APPRAISING AND IMPROVING THE ACCURACY OF SMARTPHONE AND BROWSER-BASED DELAY MEASUREMENT

WEICHAO LI

Ph.D

The Hong Kong Polytechnic University

2017

The Hong Kong Polytechnic University
Department of Computing

Appraising and Improving the Accuracy of Smartphone and Browser-based Delay Measurement

Weichao Li

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

October 2016

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

____ (Signed)

Weichao Li

____ (Name of student)

To my wife and my family.

Abstract

In recent years, using network performance benchmarking tools, such as Ookla Speedtest, to understand network performance has become popular. These measurements are available in desktop environment via Web browsers (browser-based measurements) and mobile environment via mobile apps (smartphone-based measurements), which provides a very effective approach to crowdsourcing network performance data. However, despite their popularity, very little is known about the (in)accuracy of the various methods used by these tools.

In this research, we contribute to examining and improving the accuracy of these tools. We focus on the network delay measurement, because it is the most fundamental and atomic metric. To quantify the inaccuracy, we define "delay overhead" as the difference between the value measured by the measurement tools and the actual network delay. The delay overhead, which is considered as measurement noise, should be avoided and mitigated during delay measurements to achieve more accurate results.

In browser-based measurement, we evaluate the accuracy of twelve methods, including the traditional HTTP-based and TCP socket-based methods, Navigation Timing and WebRTC techniques, with the eight most popular browsers on Linux and Windows. Our evaluation results show that the delay overheads incurred in most of the HTTP-based methods are too large to ignore. Specifically, 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. This results in a serious under-estimation of RTT. Moreover, some tools over-estimate the RTT by including the TCP handshaking or data channel establishing phase.

For the mobile network measurement, we show that the two most popular measurement apps—Ookla Speedtest and MobiPerf—have their RTT measurements inflated. We then build three corresponding test apps that cover three common measurement methods and evaluate them in a testbed. We overcome the main challenge of obtaining a complete trace of packets and their timestamps by using multiple sniffers and frame-based synchronization. Our multi-layer analysis reveals that the delay inflation can be introduced both in the user space and kernel space. The long path of subfunction invocations accounts for the majority of the delay overhead in Android runtime (both Dalvik VM and ART), and the sleeping functions in the drivers are the major source of the delay overhead between the kernel and physical layer.

Based on our evaluation in smartphones, we try to reduce and stabilize the delay inside the phone as much as possible. We report for the first time a major source of noise comes from the periodical SDIO (Secure Digital Input Output) bus sleep inside the phone. Besides, the PSM (Power Saving Mode) for WiFi networks and the RRC (Radio Resource Control) states in cellular networks will also unstably inflate the delay measurement. To mitigate these measurement noises, we propose to keep the phone in the wake-up mode or high-power state during the delay measurement by sending just a sufficient amount of warm-up and background traffic. We implement this approach in AcuteMon, an Android app, and validate it in testbed and in real users' phones. The evaluation shows that AcuteMon can effectively mitigate the delay overheads caused by various energy-saving mechanisms, and obtain more accurate network delay.

Publications Arising from the Thesis

Weichao Li, Daoyuan Wu, Rocky K. C. Chang, Ricky K. P. Mok, "Toward Accurate Smartphone-based Mobile Network Measurement", submitted to *IEEE Transaction on Mobile Computing (Under review).*

Weichao Li, Daoyuan Wu, Rocky K. C. Chang, Ricky K. P. Mok, "Demystifying and Puncturing the Inflated Delay in Smartphone-based WiFi Network Measurement", In *Proc. ACM CoNEXT (short paper)*, December 2016.

Weichao Li, Ricky K. P. Mok, Daoyuan Wu, and Rocky K. C. Chang, "On the Accuracy of Smartphone-based Mobile Network Measurement", In *Proc. IEEE INFOCOM*, April 2015.

Weichao Li, Ricky K. P. Mok, Rocky K. C. Chang, and Waiting W. T. Fok, "Appraising the Delay Accuracy in Browser-based Network Measurement", In *Proc. ACM IMC (short paper)*, October 2013.

Acknowledgements

This thesis is the end of my Ph.D journey and it would not have been possible for me to finish it without the inspiration and support of many people, to whom I would like to express the deepest appreciation.

Foremost, I would like to express my sincere gratitude to my advisor Prof. Rocky K. C. Chang for his patience, enthusiasm, and motivation. I could not have imagined having a better advisor and mentor for my Ph.D study. His professional guidance and useful recommendations helped me in all the time of research and writing of this thesis. Without his continuous support this Ph.D would not have been achievable.

Besides my advisor, I would like to thank my thesis committee members: Prof. Gary S.H. Chan of the Hong Kong University of Science and Technology, Prof. Wing Cheong Lau of the Chinese University of Hong Kong, and Prof. Lou Wei of the Hong Kong Polytechnic University, for their insightful comments and hard questions.

I would also like to thank Daniel Xiapu Luo for his mentorship, especially at the early stage of my study. I have greatly benefited from his expertise and experiences. I also gratefully acknowledge the past and current members in the Internet Infrastructure and Security Research Laboratory: Ricky Mok, Waiting Fok, Daoyuan Wu, Edmond Chan, Brent Peng Zhou, Toby Lam, Star Poon, Anson Kwan, Steven Chien, Curtis Yung, Jack Chan, Yanto Lam, Peter Membrey, Ang Chen, Lei Xue, Wei Yu, Peixin Chen, Qingjie Xu, and many more. Thank all of you for your warm friendship and your unfailing encouragements.

Last but not the least, my deep and sincere gratitude goes to my family. I am forever indebted to my parents for encouraging me to explore new directions in life and supporting me in all my pursuits. I am grateful to my brother, Dechao Li, for his unconditional support and help. I particularly thank my dearest wife, Huang Hui, for her love and care throughout these years. Thank you.

Contents

List of Figures

List of Tables

List of Abbreviations

AP Access Point.

API Application Programming Interface.

ART Android Runtime.

AS Autonomous System.

CDF Cumulative Distribution Function.

CDN Content Delivery Network.

CPU Central Processing Unit.

DNS Domain Name System.

DOM Document Object Model.

DRX Discontinuous Reception.

DTLS Datagram Transport Layer Security.

DVM Dalvik Virtual Machine.

e2e end-to-end.

EDGE Enhanced Data rates for GSM Evolution.

GPRS General Packet Radio Service.

HTTP Hypertext Transfer Protocol.

HTTPS HTTP Secure.

ICMP Internet Control Message Protocol.

IP Internet Protocol.

ISP Internet Service Provider.

JNI Java Native Interface.

LTE Long-Term Evolution.

MEM Memory.

MLME MAC Sublayer Management Entity.

MSS Maximum Segment Size.

MTU Maximum Transmission Unit.

NAT Network Address Translation.

NIC Network Interface Card.

OS Operating System.

OWD One-Way Delay.

PDCCH Physical Downlink Control Channel.

PSM Power Saving Mode.

QoE Quality of Experience.

QoS Quality of Service.

RRC Radio Resource Control.

RTT Round-Trip Time.

SCTP Stream Control Transmission Protocol.

SDIO Secure Digital Input Output.

SDN Software Defined Network.

TCP Transport Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

UMTS Universal Mobile Telecommunications System.

UTRAN UMTS Terrestrial Radio Access Network.

VM	Virtual Machine.
VP	Vantage Point.
WNIC	Wireless Network Interface Card.
WWW	World Wide Web.

Chapter 1

Introduction

With the fast growth of the Internet, residential broadband networks and mobile networks have already become an essential part of our daily lives. According to Internet World Stats [41], the number of Internet users across the world is more than 3.6 billion in July 2016, accounting for a half of the global population. ITU (International Telecommunication Union) also shows that almost 1 billion households in the world have Internet access, and the penetration rate of mobile-broadband networks reaches 84% of the global population [116]. Therefore, measuring and understanding the network performance is becoming very important for both end users and practitioners. For example, network administrators can take the advantage of network measurement to diagnose network failures and make better management decisions [88, 154]. Network application developers can benefit from network measurement on providing better performance to their customers [50, 123, 161, 195]. End users can employ network measurement to verify the compliance of service level agreement (SLA) or real network performance [127,152,208,218]. For network scientists and researchers, network measurement can help study and model the evolution of Internet [158, 169, 200], as well as developing and validating new network protocols and techniques [72].

Network measurements can be classified into passive and active measurements.

Passive methods monitor the network performance by analyzing the existing traffic on some vantage points inside the network. Active methods, on the other hand, generate and inject measurement traffic into the monitored network. As active measurement provides more opportunities to sample the network paths that are invisible to the passive vantage points, and is more flexible in deployment, we only consider active measurement in this thesis. Although many active network measurement methodologies and tools have been proposed in the past decades, such as sting [196], tulip [152], TCP Sidecar [202, 203], HTTP/OneProbe [146], and TRIO [64], applying them to the real-world scenario is still challenging. There are a few challenges to their deployment. First of all, some tools employ protocols that are not representative of the current network applications. For example, the tools based on ICMP echo requests and responses can produce different measurement results from those obtained by TCP [227]. Second, some tools may have strict running constraints and are difficult to deploy to real-world environments. A typical case is the capacity measurement tools based on packet pair or packet train techniques. They cannot be applied to mobile networks and smartphones directly because of the proportional fair scheduling used by base stations [159]. Finally, normal users usually lack the expertise to conduct meaningful measurements and interpret the measurement results, which also prevents the tools from large-scale deployments.

Recently, using network performance benchmarking tools to understand network performance becomes popular. These tools are available in desktop environment via Web browsers and in mobile environment via mobile apps, providing the functionalities of measuring round-trip delay (RTT) and uplink/downlink throughput of an end-to-end (e2e) network path. Examples of *browser-based* measurement tools include Ookla's speedtest [171], Netalyzr [134], How's My Network (HMN) [189], Speedof.me [30], and Fathom [76]. The *smartphone-based* tools support the major mobile OSes, such as Android, including MobiPerf [17], Netalyzr [18], Internet Speed Test [12], and Speedtest.net for Android [34], iOS, including Speedtest X HD for IOS [32] and Speedtest.net for iOS [33], and Windows Phone, including Network Speed Test [22] and Speedtest.net for Windows Phone [35]. Running these tools is very simple. What a user needs to do is just opening the test page in browser or launching the app on smartphone and starting the measurement by one click. The results will be presented in table or graph with readable explanation. Due to its high usability design and the ubiquity of Web browsers and smartphones, these tools attract a large number of users. For example, Ookla's Speedtest has recorded more than 8 billion tests, and its Speedtest app also has reached over 50 million downloads in the Android app market. Internet Speed Test from V-SPEED [13] has also more than 5 million installs. As the tools enable a very effective approach to crowdsourcing network performance data, they are also employed by researchers to conduct measurement studies, such as characterizing LTE networks [108] and optimizing mobile application performance [229]. The data collected by `Speedtest.net` is used for comparing the performance between cellular and WiFi networks [206].

Although the network performance benchmarking tools are widely in use today, very little is known about their (in)accuracy. Ordinary users usually do not doubt the trustworthiness of the measurement results. However, since the tools work in the application layer and that more fine-grained packet-level behavior is invisible to them, the performance metrics are inferred from the events captured in the application layer. Therefore, the tools could be affected by other tasks running in the same machine. Moreover, for the tools employing Java applet techniques or running in Android, the measurements took place in the virtual machine (VM). Therefore, the programs need to be first translated into bytecode before execution, which could introduce more overheads. Another possible source of inaccuracy that we have found is the power saving mechanisms adopted by most of the mobile devices to increase the battery life cycle. The periodical switching between active and inactive states could also lead to inaccurate measurement results. After all, they are measuring the application-level performance instead of the actual network performance.

Although the application-level performance does reflect what a user may experience, we argue that it is not equivalent to the network-level performance. Network-level performance is important for many reasons, such as for operators to know their network performance and for users to diagnose whether the network or their device is responsible for performance degradation. Moreover, the performance data, when collected from different ISPs and plans, can be used to compare their levels of services. In this thesis, we examine and improve the accuracy of the performance benchmarking tools when measuring actual network delay. We focus on the network delay measurement, because it is the most fundamental and atomic metric, from which rich information can be inferred. If network delay cannot be estimated accurately, other performance metrics, such as delay variation or jitter, capacity, and available bandwidth, will also be affected. Moreover, the network delay today is getting smaller. For example, FCC reports in 2016 that the average latency for each monitored ISP in US only ranges from 12 ms to 58 ms [87]. Therefore the results with an inaccuracy of several milliseconds could severely under- or over-estimate the actual network status, and could result in wrong management decision.

In the rest of this chapter, we introduce the benchmarking tools and their core methods in Section 1.1. Next, we introduce the methodologies to appraise the accuracy of network delay measurement in Section 1.2. After proposing our solution to improve the accuracy of smartphone-based delay measurement in Section 1.3, we summarize the contributions and organization of this thesis in Section 1.4 and Section 1.5, respectively.

1.1 Network performance benchmarking tools

The explosive growth of the Internet results in complexity in both network topology and performance management. Users are served by different Internet service providers (ISPs) and interconnected by a sea of network devices. Network traffic from one autonomous system (AS) has to traverse multiple domains, and intercepted and processed by various middleboxes and QoS policies before reaching the destination. Measuring the network performance, especially the quality of an e2e network path, is therefore a challenging and important topic in recent years. Many performance metrics, such as connectivity [155], delay [48, 49, 75], loss [48, 132], reordering [164], and capacity [70, 157], have been defined by the IP Performance Metrics Working Group (IPPM WG) [14] of the Internet Engineering Task Force (IETF) [38]. In addition, a number of measurement methodologies and tools have been proposed [55, 64, 104, 146, 199, 221], and some of them are employed by many measurement platforms, such as NIMI [178], CAIDA's Archipelago (Ark) [63], DIMES [200], ETOMIC [162], Planetopus [142], and PerfSONAR [102].

Compared to the existing measurement tools, the network performance benchmarking tools have several advantages, which can be summarized as follows.

Usability. The performance benchmarking tools are very easy to use. Users can utilize them to assess their network quality by just clicking a "start" button. The measurement results are then presented in graph or other ways that are easy to understand without requiring networking knowledge. Different from many measurement tools, the browser-based tools are OS-independent and can work in most of the Web browsers for most of the desktop OSes, such Windows, Ubuntu and MacOS without additional program installation, whereas smartphone-based tools (measurement apps) are available in the application stores of the major mobile OSes, and can be easily downloaded.

Low cost of deployment. Deploying active network measurement usually involves a number of tasks, such as deploying the source and destination nodes, scheduling the measurement execution times, and collecting measurement results. The total cost of deployment depends on the number of dedicated devices and the effort of configuring the devices and measurement tools. The performance benchmarking tools initiate network path measurements from commodity desktop PCs or smartphones of normal end users instead of professional equipments. On the other hand, the remote end points are usually Web servers. The only requirement for being a measurement target is hosting some special files. Some tools (e.g., MobiPerf) can even use the existing files on the server. The low-cost deployment makes it possible to conduct large-scale network measurement.

Representativeness. Most performance benchmarking tools rely on the Web techniques (e.g., HTTP request/response messages). Considering World Wide Web (WWW) is the most popular application on top of TCP, sampling network path performance with TCP packets can reflect more accurately what other real-world applications may experience than with other protocols, such as ICMP.

The browser-based and smartphone-based tools usually measure the RTT and uplink/downlink throughput to some fixed remote endpoints. During the RTT measurement, the tools send out a request packet to the destination to trigger a response packet back. By recording the packet sending and receiving time, the RTT of the network path can be estimated. The request and response packet can be HTTP GET/POST request and response messages, used by Speedofme [30], TCP control messages, such as TCP SYN/SYN ACK or SYN/RST by Speedchecker [12], and customized TCP data by Ookla Speedtest [34]. Some tools also support UDP and ICMP (such as MobiPerf [17]). Similarly, the uplink (downlink) throughput can be inferred by sending out (inducing) a large file to (from) the server. The file can be also transferred through

HTTP, TCP, or UDP. Based on the file size and the elapsed time, the throughput can be calculated.

For browser-based tools, the packet sending and receiving actions can be implemented through the browser built-in techniques and third-party plug-ins. The tools employing browser techniques, including XMLHttpRequest (XHR), Document Object Model (DOM) element, WebSocket, Navigation Timing, and WebRTC, can be supported by most browsers natively. But for those utilizing third-party plug-ins, such as Adobe Flash and Java applet, they require the installation of the plug-ins first. According to [45], the penetration rates for Flash and Java applet are as high as 99% and 73%, respectively. This suggests that the tools based on the third-party plug-ins are also very well supported in practice.

For smartphone-based tools, we consider only Android smartphones because of their popularity. According to [112], Android dominates the market with an 87.6% share in 2016 Q2. Furthermore, the open-source manner of Android allows us to analyze the source code of the system. Our research covers three measurement methods, i.e., using HTTP-based Java classes to send HTTP GET requests, network-related classes to send TCP SYNs, and `ping` commands external to Java to send ICMP Echo requests. With the concern of the fragmentation problem of Android system [163], we also examine the performance diversity in multiple smartphones with different Android version, including Android 4, 5, and 6.

1.2 Appraising the accuracy of delay measurement

The main objective of this thesis is to study whether the network performance benchmarking tools can produce accurate results. In particular, we focus on network delay measurement, because it is the most fundamental and atomic metric, from which rich information and many other metrics (e.g., delay variation and capacity) can be derived.

Take capacity as an example, accurately obtaining the network delay is the prerequisite to calculate the minimum delay sum or minimum delay difference, and further infer the link capacity [64, 65]. A delay difference of a few milliseconds can bias the result in megabyte per second level.

Minimizing mobile network latency is very important to many time-sensitive network services, notably instant communication, video steaming, and mobile gaming. Active network measurement is often used for detecting performance degradation, performance troubleshooting, and server deployment. All of them assume reliable and accurate network measurement, such as network RTT. The user-perceived latency comprises the latency in the phone (e.g., individual app performance), network RTT, and server latency. While the server latency can be monitored by the app/content distribution network (CDN) service provider, the other two cannot be easily segregated. It is because the latency reported by a user-level measurement app includes both latency components. The main contribution of this work is to obtain the network delay as accurately as possible from a user-level network measurement app.

For CDNs, their main concern is the network performance experienced by their users. The additional delay inside the phones, which highly depends on the models and their operating states, is considered as measurement noise which should be mitigated as much as possible. Inaccurate network delay measurement could therefore lead to suboptimal cache assignment and incorrect performance diagnosis [107, 136, 225].

Inaccurate network measurement could also directly affect the user experience of the apps. Giant app providers, such as Facebook and WeChat, deploy measurement methods in their own apps, with the aim of studying, analyzing, and debugging their users' network experience in different locations worldwide. Facebook provides in-app library for categorizing the quality of the network [20]. Developers can use the results from the `Connection Class` to adjust the app's behavior, such as requesting lower quality video/image and throttle type-ahead. [10]. Inflated network RTT could therefore lower the quality of the requested video/image. The same problem could occur to QDASH (QoE-aware Dynamic Adaptive Streaming over HTTP) which uses the network RTT to adjust the video bitrate [161]. Similar to Facebook, WeChat's in-app library, Mars [37], measures network RTTs using ping and TCP [1] in its network monitoring component called SDT [2,3]. Mars library is also very popular among other app developers (the Mars GitHub page [37] has been marked and forked by around 8K and 1.5K developers, respectively).

In this thesis, we find that it is not uncommon to have the network delay being inflated by 10ms or more using the three typical measurement methods. In some cases, they could even be close to 20ms. In order to understand whether this scale of delay inflation will have an impact on the measurement of typical mobile network latency, we have analyzed the latency dataset obtained by MopEye [228], a measurement app to obtain network delay for each active app in a smartphone. The MopEye app was so far downloaded to more than 4,000 smartphones across 126 countries. From May 2016 to January 2017, we have collected over 5 million delay measurements for more than 6,000 apps in both WiFi and cellular networks. The measurement results show that the median delay experienced by all apps is 65ms. The median RTTs for popular apps are even smaller. For example, the median RTTs for Facebook, YouTube, and WeChat are only 42ms, 32ms, and 36ms, respectively. With a delay inflation of 10ms, the error of estimating the network latency using a user-level measurement could be off by over 30% (i.e., 10ms inflation for an actual RTT of 32ms).

To quantify the accuracy of delay measurement, we follow the ISO standard 5725 [42], in which two terms "trueness" and "precision" are used to describe the accuracy of a measurement method. "Trueness" refers to the closeness of the test results and the true or accepted reference value. "Precision" refers to the closeness of agreement between test results. We therefore define *delay overhead* as the difference between the value measured by the measurement tools and the actual network delay. A smaller and stable delay overhead suggests that the measurement is more accurate and the measurement tool is more reliable.

The most challenging part in the delay overhead evaluation is how to accurately capture the actual network delay. For desktop PCs, the actual network delay can be approximated by the timestamps fetched in the kernel, because the delay between the kernel and the NIC is negligible. However, this approach is not applicable in smartphone-based measurement. The reason is that the Ethernet frames have to be translated into other types of frames (e.g., IEEE 802.11 wireless frames), which could introduce additional delays. Moreover, mobile devices usually adopt energy saving mechanisms to extend the battery life cycle. It is also possible that the packets are buffered in the wireless network interface card (WNIC) or the access point (AP) during the sleeping period.

A simple way to obtain the actual network delay in WiFi networks is listening to the WNIC of the AP [179, 204]. However, it is difficult to extract the exact time that a packet is transmitted over the air medium, because a variable delay could be introduced after the driver passes the packet to the firmware due to the queuing, carrier sensing/back-off, or retry. It is challenging to solve the problem, e.g., employing APs that support WNIC hardware timestamp, modifying the driver, and rebuilding kernel. Our solution is to build a testbed containing multiple sniffers. When the packets are arrived at the air medium, the timestamps will be captured by the sniffers and thus the actual network delay can be calculated. By performing trace merging and frame-based synchronization, we can obtain a complete packet trace for calculating the actual network RTTs.

Another objective in this thesis is to improve the accuracy for those performance benchmarking tools. A prerequisite to achieve the goal is locating the exact places that result in the inaccuracy and find out the root cause. We therefore introduce the multi-layer analysis, dividing the delay overhead into several more fine-grained components. Based on the analysis, we can examine whether the user space or kernel space accounts for the majority of the delay inflation. Our analysis shows that for browser-based measurements, the extreme delay overheads for some methods, such as Flash HTTP and WebRTC methods, are caused by that the connection or channel establishing time is included in the measurement results. Moreover, the coarse granularity of timing functions in Java applet results in significant over- or under-estimation of the RTTs. For smartphone-based measurements, the delay inflations can be introduced both in the user space and kernel space. The inefficient function calls in Android runtime is the major source of the user-level overheads, and the energy-saving mechanisms adopted by the smartphones contribute the majority of the kernel-level overheads. We will further discuss it in the next section.

1.3 Improving the accuracy of smartphone-based delay measurement

One of our objectives in this thesis is to improve the accuracy of network performance benchmarking tools. In particular, we focus on smartphone-based measurements. Different from browser-based measurement, which source of delay inflation is in the user space mainly, the delay overheads in smartphone can be introduced in user space, kernel space, and even externally. Since the inefficient function calls in Android runtime (both Dalvik VM and ART) contribute the majority of delay overhead in user space, it can be mitigated by implementing the measurement method in a native Linux binary. But for the overheads caused by various energy-saving mechanisms, they are difficult to remove.

For WiFi networks, the Secure Digital Input Output (SDIO) bus sleeping introduces additional network delay inside the smartphone, and the IEEE 802.11 Power Saving Mode (PSM) externally inflates the delay between the phone and the access point (AP). Furthermore, for cellular networks, it is well known that the low-power RRC (Radio Resource Control) states will cause additional delay. Although there are many works on addressing the impact of PSM [58, 103, 183, 194], they concentrate more on packet scheduling on the AP side to achieve a better balance between energy consumption and network delay. Similarly, previous studies on RRC state transition [108, 184, 190] have not proposed any concrete algorithms to effectively mitigate the impact of RRC states on nRTT measurement.

We accordingly try to improve the accuracy of smartphone-based delay measurement by mitigating the impact of the energy-saving mechanisms. The solution should meet a number of requirements. First of all, it should be cost-effective. Here, "cost-effective" means that the solution can effectively mitigate the delay overheads without cost too much system and network resources. Moreover, it should work across different smartphone models and Android versions. Secondly, the solution should be independent to other program. For example, the background traffic introduced by other apps may activate the smartphone, but we cannot rely on these random events. Finally, as normal users do not have the expertise to change the system and network configuration, the solution should require minor or no system modification, as well as "root" privilege.

1.4 Contributions

The contributions of this thesis are summarized below.

1. **Measurement studies to appraise the accuracy of browser-based and smartphone-based delay measurement.**

   We have conducted testbed experiments to evaluate the delay overheads introduced in the browser-based measurements. We have tested twelve Web or browser techniques that have already been employed or can be adopted for implementing browser-based network measurement tools, including the traditional HTTP-based and socket-based methods, Navigation Timing technique, and WebRTC technique. Our evaluation covers the major Web browsers on Windows and Ubuntu. The measurement results show that almost all the methods will inflate the network delay from a few milliseconds to tens of milliseconds. In particular, the delay overheads incurred in most of the HTTP-based methods are too large to ignore. For example, the median overheads of Flash GET and POST methods can be larger higher 100ms in some browsers. Moreover, the fluctuation of delay overheads for HTTP-based methods are also large. An extreme case is XHR GET. Its variation of overhead can be as large as 60ms. The socket-based methods, on the other hand, incur much smaller overhead, which values are usually smaller than 1ms. Navigation Timing method introduces relatively small and stable additional delays during the measurement, which median values and variations are all smaller than 2ms mostly. As for WebRTC tecchnique, its delay overheads are $\sim$4ms, but the variations can be $\sim$8-10ms.

   For smartphone-based tools, we have appraised their accuracy of delay measurement in WiFi networks. Different from fixed networks, obtaining actual network RTTs in wireless environment is much more challenging. We have designed and built a testbed that contains multiple sniffers. By performing trace merging and frame-based synchronization, we can obtain a complete packet trace for calculating the actual network RTTs. With the help of the reliable testbed, we have tested two of the most popular measurement apps, Ookla Speedtest and MobiPerf, and shown that they have their RTT measurements inflated, ranging from a few milliseconds to tens of milliseconds. We further build three test apps, including *Native ping*, *Inet ping*, and *HTTP ping* that cover three common measurement methods and evaluate them in the testbed. The measurement results show that the RTT measurement obtained from the three measurement methods are all inflated, and the amount of inflation is comparable to Ookla Speedtest and MobiPerf. Moreover, we also discover that the delay inflation introduced in the user space is asymmetric for packet sending and receiving. This observation facilitates the follow-up root-cause analysis.

2. **Root-cause analysis on the delay inflation.**

   We have analyzed the root causes that result in significant and unstable delay overheads for browser-based measurement. Our analysis shows 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 methods, such as Flash HTTP methods and WebRTC technique, over-estimate the RTT by including the TCP handshaking or data channel establishing phase.

   For smartphone-based measurements, we have performed multi-layer analysis, which decomposes the delay overhead into several finer-grained elements. Our analysis show that the overhead in Android runtime contributes to the majority of the delay overhead in the user space, and it is due to a long path of sub-function invocations. Moreover, the migration of runtime from DVM (Dalvik VM) to ART (Android Runtime) cannot help too much on alleviating the overhead. On the other hand, the sleeping function of the wireless network interface card (WNIC) driver is the major source of the delay overhead in the kernel.

Specifically, smartphones equipped with Broadcom WNIC chipsets implement an energy-saving mechanism in the Secure Digital Input Output (SDIO) bus. When there is no network activity, the SDIO bus will be turned into inactive mode (sleeping). It takes more than 10ms for the bus resuming active mode again if there are packets to send or receive. To our knowledge, we are the first to report the impact of SDIO bus sleeping on the network delay measurement.

3. **A novel tool for improving the accuracy of delay measurement.**

   Based on our delay overhead evaluation and root-cause analysis, we proposed to improve the accuracy of delay measurement. For browser-based measurements, although we have not implemented a applicable tool, we have given suggestions on measurement design as well as made recommendations for the best browser in practice. Specifically, the socket-based methods are recommended for their small and stable delay overheads. In the browser built-in HTTP-based methods, Navigation Timing is a good candidate of replacing XHR GET and POST methods, but the measurement immediately following TCP connection establishment should be excluded. As for the client-to-client measurement enabled by WebRTC technique, Firefox is the preferred browser.

   For smartphone-based measurement, we have proposed a novel tool, AcuteMon, to mitigate the delay inflation in smartphone-based measurements. AcuteMon mitigates the delay inflation in the user space by implementing the core measurement logic into a native C program and invoking it through an external system call in the app. For the delay inflation caused by the energy-saving mechanisms in the kernel and network layers, AcuteMon enforces the smartphones to operate in the active state by sending out a set of "warm-up" packets background packets. Our approach is not only for resolving the problems in WiFi networks, but also aims at removing the inflations caused by the RRC state transition in cellular networks. AcuteMon can run on unrooted phones and requires no system customization, such as kernel recompilation and customized ROM. Our testbed evaluation and Internet experiment show that AcuteMon can produce more accurate measurement results effectively.

1.5 Organization

The rest of this thesis consists of five chapters: Chapters 2 on the background knowledge and related works, Chapter 3 on appraising the accuracy of browser-based delay measurements, Chapter 4 on evaluating the accuracy of smartphone-based delay measurements, Chapter 5 on improving the accuracy in smartphone-based delay measurements, and finally Chapter 6 on conclusions and future work.

In Chapter 2, we will first present relevant background on network measurement and performance metrics. Next, we will revisit previous works on network delay measurement. After that, we will introduce how network performance benchmarking tools measure the network delays, and survey the existing browser and smartphone-based tools as well as the core methods they employ. Lastly, we will discuss the accuracy problem for network measurement, especially using delay overhead to quantify the inaccuracy.

Chapter 3 is devoted to appraising the accuracy of browser-based delay measurements. We will first present the experiment on the traditional HTTP-based and socket-based methods. After highlighting the results of granularity and execution cost measurements on the HR-Time, we will turn to the Navigation Timing technique. Finally, we will evaluate the performance of and WebRTC technique.

Chapter 4 is devoted to appraising the accuracy of smartphone-based delay measurements. We will first the introduce a multi-layer approach to analyze the performance problem in Android. Next, we will detail the testbed setup and our methodology if employing multiple sniffers to obtain a complete packet trace. After reporting the results obtained from the Internet experiments for Ookla Speedtest and MobiPerf, and the results from the controlled testbed experiments, we will then perform root-cause analysis on the delay overheads. Lastly, we will propose a measurement method to mitigate the delay inflation introduced in the user-space.

In Chapter 5, we will propose AcuteMon to mitigate the delay overheads caused by the energy-saving mechanisms. We will first conduct experiments and perform root-cause analysis on the delay overheads in mobile networks. And then we will describe the design and implementation of AcuteMon. Lastly, we will evaluate the performance of AcuteMon in both testbed and real Internet environment.

