Measuring latency from the browser

Agustín Formoso
LACNIC Labs
agustin@lacnic.net

ABSTRACT

Latency's increasing importance encourages research on how to develop new measurement techniques. In this scope, we present a measuring alternative which involves determining HTTP latency from a JavaScript tool. This alternative has the benefit of generating massive amounts of data, from probes located in many different places and thus reflecting the true state of a network, but the disadvantage of introducing noise to the measurements. The aim of this paper is to cover the solution found to correct the noise that was generated when the tester was ran in multiple platforms.

Key words
Measurements, latency, browser, JavaScript.

1. INTRODUCTION

Latency measurements have a strong relationship with network connectivity: the lower the latency values, the better the connectivity. Usually longer paths require more time for the same information to travel; sub-optimal routing makes geographically close points have large latency values between them. It is not rare to see neighbor countries have (ICMP) latencies of ~200 ms, or even in-country latency measurements to be above ~100 ms.

Given this scope, by making large scale "mesh" latency measurements, from different origins to different destinations, a connectivity matrix between AS, countries, or regions can be built.

For this matrix to be representative, these latency measurements have to be done in great amount, from many places to many other places, and consistently over time. The way we propose to perform such measurements is placing a background JavaScript in several websites. This would enable visitors to trigger background measurements, à la Google Analytics.

Being a web language, JavaScript is designed for other purposes but network measurements. Taking the language to its limits results unavoidably in the introduction of error, considering the fact that the script will be running in several different environments (OS and browser combinations). It is therefore of great importance to measure and mitigate the error introduced in the measurements. This paper tackles this problem and proposes a way to clear the error by normalizing the different environments in which the script was ran.

2. The tool

2.1 Implementation

Latency can be measured in a variety of ways, depending on the type of application, OSI layer, protocol, among other configurations. The approach chosen this time is to imagine the network as a black box, and generate large enough amounts of data in order to consider the measurements representative of the network's latency.

One low-cost and quick-deploy option is to use massive services already in the web and web standards, like already-there web servers and JavaScript. A web page visitor triggering background latency measurements à la Google Analytics will clearly generate a large and pseudo-random dataset.

The following snippet[1] shows a simple way of measuring HTTP latency from a web browser in JavaScript.

```
var ts, rtt, url;
ts = +new Date;
// HTTP GET to remote url
rtt = (+new Date - ts);
```

What is the snippet above measuring? Is it measuring actual HTTP RTTs? As it is, the snippet is not measuring HTTP RTTs alone. Among the factors affecting the tests are the browser's JavaScript engine performance, and the OS on which the browser is running. As mentioned previously, this paper will explain the tests by which these kinds of errors are mitigated.

3. The tests

As mentioned above, two factors affecting JavaScript latency measurements are the browser in which the test is running and the OS in which the browser is running. Not all browsers implement TCP connections in the same way, not even between different browser versions! Moreover, the browser JavaScript engine may differ between different browser releases. The same happens for OSes and how they handle network communications. These uncertainties need to be measured in order to mitigate the error introduced in the measurements.

In order to measure the uncertainty introduced by the browser + OS combination, the tester had to be ran under controlled conditions, varying the browser + OS combination. For testing how this combination affects latency measurements, we should make several measurements between a fixed origin and fixed destination, at the same time (or in a very short time span), in order to have as few variations as possible in other factors which affect latency measurements. By this means we want the only change in our experiment to be the browser and OS chosen for that experiment.

3.1 Test objectives

The main objective of the tests was to build a unified criteria for the different environments in which the tester was running. These environments are determined by the HTTP User Agent field, which identifies browser and OS of the running code. After knowing browser and OS, the only thing to do is lookup such browser + OS combination in a correction matrix and alter the measurements in order to correct them from the error that is being

[1] This is a simplistic version of the actual script used, but the script implementation is beyond the scope of this paper.

introduced in that specific platform. The snippet below shows an example of what we were looking for.

```
correct_measurement(measurement, http_user_agent):
        os, browser = parse_ua(http_user_agent)
        factor = matrix_lookup(os, browser)
        return correct(measurement, factor)
```

The example below shows how the User Agent field can be parsed and the measurement corrected (example values).

