Appraising the Delay Accuracy in Browser-based Network Measurement

Weichao Li, Ricky K. P. Mok, Rocky K. C. Chang, and Waiting W. T. Fok
Department of Computing
The Hong Kong Polytechnic University
{csweicli|cskpmok|csrchang|cswtfok}@comp.polyu.edu.hk

ABSTRACT

Conducting network measurement in a web browser (e.g., speedtest and Netalyzr) enables end users to understand their network and application performance. However, very little is known about the (in)accuracy of the various methods used in these tools. In this paper, we evaluate the accuracy of ten HTTP-based and TCP socket-based methods for measuring the round-trip time (RTT) with the five most popular browsers on Linux and Windows. Our measurement results show that the delay overheads incurred in most of the HTTP-based methods are too large to ignore. Moreover, the overheads incurred by some methods (such as Flash GET and POST) vary significantly across different browsers and systems, making it very difficult to calibrate. The socket-based methods, on the other hand, incur much smaller overhead. Another interesting and important finding is that `Date.getTime()`, a typical timing API in Java, does not provide the millisecond resolution assumed by many measurement tools on some OSes (e.g., Windows 7). This results in a serious under-estimation of RTT. On the other hand, some tools over-estimate the RTT by including the TCP handshaking phase.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations; C.4 [**Performance of Systems**]: Measurement Techniques

Keywords

Delay; Measurement; Accuracy; Web

1. INTRODUCTION

The accuracy of a network measurement system can be defined as how the measurement results deviate from the real network performance. Following ISO 5725 [3], a network measurement can be considered more accurate if its produced results are closer to the actual values (trueness) and are more consistent than the others (precision or repeatability). Measurement accuracy depends on a number of factors, including the correctness of adopted methodology, time resolution, system load, and so on. To ensure high accuracy, network measurement is traditionally performed in dedicated hosts with careful resource management [9, 24], and the network performance is sampled using Poisson process [4, 5, 6, 23, 30].

Recently, more measurement tools are available to end users to measure their network performance and diagnose problems. Such efforts include various speedtest services, residential broadband measurement [34], and several host-based tools [17, 32]. In particular, many of these tools, such as Netalyzr [19] and Ookla's speedtest [1], take advantage of browser's ubiquity by implementing them in browsers. These *browser-based measurement* tools usually can measure the network round-trip time (RTT) and throughput. Some can even measure packet loss and reordering rates. The effort towards this direction is recently stepped up by Fathom [11] which provides a number of APIs for network measurement functions.

Although browser-based measurement has gained popularity among end users, very little is known about the (in)accuracy of various methods used in these tools. This paper focuses on the accuracy of network delay measured by these tools, because a browser-based tool could easily inflate the network delay measurement (but not so much for loss and reordering). The delay obtained on the browser level may over-estimate (or under-estimate) the actual RTT due to a number of reasons. The inflated delay will also affect jitter, bandwidth measurement, and passive measurement methods (e.g., [13, 14]) that assume a very small inflation.

We quantify the delay inflation by investigating the *delay overhead* on the browser side, which is the difference between the value measured by browser-based tools and the actual value (calculated through packet capturing). The amount of this overhead depends on how the rendering engine (e.g., JavaScript engine) interprets the measurement code and invokes system function calls. We study ten different methods (seven based on HTTP and three on TCP sockets, including WebSocket) which are usually implemented using JavaScript (native in browsers), Flash, and Java applet. We implement these methods and experiment with the five major browsers on Windows and Ubuntu.

Our measurement results show that the socket-based methods incur much lower delay overhead than the HTTP-based methods in general. The Flash GET and POST methods are most unreliable, because their overheads are the highest among all methods, and their overhead variabilities are also the highest across different browsers and systems. WebSocket, on the other hand, provides the most accurate and consistent RTT measurement in the context of JavaScript and DOM (Document Object Model). Another interesting finding is that the typical timing API in Java, `Date.getTime()`, cannot return precise system time in some OSes (e.g., Windows 7). Although this function is supposed to provide timestamps with millisecond resolution, we find that the actual granularity is not constant. It can be one of the two values observed in our experiments: 1 ms or $\sim$15 ms, and each possible value will

IMC'13, October 23–25, 2013, Barcelona, Spain.

http://dx.doi.org/10.1145/2504730.2504760.

last for a period of time (several minutes) before changing to other values. Consequently, the timestamps produced by this API can significantly under-estimate or fluctuate the measured RTTs.

In this paper, we report two important cases with detailed investigations. First, we discover that some HTTP-based methods over-estimate the RTT, because they include the TCP handshaking in the delay. Second, we study the effect of timing methods in the Java applet case by replacing the timing function with `System.nanoTime()`. The experiment results show that the under-estimation of RTT disappears after introducing the new timing function. Furthermore, the new implementation introduces very small and consistent delay overhead. In particular, the variation of delay overhead in the socket-based measurement can approximate to 0 ms, meaning that it can estimate the RTTs accurately as what WinDump can do.

