ngnix, for example, there may be additional complexity with respect to dependency compatibility, deployment, debugging, monitoring, logging, and so on. One of the great value propositions that Node.js is extremely simple development and the use of Ngnix takes away from that value.

## Prior work

Nginx's primary method of flood protection is offered through the module HttpReqLimitModule [3]. HttpReqLimitModule allows for throttling of sessions or individual IPs. Karl Blessing, a software consultant set up a test environment using Nginx with HttpReqLimitModule. In his test, using the load testing service, Blitz.IO, Blessing was able to provide a response time that did not scale with the number of concurrent users. It is of note that traffic from Blitz.IO came from a single IP and therefore is not similar to traffic of an actual DDOS attack.

Lloyd Hilaiel, an engineer at Mozilla, set up a test environment using Node.js set up that consumes 5 ms of processor time in 5 asynchronous calls. The following was observed by Hilaiel [4]:

- At 6x capacity (1200 req/s), the average latency is 40 ms.
- A majority of these requests are failing.

Putting the above together, Hilaiel notes that after a long period of waiting, users will be greeted by a cryptic TCP failure.

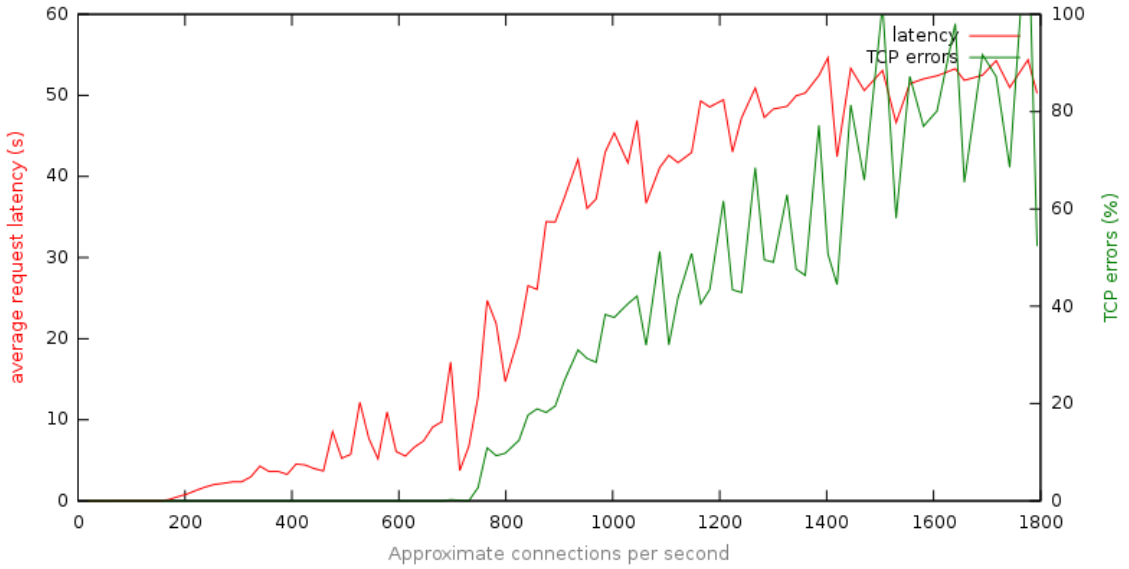


**Figure 2: TCP Errors and Latency vs. # of Connections/second [4]**

The solution Hilail proposed was to track how fast the internal event loop within Node.js would run, and if it crossed some threshold, (ie. the Node.js instance was running slower than some predefined rate) the Node.js instance would simply fail fast and return the HTTP code 503. This solution is simple, but far from optimal. It does not allow the server to throttle or differentiate traffic by IP, nor does it take into account that different requests may represent different loads on the server. For example, it may only be a handful of IPs that are attacking the server, or an attack may be targeting specific operations which result in particularly expensive operations. Finally, a programmer may know ahead of time that some endpoints are more important than others. For example, it may be more important that a user is able to use basic functionality and not as important to use functionality such as an HTTP search API. In this way, a programmer may specify how a web application may degrade gracefully.

