Concise Paper: On Measuring One-Way Path Metrics from a Web Server

Xiapu Luo[†‡], Lei Xue[†], Cong Shi[§], Yuru Shao[†], Chenxiong Qian[†], and Edmond W. W. Chan[†]
Department of Computing, The Hong Kong Polytechnic University[†]
The Hong Kong Polytechnic University Shenzhen Research Institute[‡]
School of Computer Science, Georgia Institute of Technology[§]
{csxluo,cslxue,cscqiang,csyshao}@comp.polyu.edu.hk,{shicong82,edmond0chan}@gmail.com

***Abstract*—Measuring one-way path metrics can facilitate adaptive online services (e.g., video streaming and CDN) tuning to improve quality of experience (QoE) of their clients. However, existing server-side measurement systems suffer from (i) measuring only few one-way path metrics, (ii) limited client-side support, and (iii) heavy overheads. In this paper, we propose and implement OWPScope, a novel system that can be deployed to any web server to measure four important one-way path metrics—packet loss, packet reordering, jitter, and capacity—*without* requiring software or plugin installation at their web clients. Moreover, OWPScope performs representative measurement by correlating only information gleaned from standard features in HTML5 (e.g., navigation timing, resource timing), HTTP, and TCP. Our extensive evaluations in both a testbed and the Internet show that OWPScope can effectively measure one-way path metrics with low overhead.**

I. Introduction

The asymmetry of Internet paths stimulates the *one-way* path measurement due to the asymmetric effects [1]. Online services can characterize one-way path metrics to better understand conditions of the network paths between their clients and themselves and achieve adaptive service tuning for Quality-of-Experience (QoE) improvement. For instance, by knowing the one-way path performance from a set of servers to a client, a CDN controller can direct the client to the most suitable server [2]. Such information can also facilitate video streaming services to select a proper bitrate for a client [3], [4], help service providers differentiate ISPs' performance [5], [6] and diagnose network faults [7]. Despite its usefulness, measuring one-way metrics by online services is very challenging due to the following limitations in the existing solutions.

Measuring only few one-way path metrics. Knowing low-level one-way metrics (e.g., packet loss, packet reordering, jitter, and capacity) are indispensable for characterizing and adjusting online services' performance. However, existing measurement systems, such as speedtest and boomerang, only measure limited number of path metrics, such as round-trip time (RTT) and TCP bulk-transfer capacity; and the measurement accuracy is limited due to their incapability to accessing low-level information.

Limited client-side support. Most existing measurement systems require installing specific softwares or plugins at the client side [8]–[12]. Although they may simplify the measurement, not all clients are willing to install such software/plugins due to security concerns, thus limiting their popularity. Moreover, many systems without client-side software installation conduct the measurement by sending special packets (e.g., ICMP) to induce responses from clients [13]–[15], instead of packets carrying application data in established network sessions (e.g., TCP sessions). However, their measurements will fail if the clients' perimeter firewalls filter out these unsolicited packets.

Heavy overheads. Existing systems usually perform individual measurement for each metric and, therefore, cause heavy network and system overheads when multiple metrics are measured in parallel. It is desirable to obtain multiple metrics simultaneously from the same measurement traffic to reduce the impact of the measurement on network paths [16].

To overcome these limitations, we propose OWPScope, a novel system that empowers web servers to simulatenously measure four low-level one-way path metrics, i.e., packet loss, packet reordering, capacity, and jitters. OWPScope exploits only standard features in HTML5, HTTP, and TCP without requiring specific software/plugins installed at the client side. With specially crafted probing packets in an established TCP connection, OWPScope can penetrate client-side firewalls and perform measurement with low overhead by correlating information gleaned from the application and the TCP levels. Moreover, OWPScope uses packets carrying real application data to conduct representative measurement.

OWPScope consists of two key components: (i) a server-side measurement module that sends crafted probing packets and inspects packets from clients to compute the metrics and (ii) a piece of javascript (js) code running in a client's browser to collect required timestamps through HTML5 interfaces. Fig. 1 illustrates one application scenario of OWPScope, where it is deployed to a web server and other resource servers in a CDN network. When a client visits the front page of the web server (step 1), the js code will be downloaded and executed in the client's browser (step 2) and some embedded web objects (e.g., images) will be fetched from other resource reservers (steps 3,4,3',4'). After the browser receives these web objects sent by OWPScope, the js code collects a set of timing information and sends them back to the web server (step 5). By processing such data, OWPScope obtains the one-way metrics samples between the client and each resource reserver and then redirects the client to the most suitable server.

