DOI:10.1145/2436256.2436270

Article development led by acmqueue
queue.acm.org

Website performance data has never been more readily available.

BY PATRICK MEENAN

How Fast is Your Website?

THE OVERWHELMING EVIDENCE indicates a website's performance (speed) correlates directly to its success, across industries and business metrics. With such a clear correlation (and even proven causation), it is important to monitor how your website performs. So, how fast is your website?

First, it is important to understand that no single number will answer that question. Even if you have defined exactly what you are trying to measure on your website, performance will vary widely across your user base and across the different pages on your site.

Here, I discuss active testing techniques that have been traditionally used and then explain newer technologies that permit the browser to report back to the server accurate timing data.

Traditionally, monitoring tools are used in active testing to measure the performance of a website, and the results end up being plotted on a time-series chart. You may choose specific pages to measure and geographic locations from which to measure, and then the test machines load the pages periodically from the various locations and the performance gets reported. The results are usually quite consistent and provide a great baseline for identifying variations, but the measurements are not representative of actual users.

The main limiting factors in using

active testing for understanding Web performance are:

▸ *Page Selection.* Active testing usually tests only a small subset of the pages that users visit.

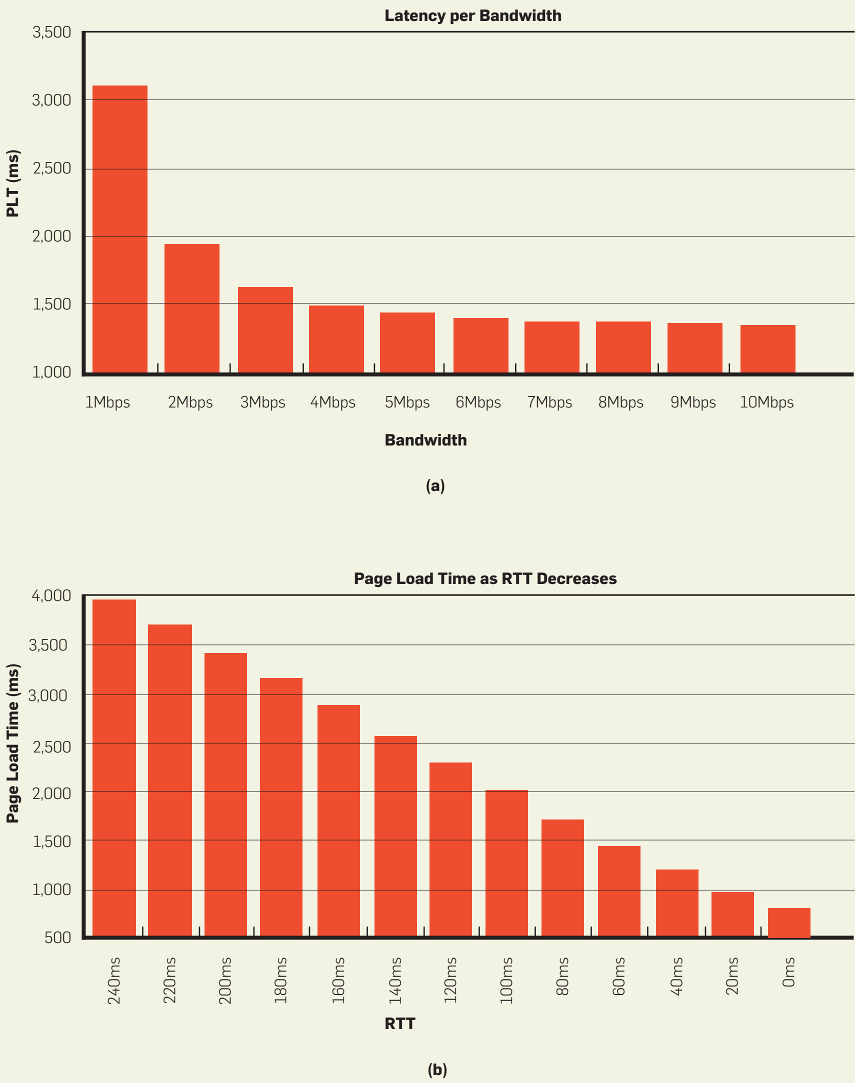
▸ *Browser Cache.* The test machines usually operate with a clear browser cache and do not show the impact of cached content. This includes content that not only is shared across pages but also is widely distributed (such as JavaScript for social-sharing widgets, shared code libraries, and code used by advertising systems).

