

|  |  |  |
| --- | --- | --- |
| Hanoi University of Science and Technology |  | School of Information and Communication Technology |

GRADUATION RESEARCH 3

Topic: Towards improvements for SpeakNow - a WebRTC based video chat system for anonymous peer practice on IELTS speaking test

Advisor: Dr. CAO TUAN DUNG

Student: TRAN TAN LINH (20132317)

Hanoi, Summer 2018

Table of Contents

1. Introduction
2. Benchmarking methodology
   1. Tools and softwares
   2. Load testing HTTPS APIs
   3. Load testing WebSocket APIs
   4. Examining WebRTC stats
3. Pre-optimization benchmarks
   1. Route xseed.tech/ieltsTest
   2. Route xseed.tech/reception
4. Analysis & optimizations
   1. WebRTC analysis
   2. HTTPS API optimizations
5. Post-optimization benchmarks
   1. Route xseed.tech/ieltsTest
   2. Route xseed.tech/reception
6. Summary & referrences

# 

# 1. Introduction

In my previous graduation researches, I have worked on building a functional webRTC based application for anonymous peer practice on IELTS speaking test. The quality of service of the application is composed of:

* The performance of the web server in serving the application’s client code (HTML/JS), serving IELTS test contents.
* The performance of the WebSocket server in serving the real-time chat board.
* The performance of the WebRTC video call:
  + Clients’ network conditions
  + How well is video data encoded in webRTC

In this research, I would like to dive deeper on improving the quality of service of the whole system where possible.

Firstly, I will benchmark the performance of the current system. Secondly, possible changes proposed in this research will be applied. Finally the system will undergo another benchmark and conclusions can be drawn.

The web server (HTTP) and WebSocket server will be mainly load tested against different test cases simulating real users and used as a measure for improvements.

The behaviour and performance WebRTC modules in popular browser (Google Chrome) will be evaluated. While it is a tough, time-consuming task to explore the source code of browsers to carry out experiments and apply improvements directly, some black-box analyses and benchmarking methods can still be applied to propose possible improvements.

# 2. Benchmarking methodology

**2.1 Tools and softwares:**

**2.1.1 Loader.io**

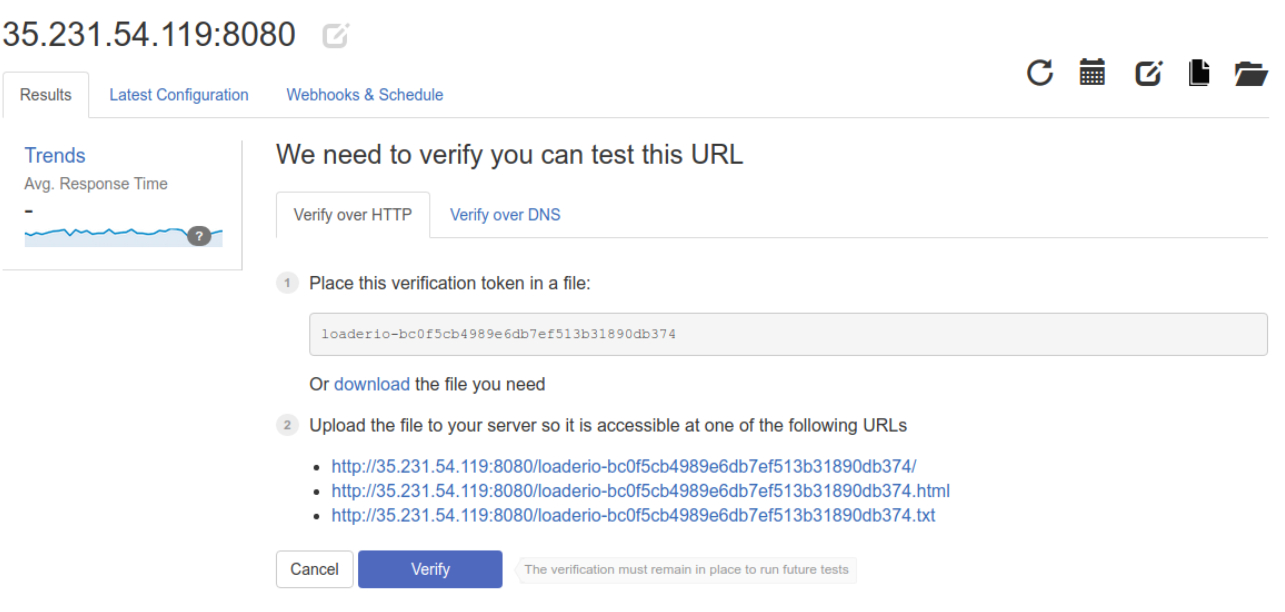
Loader.io is a popular cloud-based load testing service by SendGrid Lab that allows users to stress test HTTP APIs with thousands of concurrent connections. Loader.io uses multiple servers from different places around the world with different networking resources, computing resources, etc. that helps mimicking the dynamics of real-world trafic much better.

Advantages:

* Integration with PaaS providers, continuous integration tools, and browsers
* Allows testing up to 10,000 concurrent connections for free
* Allows testing up to 1 min for free.
* It is cloud-based so it is a no install solution and immediately available for developers to test.

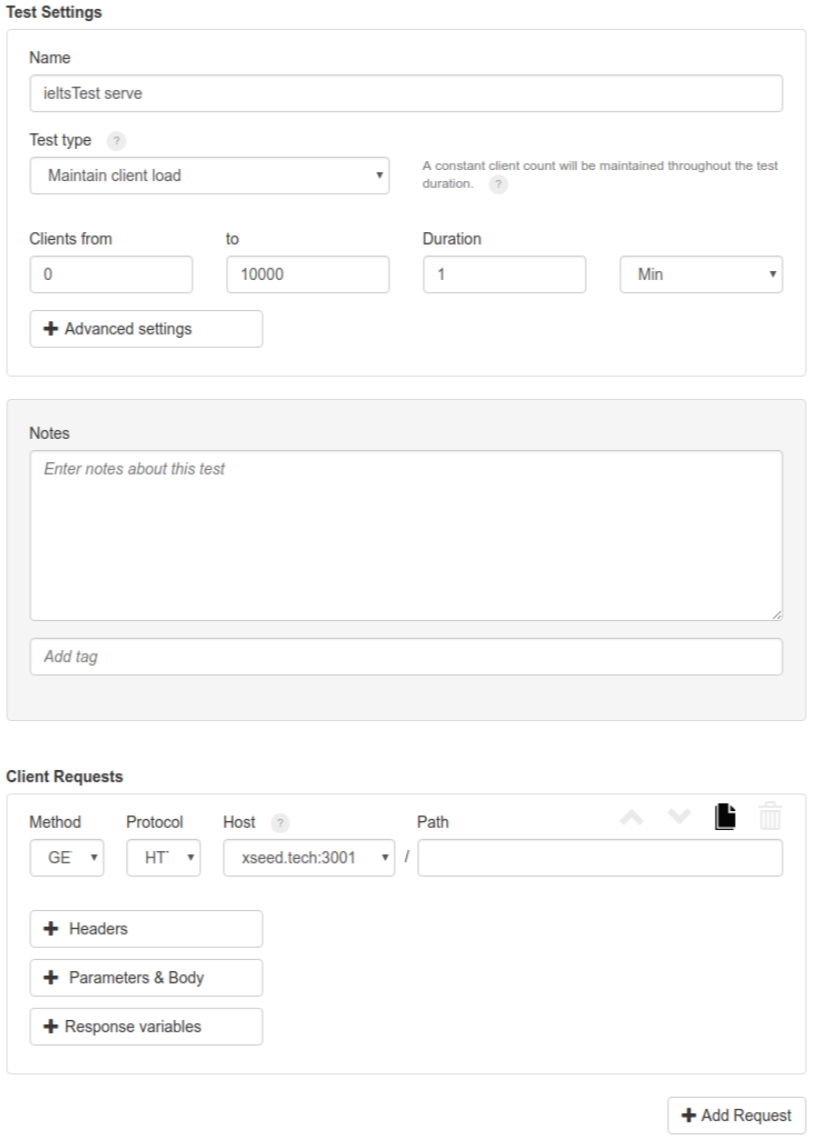
In order to setup a test with loader.io, we first need to verify ownership of servers to prevent exploitation for a DDoS attacks.

The verification is done by placing a token issued by loader.io and serving it at our web server through out the test.