In summary, our major contributions include:

978-1-4799-6204-4/14$31.00

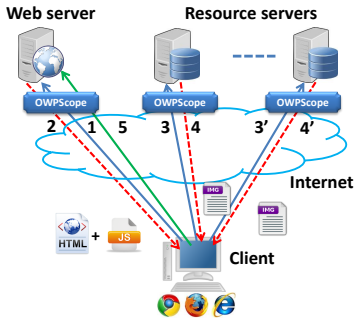


Fig. 1: Application scenario of OWPScope: selecting a suitable CDN server.

1) We propose OWPScope, a novel server-side system that can measure four important one-way path metrics without installing specific softwares or plugins at the client side.
2) To our best knowledge, OWPScope is the first system that exploits the standard features in HTML5, HTTP, and TCP together to conduct one-way path measurement. Moreover, we discover implementation deficiencies in popular browsers when evaluating the new features.
3) We implement OWPScope in 1850 lines of C and 421 lines of javascript after tackling several challenging problems, such as correlating cross-layer information and handling time resolution issues. The extensive evaluations in a testbed and through Internet show that OWPScope can effectively measure those metrics with low overhead to the hosting server and the network.

We detail the design of OWPScope in Section II. Section III reports the evaluation in a testbed and through Internet. After introducing related works in Section IV, we conclude the paper in Section V.

II. OWPScope

This section first introduces the HTML5 features exploited by OWPScope and then describes OWPScope's measurement process and methods for measuring one-way path metrics.

A. HTML5 features

Three HTML5 features, i.e., navigation timing (NT) [17], Resource timing (RT) [18] and High Resolution Time (HRT) [19], are used to collect timing information on the client side. NT and HRT are W3C recommendation (i.e., standard) and supported by major browsers, while RT is W3C's Candidate Recommendation and currently supported by IE and Chrome.

Specifically, NT provides an interface to obtain timestamps in millisecond resolution for a set of events during a web page's loading cycle [17]. Meanwhile, RT offers an interface to collect timing information associated with *each* resource within a document. From NT (or RT), OWPScope collects three timestamps: (1)*requestStart*, denoted as T_{qs}, the time immediately before the browser sends a request for a web page (or a resource); (2)*responseStart*, denoted as T_{ps}, the time immediately after the browser receives the first byte of a web page (or a resource); (3)*responseEnd*, denoted as T_{pe}, the time immediately after the browser receives the last byte of a web page (or a resource). In addition, OWPScope relies on HRT to obtain the current time in sub-millisecond resolution, which is not subject to system clock skew or adjustments [19].

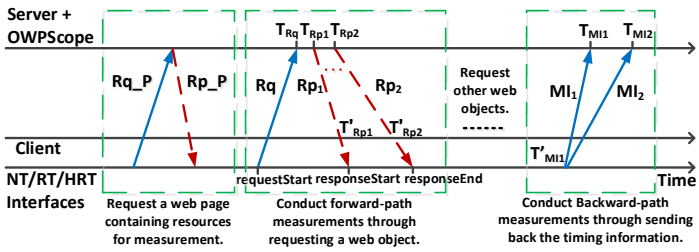


Fig. 2: The measurement process of OWPScope. The forward path is from a server to a client and the backward path is from a client to a server.

B. General measurement process

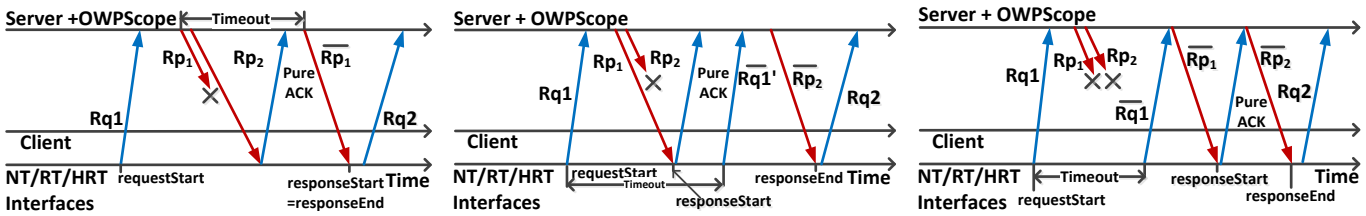
Fig. 2 illustrates OWPScope's measurement process and the collected timing information when RT is available. At the beginning, the client (i.e., $\mathbb{C}$) sends a request (i.e., Rq_P) for a web page (i.e., Rp_P) that contains several small web objects, like figures, and OWPScope's js code. For the ease of explanation, we assume that these web objects, to be fetched by the client, are in the same server as the web page.