▸ *Machine Configuration.* The test machines tend to be clean configurations without viruses, adware, browser toolbars, or antivirus and all sorts of other software that mess with the performance of the user's machine and browsing in the real world.

▸ *Browsers.* The browser wars have never been more alive, with various versions of Internet Explorer, Chrome, Firefox, Safari, and Opera all maintaining significant market share. Active testing is usually limited to a small number of browsers, so you must try to select one that is representative of your user base (assuming you even have a choice).

▸ *Connectivity.* Users have all sorts of Internet connections with huge differences in both latency and bandwidth, from dial-up to 3G mobile to cable, DSL, and fiber-broadband connections. With active testing you usually must choose a few specific configurations for testing or perform tests directly from a data center.

Figure 1. Bandwidth and latency.



Source: http://www.belshe.com/2010/05/24/more-bandwidth-doesnt-matter-much/

The impact of connectivity cannot be understated. As illustrated in Figure 1, bandwidth can have a significant impact on Web performance, particularly at the slower speeds (less than 2Mbps), and latency has a near-linear correlation with Web performance.

Testing from within a data center skews heavily with effectively unlimited bandwidth and no last-mile latency (which for actual users can range anywhere from 20 to 200-plus milliseconds depending on the type of connection). The connectivity skew also is exaggerated by CDNs (content distribution networks) because these usually have edge nodes collocated in the same data centers as the test machines for many monitoring providers, reducing latency to close to zero.

Passive Performance Reporting

Given the constraints of active testing in measuring performance, a lot of work has focused on reporting the actual performance experienced by end users as they browse a site. Historically, sites have implemented their own solutions for measuring and reporting performance, although there have always been caveats because JavaScript on the page did not have the ability to report on the full experience. Reporting on the performance from actual user sessions is generally referred to as RUM (real user measurement).

When loading a page, the browser generally:

▸ Does a DNS lookup to map the host name to an IP address.

▸ Establishes a TCP connection to the server.

▸ Negotiates an encrypted connection (for HTTPS pages).

▸ Sends a request to the server for the page HTML.

▸ If the server responds with a redirect, repeats all of the above steps for the new location.

▸ Downloads the HTML response from the server.

▸ Parses the HTML, downloads all of the referenced content, and executes the page code.

The last step is very complex and constitutes the majority of time (usually over 80%[4,6]) consumed by most sites and is the only part of the experience that you have been able to directly measure from JavaScript. When a page is instrumented using JavaScript to measure performance, JavaScript's first chance to execute is at the point where the HTML has been downloaded from the server and the browser has begun executing the code. That leaves the approximately 20% of the page-load time outside of the measurement capabilities of in-page code. Several clever workarounds have been implemented over the years, but getting a reliable start time for measurements from the real world has been the biggest barrier to using RUM. Over the years browsers implemented proprietary metrics that they would expose that provided a page start time (largely because the browser vendors also ran large websites and needed better instrumentation themselves), but the implementations were not consistent with each other and browser coverage was not very good.

In 2010, the browser vendors got together under the W3C banner and formed the Web Performance Working Group[15] to standardize the interfaces and work toward improving the state of Web-performance measurement and APIs in the browsers. (As of this writing, browser support for the various timing and performance information is somewhat varied, as shown in the accompanying table.)

