In-Browser Network Performance Measurement

Eric Gavaletz
The University of North Carolina
gavaletz@cs.unc.edu

Dominic Hamon
Google
dominic@google.com

Jasleen Kaur
The University of North Carolina
jasleen@cs.unc.edu

ABSTRACT - When looking for an excellent platform for conducting end-to-end network performance measurement that is large-scale and representative, researchers should look no further than the browser---after all, browsers are installed everywhere and are used multiple times per day by most Internet users. In this work, we investigate the use of the DOM, XHR and Navigation Timing API for measuring HTTP response times within browsers, with the goal of estimating path latency and throughput. The response times are measured using a set of popular browsers in a controlled environment---this helps us isolate the differences between the browsers as well as study how closely the measurements match the ground truth. We show that, in general, the XHR method yields the most consistent measurements across browsers, but that the new Navigation Timing and the proposed Resource Timing APIs could change that. We also use the measurements from our controlled environment to study the impact of each of our investigated measurement methods on a hypothetical measurement study.

Why the W3C Workshop on Web Performance?

Existing efforts that perform network performance measurements in the browser rely on additional resources such as Flash, Java, and browser extensions. Given recent security and availability concerns, however, each of these enables only limited user participation. We hope that by working with the Web Performance Working Group, measurement projects such as ours can motivate increased cross-browser accuracy for timing APIs. This will greatly benefit Internet measurement efforts--- equally importantly, it provides application developers a best practice for understanding their applications' performance data and the confidence that the data is not plagued by timing difference across browsers.

Introduction

Why network performance measurements?

Measurement data on how well the Internet works can aid several communities in making informed decisions regarding the design and regulation of the Internet. With the growth of broadband access networks and increasing consumer demand for fast, reliable and economical service, savvy consumers are relying more on network performance measurement to understand whether they are getting what they pay for. For content and service providers, it is fiscally important that the content gets to the user in an expedient manner---designs based on well understood empirical data seem to work well in practice. Increasingly, government and regulatory agencies are focussing on providing fast reliable Internet access to everyone (and not just the tech-savvy)---measurement data to be used in these efforts must be verifiable and reliable in order to provide any kind of credible claims to these constituencies. The goal of our research is to further the use of the browser as a platform for conducting large-scale and representative end-to-end network performance measurements on the Internet.

Why the browser?

A browser is the platform of choice for conducting large-scale network measurements for several reasons. First, the browser does not require reluctant users to install new software---it comes pre-installed on all user devices and represents a huge user-base. Second, browsers enforce strict sand boxing policies---standalone measurement software, on the other hand, presents portability as well as security challenges. Third, recent studies have also shown dramatic growth in HTTP and web traffic[1] (83% annualized growth[2]), indicating that to understand network performance from the browser's perspective is to understand how it affects the majority of users. In short, for most users the browser is the Internet---if we want a measurement tool that is useful and accessible to most people, then we have to do a good job in the browser. We have to move beyond asking users to visit an arcane website requiring plugins, or do anything that's not "just there" giving all consumers the opportunity to understand if they are getting what they pay for. With more users we get a better informed public and more complete data on the Internet - everyone wins.

Existing Approaches (Flash, Java, and Extensions)

Arguably the most popular tool for measuring network performance in the browser is the Ookla Speed Test[3] which makes heavy use of Flash to perform their measurements. Before the introduction of iOS, Flash was considered to be as ubiquitous, and until recent advancements in HTML5 it was considered a necessary component of many web-applications. Even though Ookla seems to have the attention of most consumers there are a number of tools that have gained popularity among more technical audiences. The most notable of these being the ICSI Netalyzr[4] project, and a handful of the Measurement Lab tools. Netalyzr has been very successful in reaching users (259,000 tests as of August 2011) despite the need to run their tool as a Java applet and request elevated system permissions from their users. Measurement lab has also had similar success with Java based tools running in the browser, but both groups have seen the writing on the walls - Java's days of running in the browser may be limited.

The popularity of iOS with its lack of Flash support (Android is also not supporting Flash in its more recent releases), the growing security concerns with Java Applets causing many vendors to disable Java in the browsers by default[5] and the general shift away from extensions and plugins in favor of HTML5 has prompted researchers to look for alternative methods[6] for in browser network performance measurement. The Netalyzr developers in particular have be working on reducing their Java dependency by developing a browser extension called Fathom[7] that is intended to fill the voids left between Java and JavaScript.

[1] Maier, Gregor et al. "On dominant characteristics of residential broadband internet traffic." *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference* 4 Nov. 2009: 90-102.

[2] Erman, Jeffrey et al. "Network-aware forward caching." *Proceedings of the 18th international conference on World wide web* 20 Apr. 2009: 291-300.

[3] "Ookla | Speedtest." 2006. 28 Oct. 2012 <http://www.ookla.com/speedtest>

[4] Kreibich, Christian et al. "Netalyzr: illuminating the edge network." *Proceedings of the 10th annual conference on Internet measurement* 1 Nov. 2010: 246-259.

[5] "Apple gets aggressive – latest OS X Java security update rips out ..." 2012. 29 Oct. 2012 <http://nakedsecurity.sophos.com/2012/10/18/apple-gets-aggressive-latest-os-x-java-security-update-rips-out-browser-support/>

[6] Janc, Artur, Craig Wills, and Mark Claypool. "Network Performance Evaluation in a Web Browser." *Parallel and Distributed Computing and Systems* Nov. 2009.

[7] Dhawan, Mohan et al. "Fathom: A Browser-based Network Measurement Platform." (2012).

Fathom has the promise of being ported to run on multiple browsers, but it is currently only available for Firefox (only about 25% of users[8]). In our research, we are developing a fully JavaScript based measurement tool called net-score[9].

net-score

The net-score project seeks to provide an ordinal scoring system for network performance that is more general than simply looking at throughput or latency. Given the dynamic and evolving nature of broadband access speeds our scoring is norm-referenced and as such requires a representative population sample for establishing that norm. In an attempt to reduce selection bias by requiring 3rd party plugins or special permissions we are attempting to gather performance data using only JavaScript and native browser features. In addition by not requiring that users install third-party plugins we can significantly reduce the "geek bias" experienced by other tools[10] by embedding our performance tests in third-party pages that are popular with portions of the population that might not otherwise be interested in network performance measurements.

The development of a JavaScript-only measurement client has been predictably challenging due to differences across browsers. For network measurement, we would ideally like everyone to use the exact same browser down to the version number. Since we don't live in an ideal world, our more modest goal is that measurements across browsers be reasonably consistent. Indeed, we would like our data to say more about performance of the network than what browser was used to measure it. The remainder of this paper will describe these challenges, and is organized as follows. First we investigate measuring response times based on DOM requests, XHR requests, and using the Navigation Timing API, and discuss the need for the Resource Timing API. Then we apply the data gathered about difference in browsers to see how it would effect a hypothetical research study, and discuss how network performance measurement projects could make use of some of the proposed performance APIs. Finally we provide our concluding remarks.

Measuring response times

Measuring response times in theory is very basic. We record the time immediately before and after downloading an object. This is no different in the browser, and the most accurate method for determining the response time is going to give us times that are very close to when the first byte of an object is received and when the last byte is received. One way of doing this would be using the JavaScript date API and the DOM to load the objects. The obvious advantages to fetching objects using the DOM are: (i) the technique is extremely simple, (ii) it is not restricted to same origin[11] servers, and (iii) it is supported in all browsers. Disadvantages are that we are not given any insight into the details of the loading process meaning that between the time we set the source attribute of the object and the time it is completely loaded we are in the dark. An alternative to using the DOM is to use XHR to request the objects directly from JavaScript. The tradeoffs with using this approach are that JavaScript is bound by same origin policies that don't apply to the DOM, but we are given much more insight into the intermediate states that the request goes through. The last alternative approach that we will discuss is the Navigation Timing API. The Navigation API was designed to give developers even more insight into the loading process than even the XHR method. It provides details about when a connection is established, how long it took to make the

[8] "StatCounter Global Stats - Browser, OS, Search Engine including ..." 2009. 28 Oct. 2012 <http://gs.statcounter.com/>

[9] "net-score.org." 2005. 28 Oct. 2012 <http://www.net-score.org/>

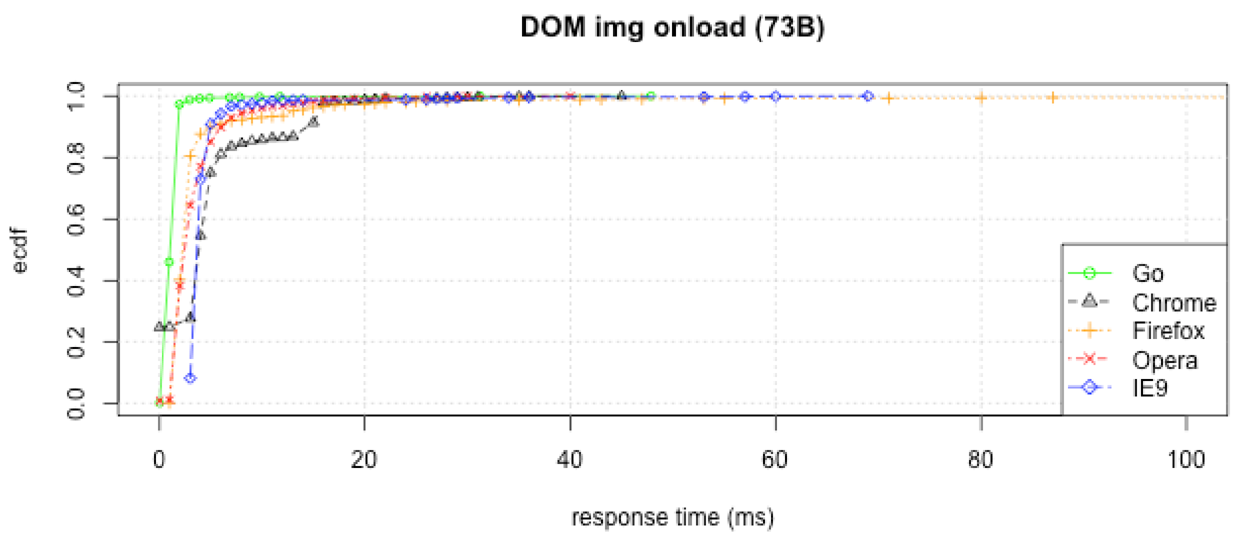
[10] Kreibich, Christian et al. "Experiences from Netalyzr with engaging users in end-system measurement." *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack* 19 Aug. 2011: 25-30.