Let $\mathcal{W}$ be one web object requested by $\mathbb{C}$ through Rq, whose sending time is recorded in T_{qs}. On the arrival of Rq, OWPScope logs its arrival time T_{Rq} and replies with 2 probing packets, which carry the content of $\mathcal{W}$, and N_u ($N_u \geq 0$) padding packets, which are dispatched between probing packets. Padding packets are the same as probing packets except that they have limited TTL values so that they will be routed through the same path as probing packets and dropped by a router a few hops away from $\mathbb{C}$.

Let T_{Rp1} and T_{Rp2} be the sending time of Rp_1 and Rp_2, and T'_{Rp1} and T'_{Rp2} denote the time when they reach $\mathbb{C}$. The browser records the time when Rp_1 (or Rp_2) is delivered to it in T_{ps} (or T_{pe}) *before* rendering $\mathcal{W}$. When Rp_P has multiple web objects, the browser will record each object's T_{qs}, T_{ps}, and T_{pe}. Finally, the js code in the web page fetches the stored values through RT and sends them along with padded content to OWPScope. The padded content is long enough so that the client will send back 2 packets (i.e., MI_1 and MI_2), whose sending times and arriving times are denoted as T'_{MIi} and T_{MIi}, respectively. After a predefined delay, the web page will be automatically reloaded for another round of measurement. If the browser only supports NT, OWPScope regards the first web page as $\mathcal{W}$. The browser will also record the arrival time of Rp_1(or Rp_2) in T_{ps} (or T_{pe}).

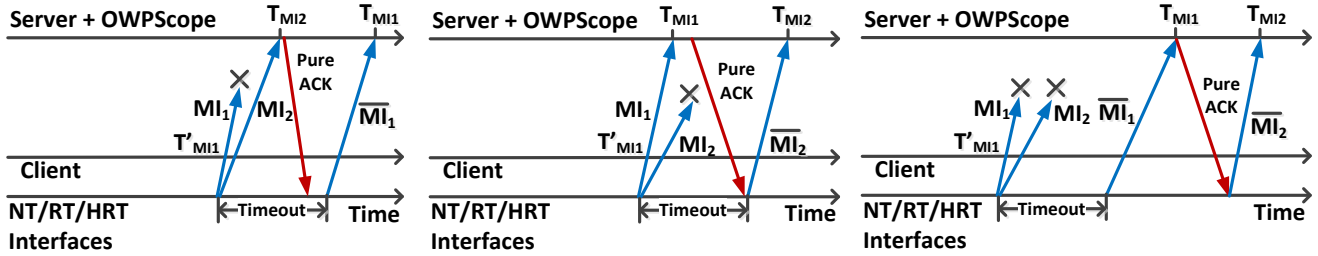
C. Metric measurement methods

One-way packet loss. It is challenging to detect the loss of probing packets because we cannot capture packets in $\mathbb{C}$. OWPScope addresses this issue by driving $\mathbb{C}$ to generate different responses in the presence or the absence of probing packets. More precisely, OWPScope instructs $Rp1$ to acknowledge part of $Rq1$. Let SN_{Rq1} and L_{Rq1} denote the sequence number and the length of $Rq1$, respectively. OWPScope sets Rp_1's acknowledgement number to $SN_{Rq1} + \frac{L_{Rq1}}{2}$ and that of Rp_2 to $SN_{Rq1} + L_{Rq1}$. As shown in Fig. 3(a), if Rp_1 is lost, Rp_2 triggers a pure ACK packet. The server retransmits Rp_1 after timeout, denoted as $\overline{Rp_1}$, and then $\mathbb{C}$ sets T_{ps} and T_{pe} with the same value (sometimes there may be negligible difference due to noise). Because of the request-response nature of HTTP, $\mathbb{C}$



(a) Rp_1 is lost. (b) Rp_2 is lost. (c) Rp_1 and Rp_2 are lost.

Fig. 3: Detecting forward-path packet losses.



(a) MI_1 is lost. (b) MI_2 is lost. (c) MI_1 and MI_2 are lost.

Fig. 4: Detecting backward-path packet losses.

can only send out $Rq2$ for next web object through the same TCP connection after $\overline{Rp_1}$ has been received.

If Rp_2 is lost as shown in Fig.3(b), a pure ACK will be sent after the delayed ACK timer expires, indicating that Rp_1 has been received. Since Rp_1 only acknowledges part of $Rq1$, $\mathbb{C}$ will retransmit the unacknowledged portion, denoted as $\overline{Rq1'}$. The server will retransmit Rp_2 after timeout (i.e., $\overline{Rq2}$). Therefore, the difference between T_{pe} and T_{ps} approximates to the server's retransmission timeout (RTO). The new request (i.e., $Rq2$) will be dispatched after $\overline{Rq2}$ is received.