In Chapter 6, we will conclude our current research and look ahead several future directions. One issue that can be explored is extending the delay measurement to throughput measurement. Providing a unified solution for accurate browser-based delay measurement and performing large-scale Internet evaluation for AcuteMon could be the other two possible directions.

Chapter 2

Background and Related Works

In this chapter, we discuss existing works related to this thesis. We begin with some background knowledge on network performance measurement in terms of performance metrics. We summarize the existing measurement methods and tools, especially on inferring network delay. Next, we introduce the popular network performance benchmarking tools. Due to the ubiquity of Web browsers and smartphones, these measurements have gain popularity on both desktop PCs (browser-based measurement) and smartphones (smartphone-based measurement). We survey the existing benchmarking tools based on their measurement methods and implementations. Finally, we discuss the accuracy problem for network measurement. In particular, we show that the inaccuracy of network delay measurement can be quantified with delay overhead. Models of investigating the delay overhead for both browser-based and app-based tools are also provided.

2.1 Measuring network path performance

Network path performance measurement plays an important role for a number of purposes, such as modeling the evolution of the Internet [158, 169, 200], developing and validating new network analytic techniques [72], diagnosing network problems and making better management decisions [59,74,88,94,111,149,153,154,226], improving performance of the network-quality-sensitive applications [50, 123, 161, 195], verifying the compliance of service level agreement (SLA), and monitoring the the ISP's real performance [47,66,71,127,152,208,218,237]. To precisely quantify the network path performance and reliability, we need well-defined and concrete performance metrics as well as reliable and sound network measurement methods.

2.1.1 Network performance metrics

Both the Telecommunication Standardization Sector of ITU (ITU-T) [117] and IP Performance Metrics Working Group (IPPM WG) [14] of the Internet Engineering Task Force (IETF) [38] have spent great effort on standardizing performance metrics for quantifying various network path characteristics. In particular, the IETF IPPM WG has carefully defined a set of performance metrics, including connectivity [155], one-way and round-trip delay (a.k.a. round-trip time) [48,49], delay variation [75], one-way loss and loss patterns [48, 132], packet reordering [164], network capacity [70], and bulk transport capacity [157]. The equivalent recommendations introduced by ITU-T are Y.1540 [118] and Y.1541 [119]. The former defines a set of performance parameters that can be applied to end-to-end IP service monitoring, such as transfer delay, delay variation, error ratio, loss ratio, and reordered ratio, whereas the latter describes six different classes of service and specifies the performance bounds for network delays, losses and errors that define these classes.

Besides the performance metrics, the IPPM WG has also developed several important concepts of measurement period, samples, and packet types [177]. More specifically, the measurement period is denoted by *wire time*, which is the time that a packet is sent from the source of its first bit on the physical wire (out of the Network Interface

Card (NIC)) till the reception of the last bit on the destination. The terms *singleton*, *sample*, and *statistical* metrics are used to describe how the metrics are derived from the measurement instances. Moreover, a *Type-P* packet is a packet that complies with certain specified condition. The set of Type-P packets are chosen for computing the desired performance metrics.

2.1.2 Measurement methods and tools

Network measurements can be generally classified into three categories based on their methodologies: *passive measurement* [39, 46, 93, 188, 236], *active measurement* [64, 104, 146, 199], and *hybrid measurement* that combines both active and passive methods [55,217,218,221]. The major difference among them is that active methods are required to generate and inject measurement traffic into the monitored network, whereas passive methods are not required to do so. Passive methods monitor the network performance from one or more vantage points inside the network by inspecting the existing user traffic, and therefore will not disturb the normal network services. However, active methods usually provide more opportunities to monitor the e2e paths that cannot be seen by the passive methods, because the autonomous operation of the Internet makes it very difficult to perform passive measurement in domains outside the control of the measurement operator. Moreover, passive measurements need to identify some special packets, such as TCP SYN and SYN ACK packets for some particular source and destination pairs, from massive volume of user traffic and sort them without any background knowledge. This is a very challenging task to both algorithm (software) and processing capability (hardware). A more detailed discussion on the pros and cons of passive and active network measurements can be found from [181], and a detailed survey on network measurement methods and tools in [21]. In this study, we focus only on active network measurement.

Active measurement methods can be further classified into *duet-active* (a.k.a. *cooperative*) network measurements, which require the control of both end-points and can readily support different one-way metrics to quantify the forward-path or reverse-path performance [80, 129, 138, 187, 209], and *solo-active* (a.k.a. *non-cooperative*) , which measure the end-to-end paths without controlling the destination end-points [51, 64, 146, 196, 203]. Compared to the duet-active methods, solo-active methods have the advantage of facilitating large-scale network measurements given a sufficient number of remote end-points. By employing the existing network protocols or services, measurement tools based on solo-active methods can be easily deployed with a relatively low cost. For example, ping and traceroute, the most popular network measurement tools based on ICMP [182], are almost available on every desktop PC. Some measurement platforms, such as CAIDA's Archipelago (Ark) [63] and DIMES [200], utilize the tools built on ICMP to measure the RTT distance and discover network topology [60, 67].

Although many research works have derived from the ICMP-based tools, Li et al. have shown that the measurement results produced by these tools can differ from those obtained by TCP [227], because the ICMP packets and TCP packets could be treated with different QoS policies or processed on different routing paths. Moreover, the ICMP packets are often rate-limited or even filtered by firewalls [212], resulting in measurement failures. Considering that most of the Internet applications today are based on TCP [139], the network performance experienced by TCP-based applications may not be reflected accurately by ICMP-based measurement methods. To take a closer look to the user-perceived performance, a number of tools have been proposed based on sampling of network performance of with TCP packets. Sting [196] measures the one-way packet loss rate without the cooperation of the remote host. TCP Sidecar [202, 203] inserts measurement probes into the existing TCP streams to discover the router-level network topology. POINTER [147] concentrates on e2e packet reordering metrics. HTTP/OneProbe [146] reports the bi-directional packet loss and reordering rate besides the round-trip delay. TRIO [64] summarizes the asymmetric capacity of a network path.

However, these TCP-based solo-active methods still have limitations. First of all, these tools usually have a strict requirement on the running environment. For example, HTTP/OneProbe and TRIO can only work on Linux, because they rely on raw socket and need to manipulate the MSS (Maximum Segment Size) when establishing a new TCP connection, both of which cannot be achieved in Windows. Another case is that capacity measurement tools based on packet pair or packet train techniques. Due to the the proportional fair scheduling used by base stations, they cannot be applied to mobile networks and smartphones directly [159]. Second, the measurement results produced by the tools require networking expertise to interpret. These limitations prevent them from being deployed to the same scale as Ping and Traceroute. preventing their prevalence. Despite the fact that mobile networks, including WiFi and cellular networks, are getting more and more important as well as mobile devices become more and more popular, these tools cannot be applied to the mobile scenario easily.

Recently, more measurement tools are available to end users to benchmark their network performance and diagnose problems. Such efforts include various speedtest services on desktop machines and measurement apps on smartphones. In particular, many of these tools, such as Netalyzr [134, 222] and Ooklas speedtest [171], implement measurement methods in browsers which are ubiquitously available on all systems. These *browser-based* measurement tools usually measure the network RTT and throughput. Some can even measure the packet loss and reordering rates. Similarly, the *smartphone-based* measurement tools are available on the major mobile OSes, including Android [12, 17, 18, 34], iOS [32, 33], and Windows Phone [22, 35]. A more detailed study of these network performance benchmarking tools will be provided in

§2.2.

2.1.3 Measuring network path delay

This thesis focuses on network delay measurement, because it is the most fundamental atomic metric, from which rich information and many other metrics can be derived. For example, the minimum value of the delay is determined by the physical medium used for communication, and the maximum transmission rate of a network path is also limited by the network delay [68, 120]. Moreover, understanding and characterizing the network delay is also a prerequisite to diagnose network problems and improve the performance of network applications.

Several metrics related to one-way delay, round-trip delay, and delay variation have been introduced by IPPM WG. In particular, the one-way delay (OWD) is defined as the time passed from the first bit of the packet that has been sent to the network to the last bit that reaching its destination [48]. The round-trip delay (RTD or RTT) is considered to be the time interval between the time instant a source node emits a request packet to a destination and the time instant a response packet is received from the destination node [49]. Lastly, the IP Packet Delay Variation (IPDV) is the difference in the OWD of a selected pair of packets in a test stream going from the source to the destination [75].

Typically, a network path consists of multiple network devices (also known as *hops*), such as switches and routers. For a packet to be transmitted from hop i to hop $i+1$, the transmission delay can be divided into several components [61], as depicted in Figure 2.1.

- Transmission delay d_T $(= t_1 - t_0)$: The time needed to put all the bits of a packet onto a data link. Let L be the packet size and C_i the link capacity between hop $i+1$ and i. We will have $d_T = L/C_i$.

- Propagation delay d_P $(= t_2 - t_0)$: The time needed to propagate a bit though the data link. When the physical distance between hop $i + 1$ and i is S, d_P can be calculated as $d_P = S_i/v_i$, where v_i is the signal propagation speed.
- Processing delay d_Q $(= t_4 - t_3)$: It is also known as queuing delay, which is the time needed to wait in a router queue before transmission.

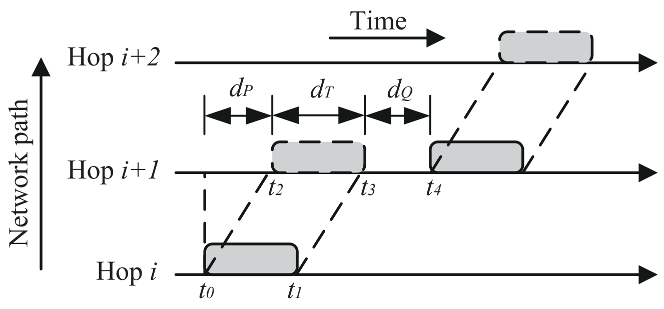


Figure 2.1: Components of network delay between two adjacent hops.

For a n-hop one-way network path, the OWD of the path can be calculated through

$$D_{OWD} = \sum_{i=1}^{n-1} \left(\frac{L}{C_i} + \frac{S_i}{v_i} + d_{Qi}\right). \tag{2.1}$$

Let D_{fw} and D_{rv} be the OWDs of the forward and reverse network path, respectively, the RTT of the round-trip path can be presented as

$$D_{RTT} = D_{fw} + D_{rv}. \tag{2.2}$$

Network delay can be monitored with both passive and active measurement methods. For passive method, the delay can be observed by capturing the in-flight data packets that flow through the vantage points (VPs). Typically, TCP packets are analyzed because they consists of sequence number and acknowledge number and therefore can be easily identified and correlated [62, 77, 124, 180]. The passive method can be deployed to monitor the backbone network [77, 89], residential network [156], and mobile network [69, 108, 179]. In this thesis, we focus on active delay measurements, which can employ ICMP, UDP, or TCP method to estimate the network path delay. Although the previous research has shown that the delay measured with ICMP can be different from those obtained with other protocols [227], active methods or tools based on ICMP are still in use in many research projects, such as CAIDA's Ark [63], DIMES [200], and DipZoom [226]. Other tools, such as pathchar [81] and pchar [150], utilize both UDP and ICMP to measure the per-hop network delay. There are a number of active measurement tools that monitor the network delay with TCP, including netperf [19], Iperf [15], HTTPing [105], and HTTP/OneProbe [146]. The speedtest-like measurement services prefer TCP (such as Ookla Speedtest [171]), or a combination of TCP, UDP, and ICMP (such as SamKnows [29]). These measurement tools are deployed in a number of scenarios, such as monitoring the performance of residential network [79, 217] and mobile network [216], as well as diagnosing network failures [88].

2.2 Performance benchmarking tools

Although a set of standard metrics has been defined and many measurement tools have been introduced in the past decades, applying them to the real-world scenario is still challenging. As summarized in Chapter 1, the reasons preventing the them from large-scale deployment can be various, such as non-representativeness, strict running constraints, and poor usability. In contrast, the network performance benchmarking tools, including browser and smartphone-based tools, have the advantages of ease of use, low cost of deployment, and representativeness, and therefore have gained popularity among end users. In the following, we will summarize and explain how the browser and smartphone-based tools work, including employing Web servers as measurement targets and the principle of measurement.

2.2.1 Employing Web servers as measurement targets

Employing Web servers as measurement targets is a cost-effective way to conduct large-scale network measurement. This method is adopted by not only browser and smartphone-based but many other measurement tools. According to Netcraft's survey published in Feb 2016, more than 900 million Web servers and Web-facing computers can be found in the world [165]. The Web servers hosted in both dedicated Internet service providers and individual homes provide a huge number of candidates for conducting large-scale network measurement. A Measurement tool can have its own dedicated Web servers [148, 171], or makes use of the existing ones [64, 88, 146].

The main concern of employing Web servers as measurement targets is the possible processing delay introduced by the servers. However, since browser and smartphone-based tools usually employ static Web objects with very small size for delay measurements, the processing delay is negligible. For example, Luo et al. show that more than 80% of the server-induced delays are observed smaller than 0.5ms [146]. If only TCP data packet is considered, the server-induced delays can be as small as tens of microseconds. In fact, in many previous research works, such as [179], the processing delay of the Web servers is always ignored.

2.2.2 Employing Web browsers as measurement sources in desktop environment

The prevalence of modern Web browsers for desktop PCs enables the large-scale deployment of browser-based network measurements. These measurements actively monitors e2e network paths from the Internet edge without requiring technical knowledge from users. Moreover, the system and configuration diversity of the involved end-hosts will not be a challenge, because most of existing browsers follow the standards developed by the World Wide Web Consortium (W3C) . A browser-based network measurement tool can generally provide many different services. Netalyzr [134], for example, provides network-layer information (e.g., RTT and path MTU), service reachability, and DNS measurement.

We have studied the RTT measurement methods employed by a number of browser-based tools, such as Netalyzr [134], Janc's methods [122], and How's My Network (HMN) [189], and the speedtest-like services, including Speedof.me [30] and Ookla [170, 171], 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 2.2. 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.

2.2.2.1 Traditional 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 2.1 summa-

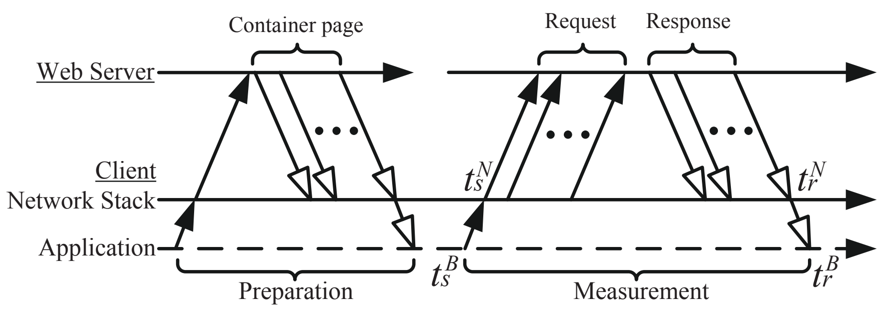


Figure 2.2: Two phases in browser-based network measurement.

rizes eleven such methods and 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 and WebRTC (Web Real-Time Communiction) technique (more details of WebRTC will be discussed in Section 2.2.2.3). Flash manipulates network socket

Table 2.1: 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	Tools / Services
HTTP-based	XHR	Native	GET	Yes	RTT, Tput	Speedof.me [30], BandwidthPlace [213]
			POST	Yes	RTT, Tput	Janc's methods [122]
	DOM	Native	GET	No	RTT, Tput	[122], [213], Wang's method [225]
	Flash	Plug-in	GET	Yes∗	RTT, Tput	Speedtest [171], AuditMyPC [52], Speedchecker [214], Bandwidth Meter [73], InternetFrog [115]
			POST	Yes∗	RTT, Tput	
	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 [134], HMN [189], JavaNws [135], Pingtest [170], NDT [148], AuditMyPC [53]
			UDP	No	RTT, Tput, Loss	
	Flash	Plug-in	TCP	Yes∗	RTT, Tput	[171]
	WebRTC	Native	UDP	No	RTT, Tput, Loss	

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

with class `Socket`. This class also provides function `addEventListener` to detect 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 returns successfully. 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.

2.2.2.2 Navigation Timing technique

Navigation Timing is an API recommended by the W3C's Web Performance Working Group [40]. It is not a Web technique of handling request sending and response receiving. Rather, it provides a simple way to get the detailed and fine-grained timing statistics for page navigation and load events. The events include DNS resolution, connection latency, and time to first byte from inside the browsers of real users. Figure 2.3 illustrates the processing model and the detailed events that can be identified through Navigation Timing.

Navigation Timing was accepted as a specification in Dec 2012. Currently 90.73% of browsers support this new feature [6]. Most of the modern browsers are compatible with it, except Safari in Windows because Apple stops updating it and its latest version is 5.1.7. Besides applying it for profiling the user experience on the Web [82, 113], some organizations also consider how to employ it for the purpose of network performance measurement. For example, cedexis launches its Radar project to benchmark the performance of cloud, content delivery, and private platforms [8]. And E. Gavaletz et al. discuss the feasibility of utilizing Navigation Timing to measure the HTTP response time [92].

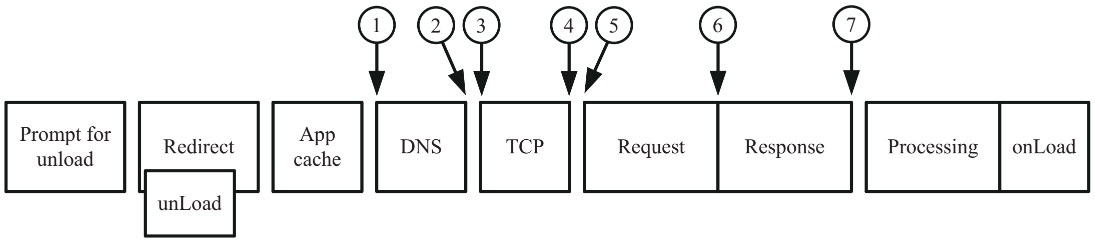


Figure 2.3: Detailed timing attributes defined by Navigation Timing technique.

Several events (events ①-⑦ in the Figure 2.3) can be utilized for network measurement:

`domainLookupStart` (event ①) returns the time immediately before the user agent starts the DNS lookup.

`domainLookupEnd` (event ②) returns the time immediately before the user agent finishes the DNS lookup.

`connectStart` (event ③) returns the time immediately before the user agent start establishing the connection to the server for retrieving the document.

`connectEnd` (event ④) returns the time immediately after the user agent finishes establishing the connection to the server for retrieving the current document.

`requestStart` (event ⑤) returns the time immediately before the user agent starts requesting the current document from the server, relevant application caches, or local resources.

`responseStart` (event ⑥) returns the time immediately after the user agent receives the first byte of the response from the server, relevant application caches, or local resources.

`responseEnd` (event ⑦) returns the time immediately after the user agent receives the last byte of the current document or immediately before the transport connection is closed, whichever comes first.

Figure 2.4 depicts the time that each event takes place. After the preparation phase, if the browser must open a new TCP connection to the remote Web server, it sends out a TCP SYN packet and waits for a TCP SYN ACK packet. Thus at times t_c^B and t_e^B, the events `connectStart` and `connectEnd` are recorded, respectively. Then an HTTP request message is sent and an HTTP response message is triggered. Here t_s^B corresponds to the event `requestStart`, while t_{r1}^B and t_{r2}^B refer to event `responseStart` and `responseEnd`, respectively. The RTT of the network path can be calculated based on the TCP connection establishment (*Measurement I* in the figure):

$$D_{RTT\_I} = t_e^B - t_c^B, \tag{2.3}$$

or the data exchange period (*Measurement II*):

$$D_{RTT\_II} = t_{r1}^B - t_s^B. \tag{2.4}$$

The Navigation Timing technique can also facilitate throughput estimation. The downstream throughput can be calculated based on the size of the response (L_{resp}) message:

$$B_{down} = \frac{L_{resp}}{t_{r2}^B - t_{r1}^B}. \tag{2.5}$$

As for the upstream direction, since the Navigation Timing has not provided the event

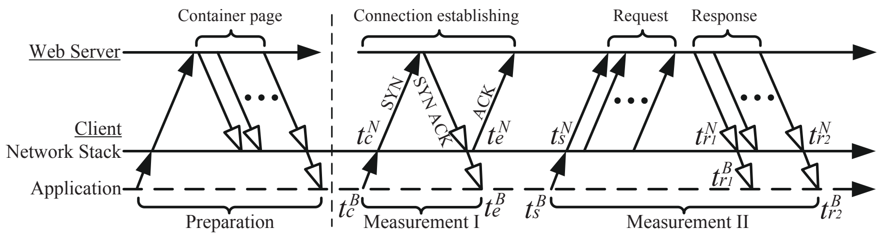


Figure 2.4: The measurement phases in measurements based on Navigation Timing technique. The RTT can be calculated based on TCP connection establishment events (Measurement I) and HTTP request/response messages (Measurement II).

that the last bit of the request message has been emitted, we need to subtract one RTT during the calculation:

$$B_{up} = \frac{L_{req}}{t_{r1}^B - t_s^B - RTT}. \tag{2.6}$$

where L_{req} is the size of request message.

Besides measuring the network path between the client and Web server through TCP and HTTP, events ① and ② can also be used to infer the DNS performance.

2.2.2.3 WebRTC technique

Similar to WebSocket, the Web Real-Time Communication (WebRTC) technique is also an in-browser JavaScript API definition drafted by the W3C. However, it differs from WebSocket in that it enables direct browser-to-browser communication whereas WebSocket is still based on the client-server model. Therefore, WebRTC is usually applied to the scenarios where point-to-point voice calling, video calling, and file sharing are required. This important characteristic also makes it possible to monitor the network performance between two clients. In the context of network measurement, the WebRTC *RTCDataChannel* API allows browsers to exchange arbitrary data with each other. By recording the packet sending time and receiving time, the RTT can be then calculated. Moreover, since WebRTC provides packet-level granularity, the packet loss rate can be estimated likewise.

The underlying protocols of WebRTC are defined by the IETF Rtcweb Working Group [28]. The exchanged data is under the control of SCTP (Stream Control Transmission Protocol) [185], which allows out-of-order delivery and retransmission of messages. To ensure the security, the WebRTC data is tunneled over an encrypted DTLS [186] tunnel. Figure 2.5 shows the underlying protocol stacks of WebRTC DataChannel, as well as the comparison to XHR and WebSocket. Note that the protocols on top of UDP are implemented by browsers. Applications on top of WebRTC do not need to maintain the retransmission timers or counters by themselves.

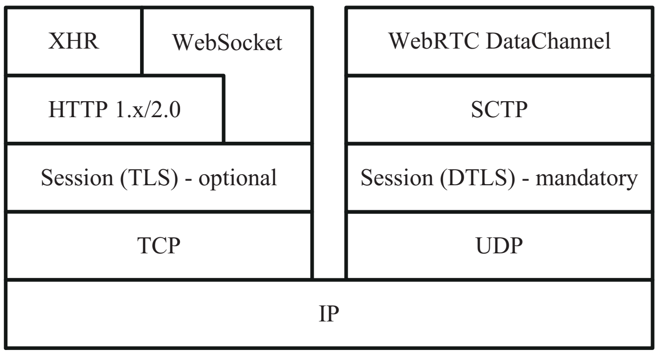


Figure 2.5: The underlying protocol stacks of WebRTC.

WebRTC is supported by major browsers. According to [7], 61.83% of the existing browsers are compatible with it. Although it has not been employed in the existing browser-based network measurement tools, several works have been published to study its performance. For example, the congestion avoidance for WebRTC video sessions in LTE networks has been studied in [131]. In [205], the performance of the congestion control algorithms currently implemented by the browsers has been evaluated. Yan et al. focus on the problem of accurate prediction of QoE of WebRTC in WiFi network and collect RTT and packet loss rate through WebRTC script. Fund et al. identify the characteristics of the cellular network that impact video QoS directly based on the measurement data collected from DASH and WebRTC implementations [91].

2.2.3 Employing measurement apps as measurement sources in mobile environment

Monitoring mobile network performance through smartphone apps has gained popularity among end users in recent years. These speedtest-like services are available on Android [12,13,34], iOS [32,33], and Windows Phone [22,35]. Besides, research communities have deployed measurement apps to crowdsource data from a large group of users. To name a few examples, Netalyzr [18] diagnoses connectivity characteristics, performance anomalies, and security issues [222, 223]. MobiPerf [17] is used to infer the radio resource control (RRC) state information of cellular networks [190, 191]. Mobilyzer provides a flexible open-source library for app developers to conduct mobile network measurement experiments in a principled manner [167].

In this thesis, we consider Android platform for its prevalence and open-source system. Although most of the apps are built on Web techniques, such as employing Web servers as measurement targets or HTTP request/response as the probe payload, they can rely on other protocols (e.g., ICMP and UDP). We will cover all possible methods in our research.

Android provides several interfaces or APIs for sending packets and recording timestamps, which can be utilized for implementing a network measurement app without rooting the devices. We have studied RTT measurement methods employed by a number of Android apps by inspecting their codes and the packets exchanged between the Android phone and servers. Table 2.2 documents the implementation details for some popular measurement apps in Android. We have summarized the supported probe packet types, core methods to send and receive packets, functions to record timestamps, number of sampling probes, and the reported results (min/mean/max). Although most apps prefer to use the class provided by Java or Android, such as `java.net.URLConnection` for handling HTTP request and response messages, the direct execution of external binary (e.g., the built-in `ping` program located at `/system/bin` by default) is also allowed to perform network measurements. Moreover, each measurement app supports one or more probe packet types, including ICMP, UDP, and TCP (both control messages and data packets). When timestamping the packet sending and receiving events, the apps also employ different timing functions, whose resolutions vary from millisecond to nanosecond. Even for the result reporting, different apps may have their own choice. For example, Ookla Speedtest sends 6 HTTP GET request packets to a Web server and reports the minimal RTT estimation in integer as the final result.

2.3 Measurement accuracy

The accuracy of a network measurement system can be defined as how the measurement results deviate from the real network performance. Following ISO 5725 [42], 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). In reality, accuracy is expressed through "inaccuracy," which reflects the unavoidable imperfection of a measurement. Here, the inaccuracy is the deviation of the measurement results from the "true value" of the measurand. Such deviation is called the *measurement error*. Since the true value is not known, the true value is often treated as an accepted reference value for the property being measured in the calibration. In this context, the inaccuracy of the "true value" must be negligible compared with the inaccuracy of the measurement instrument being calibrated.

The measurement error can be evaluated in two ways: absolute measurement error ζ and the relative measurement error ε:

$$\zeta = \hat{A} - A \tag{2.7}$$

Table 2.2: Implementation details of existing network measurement apps in Android.

App	Supported probe type	Core method	Timing function	# of Samples	Results
Ookla Speedtest [34]	HTTP GET	`java.net.URLConnection`	`SystemClock.uptimeMillis()`	6	Min
MobiPerf [17]	ICMP	Executing `ping` program	N/A	10	Min/Mean/Max
	TCP SYN/RST	`java.net.InetAddress.isReachable()`	`System.currentTimeMillis()`	10	Min/Mean/Max
	TCP SYN/ACK†	`java.net.HttpURLConnection`	`System.currentTimeMillis()`	10	Min/Mean/Max
Netalyzr [18]	UDP	`java.net.DatagramPacket`	`Date().getTime()`	200	Mean
Speedchecker [12]	TCP SYN/ACK†	`java.net.HttpURLConnection`	`Date().getTime()`	1	N/A
V-SPEED Internet Speed Test [13]	TCP SYN/ACK†	`java.net.HttpURLConnection`	`System.currentTimeMillis()`	50/30/20*	Mean
FCC Speed Test [11]	UDP	`java.net.DatagramPacket`	`System.nanoTime()`	60	Min/Mean/Max

Note *: It depends on the network status.
†: Although the HTTP-related class is used, the app only establishes a TCP connection without sending out any HTTP request messages and then closes the connection.

$$\varepsilon = (\hat{A} - A)/A. \tag{2.8}$$

Where $\hat{A}$ is the measured value and A is the true value of the measurand.

A number of factors contribute to measurement inaccuracy, including the effects of sampling, the performance bias of different methodologies, the performance bias of different environments, and the impact of self-induced interference, and so on. Generally, absolute measurement error is not a suitable quantitative characteristics of measurement accuracy, because absolute error does not always give an indication of how important the error may be. Measurement accuracy can be further characterized quantitatively by the inverse of the relative error. However, in our follow-up analysis on the accuracy of network delay measurement for both browser and smartphone-based methodologies, we find that absolute measurement error can better quantify how the measurement results deviate from the true values, because the error is independent of the actual network delay. Namely, the measurement errors for different RTTs are in the same degrees. Therefore, the relative error will be very small when the network path is long.

In the past decades, a number of research works have studied the accuracy or reliability of the measurement methodologies. In [176], how to deal with measurement errors and imperfections have been discussed. In particular, the author use the term “precision” to denote the maximum exactness that a tool can permit, and the term “accuracy” to denote how well a measurement matches the actual phenomenon. Based on the discussion and experiences, Paxon presents a number of strategies to avoid or overcome the measurement pitfalls. In [192], Roughan uses Heisenberg inequality to describe the bounds on the accuracy of network measurement. If one increases the sending rate of probe packets in order to improve the measurement accuracy, the extra packets will hurt performance for all packets. Namely, all measurements of a system's performance are correlated, and these correlations reduce the efficacy of measurements. Roughan compares two different sampling methods quantitatively and shows that irregular probing patterns are useful to prevent the periodicity in the system under observation [193]. In [54], , Baccelli et al. study the effect of sampling methods and show that PASTA (Poisson Arrivals See Time Averages) is of very limited use in active probing. Other works improve the accuracy of measuring some specific network performance metrics. For example, Sommers et al. propose BADABING to improve the accuracy for packet loss measurement [207]. In [211], Sommers et al. advocate a vitro-like methodology to calibrate the available bandwidth estimation tools.

2.3.1 Quantifying the inaccuracy of network delay measurement with delay overhead

Although there are a number of existing works that focus on the problem of network measurement accuracy, few studies consider whether browser-based and app-based measurements can produce accurate results. In this thesis, we consider the accuracy of the network RTT measurement, because both browser and smartphone-based tools, being operated on the application layer, may significantly inflate the actual network RTT. The inflation also affects jitter and throughput (Tput) measurement.