[11] "HTML Standard - Origin." 27 Oct. 2012 <http://www.whatwg.org/specs/web-apps/current-work/multipage/origin-0.html>

request as well as how long the request took to fulfill. Being a relatively new API the implementations across various browser have some inconsistencies, and it is only implemented in few modern browsers with little chance of being back ported to legacy systems.

Document Object Model[12] (DOM)

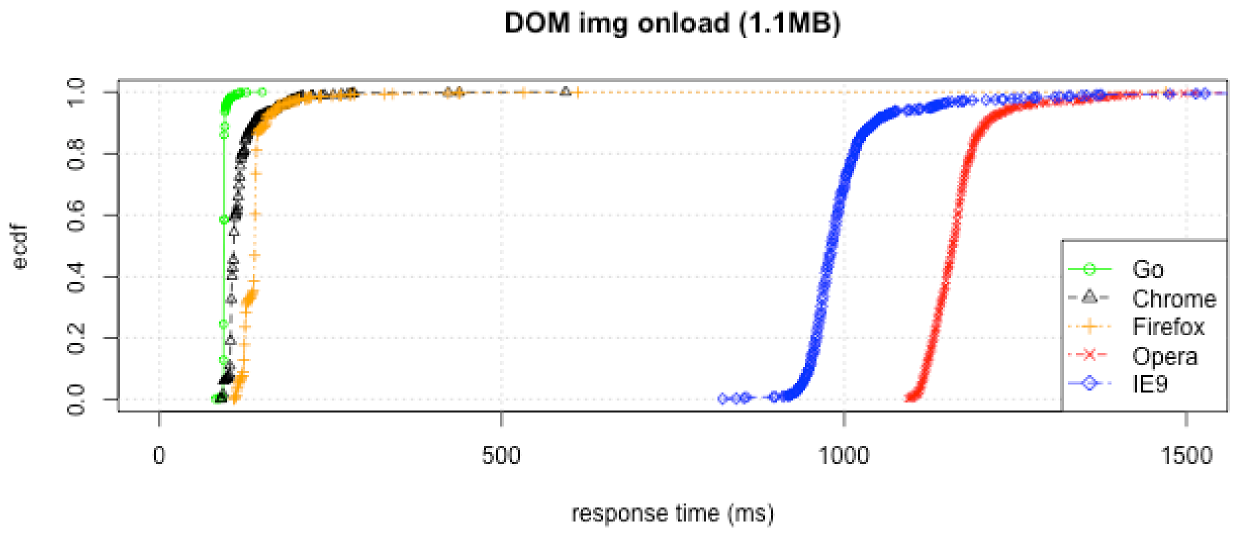
First we look at response times measured with the `onload` event fired by loading images by setting the src attribute of an img DOM tag. We use "tiny" objects (73 B) for measuring latency and "large" objects (1.1 MB) for measuring throughput. We run the experiment 10 times for each browser for a total of 1,000 response time measurements [example code]. We look at tiny and large objects to estimate the latency and capacity of the path respectively. Requests for single packet objects, on average, should complete in one round trip time (RTT). Knowing the expected RTT we can then use larger objects that would require many packets and subtract the expected RTT from the response time for a large object giving us the expected transfer time. Dividing the objects size by the transfer time can give us the throughput for the path. Here we show a comparison of response times for tiny objects across browsers.



The results for tiny objects are what we would hope to see[13], but the response times for the large objects (below) show that the values are highly dependent on the browser. In particular Internet Explorer and Opera show response times that are orders of magnitude larger than Chrome, Firefox, and our ground truth.

[12] "DOM - W3C." 27 Oct. 2012 <http://www.w3.org/DOM/>

[13] Given how closely Chrome performed to our ground truth in other experiments we are a bit surprised that the response times are high in the tiny object case, and that they have a strange step around 15 ms. This was not an isolated case as we were able to repeat this result in multiple runs across multiple setups.



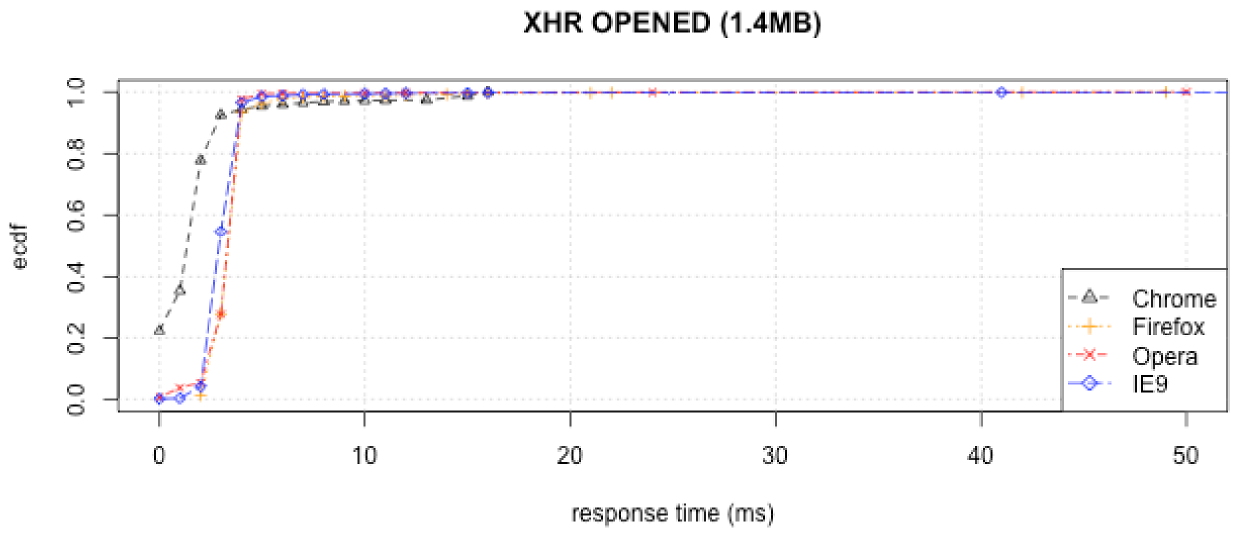
Given the less than ideal results for large files we looked at using XMLHttpRequest as an alternative for downloading the objects and measuring response times.

XMLHttpRequest[14] (XHR)

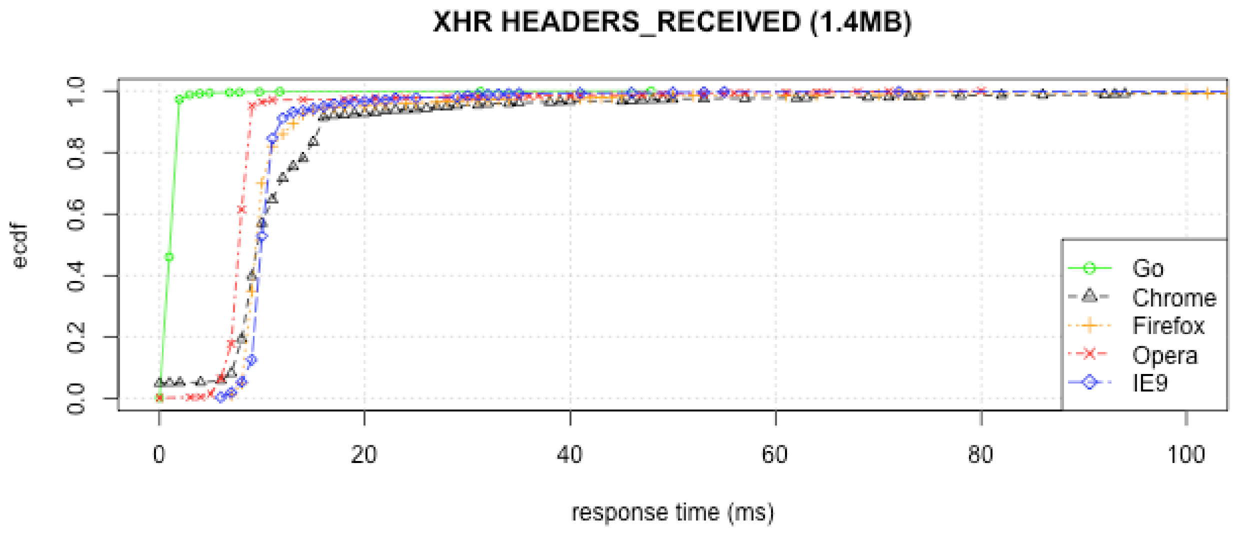
One major drawback with using XHR is that it is bound by same origin policies. We workaround this limitation by making use of cross-origin resource sharing (CORS), but we are then limited to downloading from servers where we can set the necessary HTTP headers. Luckily CORS is supported in most browsers (an estimated 91%[15]), and with the help of generous resources provided by Measurement Lab[16] we can still download from servers that are relatively close to end-users. The code used for these tests is similar to the DOM test with the major difference being the use of a XHR object in place of the DOM image tag [example code]. Since JavaScript does not deal well with binary data, the image that we used in our DOM tests above was base64 encoded and transmitted as text. Due to encoding inefficiencies the object size is 1.5 MB (as compared with the same png image with a binary encoding size of 1.1 MB), but the ground truth in the plots accounts for the extra bytes transferred.