There are a plethora of other solutions involving iptables, hardware firewalls, packet filtering and so on, which handle a plethora of lower level protocols. However, since this project aims to remove a dependency on Ngnix, HTTP traffic is focused on.

## Solution

One of the assumptions this project makes is that, as is with the case of DDOS, it is often not possible to differentiate legitimate requests from those that may be malicious but we would like to take as many steps as possible to do so.

In this project, the following JavaScript classes are publicly exposed:

### Throttler(rate)

This class is a simple base class with one method that is used to determine if something is being used more often then rate. It can be used for a multitude of cases and is used internally to implement more sophisticated throttle classes.

It has one method:
**throttle(rate)**
In order to reduce API surface, throttle serves multiple purposes. At a high level, it simply returns 0 if the operation should be permitted. If the operation should not be permitted, it returns the amount of time that must elapse until the next operation is allowed. The intention is to be called before an operation is performed.

It takes in an argument rate, which does not have to be specified. It is simply syntactic sugar in the case that the user wants to change the rate before determining how much time must elapse before the next request.

An example use case is as follow to impose a global throttle rate (ie. what Hilail proposes):

```
server = http.createServer(function(req, res) {
    var t = globalThrottler.throttle();

    if (t != 0) {
        res.writeHead(503, {"Content-Type": "text/plain"});
        res.write("Service throttled.");
        res.end();
        return;
    }

    res.writeHead(200, {"Content-Type": "text/plain"});
    res.write("Hello World");
    res.end();
});
```

## IPThrottler(rate, numConn)

This class is used to impose two restrictions on a per IP basis. First, it imposes a request rate specified by the developer as rate. Second, it imposes a hard limit on the number of concurrent connections someone may have.

It has two methods:
**throttle(req)**
throttle(req) requires a request object to determine what IP a request is being made from.

It returns -1 if the number of connections is exceeded. In the case that it is not, its behavior is similar to that of Throttler.throttle() in that it will return 0 if the request should be allowed. Otherwise, it will return the amount of time that must elapse before the next request may be made.

**markResponseEnd(req)**
Used to mark when a request has been served. This must be called or else IPThrottler will believe that the request is still alive.

## AdaptiveThrottler(infos)

AdaptiveThrottler is used to adaptively throttle requests according to user specified priorities, their performance characteristics and RegExp which is used to differentiate request characteristics. For example, say we have two endpoints, /search?q=myquery, and /main.

The RegExp patterns that might be used might be something like "^\/search.*" and "^\/main$". In addition, we might know that /main is more important for users than /search is. So we might propose priorities 1 for /main, and 10 for /search. Both the pattern and priority is encapsulated in the class EndpointInfo and is discussed later.

It has two methods:

**throttle(req)**

AdaptiveThrottler.throttle() behavior is similar to that of Throttler.throttle() in that it will return 0 if the request should be allowed. Otherwise, it will return the amount of time that must elapse before the next request may be made.

**markResponseEnd(req)**

Used to mark when a request has been served. This must be called or else AdaptiveThrottler will believe that the request is still alive.

### Principles of Operation

As far as the author is aware of, there are products that allow developers to specify hard throttling rates on a per endpoint basis, but there isn't anything that allows a developer to specify relative priorities of endpoints. Therefore, a first principles approach was taken to derive the principles of operation.

From a high level, we would like to define a variable, L, which represents an abstract quantity: the load that the server can take. Furthermore, we define a portion of L denoted as $l_i$ for the i-th endpoint such that:

$$L = \sum_{i=0}^{n} l_i$$

Where n is the number of endpoints.

Since L is an abstract quantity, we may define it however we would like. Factors we may want to consider are CPU time, memory usage, I/O time, and so on. In fact, L need not be a scalar, but a vector quantity. However, for the purposes of this project, I have defined each $l_i$ as:

$$l_i = r_i t_i$$

Where $r_i$ and $t_i$ is the request rate and the average time required to serve each request for the i-th endpoint.