After successful verification, we can now create a test with various options:

* Type of tests, test duration
* Add HTTP(S) requests with differrent methods and paths
* Request headers, parameters & body, response variables



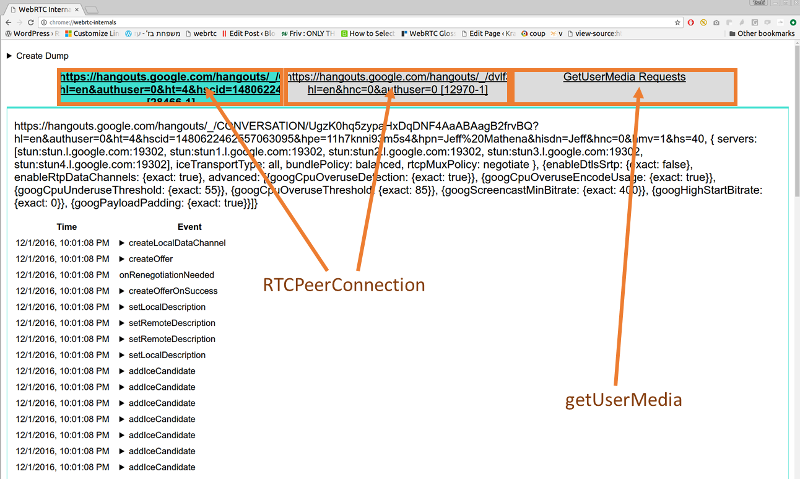
Referrences: <https://en.wikipedia.org/wiki/Loader.io>

**2.1.2 WebRTC Internal**

An internal module of Google Chrome for collecting raw data and statistics about WebRTC connections.

* Can be accesed at “[chrome://webrtc-internals](https://testrtc.com/webrtc-internals-documentation/)” on a normal Google Chrome tab.
* Allows exporting data to JSON format
* Include graphs for data visualisation, however the graph is small and cannot be expanded.
* The exported data can be visualised much better using WebRTC Dump Importer <https://fippo.github.io/webrtc-dump-importer/>

The upper tab of WebRTC Internal allows choosing an entity, be it a RTCPeerConnection or getUserMedia requests:



For each RTCPeerConnection, we can access four data components as shown in the image below:

1. How the RTCPeerConnection was configured, i.e. what STUN and TURN servers are used and what options are set
2. A trace of the PeerConnection API calls on the left side. These API traces show all the calls to the RTCPeerConnection object and their arguments (e.g. createOffer) as well as the callbacks and event emitters like onicecandidate.
3. The statistics gathered from the getStats() API on the right side
4. Graphs generated from the getStats() API at the bottom



**2.2 Load testing HTTPS APIs:**

The following three API endpoints will be load tested independantly, one after the other:

“<https://xseed.tech/reception>” - serving the entire application code

“[https://xseed.tech/ieltsTest](http://xseed.tech/ieltsTest) ” - serving the content of the Ielts tests

“[wss://xseed.tech/](http://xseed.tech)” - serving the synchronized whiteboards between users

Two types of load test will be applied:

* Client per seconds: A particular number of concurrent clients will be maintained through out the test duration
* Maintainance client load: The number of concurrent clients rise gradually from a starting value, reaching a ceiling value at the end of the test duration

For a more detailed and visual explaination, please visit: <https://support.loader.io/article/16-test-types>

Test results provide the a graph of number of concurrent clients and average response time plotted against time. Example is below.



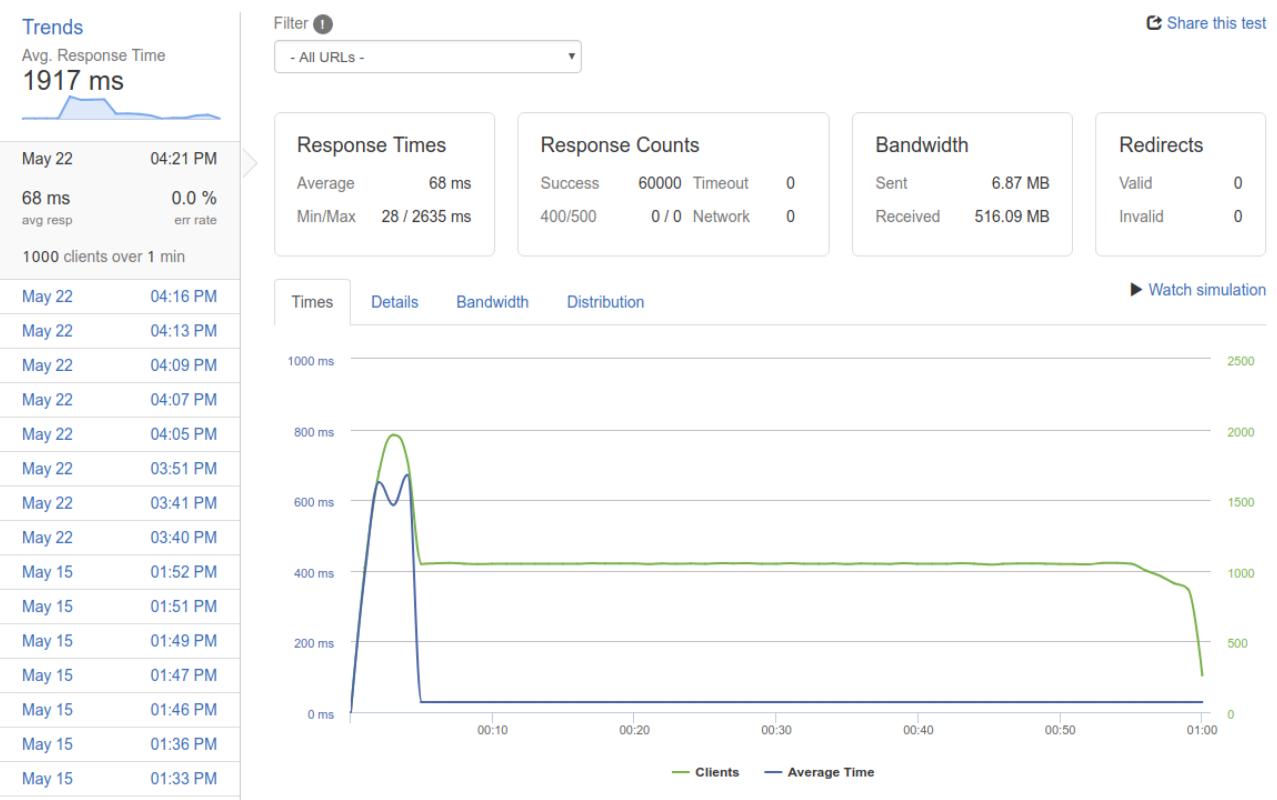
# 3. Pre-optimization benchmark:

The benchmarks are performed on a machine on Google Cloud Platform with 4 virtual CPUs, 3.6Gb of RAM.

**3.1 Route xseed.tech/ieltsTest:**

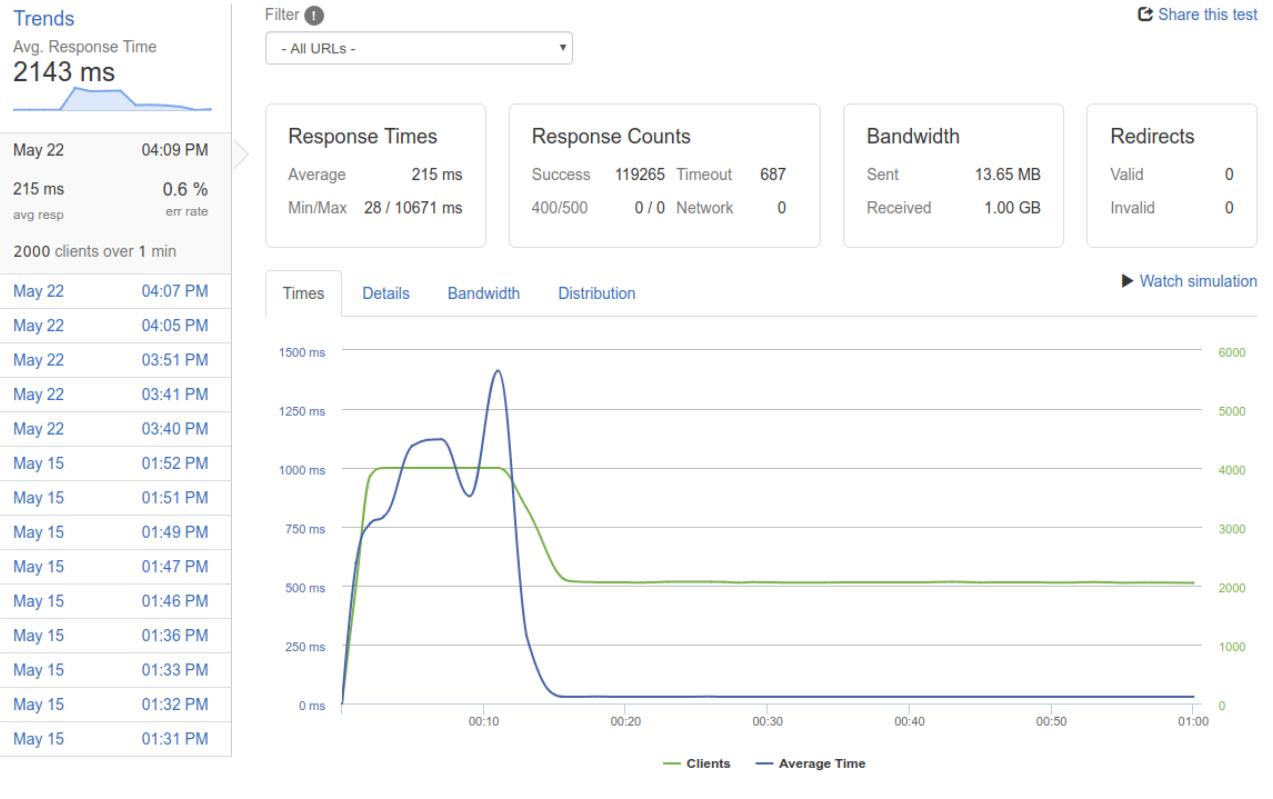
Requests spanning over 10 seconds are considered errors.