If both Rp_1 and Rp_2 are lost, the client will retransmit the whole request (i.e., $\overline{Rq1}$) again, as shown in Fig. 3(c). The interval between the arrival time of $Rq1$ and that of $\overline{Rq1}$ is around $\mathbb{C}$'s RTO. The server will first retransmit Rp_1 (i.e., $\overline{Rp_1}$) that will trigger a pure ACK, and then retransmit Rp_2 (i.e., $\overline{Rp_2}$) after receiving the pure ACK. Then, the difference between T_{pe} and T_{ps} approximates to RTT. Another request (i.e., $Rq2$) will be sent after $\overline{Rp_1}$ and $\overline{Rp_2}$ arrive.

Fig.4 illustrates how to detect backward-path packet losses. It is easy as two packets will be returned and OWPScope can capture them. If MI_1 is lost, OWPScope first observes MI_2 and then the retransmitted MI_1. We use $d_M = |T_{MI2} - T_{MI1}|$ to differentiate it from the scenario when MI_1 and MI_2 is reordered, because in the former case d_M is close to $\mathbb{C}$'s RTO whereas in the latter case d_M is usually much smaller [20]. If MI_2 is lost, OWPScope first observes MI_1 and d_M is close to $\mathbb{C}$'s RTO. If both packets are dropped and retransmitted, $(T_{MI1} - T'_{MI1})$ approximates to the sum of its normal value and $\mathbb{C}$'s RTO.

One-way packet reordering. It is straightforward to detect packet reordering on the backward path because OWPScope captures all packets from $\mathbb{C}$. However, it is nontrivial to detect forward-path packet reordering since OWPScope cannot capture packets in $\mathbb{C}$. We tackle this problem by letting in-order probing packets trigger responses different from that caused by out-of-order probing packets. As shown in Fig.5, the arrival of Rp_2 will induce a pure ACK packet whose acknowledge

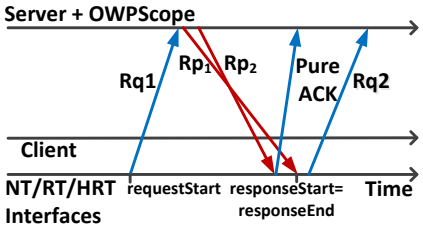


Fig. 5: Detecting forward-path packet reordering.

number equals to the sequence number of Rp_1. After receiving Rp_1, $\mathbb{C}$ sends out a new request $Rq2$. Note that OWPScope can distinguish the forward-path packet reordering from forward-path packet loss according to $Rq2$ before retransmitting any packet, because $\mathbb{C}$ cannot send it until receiving Rp_1 and Rp_2 to the current request (i.e., $Rq1$). This method is effective because the time lag of reordered packets is quite small compared to the minimal one-way delay [20].

One-way path capacity. Let C_j be the maximum number of bits that can be transmitted on the jth link. The one-way path capacity equals to $\Omega = \min\{C_j\}$, $j = 1, \ldots, L$, where L is the number of links that compose the path. OWPScope uses packet train to measure the forward-path capacity by sending $N = 2 + N_u$ packets of size S bytes back-to-back. The packet dispersion observed by $\mathbb{C}$ is $\delta_N = T'_{Rp2} - T'_{Rp1}$. However, they cannot be obtained by OWPScope because we do not control $\mathbb{C}$. Instead, we estimate δ_N using $T_{pe} - T_{ps}$ and then the capacity can be computed by Eq.(1) following [21].

$$\Omega_F = \frac{(N-1)S}{\delta_N + \varepsilon} = \frac{(1-\frac{1}{N})S}{\frac{\delta_N}{N} + \frac{\varepsilon}{N}}, \quad (1)$$

where ε denotes the noise due to the approximation.

The rational behind this packet train based approach is three-fold. First, the resolution of NT/RT (i.e., millisecond) limits the minimal δ_N that can be measured and, thus, the maximal capacity that can be measured by a packet-pair based method (i.e., $N = 2$). In contrast, δ_N can be increased by a long packet train. Second, the approximation may be biased

by the noise from OS/browser. Eq.(1) shows that the effect of noise can be mitigated by increasing N. Third, although a packet train measures the average dispersion rate (ADR) in the presence of severe cross traffic, ADR has two good properties [21]: it is independent of the length of packet train (i.e., N) so that OWPScope can increase N to mitigate the effect of noise; the effect of cross traffic can be alleviated by increasing the sending rate of probing packets. Tools like `DSLprobe` [14] have demonstrated the accuracy of packet train.