We would also like to define another variable, $\alpha_i$ which represents the portion of $L_i$ that the i-th endpoint consumes. That is:

$$\alpha_i L = r_i t_i$$

$$\text{Then, } r_i = \frac{\alpha_i L}{t_i} \quad (1)$$

$r_i$ is the one degree of freedom we can restrict/impose on a per endpoint basis which makes this a key equation.

We must now relate $\alpha_i$ to $p_i$ which is the priority of the i-th endpoint. Ideally, we would like to have a $p_i$ value of 1 to consume 10x the percentage of L that a $p_i$ value of 10. One simple relation that satisfies the above is:

$$\beta_i = \frac{1}{p_i}$$

$$\text{Then, } \alpha_i = \frac{\beta_i}{\sum_{j=0}^{n} \beta_j} = \frac{1}{p_i} \cdot \frac{1}{\sum_{j=0}^{n} \frac{1}{p_j}} \quad (2)$$

With (1), and (2), we have an expression for $r_i$ dependent on L, $p_i$, and $t_i$. We may now change L according to any metric we would like and all endpoints will throttle proportionately based on what rate we set.

For the purposes of this project, I've defined L' (ie. the new L) as:

$$L' = \max(L \cdot (1 - \frac{t_r - t_t}{t_t}), 1)$$

Where $t_r$ is the global average response time and $t_t$ is the target response time which may be user defined. If the server is slower than our target response time, we decrease L and thus, the amount of load each endpoint may take. Conversely, if the server is faster than our target response time, we increase L and each endpoint may take more requests. In order to prevent L going to infinity, a cap is imposed.

It is of note, that average response times are not absolute response times, but averages over a predefined number of requests (ie. average response times over the last 100 requests).

### EndpointInfo(path, priority)

EndpointInfo is used in conjunction with AdaptiveThrottler. Path is a RegExp that allows someone to express an endpoint or a collection of endpoints (ie. the pattern "(\/foo|\/bar)" will match both /foo and /bar). Priority is a number which specifies relative importance compared to other endpoints.

## Verification

All test code may be found in Appendix A.

### Global IP Throttling Test

This test is specifically for the Throttle class.

- The test initializes an instance of Throttle with a rate of 1 req/s.
- It will make two immediate requests, the first one should be received properly, and the second should be throttled.
- The test will wait two seconds before making another request, which should not be throttled.

The following is the output of the test case:

```
Creating server with throttle rate of 1 req/second.
Server listening.
Requesting localhost.
OK. Requesting localhost, should be throttled.
OK. Waiting 2 seconds then requesting localhost.
OK. Test passed.
```

### IP Concurrent Connection Throttle Test

This test is specifically for the IPThrottler, specifically in the case where many fast requests are permitted but only one connection can be made at a time.

- The test will initialize an IPThrottler with a request rate of 12 req/s and 1 concurrent connection (ie. requests must be serial).

- The test will make 10 requests serially very quickly, none of the requests should be throttled.
- The test will wait a second, then make two concurrent requests. One should be responded to properly, the other should get HTTP code 503.

The following is the output of the test case:

```
Creating server with a per ip throttle rate of 12 req/second with 1 connection allowed.
Server listening.
Requesting localhost serially 10 times.
Requesting.
OK. Request 1 returned correctly.
Requesting.
OK. Request 2 returned correctly.
Requesting.
OK. Request 3 returned correctly.
Requesting.
OK. Request 4 returned correctly.
Requesting.
OK. Request 5 returned correctly.
Requesting.
OK. Request 6 returned correctly.
Requesting.
OK. Request 7 returned correctly.
Requesting.
OK. Request 8 returned correctly.
Requesting.
OK. Request 9 returned correctly.
Requesting.
OK. Request 10 returned correctly.
Cooldown 1s to allow for throttling to settle.
Requesting two concurrent longstanding requests. One request should be throttled.
Recieved 503
Recieved 200
OK. One request throttled. Test passed.
```

## IP Throttle Test

This test is specifically for the IPThrottler, specifically in the case where one IP should be throttled.