The XHR standard specifies five possible states for the object: `UNSENT`, `OPENED`, `HEADERS_RECEIVED`, `LOADING`, and `DONE` which are represented by their numerical values 0-4 respectively. For our work the `UNSENT` state is time zero, and we begin by looking at the `OPENED` state where as expected there is very little variation between browsers. The time required to get to this state is of interest as it may be an indicator of processing overhead that can be accounted for later.

[14] "XMLHttpRequest Level 2." 2006. 27 Oct. 2012 <http://www.w3.org/TR/XMLHttpRequest/>
[15] "Can I use... Support tables for HTML5, CSS3, etc." 2009. 27 Oct. 2012 <http://caniuse.com/>
[16] "M-Lab | Welcome to Measurement Lab." 2009. 27 Oct. 2012 <http://www.measurementlab.net/>



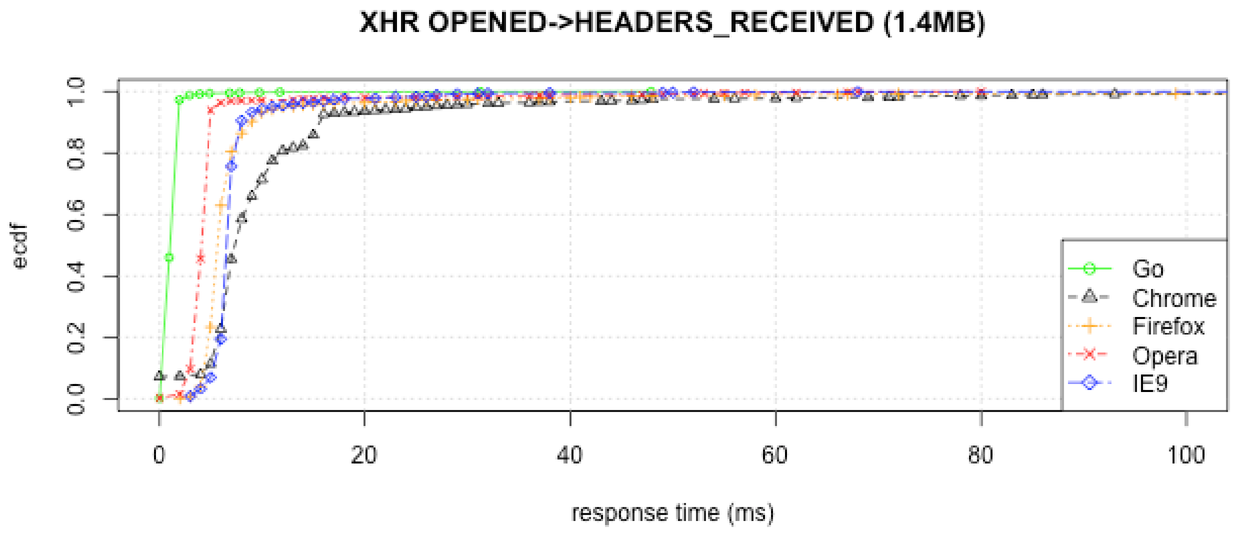
Once the request has been sent to the server and the response headers have been received at the client the XHR object enters the `HEADERS_RECEIVED` state. Since we are using a persistent HTTP connection and make many requests reusing that persistent connection, the majority of those requests will reach this point after one RTT eliminating the need to request tiny objects to estimate the connection's latency. For this reason we are comparing these values to the response times for tiny objects as a ground truth. What we see is that the values obtained in the various browsers are slightly higher, and can likely be attributed to HTTP header processing overhead that we have avoided in our Go client. The results here are similar to what we saw for the tiny DOM requests, and we even see the same strange artifact for values less than 15 ms for Chrome.



Since the time between the `OPENED` state and the `UNSENT` state is our best estimate of any extra processing on the client[17] (a loose lower bound if the HTTP headers are inspected before the event is fired) we can improve our latency estimate by subtracting the `OPENED` time from the `HEADERS_RECEIVED` time.

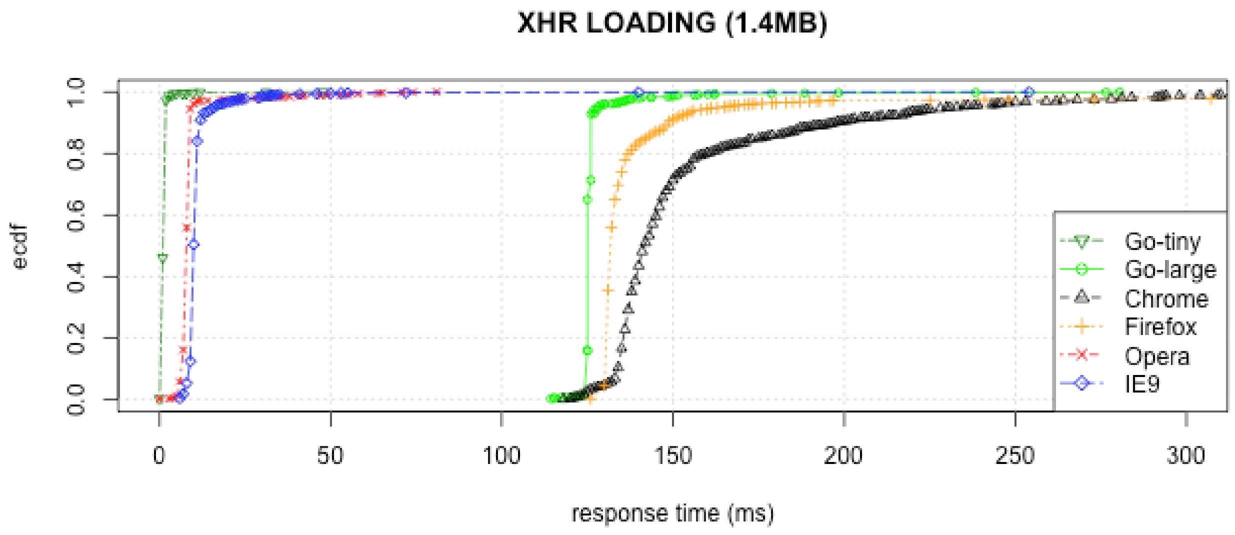
[17] In our case we call `send()` immediately following the call to `open()`.

This does have the effect of both bringing our estimate closer to the ground truth, as seen in the following plot.

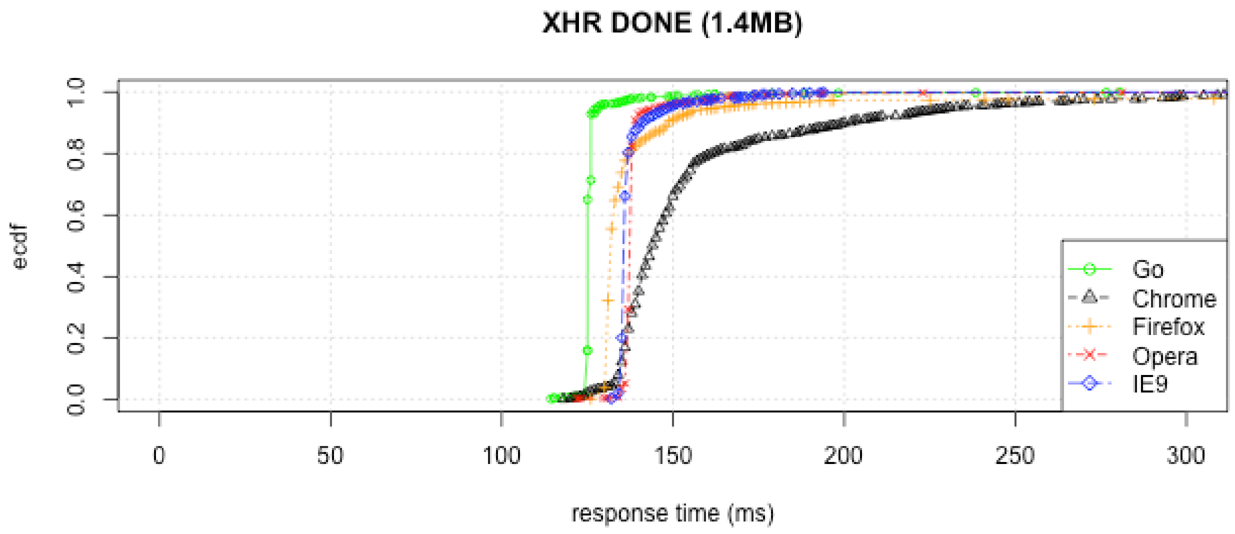


The specification[18] seems clear that the `LOADING` event should be fired when "the response entity body is being received" where the beginning of the response entity body presumably resides in the same packet delivering the response HTTP headers (this is true in our experiments). Then it seems reasonable that a time for `LOADING` would be the time of `HEADERS_RECEIVED` plus the time required to process the HTTP headers (this being very little overhead in our case) meaning that it should be close to `HEADERS_RECEIVED`. We find this is true for both Opera and IE9, but both Chrome and Firefox seem to have interpreted the specification differently. It may be that the `LOADING` name may be responsible for the differences seen between browsers as *RECEIVING* is more descriptive of the state as specified. If this `LOADING` value were more reliable it could provide a better reference for improving our estimation of the amount of time between receiving the first byte of the object and the last (this duration being used to estimate throughput). Since the accuracy of our latency estimation bounds the accuracy of all our other estimates, it is disheartening that we cannot confidently use the `LOADING` value.

[18] "XMLHttpRequest Level 2." 2006. 27 Oct. 2012 <http://www.w3.org/TR/XMLHttpRequest/>



The XHR specification tells us that once the data transfer has been completed that the object transitions to the `DONE` state. Here we see again that where Chrome performed relatively well in respect with being closer to the ground truth in our DOM tests, it actually lags behind here. It is worth mentioning here again that in our experiments there were no other requests being made from our page, no other tabs open, no added extensions, and no contention for network resources on the operating system.