In late 2010 the group released the Navigation Timing specification,[9] which has since been implemented in Internet Explorer (9+ desktop and mobile), Chrome (15+ all platforms), Firefox (7+), and Android (4+).

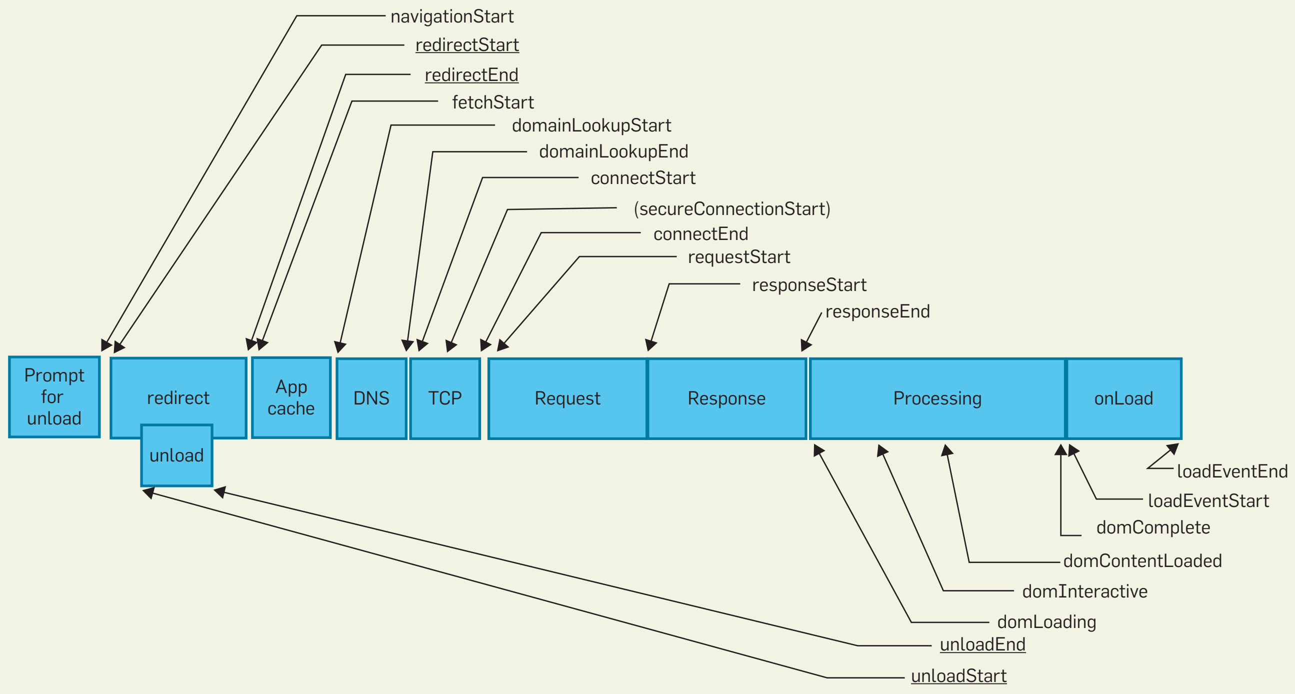
The largest benefit navigation timing provides is that it exposes a lot of timings that lead up to the HTML loading, as shown in Figure 2. In addition to providing a good start time, it exposes information about any redirects, DNS lookup times, time to connect to the server, and how long it takes the Web server to respond to the request—*for every user and for every page the user visits*.

The measurement points are exposed to the Document Object Model (DOM) through the performance object and make it trivial to calculate load times (or arbitrary intervals, really) from JavaScript.

Browser support for performance information.