The outline of this paper is as follows. In Section 2, we first survey the measurement methods used in browser-based network measurement tools and services. We then describe our measurement setup in Section 3, followed by a report of the results in Section 4. Based on our evaluation, we summarize several practical considerations in Section 5. After highlighting the related works in Section 6, we conclude the paper in Section 7.

2. BROWSER-BASED NETWORK MEASUREMENT

A browser-based network measurement tool can generally provide many different services. Netalyzr [19], for example, provides network-layer information (e.g., RTT and path MTU), service reachability, and DNS measurement. In this paper, we consider the accuracy of the network RTT measurement, because a browser-based tool, being operated on the application layer, may significantly inflate the actual network RTT. The inflation also affects jitter and throughput (Tput) measurement. However, we do not anticipate such impact on packet loss and reordering measurement.

We have studied the RTT measurement methods employed by a number of browser-based tools, such as Netalyzr [19], Janc's methods [16], and How's My Network (HMN) [31], and speedtest services, including Speedof.me [2] and Ookla [26, 25], by inspecting their codes and the packets exchanged between browsers and servers. Their methods comprise a preparation phase and a measurement phase, as shown in Figure 1. In the preparation phase, the browser first loads from a web server a container page containing a piece of measurement code. In the measurement phase,

1. (Send) The measurement code is executed at the browser to instantiate an object which sends a "request" message (e.g., HTTP `GET` or binary data) to the origin server or another web server to elicit a "response" message. The timestamp (t_s^B) is recorded just before sending the request message which may be sent in one IP packet (for RTT measurement) or multiple IP packets (for throughput measurement).
2. (Receive) The web server that receives the response message returns a "response" message (e.g., HTTP response message or binary data) to the browser. The timestamp (t_r^B) is recorded immediately after receiving the response message. The RTT is then estimated by $t_r^B - t_s^B$. Similar to the send case, the response message may be sent in one or more IP packets.

2.1 HTTP-based and socket-based methods

The methods for a browser to send a request message to a web server for RTT measurement can be classified into HTTP-based and socket-based. Table 1 summarizes eleven such methods and

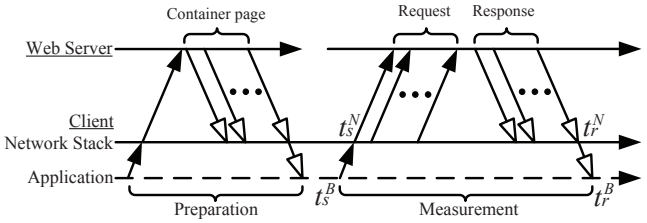


Figure 1: Two phases in browser-based network measurement.

the tools that use them. The HTTP-based method could be implemented through JavaScript, Flash, or Java applet. A JavaScript code imbedded in the container page creates an XHR object and calls the `send()` function to send out an HTTP request. The object records t_s^B using the JavaScript function `Date.getTime()` and uses the `onreadystatechange` event listener to determine whether the response has been received for recording t_r^B. Another JavaScript method is based on DOM element that first records t_s^B before inserting a new DOM element to the page using a <`script`> tag or <`img`> tag. This tag points to a specified URL to download the requested object. A successful loading triggers an `onload` event which prompts the logging of t_r^B. Flash, on the other hand, provides class `URLLoader` to handle HTTP data, and Java applet offers class `URL`. Both of them provide function `Date.getTime()` to log the current timestamps, and they record t_s^B just before sending out the request. Flash detects the completion of receiving the response via function `addEventListener` for recording t_r^B. Although there is no such event listener in Java applet, the completion can be detected by reading the response content.

The socket-based method, on the other hand, establishes network connections/associations through TCP or UDP sockets for exchanging binary data. TCP socket is supported by Flash, Java applet, and WebSocket, whereas UDP socket is only supported by Java applet. Flash manipulates network socket with class `Socket`. This class also provides function `addEventListener` which detects data arrival. In Java applet, the sockets are created via class `Socket` for TCP and `DatagramSocket` for UDP. The timestamps are recorded after the receive function call. WebSocket provides its functionality through JavaScript. WebSocket is like TCP socket on the abstraction level, except that the data transmissions are based on messages. Therefore, WebSocket obtains the timestamps in a similar way as Flash and Java applet.

All the HTTP-based methods, except for DOM, suffer from the restriction imposed by the same-origin policy which prevents a browser from accessing other servers except the original one hosting the container page. However, Flash can bypass this restriction through the Flash cross-domain policy, and Java applet's approach is through a signed Java applet. On the other hand, except for Flash, the socket-based methods are not affected by the same-origin policy, but they are required to open service ports for socket connections. Another important consideration is that Flash and Java applet, as the third-party plug-ins, are not supported in mobile computing platforms. As a result, WebSocket is the remaining choice for performing socket-based measurement in both fixed and mobile network platforms.