Navigation Timing

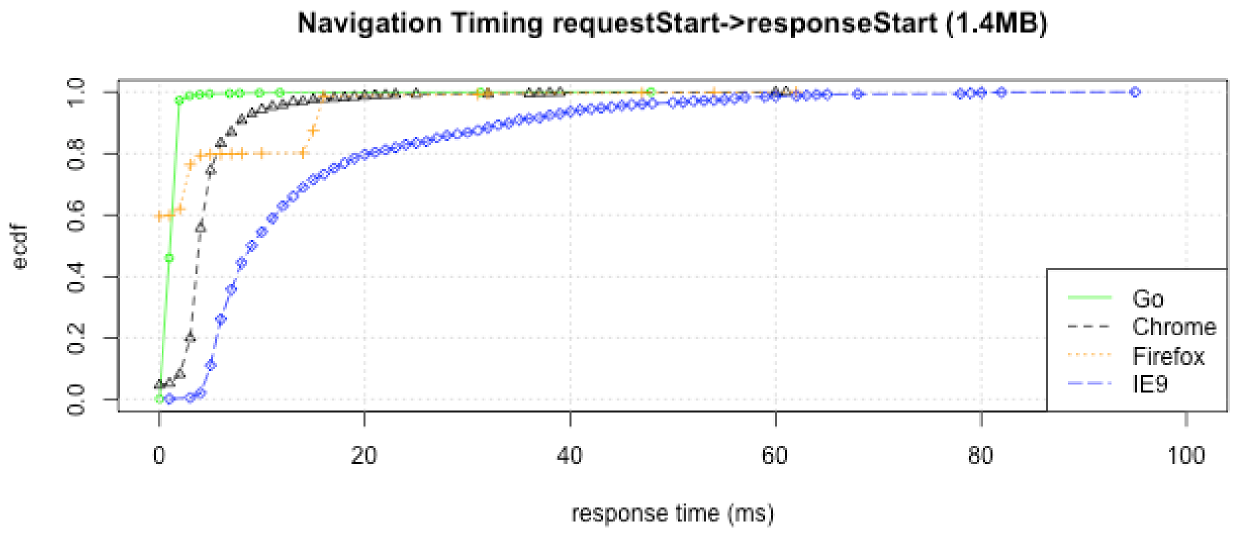
The Navigation Timing API[19] provides a wealth of information about how network performance might impact loading times. When measuring page load times using the `onload` event (from an iframe) we were left wondering what portion of the time between setting the `src` of the iframe and the firing of the onload event was due to network delays - now we can look at delay due to network performance explicitly. We hope that the analysis of navigation timing data will augment existing performance measures giving us a more

[19] "Navigation Timing - W3C." 2010. 27 Oct. 2012 <http://www.w3.org/TR/navigation-timing/>

complete understanding of how performance may vary and if there are important factors that existing tools are not accounting for in their analysis. A significant drawback in using the Navigation Timing API is that it is only available in roughly 64% of clients.[20]

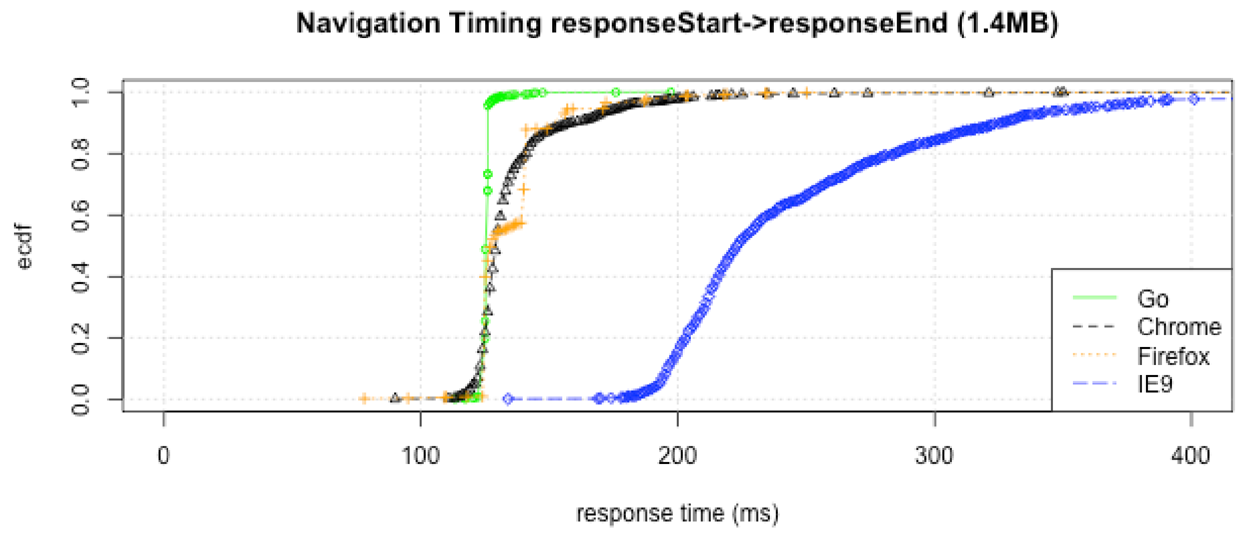
Our page that we are measuring is really a base64 encoded png image with a little HTML and JavaScript padding (945 Bytes to be exact). The result is that we are approximating a self-timing object by measuring a page that mostly consists of the single object (%99.93). The parent document code allocates an iframe object and sets the `src` attribute to our specially crafted page. Once the page has finished loading it will call the `recordTiming` function passing the `window.performance.timing` object.

The code loaded into the iframe simply waits for the `onload` event, and then passes the `window.performance.timing` object to the recording function in the parent document. Code in the parent document maintains state for the experiment. We load the iframe by setting the `src` attribute from the parent document initially and then then by setting the `self.location` attribute for successive loads (we found this mitigates the memory problems we see when setting the `src` attribute many times in Firefox). The example code for the parent document is slightly simplified as we did also record the response times indicated by the `onload` event in addition to the values provided by the `window.performance.timing` object. Just as with our previous experiments we do 10 trials of 100 requests each for a total of 1,000 data points for each browser. Our analysis is focusing on the response times for objects so we are using the difference between the `responseEnd` and `responseStart` attributes. Just like we looked at the `HEADERS_RECEIVED` state for XHR methods as an indicator of latency on the path we investigate the use of `responseStart` less `requestStart`.



What we see is a substantial improvement for Chrome both in terms of being closer to the ground truth and we have a nice distribution without the jump around 15 ms. Unfortunately using the navigation timing API yields worse results than the DOM and XHR tests for both Firefox and IE9 (where we compare to latency estimations). In Firefox's case the unusual steps around 5 ms and 15 ms show up in a similar fashion in the response timings, and IE9 shows the same trend of elevated times and a relatively heavy tailed distribution.

[20] "Can I use... Support tables for HTML5, CSS3, etc." 2009. 27 Oct. 2012 <http://caniuse.com/>

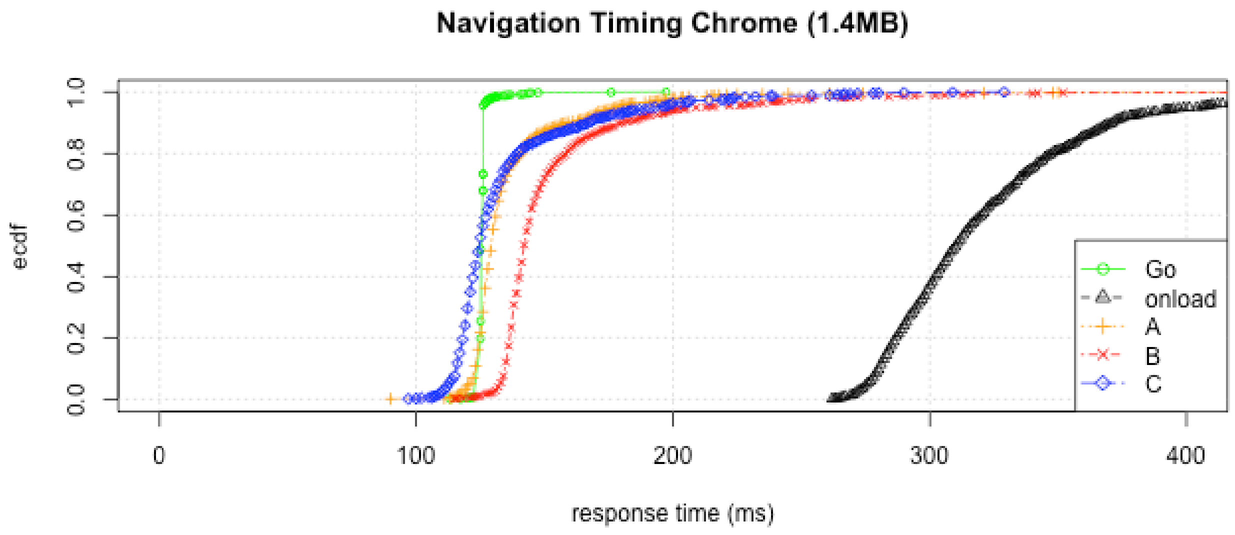


While not apparent in these plots, IE9 and Firefox are not alone in having problems with the Navigation Timing API. According to the Navigation Timing specification, the value of the timing attributes must monotonically increase ensuring timing attributes are not skewed by adjustments to the system clock during navigation.[21] This is not always the case as Chrome has a documented bug[22] citing that there are instances where negative values appear - we dropped the negative values and the remaining values seem to be reasonable. In addition to getting rid of negative outliers we also looked at the case where placing the code to record the navigation timing data could be placed at various points in the iframe document.