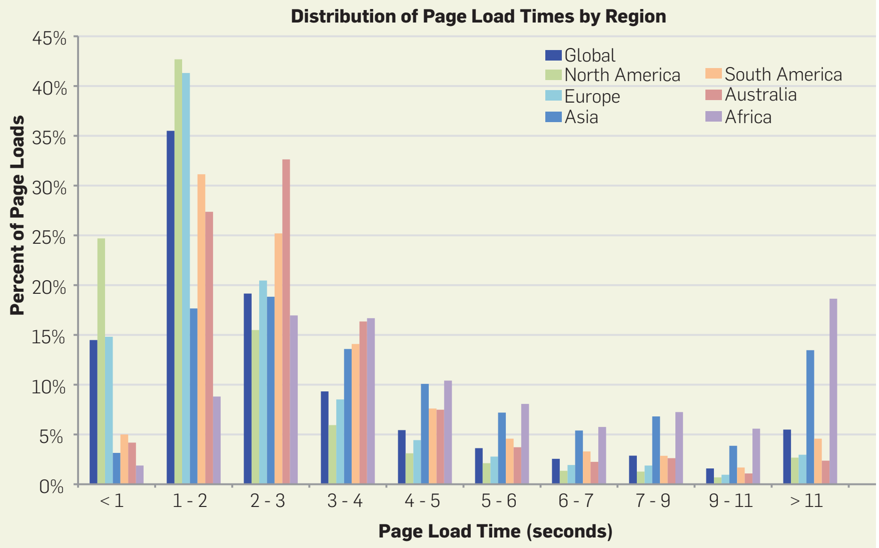
	IE (Desktop and Mobile)	**Chrome (Desktop and Android)**	**Firefox (Desktop and Android)**	**Safari (Desktop and Mobile)**
Navigation	9+	6+	7+	
Resource Timing	10+	26+		
requestAnimationFrame	10+	10+	4+	6+
High Resolution Time	10+	21+	15+	

Figure 2. Navigation timing.



Source: http://www.w3.org/TR/navigation-timing/

Figure 3. Distribution of page load times.



The following code calculates the page load time:

```
var loadTime = performance.timing.loadEventStart - performance.timing.navigationStart;
```

Just make sure to record measurements after the point you are measuring (`loadEventStart` will not be set until the actual load event happens, so attaching a listener to the load event is a good way of knowing it is available for measurement).

The times are reported as regular JavaScript times and can be compared with arbitrary points in time:

```
var now = new Date().getTime();
var elapsed = now - performance.loadEventStart;
```

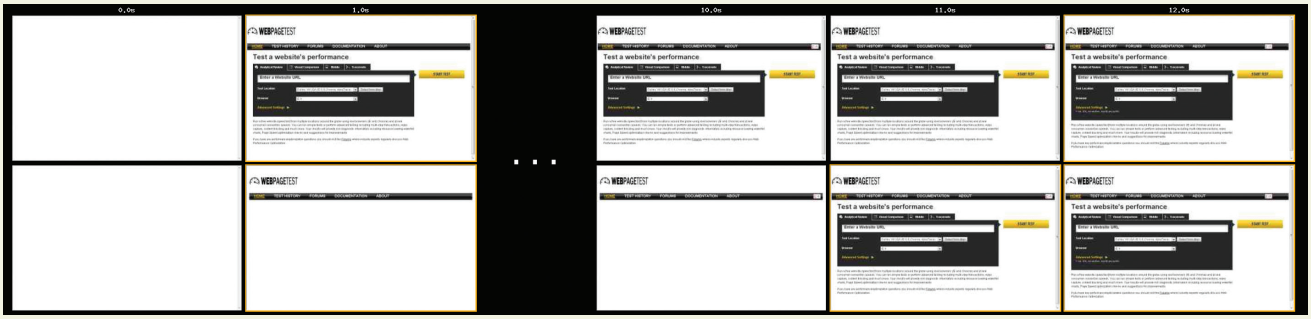
What is interesting about users' clocks is that they don't always move forward and are not always linear, so error checking and data filtering are necessary to make sure only reasonable values are being reported. For example, if a user's computer connects to a time server and adjusts the clock, it can jump forward or backward in between measurements, resulting in negative times. If a user switches apps on a phone, then the page can be paused midload until the user opens the browser again, leading to load times that span days.