**3.1.1 1000 concurrent clients per seconds**

****

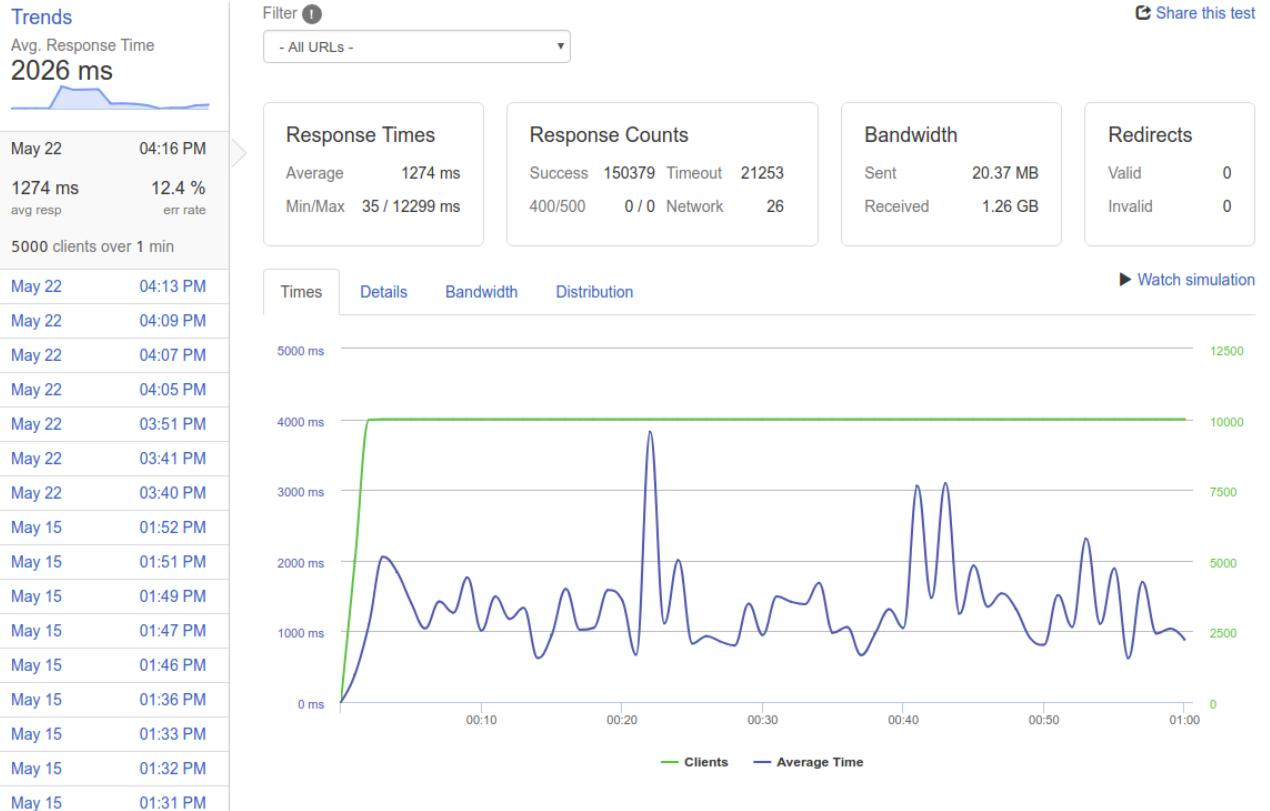
* Average/Min/Max response time: 68/28/2635 ms
* Error rate: 0%

**3.1.2 2000 concurrent clients per seconds**

****

* Average/Min/Max response time: 215/28/10671 ms
* Error rate: 0.6%

**3.1.3 5000 concurrent clients per seconds**

****

* Average/Min/Max response time: 1274/35/12299 ms
* Error rate: 12.4%

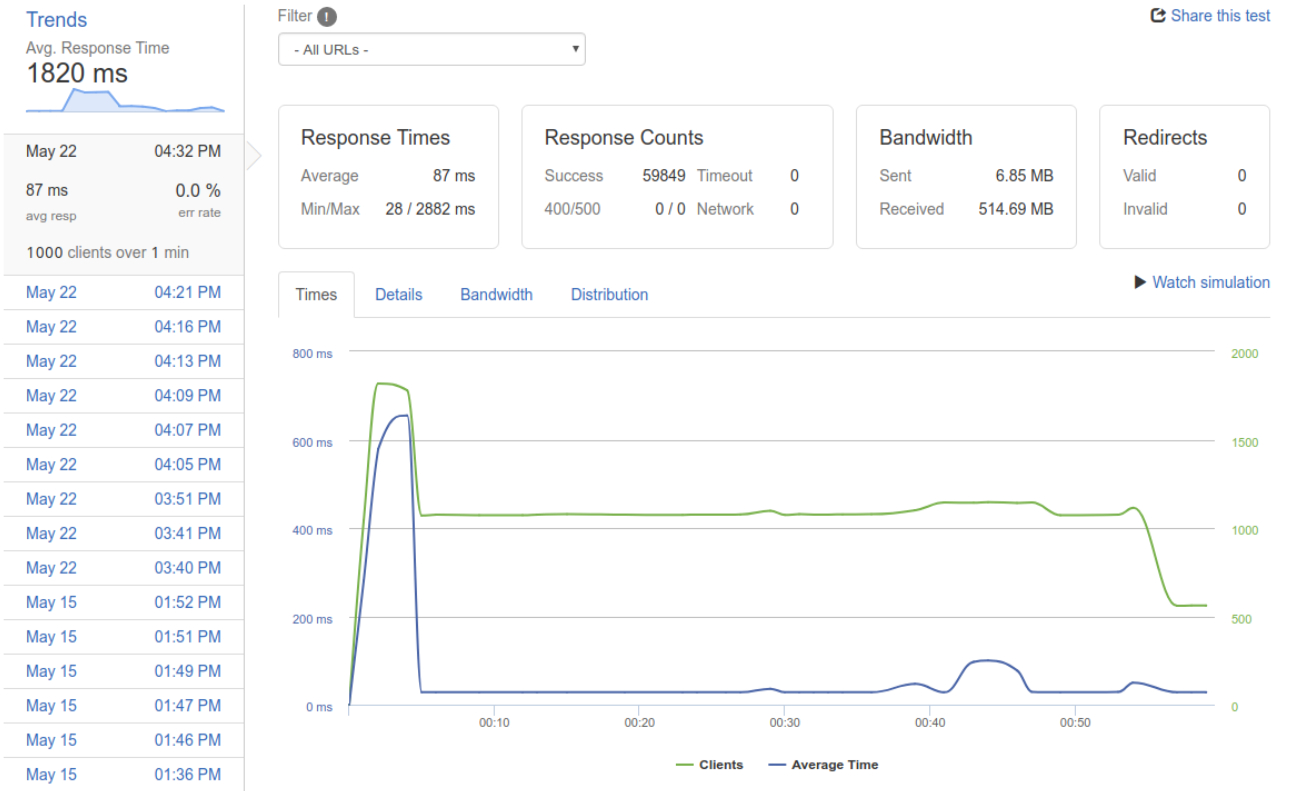
**3.1.4 From 0 to 10000 clients in 1 minute:**