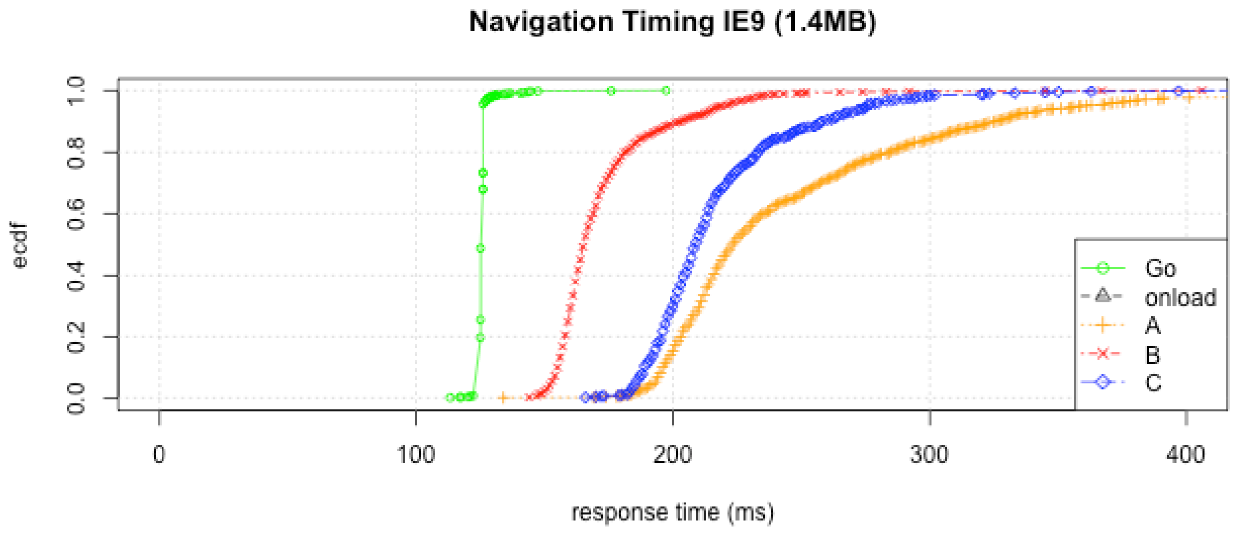
Case	**iframe document JavaScript placement/calling method**
A	The code is placed in the head of the document, and is called via the body's onload attribute.
B	The code is placed at the very end of the body, and is evaluated inline (not called as a function).
C	The code is placed at the very end of the body, and is wrapped in a function and called via the body's onload attribute.

The following plot shows that Chrome is only slightly affected by the code placement, but performs slightly better when the code is wrapped in a function and called via the `onload` event. Also of interest is that of all the browsers evaluated Chrome is the only one who's `onload` events are close enough to be visible on the same plot.

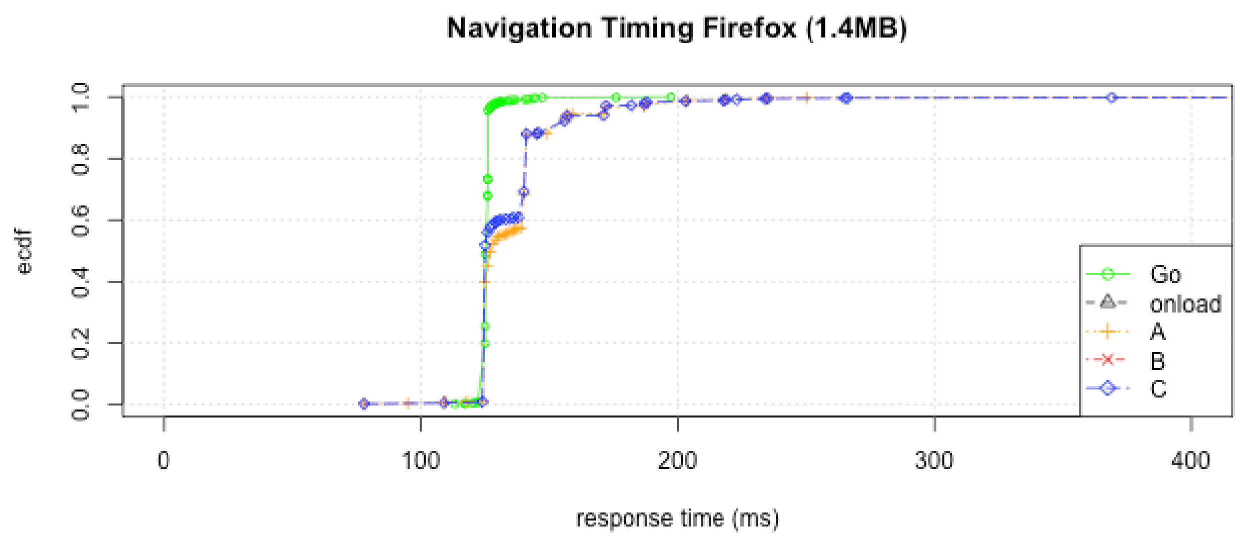
[21] "Navigation Timing." 2010. 25 Oct. 2012 <http://www.w3.org/TR/navigation-timing/>
[22] "Issue 127644 - chromium - Navigation Timing ... - Google Code." 2012. 25 Oct. 2012 <http://code.google.com/p/chromium/issues/detail?id=127644>



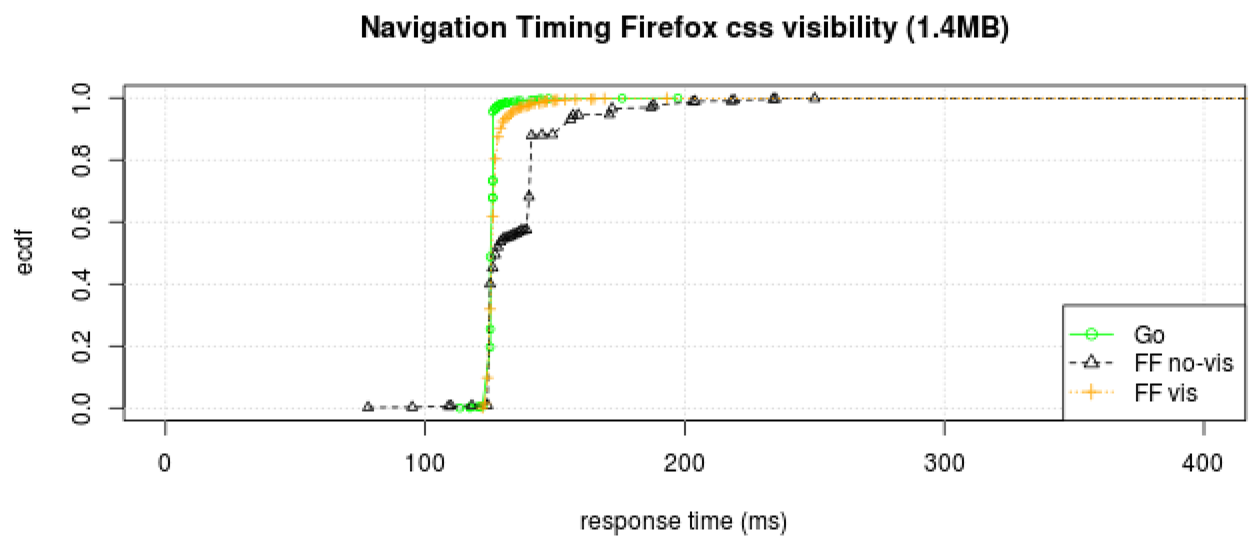
Internet Explorer on the other hand does significantly better when the code is evaluated inline, but is still much further from the ground truth than either Chrome or Firefox.



Firefox actually crashed when the code was evaluated inline as a part of the body, and saw little improvement with the code called via the onload event from the body. There are still strange steps around 125 ms and 140 ms.



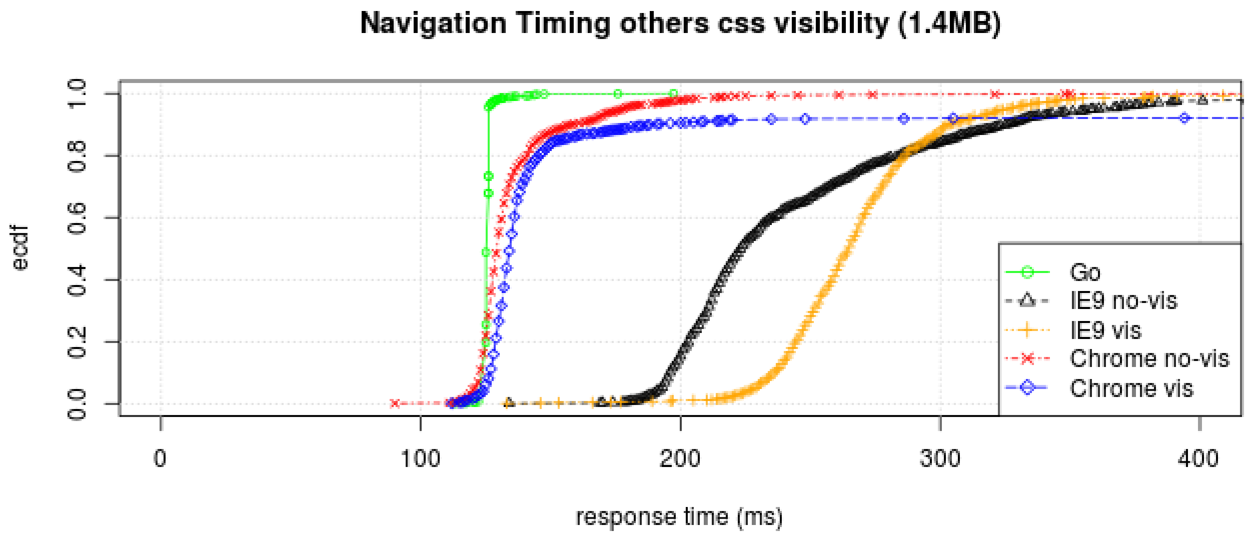
Thanks to insight from the Fathom authors[23] we looked into the possibility that the same mechanism that is used in Firefox[24] to "clamp callback timer accuracy for in-page JavaScript to low precision when the user is viewing a page in a different tab" could affect the accuracy of the numbers reported for Navigation Timing. It seemed like a long shot since our page that was being tested was not in an inactive tab, but we made a small modification to the data collection code eliminating the `i.style.visibility = 'hidden';` line making the iframe visible[25]. To our surprise the results for Firefox were drastically improved (where FF no-vis is the original version), but unfortunately this improvement did not translate to IE9 or Chrome - they were worse.