The W3C Web Performance Group recognized the clock issue, as well as the need for granularity beyond one-millisecond resolution, and also introduced the High Resolution Time specification.[7] This is a time measurement as accurate as the platform it is running on supports (at least one-millisecond resolution but significantly better than that on most platforms), and it is guaranteed always to increase and not be skewed by clock changes. This makes it significantly better for measuring elapsed time intervals. It is exposed on the DOM to JavaScript as `performance.now()` and is relative to the page navigation start time.

Sites can now report on the actual performance for all of their users, and monitoring and analytics providers are also providing ways to mine the data (Google Analytics reports the navigation timing along with business metrics, for example[2]), so websites may already be getting real user performance-measurement data. Google Analytics also makes an interface available for reporting arbitrary performance data so website operators can add timings for anything else they would like to measure.[3]

Unlike the results from active testing, the data from real users is noisy, and averages tend not to work well at all. Performance data turns into regular analytics like the rest of a company's business metrics, and managers have to do a lot of similar analysis such as looking at users from specific regions, looking at percentiles instead of averages, and looking at the distribution of the results.

Figure 4. Different page loads with the same load event time.



For example, Figure 3 shows the distribution of page-load times for recent visitors to a website (approximately two million data points). In this case U.S. and European traffic skews toward faster load times, while Asia and Africa skew slower with a significant number of users experiencing page-load times of more than 11 seconds. Demographic issues drive some of the skew (particularly around Internet connectivity in different regions), but it is also a result of the CDN used to distribute static files. The CDN does not have nodes in Eastern Asia or Africa, so one experiment would be to try a different CDN for a portion of the traffic and watch for a positive impact on the page-load times from real users.

When is Timing Done?

One thing to consider when measuring the performance of a site is when to stop the timing. Just about every measurement tool or service will default to reporting the time until the browser's load event fires. This is largely because it is the only well-defined point that is consistent across sites and reasonably consistent across browsers.[8] In the world of completely static pages, the load event is the point where the browser has finished loading all of the content referred to by the HTML (including all style sheets and images, among others).

All of the browsers implement the basic load event consistently as long as scripts on the page do not change the content. The HTML5 specification clarified the behavior for when scripts modify the DOM (adding images, other scripts, and so on), and the load event now also includes all of the resources that are added to the DOM dynamically. The notable exception to this behavior is Internet Explorer prior to version 10, which did not block the load event for dynamically inserted content (since it predated the spec that made the behavior clear).

The time until the load event is an excellent technical measurement, but it may not convey how fast or slow a page was for the user. The main issues are that it measures the time until every request completes, even those that are not visible to the user (content below the currently displayed page, invisible tracking pixels, among others), and it does not tell you anything about how quickly the content was displayed to the user. As illustrated in Figure 4, a page that displays a completely blank screen right up until the load event and a page that displays all visible content significantly earlier can both have the same time as reported by measuring the load event.

One thing to consider when measuring the performance of a site is when to stop the timing. Just about every measurement tool or service will default to reporting the time until the browser's load event fires.

It is also quite easy to optimize for the load-event metric by loading all of the content through JavaScript after the page itself has loaded. That will make the technical load event fire very fast, but the user experience will suffer as a result (and will no longer be measured).