****

* Average/Min/Max response time: 988/28/10972 ms
* Error rate: 0.1%

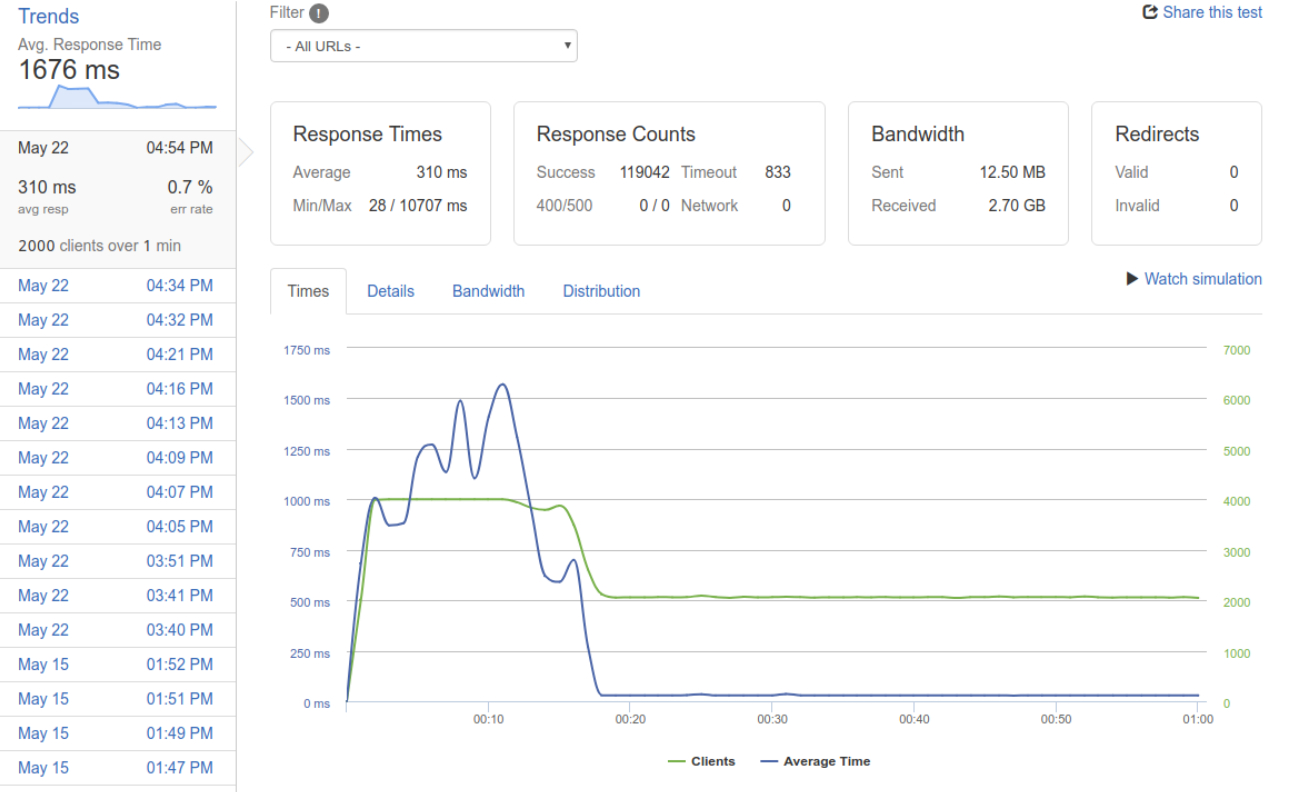
**3.2 Route xseed.tech/reception:**

Requests spanning over 10 seconds are considered errors.

**3.2.1 1000 clients per second:**

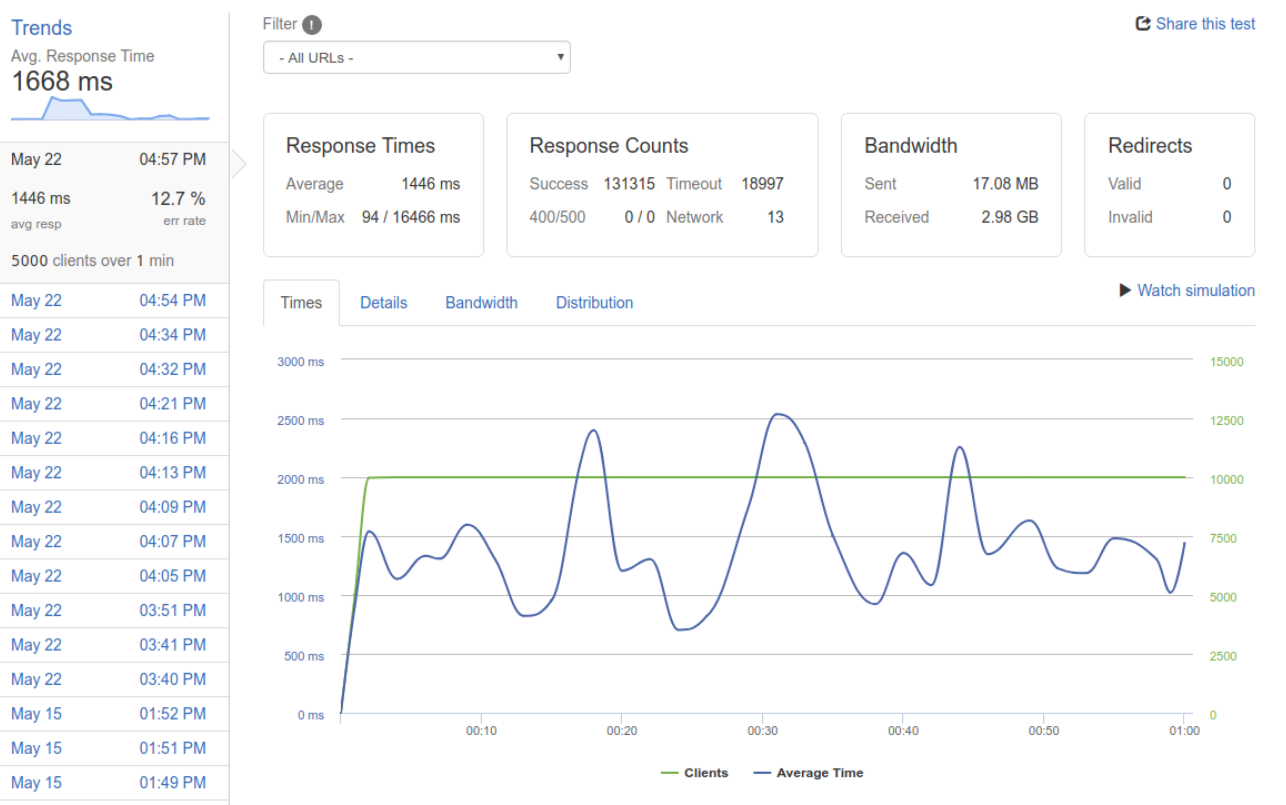
* Average/Min/Max response time: 87/28/2882 ms
* Error rate: 0%

**3.2.2 2000 clients per second:**

****

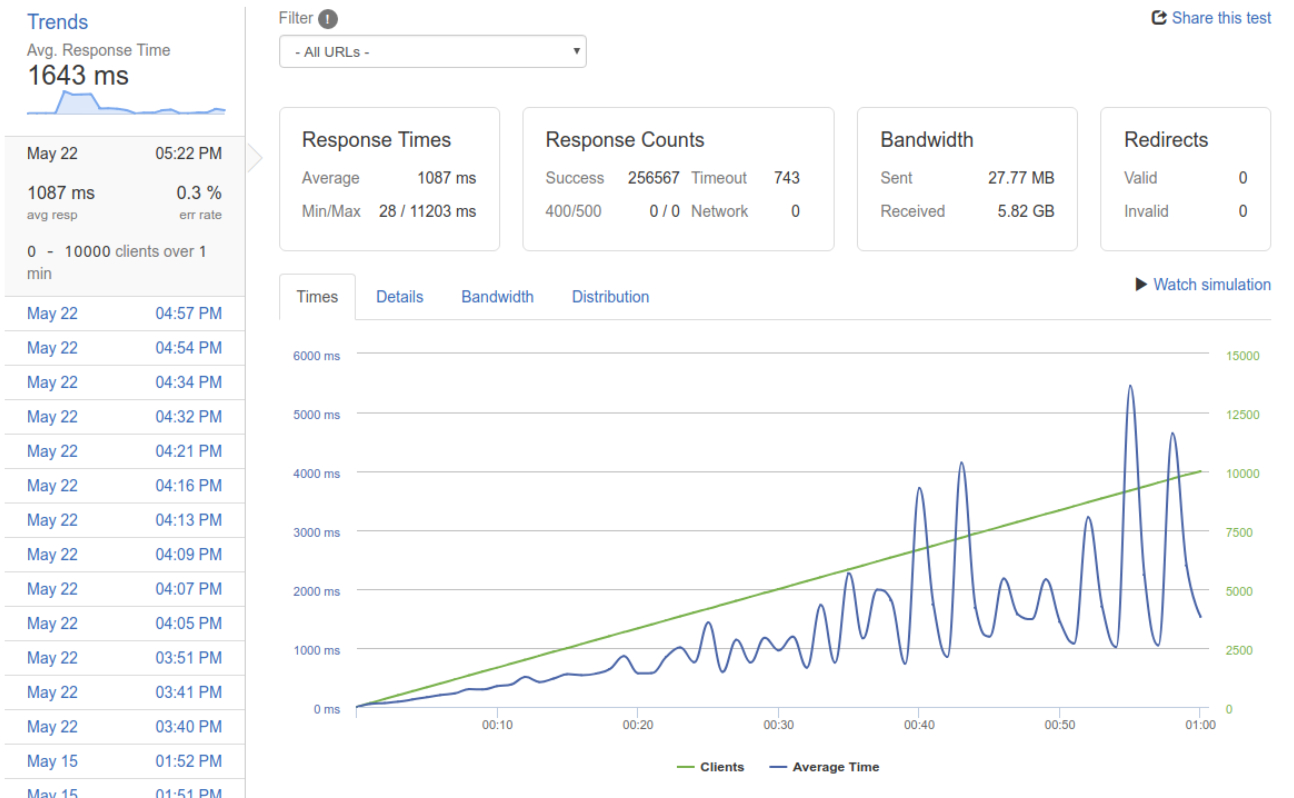
* Average/Min/Max response time: 310/28/10707 ms
* Error rate: 0.7%