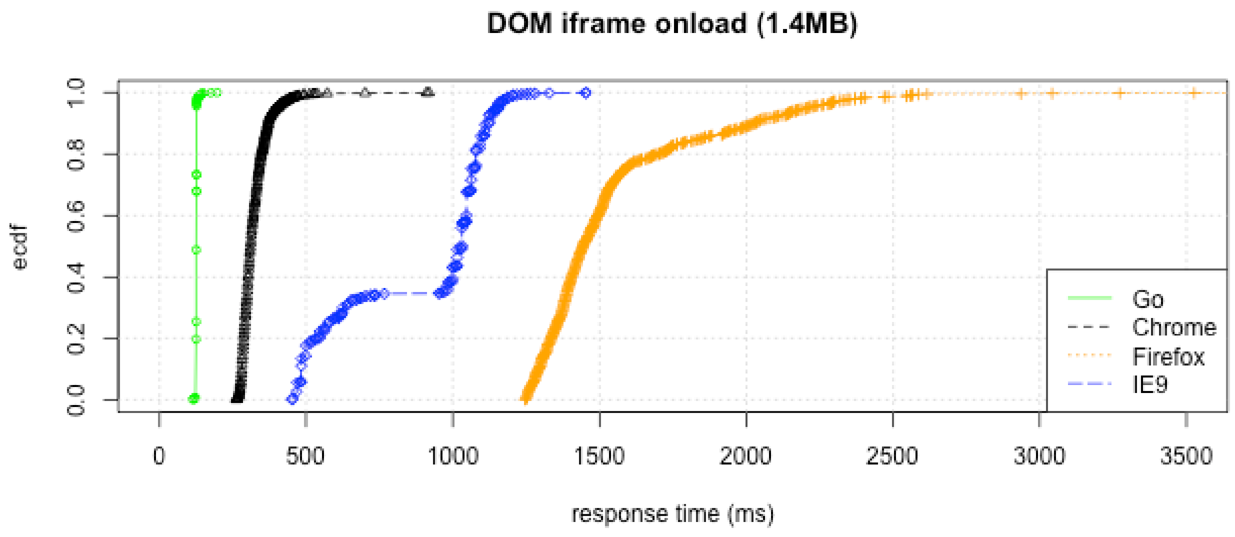
[23] Dhawan, M. "Fathom: A Browser-based Network Measurement Platform." 2012. <http://www.icir.org/vern/papers/imc077-dhawan.pdf>

[24] "window.setTimeout - Mozilla Developer Network." 2012. 28 Oct. 2012 <https://developer.mozilla.org/en-US/docs/DOM/window.setTimeout>

[25] The initial thought being that if the iframe were not visible that the browser would not expend resources displaying the object.



In this lab work what we have defined as a page (the content loaded into our iframe) is very simple and static, but browsers are being used for applications that are increasingly dynamic and blur the lines of what is and is not a page. Discussing this issue recently with a fellow researcher I raised the point that as the web as it is used today we may have outgrown the page abstraction. To illustrate this point we can look at how various browsers differ in what they report as when the "page" is loaded, and in our case this page is trivial in its simplicity when compared to modern interactive web applications (these values are from the same runs plotted with Navigation Timing values above).



Even with the adoption of the Navigation Timing specification when looking at the `responseStart` and `responseEnd` attributes we get the notion that the consequential components of a page are included in line with the primary request. Recent work[26] has shown this is not the case for modern websites where the

[26] Butkiewicz, Michael, Harsha V Madhyastha, and Vyas Sekar. "Understanding website complexity: Measurements, metrics, and implications." *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* 2 Nov. 2011: 313-328.

average page is composed of 40-100 objects fetched from at least 10 servers spread across multiple non-origin domains. Instead of trying to define what is and is not a page or when a page is done (which obviously varies based on the interpretation of what counts as loaded) we may find it more productive to focus on individual resources. While it is convenient to talk about page hits from an engineering point of view we may be better served by looking at the initial HTML fetch as just one resource in a series of resources and to time those resources individually.

Resource Timing

Timing page loads and individual objects is far from perfect. The navigation timing API is exciting, but for our application we see resource timing API[27] as a real boon for doing performance analysis in the browser. While the initial page load is important, it is the loading of objects in the background that drives many of the applications today. We feel like the Resource Timing API will hugely benefit network measurement, but given the breakdown of what constitutes a page and how dynamic and modular modern sites are we see much more utility here for all developers. For our work not having to deliver objects masquerading as documents simplifies our code, and more importantly having a specification that is implemented with as good of consistency as the XHR objects with more timing details could give us greatly increased confidence in data obtained from network performance measurement in the browser.

We demonstrated with our Navigation Timing results that even with a change as small as where the timing code was placed can cause significant variation in the results. When considering a page it is not a simple matter of timing how long it took the bytes to make it to the client. The resource is necessarily being evaluated as it is loading and this overhead is evident even for the simplest pages. We could however concentrate on solving the easier and arguably more fruitful problem of measuring the timing for individual objects.

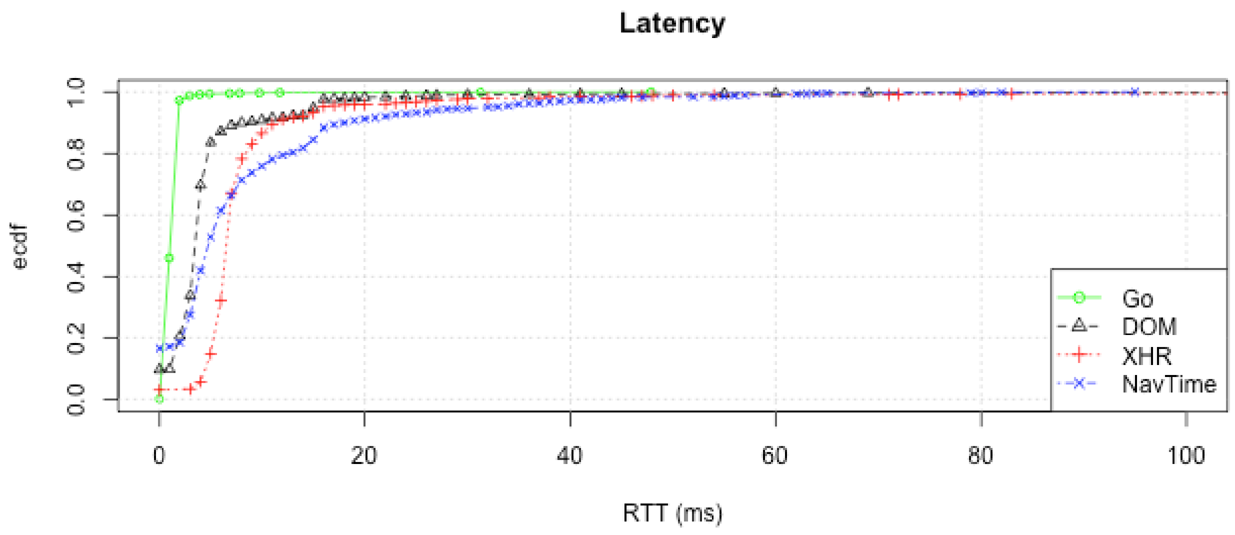
Applying the data: what if...

So what difference would this make in a measurement study? If our lab link was a representative access network path, and the distribution of browsers for that network was representative of the global browser market share then we can randomly (uniformly) sample our lab data weighted by the browser market share[28] (since we don't have Safari data we renormalized the data around 93% removing Safari's 7% market share).

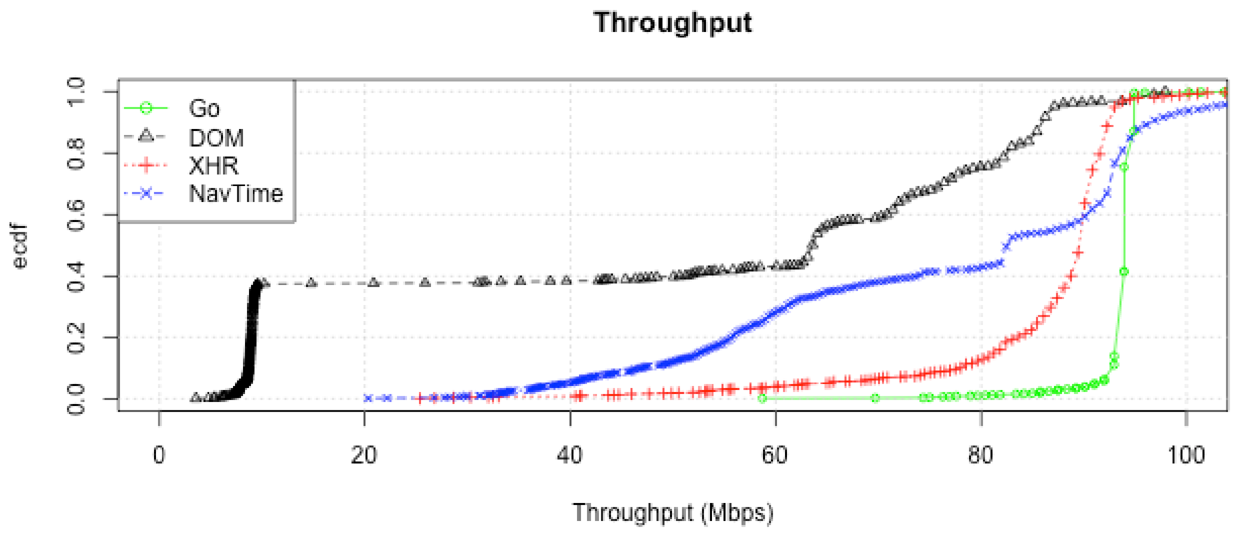
Browser	**Sample weight**	**Personal Best Latency Method**	**Personal Best Throughput Method**
Chrome	35%	Navigation Timing	DOM
Firefox	24%	DOM	XHR
IE9	35%	DOM	XHR
Opera	2%	XHR	XHR

These hypothetical results in the plots labeled "Latency" and "Throughput" below assume that one method was used across all browsers. We see that with a realistic mix of browsers that the DOM method is the most promising method for accurately measuring the latency of a path.