Several attempts have been made to find a generic measurement that can accurately reflect the user experience:

- *Time to first paint.* If implemented correctly, this can tell you the first point in time when the user sees something other than a blank white screen. It doesn't necessarily mean the user sees anything useful (as in the previous example, it is just the logo and menu bar), but it is important feedback telling the user that the page is loading. There is no consistent way to measure this from the field. Internet Explorer exposes a custom `msFirstPaint` time on the `performance.timing` interface, and Chrome exposes a time under `chrome.loadTimes().firstPaintTime`; in both cases, however, it is possible that the first thing the browser painted was still a blank white screen (though this is better than nothing and good for trending).
- *Above-the-fold time.*[1] This measures the point in time when the last visual change is made to the visible part of the page. It was an attempt at a lab-based measurement that captured the user experience better. It can be tuned to work reasonably well to reveal when the last bit of visible content is displayed, but it doesn't distinguish between a tiny social button being added to the page and the entire page loading late (in the example, the first page adds a social button at the end, and both pages would have the same above-the-fold measurement time). The above-the-fold time

Figure 5. Visual progress display.

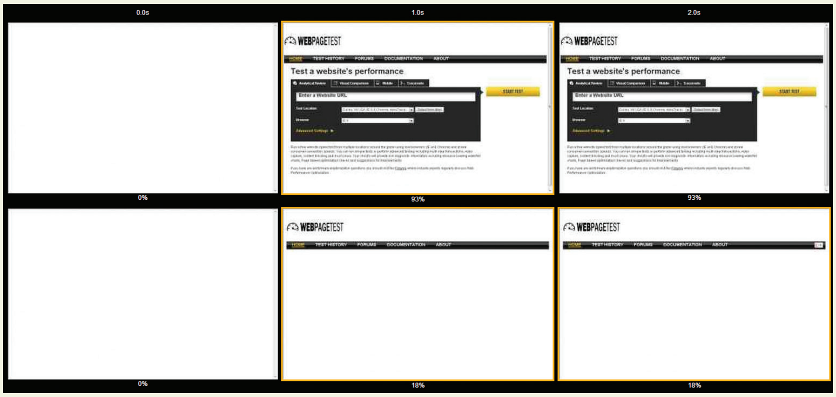
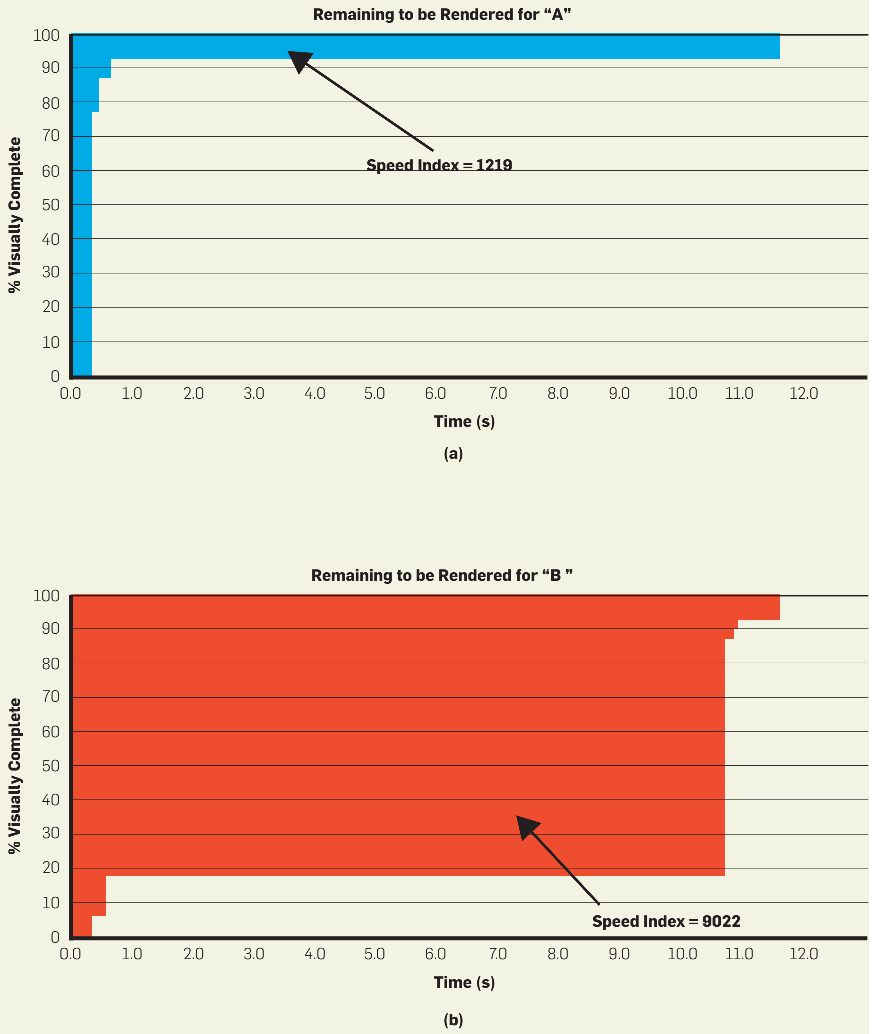