**3.2.3 5000 clients per second:**

****

* Average/Min/Max response time: 1446/94/16466 ms
* Error rate: 12.7%

**3.2.4 From 0 to 10000 clients in 1 minute:**



* Average/Min/Max response time: 1087/28/11203 ms
* Error rate: 0.3%

# 4. Analysis and optimization:

**4.1 WebRTC analysis:**

**4.1.1 Bandwidth drop on moving mobile device:**

WebRTC is meant for global compatibility for real-time data transportation, so use cases related to mobile devices should be considered important.

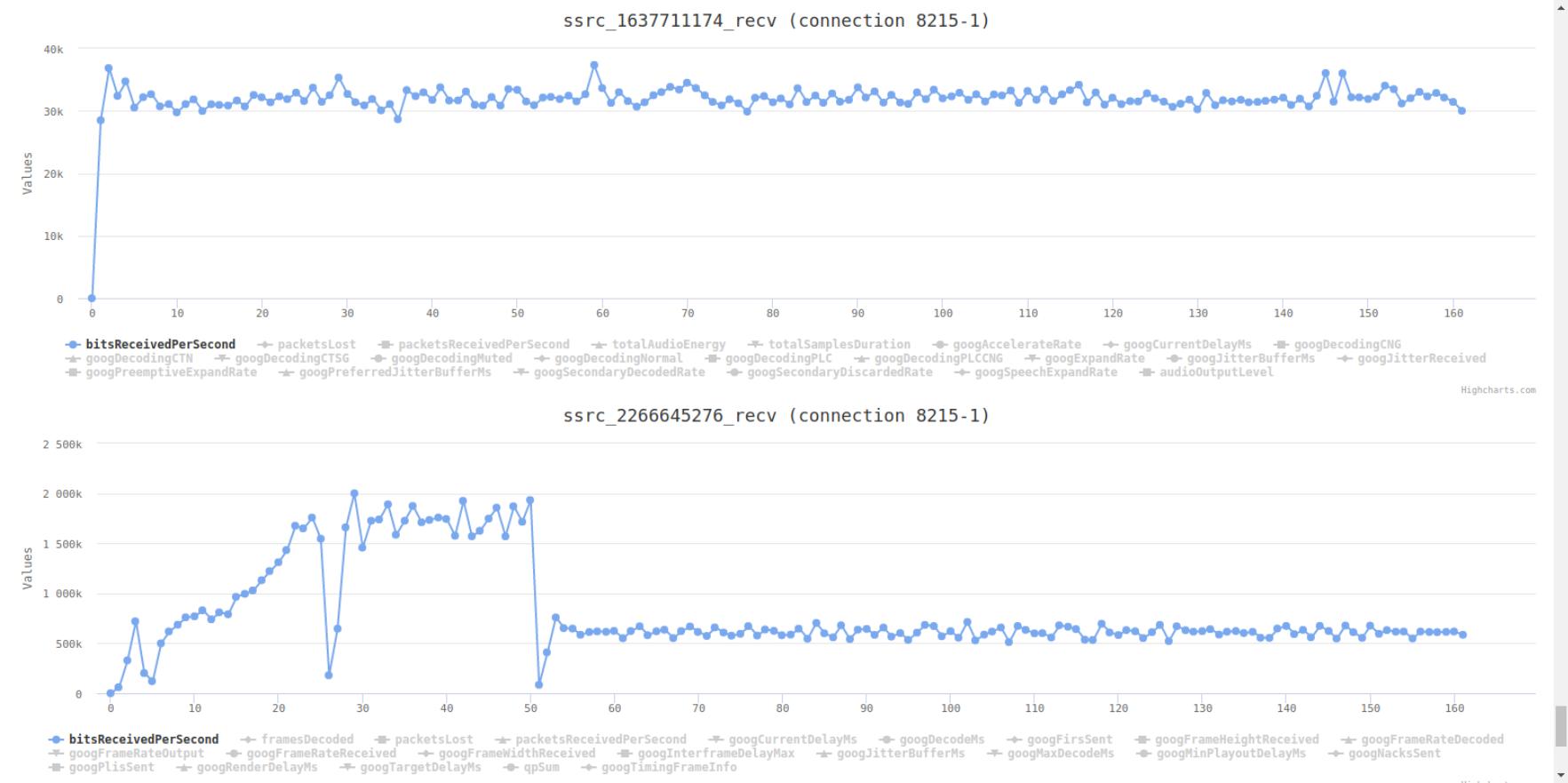
Now imagine a video call happening when 1, or even 2 parties are moving fast (e.g: on a bus, train). The bandwidth will drops and depending on the severity of the event the system should behave in a way that minimize the possible effects to user experience.

In this section we will compare 2 video calls:

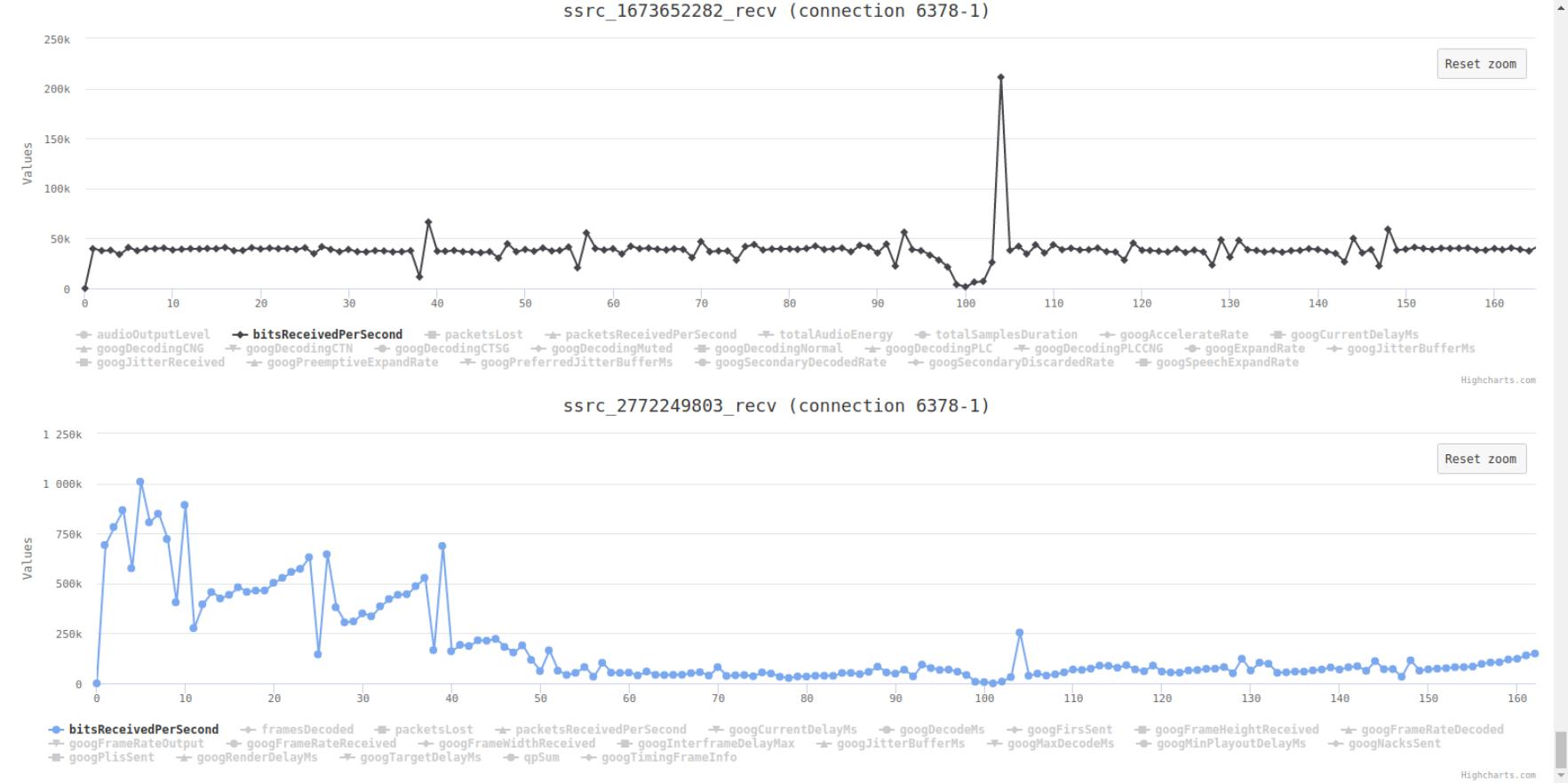
* The first one, between a computer with a wired Internet and a none-moving mobile device on 3G network.
* The second one, between the same computer and mobile device, but with the mobile device moving at a speed of 20-30km per hour.