To quantify the inaccuracy of network delay measurement, we use the term *delay overhead* to refer to the difference between the value measured by the measurement tools and the actual network delay. A smaller and stable delay overhead suggests that the measurement is more accurate and the measurement tool is more reliable. Let d_u be the RTT result reported by the tool, and d_n the actual network delay. The delay overhead Δd can be defined as

$$\Delta d = d_u - d_n. \tag{2.9}$$

2.3.2 Measuring the delay overheads for browser-based tools

Back to Figure 2.2, supposing that the request and response messages are sent in one packet each, the network RTT is given by the difference of the packet's receive and send timestamps which are measured by WinDump in Windows and tcpdump in Linux: $t_r^N - t_s^N$. Here we use the timestamps fetched in the kernel to approximate the exact time that the packets are on the wire, because the delay between the kernel and the NIC is considered negligible. 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 timing function (such as `Date.getTime()` in Java applet). The accuracy of the browser-level RTT measurement thus 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

$$\Delta d = (t_r^B - t_s^B) - (t_r^N - t_s^N). \tag{2.10}$$

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.

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. [122] 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. [128] 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. In [92], Gavaletz et al. simply compared the HTTP response times for DOM, XHR and Navigation Timing API in different browsers. However, their work only focuses on the performance difference for those methods, and has not touched the accuracy problem compared to the actual network delay.

Krintz and Wolski [135] compared the performance between Java applet and C program with JavaNws, and found that Java applet is comparable with C socket. Yeboah et al. [232] performed an Internet measurement study to compare the delay measurement results from ICMP ping, King [100], 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.

Our research differs from these existing works in that we systematically investigate whether the browser-based measurements can yield accurate result compared to the actual network delay. Moreover, the evaluation covers almost all the techniques that have already been or could be employed by browser-based measurements in major browsers on both Windows and Linux. We believe our study can be a comprehensive guideline for designing and conducting more reliable network measurements.

2.3.3 Measuring the delay overheads for app-based tools

Carrying out network measurement in mobile devices is much more challenging than the desktop environment, even though the core methodology is similar in both cases. A major difference is the operating system architecture. Android measurement apps usually run in a virtual machine. The apps could therefore encounter a larger system overhead in sending probe packets and receiving response packets, thus inflating the actual network delay.

Considering a simple probe-response scenario in Fig. 2.6, a measurement app sends out a probe packet at time t_u^o to a web server (or other types of target). The probe packet elicits a response packet from the server, which arrives at the measurement app at time t_u^i. The measurement app thus records d_u (= $t_u^i - t_u^o$) as the network RTT. Obviously, this measured RTT is generally larger than the actual RTT d_n (= $t_n^i - t_n^o$), where t_n^o (t_n^i) is the time for the probe (response) packet to leave (arrive at) the smartphone. The delay overhead is therefore defined as

$$\Delta d = d_u - d_n = (t_u^i - t_u^o) - (t_n^i - t_n^o). \tag{2.11}$$

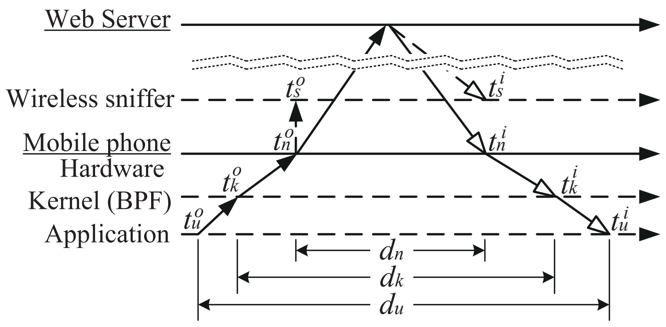


Figure 2.6: Measurement flow for Android apps.

There are three possible factors contributing to the delay overhead: (i) the timestamping accuracy of the outgoing and receiving packets, (ii) the delay for Android to propagate the probes to the kernel and network stack, and the delay for delivering the responses to the app, and (iii) the delay for the hardware (wireless network adaptor) to send and receive packets.

For factor (i), Android provides several timing functions, such as `System.nanoTime()` and `System.currentTimeMillis()`. Although these two functions have different resolutions (ns vs. ms) and map to different POSIX functions `clock_gettime()` and `gettimeofday()`, they share the same back-end function `clock_gettime()` through vsyscall according to POSIX.1-2008 [219]. Giucastro tested the granularity and performance of the two functions on some Android phones, and found that the average cost for executing the timing function is about 1μs [95]. Considering that the network delay is usually at ms level, the overhead of calling the timing functions is negligible.

We will therefore focus on the other two factors. To further quantify them, we also include two other timestamps t_k^i and t_k^o which are obtained when the packets are at the kernel. While we could obtain the kernel timestamps by using `tcpdump`, it is much more challenging to obtain the two network timestamps t_n^o and t_n^i. In wired network, these two timestamps can be easily obtained by placing an external packet sniffer to capture the packets copied from a network tap. This is because the fixed network is more reliable (i.e., the packets are seldom dropped by the sniffer), and the measurand and the sniffer can be easily time-synchronized.

Moreover, in wireless network a single wireless sniffer is not reliable enough to capture all the packets in the air (see Section 4.1.3). Using multiple sniffers, however, requires a careful trace merging and timestamp recovery. Moreover, as Android phones do not support PTP, synchronizing the clocks between the external sniffer and the phone is difficult. Another concern is due to the mechanism of FullMAC MLME (MAC Sublayer Management Entity) in which all 802.11 wireless frames are first translated into IEEE 802.3 Ethernet frames before being delivered to the kernel. Such packet translation could further increase the delay overhead. We will explain how we tackle these issues in Chapter 4.

The measurement studies based on smartphones users include [86, 108, 110, 201]. In particular, a simple logger was employed in [86] to collect the network usage information from Android and Windows Mobile users, whereas LiveLab [201] measured wireless networks in iOS. In [108] and [110], the performance of 4G LTE and 3G networks was evaluated using 4GTest and 3GTest, respectively. MobiPerf, the successor of 4GTest and 3GTest, has been employed to uncover the RRC state dynamics in cellular networks [190, 191] and study the network performance from end users' perspectives [109, 166]. Netalyzr, another measurement app in Android, characterizes middlebox behavior and business relationships in cellular networks [223]. These existing apps are designed with more concern on privacy issues and energy consumption, but their accuracy has not received any attention.

In the system performance area, several studies evaluated the performance of JNI or DVM. For example, Oh et al. investigated the performance impact of DVM on Android apps [168]. Batyuk et al. compared the performance between native C and Java applications for identical tasks [56], and showed that native C applications can be up to 30 times faster than running Java in DVM. But their work drew conclusions from Android emulator and Linux x86 platform. Lee and Jeon also carried out similar study for five algorithms [141] and found that JNI communication delay was about 0.15ms. These works focus mainly on the performance comparison of specific algorithms but do not study the relationship between system delay and network delay measurement. In [231], Xue et al. proposed a profiling system called AndroidPerf, which supports cross-layer function call trace and performance analysis from the DVM layer to the kernel layer. In [137], a tool based on `kprobes` was utilized to intercept system events in Android. However, these two works do not analyze the network behavior systematically, nor the device driver as we do.

Chapter 3

Towards accurate browser-based network measurement

In recent years, browser-based network measurement tools, such as Ookla's speedtest [23] and Netalyzr [134], have gained popularity among end users by enabling them to monitor their providers' performance and diagnose network problems, especially in desktop environment. However, very little is known about the (in)accuracy of various methods used in these tools. In this chapter, we focus on the accuracy that these methods can achieve in network delay measurements. 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. Our study covers the traditional HTTP-based and socket-based techniques that have already been employed by the existing measurement tools, such as JavaScript, Flash, and Java applet, as well as the Navigation Timing and WebRTC techniques. For each method, we implement a test page and experiment with the major browsers on Windows and Ubuntu.

For the traditional browser techniques, 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). Our further analysis show that some HTTP-based methods over-estimate the RTT, because they include the TCP handshaking in the delay. 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 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. After replacing the timing function with `System.nanoTime()`, the experiment results show that the under-estimation of RTT disappears.

For the Navigation Timing technique, our investigation shows that the event `connectStart` and `connectEnd` can be fully supported by Chrome and Opera only, indicating that estimating the delay of TCP connection establishment is not applicable. Moreover, openning a new TCP connection could also affect the subsequent event (requestStart) and (responseStart) for some browsers. Only when the HTTP request/response messages are exchanged in an existing connection, these two data events can be utilized for more reliable delay measurement. In this case, the delay overheads can be restricted within 2ms for all browsers and systems under test. Compared to WebSocket, although Navigation Timing technique introduces slightly larger delay overheads, it is still a good choice to replace the XHR GET and POST methods, because it does not require any third-party plug-ins and additional socket servers. The WebRTC technique, on the other hand, makes it possible to perform client-to-client type of measurements within browsers. Our testbed evaluation demonstrates that timestamping the request and response messages following the data channel establishment could include the channel setup delay, and therefore should be ignored in practice. Without such additional delay, the median delay overhead can be ∼4ms for Chrome and Opera and ∼2ms for Firefox.

Besides appraising the accuracy of browser techniques, we also study the performance of the new in-browser timing function, High Performance Time (HR-Time), because accurately capturing the packet sending or receiving time is a prerequisite to perform reliable delay measurements. Our evaluation shows that HR-Time achieves microsecond-level granularity in most browsers. Even for Chrome and Opera in windows, their granularity (1ms) can be the same to the traditional timing API `Data.getTime()`. Since the execution cost of HR-Time is negligible, we recommend using HR-Time for browser-based network measurements.

The outline of this chapter is as follows. We first conduct experiments to evaluate the accuracy of traditional HTTP-based and socket-based methods in Section 3.1. After highlighting the results of granularity and execution cost measurements on the HR-Time in Section 3.2.2, we turn to the Navigation Timing technique and WebRTC technique in Section 3.3 and 3.4, respectively. Finally, we summarize the chapter in Section 3.5.

3.1 Traditional HTTP-based and socket-based browser techniques

In this section, we consider the traditional HTTP-based and socket-based browser techniques. The former includes XHR GET/POST, DOM, Flash GET/POST, and Java applet GET/POST, while the latter covers WebSocket, Flash socket, and Java applet socket. Most of these techniques have already been implemented as the measurement cores for some existing tools, as listed in Table 2.1. Although WebRTC is also based on UDP socket, we evaluate its performance separately in Section 3.4, because it enables a new type of network measurement.

3.1.1 Experiment setup

In measuring the delay overhead incurred on the RTT measurement, we consider the ten aforementioned HTTP/TCP measurement methods. Besides the 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 3.1 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, all we have tested are 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 3.1. Both machines have the same hardware configura-

Table 3.1: Configurations of the browsers and systems used in the experiments.

OS	Browsers	Version	Flash	Java applet	WebSocket
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	√

tion: equipped with a 1.86GHz 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.

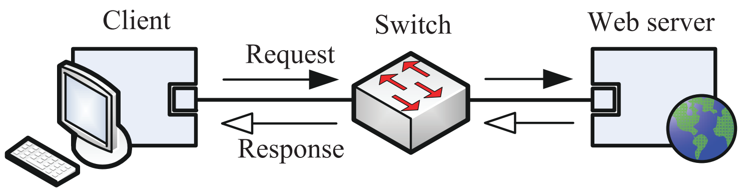


Figure 3.1: Testbed Setup.

We have prepared a container page using PHP or HTML for each measurement method imbedded with JavaScript code, Flash object, or Java applet. 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 2.2.2 and Figure 2.2, 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.

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 reuse 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.

3.1.2 Measurement Results

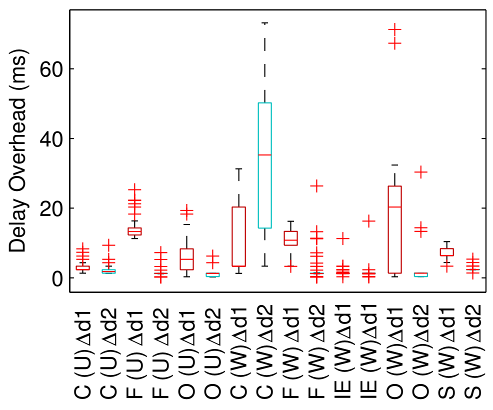
We plot the ten sets of measurement results (one per measurement method) in Figure 3.2 and 3.3 by using box-and-whisker plots. The first row includes the four methods using native features in browsers. The second comprises the Flash methods, and the third the Java applet methods. Each plot (except for 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.

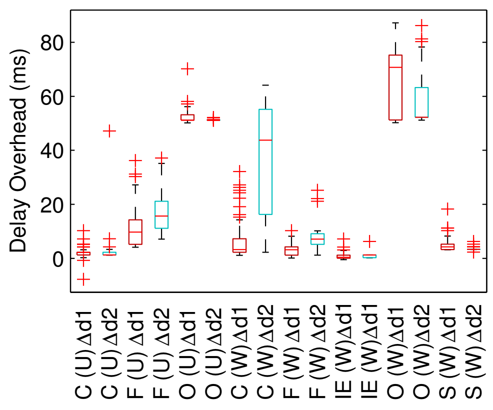
Figures 4.5(a), 4.5(b), 3.2(c), 3.3(a), 3.3(b), 3.3(d), and 3.3(e) for the HTTP-based methods show that the delay overhead generally cannot be ignored. The XHR methods' delay overheads range from a few milliseconds to tens of milliseconds. The overheads in Flash are extremely high. The median overheads 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.

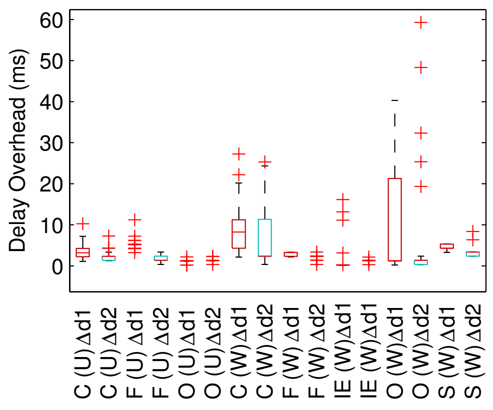
On the other hand, Figures 3.2(d), 3.3(c), and 3.3(f) show that the delay overheads incurred by the socket-based methods are considerably small. The median overheads are mostly smaller than 1 ms. Nevertheless, the overheads for some browsers fluctuate within a range of around 10 ms (e.g., Java applet 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.



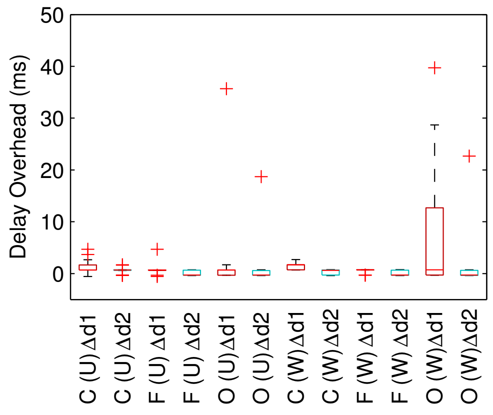
(a) XHR GET.



(b) XHR POST.

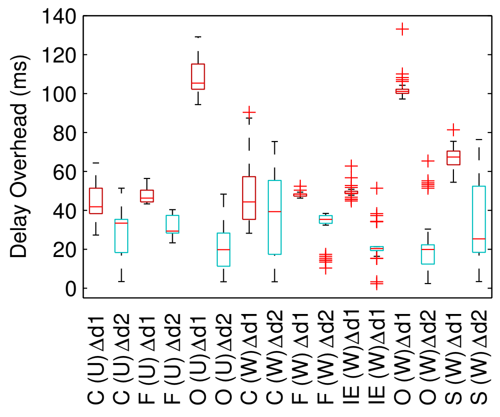


(c) DOM.

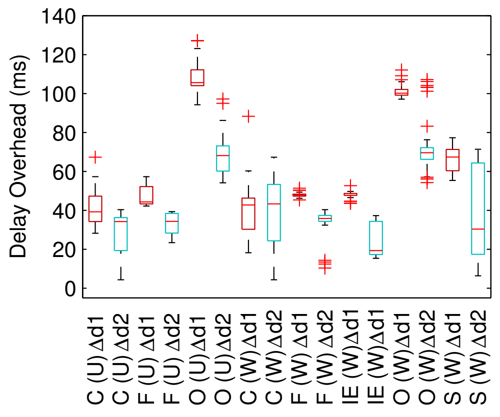


(d) WebSocket.

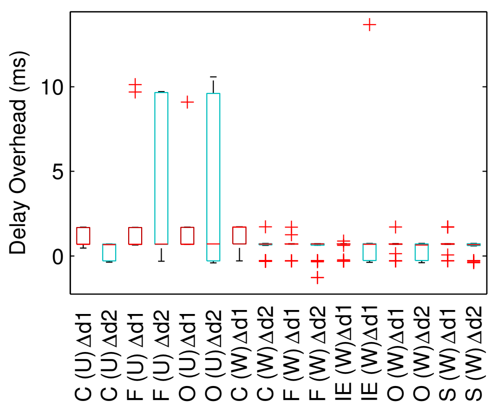
Figure 3.2: Box plots of the delay overheads (by methods).



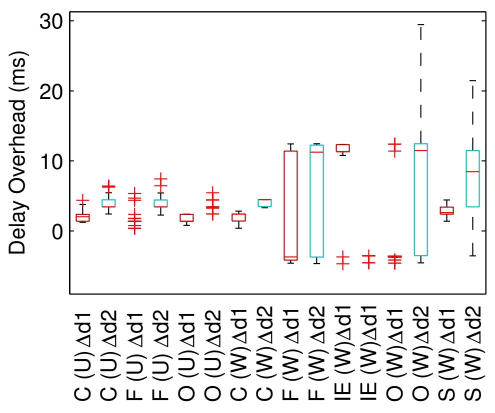
(a) Flash GET.



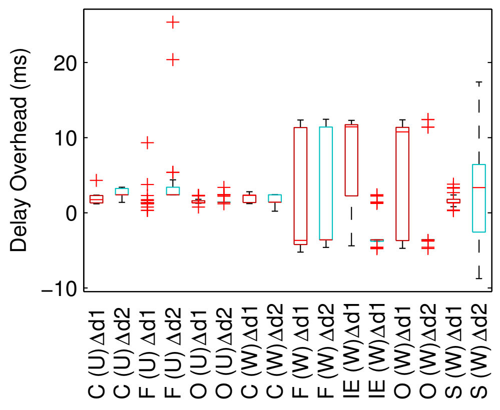
(b) Flash POST.



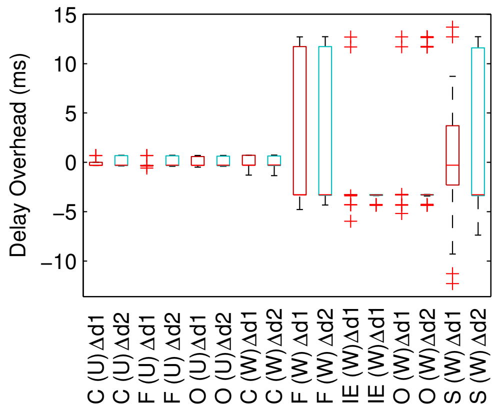
(c) Flash TCP socket.



(d) Java applet GET.



(e) Java applet POST.



(f) Java applet TCP socket.

Figure 3.3: Box plots of the delay overheads (by methods).

3.1.2.1 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 3.2 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 3.2: 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

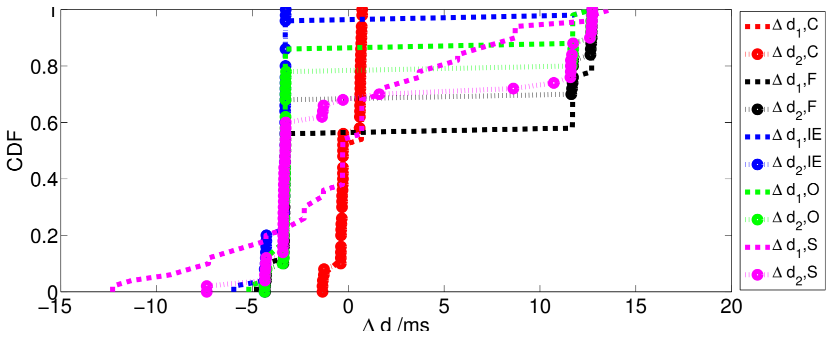
The packet capture files show that Opera opens a new TCP connection to handle the HTTP request issued by the Flash object for the first RTT measurement, therefore inflating the Δd_1 measurement. In the GET method, this existing connection can be reused for the second measurement. Therefore, the Δd_1 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.

3.1.2.2 The effect of timestamp granularity

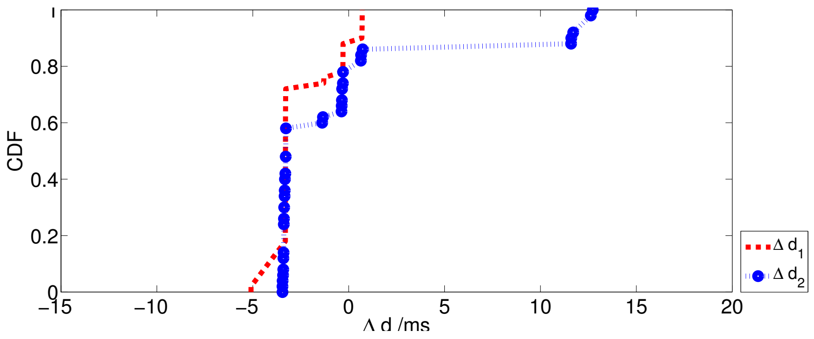
From Figure 3.3(d), 3.3(e), and 3.3(f), all three Java applet methods suffer from the negative delay overheads on Windows, which indicates that performing path measurement with Java applet can severely under-estimate the RTT. At the same time, significant variance can be observed. For example, Safari's overhead in Java applet socket method spans in the range of -13 ms and 13 ms, as illustrated in Figure 3.3(f). 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 3.4(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 [173], 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` provided by Oracle Java Development Kit (JDK). We plot the CDFs of Δd_1 and Δd_2 in Figure 3.4(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 as the causes 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 [174]. We test the timestamp granularity with the code shown in Figure 3.5. 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 [172], 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 3.4: CDF plots of Δd_1 and Δd_2 using the Java applet socket in Windows.

Table 3.3: 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 ±0.02	4.80 ±0.09	2.71 ±0.03	1.84 ±0.00	0.01 ±0.00	0.07 ±0.01
Firefox	2.73 ±0.02	4.38 ±0.08	2.41 ±0.03	1.49 ±0.01	0.00 ±0.00	0.07 ±0.01
IE	2.73 ±0.03	4.56 ±0.09	2.57 ±0.09	1.49 ±0.04	0.02 ±0.01	0.06 ±0.01
Opera	2.83 ±0.03	4.46 ±0.07	2.51 ±0.03	1.57 ±0.01	0.01 ±0.00	0.06 ±0.01
Safari	1.88 ±0.05	1.52 ±0.02	1.62 ±0.07	1.42 ±0.01	0.07 ±0.00	0.13 ±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 disappear 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.

```
long start = 0;
long end = 0;
while (true) {
    if (start == 0) {
        start = new Date().getTime();
    } else {
        long current = new Date().getTime();
        if (current != start) {
            end = current;
            break;
        }
    }
}
System.out.println((end - start) + "ms");
```

Figure 3.5: Codes for testing the timestamp granularity.

3.1.3 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 [53, 134, 148]. 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 3.1.2.1, always open new connections for measurement whether the measurement object can be reused or not. In this case, the additional delay cannot be avoided.

3.2 High Performance Time

In order to measure the network delay, a critical step is to accurately capture the times that a packet is sent or received. The existing browser-based network measurements typically employ the in-browser timing function `Date.getTime()` defined by the ECMAScript language specification [114], which provides timestamps with millisecond-level resolution. Since `Date.getTime()` is neither monotonical no precise enough, a new timing API, High Performance Time (HR-Time) is proposed by the W3C Web Performance Working Group [224]. As a monotonic clock, HR-Time reports the current time in sub-millisecond resolution through function `performance.now()`. Currently, HR-Time is supported by major browsers. According to [5], 89.91% of the browsers have the feature enabled by default.

However, although HR-Time is claimed to be able to provide timestamps in nanosecond, few studies have evaluated its actual performance systematically. In this section, we measure the actual performance of HR-Time in terms of granularity and execution cost. We run experiments on five browsers in Windows 10 and three browsers in Ubuntu 14. The detailed configurations of the browsers are listed in Table 3.4. Here we do not consider Safari because Apple has stopped releasing new version of the browser in Windows. The up-to-date version of Safari (5.1.7) does not support HR-Time.

3.2.1 Granularity

The granularity test is similar to the test described in § 3.1.2.2. As the time granularity is the minimal distinguishable grains that a clock system can report, our experiment code keeps querying the timestamp with `performance.now()` until the current value is different from the previous one. The difference in the two timestamps is the granularity that this function can achieve. However, due to the multi-task paradigm of the OS, the timestamp query can be postponed by other tasks. We therefore repeat the experiment for 20 times and record the smallest value for each browser. Moreover, even in our simple execution loops, the comparison between the two consecutive timestamps still needs time. As a result, the actual granularity could be smaller than the values we report for the case that the two consecutive queries return different values.

As shown in Table 3.4, the granularity of HR-Time for most of the browsers are in microsecond-level, which is small enough for measuring network delay. For example, the identifiable increments for Edage and Firefox are smaller than 5μs. However, we still find that the granularity for Chrome and Opera in Windows can be the same as

Table 3.4: Configurations and measured granularity of the browsers used in the experiments.

OS	Browsers	Version	Granularity (in ms)
Windows	Chrome	50.0	1
	Firefox	48.0	<0.005
	Opera	39.0	1
	Edge	25.10586	<0.0047
	IE	11.0	<0.0062
Ubuntu	Chrome	51.0	<0.025
	Firefox	46.0.1	<0.005
	Opera	37.0	<0.025

the traditional timing function `Date.getTime()` (1ms). Although HR-Time cannot achieve nanosecond-level granularity, it outputs timestamps with better precision in most cases.

3.2.2 Execution cost

We then compare the execution costs for running the two different timing functions. In the experiment, the timing function is executed for 1,000,000 times continuously. By recording the start time and end time, the total time cost can be estimated. We repeat the test for each browser for 50 times and calculate the mean and 95% confidence interval, as shown in Table 3.5. Overall, the total execution times are all several hundred milliseconds, except that HR-Time in Firefox performs best with extremely small values (2-3ms only) for both Windows and Ubuntu. Although `Date.getTime()` usually has a smaller execution time, but the difference between the two timing functions is not so significant. Especially when one single function call is considered, the execution times of both are trivial.

In summary, our experiment reveals that HR-Time usually provides timestamps with better precision. Even for the worst case (Chrome and Opera in Windows), HR-Time can achieve a comparable granularity to the traditional `Date.getTime()`. Al-

Table 3.5: Time cost of executing 1,000,000 timing functions for each browser (mean with 95% confidence interval, in ms).

OS	Browsers	`performance.now()`	`Date.getTime()`
Windows	Chrome	238.54 ±3.61	296.54 ±0.63
	Firefox	2.33 ±0.12	365.98 ±3.49
	Opera	376.74 ±4.10	365.66 ±1.04
	Edge	391.21 ±0.87	249.90 ±2.46
	IE	1259.41 ±6.70	416.792 ±0.88
Ubuntu	Chrome	332.30 ±2.64	173.47 ±0.26
	Firefox	3.24 ±0.11	347.11 ±0.16
	Opera	325.63 ±3.07	176.30 ±0.08

though its execution cost can be a bit higher than `Date.getTime()`, it can be ignored because only two function calls are needed in each RTT measurement. Therefore, HR-Time is recommended for browser-based measurement, especially for estimating the throughput.

3.3 Navigation Timing technique

As discussed in Section 2.2.2.2, the Navigation Timing technique enables us to monitor the events how the Web browsers load a page. By acquiring the timestamps of events `connectStart`, `connectEnd`, `requestStart`, and `responseStart`, we can obtain two types of network RTTs, the RTT (d_{RTT_C}) estimated by the TCP SYN/SYN ACK control messages and the RTT (d_{RTT_D}) by the TCP data, respectively.

We first validate whether the aforementioned events can be fully supported by the major browsers. Here we define "fully supported" as those events can return reasonable timestamps. Namely, if a browser reports same values for event `requestStart` and `responseStart`, for example, we do not consider it as being able to support the two events, because it cannot capture the events correctly. We build a page displaying the timestamp of each event through class `performance.timing` when the page is fully loaded. The page is hosted on a local Web server. The RTT between the client and the server is set to 50ms. We then examine the timestamp value of each event reported by the browser. If the RTTs (both d_{RTT_C} and d_{RTT_D}) are not within a reasonable range (e.g., the estimated RTT is only several milliseconds), we mark it as unsupported.

Table 3.6: Summary of whether the Navigation Timing events are supported by the browsers.

OS	Browsers	Version	`connectStart` and `connectEnd`	`requestStart` and `responseStart`
Windows	Chrome	50.0	$\surd$	$\surd$
	Firefox	48.0	$\times$	$\surd$
	Opera	39.0	$\surd$	$\surd$
	Edge	25.10586	$\times$	$\surd$
	IE	11.0	$\times$	$\times$
Ubuntu	Chrome	51.0	$\surd$	$\surd$
	Firefox	46.0.1	$\times$	$\surd$
	Opera	37.0	$\surd$	$\surd$

Table 3.6 summarizes the browser configurations and whether the connection events (`connectStart` and `connectEnd`) and data events (`requestStart` and `responseStart`) are supported. Although Navigation Timing technique has been proposed as a recommendation since 2012, it is still not well supported by the major browsers. For example, only Chrome and Opera can support the connection events. IE does return timestamps for the events, but the values of the connection events are the same and so do the data events.

3.3.1 Experiment setup

We then conduct experiments to evaluate the accuracy of the Navigation Timing method. For the browsers that support Navigation Timing feature, a class named `window.performance` can be accessed. In our implementation, the container page and the measurement target are hosted in the same Web server. When the container page has been downloaded and rendered, the `window.onload` event is triggered. By reading the attribute `connectStart` and `connectEnd` of the class `window.performance.timing`, we can calculate the connection RTT d_{RTT_C} and its corresponding delay overhead Δd_1. Similarly, the delay overhead Δd_2, which corresponds to the data RTT d_{RTT_D}, can be obtained through event `requestStart` and `responseStart`. In order to prevent the browsers from using the cached content, we add the current timestamp as a parameter of the URL every time.