HTTP User Agent field	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_2) AppleWebKit/537.75.14 (KHTML, like Gecko) Version/7.0.3 Safari/537.75.14
Tester environment	OS: OSX 10.9.2 Browser: Safari 7.0.3
Correction factor	Shift the dataset 10 ms to the left

Table 1. Correction factors for a specific User Agent

3.2 Test products

The matrix mentioned above is nothing more than the aggregation of all the test results. This matrix should give an idea of how to correct the measurements based on a chosen environment. The matrix below gives us an idea of how the matrix should look like (example values):

Browser / OS	Windows 8	Windows 7	OSX 10.9
Chrome v.39	268 ms	270 ms	266 ms
Safari v.7	266 ms	269 ms	270 ms

Table 2. Correction matrix example

Alternatively, and choosing one environment as our base configuration, we could build a delta matrix to show the correction factor to be used in the correction process. For this example we chose Chrome v.39 on Windows 8 as our reference environment (example values):

Browser / OS	Windows 8	Windows 7	OSX 10.9
Chrome v.39	0 ms	+2 ms	-2 ms
Safari v.7	-2 ms	+1 ms	+2 ms

Table 3. Normalized correction matrix example

Having this table would be enough to shift the measurements in the amount needed to correct environment differences. For the example above (Safari 7.0.3 on OSX 10.9.2), we would have to add 2 ms to the measured data.

4. Test results

After setting up our configurations in a virtualized cloud service, which would run our tests in controlled conditions, we gathered 10K samples for each browser + OS combination and then compared the results.

4.1 Comparing raw results

In order to visualize the tests done in each environment and draw conclusions easier from the raw data, a series of charts were generated. These charts show the whole dataset as histograms, divided by configuration.

The following chart shows the histograms representing the data corresponding to the Google Chrome family (latest 10 versions) on Windows 8. For each Chrome version, one translucent histogram was generated and overlapped with the rest. The final product is a chart that includes 10 overlapped histograms, plus an aggregated histogram (darkest red) that includes all the 10 environments together.

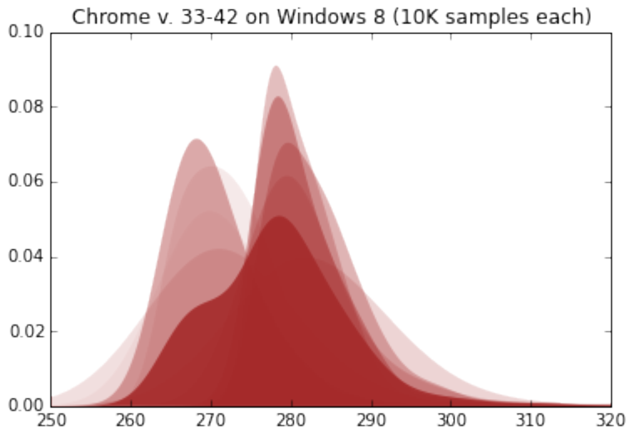

Figure 1. Uncorrected results for Chrome on Windows 8

One of the first conclusions we can draw from the chart is that the different environments for Chrome are grouped in two: one group is centered around ~270 ms and the other around ~280 ms. These two "clusters" show that even when using the same browser family, the tests results may vary in around ~10 ms. Going a bit further, the 270 ms cluster is formed by Chrome versions 34, 35, and 36, while the 280 cluster is formed by the rest; this makes us think versions 34, 35, and 36 have something different from the rest, probably the way they run the JavaScript code or the way they handle TCP connections (we're yet unable to tell what the difference is, but we can see there is one and we can correct it). Whatever the difference is, these tests give us an idea on how to correct the latency measurements.

4.2 Correcting raw results

Visualizing the raw data is not enough to correct the results. What we need to do is to identify the necessary transformation to be done in order to normalize our dataset. To do this we need to do three things

1. Identify a reference environment against which to perform the transformations
2. Identify the transformation to be applied
3. Perform the actual transformations.

For this example, we chose Chrome v. 39 on Windows 8 as our reference set, and then identified the subsequent transformations to be applied for the rest of the dataset. As a first approach, a simple transformation is to shift the different environments to match the reference environment's *median*. Our transformations values are listed in the table below under the Δ column:

Version	Std. dev. (ms)	Median (ms)	Δ (ms)
33	18	284	-4
34	21	271	+9
35	8	272	+8
36	30	271	+9

37	32	282	-2
38	19	280	0
39	7	280	0
40	10	282	-2
41	13	269	+11
42	10	280	0

Table 4. Results with the corresponding transformation value for different Chrome versions

After applying the corresponding transformations we end up with the following result dataset:

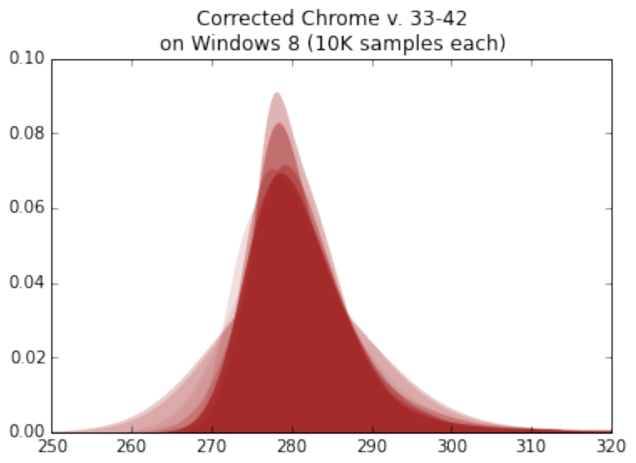

Figure 2. Corrected results for Chrome on Windows 8

The corrected result set has one great mean and not "two means" as the uncorrected dataset, as the samples are now shifted, and has a lower std. dev, as the datasets are now less spread over the *x* axis. For this example the mean was shifted to 280 ms (Chrome v. 39) and the resulting std. dev. of the whole dataset changed from 20 to 19 ms. It might not seem a great change at first sight, but thanks to the visualizations we know that the underlying dataset now is better represented by the statistics and that different browser versions are not too distant from one another.