Figure 6. Speed index value.



(a)

(b)

is available only in lab environments where the visual progress of a page loading can be recorded.

Speed index.[14] After experimenting with several point-in-time measurements, we evolved our visible measurement frameworks to capture the visual progress of displaying a page over time. This current best effort for representing the user experience in a single number takes the visual progress as a page is loading and calculates a metric based on how quickly content is painted to the screen. Figure 5 shows the example from Figure 4 but with visual progress included each step of the way.

Figure 6 shows how to plot the progress and calculate the unrendered area over time to boil the overall experience down to a single number. The speed index is an improvement in the ability to capture the user experience, but it is only measurable in a lab. It still requires that you determine what the endpoint of the measurement is, and it does not deal well with pages that have large visual changes as they load.

▸ *Custom measurements.* Nothing beats application-specific knowledge and measurements. While there is no perfect solution for measuring the user experience generically, each application owner can instrument specific performance points.

As browsers go through a page and build the DOM, they also execute inline scripts when they get to them. They cannot execute the scripts until all previous scripts have been executed. If you want to know when the browser made it to the main article, then you can just add a tiny inline script that records the current time right after the HTML for the main article. When you report the performance metrics, you can gather the different custom measurements and include them as well.

The W3C Web Performance Group considered this use case as well and created the User Timing specification.[13] It provides a simple way of marking points in time using `performance.mark("label")` and a standard way of querying all of the custom measurements later. This is more for standardization and ease of use than anything else since you could always store the timings in your own JavaScript variables, but by using the standard interfaces, you also make it possible for third-party reporting tools to report on your custom metrics.

This still isn't perfect because there is no guarantee the browser actually painted the content to the screen, but it is a good proxy for it. To be even more accurate with the actual painting event, the animation timing interface can request a callback when the browser is ready to paint through `requestAnimationFrame()`.[12]

Per-Resource Timings

One specification the community has been eager to see implemented is the resource timing interface.[11] It exposes timing information about every network request the browser had to make to load a page and what triggered the resource to be requested (whether stylesheet, script, or image).

Some security provisions are in place to limit the information that is provided across domains, but in all cases the list (and count) of requests that were made on the network are available, as are the start and end times for each of them. For requests where you have permission to see the granular timing information, you get full visibility into the individual component timings: DNS lookup, time to connect, redirects, SSL (Secure Sockets Layer) negotiation time, server response time, and time to download.

By default the granular timing is visible for all of the resources served by the same domain as the current page, but cross-domain visibility can also be enabled by including a "Timing-Allow-Origin" response header for requests served by other domains (the most common case being a separate domain from which static content can be served).

You can query the list of resources through:

```
performance.getEntriesByType:
   var resourceList
   = performance.getEntriesBy
   Type("resource");
   for (i = 0; i < resourceL-
   ist.length; i++)
   ...
```

The possibilities for diagnosing issues in the field with this interface are huge:

- Detect third-party scripts or beacons that intermittently perform poorly.
- Detect performance issues with different providers in different regions.
- Detect the most likely cause of a slow page for an individual user.
- Report on the effectiveness of your static resources being cached.
- And a whole lot more...