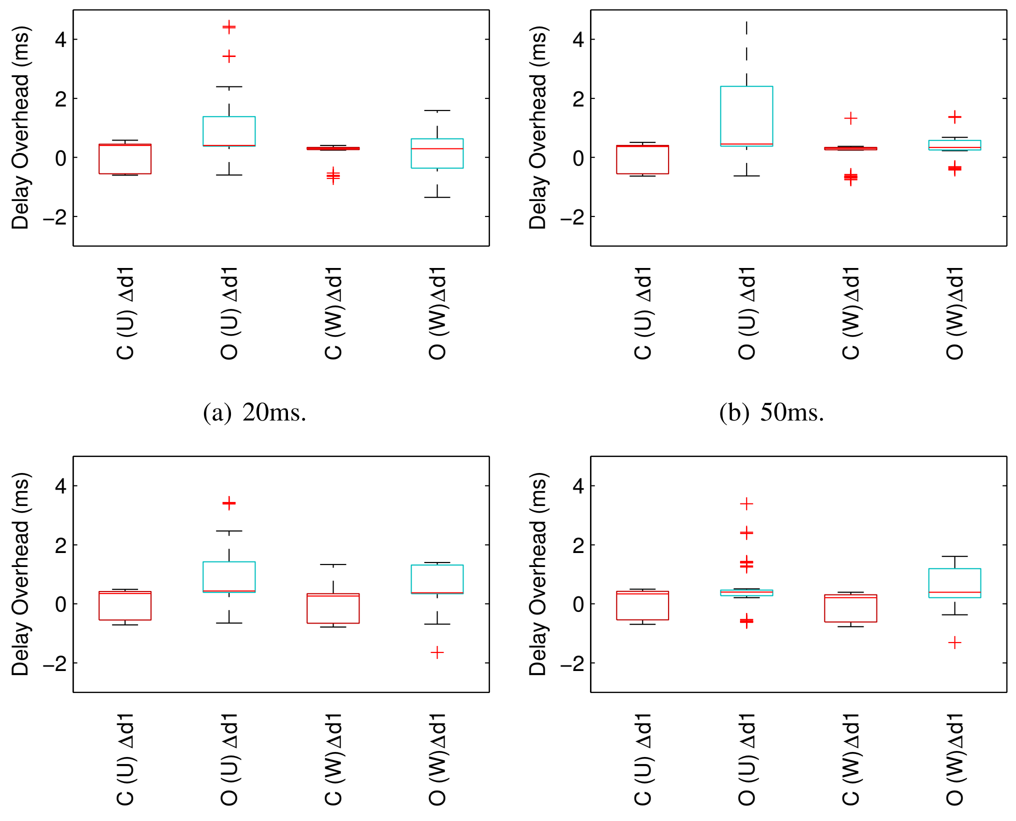
After container page being loaded, an `<iframe>` element will be inserted, with source URL being pointed to a new page on the same server. Hence we have the another data RTT d_{RTT_D} and its corresponding delay overhead, denoted by Δd_3. Since the browser has already established TCP connections to the server (usually two connections are opened), the iframe will reuse the existing connection and cannot obtain a new connection RTT. The aim of this process is to investigate whether the TCP connection establishment will make impact on the data events. The difference between $\Delta d2$ and $\Delta d3$ is that $\Delta d3$ is based on the HTTP request/response on an existing TCP connection. Moreover, the response size of the second request is smaller than the first one. When the measurement phase ends, all the recorded timestamps are stored to a database.

We use the same testbed described in Figure 3.1 except that the OSes for the client machine have been upgraded to Windows 10 and Ubuntu 12.04. We introduce an additional delay on the server side to simulate the RTT because without such delay the link RTT is too small to sample. To cover the RTTs range from small to large, we introduce four different delay respectively: 20, 50, 85, 135 ms. Our experiments are carried out through automation scripts. During the experiment, `tcpdump` is running in the background to capture the actual network delays. For each experiment, we run it for 50 times and compute from them with useful statistics, such as minimum, median, and 25% and 75% percentiles.

3.3.2 Measurement Results

In this subsection, we present our experiment results. Similar to Section 3.1, we utilize box plots to demonstrate the distribution of delay overhead. We first examine Δd_1, the delay overhead introduced in the RTT measurement using connection events. Since only two browsers (Chrome and Opera) support this type of measurement, we group their delay overheads together in Figure 3.6, in which each subplot represents how the measured RTTs are inflated in a specific emulated network delay case. We use "C (U) Δd_1" to denote the delay overhead obtained by Chrome in Ubuntu, and so forth. The plots show that the Δd_1 for both Chrome and Opera are almost all within the range from -1.5 to 2.5ms. Although the delay overhead distributions for the two browsers in different OSes can be distinct, the median values are very close to $\sim$0.4ms. Moreover, the delay overheads are observed independent to the network delay. Namely, for the same browser in the same OS, its delay overhead patterns are very similar no matter what the actual network delays are. As a consequence, the delay overheads can be more easily calibrated by repeating the measurement for many times and output the median as the final result.

We next focus on Δd_2 and Δd_3, which are introduced in the RTT measurements using data events. As depicted in Figure 3.7, most browsers can achieve very small Δd_2 (close to 0), except Opera, especially in Ubuntu. The delay overheads for Opera in Ubuntu can be as large as 58ms if the emulated RTT is 20ms. When the emulated RTTs increase to 50ms and 135ms, the upper bounds decrease to 28ms and 23ms, respectively. Curiously, when the emulated RTT is set to 85ms, Opera is observed with much smaller delay overhead, which ranges from 0 to 2ms and the median value is very close to 0. In windows, Opera performs much better than in Ubuntu. Its delay overheads are always within the range from -1ms to 2ms. The other browsers have considerable small and consistent delay overheads. When the network delays are small



(a) 20ms. (b) 50ms.

(c) 85ms. (d) 135ms.

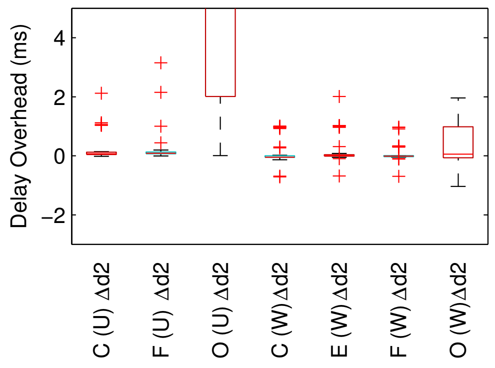
Figure 3.6: Box plots of Δd_1 obtained when the emulated network delays are different.

(20ms and 50ms), these browsers can report very accurate network delays. Even when the length of network path increases (85ms and 135ms), their delay overheads are all within -1ms to 2ms.

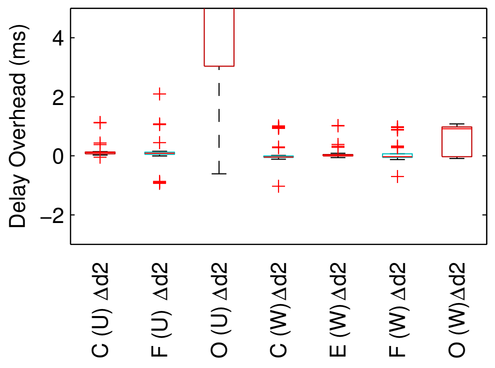
As for Δd_2, Figure 3.8 shows that all the browsers have similar performance no matter what the emulated network RTTs are. The delay overheads are all limited within the range from -1ms to 2ms. Here, the negative delay overheads are caused by the granularity of timing function. The timestamps reported by Navigation Timing technique and tcpdump/WinDump are in millisecond level and nanosecond level, respectively. If the time reported by Navigation Timing has been round up, the delay overhead could be negative values. When we consider the 25th and 75 percentiles only, the distribution of delay overheads are further narrowed down to the range from

-0.3ms to 1ms. Compared to Δd_2, although the browsers may have slightly larger Δd_3 for some cases, they show less diversity in the network delay measurement. Moreover, a more consistent result implies that the calibration can be easier.

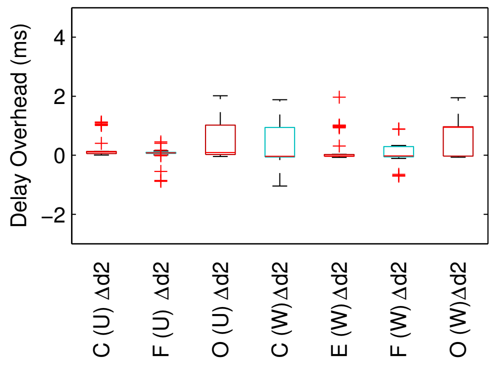
The major difference between Δd_2 and Δd_3 is that the Δd_3 is obtained on an existing TCP connection whereas the browser under test needs to make a new connection before it can measure the Δd_2. Figure 3.7 and 3.8 clearly show that event `requestStart` and `responseStart` will be significantly affected by the TCP connection establishment. Therefore, we need to avoid using the first HTTP request in practice.



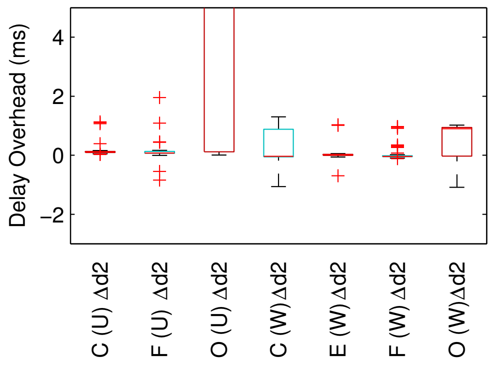
(a) 20ms.



(b) 50ms.

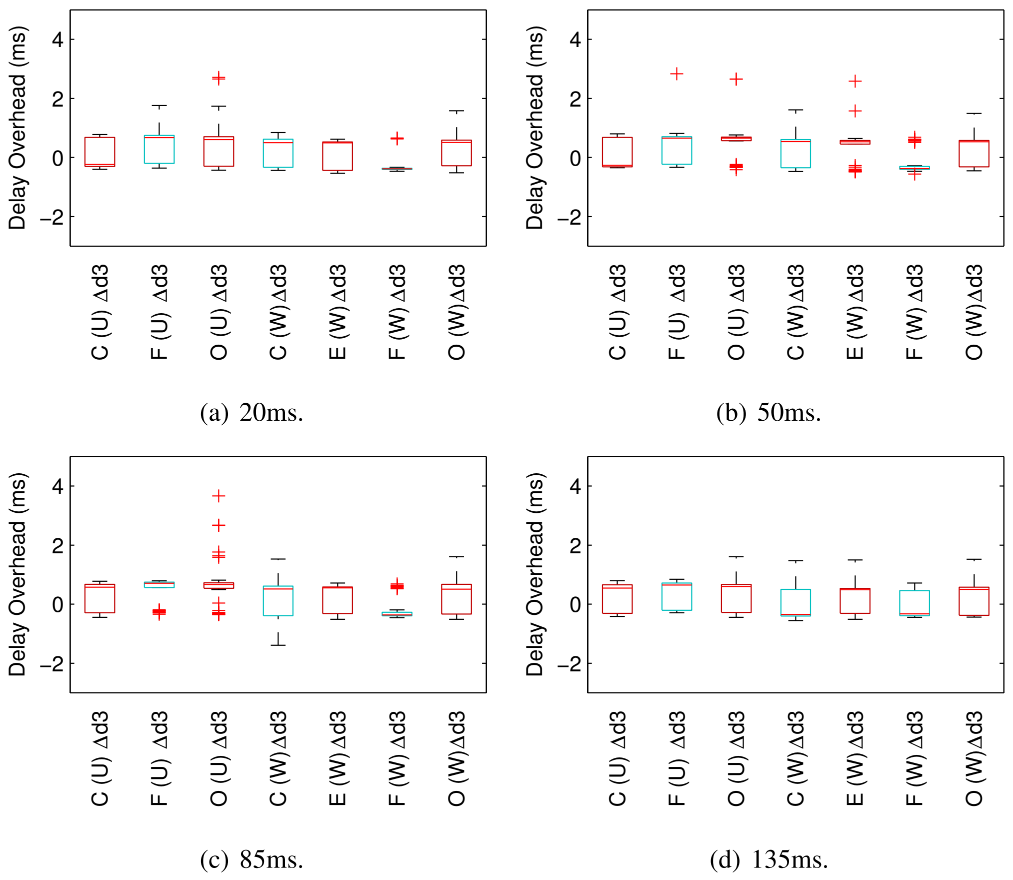


(c) 85ms.



(d) 135ms.

Figure 3.7: Box plots of Δd_2 obtained when the emulated network delays are different.



(a) 20ms. (b) 50ms. (c) 85ms. (d) 135ms.

Figure 3.8: Box plots of Δd_3 obtained when the emulated network delays are different.

3.4 WebRTC technique

The WebRTC technique provides *RTCDataChannel* API for point-to-point data exchange between two arbitrary browsers. Although the technique has not been employed by existing tools for network delay measurement, it enables client-to-client type of measurements instead of the traditional client-to-server manner. However, WebRTC is built on top of SCTP, DTLS, and UDP, which requires additional message encoding/decoding and encryption/description compared to HTTP. Therefore, it is important to investigate whether WebRTC will introduce significant delay overhead and whether it is suitable for network measurement.

3.4.1 Experiment setup

Besides a Web server hosting the container page, the browsers that intend to connect to each other need the help of a signaling server. In our experiment, we build the signaling server based PeerJS library [24]. As shown in Figure 3.9, the signaling server and Web server are hosted in the same machine with different ports. The server, local client (LC), and remote client (RC) are connected together via a switch. Here both clients have the same hardware configuration: equipped with a 1.86GHz Intel Core 2 Duo processor (E6320) and 2GB memory. Similarly, we introduce an additional delay on the RC side to simulate the RTT, because without such delay, the link RTT will be too small to sample. To cover the RTTs range from small to large, we introduce four different delay respectively: 20, 50, 85, 135 ms.

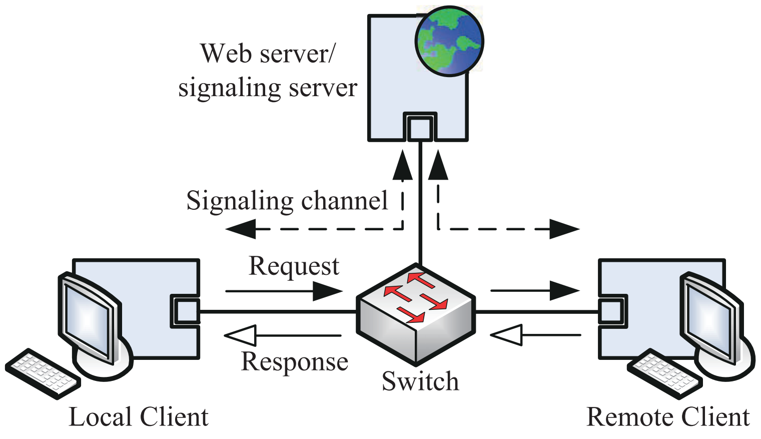


Figure 3.9: Testbed Setup for WebRTC measurement.

Before the experiment, a Chrome browser is opened and the container page is downloaded on the RC. The container page, which is also built on PeerJS library, will instantiate a `RTCPeerConnection` object and try to communicate with the signaling server, informing the server that it is ready for establishing a data channel with the LC. On the LC, our automation scripts launch the browser under test and download the container page. Likewise, the container page talks to the signaling server, and tries to connect to the RC. After negotiating the session description parameters, the data channel between the two browsers is finally established. The LC then sends out a short request message “init” to the RC, triggering a response message “ready”. By recording the message sending and receiving time, we can measure the network delay d_1. To avoid the possible effect of channel establishment, the LC subsequently sends out another request message “ping” and elicits the corresponding response message “pong”. Hence we have another network RTT d_2. Here we use short messages of smaller than 140 bytes for each so that every message can be sent in one packet [1] and can be sent in one packet[2].

We capture the actual network delay with `tcpdump` during the measurement phase. However, since we cannot decrypt the DTLS payload, it is impossible for us to map the data channel messages to the exact network packets. We therefore employ the network delay obtained with the DTLS handshake packets as the actual network delay instead. For the two measured RTTs d_1 and d_2, we then calculate their corresponding delay overheads Δd_1 and Δd_2. We have tested Chrome, Firefox, and Opera in both Ubuntu 12.04 and Windows 10. The detailed version number of each browser can be found in Table 3.6. For each browser, we run the test for 50 times and compute useful statistics from them. The experiment results are stored to a database for further analysis.

3.4.2 Measurement Results

We still utilize box plots to demonstrate the distribution of delay overhead Δd_1 and Δd_2. In Figure 3.10, each subplot corresponds to the different emulated network delay case. We use “C (U) Δd_1” to denote Δd_1 obtained by Chrome in Ubuntu, and so on.

The plots clearly show that Δd_1 is severely inflated for both Chrome and Opera.

[1]The overall header length is smaller than 130 bytes, including IP header (20-40 bytes), UDP header (8 bytes), DTLS header (20-40 bytes), and SCTP header (28 bytes).

[2]The maximum transmission unit for SCTP is 1,280 bytes.

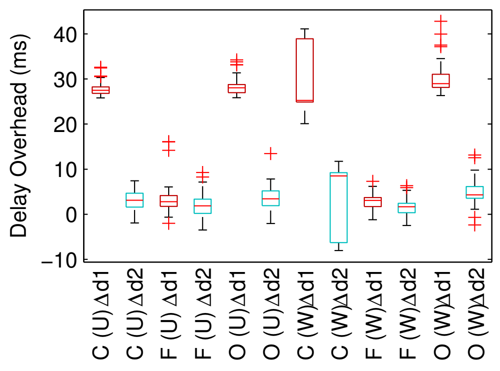
The increment can be larger than one RTT, which means that the delay of channel establishment is also included when measuring the delay d_1. Although Firefox will not include the channel establishment delay and hence has similar Δd_1 and Δd_2, its Δd_2 is still slightly smaller than Δd_1. Therefore, when WebRTC technique is employed for delay measurement, the first round of request and response should be ignored to avoid the impact of channel establishment.

When only the the second round delay measurement (d_2) is considered, the delay overhead Δd_2 is independent to the actual network delay. Namely, no matter what the emulated network RTT is, the delay overheads are very close for the same browser. However, Firefox still has the best performance in both OSes. The median values of Δd_2 for it are smaller than 2ms in all RTT cases, whereas the other two browsers are $\sim$4ms.

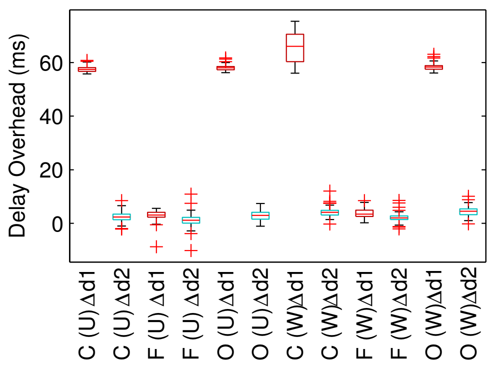
In summary, WebRTC technique enables a new type of browser-to-browser network measurement, which is different to the traditional browser-to-server measurement. Such measurement extends the scope and scale of what browser-based measurements can observe, especially provides the views from Internet edge to edge. Based on our evaluation, the measurement following the channel establishment should be ignored because there is a high possibility that the measurement results include the channel setup delay. Firefox is the preferred browser in both Ubuntu and Windows. By repeating the measurement and outputting the median values, we can keep the delay overheads caused by WebRTC smaller than 2ms. As for other browsers (Chrome and Opera), calibration could be considered to improve the measurement accuracy.

3.5 Summary

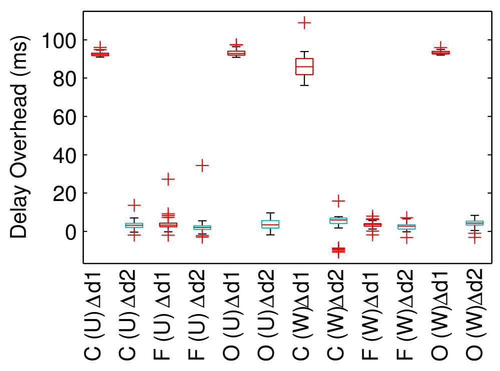
In this chapter, we studied the impact of application-level delay overheads on browser-based network measurement methodologies, including the traditional HTTP-based and



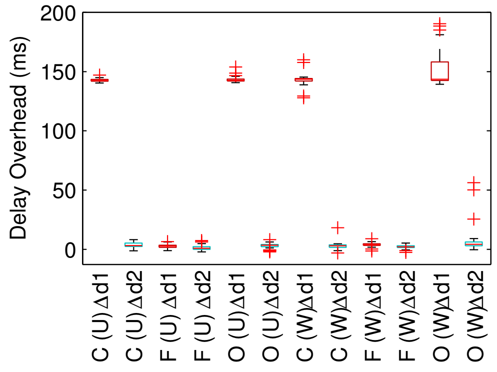
(a) 20ms.



(b) 50ms.



(c) 85ms.



(d) 135ms.

Figure 3.10: Box plots of the delay overheads (by emulated RTTs).

socket-based techniques that have already been applied to the existing measurement tools, such as XHR, Flash, WebSocket, and Java applet, and the techniques that could be employed for network measurement, such as Navigation Timing and WebRTC. By evaluating all the traditional 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. As the results show, the socket-based methods are generally more reliable than the HTTP-based methods.

On the other hand, the Navigation Timing and WebRTC methods can achieve relatively consistent delay overheads after excluding the measurement following the connection or data channel setup. Given that Navigation Timing is supported by browsers natively and does not need additional socket server, it can be a good replacement for the traditional HTTP-based and socket-based methods.

Chapter 4

Towards accurate smartphone-based network measurement

Mobile devices, notably smartphones and tablets, have already become essential parts of our daily lives because of their mobility and rich functionalities. Due to their limited computational power and storage, they rely on network access to offload intensive computation tasks to remote servers or cloud. Moreover, the offloading approach can save energy, thus extending the battery lifespan [140, 145]. Tongaonkar et al. find that 84% of apps require permission of Internet access [220] from a pool of 55K Android apps randomly chosen from the official Android app market. Therefore, understanding mobile network performance is critical for providing good quality of experience to users. For example, recent performance studies characterize LTE networks [108] and optimize mobile application performance [229]. The data collected by `Speedtest.net` is used for comparing the performance between cellular and WiFi networks [206].

The importance of monitoring mobile network quality motivates a number of studies on network performance measurement. These measurement works are conducted on mobile devices using browsers or measurement apps. The browser-based measurement is similar to Speedtest for desktop environment in that the measurement is conducted through mobile browsers [16, 31]. A more popular approach is using measurement apps on smartphones, such as [12, 17, 18, 34] for Android, [32, 33] for iOS, and [22, 35] for Windows Phone. In particular, the Ookla Speedtest app [34] has recorded over 50 million downloads in the Android app market. These measurement apps can measure network round-trip time (RTT) and upload/download throughput. Some of them can even perform traceroute, measure DNS performance, and characterize HTTP caching behavior [18].

Despite the availability of many measurement apps, their measurement accuracy has not received sufficient scrutiny. In this chapter, we appraise the accuracy of smartphone-based network performance measurement. We focus on the RTT measurement, because it is the most available atomic metric. Moreover, we consider only Android smartphones, because the source codes of the measurement apps can be available to us. In particular, we have evaluated the accuracy of Ookla Speedtest app and Mobiperf, and their RTT measurements are all inflated from a few milliseconds to tens of milliseconds. For the purpose of evaluation, we develop three test apps, each of which implements a specific measurement method: *Native ping* (using `ping` commands external to Java to send ICMP Echo requests), *Inet ping* (using network-related classes in Java/Android to send TCP SYNs), and *HTTP ping* (using HTTP-based Java classes to send HTTP GET requests). These three methods are adopted by the existing measurement apps.

Our multi-layer analysis method collects the timing information at the user space, kernel space, and the wireless network link when a packet is sent out and received by a test app. This multi-layer delay information therefore enables us to compute the delay overheads introduced by different parts of the Android phone. A major challenge in the multi-layer analysis is setting up a reliable testbed environment to obtain accurate timestamps at those localities. Unlike fixed network measurement, a single sniffer is not able to capture all the packets because of frequent missing frames. By employing multiple sniffers, we are able to merge partial traces into an almost complete trace. The entire process requires us to resolve synchronization issues for the smartphone and sniffers, recover the timestamps, and investigate the impact of clock skew between the smartphones and sniffers on the results.

We have conducted extensive testbed experiments using three Android phones with different configurations installed with the three test apps. Although the experiments are conducted in a WiFi network, part of the results can also be applied to cellular networks. Below is a summary of our findings.

1. (Highly inflated RTT measurement) The RTT measurement obtained from the three measurement methods are all inflated. Our analysis reveals that the delay inflation can occur in both user space and kernel space. We also discover that the delay inflation introduced in the user space is asymmetric for packet sending and receiving.

2. (Root-cause analysis) The overhead in Android runtime contributes to the majority of the delay overhead in the user space, and it is due to a long path of sub-function invocations. Moreover, the migration of runtime from DVM (Dalvik VM) to ART (Android Runtime) cannot help too much to alleviate the overhead. On the other hand, the sleeping function of the wireless network interface card (WNIC) driver is the major source of the delay overhead in the kernel.

3. (Mitigating the user-space delay inflation) Our approach of mitigating the delay inflation is to bypass DVM. We implement the core measurement logic into a native C program and invoke it through an external system call in the app. Experiment results show that the user-space delay inflation can be kept under 1.5ms for most of the cases.

The remainder of the chapter is organized as follows. In Section 4.1, we introduce a multi-layer approach to analyze the performance problem in Android, as well as detail the different aspects of our testbed setup, including the use of multiple sniffers to obtain a complete trace for acquiring timestamp information. Section 4.2 and §4.3 report evaluation results obtained from the Internet experiments for Ookla Speedtest and MobiPerf, and the controlled testbed, respectively. We then perform root-cause analysis in §4.4 and propose a measurement method for mitigating the delay inflation. Finally, we summarize the chapter in §4.5.

4.1 Methodology and testbed setup

4.1.1 Multi-layer analysis

To locate where the overheads are introduced, we perform multi-layer analysis by dissecting the delay overheads into several components. As mentioned in packet sending and receiving processes shown in Figure 2.6, a packet needs to be delivered to the Linux kernel before it reaches the network (for the outgoing direction) or the app (for the incoming direction). Supposing that the outgoing and incoming packets arrive at the kernel at times t_k^o and t_k^i, respectively, we calculate the *kernel-phy delay overhead* Δd_k occurred between the kernel and PHY (WNIC) as

$$\Delta d_k = d_k - d_n = (t_k^i - t_k^o) - (t_n^i - t_n^o). \tag{4.1}$$

Similarly, the *user-kernel delay overhead* Δd_u that takes place between the app and kernel can be computed as

$$\Delta d_u = d_u - d_k = (t_u^i - t_u^o) - (t_k^i - t_k^o). \tag{4.2}$$

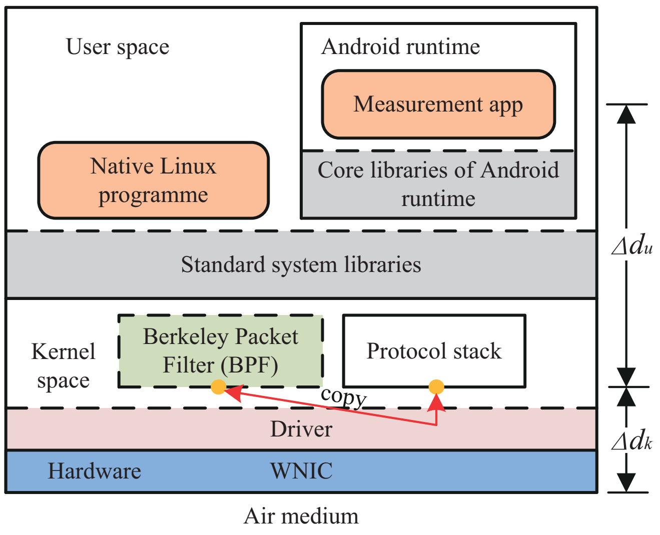


Figure 4.1: The user-kernel delay overhead and kernel-phy delay overhead.

By analyzing these two types of delay overheads, we can identify the place where the delay overheads are introduced. Note that the two overhead components are independent. As shown in Figure 4.1, since `tcpdump` timestamps a packet in the Berkeley Packet Filter (BPF), which is on top of the WNIC driver, Δd_k depends on the performance of hardware and driver of WNIC, whereas Δd_u on the performance of the user space and part of the kernel space. Although our evaluation in this chapter is based on IEEE 802.11g network, the analysis of Δd_u is still valid when the mobile network is changed to others, such as HSPA and LTE.

4.1.2 Runtime environment in Android

Although built on Linux, Android differs from other Linux distributions by employing Java as the official programming language for its platform-independency. Java achieves this characteristic by compiling the application code into bytecode and executing the bytecode through the runtime environment, Java Virtual Machine (JVM).

In Android, Dalvik VM (DVM) used to be the original runtime. But in Android 4.4, a new runtime called Android Runtime (ART) was introduced, and it finally replaces DVM since Android 5.0. The major difference between DVM and ART is the bytecode compilation methodology. DVM employs just-in-time (JIT) compilation technique, which dynamically translates the frequently executed part of bytecode into native machine code each time the app runs. On the other hand, ART utilizes ahead-of-time (AOT) compilation, compiling the entire application into native machine code during installation. With the help of AOT, ART improves the overall execution efficiency for the apps, in terms of better memory allocation and garbage collection mechanisms [4, 90].

In this chapter, we also consider the performance divergence between DVM and ART, because a faster Android runtime could lead to shorter Δd_u according to Figure 4.1. Moreover, a recent report shows that Android 4, 5, and 6 still coexist at the time of this writing (i.e., March 2016), whose market shares account for 58.9%, 36.1%, and 2.3%, respectively [25]. Therefore, our experiment design (see Section 4.3) covers the smartphones with different Android version ranging from 4 to 6, and provides comprehensive understanding of the effect of Android runtime on network measurement.

4.1.3 A multiple-sniffer testbed

To evaluate the accuracy of measurement apps, we build a multiple-sniffer testbed in Figure 4.2. The testbed consists of a measurement server (for local measurement only), which is equipped with a 1.86GHz Intel Core 2 Duo processor (E6320) and 2GB memory, and Netgear WNDR3800, an IEEE 802.11g wireless AP. The smartphone under test has been rooted, so that they can run the cross-compiled version of `tcpdump` through `adb` (Android Debug Bridge) and scripts. During the experiment, `tcpdump` is running in the background on the phone to obtain the kernel timestamps t_k^i and t_k^o. The impact of running `tcpdump` is negligible, because the traffic volume in each experiment is very small. The three external packet sniffers are run on IBM T43 laptops running Ubuntu 12.04. We also wire-connect the sniffers to the AP, so that they can be controlled through SSH.

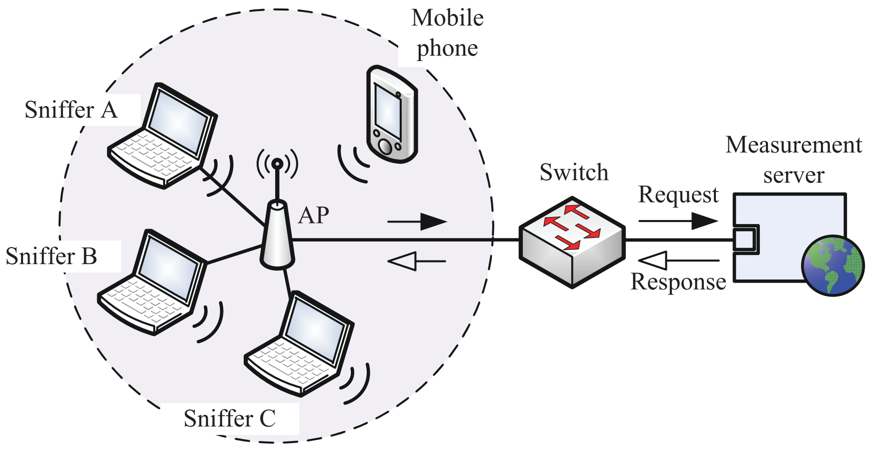


Figure 4.2: The testbed setup where the packet sniffers, mobile phone, and wireless AP are placed within a distance of 0.5m.