2.2 Measuring the delay overhead

The main objective of this paper is to accurately measure the delay overhead incurred at browsers when measuring RTT. Back to Figure 1, supposing that the request and response messages are sent in one packet each, the network RTT is given by the difference

Table 1: A summary of the browser-based network measurement methods and tools.

Approaches	Technology	Availability	Methods	Subject to the same-origin policy by default?	Measured path-quality metrics	Tools / Services
HTTP-based	XHR	Native	GET	Yes	RTT, Tput	Speedof.me [2], BandwidthPlace [21]
			POST	Yes	RTT, Tput	Janc's methods [16]
	DOM	Native	GET	No	RTT, Tput	[16], [21], Wang's method [35]
	Flash	Plug-in	GET	Yes∗	RTT, Tput	Speedtest [26], AuditMyPC [7], Speedchecker [33]
			POST	Yes∗	RTT, Tput	Bandwidth Meter [10], InternetFrog [15]
	Java applet	Plug-in	GET	Yes∗	RTT, Tput	
			POST	Yes∗	RTT, Tput	
Socket-based	WebSocket	Native	TCP	No	RTT, Tput	
	Java applet	Plug-in	TCP	No	RTT, Tput	Netalyzr [19], HMN [31], JavaNws [20],
			UDP	No	RTT, Tput, Loss	Pingtest [25], NDT [22], AuditMyPC [8]
	Flash	Plug-in	TCP	Yes∗	RTT, Tput	[26]

Note: ∗ The same-origin policy can be bypassed.

of the packet's receive and send timestamps which is measured by WinDump and tcpdump: $t_r^N - t_s^N$. Since browsers cannot access to network stack directly, the measured RTT is based on $t_r^B - t_s^B$. The time resolution for this browser-level measurement is usually assumed to be 1 ms, determined by the function `Date.getTime()` (we will discuss the real-time granularity of this function for Java applet in Section 4.2). The accuracy of the browser-level RTT measurement depends on several factors:

1. Accuracy of the timing function invoked by the adopted measurement method,
2. The delay for the browser to propagate the request message to the network stack and the delay for delivering the response message to the browser, and
3. The behavior of how the browser sends the message, for example, whether the delay includes the time for establishing a TCP connection.

To appraise the delay accuracy in browser-based network measurement, we therefore measure the delay overhead as

$$\Delta d = (t_r^B - t_s^B) - (t_r^N - t_s^N). \quad (1)$$

Besides affecting the RTT measurement, the delay overhead, if not stable enough, will also affect the jitter measurement. Moreover, the actual round-trip throughput could be seriously under-estimated by an inflated RTT.

3. EXPERIMENT SETUP

In measuring the delay overhead incurred on the RTT measurement, we consider all the HTTP/TCP measurement methods in Table 1. To make the comparison more comparable, we do not include Java's UDP socket method. Besides the ten measurement methods, we investigate the consistency of delay overhead of a given method across browsers and systems. Ideally, a browser-based tool is expected to incur similar delay overhead, regardless of which browser and system it is operated on. To this end, we consider the five major browsers on Windows 7 and Ubuntu in Table 2 with the Flash and Java applet plug-in configurations. Note that the IE and Safari versions used in the experiments do not support WebSocket. Although the latest IE 10 and Safari 6 both support WebSocket, we use IE 9 and Safari 5 instead, because IE 9 is the default browser for Windows 7 and Safari 6 is not available in Windows 7. For fair comparison, what we have tested are all 32-bit browsers, because some of the browsers do not provide 64-bit version.

We set up a testbed consisting of two machines connected to a switch by 100-Mbps Ethernet, as shown in Figure 2. Both machines have the same hardware configuration: equipped with a 1.86GHz

Table 2: Configurations of the browsers and systems used in the experiments.

OS	Browsers	Version	Flash	Java applet	Web Socket
Windows	Chrome	23.0	11.7.700	1.7.0	√
	Firefox	17.0	11.5.502	1.7.0	√
	IE	9.0.8	11.5.502	1.7.0	×
	Opera	12.11	11.5.502	1.7.0	√
	Safari	5.1.7	11.5.502	1.7.0	×
Ubuntu	Chrome	23.0	11.5.31	1.6.0	√
	Firefox	17.0	11.2.202	1.6.0	√
	Opera	12.11	11.2.202	1.6.0	√

Intel Core 2 Duo processor (E6320) and 2GB memory. One is a dual-boot system with Windows 7 and Ubuntu 12.04 LTS, and is installed with the five browsers. The other machine hosts an Apache web server version 2.2 on Ubuntu 10.04. We also introduce an additional delay of 50 ms on the server side to simulate the Internet environment. Without such delay, the link RTT (< 1 ms) is too small to sample. Beyond that, as we shall see in the next section, this delay is a major factor determining the amount of RTT inflation when a measurement method includes TCP handshaking in the delay measurement.

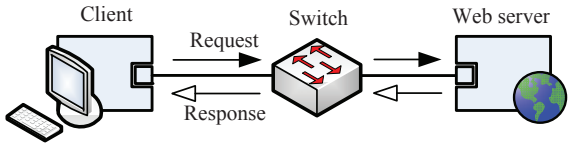


Figure 2: Testbed Setup.