[27] "Resource Timing - W3C." 2011. 27 Oct. 2012 <http://www.w3.org/TR/resource-timing/>
[28] "StatCounter Global Stats - Browser, OS, Search Engine including ..." 2009. 28 Oct. 2012 <http://gs.statcounter.com/#browser-ww-monthly-201210-201210-bar>



However, when measuring throughput using large objects the XHR method is most promising. Considering you get latency measurements for free when using the XHR method, and that those latency measurements can be used one to one with the corresponding response times you may be able to make a good argument that the XHR latency values are good enough. The Navigation Timing API does not yet appear to be a viable choice as it yields worse results for latency and throughput where the throughput results are affected by the poor latency results. The poor Navigation Timing values are the result of a large sample weight for IE9 and its inflated response times.



Potentially useful APIs

Page Visibility

Continuing in our effort to increase confidence in results obtained from in-browser performance measurements we have learned that some browsers will prioritize the network traffic for the tab that has focus, and knowing if the tab that is running the measurement tool has lost focus would be an important piece of metadata. This may indicate that the performance measurements may not have been as high as they could be and these results could be easily flagged by making use of the page visibility API.[29]

Memory and CPU Utilization

One of the problems with end-to-end measurements from consumer systems is that inconsistencies and bottlenecks at the client can skew results. Even doing our best to make our tests as efficient and lightweight as possible there will always be the situation where the machine running the test has insufficient resources creating a performance bottleneck that resides on the client and not the network. Having an API that allows us to detect and filter these cases would be quite handy.

The wealth of information that would be provided by the network information API[30] and the system information API[31] would not eliminate the possible performance constraints that come with measuring network performance on end-systems, but it would give us a real opportunity to detect and account for them in ways that are inaccessible to many native applications.

High Resolution Time[32]

Because of inconsistencies in when a particular browser fires an event, we have a margin of error that is orders of magnitude larger than the millisecond resolution provided by the existing timing methods. If events were consistent across browsers then it is plausible that a timer with greater accuracy could enable the measurement of higher bandwidth links and making more nuanced assessments afforded the more detailed data.

Conclusion

Performance measurement in the browser is hard, and if a study can work with a standalone binary client (with most likely a smaller user population) that should definitely be considered as an option first. If the measurement work has to be done in the browser, then for now the best tool for the job remains XHR, but when and if the Resource Timing API will become widely available as a more accurate replacement. To this point we feel that working on measuring the simpler case of individual objects is a much needed primitive that could be used effectively while the Navigation API matures.

[29] "Page Visibility API - W3C." 2011. 25 Oct. 2012 <http://www.w3.org/TR/page-visibility/>
[30] "The Network Information API." 2011. 27 Oct. 2012 <http://www.w3.org/TR/netinfo-api/>
[31] "The System Information API." 2010. 27 Oct. 2012 <http://www.w3.org/TR/system-info-api/>
[32] "High Resolution Time - W3C." 2012. 28 Oct. 2012 <http://www.w3.org/TR/hr-time/>

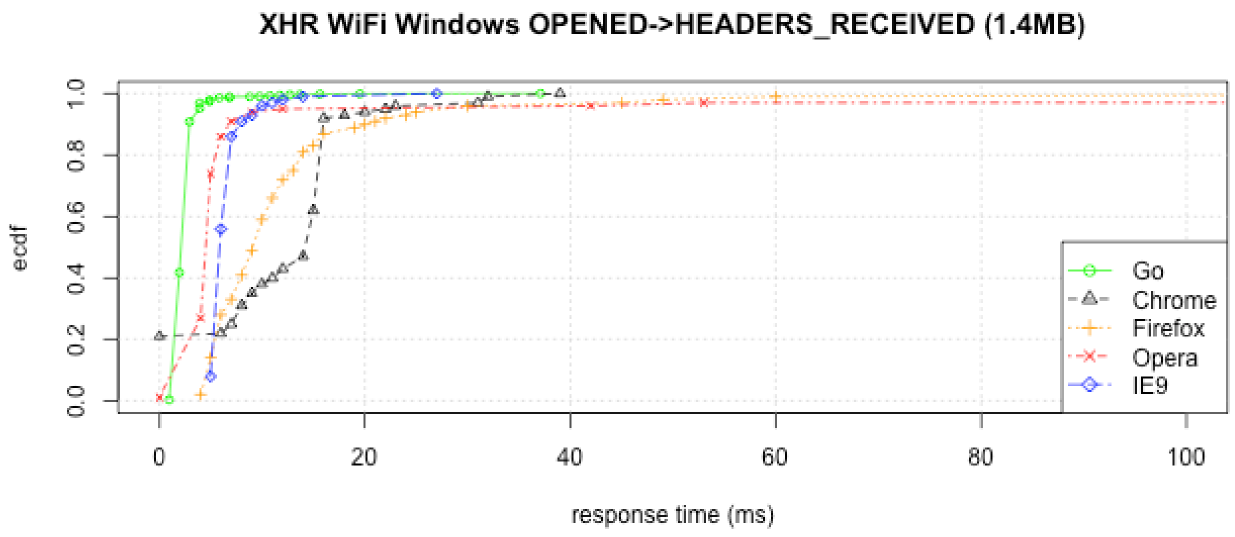
Appendices:

Mobile/Wifi

This is preliminary work where we need to do a better job of controlling confounding variables, repeat the tests with larger sample sizes, and develop a better control that measures the "ground truth" directly on the mobile device that accounting for the hardware differences but remaining independent of the browser (porting the Go client to a mobile application). It would also be nice to instrument and investigate an iOS device as well.

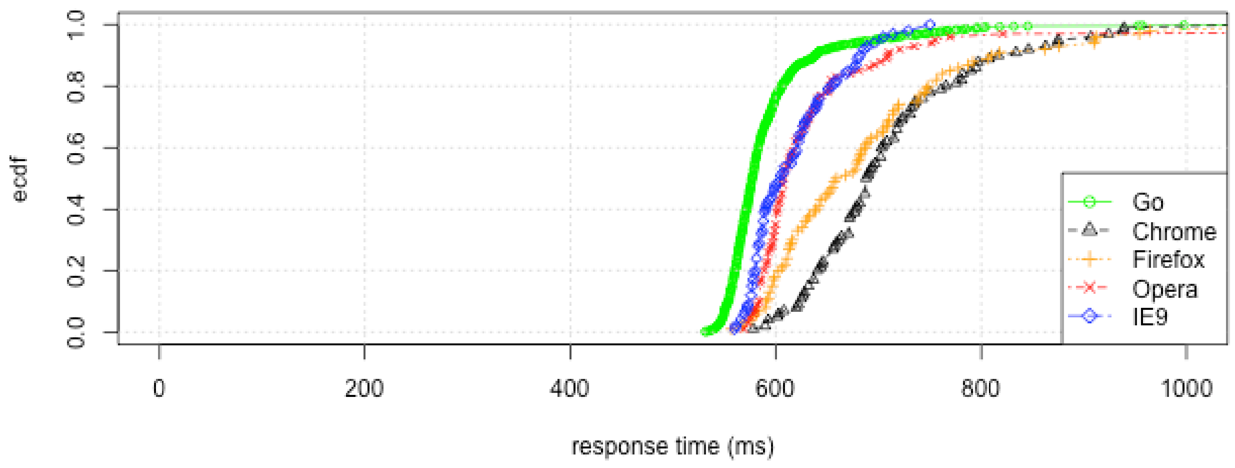
Desktop browsers over WiFi

The move to a wireless network had little effect other than a slight shift due to a higher path latency and possibly increased variability due to interference, but we see that there were a few changes. Most obvious is the exacerbation of the 15 ms Chrome behavior, and Firefox exhibits inflated values.



The total time for the response is not surprising given the lower bandwidth of wireless connections (when compared to 100Mbps wired connections). For this preliminary study we consider this as a weak control for the mobile browser tests demonstrating that the path including the wireless hop has no significant problems and produces results that agree with the wired tests.

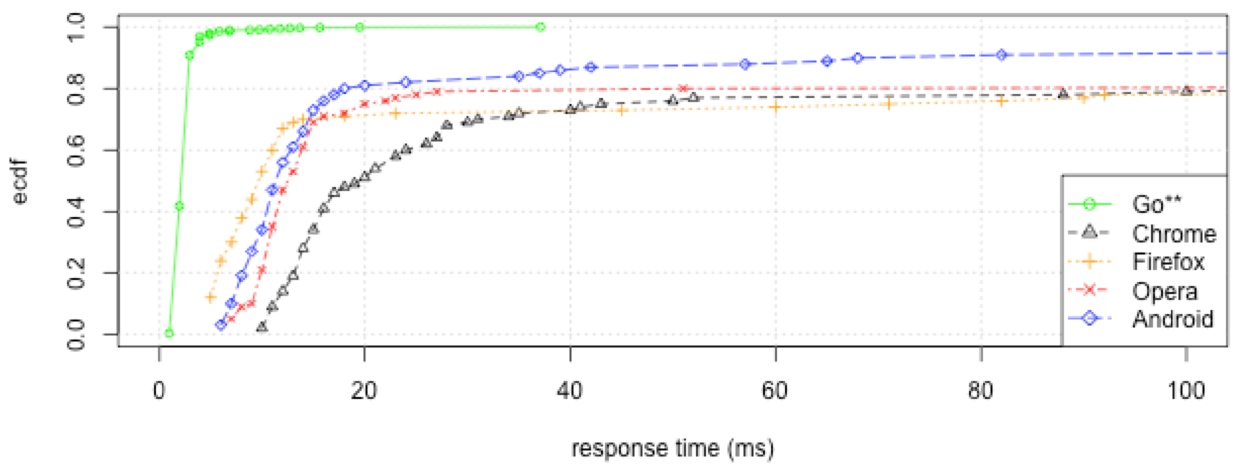
XHR WiFi Windows DONE (1.4MB)