We further employ the minimum-delay-sum principle [15] to filter out biased samples. Let $d_{f1} = T_{ps} - T_{Rp1}$ and $d_{f2} = T_{pe} - T_{Rp2}$. The principle specifies that if the probing packets are affected by cross traffic, the sum of packet delays will be increased [15]. Therefore, only samples that fulfill Eq.(2) should be used to compute the forward-path capacity.

$$\min\{d_{f1} + d_{f2}\} = \min\{d_{f1}\} + \min\{d_{f2}\} \tag{2}$$

OWPScope use packet pair to measure backward-path capacity as shown in Eq.(3), because it can capture packets from $\mathbb{C}$ with high-resolution timestamp.

$$\Omega_B = \frac{S}{T_{MI2} - T_{MI1}} \tag{3}$$

Similarly, we define $d_{b1} = T_{MI2} - T'_{MI1}$ and $d_{b2} = T_{MI1} - T'_{MI1}$ and use Eq.(4) to select unbiased samples for calculating the backward-path capacity. T'_{MI1} is obtained through HRT.

$$\min\{d_{b1} + d_{b2}\} = \min\{d_{b1}\} + \min\{d_{b2}\} \tag{4}$$

One-way jitter. Let D_f = T_{ps} - T_{Rp1} and D_b = T_{Rq} - T_{qs}. Note that D_f and D_b are not one-way delays, because the server and the client are usually not synchronized. Given a sequence of D_f and D_b samples, we can compute the one-way jitter as $\theta_f(i) = D_f(i+1) - D_f(i)$ and $\theta_b(i) = D_b(i+1) - D_b(i)$. Since the clock shew in typical computer is around 1 part per million (ppm) [22], if the interval between samples is small (e.g., 100ms), the error in jitter measurement due to clock skew is negligible (i.e., 0.1μs). Otherwise, we follow the methods in [22] to remove the relative clock skews.

III. Evaluation

We have implemented OWPScope with 1,850 lines of C and 421 lines of javascript. We conduct extensive experimental evaluation to answer three questions: (1) Does major browsers well support NT and RT? (2) Does OWPScope function correctly and how is its overhead? (3) What observations can OWPScope make when measuring Internet paths?

A. Approximation accuracy of using NT/RT

Since $(T_{pe} - T_{ps})$ is used to approximate to $(T'_{Rp2} - T'_{Rp1})$, we first evaluate the approximation accuracy in two settings, including a Linux machine(i3 CPU 2.4GHZ and 8GB memory) running Ubuntu 12.04 with FireFox (v26) and Chrome (v32), and a Window machine with the same hardware running Windows 7 with IE (v11), FireFox (v26) and Chrome (v32). We delay Rp_2 by β ($\beta \in \{30, 50, 100, 150\}$ms) to emulate packet dispersion and use WireShark to capture Rp_1 and Rp_2 at the client side for calculating $(T'_{Rp2} - T'_{Rp1})$. For each setting, we run the experiment for 30 times and calculate the mean and the standard deviation. Moreover, we also examine the result after introducing 25% CPU load to the PC.

Fig. 6 shows that the majority of the differences are within [-1, 1]ms. It is acceptable as OWPScope can further increase N to mitigate the effect of noise as shown in Eq.(1). The difference obtained in Linux is usually smaller than that in Windows. The largest difference was observed when using NT within IE in Windows. In contract, the difference is not significant when RT is used in IE. It may be due to the implementation deficiencies as NT and RT are new standards.

By studying the source codes of FireFox and Chrome, we did observe some implementation issues, including (1) Chrome records `responseStart` after processing an HTTP header while Firefox does it before processing the header. It may be the reason why Chrome's difference increases with additional CPU load; (2) in Windows, Chrome uses the function `timeGetTime()` to retrieve the system time in milliseconds while Firefox uses the function `QueryPerformanceCounter()` for retrieving timestamp with higher resolution. It may be the reason why Chrome has worse performance than FireFox; (3) To ensure the timestamp increases monotonically, Chrome introduces a set of functions that will adjust the raw timestamp. These functions may introduce additional noise; (4) Some issues in Chrome might have been discovered without fix. For example, we found a "FIXME" comment in the function `responseStart()` saying that the time of `responseStart` may be delayed;

B. Controlled experiments

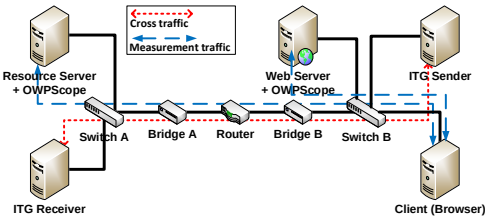


Fig. 7: The topology of the testbed.

We validate OWPScope in a testbed shown in Fig.7, where OWPScope is deployed in a resource server and a web server. A MikroTik router is used to limit the network capacity and D-ITG is employed to generate cross traffic.

Packet loss and packet reordering. To validate the detection of packet losses, we intentionally drop Rp_1 and/or Rp_2 following Fig. 8. To emulate packet reordering on the forward path, we let OWPScope send Rp_2 before Rp_1. The responses from the client in these scenarios follow our expectations.