The stats will be recorded at the computer using tools mentioned in chapter 2.

The graphs showing bandwidth plotted against time are presented below:



First call: Bandwidth for audio is stable around 40kbit/s, for video is stable around 500kbits/s

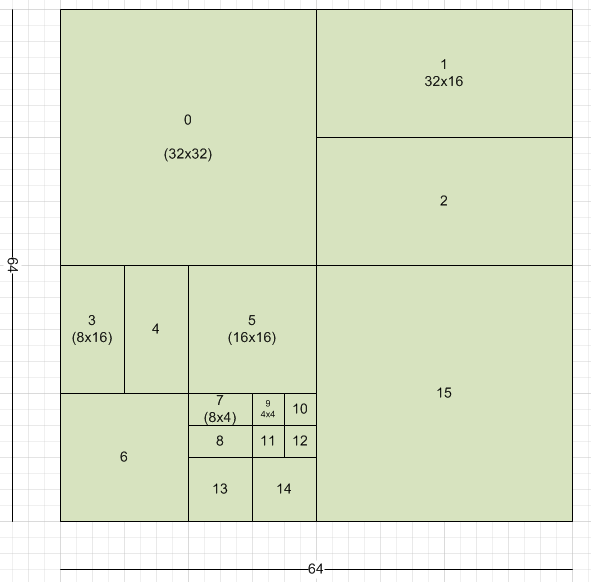


Second call: Bandwidth for video is stable around 40kbits. At the start when the mobile device is not yet moving the bandwidth for video is still around 500kbits, but then drops significantly to 50kbits/s when the device starts moving.

From the information above we can perceive that videos bandwidth is the bottleneck for the performance of a WebRTC call. The codecs used for encoding WebRTC video data in the most popular platforms is VP9. In the next part I will analyse how VP9 handle video encoding and propose possible improvements.

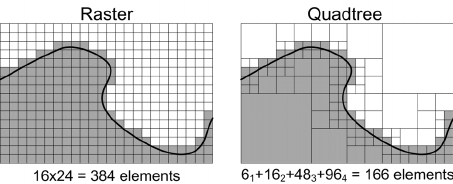
**4.1.2 VP9 codecs:**

**Intra-frame coding - block division:**

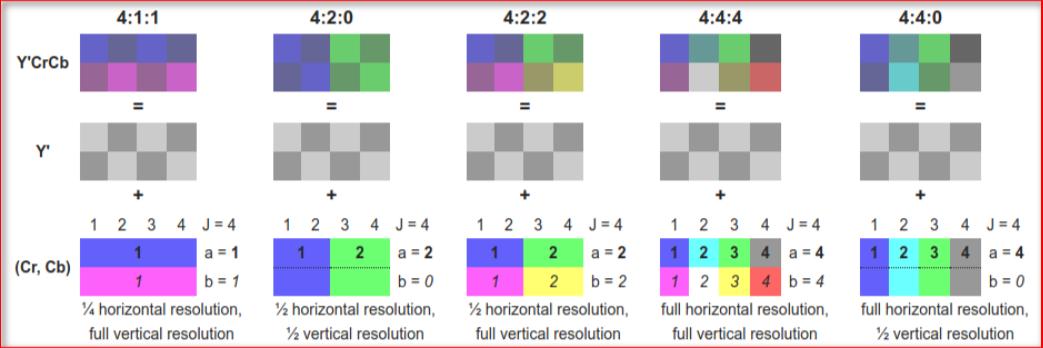


Most classical image compression algorithms divide an image into equal blocks of size 16\*16 or 32\*32 pixels, but recent algorithms like VP9 uses a more flexible scheme for block division:

The whole image is divided into 64\*64 blocks called super-blocks, then the depending on the complexity of the blocks - i.e is there a single area of color or are there multiple areas, these super-blocks will continue to be divided into smaller blocks for better compression rate. The sub-division procedure is similar to the quad-tree data structure used for complex terrain collision detection in game engines to reduce the bits needed to represent the terrain - or for video compression, to increase the compression rates.



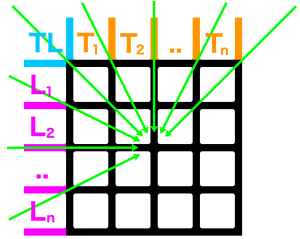
**Intra-frame coding - chroma subsampling:**



As with most image compression techniques, VP9 converts the color space from RGB to YCbCr, and then subsample the Chroma elements while keeping the Luma elements the intact.

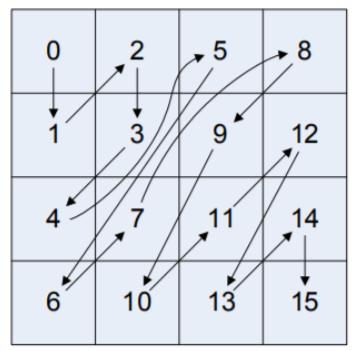
VP9 uses YCbCr 4:2:0 scheme meaning for each blocks of 4\*2 pixels, the number of chroma sample in a row is 2 and the difference of chroma sample between the two rows is 0. This effectively reduce the chroma resolution by half both vertically and horizontally.

**Intra-frame coding - intra block prediction:**



VP9 has 10 intra prediction modes for a particular block, that uses edge pixels to predict the contents of the current block.

* 8 directional modes, with 8 different values, each indicating a different directional angle – see schematic;
* TM (true-motion) mode, where each predicted pixel(x,y) = top(x) + left(y) – topleft
* DC (direct current) mode, where each predicted pixel(x,y) = average(top(1..n), left(1..n)).

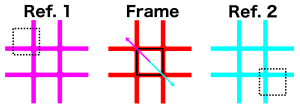
**Intra-frame coding - residual/entropy coding:**

VP9 uses 2D Discrete Cosine Transform (DCT) and Asymetric Cosine Transform (ADST) to separate and curtail the high frequencies components which is considered not visually important for the human eyes. The result of this process is a coefficient array which is achieved by scanning the 2D image in a particular zig-zag-like order. While most classical codecs just follow only the standard zig-zag order, VP9 uses a more sophisticated scan order.

**Inter-frame coding - reference frame:**

VP9 maintains a sliding window of 8 reference frame, called the reference pool. Each frame will be motion predicted using 3 frames in the pool. The encoded frame could then replace several frames inside the referrence pool. A special points in the architecture of VP9 is that each frame can be decoded from other frames with different resolution. That allows for real-time bitrate change which gracafully handles the bandwidth drop cases mentioned in the previous part that is a nature of mobile devices.

**Inter-frame coding:**

****

* One block can use 1 or 2 reference blocks
* The reference(s) are selected from the internal list of 3 active references per frame
* There are 4 inter mode that specifies how motion vectors are coded:
  + Zeromv: no motion.
  + nearestmv and nearmv: Use the first or second motion vectors in the list of motion vectors from nearby blocks in the previous frame.
  + newmv: generate a new motion vector
* The sub-pixel motion filter can have 3 values: regular, sharp or smooth.
* If the *inter mode* is newmv, the motion vector residual is added to the nearestmv value to generate a new motion vector.

**4.1.2 Proposal for video encoding improvements:**

After a thorough study of VP9 and many other popular codecs’ architectures, I have come up with an idea for the improvement of the video encoding process:

* Within a small time frame, the color space used in the image sequence is limited and does not variate much, and if it does it may not be of significant effects to human eye. Imagine the cases of interviewing videos where the color space is just important to describe the differences in the human facial details, not other objects in the background. This could allows us to use a smaller color space than the current color compression methods which always use chroma subsampling through out the video. I have never seen this information utilized in any codecs.

Because of the limited resouces and the difficulties of modifying source code of VP9 inside popular browsers, the implementation of the mentioned proposal is left out of scope for this research. Maybe this will be the inspiration for future researches by professionals with a deep interest for video compresion.