4.1.3.1 Wireless packet capturing

A simple way to passively monitor the wireless network traffic is to listen to the WNIC of the AP [179, 204]. However, it is difficult to extract the exact time that a packet is transmitted over the air medium, because a variable delay could be introduced after the driver passes the packet to the firmware due to the queuing, carrier sensing/back-off, or retry. It is challenging to solve the problem, e.g., employing APs that support WNIC hardware timestamp, modifying the driver, and rebuilding kernel. We therefore use the packet capturing method described in [233]. We enable the monitor mode and promiscuous mode in the wireless network adaptors of the sniffers to capture the wireless frames (including the IEEE 802.11 header, physical layer header, and higher-layer protocols' information) using `tcpdump`. To simplify the decoding of wireless frames, we also disable the security options, such as WPA. We have not performed clock synchronization among the sniffers, because hardware timestamping is not supported and software timestamping cannot meet our requirement. Instead, we use the method to be described in Section 4.1.3.2 to evade the clock drift offline.

We employ three sniffers, because a single sniffer will miss many packets [197, 233]. Although we put the AP, mobile phone, and the sniffers very close to one another (within a distance of 0.5m), we still find random frame losses and duplications in the captured traces. Such frame losses are unpredictable and independent across sniffers. To ensure the completeness of a packet trace, Serrano et al. propose to use multiple sniffers to merge the individual traces [197], because the unseen packets by one sniffer could be captured by the other sniffers. In our case, the average frame loss rate for a single sniffer is 7.3%. After merging the packet traces from the three sniffers, the trace completeness can reach to more than 99%.

4.1.3.2 Trace merging and time recovering

The basic idea of trace merging is to identify the missing frames and copy them to the incomplete trace. The detailed algorithm is shown by pseudocode in Algorithm 1. Given L ($L = 3$ in our case) packet traces for which the i^{th} trace is denoted by S_i, we first extract beacon frames and data frames into their subtraces B_i and D_i, respectively (line 2). Then we randomly assign a data trace as the main trace (denoted by D_{main}) and others as reference traces (line 3). The missing data frames in D_{main} can be identified after comparing with all other data traces (line 4). Finally, we insert the missing frames to the correct locations in the main trace (line 16) and adjust their timestamps (line 14), so that they are coherent to the local frames.

The most challenging part in this algorithm is to accurately recover the timestamps of the missing frames. A most straightforward approach is to synchronize the sniffers [234], which usually requires timestamping in the PHY for higher time synchronization accuracy. However, our sniffers do not support the feature. We therefore use reference frames (e.g., beacon frames) for "frame-level synchronization." Consider-

Algorithm 1 Trace Merging and Time Recovering Algorithm

```
Input: $\{S_i\}$ $(i = 1, \cdots, L)$.
Output: $D_o$.
 1: Count the total frame number $N$ in $\{S_i\}$.
 2: Extract data traces $B_i$ and beacon traces $D_i$ from each $S_i$.
 3: $D_{main} = D_o \leftarrow D_r$, where $r = random(1, \cdots, L)$.
 4: $D_{miss} \leftarrow \bigcup_{\substack{m=1 \\ m \neq r}}^{L} D_m - D_{main}$.
 5: Compute $\delta_{\{main,j\}}$ for each pair $\langle B_{main}, B_j \rangle$ $(1 \leq j \leq L,\ and\ j \neq r)$ with lin-
    ear regression.
 6: $f_{ref} \leftarrow f_g = FirstFrame(B)$, where $B \leftarrow \bigcap_{i=1}^{N} B_i$, $1 \leq g \leq N$.
 7: $c_0 \leftarrow C_{main}(f_{ref})$.
 8: for $n = g$ to $N$ do
 9:     if $f_n \in B$ then                          ▷ Update the reference frame.
10:         $f_{ref} \leftarrow f_n$.
11:         $c_0 \leftarrow C_{main}(f_{ref})$.
12:     else
13:         if $f_n \in D_{miss}$ then                ▷ Perform time recovering.
14:             RecoverTime($f_n$, $f_{ref}$).
15:         end if
16:         $D_o \leftarrow D_o \cup f_n$.
17:     end if
18: end for
19: Sort $D_o$ by the timestamps of each frame.
20: procedure RECOVERTIME($f_n, f_{ref}$)
21:     for $j = 1$ to $L$ and $j \neq r$ do
22:         if $f_n \in D_j$ then
23:             $c_{j0} \leftarrow C_j(f_{ref})$.
24:             $c_{jn} \leftarrow C_j(f_n)$.
25:             $C_o(f_n) \leftarrow c_0 + (c_{jn} - c_{j0}) \times \delta_{\{main,j\}}$.
26:             break
27:         end if
28:     end for
29: end procedure
```

ing the timestamp variation when a system reports its current time, simply performing linear translation between two reference frames [151] could lead to fitting errors. Accordingly, we employ a linear regression algorithm to take care of the time fluctuations and clock skews present in the sniffers (line 5).

Our time recovering algorithm first obtains the clock skews between each pair of the sniffers by applying linear regression to the beacon frames, because beacon frames are observed with the smallest time fluctuations by all sniffers. The clock skew is re-calculated for every set of data which is collected in around 180s. Our measurement results show that the clock skew progresses linearly during such a short period. When computing the timestamps of missing frames, we treat beacon frames as reference frames for the same reason.

Figure 4.3 illustrates the procedure of recovering the timestamp of a missing frame. Suppose that we want to recover a lost frame *pkt2* in the main trace A from the reference trace B. Let $C_A(t)$ and $C_B(t)$ be the times reported by sniffers A and B at time t. We denote the clock skew of A relative to B at time t by $\delta_{\{A,B\}}(t) = C'_A(t) - C'_B(t)$, where $C'_A(t) \equiv dC_A(t)/dt$ and $C'_B(t) \equiv dC_B(t)/dt$, $\forall t \geq 0$. In fact, as the clock skew between two sniffers is observed stable in our experiments for a short period (e.g., 180s), we can treat it as a constant value denoted by $\delta_{\{A,B\}}$. To recover the timestamp $C_A(t_2)$, we make use of the previous closest beacon frame as the reference frame, which is *pkt1* in Figure 4.3, in both traces:

$$C_A(t_2) = C_A(t_1) + (C_B(t_2) - C_B(t_1)) + \int_{t_1}^{t_2} \delta_{\{A,B\}} dt. \tag{4.3}$$

Since $\int_{t_1}^{t_2} \delta_{\{A,B\}} dt = \delta_{\{A,B\}} \times (t_2 - t_1)$, we have

$$C_A(t_2) = C_A(t_1) + (C_B(t_2) - C_B(t_1)) + \delta_{\{A,B\}} \times (t_2 - t_1). \tag{4.4}$$

As the interval between two consecutive beacon frames is around 100ms, the missing frame is separated from the previous closest beacon frame by no more than this value if that beacon frame is not missing. In fact, our experiments show that beacon frame is seldom lost. Given such short period of time and the typical clock skew for computer grade crystals, $t_2 - t_1$ can be replaced by $C_B(t_2) - C_B(t_1)$. Therefore, we

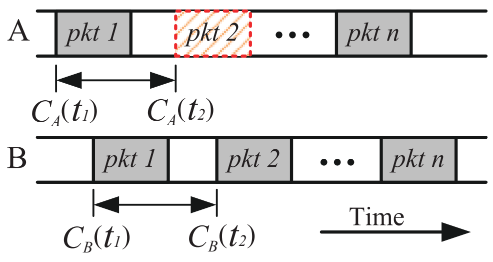


Figure 4.3: Procedure of trace merging and time recovery.

can recover $C_A(t_2)$ by

$$C_A(t_2) \approx C_A(t_1) + (C_B(t_2) - C_B(t_1)) \times (1 + \delta_{\{A,B\}}). \tag{4.5}$$

4.1.3.3 Clock skew handling

External sniffers and phones are also running different clocks. As t_s^o and t_s^i are measured from outside, we would like to know whether the RTTs estimated by the sniffers are comparable to the phones'. Similar to Section 4.1.3.2, let $C_p(t)$ and $C_s(t)$ be the times reported by the phone and sniffer at time t, and $\delta_{\{p,s\}}$ the clock skew between the phone and the sniffer. For a time interval (t_1,t_2), the difference of the measured duration $\Delta D_{\{p,s\}}$ is

$$\begin{aligned} \Delta D_{\{p,s\}} &= (C_p(t_2) - C_p(t_1)) - (C_s(t_2) - C_s(t_1)). && (4.6) \\ &= \int_{t_1}^{t_2} \delta_{\{p,s\}} dt. && (4.7) \end{aligned}$$

We have tested several Android phones and wireless sniffers. The clock skews among them are all within the range of $\pm$100 ppm (parts per million). For an end-to-end network path, the RTT is usually tens to hundreds milliseconds [85]. Taking 100ms as an example, the measured RTT difference could be smaller than 10μs, which is small enough to ignore. Therefore, the delay overhead can be computed by

$$\Delta d \approx (t_u^i - t_u^o) - (t_s^i - t_s^o). \tag{4.8}$$

4.2 Ookla Speedtest and MobiPerf

In this section, we conduct Internet experiments for two popular apps, Ookla Speedtest and MobiPerf, as well as perform multi-layer analysis to study whether they can measure the network delay accurately. We choose Speedtest and MobiPerf because Speedtest is the most popular network measurement app in Google Play (>50M installs) and MobiPerf is used for a number of research works [109, 166, 190, 191]. Moreover, these two apps cover nearly all the probe types supported by the existing apps: ICMP, TCP control packet, and TCP data packet (HTTP message). Here we do not consider UDP, because UDP-based measurements often require setting up UDP servers for the measurement.

When measuring network paths, Speedtest sends out 6 HTTP GET requests one after the other to fetch a small text file (`latency.txt`) from a web server through class `java.net.URLConnection`. By recording the packet sending and receiving times with function `SystemClock.uptimeMillis()`, Speedtest outputs the smallest RTT in integer as the final result. On the other hand, MobiPerf supports three probe types: ICMP via executing the `ping` program, TCP SYN/RST packets (on port 7) through function `java.net.InetAddress.isReachable()`, and TCP SYN/SYN ACK packets by class `java.net.HttpURLConnection`. Different from Speedtest, MobiPerf summarizes the minimal/mean/maximal RTTs from ten trials, so that users can have a more comprehensive understanding of the network quality.

4.2.1 Experiment setup

We conduct our experiments in testbed described in Section 4.1, except that the measurement target is not a local machine but a remote Web server. We randomly pick three servers, which are hosted in Hong Kong (IP: 202.45.189.9, HK in short), Taiwan (IP: 60.199.206.251, TW in short), and the Philippines (IP: 112.198.111.43, PH in short), from the Ookla server list. The actual network RTTs from our phone to these servers range from a few milliseconds to tens of milliseconds. We run the two apps one by one on Google Nexus 5, whose hardware configuration and OS version are shown in Table 5.1. Each type of measurement is repeated for 50 times. Although MobiPerf supports three probe types, we can only utilize one each time. MobiPerf performs ICMP ping measurement by default, and turns to TCP SYN/RST when ICMP ping is not successful. If failing again, it further changes to TCP SYN/ACK. In the experiments, we remove the `ping` binary after the ICMP experiments are finished, forcing MobiPerf to adopt the rest two methods. Among the three servers, the TW and PH server always refuse the TCP connection attempt on port 7 by responding RST packets, therefore MobiPerf performs only TCP SYN/RST measurements for these two servers. As for the HK server, MobiPerf employs TCP SYN/ACK packets for measurement.

4.2.2 Evaluation

4.2.2.1 Ookla Speedtest

Capturing packets both internally (through `tcpdump`) and externally (through sniffers) allows us to compare the RTTs measured by the apps (d_u), in the kernel (d_k) and in the air (d_n). As shown in Table 4.1, although Ookla Speedtest reports the smallest value as the final RTT for each measurement, this value can be around $3-7$ms larger than the actual network RTT. Even compared to d_k measured by `tcpdump`, it still overestimates the delays by $1-3$ms.

Table 4.1: RTTs measured by Ookla Speedtest (d_u), in the kernel (d_k) and in the air (d_n) (mean with 95% confidence interval, in ms).

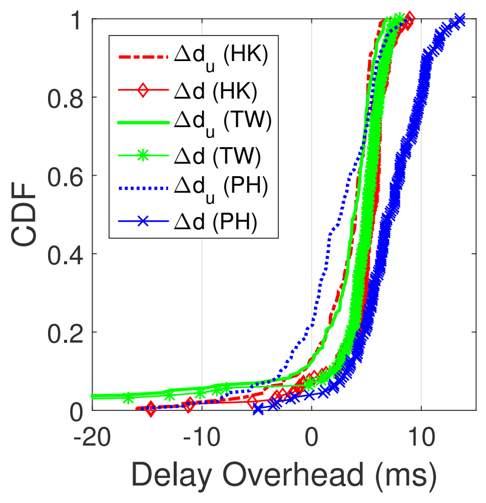
Target Server	d_u	d_k	d_n
HK	8.3 ± 0.147	5.426 ± 0.524	3.710 ± 0.459
TW	30.654 ± 0.169	29.471 ± 1.926	27.426 ± 1.381
PH	66.564 ± 0.354	64.495 ± 0.605	59.526 ± 0.432

To better understand how Speedtest inflates the network RTTs, we plot cumulative distribution functions (CDF) of delay overheads in Figure 4.4. We consider two cases: using all 6 samples for each measurement and using only the samples with the smallest d_k and d_n. For the former cases (as shown in Figure 4.4(a)), although Speedtest returns the smallest sample as the final result, it still inflates the actual network RTTs for most of the cases. For example, Δd for server PH can be as large as $\sim$14ms. Although some RTT underestimation events can be found because of performance fluctuation during the measurement, they account for only a small portion.

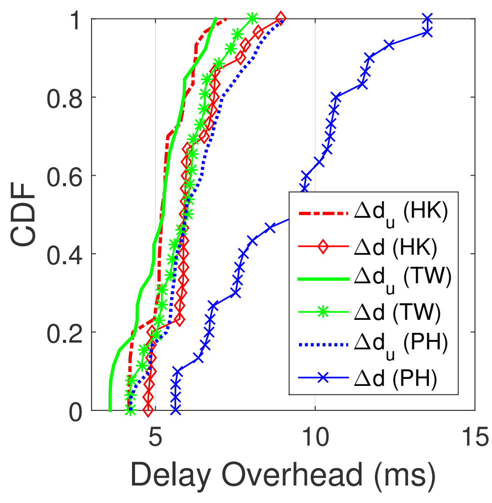
For the latter cases, since smaller d_n usually corresponds to smaller d_k and d_u, they are not affected by the performance fluctuation during the measurement. Figure 4.4(b) shows two delay overhead patterns: the inflations mainly occur between the app and kernel for servers HK and TW, because the gap between Δd_u and Δd is small and relatively constant. But for server PH, the large gap indicates that the driver and WNIC play an important role in the overall delay overheads. However, although server PH with larger network RTTs also has larger delay overheads, we cannot conclude that larger network RTT will result in larger delay overhead, because servers HK and TW have similar overhead characteristic but have distinct RTTs.

4.2.2.2 MobiPerf

Table 4.2 presents the statistics of d_u, d_k and d_n for MobiPerf. Since MobiPerf reports the min/mean/max values of its ten trials in each measurement, we also include the



(a) With all samples.



(b) With the smallest d_k and d_n.

Figure 4.4: CDF plots of Δd and Δd_u for Ookla Speedtest.

means of the reported minimum and maximum d_u in the table. In general, the mean RTTs measured by MobiPerf are inflated for all three measurement methods. Specifically, the ICMP `ping` method adds around 7−11ms additional delay, whereas the TCP SYN/ACK method up to 17ms. Even for the TCP SYN/RST method with the best performance, the overhead can be larger than 4ms. But if we use the minimum d_u as the final result, just like Ookla Speedtest, there are chances for ICMP `ping` method to underestimate the network RTT. A further analysis on the user-kernel delay overhead (Δd_u) indicates that the overheads for the ICMP `ping` method and TCP SYN/RST method mainly take place between the kernel and hardware, as the disparity between their d_u and d_k is very small (<1ms). Moreover, the TCP SYN/ACK method has a very large Δd_u (up to ∼16ms).

4.3 Testbed evaluation

Running experiments in a fully controlled environment allows us to study the behavior of different measurement methods systematically. We use six Android phones to

Table 4.2: RTTs measured by the app (d_u), in the kernel (d_k) and in the air (d_n) for MobiPerf (mean with 95% confidence interval, in ms).

Probe Type	Target Server	d_u			d_k	d_n
		min	mean	max		
ICMP `ping`	HK	2.833 ± 0.117	10.585 ± 0.130	20.621 ± 0.703	10.468 ± 0.639	3.388 ± 0.268
	TW	60.634 ± 0.238	73.184 ± 1.814	103.993 ± 16.276	73.136 ± 5.560	62.960 ± 5.557
	PH	59.719 ± 0.285	69.388 ± 0.249	79.665 ± 0.755	69.310 ± 0.802	59.009 ± 0.399
TCP S/A	HK	9.646 ± 0.207	21.658 ± 0.368	58.167 ± 1.670	5.636 ± 0.441	3.934 ± 0.361
TCP S/R	TW	59.480 ± 0.144	65.338 ± 0.144	76.083 ± 0.653	64.661 ± 0.660	60.738 ± 0.559
	PH	58.625 ± 0.154	63.563 ± 0.193	72.0 ± 0.627	62.766 ± 0.672	58.898 ± 0.424

conduct the experiments. Their detailed hardware configurations and OS versions are listed in Table 5.1. We choose these phones for their diverse hardware capability which may produce different results. The OS versions cover 4, 5, and 6. Note that all three Android 4 phones run on DVM, whereas the other three Android 5 and 6 phones are based on ART. We run three test apps (see Section 4.3.1) one by one on each phone. These apps send probes to the measurement server to elicit response packets and record the timestamps. We introduce an additional delay on the server side to simulate four different RTTs: 20ms, 50ms, 85ms, and 135ms. To avoid the RTT being affected by packet retransmission, we ensure no probe losses during the measurement. The experiment for each configuration set (phone, app, and network delay) is repeated for 100 times.

4.3.1 Building test measurement apps

Employing existing apps for systematic evaluation is difficult, because we cannot switch the measurement target to our measurement server simply and control the ac-

Table 4.3: The mobile phones used in the experiment.

Models	OS Ver.	Hardware spec.	WNIC
Sony Xperia J	4.0.4	Qualcomm MSM7227A CPU (1GHz), 512M RAM	Broadcom BCM4330
HTC One 802W	4.2.2	Qualcomm APQ8064T CPU (quad-core 1.7GHz), 2GB RAM	Qualcomm WCN3680
Google Nexus 5	4.4.2	Qualcomm MSM8974 CPU (quad-core 2.26GHz), 2GB RAM	Broadcom BCM4339
Huawei G7 Plus	5.1	Qualcomm MSM8939 CPU (quad-core 1.5GHz + quad-core 1.2GHz), 2GB RAM	Qualcomm WCN3660
Huawei Honor 7	5.0.1	HiSilicon Kirin935 CPU (quad-core 2.2GHz + quad-core 1.5GHz), 3GB RAM	Broadcom BCM4339
Huawei Mate 8	6.0	HiSilicon Kirin950 CPU (quad-core 2.3GHz + quad-core 1.8GHz), 3GB RAM	Broadcom BCM43455

tual network path delay. Moreover, their complicated GUI designs also prevent us from executing the measurements and recording results automatically. We therefore implement three test apps, each of which implements one of the three methodologies (i.e., ICMP, TCP, and HTTP GET) presented in Table 2.2. The apps follow the original design of MobiPerf and Speedtest, and the implementation details are described below:

Native ping. This app executes external shell commands through a Java `Runtime` class. It directly invokes the `ping` program, which is located at a default location `/system/bin`, to perform ICMP-based RTT measurements[1]. The `ping` program sends and receives the ICMP Echo messages on behalf of the measurement app and returns the measurement results. Although the `ping` program can

[1]Other than `ping` program, we find that executing any pre-compiled C program packaged with the app is also feasible.

only provide the resolution of 1ms or 0.1ms, it is the only way to handle ICMP packets without modifying the Android framework.

Inet ping. This app employs the method `isReachable` of class `java.net.InetAddress` to send TCP SYN packets on port 7 (Echo) to a remote host[2], eliciting TCP SYN ACKs (when the port is open) or TCP RST packets (when the port is closed).

HTTP ping. We make use of class `java.net.HttpURLConnection` to implement this app. Here the outgoing and incoming packets are complete HTTP GET request and response messages. We limit the size of HTTP request and response messages to no larger than 300 bytes, so that each message can be sent in a single TCP packet. Moreover, we record the sending time after the completion of TCP three-way handshake to avoid including the delay of connection establishment into the measurement.

To minimize the workload of the test apps on the phone, we compute all RTT estimates offline. For Native ping, the test app only parses and saves the output from the `ping` program without any further computation. Inet ping and HTTP ping simply log the timestamps of packet sending and receiving events with the system time function `System.currentTimeMillis()` or `System.nanoTime()`.

4.3.2 Overview

Table 4.4 presents the means and 95% confidence intervals of the delay overheads (Δd) measured for the three test apps (methods) and four emulated RTTs on the six test phones. Compared with the RTTs observed by the external sniffers, the RTTs measured by the apps are inflated significantly for all six phones. The delay overheads can range from a few milliseconds to tens of milliseconds, and the 95% confidence interval

[2]Although the official documentation (http://developer.android.com/reference/java/net/InetAddress.html) states that the method first tries ICMP and falls back to TCP when it fails, we find that the ICMP option has not been implemented.

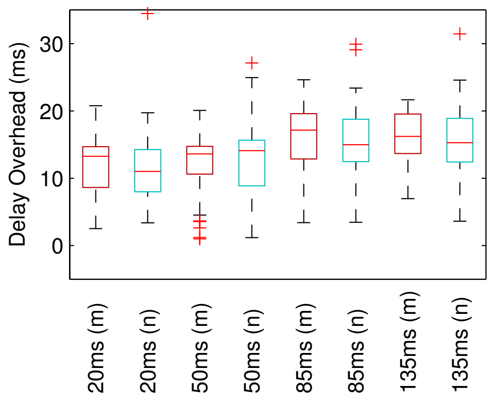
can be as high as 2.9ms. The inflated RTT measurement is too significant to ignore, considering the network delay today is getting smaller due to the prevalence of CDNs and cloud services. For example, the median RTT from University of Connecticut to Akamai-Hartford servers is only 8.5ms [69].

Generally speaking, HTTP ping exhibits comparatively smaller delay overheads for most of the cases (except phone S). For example, the mean delay overheads for phone W1 ($<$2.6ms) are much smaller than its Native ping ($>$7.8ms) and Inet ping ($>$13ms) cases. Inet ping has relatively larger Δds, which mean values are usually larger than 10ms. For some extreme cases, the overheads can be close to 20ms. Another observation is that ART cannot help much on reducing the delay overhead, though the three smartphones run on Android 5 and 6 (W1, W2, and W3) have more powerful computation capabilities. Compared to the three Android 4 phones, their mean delay overheads are close to or even larger for all three measurement methods.

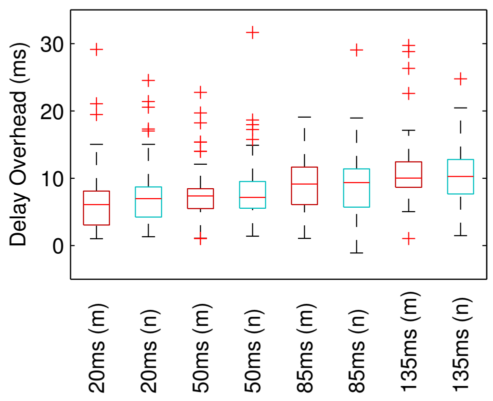
Two different delay inflation behaviors can be also observed. For the phones equipped with Qualcomm WNIC chipsets (H and W1), their delay overheads can be considered RTT-independent due to the small variations when the emulated RTTs increase. However, the other four phones powered by Broadcom (G, S, W2, and W3), there are significant delay overhead increments when the emulated RTTs are long. A typical example is phone G. When the emulated RTTs are 20ms and 50ms (short RTTs), its mean Δds are $\sim$7ms, $\sim$12ms, and $\sim$7ms for Native ping, Inet ping, and HTTP ping, respectively. But when the RTT increases to 85ms and 135ms (long RTTs), the mean Δds increases to $\sim$14ms, $\sim$16ms, and $\sim$10ms, which increments can be 3-7ms.

4.3.3 Effect of timing functions

The results presented in Table 4.4 are measured when `System.currentTimeMillis()` is used. Since it is reported that this function could have coarse granularity (such as ∼15ms) in some OS [143], we also implement the test apps with the more precise `System.nanoTime()` for the purpose of comparison. We perform experiments with the same setting described in Section 4.1.3, and link the results together with the ones obtained by `System.currentTimeMillis()`. To better visualize the effect of the two timing functions, we use box plots to present the data in Figure 4.5. In each box-and-whisker plot, the top and bottom of the box are given by the 75th 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.



(a) Inet ping.



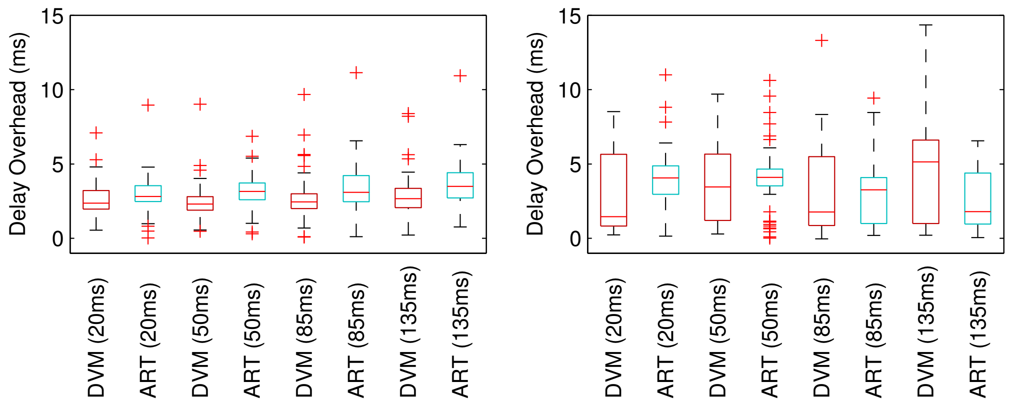
(b) HTTP ping.

Figure 4.5: Delay overhead comparison in box plot for phone G (red/m for System.currentTimeMillis(), and cyan/n for System.nanoTime()).

We only present the data of phone G in detail, since the other two phones have similar results. The figures show that the delay overheads measured by `System.nanoTime()` is similar to those by `System.currentTimeMillis()`. Considering the relatively large delay inflation, the overhead of executing a timing function is therefore not a key factor to consider for measurement accuracy.

4.3.4 Effect of runtime

Android 4.4.2 allows us to switch runtime between DVM and ART in the developer options. Therefore, we run Inet ping and HTTP ping on phone G with the same experiment settings to examine their delay overheads in ART. We link the results with those obtained in DVM on the same phone (described in Section 4.3.2) in box plots, as shown in Figure 4.6. Here we do not consider Native ping, because the `ping` program is not executed in the runtime but runs as a native Linux program.



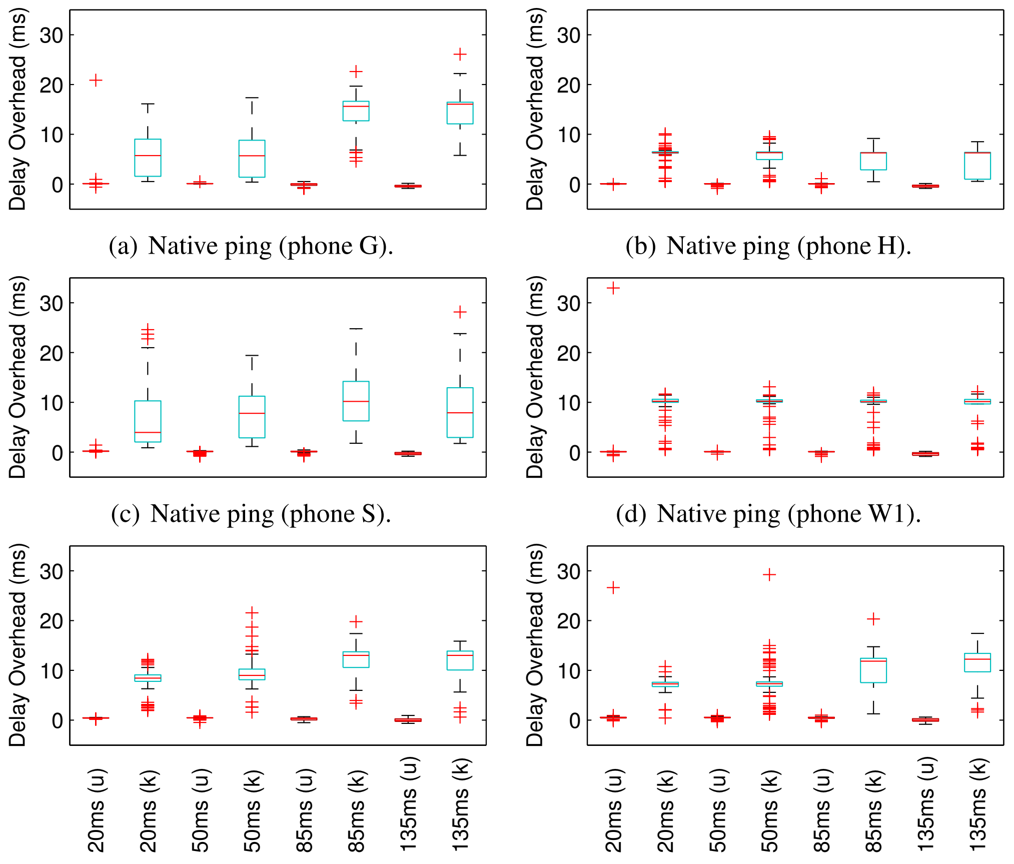
(a) Inet ping. (b) HTTP ping.

Figure 4.6: Delay overhead comparison in box plot for phone G when different runtimes are adopted (red for DVM, and cyan for ART).