Capacity. To evaluate OWScope's capability of measuring capacity, we change the capacity of the path between the client and the resource server, and adjust the packet train's length. We run the experiment 30 times for each setting and list the mean and the standard deviation of estimated capacities in Tab.I. The results show that OWScope can accurately estimate the capacity with small standard deviation. Moveover, a longer packet train leads to better estimates, thus validating Eq.(1).

System load. We use siege (www.joedog.org) to simulate visitors to the resource server, who generate different number

	Normal scenario				With 25% CPU load			
β (ms)	30	50	100	150	30	50	100	150
L_FF_NT	0.067/0.442	0.2/0.476	-0.033/0.18	-0.067/0.573	0.033/0.482	0.111/0.416	-0.033/0.482	0.033/0.482
L_CH_NT	0.033/0.657	0.233/0.616	0.267/0.68	0.2/0.6	0.267/0.573	0.467/0.718	0.333/0.537	0.533/0.67
L_CH_RT	0.1/0.651	0.067/0.359	0.3/0.526	0.267/0.629	0.033/0.657	0.367/0.657	0.367/0.657	0.3/0.526
W_FF_NT	0.167/0.373	0.3/0.458	0.433/0.667	0.867/0.499	0.333/0.537	0.367/0.482	0.033/0.18	0.233/0.761
W_CH_NT	0.3/1.969	-0.2/0.6	-0.067/1.611	-1.067/0.573	-1.8/0.763	-0.033/0.547	-2.533/4.209	-0.1/0.539
W_CH_RT	0.067/0.68	0.367/1.816	-0.867/1.784	-1.233/0.803	0.133/0.562	-0.133/0.921	-3.8/7.786	-0.4/0.952
W_IE_NT	-16.433/5.869	-18.6/11.005	-32.533/24.838	-41.3/38.569	-17.4/5.851	-27.267/9.465	-45.633/19.541	-54.9/34.388
W_IE_RT	-0.133/0.806	0.067/0.359	-0.433/0.989	-0.067/0.892	0.133/0.718	-0.133/0.427	-0.1/0.473	-0.533/1.087

Fig. 6: Approximation accuracy of using NT/RT in different OS/browsers with/without intentionally introduced CPU load. Each cell contains the average value in millisecond and the standard deviation. L: Linux; W: Windows; FF: FireFox; CH: Chrome; IE: Internet Explorer. FireFox (v26) does not support RT.

of packets (i.e., 10, 30, 100). For each setting, siege runs for 10 minutes and we log the average load at the end of each minute. Tab.II lists the mean of the ten results, showing that OWScope introduces light overhead to the hosting server.

TABLE I: Capacity measurement in the testbed. N is the length of packet train and Ω_F is the estimated capacity.

	2Mbps		5Mbps		10Mbps	
N	10	30	30	50	50	100
Ω_F	2.17/0.19	2.07/0.05	5.61/0.37	5.35/0.15	10.09/0.2	10.02/0.07

TABLE II: Load of the resource server.

Number of users	10 packets	30 packets	100 packets
50	0.04	0.046	0.08
100	0.045	0.056	0.085

C. Internet experiments

Capacity. Following Fig. 1, we set up a web server in our campus network with limited capacity of 5Mbps and deploy four resource servers in Amazon EC2, which are located in Singapore (SG), California (US), Tokyo (JP), and Sao paulo (BR), individually. OWPScope is deployed on those servers and uses RT to perform the measurement. We run IE 11 on window 8.1 and Chromium 32.0 on Ubuntu 12.04 from a residential network to visit the web server's front page, which includes images in different resource servers. The download capacity of the residential network is 10Mbps (i.e., the forward-path capacity).

Due to limited pages, we only report the result for forward-path capacity as shown in Tab. III. Since the estimated capacity of the four Internet paths are all around 10Mbps, the bottleneck may be the residential network. As the web server has smaller capacity (i.e., 5Mbps), it becomes the bottleneck of that path. Tab. III shows that the estimation accuracy increases with N, which is in consistent with Eq.(1), and two browsers lead to similar results.

TABLE III: The estimated forward-path capacity (in Mbps) of five Internet paths. Each cell has two average values obtained from Chrome and IE.

N	SG	US	JP	BR	Campus
32	11.27/10.99	11.25/10.69	10.98/10.68	11.07/11.15	5.55/5.27
42	10.30/10.10	10.49/10.84	10.40/11.02	10.27/10.61	5.25/5.40
52	10.33/10.24	10.07/10.32	9.96/10.11	9.61/10.05	5.22/5.21

TABLE IV: Other systems' results and their traffic consumption.