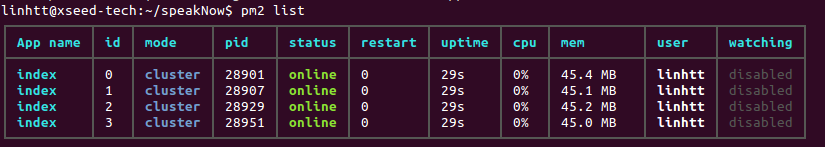
**4.2 HTTPS API optimizations:**

**4.2.1 Ultilizing multicore architectures by using load-balancing:**

The current problem with the HTTPS APIs is that they are running on a single core NodeJS process. This wastes the resouces of the 4 cores machine that the system is running on.

In this research, I have experimented three solutions for load balancing requests:

* PM2 - an automated NodeJS process manager that provides replications function without code modifications and load-balancing HTTP(S) requests among the replicas.
* Nginx - popular webserver and reverse proxy written in C.
* Iptables - linux kernel module, allows HA

****

*Starting four processes managed by pm2 listening simultaneously at port 3001*

|  |
| --- |
| upstream workers {  server localhost:3002;  server localhost:3003;  server localhost:3004;  server localhost:3005; }  server {  listen loghost:3001;  server\_name localhost;  location / {  proxy\_pass http://workers;  }  error\_page 500 502 503 504 /50x.html;  location = /50x.html {  root /usr/share/nginx/html;  } } |

*Nginx load-balancing configuration with 4 upstream workers*

|  |
| --- |
| root@xseed-tech:/home/linhtt# iptables -t nat -S -P PREROUTING ACCEPT -P INPUT ACCEPT -P OUTPUT ACCEPT -P POSTROUTING ACCEPT  -A PREROUTING -p tcp -m tcp --dport 3001 -m statistic --mode nth --every 4 --packet 0 -j REDIRECT --to-ports 3002  -A PREROUTING -p tcp -m tcp --dport 3001 -m statistic --mode nth --every 3 --packet 0 -j REDIRECT --to-ports 3003  -A PREROUTING -p tcp -m tcp --dport 3001 -m statistic --mode nth --every 2 --packet 0 -j REDIRECT --to-ports 3004  -A PREROUTING -p tcp -m tcp --dport 3001 -j REDIRECT --to-ports 3005 |

*Iptables configurations with statistic module - forwarding packets in a round robin fashion to 4 independent NodeJS processes.*

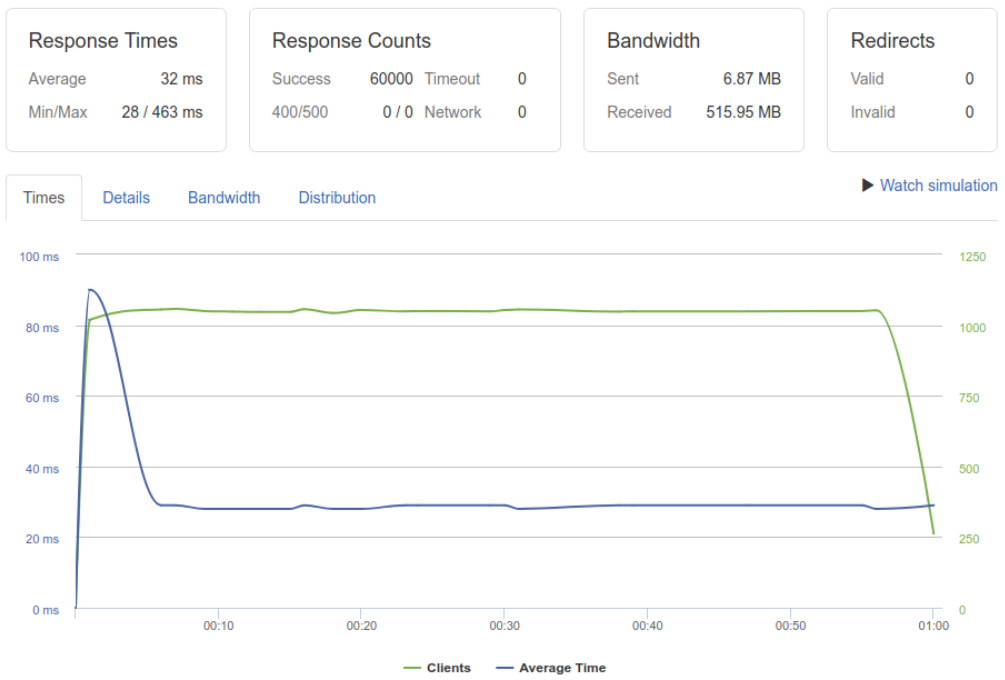
**Overall:** Iptables gives the best result so far, handling about 1,8% more requests then nginx and 5,8% more than pm2. In the scope of this research I will not go futher into detailed analysis and explaination. Instead, I will show the improvements of applying load-balancing and replication in real benchmarks with lively visualisations in the next chapter.

# 5. Post optimization benchmark

**5.1 Route xseed.tech/ieltsTest:**

Requests spanning over 10 seconds are considered errors.

**5.1.1 1000 concurrent clients per seconds**

****

* Average/Min/Max response time: 32/28/463 ms
* Error rate: 0%

**5.1.2 2000 concurrent clients per seconds**

****

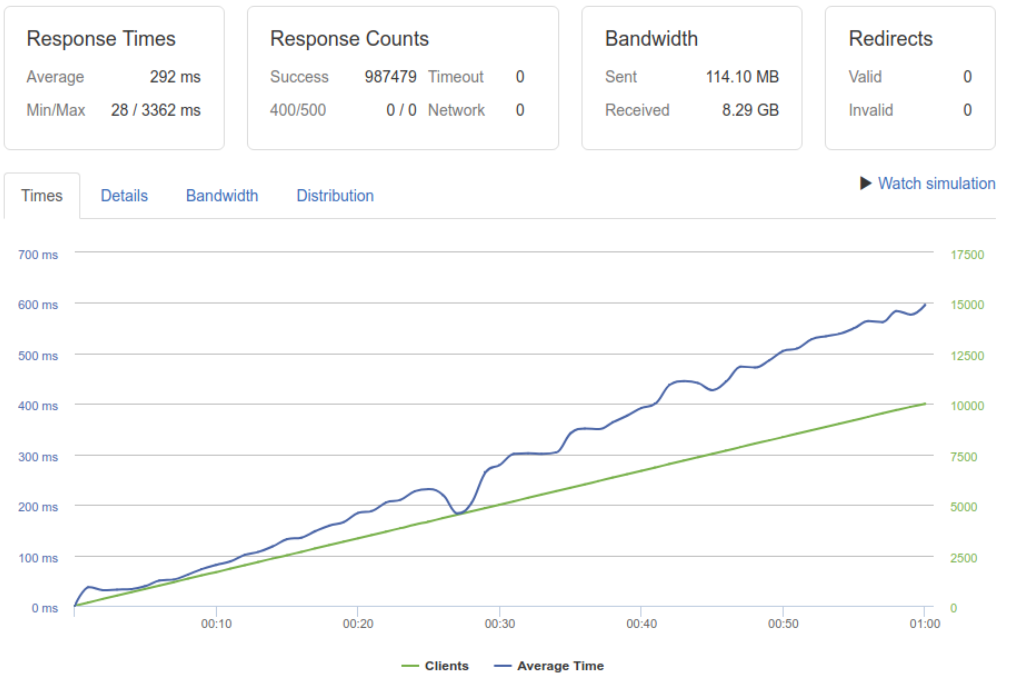
* Average/Min/Max response time: 41/28/1966 ms
* Error rate: 0.0%