We have prepared a container page using PHP or HTML for each measurement method imbedded with JavaScript code, Flash object, or Java applet[1]. The entire suite of experiments is executed automatically. Each browser program is executed on command line, and it retrieves from the server a container page for a given measurement method. When the browser renders the page, it executes the measurement code to instantiate the required object which sends a request message to the same web server which returns a reply message with the 50 ms delay. As discussed in section 2, the measurement code records t_s^B and t_r^B. At the same time, the client machine runs WinDump/tcpdump to capture t_s^N and t_r^N.

[1]Source codes are available at `http://www4.comp.polyu.edu.hk/~oneprobe/src.php`

Considering the possible impact on the browser to instantiate the object for the first RTT measurement, we conduct a second RTT measurement immediately after the first one and reusing the same object. Therefore, for each setting, we obtain two sets of delay overheads, denoted by Δd_1 and Δd_2. Moreover, we choose small request and reply messages, each of which can be sent in one packet. This setting allows us to remove other possible delay due to data segmentation, send and receive buffering, and throttling by the send window. During the measurement period, we also ensure that the network was free of cross traffic, packet loss, and retransmissions. Although the web server could bias the RTT, the bias, if any, is mitigated by the subtraction of $t_r^B - t_s^B$ and $t_r^N - t_s^N$ in the same round of measurement.

For each experiment, we run it for 50 times and compute from them useful statistics, such as minimum, median, and 25% and 75% percentiles. We do not record the system load, but we ensure that all the necessary processes (e.g., `explorer.exe` in Windows, `init` in Linux, and so on) run in the background. Besides, some other programs, such as packet capturing program and automation scripts, need to be dynamically invoked during the measurement procedure. The browsers themselves also consume resources to render the measurement objects. As a result, the delay overheads may still vary, depending on how sensitive the measurement methods are to these system loads.

4. MEASUREMENT RESULTS

We plot the ten sets of measurement results (one per measurement method) in Figure 3 using box-and-whisker plots. The first row includes the four methods using native features in browsers. The second comprises the Flash methods, and the third the Java applet methods. Each plot (except for WebSocket) includes the measurement by the eight browser-OS cases which are identified by the browser's initial (system's initial). They are then followed by Δd_1 (in red) or Δd_2 (in cyan). For example, "C (U) Δd_1" refers to Δd_1 obtained by Chrome in Ubuntu.

In each box-and-whisker plot, the top and bottom of the box are given by the 75th percentile and 25th percentile, and the mark inside is the median. The upper and lower whiskers are the maximum and minimum, respectively, after excluding the outliers. The outliers above the upper whiskers are those exceeding 1.5 of the upper quartile, and those below the minimum are less than 1.5 of the lower quartile.

Figures 3(a), 3(b), 3(c), 3(e), 3(f), 3(h), and 3(i) for the HTTP-based methods show that the delay overhead generally cannot be ignored. The XHR methods' delay overheads range from a few milliseconds to tens of milliseconds. The overheads in Flash are extremely high. The median overheads are between 20 ms and 100 ms. Even for the minimum overheads, they can reach as high as 100 ms (Δd_1 of Opera in both Windows and Ubuntu). The DOM methods achieves a better result than XHR and Flash. Most of the median overheads are smaller than 5 ms. The Java applet methods differ from the previous group in that they could (e.g., Firefox and Opera) under-estimate the RTT (i.e., negative overhead) by as much as 5 ms.

Another important result concerns the consistency of a measurement method across different browsers and systems. If the overheads are dependent on specific browsers and systems, it will make the calibration very difficult. The delay overheads for the HTTP-based methods generally see a very high variability across browsers for the Flash methods. The DOM method provides the most consistent overhead across all browsers, especially those on Ubuntu. The two Java applet methods are also quite consistent on the Ubuntu but less consistent on Windows.

On the other hand, Figures 3(d), 3(g), and 3(j) show that the delay overheads incurred by the socket-based methods are considerably small. The median overheads are mostly smaller than 1 ms. Nevertheless, the overheads for some browsers fluctuate within a range of around 10 ms (e.g., Java applet for Firefox in Windows). Overall, the WebSocket method achieves the most stable result, except for Opera (W) Δd_1. Similar to the other two Java applet methods, the Java applet socket method will under-estimate the delay, especially those in Windows.

4.1 The effect of network behavior on HTTP-based methods

The major difference between the HTTP-based and socket-based methods is that the former needs to parse the additional HTTP header. However, parsing HTTP alone cannot explain those high delay overheads. We consider some of these cases next and analyze other possible reasons responsible for the RTT inflation.

Table 3 shows the median overheads for the Flash GET and POST methods, obtained by Opera in Windows and Ubuntu. Although the data are collected from different OSes, the delay overheads behave similarly. For the GET method, O(W) and O(U) both suffer from a very large Δd_1 (> 100 ms) but a relatively small Δd_2 (< 20 ms). For the POST method, the median Δd_1 is still high, but the median Δd_2 is much larger than that for the GET method.