Tool	Traffic volume(MB)	Number of packets	Ω_F (Mbps)
speedtest	38.6	40,336	9.14
npad	30.70	21,826	8.12
netalyzr	98.86	198,936	9.29
boomerang	2.06	2,111	1.78

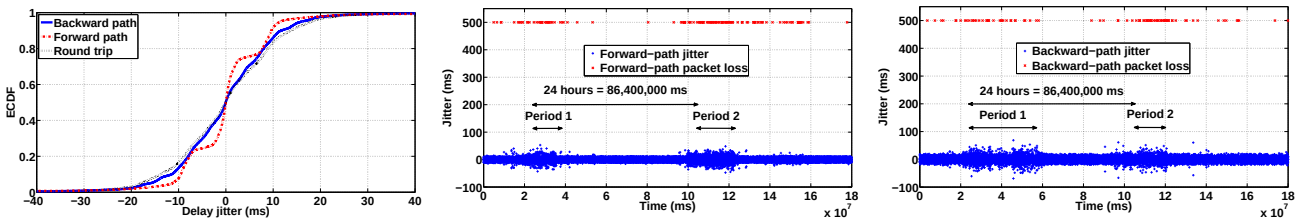
For comparison, we use other tools, including Speedtest, NPad, Netalyzr and Boomerang, to estimate the capacity from their servers to the same client. Speedtest selects one of its servers in the same region. NPad's server is hosted by M-Lab and Netalyzer has its own server. Since boomerang requires the user to set up a server, we deploy it on an EC2 host in US. For each tool, the measurements were repeated for 10 times and the average values for the traffic volume, number of packets, and the estimated capacity are computed and shown in Tab. IV. While Speedtest and Netalyzr can achieve better performance in capacity measurement than NPad and boomerang, their accuracies are still lower than that of OWPScope. Speedtest generated around 40MB traffic for estimating RTT and upload/download speed. In contrast, OWPScope can measure multiple one-way path metrics with much fewer packets. Although boomerang only generated around 2MB traffic, its estimation is not reliable. Since Netalyzr conducted many other measurements besides capacity estimation, it generated almost 200MB traffic, consuming much bandwidth.

Path performance over time. We deploy OWPScope and a web server on an EC2 host in US, and launch a Chrome browser in our campus network to periodically visit the server for two days. As shown in Fig.8(a), the forward-path jitter does not have the same distribution as the backward-path jitter and the round-trip jitter. Note that knowing one-way jitter is useful for services sensitive to it (e.g., online streaming). Fig.8(b) and Fig.8(c) show the time sequence of one-way jitter and packet loss. Both metrics demonstrate a diurnal pattern (e.g., period 1 and period 2 in both figures). Moreover, there is an obvious correlation between jitter and packet loss (i.e., larger jitter accompanied with more packet loss). Note that the forward path and the backward path exhibited different performance.

IV. Related work

NT and RT have been quickly adopted by the industry. For example, Google uses them to measure "perceived latency" and provides site speed reports [23], [24]. Yahoo adds the support of NT in boomerang. However, to our best knowledge, OWPScope is the first system exploiting NT/RT for measuring low-level one-way path metrics.

Although several server-side measurement systems have been proposed, none of them can measure one-way metrics like OWPScope. `WhyHigh` measures client latencies across CDN servers to identify the prefixes suffering from inflated latencies [2]. `Festive` uses the throughput measured by a media player to guide the bitrate selection for steaming [4]. `QDash` employs coarse available bandwidth measurement to facilitate the section of video quality level [3]. Rajamony et al.



(a) One-way delay jitter and rout-trip delay jitter. (b) Forward-path delay jitter and packet loss. (c) Backward-path delay jitter and packet loss.

Fig. 8: One-way delay jitter and one-way packet loss over time.

used HTML4 and js to measure user perceived response time [25]. Janc et al. employed js and Flash to estimate download throughput and RTT [11]. `Fathom`, a Firefox extension, supports launching network measurements through js [12] while `Netalyzr` uses Java Applet to do similar tasks [9].

While some non-cooperative tools have been developed to measure one-way metrics, the majority of them were designed as a client-side tool without considering the requirements of server-side measurement [13]–[15], [26]–[29]. For example, client's firewall will filter out unsolicited TCP/UDP/ICMP packets and thus renders some tools [13]–[15] useless. Some tools only support one or two types of one-way metrics (e.g., `Sting` for packet loss [26], `CapProbe` for packet reordering [15]). Although `TRIO` can measure one-way capacity on top of `OneProbe`, the estimation of forward-path capacity may be affected by the noise in the reverse path [28]. In summary, none of these tools has the same capability as OWPScope.