Figure 4.6(b) clearly shows that for HTTP ping, both the interquartile range and the total range of delay overheads have been narrowed down significantly when ART is applied. Although the median Δds in ART may be higher than those in DVM, we can conclude that ART can make the delay overheads more stable for HTTP ping. However, as depicted in Figure 4.6(a), Inet ping has higher Δds with ART. This observation can also be confirmed by Table 4.4, where the delay overheads measured by

W1-W3 are usually higher than the other three phones. We will discuss the reasons in Section 4.4.1.

4.3.5 User-space and kernel-space overheads



(a) Native ping (phone G).

(b) Native ping (phone H).

(c) Native ping (phone S).

(d) Native ping (phone W1).

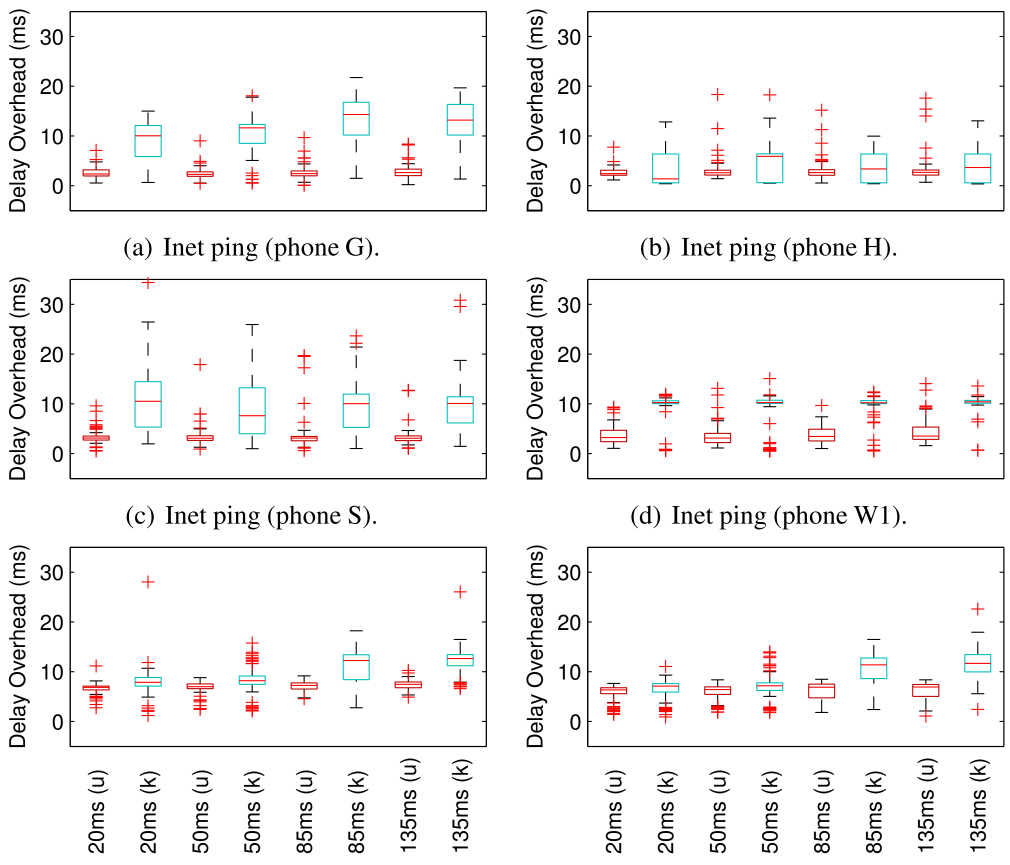
(e) Native ping (phone W2).

(f) Native ping (phone W3).

Figure 4.7: Box plots for the user-kernel delay overheads (Δd_u, red) and kernel-phy delay overheads (Δd_k, cyan) measured by Native ping.

As described in Section 4.1.3, during our previous experiments, we also run `tcpdump` in the background on those three test phones to obtain t_k^o and t_k^i in the kernel space, which allows us to perform multi-layer analysis. We calculate and plot Δd_u and Δd_k in box plot in Figure 4.7, 4.8, and 4.9.

We first focus on Δd_u experienced by the three test apps. In general, Δd_u can be



(a) Inet ping (phone G).

(b) Inet ping (phone H).

(c) Inet ping (phone S).

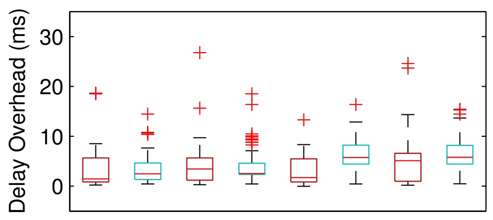
(d) Inet ping (phone W1).

(e) Inet ping (phone W2).

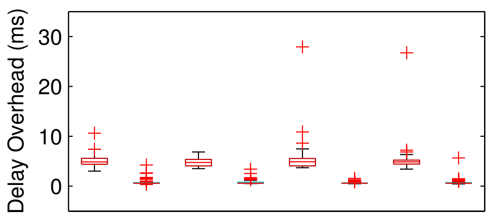
(f) Inet ping (phone W3).

Figure 4.8: Box plots for the user-kernel delay overheads (Δd_u, red) and kernel-phy delay overheads (Δd_k, cyan) measured by Inet ping.

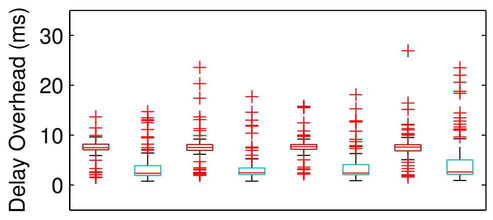
considered as RTT-independent, because each test app experiences very close Δd_u in a same phone no matter what the emulated network RTT is. Figs. 4.7(a), 4.7(b), 4.7(c), 4.7(d), 4.7(e), and 4.7(f) for Native ping clearly show that Δd_u for all six phones is very close to 0, suggesting that the packets are mainly delayed between the kernel and physical link. Native ping shows two different types of patterns. For the phones run DVM (G, H, and S), Δd_k contributes the majority of the total delay overheads, as shown in Figure 4.8(a), 4.8(b), and 4.8(c), which is similar to Native ping except that the layer above the kernel space adds 2-4ms more delay. But for the rest of the phones, Inet ping encounters much larger Δd_u (see Figure 4.8(d), 4.8(e), and 4.8(f)). Especially for W2 and W3, Δd_u can be around 6-7ms. As for HTTP ping, the phones



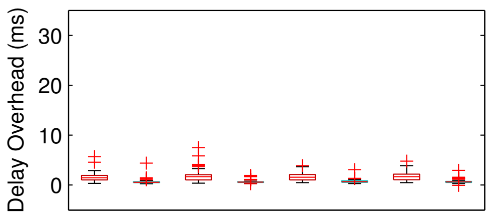
(a) HTTP ping (phone G).



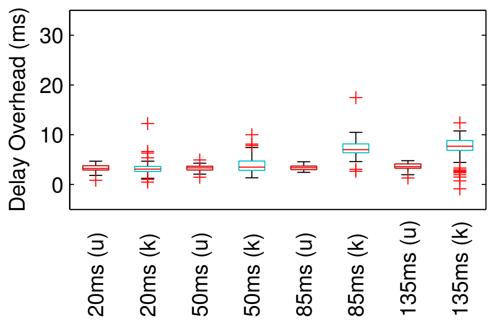
(b) HTTP ping (phone H).



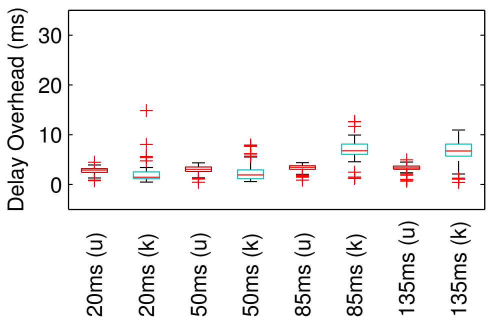
(c) HTTP ping (phone S).



(d) HTTP ping (phone W1).



(e) HTTP ping (phone W2).



(f) HTTP ping (phone W3).

Figure 4.9: Box plots for the user-kernel delay overheads (Δd_u, red) and kernel-phy delay overheads (Δd_k, cyan) measured by HTTP ping.

with DVM (phone G[3], H, and S) experience much larger Δd_u (usually larger than 5ms) compared to the other three ART phones (∼2-4ms).

To sum up, our analysis shows that Native ping introduces nearly no overhead between the app and kernel, but Inet ping and HTTP ping will. Note that the major difference between Native ping and the others is the measurement execution manner: external system call vs. in app. In the external system call, the external `ping` runs as a native Linux program, whereas the app in the in-app approach is implemented in Java APIs and runs as an instance of the runtime virtual machine. In fact, invoking

[3]Although phone G has a relatively small median Δd_u when the emulated RTT is 20ms, its 75th percentile and maximum values are close to or even larger than 10ms. Therefore we still classify phone G in the same group for phones H and S.

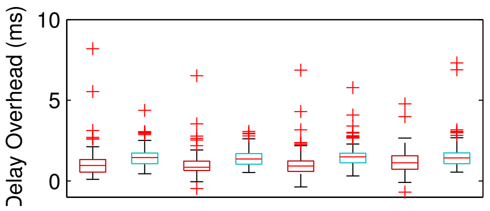
a Java API usually involves several more function calls (see Section 4.4.1). For each additional call, the runtime needs to consume more bytecode instructions (e.g., pushing parameters into virtual registers). Moreover, network-related Java APIs are finally mapped to the bionic C library, which is equivalent to the BSD's standard C library, through Java Native Interface (JNI). Due to the extra translation, JNI could also lower the performance. Therefore, performing network measurement within an app could result in more delay than a native Linux program.

Different from Δd_u, Δd_k shows two different behaviors. For phones H and W1, which employ the Qualcomm WNIC chipsets, their Δd_k can be also considered as RTT-independent. But for the rest equipped with Broadcom WNIC chipsets, Δd_k increases significantly when the RTT is long (85ms and 135ms). The inconsistency of Δd_k is the main reason why we observe the obvious increment of overall delay overheads in Section 4.3.2.

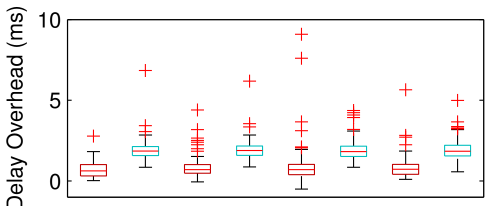
4.3.6 Delay overhead asymmetry

Running `tcpdump` also allows us to analyze the (a)symmetry of the delay overheads occurring in the app. Since Android uses the same clock source of the underlying Linux system, the timestamps recorded by the measurement apps and `tcpdump` are comparable. Therefore, we can measure the outgoing user-kernel delay overhead $\Delta d_u^o = t_k^o - t_u^o$, and the incoming delay overhead $\Delta d_u^i = t_u^i - t_k^i$. We plot the distributions of the overheads per direction in Figure 4.10 for Inet ping and in Figure 4.11 for HTTP ping. Note that we cannot analyze Native ping, because the external `ping` program does not provide the packet send and receive times.

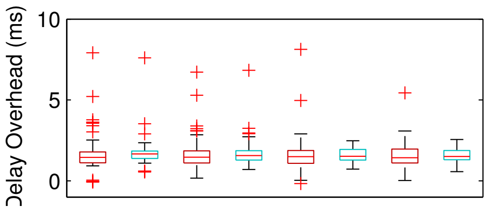
Both Figure 4.10 and 4.11 show significant delay asymmetry. For example, for Inet ping, establishing a TCP connection costs more time in the outgoing direction. Especially for phones W1, W2, and W3, the disparity can be larger than 3ms. On the



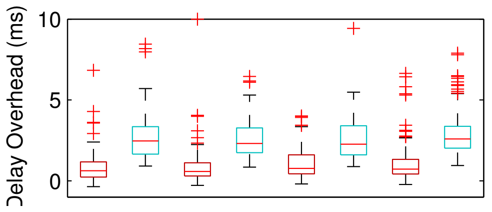
(a) Phone G (Google Nexus 5).



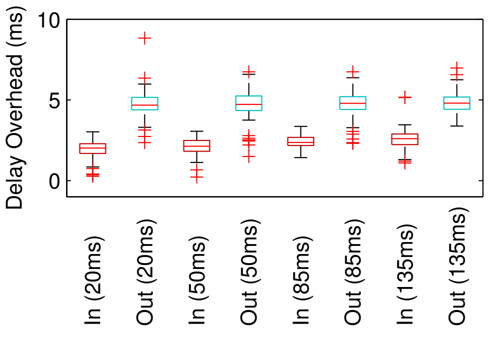
(b) Phone H (HTC One).



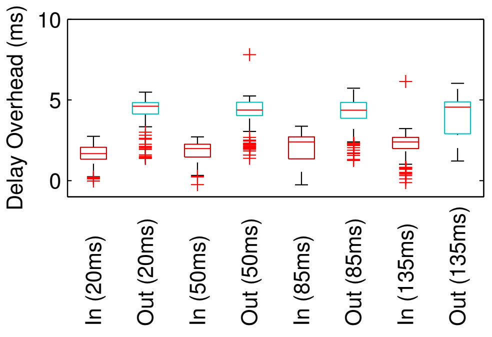
(c) Phone S (Sony Xperia J).



(d) Phone W1 (Huawei G7 Plus).



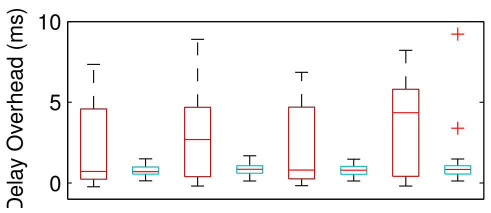
(e) Phone W2 (Huawei Honor 7).



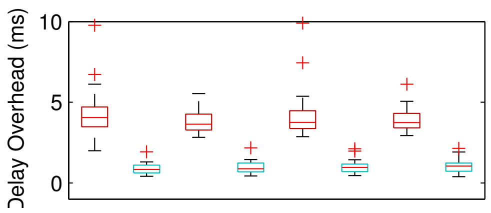
(f) Phone W3 (Huawei Mate 8).

Figure 4.10: Box plots of the delay overhead asymmetry for Inet ping.

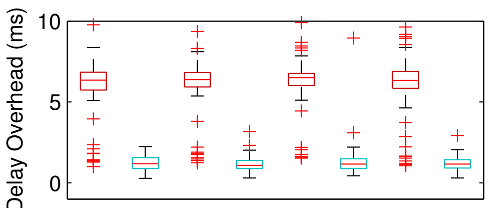
other hand, the majority part of the user-kernel delay overhead occurs when receiving and processing HTTP messages for HTTP ping. The only exception is phone W1, which spends more time on sending HTTP messages. Moreover, phones W1, W2, and W3 experience much smaller incoming delay overheads than phone G, H, and S. Our further analysis in Section 4.4.1.3 shows that the performance difference between Android 4 and 5/6 is mainly due to the Java I/O library change.



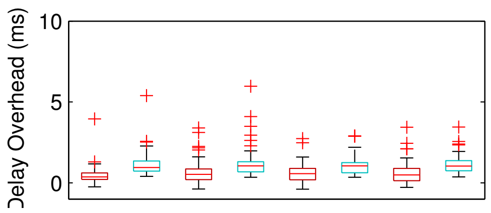
(a) Phone G (Google Nexus 5).



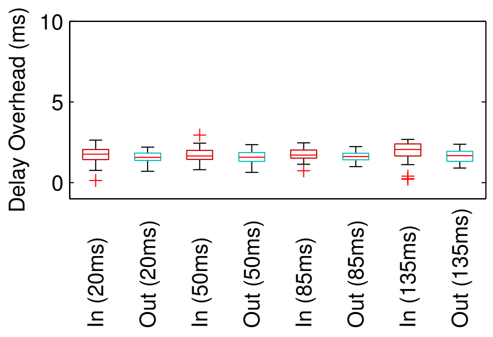
(b) Phone H (HTC One).



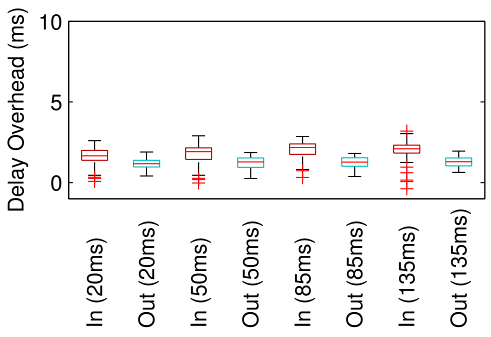
(c) Phone S (Sony Xperia J).



(d) Phone W1 (Huawei G7 Plus).



(e) Phone W2 (Huawei Honor 7).



(f) Phone W3 (Huawei Mate 8).

Figure 4.11: Box plots of the delay overhead asymmetry for HTTP ping.

4.4 Discussion

4.4.1 Delay overhead in user space

Android provides `Debug` class and `Traceview` tool to trace and profile function executions in runtime [26]. When the trace/method based profiling feature is enabled, the names of the function/class/method, thread IDs, and execution times of each action involved in a function invocation in runtime will be recorded. We start and stop function tracing by calling `Debug.startMethodTrace()` and `Debug.stopMethodTrace()`, respectively, before and after the core measurement methods to obtain the execution times. As Traceview can analyze the function behavior only in the runtime layer, we further examine the source code of the Android framework (e.g., `libcore_io_Posix.cpp`) and map the functions to the native ones in the system layer.

Besides tracing the function calls, we also study the performance of each function in terms of its execution time. Since `Debug` class provides the timestamps of a function with microsecond resolution at its entry point and exit point, we can use them to calculate its execution time. As for the functions executed in the system layer, we use `strace` [36] to profile their performance. Note that we cannot enable `Debug` feature and `strace` simultaneously, because `strace` could introduce significant system overhead for `Debug` class to record timestamps.

We trace the function calls in Google Nexus 5 when DVM is enabled. Figs. 4.12, 4.13, and 4.14 show the cross-layer function call sequences for the three Java functions that can be employed for different types of network measurements in DVM. We include only the key functions that consume most of the time and leave other minor functions to the grey boxes. The value within the angle brackets behind a function name shows the execution time of this function in μs. We obtain this value by repeating the function profiling process for ten times and calculating the mean execution time. For the purpose of comparison, we also perform function tracing in Huawei G7 Plus and Huawei Mate 8, as well as Nexus 5 with ART enabled. However, as ART supports only sampling-based profiling, the function calls cannot be traced step by step. As a result, some of the functions may be skipped and do not show in the log file. We consequently highlight the major difference between ART and DVM in red dotted rectangle in the figures. Although we cannot directly compare the execution time of each function call between Nexus 5 and other phones, such analysis can still help better understand the delay overhead diversity described in Section 4.3.

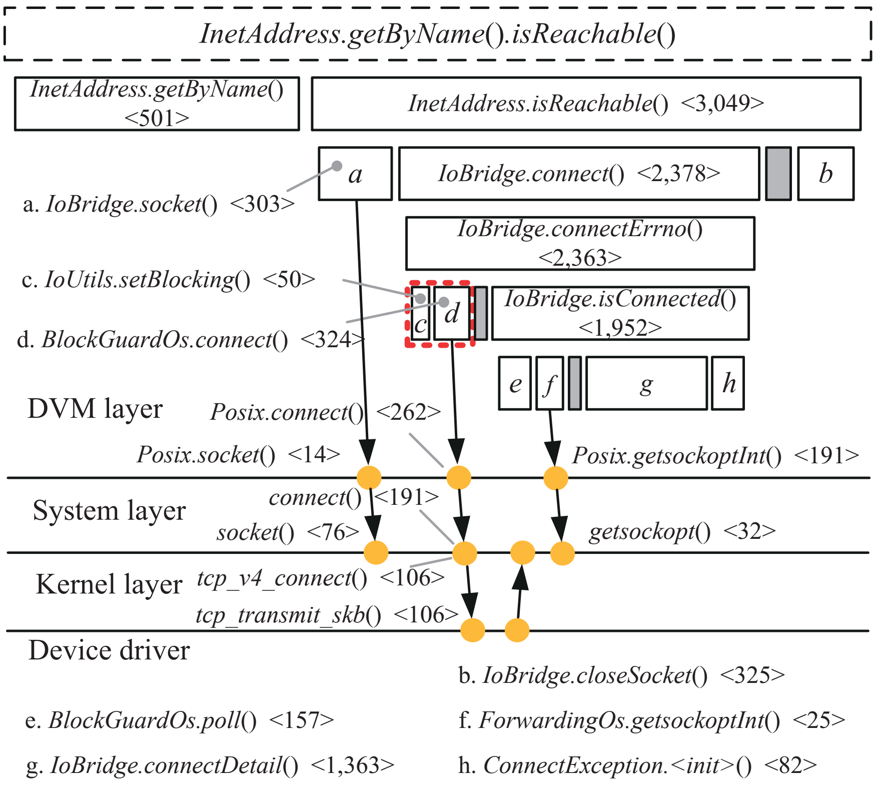
4.4.1.1 `InetAddress.getByName().isReachable()`

This function consists of two methods. It takes $\sim$500μs for the class `java.net.InetAddress` to parse the address information (via method `getByName`). For method `isReachable`, it involves several sub-functions defined in `libcore/io/IoBridge.java`, such as `IoBridge.socket()` and `IoBridge.connect()`, which can be further traced to POSIX functions `Posix.socket()` and `Posix.connect()`, and finally bridged to system-level functions `socket()` and `connect()` defined in the bionic libc library (`sys/socket.h`). Since the connect timeout value has been set, this socket is therefore set to non-blocking mode (via `IoUtils.setBlocking()`), and `connect()` returns an error code for “Operation In Progress” immediately. DVM then checks the socket status (via `BlockGuardOs.poll()`). If it finds that the connection trial is refused by the remote endpoint, it obtains the error message (via `Posix.getsockoptInt()`) and closes the socket (via `IoBridge.closeSocket()`).

We believe the asymmetry of Δd_u described in Section 4.3.6 for Inet ping is caused by the interpretation in the runtime virtual machine, because the execution time analysis clearly shows that it also takes more than 300μs for the runtime to prepare a socket and another 300μs to send out a TCP SYN packet. Along with the overhead caused by method `getByName` ($\sim$500μs), it totally costs more than 1.1ms from the start point (which is also the point that we record the sending time t_u^o in the app) to the point the TCP SYN packet is sent (which is captured by `tcpdump`).

The experiments in Section 4.3.4 and 4.3.6 indicate that Δd_u can increase slightly (see Fig. 4.6(a)) or significantly (see Fig. 4.10(d)-4.10(f)) when ART is adopted. Our function call trace on Nexus 5 with ART enabled show that there is no obvious difference between ART and DVM in function calls. Although Huawei G7 Plus introduces many additional string operations between function c and d (as shown in Fig. 4.12, it may not be the source of the increase of Δd_u in outgoing direction, because Huawei

Mate 8 has a similar function call behavior compared to Nexus 5's DVM case. That is, if the string operations are the root cause, Mate 8 would not inflate its Δd_u. Therefore, we tend to attribute the increment of Δd_u to the worse runtime efficiency of ART when handling the `InetAddress` class.



InetAddress.getByName().isReachable()
InetAddress.getByName() <501>
InetAddress.isReachable() <3,049>
a
IoBridge.connect() <2,378>
b
a. IoBridge.socket() <303>
IoBridge.connectErrno() <2,363>
c. IoUtils.setBlocking() <50>
c
d
IoBridge.isConnected() <1,952>
d. BlockGuardOs.connect() <324>
e
f
g
h
DVM layer
Posix.connect() <262>
Posix.socket() <14>
Posix.getsockoptInt() <191>
System layer
connect() <191>
socket() <76>
getsockopt() <32>
Kernel layer
tcp_v4_connect() <106>
tcp_transmit_skb() <106>
Device driver
b. IoBridge.closeSocket() <325>
e. BlockGuardOs.poll() <157>
f. ForwardingOs.getsockoptInt() <25>
g. IoBridge.connectDetail() <1,363>
h. ConnectException.<init>() <82>

Figure 4.12: Major function calls involved in `InetAddress.getByName().isReachable()`. The mean function execution time is also included in the angular brackets (in μs).

4.4.1.2 `HttpURLConnection.connect()`

This function first creates an HTTP engine (via `HttpURLConnectionImpl.initHttpEngine()`) in runtime and then establishes a TCP connection (via `HttpURLConnectionImpl.execute()`). Similar to `InetAddress.isReachable()`, the socket is also set to non-blocking mode due to the existence of connection timeout. Our performance analysis shows that this function is not appropriate for network measurement, because it requires more than 3ms to prepare the HTTP engine, and ∼1.5ms before sending out the TCP SYN packet. Even for the reverse path, the function `Platform.getMtu()` adds a large delay. The analysis also agrees with our Internet experiment results for MobiPerf's TCP SYN/ACK method.

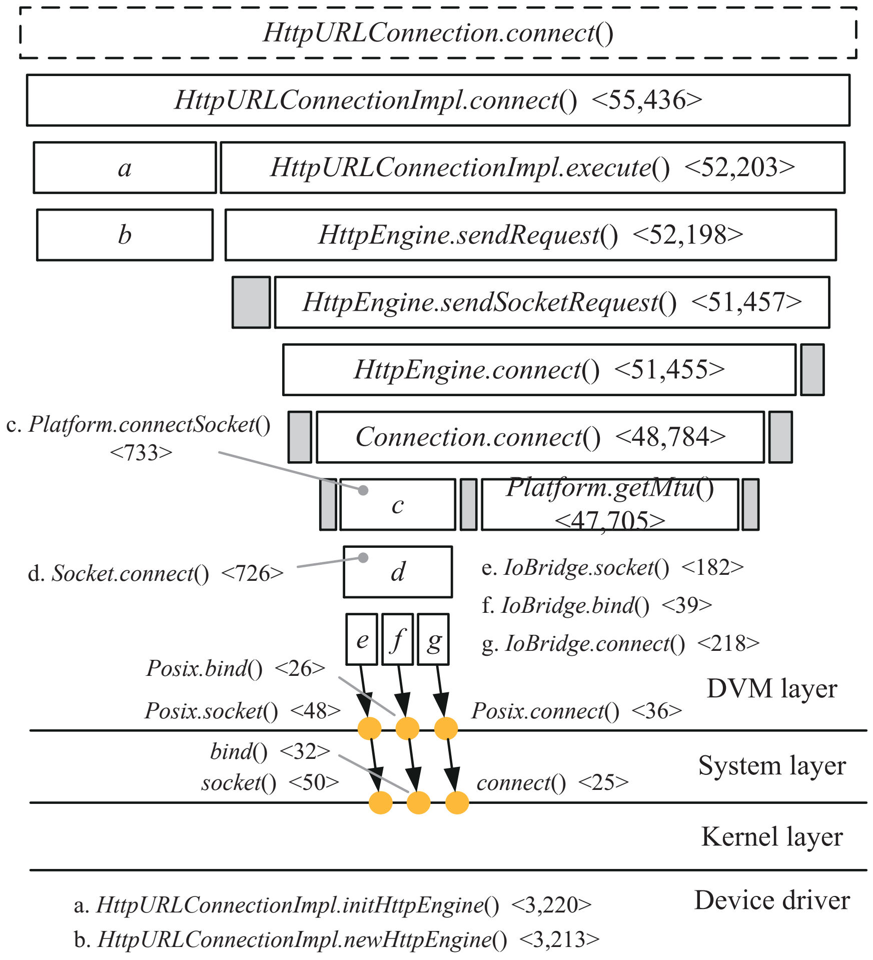

HttpURLConnection.connect()
HttpURLConnectionImpl.connect() <55,436>
a
HttpURLConnectionImpl.execute() <52,203>
b
HttpEngine.sendRequest() <52,198>
HttpEngine.sendSocketRequest() <51,457>
HttpEngine.connect() <51,455>
c. Platform.connectSocket() <733>
Connection.connect() <48,784>
c
Platform.getMtu() <47,705>
d. Socket.connect() <726>
d
e. IoBridge.socket() <182>
f. IoBridge.bind() <39>
g. IoBridge.connect() <218>
e
f
g
Posix.bind() <26>
Posix.socket() <48>
Posix.connect() <36>
DVM layer
bind() <32>
socket() <50>
connect() <25>
System layer
Kernel layer
Device driver
a. HttpURLConnectionImpl.initHttpEngine() <3,220>
b. HttpURLConnectionImpl.newHttpEngine() <3,213>

Figure 4.13: Major function calls involved in `HttpURLConnection.connect()`. The mean function execution time is also included in the angular brackets (in μs).

4.4.1.3 `HttpURLConnection.getInputStream()`

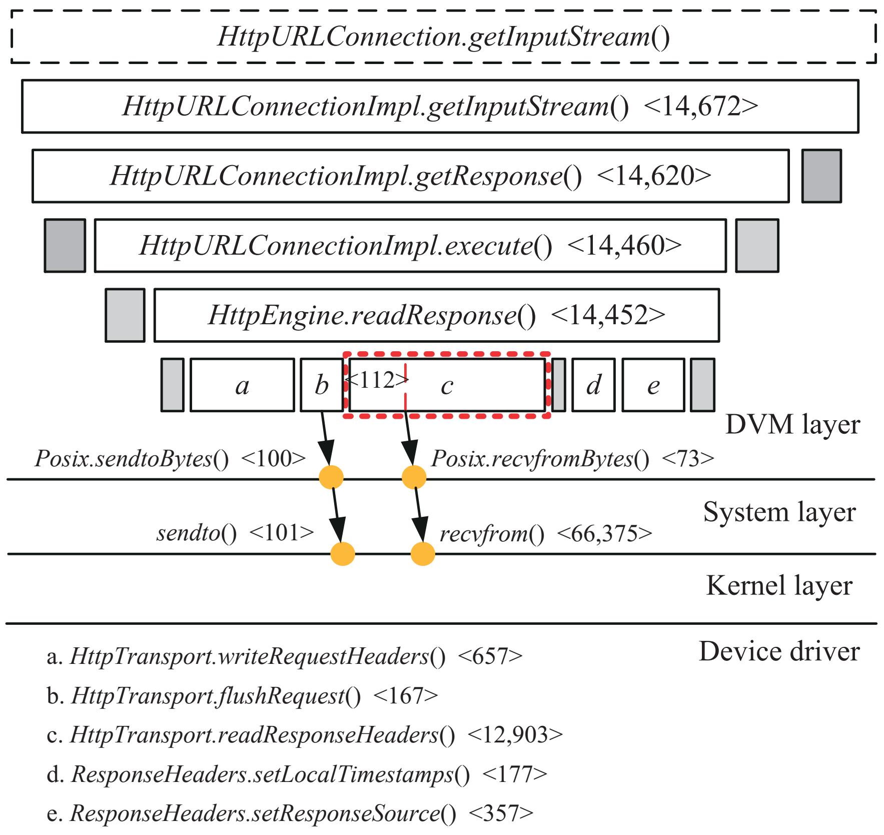
As the HTTP engine has already been prepared, this function can send and receive messages in an established connection. However, we still observe more than 600μs overhead for the HTTP engine to construct the request packet (`HttpTransport.writeRequest`

Headers()). The actual packet sending operation is realized by HttpTransport.flush Request(), which further invokes a POSIX function Posix.sendtoBytes() and is finally bridged to the system function sendto(). In the incoming direction, the response packets are received by the runtime virtual machine (via Posix.recvfromBytes()), whose native function in the system layer is recvfrom(). The HTTP engine receives the whole HTTP message (via HttpTransport.readResponseHeaders()), which totally consumes more than 12ms. Our analysis shows that it takes $\sim 20\mu$s to exit from the packet receiving function and spends most of the time ($>$10ms) on parsing the received data. This explains why the HTTP ping method introduces more delay overheads in the incoming direction in Fig. 4.11.