Table 3: Median Δd_1 and Δd_2 for the Flash HTTP methods in Opera.

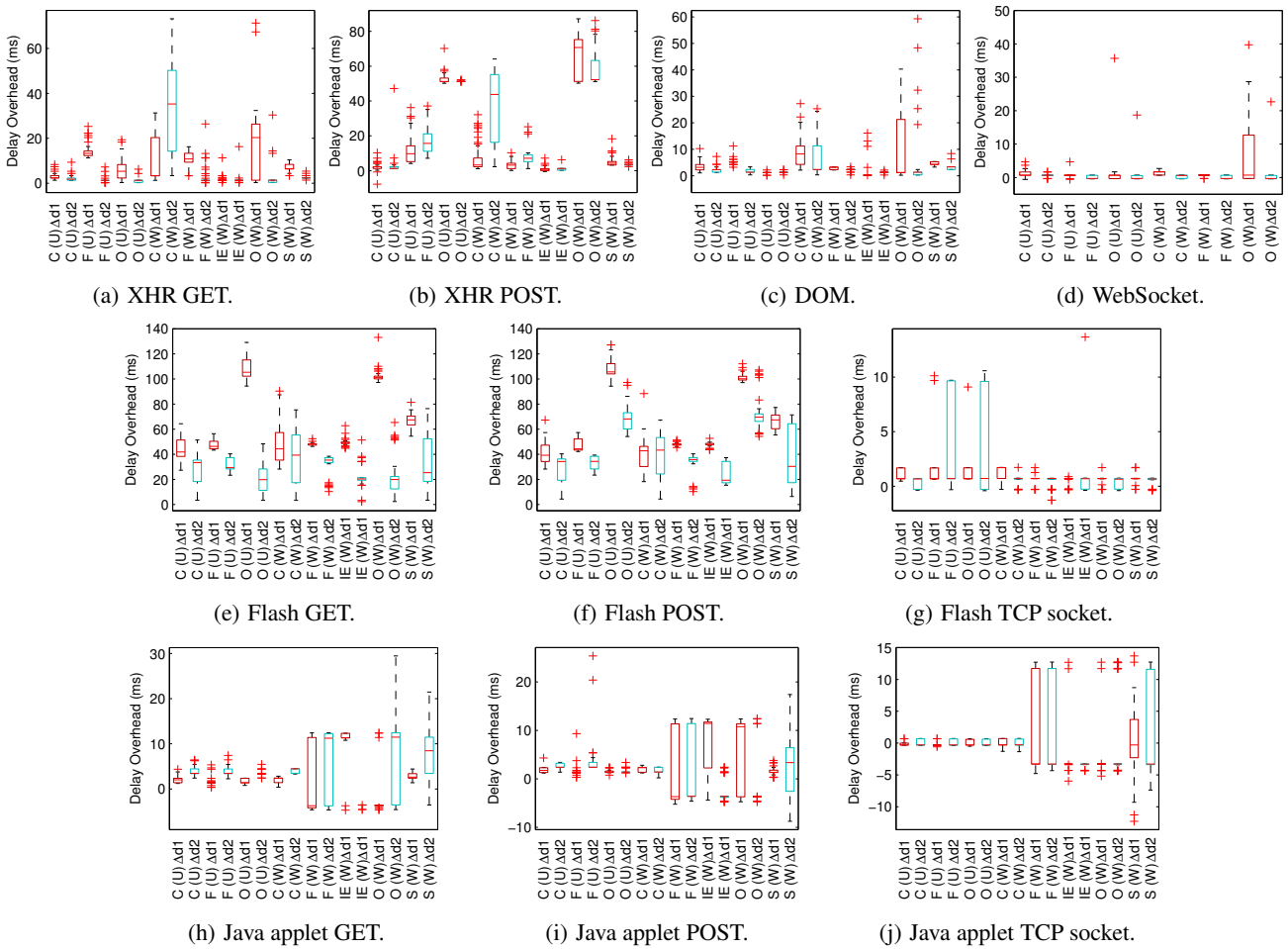
		O(W)	O(U)
GET	Δd_1	101.1	105.3
	Δd_2	19.8	19.8
POST	Δd_1	100.1	105.6
	Δd_2	69.6	68.1

The packet capture files show that Opera opens a new TCP connection to handle the HTTP request issued by the Flash object for the first RTT measurement, therefore inflating the Δd_1 measurement. In the GET method, this existing connection can be reused for the second measurement. Therefore, the Δd_1 measurement excludes the TCP handshaking. However, a new connection will still be opened for the POST method. We confirm this by subtracting 50 ms, the simulated network delay, from Δd_2 in the POST method, the result ($\sim$20 ms) is almost the same as the GET method. Moreover, we compare the behavior of other browsers and find that even for the first RTT measurement they will reuse the TCP connection for downloading the container page in the preparation phase, thus resulting in a much lower overhead.