V. Conclusion

We designed and implemented OWPScope, a novel server-side system for measuring low-level one-way path metrics *without* installing specific softwares or plugins at the client side. It can penetrate client-side firewalls and conduct representative measurement by using packets carrying real application data. Its capability results from exploiting the standard features in HTML5/HTTP/TCP and correlating the information from different layers. We have implemented OWPScope and the extensive evaluations in a testbed and through Internet show that OWPScope can effectively measure the metrics with low overhead to the hosting server and the network.

VI. Acknowledgment

We appreciate the reviewers for their comments, and thank ChengYan Wang and Guihua Wang for their contributions to the preliminary version of OWPScope. This work is supported in part by the CCF-Tencent Open Research Fund, the Hong Kong GRF (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396,60903185), and the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province.

References

[1] H. Balakrishnan, V. Padmanabhan, G. Fairhurst, and M. Sooriyabandara, "TCP performance implication of network path asymmetry," RFC 3449, IETF, 2002.

[2] R. Krishnan, H.Madhyastha, S. Srinivasan, and S. Jain, "Moving beyond end-to-end path information to optimize CDN performance," in *Proc. ACM IMC*, 2009.

[3] R. Mok, X. Luo, E. Chan, and R. Chang, "QDASH: a QoE-aware DASH system," in *Proc. ACM MMSys*, 2012.

[4] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in HTTP-based adaptive streaming with Festive," in *Proc. ACM CoNext*, 2012.

[5] R. Mahajan, M. Zhang, L. Poole, and V. Pai, "Uncovering performance differences in backbone ISPs with Netdiff," in *Proc. NSDI*, 2008.

[6] M. Z. Ying Zhang, Z. Morley Mao, "Detecting traffic differentiation in backbone isps with NetPolice," in *Proc. ACM IMC*, 2009.

[7] P. Kanuparthy and C. Dovrolis, "Pythia: Diagnosing performance problems in wide area providers," in *Proc. USENIX ATC*, 2014.

[8] "perfsonar," http://www.perfsonar.net/.

[9] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the edge network," in *Proc. ACM IMC*, 2010.

[10] "Network diagnostic tool (NDT)," http://goo.gl/DXl2z8.

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

[12] 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 IMC*, 2012.

[13] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson, "User-level Internet path diagnosis," in *Proc. ACM SOSP*, 2003.

[14] D. Croce, T. En-Najjary, G. Urvoy-Keller, and E. Biersack, "Capacity estimation of ADSL links," in *Proc. ACM CoNEXT*, 2008.

[15] R. Kapoor, L. Chen, L. Lao, M. Gerla, and M. Sanadidi, "CapProbe: A simple and accurate capacity estimation technique," in *Proc. ACM SIGCOMM*, 2004.

[16] J. Sommers, P. Barford, N. Duffield, and A. Ron, "A framework for multi-objective sla compliance monitoring," in *Proc. INFOCOM*, 2007.

[17] W3C, "Navigation timing," http://www.w3.org/TR/navigation-timing/, 2012.

[18] ——, "Resource timing," http://www.w3.org/TR/resource-timing/, 2012.

[19] ——, "High resolution time," http://www.w3.org/TR/hr-time/, 2012.

[20] X. Zhou and P. Mieghem, "Reordering of IP packets in Internet," in *Proc. PAM*, 2004.

[21] C. Dovrolis, P. Ramanathan, and D. Moore, "Packet dispersion techniques and a capacity-estimation methodology," *IEEE/ACM Trans. Netw.*, vol. 12, no. 6, 2004.

[22] T. Kohno, A. Broido, and K. Claffy, "Remote physical device fingerprinting," *IEEE TDSC*, vol. 2, no. 2, 2005.

[23] I. Grigorik, "Measuring site speed with navigation timing," http://www.igvita.com, 2012.

[24] ——, "Measuring network performance with resource timing api," http://www.igvita.com/, 2013.

[25] R. Rajamony and M. Elnozahy, "Measuring client-perceived response time on the WWW," in *Proc. USENIX USITS*, 2001.

[26] S. Savage, "Sting: A TCP-based network measurement tool," in *Proc. USENIX USITS*, 1999.

[27] X. Luo, E. Chan, and R. Chang, "Design and implementation of TCP data probes for reliable and metric-rich network path monitoring," in *Proc.USENIX ATC*, 2009.

[28] E. Chan, A. Chen, X. Luo, R. Mok, W. Li, and R. Chang, "Trio: Measuring asymmetric capacity with three minimum round-trip times," in *Proc. ACM CoNEXT*, 2011.

[29] L. Xue, X. Luo, E. Chan, and X. Zhan, "Towards detecting target link flooding attack," in *Proc. USENIX LISA*, 2014.