- An instance of IPThrottler will be initialized
- The test will make one request which should be returned fine and ensure that that particular IP is being tracked.
- The test will then wait for a time interval dependent on the throttle rate (6000/rate) to ensure that that IP is no longer being tracked (or else this process would eventually consume too much memory).
- The test will then make two fast requests serially, the second one should be throttled.

The following is the output of the test case:

```
Creating server with a per ip throttle rate of 1 req/second with 10 concurrent connections
allowed.
Server listening.
Requesting localhost.
OK. Checking number of IPs checked up on.
OK. Waiting for IP info to expire.
Checking number of IPs checked up on, should be 0 since the server should forget about 127.0.0.1.
OK. Sending two fast running requests to get this IP throttled.
200
OK. First request responeded to properly, sending second request.
OK. Second request throttled. Test passed.
```

### Endpoint Throttle Test

This test does not have a binary pass or fail per-say since endpoint throttling has probabilistic characteristics to it. It is used to see the how the server throttles different endpoints with different parameters.

- The test will create n endpoints with a response time greater than the target response time such that the server will detect that the average response time is greater than the target response time.
- The test will request each endpoint a number of times.
- The test will report for each endpoint and priority the percentage of requests that have been throttled.

With 10 endpoints, a target response time of 10 ms, and response time 10% (ie. the server responds in 11 ms to simulate computational load) more than the target response time, we get the following results:

```
Endpoint 0 with priority 1 with proportion 100%
Endpoint 1 with priority 2 with proportion 100%
Endpoint 2 with priority 3 with proportion 100%
Endpoint 3 with priority 4 with proportion 98.6%
Endpoint 4 with priority 5 with proportion 95%
Endpoint 5 with priority 6 with proportion 85%
Endpoint 6 with priority 7 with proportion 75.4%
Endpoint 7 with priority 8 with proportion 66.2%
Endpoint 8 with priority 9 with proportion 56%
Endpoint 9 with priority 10 with proportion 3.2%
```

With 5 endpoints, a target response time of 10 ms, a response time of 10% more than the target response time, and higher priorities, we get the following results:

```
Endpoint 0 with priority 1 with proportion 100%
Endpoint 1 with priority 4 with proportion 100%
Endpoint 2 with priority 9 with proportion 99.7%
Endpoint 3 with priority 16 with proportion 99.8%
Endpoint 4 with priority 25 with proportion 0.8%
```

## Conclusions and Potential Impact

This project has the potential to reduce dependency on Ngnix as a reverse proxy for Node.js, allowing for Node.js to become an independent web server used without any dependencies. This allows for simpler deployment, debugging, logging, and development of web applications using Node.js.

In addition, this novel approach to graceful degradation of web applications may have applications in a plethora of other web servers and web frameworks where heavy load in one area of the web application does not necessarily mean the entire web application is inaccessible. As a corollary, this also allows for applications to deploy new features, which may not have been profiled with respect to real world behavior, without fear that they will bring down the entire web application.

This project is to be released to the public under a MIT license, prepared as a Node.js package and hosted on GitHub so other developers may use, improve, and extend the project.

## Bibliography


[1] Netcraft. (2013, Jan.) Internet Research, Anti-Phishing and PCI Security Services. [Online]. http://news.netcraft.com/archives/2012/07/03/july-2012-web-server-survey.html

[2] Google. (2013) Web Search Interest: node.js, apache httpd. Worldwide, 2004 - present. [Online]. http://www.google.com/trends/explore?hl=en#q=node.js%2C%20apache%20httpd&cmpt=q

[3] Karl Blessing. KBreezie. [Online]. http://kbeezie.com/nginx-protection/

[4] Lloyd Hilaiel. (2013, January) Mozilla Hacks. [Online]. https://hacks.mozilla.org/2013/01/building-a-node-js-server-that-wont-melt-a-node-js-holiday-season-part-5/