4.2 The effect of timestamp granularity

From Figure 3(h), 3(i), and 3(j), all three Java applet methods suffer from the negative delay overheads on Windows, which indicates that performing path measurement with Java applet can severely under-estimate the RTT. At the same time, significant variance can be observed. For example, Safari's overhead in Java applet socket method spans in the range of -13 ms and 13 ms, as illustrated in Figure 3(j). Due to the page limit, we only discuss the socket case for evaluation.

We show the CDFs of Δd_1 and Δd_2 of those experiments in Figure 4(a). The figure depicts that both Δd_1 and Δd_2 for Firefox and Opera, and Δd_1 for Safari have two discrete levels, whereas Δd_2 for Safari spans continuously over the range.According to [28], web browsers instantiate Java applet through Java Plug-in. In fact, an applet runs in an instance of the Java Runtime Environment (JRE) software, not within the browsers. To mitigate the influence of browsers, we directly launch the applet with `appletviewer`



(a) XHR GET. (b) XHR POST. (c) DOM. (d) WebSocket.

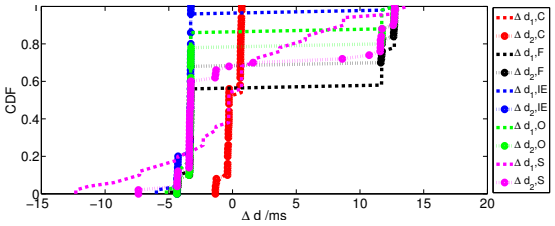
(e) Flash GET. (f) Flash POST. (g) Flash TCP socket.

(h) Java applet GET. (i) Java applet POST. (j) Java applet TCP socket.

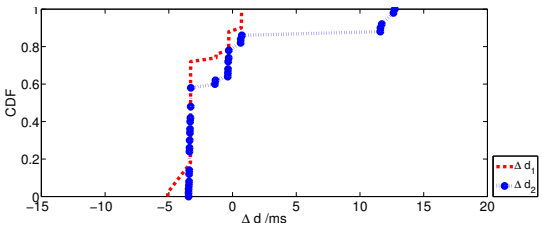
Figure 3: Box plots of the delay overheads (by methods).

provided by Oracle Java Development Kit (JDK). We plot the CDFs of Δd_1 and Δd_2 in Figure 4(b). Similar discrete levels are observed without web browser and Java Plug-in. We thereupon can rule out browsers and their corresponding Java Plug-ins being the cause of this problem.

We then focus on the JRE itself. The timing function in Java, `Date.getTime()`, is implemented with another Java function `System.currentTimeMillis()`. An Oracle's documentation warns that while the resolution of the return value is 1 ms, the granularity depends on the underlying system [29]. We test the timestamp granularity with the code shown in Figure 5. The piece of code keeps querying the timestamp with `Date.getTime()` until the current value is different from the previous one. The difference in the two timestamps is the granularity that this function can achieve. Surprisingly, we find that the granularity is not a constant value. It can be 1 ms, or $\sim$15 ms. Each possible value will last for a period of time (several minutes) and then change to other values. While such a coarse granularity of timestamp in Windows was reported [27], it has not mentioned the non-constant granularity. Initially, we conjecture that the varying time granularity is related to the 32-bit JRE. However, we later find that 64-bit JRE also suffers from the same problem. To further validate our findings, we analyze the data obtained from the delay overhead experiments. The gap between the two significant discrete levels is about 16 ms, which concurs with one of the timestamp granularity obtained from the test codes. Hence, we believe that the coarse and instable timestamp granularity is the main reason for the bizarre behavior observed in the previous delay overhead experiments.



(a) Launched in browsers.



(b) Launched with appletviewer.

Figure 4: CDF plots of Δd_1 and Δd_2 using the Java applet socket in Windows.

Table 4: Delay overheads measured by Java applet methods in Windows when function `System.nanoTime()` is adopted (mean with 95% confidence interval, in ms).

Method	GET		POST		Socket	
	Δd_1	Δd_2	Δd_1	Δd_2	Δd_1	Δd_2
Chrome	2.96 $\pm$0.02	4.80 $\pm$0.09	2.71 $\pm$0.03	1.84 $\pm$0.00	0.01 $\pm$0.00	0.07 $\pm$0.01
Firefox	2.73 $\pm$0.02	4.38 $\pm$0.08	2.41 $\pm$0.03	1.49 $\pm$0.01	0.00 $\pm$0.00	0.07 $\pm$0.01
IE	2.73 $\pm$0.03	4.56 $\pm$0.09	2.57 $\pm$0.09	1.49 $\pm$0.04	0.02 $\pm$0.01	0.06 $\pm$0.01
Opera	2.83 $\pm$0.03	4.46 $\pm$0.07	2.51 $\pm$0.03	1.57 $\pm$0.01	0.01 $\pm$0.00	0.06 $\pm$0.01
Safari	1.88 $\pm$0.05	1.52 $\pm$0.02	1.62 $\pm$0.07	1.42 $\pm$0.01	0.07 $\pm$0.00	0.13 $\pm$0.01