Our profiling shows that no significant function call difference can be observed between ART and DVM on Nexus 5. Therefore, the delay overhead diversity of HTTP ping (as described in Section 4.3.4) may be considered as a result of the improvement of runtime efficiency. But for Android 5/6, our function call tracing shows that they differ from DVM in that the fundamental IO functions when receiving data have been replaced from the default java.io to a third-party library okio. We believe both the runtime efficiency and the new IO library lead to the significant delay overhead degradation of HTTP ping in the incoming direction.

4.4.2 Delay overhead in kernel space

By studying the source codes of Android and Linux, we can map the functions in the system layer to the kernel space. We focus on the kernel functions involved when the system calls the socket functions (i.e., connect(), sendto(), and recvfrom()). To profile the performance of those related kernel functions, we compile a custom kernel with kprobes [130] enabled and replace the default kernel on Nexus 5. We then build a loadable kernel module and hook the function to be monitored. When the kernel



HttpURLConnection.getInputStream()
HttpURLConnectionImpl.getInputStream() <14,672>
HttpURLConnectionImpl.getResponse() <14,620>
HttpURLConnectionImpl.execute() <14,460>
HttpEngine.readResponse() <14,452>
a
b
<112>
c
d
e
DVM layer
Posix.sendtoBytes() <100>
Posix.recvfromBytes() <73>
System layer
sendto() <101>
recvfrom() <66,375>
Kernel layer
Device driver
a. HttpTransport.writeRequestHeaders() <657>
b. HttpTransport.flushRequest() <167>
c. HttpTransport.readResponseHeaders() <12,903>
d. ResponseHeaders.setLocalTimestamps() <177>
e. ResponseHeaders.setResponseSource() <357>

Figure 4.14: Major function calls involved in `HttpURLConnection.getInputStream()`. The mean function execution time is also included in the angular brackets (in μs). Here the execution time of function `recvfrom()` includes the network delay.

enters and leaves the function, the corresponding trap functions will be triggered, thus allowing us to measure its performance. We collect ten samples for each major kernel function and report their mean execution times in Table 4.5. The execution time analysis in both user space and kernel space show that the system socket functions and their underlying kernel functions are not the major sources of the user-kernel delay overhead.

As it is not easy to hack the hardware parts of WNIC, we analyze the source code of the driver to seek the root cause on how the kernel-phy delay overhead is introduced. Taking Nexus 5 as an example, its WNIC chipset (Broadcom BCM4339) connects to the system through SDIO bus and “bcmdhd” driver (in “`drivers/net/wireless/bcmdhd`”).

The "bcmdhd" driver maintains a `dpc` kernel thread for handling packet sending and receiving. Before the `dpc` thread can send or receive packets, it always checks the status of SDIO bus (via `dhdsdio_bussleep()`), as well as the readiness of backplane clock (via `dhdsdio_clkctl()`). We find that the driver puts the SDIO bus into sleeping state frequently if the data transmission rate is not high. When there are packets to send or receive, it takes around 10ms for the driver to bring up the bus. We also investigate the criteria that trigger the bus to enter the sleep mode. Our driver analysis shows that the driver maintains a counter ($idlecount$) for that purpose. For every $dhd\_watchdog\_ms$ of time (default value is 10ms), the driver increases this counter by 1 if the hardware is idle. When the counter reaches a threshold ($idletime$ whose default value is 5), the driver will instruct the bus to sleep. Therefore, the default idle period is 50ms. Moreover, $idletime$ and $dhd\_watchdog\_ms$ are both configurable when the driver is loaded.

In fact, the "bcmdhd" driver also supports other Broadcom chipests, such as BCM4329, BCM4330 (used by Sony Xperia J), and BCM4335 (used by HTC One). Therefore, other phones that use Broadcom chipsets could encounter the same problems that will introduce significant kernel-phy delay overhead to the RTT measurement. This explains why phones G, S, W2, and W3 will have a large Δd_k (>10ms) when the emulated RTTs are increased to 85ms and 135ms (described in Section 4.3.5: the SDIO bus has entered the sleeping mode before the phone can receive the response packet (>50ms), and it takes more than 10ms for it to wake up to process the packet. The detailed analysis of the driver is discussed in Chapter 5.

4.4.3 A better practice

Our analysis in both user space and kernel space shows that the long path of subfunction invocations in the DVM layer is responsible for the user-kernel delay overhead.

Therefore, to improve the measurement accuracy, we must avoid using those functions that will incur too many irrelevant subfunctions. Another strategy is to bypass the DVM layer altogether and migrate the timestamping and networking functions to native Linux environment. We therefore implement a simple C socket program which supports RTT measurements with TCP SYN/RST packets and HTTP GET request/response messages. Similar to HTTP ping, we limit the size of the HTTP messages to no more than 300 bytes, so that each message can be transmitted in one TCP packet. We employ `clock_gettime()` to record the send and receive timestamps. After cross-compilation, the executable binary is packed into a test app, called *External ping*. This app can invoke the binary through the Java class `Runtime`. We test the app with the same settings described in Section 4.1.3 and compute Δd_u based on Eqn. (4.2).

We compare Δd_u measured by External ping to the other two in-DVM apps in Table 4.6. We present only the results obtained by Nexus 5, because the other two phones have similar characteristics. As expected, Δd_u drops after employing the external system call, with a decrease of 1.6ms$-$2.2ms for the TCP SYN/RST method and 1.9ms$-$3.2ms for the HTTP GET method. Besides, the overheads are more stable with the confidence intervals smaller than 0.2ms. We also find that the HTTP ping introduces 0.4ms$-$0.5ms more delay than Inet ping. The additional delay is due to the fact that HTTP messages need to be further processed in the user space, but handling TCP SYN/RST packets can be completed within the kernel.

Our modification of External ping does not require root privilege, thus facilitating a wide deployment of the app. By repeating the measurements and computing the mean or median, we can keep the user-kernel delay inflation under 1.5ms for most of the cases. Although External ping cannot completely remove the delay overhead, the measurement results it produces are much closer to the real network RTTs. On the other hand, the overhead in the driver is difficult to remove without modifying the driver source code. A possible solution is to increase the packet sending rate, preventing the driver from entering the sleeping mode. In Chapter 5, we will provide a solution to address this problem.

4.4.4 Beyond WiFi and delay measurement

Our modification of External ping and part of the analysis on delay overheads can be applied to cellular network, because measurement apps in cellular network still have to face the problem of inefficient runtime. Similar to WNIC, the cellular network interface is responsible for translating between the PHY PDUs and IEEE 802.3 Ethernet frames. Therefore, the data that the kernel can handle are the same as the WiFi network. That is, there is no difference in the data path of the kernel and user space. Bypassing the overhead in the runtime can definitely mitigate the user-kernel delay overheads and improve the measurement accuracy in cellular network.

Although our study focuses on network delay measurement, other performance metrics, such as delay variation or jitter, will also be affected by the runtime and the driver's energy-saving mechanisms. On the other hand, throughput measurements, which are usually based on flooding methodology, can be slightly underestimated due to the longer data path in the runtime. The energy-saving mechanism will not impact the throughput measurement too much, because the continuous packet sending and receiving activities will keep the driver always in the awake state. Therefore, when calculating the throughput, excluding the first few papers will give more accurate results.

For other apps that are sensitive to the network delay, our analysis and strategy can also shed light on improving their performance, because they are also affected by the inefficient Java function calls and various energy-saving mechanisms when sending and receiving data packets. Implementing the network functions in native Linux programs and executing them through JNI can avoid many irrelevant subfunction calls. Moreover, periodically sending out small packets in the background can keep the underlying network interface awake, so that packets can be sent and received immediately.

4.5 Summary

In this chapter, we appraised the accuracy of measurement apps in Android phones. We overcame the main challenge of obtaining accurate packet timestamps from the wireless medium and set up a reliable wireless testbed. Both Internet experiments and testbed evaluation showed that the RTTs measured by the apps with different methods are significantly inflated. After conducting careful investigations through multi-layer analysis, we identified the delay overhead introduced by the runtime virtual machine is significant and asymmetric in the send and receive directions. Our analysis further showed that the long path of subfunction invocations in runtime accounts for the overhead in the user space, while the sleeping features in the driver cause the kernel-phy delay inflation. Finally, we proposed to mitigate the delay overhead by implementing a native measurement app, so that its user-kernel delay overhead can be reduced to less than 1.5ms.

Table 4.4: Delay overheads measured when `System.currentTimeMillis()` is used (mean with 95% confidence interval, in ms).

	Phone*	Emulated RTT (ms)			
		20	50	85	135
Native ping	G	7.700 ±2.331	6.028 ±0.811	14.078 ±0.684	13.963 ±0.691
	H	6.02 ±0.352	5.355 ±0.517	4.880 ±0.549	4.216 ±0.553
	S	6.779 ±1.129	7.840 ±0.932	9.999 ±1.039	8.387 ±1.191
	W1	9.623 ±0.514	9.328 ±0.615	8.842 ±0.722	7.868 ±0.861
	W2	8.447 ±0.478	11.031 ±2.335	12.165 ±0.607	11.825 ±0.648
	W3	10.169 ±2.812	9.857 ±2.789	10.785 ±0.713	13.221 ±2.923
Inet ping	G	11.931 ±1.063	12.514 ±0.779	16.211 ±0.833	15.874 ±0.787
	H	7.243 ±1.907	7.470 ±0.815	8.551 ±2.413	7.060 ±0.821
	S	13.822 ±1.327	12.223 ±1.142	12.814 ±1.146	12.511 ±1.055
	W1	13.460 ±0.613	13.044 ±0.968	13.576 ±0.591	14.561 ±0.608
	W2	14.576 ±0.676	15.157 ±0.606	18.448 ±0.720	19.433 ±0.656
	W3	12.209 ±0.569	12.917 ±0.634	16.792 ±0.759	17.447 ±0.821
HTTP ping	G	6.481 ±0.855	7.651 ±0.963	9.156 ±0.703	10.790 ±0.911
	H	5.861 ±0.307	5.541 ±0.218	6.002 ±0.813	5.945 ±0.709
	S	11.206 ±0.947	11.153 ±0.855	11.805 ±0.987	12.987 ±1.312
	W1	2.269 ±0.257	2.517 ±0.308	2.450 ±0.266	2.478 ±0.262
	W2	6.557 ±0.360	7.211 ±0.510	10.575 ±0.557	10.780 ±0.769
	W3	4.826 ±0.510	5.526 ±0.488	10.187 ±0.529	9.942 ±0.662

Note *: G for Google Nexus 5, H for HTC One, S for Sony Xperia J, W1 for Huawei G7 Plus, W2 for Huawei Honor 7, and W3 for Huawei Mate 8.

Table 4.5: The execution times (in μs) of major kernel functions for the socket functions in the system layer.

System	Kernel	Execution time
`connect`	`tcp_v4_connect()`	106
	`tcp_v4_rcv()`	68
	`tcp_ack()`	72
	`tcp_send_ack()`	86
`sendto`	`tcp_sendmsg()`	83
	`tcp_transmit_skb()`	85
	`ip_queue_xmit()`	81
	`dev_queue_xmit()`	96
`recvfrom`	`tcp_v4_rcv()`	ditto
	`ip_rcv()`	86
	`netif_rx()`	73

Table 4.6: A comparison of Δd_u for external C socket program (Ext) and in-DVM measurement (App) (mean with 95% confidence interval, in ms).

	Type	Emulated RTT (ms)			
		20	50	85	135
TCP S/R	App	2.946 ± 0.695	2.443 ± 0.200	2.637 ± 0.251	2.828 ± 0.236
	Ext	0.736 ± 0.121	0.794 ± 0.139	0.798 ± 0.154	0.830 ± 0.134
HTTP GET	App	3.312 ± 0.663	3.824 ± 0.721	3.157 ± 0.540	4.542 ± 0.834
	Ext	1.095 ± 0.075	1.246 ± 0.098	1.289 ± 0.112	1.365 ± 0.186

Chapter 5

Mitigating Delay Overheads in Smartphone-based Measurement

In Chapter 4, our evaluations have shown the the network-level round-trip time (denoted by nRTT) are all inflated by the smartphone-based measurements, ranging from a few milliseconds (ms) to tens of milliseconds, and the amount of inflation varies across smartphone models, measurement methods, the actual nRTT, and the sending direction. As CDN and cloud services continue to reduce the end-to-end delay, this delay inflation will significantly over-estimate the actual nRTT. Although the overhead introduced in the Dalvik VM (DVM) can be mitigated by a native C implementation of measurement methods which bypasses the expensive function calls in the DVM, the overhead introduced in the driver is difficult to remove without modifying the driver source code. Moreover, we have studied only WiFi network so far.

In this chapter, we systematically investigate and identify the sources of the delay inflation on Android phones, including both WiFi and 2/3/4G cellular networks. As the first main contribution, we discover and demonstrate that various energy-saving mechanisms employed in these mobile networks and smartphones are the main sources of the delay inflation. Our analysis also shows that these delay inflation is dependent to the smartphone, network path, or the ISP's own configuration. Therefore, given the same network path, two different smartphones may obtain quite different nRTTs. Our objective is to mitigate and stabilize these noises as much as possible. Besides, other performance measurement, such as one-way delay [83, 84], jitter, available bandwidth, and capacity [159], will also benefit from more accurate nRTTs.

For WiFi networks, a main energy-saving mechanism is the Secure Digital Input Output (SDIO) bus sleeping in the phones. To our knowledge, we are the first to report the effect of SDIO bus sleeping on the nRTT measurement. Moreover, the IEEE 802.11 Power Saving Mode (PSM) operated between the phone and the access point (AP) also inflates the nRTT. Although this additional delay is caused by the additional time for buffering packets at the AP, we still consider it as a measurement noise, because it also depends on the state of the smartphones (i.e., sleep or not). There are many works on addressing the impact of PSM [58, 103, 183, 194], but their focus is more on scheduling the packets on the AP side to achieve a better balance between energy consumption and network delay. For cellular networks, it is well known that the low-power RRC states will cause additional delay. However, the previous studies on RRC state transition [108, 184, 190] do not propose any concrete algorithm to effectively mitigate the impact of RRC states on nRTT measurement.

In our second contribution, we propose to mitigate the impact of the energy-saving mechanisms by enforcing the smartphones to operate in the wake-up mode for WiFi networks and high-power state for cellular networks during the delay measurement. Therefore, in addition to sending probes to measure nRTT, we employ a set of "warm-up" packets to move the phone to the wake-up mode/high-power state and a set of background packets to keep it stay in those states. The main challenge of realizing this approach is to use a smallest number of background and warm-up packets to achieve the objective, because sending too many packets may bias the measurement. Our approach is to send these packets to the first-hop routers and providers' DNS servers which are usually close to the smartphones. To minimize the number of warm-up packets, we exploit the values of the state promotion delays and demotion timers to maintain the smartphones in the wake-up mode and high-power state. We have implemented this approach in AcuteMon, an Android app run on unrooted phones and requiring no system customization, such as kernel recompilation and customized ROM.

We have conducted testbed experiments to validate the approach in WiFi networks by using five smartphones equipped with different versions of Android systems. The additional delay is kept within 4-5ms for all the tests, regardless of the actual nRTTs. Moreover, the delay variations are very small. Since it is not possible to perform testbed experiments for cellular networks, we have implemented a set of pre-defined experiments for volunteers to run them on their phones. We have collected 252 sets of valid results from 28 participants who use 26 different smartphone models. Among them, 146 sets of tests were done in cellular networks, involving 9 ISPs. The experiment results show that AcuteMon can achieve much smaller nRTTs compared to four other popular methods. A further analysis indicates that AcuteMon can effectively mitigate the effect of RRC state transition and therefore achieve accurate results.

The rest of the chapter is organized as follows. We first conduct experiments and perform root-cause analysis on the delay overheads in mobile networks in Section 5.1, and then propose AcuteMon in Section 5.2 to mitigate the delay inflation. After evaluating its performance in Section 5.3, we summarize this chapter in Section 5.4.

5.1 Delay overhead caused by energy-saving mechanisms

In this section, we perform root-cause analysis to explain how the network delay are inflated in both WiFi and cellular networks. To locate the exact place that incurs the inflation, we apply the multi-layer analysis described in Section 4.1.1.

5.1.1 Effect of packet sending interval

Our root-cause analysis begins with an ICMP ping experiment conducted in the testbed described in Section 4.1.3. We run a `ping` program through `adb shell` for 100 times with two packet sending intervals (10ms and 1s), measuring the nRTTs between the smartphones under test and the measurement server. In this chapter, we test five different smartphones in Table 5.1. For the root-cause analysis, we use only Google Nexus 4 and Nexus 5, because they employ the WNIC chipsets manufactured by Qualcomm (WCN 3660) and Broadcom (BCM 4339), respectively. As most smartphones employ the WNIC chipsets provided by these two manufacturers [198], any delay overhead caused by these chipsets and drivers can be captured by these two phones. To emulate the real Internet environment, we set the nRTT to 60ms with `tc` command on the server side.

Table 5.1: The smartphones used in the testbed evaluation.

Models	Ver.	Hardware spec.	WNIC
Google Nexus 5	4.4.2	Quad-core 2.26GHz CPU, 2GB RAM	Broadcom BCM4339
Google Nexus 4	4.4.4	Quad-core 1.5GHz CPU, 2GB RAM	Qualcomm WCN3660
HTC One 802W	4.2.2	Quad-core 1.7GHz CPU, 2GB RAM	Qualcomm WCN3680
Sony Xperia J	4.0.4	1GHz CPU, 512M RAM	Broadcom BCM4330
Samsung Grand	4.1.2	Dual-core 1.2GHz CPU, 1GB RAM	Broadcom BCM4329

Table 5.2 summarizes the result of the multi-layer nRTT measurement. The measurement for both phones consistently reports smaller d_u when the packet sending interval is small. As the sending interval increases to 1s, d_u for Nexus 4 is increased by more than 100% and that for Nexus 5 is around 30%. Moreover, the nRTT variations measured by both phones are much more significant for a longer sending interval. For

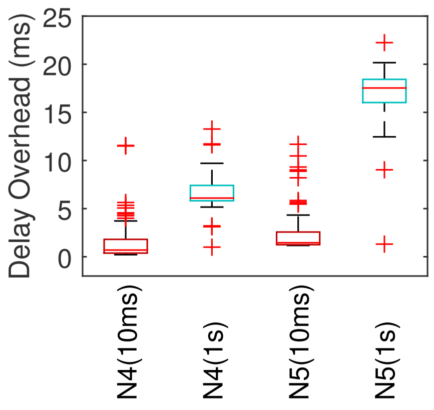
example, the 95% confidence interval for Nexus 4 can be as large as 6ms.

When the sending interval is small, the RTTs captured by `tcpdump` (i.e., d_k) and the external wireless sniffers show that both are very close to d_n. However, with the interval of 1s, the nRTT for Nexus 4 is inflated mainly in the network, because d_n is increased by more than double. In contrast, Nexus 5's nRTT inflation occurs inside the phone, because most of the delay inflation is observed by d_k (28%) but not by d_n (1%).

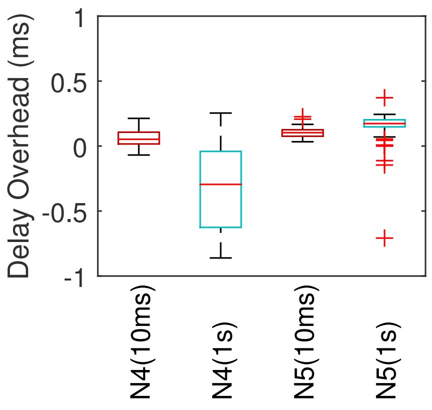
Table 5.2: Experiment results (mean with 95% confidence interval in ms).

Phone	Interval	d_u	d_k	d_n
Google Nexus 4	10ms	63.877 ± 0.652	63.765 ± 0.686	62.253 ± 0.432
	1s	136.330 ± 6.023	136.656 ± 5.968	130.032 ± 6.107
Google Nexus 5	10ms	64.184 ± 0.675	64.080 ± 0.672	61.613 ± 0.346
	1s	81.983 ± 2.035	81.829 ± 2.045	62.353 ± 0.428

To better visualize the distribution of the delay overheads, we employ box-and-whisker plots to present Δd_{k-n} in Figure 5.1(a) and Δd_{u-k} in Figure 5.1(b). In each plot, the mark inside the box is the median and the top and bottom are the 75th and 25th percentile. The upper and lower whiskers are the maximum and minimum, respectively, after excluding the outliers. Figure 5.1(a) clearly shows that Nexus 4 and 5 experience comparably small Δd_{k-n}, which are smaller than $\sim$4ms, when the packet sending interval is small. With the interval of 1s, Nexus 5 has a much larger Δd_k than Nexus 4 ($\sim$18ms vs. $\sim$6ms in median). On the other hand, since Δd_{u-k} is very close to 0 for both Nexus 4 and 5, Δd_{u-k} is not a major source of delay inflation. Moreover, some values of Δd_{u-k} are negative due to the low resolution of the reported `ping` results.



(a) Δd_{k-n}.



(b) Δd_{u-k}.

Figure 5.1: Kernel-phy delay overhead (Δd_{k-n}) and User-kernel delay overhead (Δd_{u-k}) for Google Nexus 4 and 5.

5.1.2 Root cause analysis

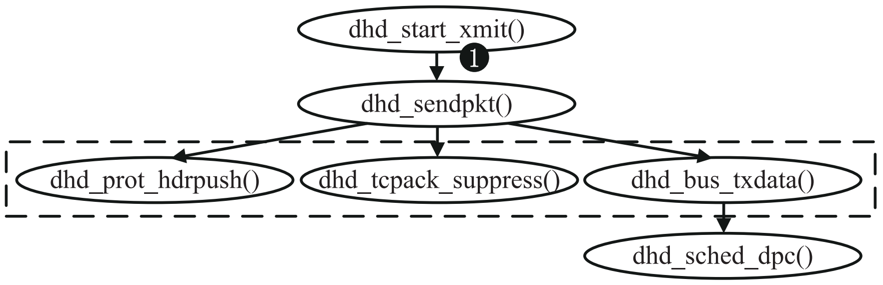
In this section, we investigate the root causes for the two sources of delay overhead discovered in Section 5.1.1. For the kernel-phy delay experienced by both phones (and more so for Nexus 5), we will dissect the WNIC driver's source codes in Nexus 5. For the overhead in the wireless link experienced by Nexus 4, we will measure the PSM timeout values and correlate them with the increase in d_n. We then examine the impact of the low-power RRC states in the cellular networks on the delay measurement.

5.1.2.1 Driver analysis

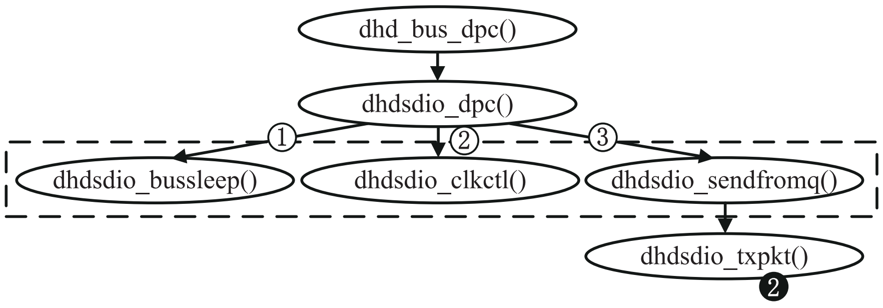
We first analyze the source code of the WNIC driver in Nexus 5. The WiFi chipset used by Nexus 5 (Broadcom BCM 4339) connects to the system through SDIO bus and adopts the "bcmdhd" driver [1]. This driver also supports other Broadcom WNIC chipsets, such as BCM 4329, 4330, 4335, and others. Therefore, the finding here is also applicable to the Sony Xperia and Samung Grand in Table 5.1 and other phones equipped with Broadcom WiFi chipset, especially those with FullMAC MLME (MAC Sublayer Management Entity).

[1]In "`drivers/net/wireless/bcmdhd`"

We trace the function calls in the packet sending and receiving directions. For packet sending, the kernel function `dev_queue_xmit()`, which transmits `sk_buff` to the network device, further maps to driver function `dhd_start- _xmit()`. As shown in Figure 5.2(a), this function further calls `dhd_sched_dpc()`, registering a packet sending task in a kernel thread, `dpc`. The `dpc` maintains a `while(1)` loop in its sub-function `dhdsdio_dpc()`. Before the `dpc` thread can send packets (③ in Figure 5.2(b)), `dhdsdio_dpc()` needs to check the status of SDIO bus and the readiness of backplane clock (① and ②, respectively). Eventually, function `dhdsdio_txpkt()` is executed, and the data are written to the bus.



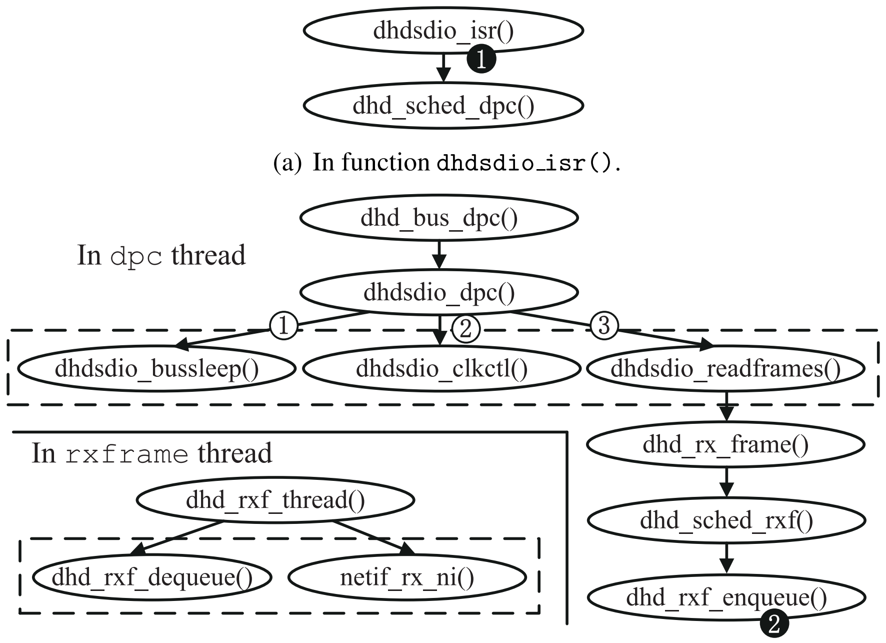
(a) In function `dhd_start_xmit()`.



(b) In `dpc` thread.

Figure 5.2: Key WNIC driver functions for packet sending.

For packet receiving, the `dpc` thread is also responsible for data processing. As shown in Figure 5.3(a), the interrupt handling function `dhdsdio_isr()` first register a task in the `dpc` thread. Similar to packet sending, the `dpc` thread also needs to check the status of SDIO bus and backplane clock (i.e., ① and ② in Figure 5.3(b), and then tries to receive frames from the bus with function `dhdsdio_readframes()`. After the frames are queued using `dhd_rxf_enqueue()`, another kernel thread `rxframe` dequeues the frames and invokes `netif_rx_ni()` to deliver the packets to the system.



(a) In function `dhdsdio_isr()`.

(b) In `dpc` and `rxframe` thread.

Figure 5.3: Key WNIC driver functions for packet receiving.

We enable the driver debug message by re-compiling the Android kernel. The kernel log information shows that the driver puts the SDIO bus into sleeping state frequently if the data transmission rate is not high. When there is a packet sending request or a packet arrival interrupt, it takes time for the driver to bring the bus up. We modify the source code by adding two timestamping points at the entrances of function `dhd_start_xmit()` and `dhdsdio_txpkt()` (i.e., ❶ and ❷ in Figure 5.2), so that we can measure the delay for the driver to send out a packet which is denoted by d_{vsend}. We use `printk()` to measure d_{vsend}, because Dtrace [9] is not available for ARM-based architecture. We re-compile the code and run the customized kernel in Nexus 5. To evaluate the effect of bus sleep, we also disable the sleep feature in `dhdsdio_bussleep()`. Table 5.3 presents the minimum, mean, and maximum values of d_{vsend} when sending out 100 ICMP packets with the two packet sending intervals (10ms and 1s). After disabling the bus sleep feature, d_{vsend} drops below 1ms regardless of the packet send rate. Otherwise, the mean value can be as high as 13ms when the packet sending interval is 1s.

Table 5.3: d_{vsend} measured by Nexus 5 with the SDIO bus sleep mode enabled or disabled (in ms). It can cost up to ∼14ms for the bus to wake up (promotion delay, T_{prom}.

Bus sleep	Packet interval	Min	Mean	Max
Enabled	10	0.096	0.321	10.184
	1000	0.139	10.151	13.547
Disabled	10	0.092	0.229	0.836
	1000	0.139	0.720	0.858

Similarly, we timestamp at the entrances of `dhdsdio_isr()` and `dhd_rxf_enqueue()` (i.e., ➊ and ➋ in Figure 5.3) to measure the delay for the driver to receive a packet, denoted by d_{vrecv}. Considering that packet receiving shares similar data path as for packet sending, we test only the case of disabling the SDIO bus sleep mode for Nexus 5. Table 5.4 shows that a large packet arrival interval will not result in large d_{vrecv} when the bus sleep mode is off. As result, we have verified that the SDIO bus sleep is the main component in d_{k-n}.

Table 5.4: d_{vrecv} measured by Nexus 5 with the SDIO bus sleep mode disabled (in ms).

Packet interval	Min	Mean	Max
10	0.311	1.589	2.651
1000	0.362	1.756	2.088

We also investigate the criteria that trigger the bus sleep mode. Our driver analysis shows that the driver maintains a counter $idlecount$. For every $dhd\_watchdog\_ms$, whose default value is 10ms, the driver increases this counter by 1 if the hardware is idle. When the counter reaches to a threshold $idletime$, whose default value is 5, the driver will instruct the bus to sleep. Therefore, the default idle period is 50ms. Both $idletime$ and $dhd\_watchdog\_ms$ are configurable when the driver is loaded. Our experiments also confirm that the idle period (T_i) for Nexus 5 is 50ms.