The default buffer configuration will store up to 150 requests, but there are interfaces available to:

- Adjust the buffer size (`performance.setResourceTimingBufferSize`).
- Clear it (`performance.clearResourceTimings`).
- Get notified when it is full (`onresourcetimingbufferfull`).

Theoretically, you could report all of the timings for all of your users and be able to diagnose any session after the fact, but that would involve a significant amount of data. More likely than not you will want to identify specific things to measure or diagnose in the field and report on the results only of the analysis that is done in the client (and this being the Web, you can change and refine this as needed when you need more information about specific issues).

Join the Effort

If you run a website, make sure you are regularly looking at its performance. The data has never been more readily available, and it is often quite surprising.

The W3C Web Performance Working Group has made great progress over the past two years in defining interfaces that will be useful for getting performance information from real users and for driving the implementation in the actual browsers. The working group is quite active and plans to continue to improve the ability to measure performance, as well as standardize on solutions for common performance issues (most notably, a way to fire tracking beacons without holding up a page).[5] If you have thoughts or ideas, I encourage you to join the effort, to join the mailing list[10] and share them with the working group.

Related articles on queue.acm.org

High Performance Web Sites
Steve Souders
http://queue.acm.org/detail.cfm?id=1466450

Building Scalable Web Services
Tom Killalea
http://queue.acm.org/detail.cfm?id=1466447

Improving Performance on the Internet
Tom Leighton
http://queue.acm.org/detail.cfm?id=1466449

References

1. Brutlag, J., Abrams, Z. and Meenan, P. Above-the-fold time: measuring Web page performance visually (2011); http://cdn.oreillystatic.com/en/assets/1/event/62/Above%20the%20Fold%20Time_%20Measuring%20Web%20Page%20Performance%20Visually%20Presentation.pdf.
2. Google Analytics (2012); Measure your website's performance with improved Site Speed reports; http://analytics.blogspot.com/2012/03/measure-your-websites-performance-with.html.
3. Google Developers (2013); User timings – Web tracking (ga.js); https://developers.google.com/analytics/devguides/collection/gajs/gaTrackingTiming.
4. Hallock, A. Some interesting performance statistics; http://torbit.com/blog/2012/09/19/some-interesting-performance-statistics/.
5. Mann, J. W3C Web performance: continuing performance investments. IEBlog; http://blogs.msdn.com/b/ie/archive/2012/11/27/w3c-web-performance-continuing-performance-investments.aspx.
6. Souders, S. The performance golden rule; http://www.stevesouders.com/blog/2012/02/10/the-performance-golden-rule/.
7. W3C. High Resolution Timing (2012); http://www.w3.org/TR/hr-time/.
8. W3C. HTML5 (2012); http://www.w3.org/TR/2012/CR-html5-20121217/webappapis.html#handler-window-onload.
9. W3C. Navigation timing (2012); http://www.w3.org/TR/navigation-timing/.
10. W3C. Public-web-perf@w3.org mail archives (2012); http://lists.w3.org/Archives/Public/public-web-perf/.
11. W3C. Resource timing (2012); http://www.w3.org/TR/resource-timing/.
12. W3C. Timing control for script-based animations (2012); http://www.w3.org/TR/animation-timing/.
13. W3C. User timing (2012); http://www.w3.org/TR/user-timing/.
14. WebPagetest documentation; https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.
15. Web Performance Working Group. W3C; http://www.w3.org/2010/webperf/.

Patrick Meenan has worked on Web performance in one form or another for the last 12 years and currently works at Google to make the Web faster. He created the popular open source WebPagetest Web performance measurement tool, runs the free instance of it at http://www.webpagetest.org/, and can frequently be found in the forums helping site owners understand and improve their website performance. He also helped found the WPO foundation, a non-profit organization focused on Web Performance Optimization.