4.3 Inter-browser results

Another behavior we were looking for is the one between browser families (for a fixed OS): aggregating each family and then comparing them with other families might give us better insight on how each browser behaved on a specific platform. For this case we chose the families of Chrome, Firefox, and Internet Explorer families on Windows 8.

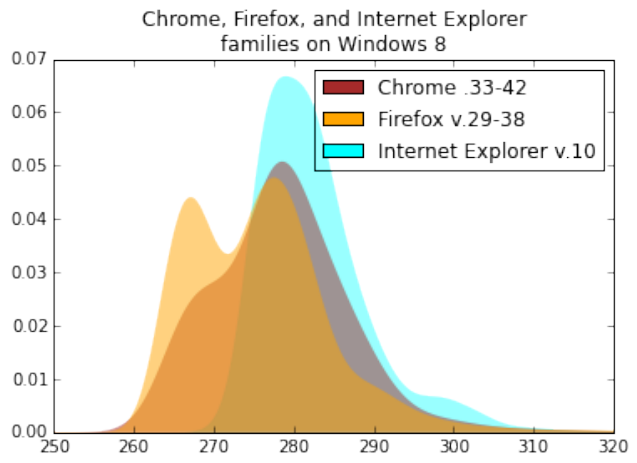

Figure 3. Results for Chrome, Firefox, and Internet Explorer

This new diagram showed no surprising results. The three families show different characteristics, but are bounded in a 200 – 300 ms range. Specifically, the datasets show the following properties:

Browser family	Median	Std. Dev.
Chrome	278	20
Firefox	275	13
Internet Explorer	281	9
Aggregation	***279***	***14***

Table 5. Results for the Chrome, Firefox, and Internet Explorer families

The table shown previously gives us an idea of how the measurements may vary with the browser given they're running on Windows 8 (for the three families considered, but it can be scaled very easily to any existent browser family and any other OS). One of the things that is relevant from the table is the values corresponding the "Aggregation" row, which aggregates all the samples from the three families considered. The whole aggregation has 30K samples (10K each family), a median of 279 ms and a std. dev. of 14 ms, which means that most of the measurements will be bound in a 279 ± 28 ms box. Those figures indicate that for the measurements that had not been corrected yet, and considering our 30K sample is big enough, the error margin is at most of 10% (for a ~95% confidence interval), which is not excessively high, considering that this is not a traditional way of measuring latency and the amount of variables affecting the measurements is high.

4.4 Correction matrix

From transformations and histograms like the ones shown above, where we can visualize the statistical profile of the measurements, we can build a matrix that relates all the results and let us implement our test correction function we are looking for.

The matrix should have enough information in order to transform an HTTP latency result into a corrected result. This matrix should have information of OS, OS version, browser, and browser version, as well as the correction factor explained at the beginning of this paper.

After exhaustive testing, the matrix was built, accepting the following parameters:

- Browsers: Chrome versions 33-42, Firefox versions 29-38, Safari, and Opera.
- OSes: Windows 10, Windows 8.1, Windows 8, Windows 7, Windows XP, OSX 10.10, OSX 10.9, OSX 10.8, OSX 10.7, and OSX 10.6.

Please note that not all browser and OS combinations are compatible.

5. CONCLUSIONS

Latency tests measured from the browser, via JavaScript scripts, are a good means of collecting large amounts of network data, and it is representative of the end-user perspective. However, the considerations covered in this paper have to be taken into account in order to correct browser and OS implementation differences, specially those related to JavaScript performance and TCP connections. If applied correctly, such corrections enable a layer of abstraction and someone using the measurements dataset does not have to consider the platform in which the tests were ran over.

6. FURTHER WORK

Further work is to be done in order to consider different use cases, specially those contemplating higher and lower HTTP ping times (well below and well above ~270 ms). There is also room for improvement in seeking a correction factor for standard deviation, as this paper only covered the correction factor for the median value of the measurements; one suggestion for correcting the standard deviation is shrinking the *x* axis having centre its median.

Further work could also be done regarding mobile environments.