We replace the timing function `Date.getTime()` with a more precise `System.nanoTime()` and then rerun the experiments with the same configurations. The measurement results are summarized in Table 4. We present the mean delay overhead as well as the 95% confidence intervals. The under-estimation and the large variation of RTTs disappears after the replacement, including the other two Java applet methods. For the GET and POST methods, the mean delay overheads range from 2 ms to 5 ms, only a little larger than the WebSocket cases. As for the socket methods, the delay overheads are trivial. Considering the accuracy of software packet capturer being larger than 0.3 ms [7], we can regard the accuracy of the Java socket method comparable to tcpdump/WinDump if `System.nanoTime()` is adopted.

```
1  long start = 0;
2  long end = 0;
3  while (true) {
4     if (start == 0) {
5        start = new Date().getTime();
6     } else {
7        long current = new Date().getTime();
8        if (current != start) {
9           end = current;
10          break;
11       }
12    }
13 }
14 System.out.println((end - start) + "ms");
```

Figure 5: Codes for testing the timestamp granularity.

5. PRACTICAL CONSIDERATIONS

Based on the overall evaluation, the Java applet socket method is recommended if the proper timing function is applied. However, our inspection of some Java applet-based tools shows that many of them are still using `System.currentTimeMillis()` or `Date.getTime()`, such as [8, 19, 22]. Switching to the more precise function `System.nanoTime()` can greatly improve their accuracy in Windows. Based on our evaluation, the Flash GET and POST methods are not so suitable for the purpose of measurement.

For the measurements performed in Windows, Firefox is the preferred browser, whereas in Ubuntu Chrome is a better choice. We do not recommend Safari even for the Java applet socket method due to the fact that its default Java interface (`JavaPlugin.jar` and `npJavaPlugin.dll`) runs into problems easily. The measurement results obtained from Safari are much higher than the other browsers. After deleting the two files, we can force it to use the JRE provided by Oracle, and the inaccuracies are subsequently removed.

There are also issues of reusing existing connections and web objects for network measurement. The real-world applications are more complicated than our experiment settings. The browsers have to establish new connections due to the competition of downloading the other files. If a measurement object can be reused, the delay overhead can be better estimated by Δd_2 without including the TCP handshaking delay. However, some methods, as described in Section 4.1, always open new connections for measurement whether the measurement object can be reused or not. In this case, the additional delay cannot be avoided.

6. RELATED WORKS

Although browser-based network measurement tools and services have been widely deployed, only a handful of studies are devoted to appraising them. These previous works consider only a small number of methods. Janc et al. [16] proposed HTTP-based methods using JavaScript and Flash for measuring network performance, and performed control and web experiments to compare the methods. Later, Kaplan et al. [18] performed testbed experiments to investigate the delay overhead incurred by browser with four HTTP-based methods using JavaScript and Flash. Both papers concluded that JavaScript performs better than Flash for delay measurement, which is coherent with our results. However, they did not compare the HTTP-based methods with socket-based methods.

Krintz and Wolski [20] compared the performance between Java applet and C program with JavaNws, and found that Java applet is comparable with C socket. Yeboah et al. [36] performed an Internet measurement study to compare the delay measurement results from ICMP ping, King [12], Flash (socket-based), and JavaScript (HTTP-based). They found that the results from Flash socket measurement were close to ping, whereas JavaScript had an inflated delay. However, both papers did not utilize any network stack information, such as tcpdump capture, to investigate the actual overhead caused by the applications.

7. CONCLUSION

In this paper, we studied the impact of application-level delay overheads on browser-based network measurement tools. By evaluating all the HTTP/TCP methods employed by the current browser-based measurement tools and services with our carefully designed testbed experiments, we showed that both socket-based and HTTP-based methods may introduce different degrees of inaccuracy in measuring the RTT due to a number of intrinsic and system issues. Based on the results, the socket-based methods are generally more reliable than the HTTP-based methods. Although our work is done in desktop environment, the methodology can be extended to the mobile environment. Another extension is to investigate the delay overhead incurred on the server side.

Acknowledgement

We thank the four anonymous reviewers for their very useful comments and feedback for improving the paper, and our shepherd Jeffrey Pang for guiding us during the revision process. This work is partially supported by an ITSP Tier-2 project grant (ref. no. GHP/027/11) from the Innovation Technology Fund in Hong Kong.

8. REFERENCES

[1] Ookla.com. http://www.ookla.com.

[2] Speedof.me. http://speedof.me/.

[3] Accuracy (trueness and precision) of measurement methods and results – part 1: General principles and definitions. ISO 5725-1, 1994.

[4] G. Almes, S. Kalidindi, and M. Zekauskas. A one-way delay metric for IPPM. RFC 2679, IETF, Sept. 1999.