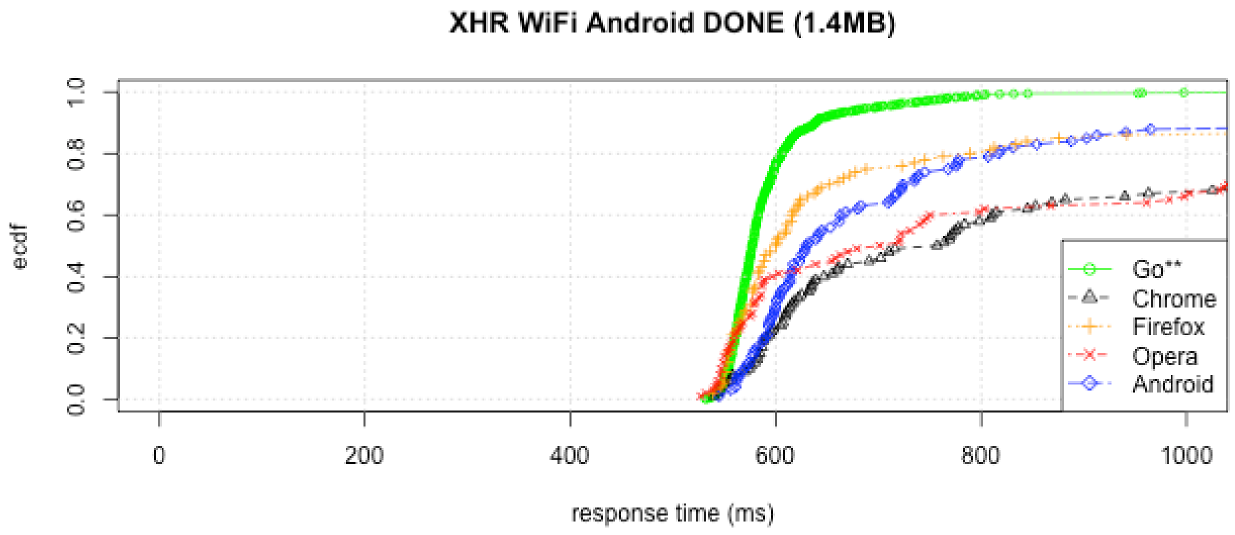
Android browsers over WiFi

As mentioned previously this work lacks a good control as we have not implemented a native application for Android in the same manner as we have with Go for Windows. We are showing the results from our Windows WiFi Go controls here for reference, but given the more constrained operating environment on the mobile device it seems reasonable that our Android control would be closer to our in-browser results.

XHR WiFi Android OPENED->HEADERS_RECEIVED (1.4MB)

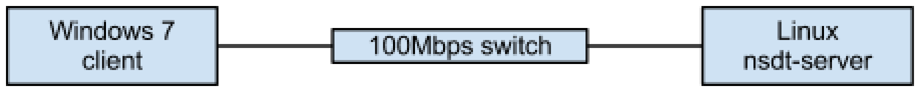


For both the plots the Android browsers maintain roughly the same characteristics as their desktop counterparts.

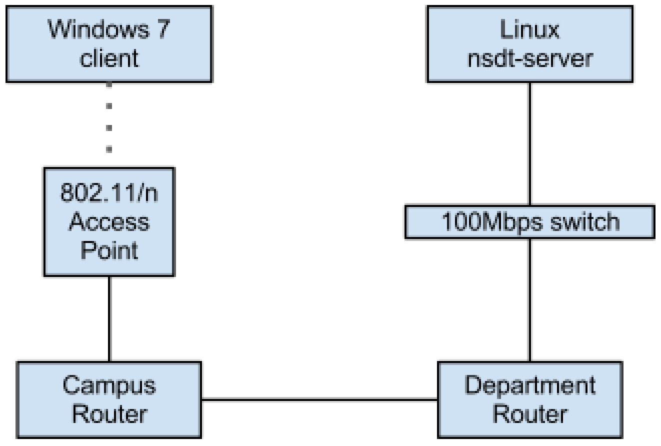


Lab environment

For these experiments we use a machine running Ubuntu 12.04 to host a custom HTTP server written in Go. Connections to this server are made across a single 100Mbps ethernet switch from our client machine running a fully patched and updated installation of Windows 7.



The primary difference for our wireless experiments is that our wireless network is on a separate VLAN requiring extra hops to reach our server. These extra hops do not significantly increase the round trip time as the difference is but 1ms for the wired path to 3ms for the wireless path.



The specific versions for each of the browsers used in this study are listed in the table below. Each has no modifications, extensions or extra plugins - they are used just as they are downloaded from the vendor. An effort to clear any cache and saved state in the browser is made between each run.

Browser	Version	User Agent
Android Chrome	18.0.1025.166	Mozilla/5.0 (Linux; Android 4.1.1; Galaxy Nexus Build/JRO03C) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.166 Mobile Safari/535.19
Android Firefox	16.0.1	Mozilla/5.0 (Android; Mobile; rv:16.0) Gecko/16.0 Firefox/16.0
Android System	4.1.1-398337	Mozilla/5.0 (Linux; U; Android 4.1.1; en-us; Galaxy Nexus Build/JRO03C) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0 Mobile Safari/534.30
Android Opera	12.10.ADR-1210091050	Opera/9.80 (Android 4.1.1; Linux; Opera Mobi/ADR-1210091050) Presto/2.11.355 Version/12.10
Windows Chrome	22.0.1229.94	Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.4 (KHTML, like Gecko) Chrome/22.0.1229.94 Safari/537.4
Windows Firefox	16.0.1	Mozilla/5.0 (Windows NT 6.1; rv:16.0) Gecko/20100101 Firefox/16.0
Windows Opera	12.02.1578	Opera/9.80 (Windows NT 6.1; U; en) Presto/2.10.289 Version/12.02
Windows IE	9.0.8112.16421	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
Go[33]	1.0.3	

Example code

Because JavaScript is such a nuanced language where even seemingly meaningless changes can have significant performance effects we have included the code used to run our experiments here.

DOM

```
d = []; // delay
c = 0;  // count
u = 'http://lab.cs.unc.edu/large'

startExp = function() {
  i = document.createElement('img');
  i.onload = imgDone;
  st = Date.now();
  i.src = u + '?t=' + st + '&c=' + c;
};

imgDone = function() {
  d[c] = Date.now() - st;
  c++;
  if (c < 100) {
    st = Date.now();
    i.src = u + '?t=' + st + '&c=' + c;
  }
};
```

XHR

```
d = []; // delay
c = 0;  // count
u = 'http://lab.cs.unc.edu/largeTxt';
```

[33] Go was used to write a client that performs HTTP requests to provide a ground truth without browser optimizations.

```
startExp = function() {
  if (window.XMLHttpRequest) {
    r = new XMLHttpRequest();
  }
  else if (window.ActiveXObject) {
    r = new ActiveXObject('Microsoft.XMLHTTP');
  }
  r.onreadystatechange = imgDone;
  st = Date.now();
  r.open('GET', u + '?t=' + st + '&c=' + c, true);
  r.send();
};

imgDone = function() {
  if (r.readyState == 4) {
    d[c] = Date.now() - st;
    c++;
    if (c < 100) {
      st = Date.now();
      r.open('GET', u + '?t=' + st + '&c=' + c, true);
      r.send();
    }
  }
};
```

Navigation Timing data collection script

```
d = []; // delay
c = 0;  // count
u = 'http://lab.cs.unc.edu/largeIframe';

startExp = function() {
  document.domain = 'lab.cs.unc.edu';
  if (window.performance && window.performance.timing) {
    i = document.createElement('iframe');
    i.style.visibility = 'hidden';
    document.body.appendChild(i);
    st = Date.now();
    i.src = u + '?t=' + st + '&c=' + c;
    return;
  }
};

// wpt = window.performance.timing
recordTiming = function(wpt) {
  d[c] = wpt.responseEnd - wpt.responseStart;
  c++;
  if (c < 100) {
    st = Date.now();
    return u + '?t=' + st + '&c=' + c;
  }
};
```

Navigation timing iframe (case A)

```
<html>
  <head>
    <script type="text/javascript">
      perfTime = function() {
        document.domain = 'lab.cs.unc.edu';
        if (window.performance && window.performance.timing) {
          var u = window.top.recordTiming(window.performance.timing);
          if (u) {
            self.location = u;
          }
        }
      };
    </script>
  </head>
  <body onload="perfTime();">
    <img alt="data" src="data:image/png;base64,iVBORw0KGgoA..." />
  </body>
</html>
```

Navigation timing iframe (case B)

```
<html>
  <head>
  </head>
  <body onload="perfTime();">
    <img alt="data" src="data:image/png;base64,iVBORw0KGgoA..." />
    <script type="text/javascript">
    document.domain = 'lab.cs.unc.edu';
    if (window.performance && window.performance.timing) {
      var u = window.top.recordTiming(window.performance.timing);
      if (u) {
        self.location.href = u;
      }
    }
    </script>
  </body>
</html>
```

Navigation timing iframe (case C)

```
<html>
  <head>
  </head>
  <body onload="perfTime();">
    <img alt="data" src="data:image/png;base64,iVBORw0KGgoA..." />
    <script type="text/javascript">
      perfTime = function() {
        document.domain = 'lab.cs.unc.edu';
        if (window.performance && window.performance.timing) {
          var u = window.top.recordTiming(window.performance.timing);
          if (u) {
            self.location = u;
          }
        }
      };
    </script>
  </body>
</html>
```