7. Brief section about latency results

This section contains some visualizations and primary conclusions derived from the results gathered so far, with the aim of introducing the reader in the context of the LAC region connectivity and the project deliverables. The charts shown in this section are the result of letting the script run at the LACNIC home page for about 12 months.

One value we are interested in looking at is the latency inside any country. From the top of our minds we think that this value should be low, as paths should be short, but in practice we see that these values are way above our expectations; most of the results are above 100 ms. The following chart summarizes our first impressions about in-country latencies.

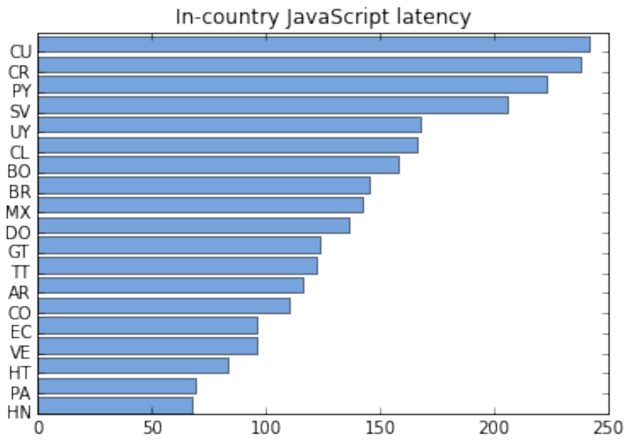

Figure 4. In-country latencies for the LAC region measured by the JavaScript tool

Another visualization that might give us better insight is a country-level matrix depicting the country originating the measurement in the *y* axis and the country of destination in the *x* axis. The previous in-country bar chart corresponds with the matrix's main diagonal.

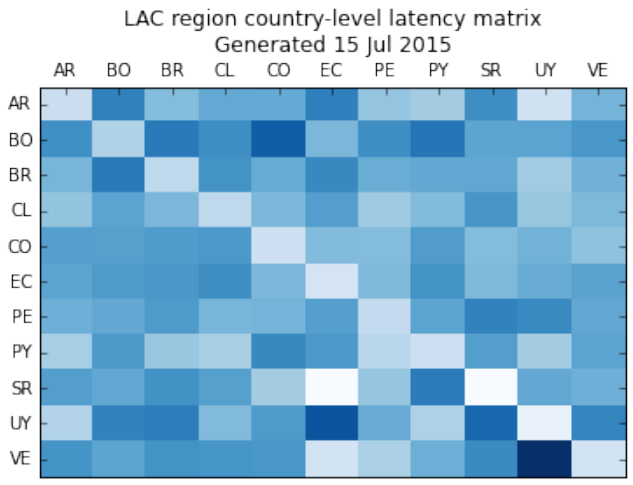

Figure 5. Latency matrix for the LAC region measured by the JavaScript tool

One quick conclusion is that the matrix is not symmetrical, not even close. This might give us a hint on asymmetric paths between countries.

As a final example we will present a chart showing a histogram representing the latencies between two neighbor countries: Uruguay and Brazil. The following series of charts show the results gathered in the last four months (ending mid July 2015).

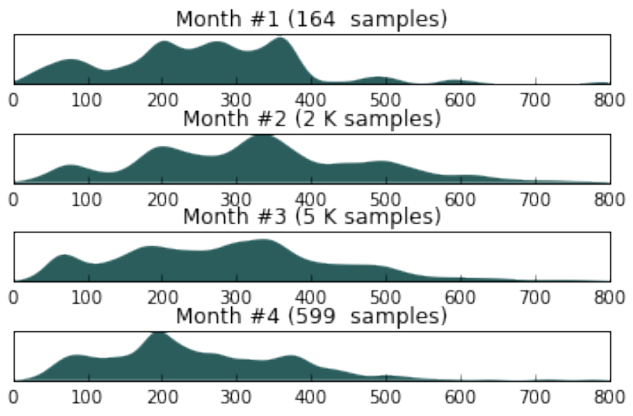

Figure 6. Results between Uruguay and Brazil (last four months)

One of the main characteristics of this chart is that measurements are clearly grouped in three: at ~75, ~200, and ~350 ms (and a shy group at ~280 ms that appears only at months #1 and #4). Lets call them A, B, C (and D).

The existence of three (or four) groups is probably consequence of the existence of more than one path between these two countries. Another aspect to look at is the great amount of measurements that fall inside the groups that are not group A. Network operators from these two countries might want to watch what is going on between their networks. People doing research about network connectivity might want to look at the dataset available via the project's RESTful API [1]. More protocols, filters, and visualizations are available at the project's Reports section [2].

8. ACKNOWLEDGEMENTS

Special acknowledgments go to the LACNIC Software Development Team which made this work possible. Thanks!

9. References

[1] The Simon Project API Documentation. http://simon.lacnic.net/simon/api

[2] The Simon Project Reports. http://simon.lacnic.net/simon/reports