[5] G. Almes, S. Kalidindi, and M. Zekauskas. A one-way packet loss metric for IPPM. RFC 2680, IETF, Sept. 1999.

[6] G. Almes, S. Kalidindi, and M. Zekauskas. A round-trip delay metric for IPPM. RFC 2681, IETF, Sept. 1999.

[7] Audit My PC.com. AuditMyPC.com Broadband Speed Test (Flash). http://www.auditmypc.com/internet-speed-test.asp.

[8] Audit My PC.com. Internet Speed Test (Java). http://www.auditmypc.com/internet-speed-test.asp.

[9] CAIDA. Archipelago Measurement Infrastructure. http://www.caida.org/projects/ark/.

[10] cnet.com. Bandwidth Meter Online Speed Test. http://reviews.cnet.com/internet-speed-test/.

[11] M. Dhawan, J. Samuel, R. Teixeira, C. Kreibich, M. Allman, N. Weaver, and V. Paxson. Fathom: A browser-based network measurement platform. In *Proc. ACM/USENIX IMC*, 2012.

[12] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: estimating latency between arbitrary internet end hosts. In *Proc. SIGCOMM IMW*, 2002.

[13] E. Halepovic, J. Pang, and O. Spatscheck. Can you GET me now? Estimating the time-to-first-byte of HTTP transactions with passive measurements. In *Proc. ACM/USENIX IMC*, 2012.

[14] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An in-depth study of LTE: Effect of network protocol and application behavior on performance. In *Proc. ACM SIGCOMM*, 2013.

[15] InternetFrog.com. InternetFrog.com Speed Test. http://www.internetfrog.com/mypc/speedtest/.

[16] A. Janc, C. Wills, and M. Claypool. Network performance evaluation in a web browser. In *Proc. IASTED PDCS*, 2009.

[17] D. Joumblatt, R. Teixeira, J. Chandrashekar, and N. Taft. HostView: Annotating end-host performance measurements with user feedback. In *Proc. ACM HotMetrics*, 2010.

[18] M. Kaplan, M. Zeljkovic, M. Claypool, and C. Wills. Javascript and Flash overhead in the web browser sandbox. Tech. Rep. WPI-CS-TR-10-14, Computer Science Department, Worcester Polytechnic Institute, 2012.

[19] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *Proc. ACM/USENIX IMC*, 2010.

[20] C. Krintz and R. Wolski. Using JavaNws to compare C and Java TCP-Socket performance. *Concurrency Computat.: Pract. Exper.*, 13(8-9):815–839, 2001.

[21] S. Limited. BandwidthPlace Speed Test. http://www.bandwidthplace.com/.

[22] M-Lab. NDT (Network Diagnostic Tool). http://measurementlab.net/run-ndt.

[23] J. Mahdavi and V. Paxson. IPPM metrics for measuring connectivity. RFC 2678, IETF, Sept. 1999.

[24] D. Morato, E. Magana, M. Izal, J. Aracil, F. Naranjo, F. Astiz, U. Alonso, I. Csabai, P. Haga, G. Simon, J. Steger, and G. Vattay. The European Traffic Observatory Measurement Infraestructure (ETOMIC): A testbed for universal active and passive measurements. In *Proc. Tridentcom*, 2005.

[25] Ookla. Pingtest.net. http://www.pingtest.net/.

[26] Ookla. Speedtest.net. http://www.speedtest.net/.

[27] Oracle. Bad timing using System.currentTimeMillis() instead of System.nanoTime(). http://whileonefork.blogspot.hk/2010/12/bad-timing-using-systemcurrenttimemilli.html.

[28] Oracle. Java Plug-in and Applet Architecture. http://docs.oracle.com/javase/7/docs/technotes/guides/jweb/applet/applet_execution.html.

[29] Oracle. System. http://docs.oracle.com/javase/6/docs/api/java/lang/System.html#currentTimeMillis().

[30] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. RFC 2330, IETF, May 1998.

[31] A. Ritacco, C. Wills, and M. Claypool. How's My Network? - A Java approach to home network measurement. In *Proc. IEEE ICCCN*, 2009.

[32] M. Sánchez, J. Otto, Z. Bischof, D. Choffnes, F. Bustamante, B. Krishnamurthy, and W. Willinger. Dasu: Pushing experiments to the Internet's edge. In *Proc. USENIX NSDI*, 2013.

[33] Speedchecker Limited. Broadband Speedchecker. http://www.broadbandspeedchecker.co.uk/.

[34] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape. Broadband Internet performance: A view from the gateway. In *Proc. ACM SIGCOMM*, 2011.

[35] Y. Wang, C. Huang, J. Li, and K. Ross. Estimating the performance of hypothetical cloud service deployments: A measurement-based approach. In *Proc. IEEE INFOCOM*, 2011.

[36] Y. Yeboah Jr., R. Nketia, and X. Hei. A measurement study of application layer latency. Technical report, Huazhong University of Science and Technology, 2011.