**5.1.3 5000 concurrent clients per seconds**

****

* Average/Min/Max response time: 1274/35/12299 ms
* Error rate: 0.1%

**5.1.4 From 0 to 10000 clients in 1 minute:**

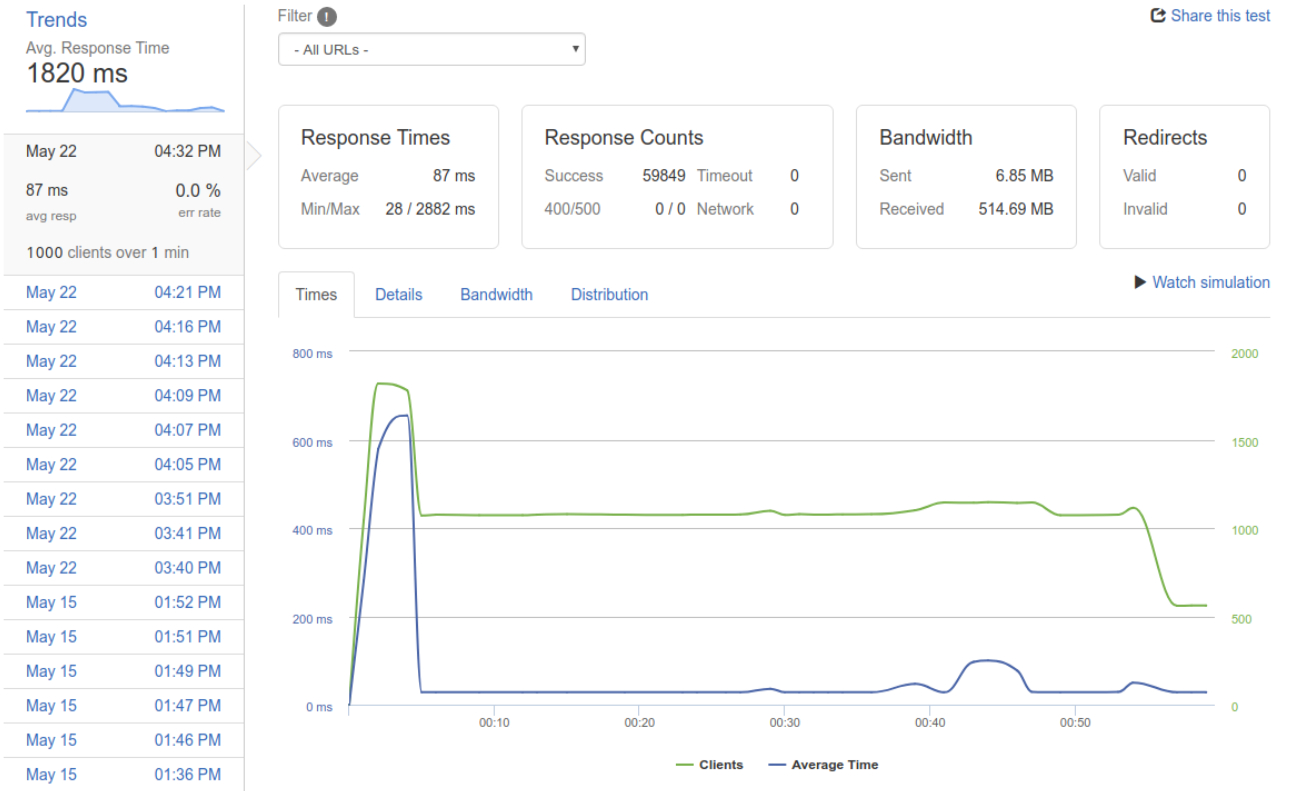
****

* Average/Min/Max response time: 988/28/10972 ms
* Error rate: 0.0%

**5.2 Route xseed.tech/reception:**

Requests spanning over 10 seconds are considered errors.

**5.2.1 1000 concurrent clients per seconds**

****

* Average/Min/Max response time: 87/28/2882 ms
* Error rate: 0%

**5.2.2 2000 concurrent clients per seconds**

****

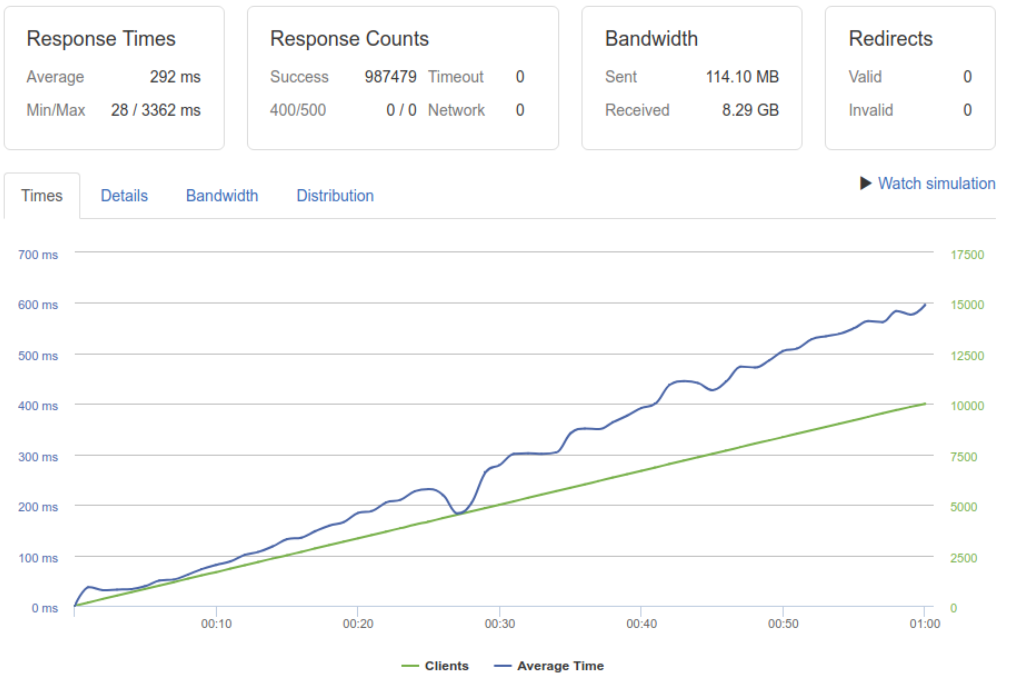
* Average/Min/Max response time: 41/28/1966 ms
* Error rate: 0.0%

**3.1.3 5000 concurrent clients per seconds**

****

* Average/Min/Max response time: 1274/35/12299 ms
* Error rate: 0.1%

**3.1.4 From 0 to 10000 clients in 1 minute:**

****

* Average/Min/Max response time: 988/28/10972 ms
* Error rate: 0.0%

# 6. Summary & referrences:

In this research, I have proposed, implemented, and tested different solutions for improving the availability of the HTTPS APIs by utilizing the power of multicore architecture and iptables load-balancing. Additionally, I have looked at how the underlying codec VP9 of WebRTC are performing video encoding. While no promising improvements are made for VP9 video encoding process, the improvements for the HTTPS APIs are significant. The below summarization of the benchmarking results wraps up the impact of the changes made:

Route xseed.tech/ieltsTest - serving the content of the IELTS tests:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Test type | Pre-optimization | | | | Post-optimization | | | |
| Min  resp(ms) | Max  resp(ms) | Avg  resp | Err rate | Min  resp | Max  resp | Avg  resp | Err rate |
| 1000cps | 28 | 2635 | 68 | 0% | 28 | 463 | 32 | 0% |
| 2000cps | 28 | 10671 | 215 | 0.6% | 28 | 1966 | 41 | 0% |
| 5000cps | 35 | 12299 | 1274 | 12.4% | 28 | 9876 | 704 | 0.1% |
| 0-10000cps | 28 | 10972 | 988 | 0.1% | 28 | 4593 | 339 | 0% |

Route xseed.tech/reception - serving the application’s code:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Test type | Pre-optimization | | | | Post-optimization | | | |
| Min  resp | Max  resp | Avg  resp | Err rate | Min  resp | Max  resp | Avg  resp | Err rate |
| 1000cps | 28 | 2882 | 87 | 0% | 28 | 467 | 30 | 0% |
| 2000cps | 28 | 10707 | 310 | 0.7% | 28 | 3601 | 105 | 0% |
| 5000cps | 94 | 16466 | 1446 | 12.7% | 29 | 11434 | 1417 | 0.1% |
| 0-10000cps | 28 | 11203 | 1087 | 0.3% | 28 | 3362 | 292 | 0% |

From the data we can conclude that when working with multicore computing architecture, the uses of multiprocesses and a good load-balancer plays a vital role in the system’s availability.

*Referrences:*

All of the benchmarks and data recorded in this research can be found here:

<https://drive.google.com/open?id=11YVF1i0G5sL34j4Z-G_nWw8sOpN6l7hdMcKqva86Rmk>

*WebRTC internal documentation* -

<https://testrtc.com/webrtc-internals-documentation/>

WebRTC Dump Importer -

<https://fippo.github.io/webrtc-dump-importer/>

VP9 Bitstream & Decoding Process Specification - <https://storage.googleapis.com/downloads.webmproject.org/docs/vp9/vp9-bitstream-specification-v0.6-20160331-draft.pdf>

Chroma subsampling -

<https://en.wikipedia.org/wiki/Chroma_subsampling>

Quarter-pixel motion -

<https://en.wikipedia.org/wiki/Quarter-pixel_motion>

Overview of the VP9 video codec - <https://blogs.gnome.org/rbultje/2016/12/13/overview-of-the-vp9-video-codec/>

Simple Stateful Load Balancer with iptables and NAT - <https://www.webair.com/community/simple-stateful-load-balancer-with-iptables-and-nat/>

Node.js process load balance performance: comparing cluster module, iptables and Nginx - <https://medium.com/@fermads/node-js-process-load-balancing-comparing-cluster-iptables-and-nginx-6746aaf38272>