The "wcnss" driver used by Qualcomm WNIC chipsets shares similar mechanism, although the chipsets connect to the system via SMD interface instead of SDIO. To simplify our presentation, we refer also this energy-saving mechanism to SDIO bus sleep mode.

5.1.2.2 Effect of Power Save Mode

The PSM allows WNIC to switch from *active* state (a.k.a. Constantly Awake Mode, CAM) to *sleep* state (a.k.a. Power Save Mode, PSM) in order to reduce energy consumption and prolong battery lifetime. During PSM, the packets sent to the WiFi station (STA) are buffered by the access point (AP) and will not be delivered to the STA until the STA awakes. Based on the implementation strategy, PSM can be further classified into *static PSM* and *adaptive PSM*. In static PSM, the STA and AP agree upon a *listen interval*, which is the number of *beacon intervals* that the STA will ignore before turning on the receiver. Right before the end of the listen interval, the STA wakes up and listens for the beacon frame, checking the Traffic Indication Map (TIM). If there are packets buffered on the AP, it sends a PS-POLL message to the AP to request those packets. After receiving the packets, the STA switches back to PSM immediately. The adaptive PSM differs from static PSM in that the STA will stay in CAM for a pre-defined idle period (PSM timeout), preventing the STA from directly going to sleep during data transfer. To switch between the two states, the STA sends a NULL frame with the POWER field set to "enabled" or "disabled" to the AP.

As static PSM could lead to RTT round-up effect and degrade network performance [133], adaptive PSM are usually adopted in smartphones today [78, 183, 194]. We have also confirmed this for the five smartphones used in our experiments. However, adaptive PSM could still inflate the nRTT. Figure 5.4 illustrates two possible scenarios. Let d_p and d be the measured and actual network delay, respectively. In scenario (I) where d' is smaller than the PSM timeout threshold T_i, the STA is able to receive the response packet before entering PSM, resulting in no delay inflation (i.e., $d_p = d$). However, for $d > T_i$ in scenario (II), the STA has already turned off its receiver when the response packet arrives at the AP. Only after the STA listens to the beacon frame, can it change the state to CAM and receive the packet. As a result, d could be inflated by up to $I_B * (L + 1)$, where I_B is the *beacon interval* with a value of 100 TUs (Time Units, 1.024ms per TU), and L is the listen interval.

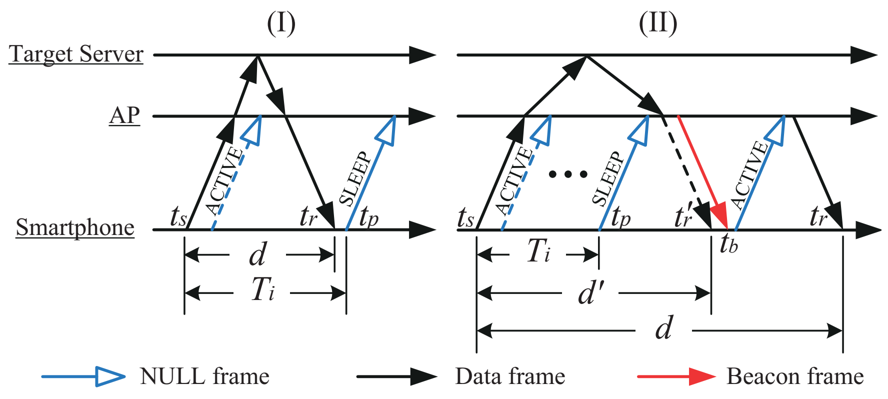


Figure 5.4: Two scenarios for adaptive PSM to inflate nRTT.

We measure the PSM timeout value by carefully sending out packets with increased packet sending interval. Table 5.5 summarizes the PSM timeout values for the five smartphones and shows that this value is smartphone-dependent. As an extreme case, Nexus 4 enters PSM in 40ms when the WNIC is idle. Therefore, there is a higher possibility for Nexus 4 to report an inaccurate result when measuring a network path longer than 40ms. Since the nRTT is set to 60ms in the last section, the PSM introduces a significant network delay to the Nexus 4 measurement. On the other hand, the values of the listen interval determines how much nRTT can be inflated by PSM. Although STA announces a default listen interval during the *association period* (1 for “wcnss” driver and 10 for “bcmhd” driver by default), we find that the smartphones do not adopt this default value in adaptive PSM. The actual listen intervals for the phones are all 0, which means that the length is 1 beacon cycle of 102.4ms. Thus, the adaptive PSM can inflate the nRTT by over 100ms.

Table 5.5: Timeout values and initial listen intervals of the smartphones under test.

Phone	PSM timeout (T_i)	Listen interval (Associated)	Listen interval (Actual)
Google Nexus 4	∼40ms	1	0
Google Nexus 5	∼205ms	10	0
Samsung Grand	∼45ms	10	0
HTC One	∼400ms	1	0
Sony Xperia J	∼210ms	10	0

5.1.2.3 Effect of low-power RRC states

Performing network measurement accurately in cellular networks is much more difficult than WiFi network due to the frequent RRC state transitions. The RRC protocol handles the control plane signalling between the smartphones (a.k.a. "user equipments") and the UMTS Terrestrial Radio Access Network (UTRAN), allowing the network to allocate radio resources and enable energy-efficient operation. Figure 5.5 shows the possible state machines for 3G (UMTS), 4G (LTE), and 2G (GPRS/EDGE) networks. The promotion delays and inactivity timers are also labeled in the figure.

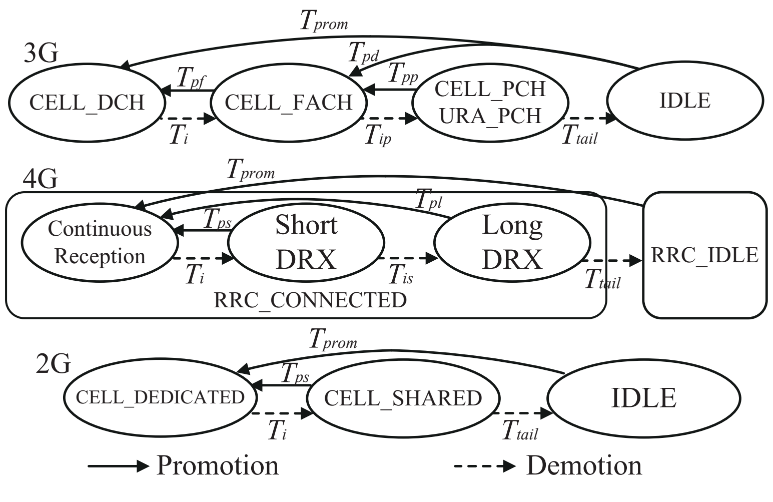


Figure 5.5: RRC state machine for 3G, 4G, and 2G network.

3G. When a smartphone is powered on, it enters `IDLE` state, in which there is no radio resource allocated and no data transfer is engaged. When an RRC connection is established, the phone can stay in one of the following states: `CELL_DCH` for the highest power consumption with highest throughput and lowest latency, `CELL_FACH` for a low capacity channel shared across all mobile users, and `CELL_PCH` or `URA_PCH` for only listening to the paging messages and no data transmission allowed. Rosen et al. have shown that when the phone is under low power state or during state transitions, it can lead to substantial, unexpected latencies [190].

4G. The RRC states in LTE network have been simplified to only `RRC_IDLE` and `RRC_CONNECTED` [44]. However, after being promoted from `RRC_IDLE` to `RRC_CONNECTED`, the smartphone is allowed to work in `Continuous Reception` mode or `Discontinuous Reception` (DRX) mode [43]. In the DRX mode, the smartphone turns off its transceiver to minimize the power consumption and only wakes up on the "On Duration" periodically, monitoring the *Physical Downlink Control Channel* (PDCCH). If there is any data scheduled for it, the smartphone transitions to the `Continuous Reception` mode to receive the data; otherwise, it goes back to sleep. LTE usually supports Short DRX and Long DRX. The smartphone first enters the Short DRX when there is no data activity and the *DRX inactivity timer* expires after T_i, and it further transits to Long DRX if the *DRX short cycle timer* expires after T_{is}. The difference between Short DRX and Long DRX is that the inactivity period for the Long DRX is much longer. Therefore, the smartphone can wake up more quickly but consume more energy in Short DRX.

2G. There also exists a similar RRC state machine model for 2G networks. To transmit data, the smartphone has to be promoted to the `CELL_DEDICATED` state. When there is no data activity, it first switches to `CELL_SHARED` before falling back to the `IDLE` state.

There are two scenarios that the RRC states can inflate the nRTT measurement. The first is when the smartphone performs network measurement when it is in a low-power state. It therefore has to wait for a long time (e.g., several hundreds of milliseconds to several seconds for 3G) for state promotion before sending and receiving measurement packets. The second scenario is that after the smartphone has sent out a probe packet, it could be demoted to a lower power state (e.g., Short DRX mode for LTE) if it cannot receive the response packet before the demotion timer expires. Therefore, when the response packet arrives, the smartphone has to wait for some time (up to a "On Duration" cycle) to receive it.

5.2 AcuteMon

Our delay overhead analysis in Section 5.1 shows that the energy-saving mechanisms could lead to significant delay inflations in both WiFi and cellular networks during the network measurement. Different from the additional delays introduced by base stations (e.g., the effect of bufferbloat [125,126] and queuing policy [230]), these delay overheads cannot be considered as part of the network characteristics, because they can be excluded if the smartphones are in the active states. Their unpredictability prevents users from understanding the actual network performance, as well as makes it difficult to build a uniform model to calibrate the network delays measured by the traditional network measurement apps. This is because the overheads are shown to be affected by several factors: *i*) the length of the network path; *ii*) the chipset adopted by the device and the factory default; and *iii*) the ISP's own configuration. Although factor *ii*) and *iii*) could be bypassed via building large database which stores the pre-collected default parameters for each phone and ISP, some of the parameters can be changed by the end user or the third-party customized ROM. Moreover, the network traffic introduced by other apps could change the state of the network interface card. Since the fine-grained packet-level behavior is infeasible to measurement apps without "rooting" the device, we cannot predict the behavior of other apps, even if we can model the delay overhead for a specific smartphone correctly. Therefore, calibrating the measurement results based on a pre-built model is not feasible in practice.

Another possible way to estimate the network delay is making use of network throughput measured by the apps with flooding-based techniques. In detail, the tools upload or download large files, and calculate the throughput based on the sending/receiving volume of traffic and the corresponding transmission time. In this case, the continuous packet sending or receiving can keep the NIC in active modes. Only the first several packets could be affected by the energy saving mechanisms. Therefore, the throughput is more accurate compared with network delay measurement, especially after excluding the first several packets. However, it is still infeasible to using the throughput models (such as [175]) to infer the network delay. The reasons could be as follows:

- The throughput reported by the tools is an average for a long period (could be tens of seconds). Using this value to infer the network delay can miss the network fluctuation events.

- The packet loss rate cannot be obtained through those measurement tools.

- The throughput models themselves are not so accurate, or they can be only applied to some special flows (e.g., TCP Reno flows).

As a result, it is not applicable to use the throughput to calculate the network delay.

Simply increasing the packet sending rate could be the easiest way to mitigate the delay overheads. However, it is not applicable for two reasons. First, some tools require root privilege when the sampling rate increases (e.g., `ping` cannot be executed without root if the probe sending interval is shorter than 200ms). Second, a more frequent packet sending incurs extra cost to users and may even self-congest the network path. Although the background traffic introduced by other apps may activate the smartphone, it is obvious that we cannot rely on these random events. Instead, we need to develop a reliable and cost-effective methodology to measure the actual network delay without being affected by the aforementioned power saving mechanisms.

In this section, we present the design and implementation of AcuteMon to mitigate the delay overhead in the WiFi and cellular network measurement without modifying the underlying system, kernel, and driver. AcuteMon is designed based on three assumptions during the measurement phase: *i*) the initial state of the smartphone is inactive as it is difficult to detect the status from the user level; *ii*) there is no background traffic, so that AcuteMon does not rely on the activities of any other apps; and *iii*) power consumption is not an issue. As a result, AcuteMon can fully remove the impact of the smartphone's energy-saving mechanisms and obtain accurate nRTT under any environment.

5.2.1 Implementation details

As illustrated in Figure 5.6, AcuteMon consists of two concurrent threads—background traffic thread (BT) and measurement thread (MT). The goal of the BT is to keep the smartphone in the wake-up or high-power state during nRTT measurement. The process starts with a *warm-up phase* where the BT sends warm-up packets to activate the phone and in the subsequent *measurement phase* it continuously sends lightweight background traffic to prevent any state transition.

The MT, on the other hand, sends K measurement probes to measure the nRTT between the phone and a target server. In the current version, AcuteMon uses TCP control messages (TCP SYN/ACK packets) and TCP data packets (HTTP request and response) to measure nRTT to any TCP servers. The implementation can be easily extended to UDP and ICMP packets. We implement the MT as a pre-compiled C binary (instead of running within the Android runtime) to mitigate the user-kernel delay

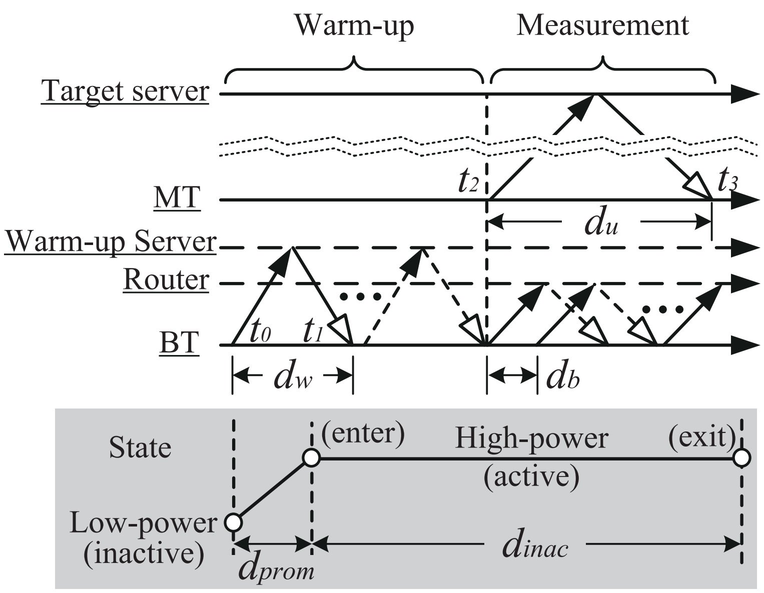


Figure 5.6: Measurement process of AcuteMon. The power-state diagram below the time-line diagram shows that the smartphone is always in a high-power state during the network measurement.

overheads [144]. The whole process is described in Algorithm 2.

5.2.1.1 Warm-up phase

In the warm-up phase, the BT is to ensure the smartphone to enter the active or high-power state (i.e., SDIO bus awake mode and CAM in WiFi, `CELL_DEDICATED` in 2G, `CELL_DCH` in 3G, and `Continuous Reception` in LTE) before the real measurement starts. Our approach is to send one or more appropriate packet(s) to a warm-up server (denoted by $dnsIP$), which triggers the smartphone to promote its state if it is not active. The warm-up process is implemented as a procedure `DoWarmUp()` (line 13-21). We use d_w to denote the RTT of the warm-up packet (i.e., $t_1 - t_0$) and d_{prom} to denote the warm-up time required by the smartphone to enter the high-power state. Because the initial state of a smartphone is unknown to AcuteMon, d_{prom} varies each time. We summarize the possible promotion delay in Table 5.6 for the different network types based on our previous analysis and the existing works. To allow some safe margin, d_{prom} should be set to a value larger than them. We then denote the maximum allowable time that the smartphone remains in the high-power RRC state without sending or receiving packets by d_{inac} which should be smaller than T_i, the timeout value for the inactivity timer (as summarized in Table 5.7).

Table 5.6: Possible promotion delay for entering into the active or high-power state.

Type	Initial State	Promotion Delays	Typical Values
WiFi	SDIO bus sleep mode	T_{prom}	14ms
4G	`RRC_IDLE`	T_{prom}	260ms [108]
3G	`IDLE`	T_{prom} or $T_{pd} + T_{pf}$	2s [184]
	`CELL_FACH`	T_{pf}	1.5s [184]
2G	`IDLE`	T_{prom}	500ms [184]
	`CELL_SHARED`	T_{ps}	500ms [184]

Table 5.7: Possible state demotion timer (T_i) for leaving the active or high-power state.

Type	State Transition	T_i
WiFi	SDIO bus awake $\rightarrow$ sleep mode	50ms
	CAM $\rightarrow$ PSM	40ms
4G	Continuous Reception $\rightarrow$ Short DRX	100ms [108]
3G	`CELL_DCH` $\rightarrow$ `CELL_FACH`	5s [184]
2G	`CELL_DEDICATED` $\rightarrow$ `CELL_SHARED`	1s [184]

The most simple way to determine whether the smartphone is ready for measurement is to check d_w. However, triggering a proper response packet is difficult (see the discussion in Section 5.2.2). AcuteMon therefore ignores the response packets by setting the TTL (time-to-live) value of the warm-up packets to 1, so that the packets will be dropped at the first-hop router. After the first warm-up packet is sent, AcuteMon assumes that the smartphone is promoted to the high-power state after d_{prom}, during which more warm-up packets are sent with an interval of d_{inac}. This can prevent possible state demotion, because some providers use a relatively small value for T_i. Therefore, totally m ($m = \lceil \frac{d_{prom}}{d_{inac}} \rceil$) packets are sent. After that, the measurement phase starts.

The empirical values of the four input parameters d_{prom}, d_{inac}, d_b, and L (bytes) adopted by AcuteMon are given in Table 5.8. d_{prom} is obtained from Table 5.6 and 5.7. The packet size L is set to 400 bytes, 100 bytes, and 1024 bytes for WiFi, 4G, and 2/3G networks, respectively. We employ a larger L for 2/3G network, because a large packet is required for 3G to exceed the RLC buffer threshold [190], whereas 4G does not have this requirement [108].

Table 5.8: The empirical values of the inputs to the algorithms adopted by AcuteMon.

Network Type	d_{prom}	d_{inac}	d_b	L (bytes)	m
WiFi	20ms	38ms	20ms	400	1
4G	280ms	90ms	45ms	100	4
3G	2.5s	3s	1.5s	1024	1
2G	1.5s	400ms	200ms	1024	4

5.2.1.2 Measurement phase

In the measurement phase, according to `AcuteProbe()` (line 22-29), the MT sends out K probe packets to the target $destIP$, and each nRTT is denoted by d_u. At the same time, the BT sends L-byte background packets to $dnsIP$ periodically with an inter-packet interval of d_b. Similar to warm-up packets, the TTL value of background packets is also set to 1 to restrict the impact of extra packets to the closest hop. With a proper choice of d_b, the background traffic can reset the state demotion timers to avoid any state change. Here we assign a much safer d_b, which is set to be half or close half of d_{inac}. Our evaluation in Section 5.3 shows that the empirical values adopted by AcuteMon work effectively. For example, d_b of 20ms in WiFi network is also appropriate for the smartphones employing "wcnss" driver. `AcuteProbe()` finally returns the average of K d_us.

5.2.2 Speeding up warm-up phase for cellular networks

According to Table 5.8, AcuteMon has to wait a long period before a measurement can start in cellular networks (e.g., 2.5s for 3G networks), which is inefficient, because the smartphone may stay in the middle, or even active, states during the warm-up. To speed up the warm-up phase, we utilize the first response packet, because a large enough response can verify that the smartphone is in the high-power state. Therefore, we try to elicit a large response from the warm-up server, instead of dropping the warm-up packets at the first hop. The most challenging part in this attempt is to choose an appropriate echo server for the warm-up packets. There are several requirements for this server. First, the warm-up packet is able to always elicit a response with enough size (i.e., larger than the RLC buffer threshold). Second, AcuteMon must be able to verify the reception of the response packet by measuring d_w. A suitable candidate to meet the requirements is the default DNS server for the phone, which is usually assigned by the network provider. Compared to other types of servers, the default DNS server is more reliable (e.g., a Web server could be unaccessible in some ISPs, and an ICMP request could not always trigger a response). Moreover, our experiments show that the DNS server is often close to the smartphone (within 5 hops and RTT below 30ms), so that the warm-up phase will not last too long.

However, constructing a large DNS query packet (1024 bytes) to induce a response for 2/3G networks is not straightforward. The main reason is that the maximum length of a DNS payload is only 512 bytes [160], and an over-length query may be ignored by some DNS servers or routers. Although some DNS servers answer invalid DNS query, the response size could be limited to around 560 bytes, which could not meet the RLC buffer threshold for some ISPs. To address this problem, AcuteMon sends an over-length DNS query packet followed immediately by a normal one. Therefore, even the DNS server drops the first query, the second query can still trigger a response.

Considering the response size, we need to further check d_w: a smaller d_w compared to d_{inac} ($d_w < d_{inac}$) means that the smartphone has not been demoted to a lower power state. The warm-up process for 4G network is much simpler because of no limitation on the RLC buffer threshold. AcuteMon can therefore measure d_w with normal DNS query/response.

In the new efficient warm-up process (`QuickWarmUp()`, line 9-12), up to three warm-up requests are sent to $dnsIP$. If `QuickWarmUp()` succeeds, AcuteMon starts the measurement by executing `AcuteProbe()`. Otherwise, it falls back to the normal warm-up process (`DoWarmUp()`).

5.2.3 Additional cost

The additional packets consumed by AcuteMon in the warm-up phase is very light, because only up to two warm-up packets are sent to promote the smartphone's state. Even if `QuickWarmUp()` fails, AcuteMon still sends four more packets for 4G and 2G networks. AcuteMon also works effectively in the measurement phase. The background packets are dropped in the first hop and will not burden the remaining part of a network path. For 2/3G networks, the large d_bs results in very few extra packets. Although d_b is relatively small in WiFi network, the first-hop router usually has a high data rate and is not the bottleneck of a network path, because it is often the gateway of a home or a campus. For example, supposing that AcuteMon sends five probe packets to measure a path with nRTT of 100ms, the BT will send around 25 additional packets to the gateway for the nRTT measurement. Therefore, we believe that the impact of the background traffic on the measurement results is negligible.

Algorithm 2 Accurate RTT probing

```
Input: $destIP$, $dnsIP$, $d_{prom}$, $d_{inac}$, $d_b$, $L$
Output: $t_u$
 1: Determine the network type $netType$
 2: if $netType =$ cellular then
 3:     if not QUICKWARMUP() then
 4:         DOWARMUP()
 5:     end if
 6: else
 7:     DOWARMUP()
 8: end if
 9: ACUTEPROBE()
10: procedure QUICKWARMUP()
11:     $m \leftarrow \left\lceil \frac{d_{prom}}{d_{inac}} \right\rceil$
12:     Send $m$ $L$-byte packets to $dnsIP$ with interval $d_{inac}$ and TTL 1
13: end procedure
14: procedure DOWARMUP()
15:     for $n = 0$ to 2 do
16:         Send a $L$-byte packet to $dnsIP$ and receive the response packet
17:         Compute $d_w$
18:         if $d_w < d_{inac}$ then
19:             return TRUE
20:         end if
21:     end for
22: end procedure
23: procedure ACUTEPROBE()
24:     Send $L$-byte packets to $dnsIP$ with interval $d_b$ and TTL 1
25:     for $i = 1$ to $K$ do
26:         Send a probe packet to $destIP$ and receive the response packet
27:         Compute $d_u$
28:     end for
29:     return the average of $d_u$
30: end procedure
```

5.3 Evaluation

In this section, we evaluate the performance of AcuteMon in testbed for WiFi network and in the wild for cellular networks. The evaluation results show that AcuteMon can effectively mitigate the energy-saving effects.

5.3.1 Testbed evaluation

We evaluate the performance of AcuteMon's WiFi implementation in the testbed described in Section 4.1.3. For each test, we run AcuteMon on the smartphone to measure the nRTT between the phone and the measurement server by sending out 100 probes ($K = 100$). We introduce additional delay on the server side to emulate four different nRTTs: 20ms, 50ms, 85ms, and 135ms.

5.3.1.1 Actual network RTT

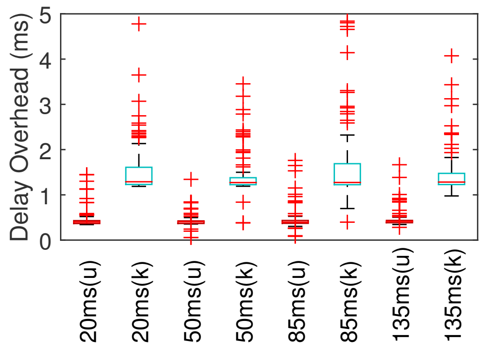
Table 5.9 presents the means and 95% confidence intervals of the actual nRTTs (d_n) measured by the external sniffers. For all five smartphones, d_ns are very close to their emulated values. In fact, no significant nRTT inflation can be observed, and most of the deviations are kept within 3ms, implying that the measurement packets have not been delayed at the AP. Our further analysis of the raw `pcap` files also confirms that no PSM activity can be detected when the smartphone receives response packets. Compared with the results shown in Table 5.2, AcuteMon successfully prevents the smartphones from entering PSM.

Table 5.9: The actual nRTTs (d_n) measured by external sniffers (mean with 95% confidence interval, in ms).

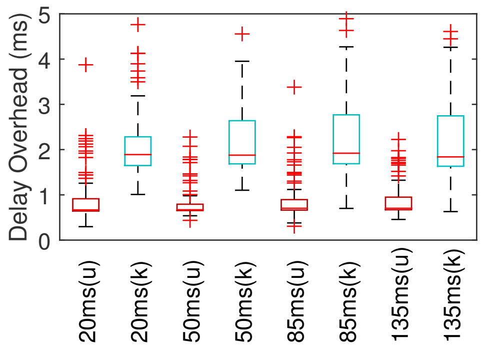
Phone	Emulated RTT (ms)			
	20	50	85	135
Google Nexus 5	22.461 ± 0.545	51.683 ± 0.168	87.198 ± 0.387	137.090 ± 0.320
Sony Xperia J	21.584 ± 0.184	51.597 ± 0.149	86.868 ± 0.275	136.79 ± 0.178
Samsung Grand	22.020 ± 0.382	52.614 ± 0.485	86.675 ± 0.177	137.0 ± 0.217
Google Nexus 4	21.680 ± 0.181	51.673 ± 0.202	86.888 ± 0.358	137.98 ± 1.101
HTC One	21.874 ± 0.200	51.786 ± 0.198	86.810 ± 0.192	136.850 ± 0.154

5.3.1.2 Delay overheads

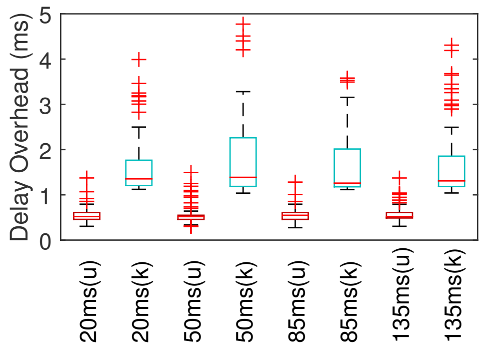
Next we analyze the measurement accuracy of AcuteMon in terms of delay overhead. We use box plots to present Δd_{u-k} and Δd_{k-n} introduced by AcuteMon in Figure 5.7. On the x-axis, we use (*u*) and (*k*) after the RTT to denote Δd_{u-k} and Δd_{k-n}, respectively.



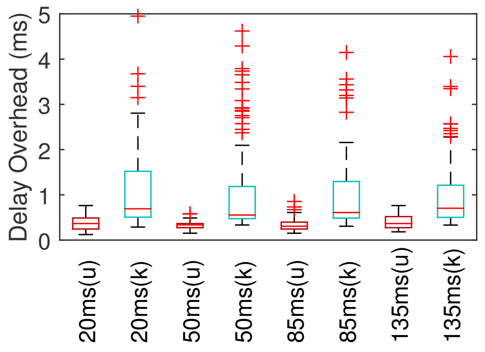
(a) Google Nexus 5.



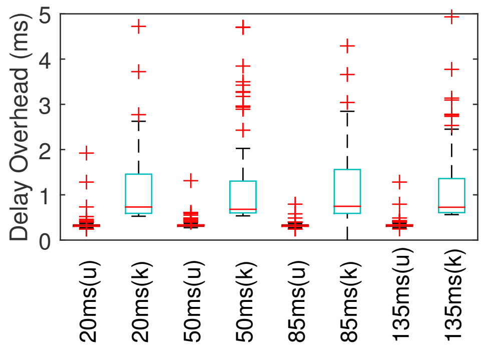
(b) Sony Xperia J.



(c) Samsung Grand.



(d) Google Nexus 4.



(e) HTC One (W802).

Figure 5.7: Box plots of Δd_{u-k} and Δd_{k-n} obtained by AcuteMon.

The previous study [144] has shown that executing the measurement logic as a native Linux program can mitigate the delay overheads caused in the DVM. The measurement results support this claim. We observe very small Δd_{u-k} for all the phones, most of which are smaller than 0.5ms. Even for the two smartphones with relatively low hardware configurations (i.e., Sony Xperia J and Samsung Grand), their Δd_{u-k}s are smaller than 1ms.

On the other hand, Δd_{k-n} accounts for the majority of the delay overhead. Although Δd_{k-n} is much larger than Δd_{u-k}, their medians are all less than 2ms, and the upper bounds are still less than 3ms (except for Sony Xperia J, whose upper bounds can be 4ms). For the smartphones equipped with Qualcomm's WNIC chipsets (Google Nexus 4 and HTC One), Figures 5.7(d) and 5.7(e) show that their medians of Δd_{k-n} can be as small as $\sim$0.8ms. As a result, the overall delay overheads are kept within 4-5ms.

Another important observation is that the delay overheads for AcuteMon are independent of nRTTs, and the values of the overheads are much more stable. Therefore, the true value can be obtained by performing calibration.

5.3.2 Internet evaluation

Since it is not possible to obtain d_n for cellular network in a testbed, we compare the performance of AcuteMon with Ookla Speedtest and MobiPerf in real users' phones for their popularity. As discussed in Section 2.2.3, Speedtest samples network paths with TCP data packets (HTTP GET messages) for 6 times, whereas MobiPerf supports ICMP (with ping program), TCP SYN/RST (on port 7), and TCP SYN/ACK (on port 80). MobiPerf sends 10 sampling probes and reports the mean values. We denote these four reference methods by SP (Speedtest), MPI (MobiPerf ICMP), MPS (MobiPerf TCP SYN/RST), and MPH (MobiPerf TCP SYN/ACK). For AcuteMon, it supports HTTP GET messages, denoted by AMH, and TCP SYN/ACK packets, denoted by

AMS.

We conduct the Internet experiments through a test app, which is pre-configured with 6 sub-tests. Each sub-test conducts measurement using one of the six aforementioned methods. We randomly choose 10 Web servers obtained from the Speedtest server list. During the experiments, the app randomly selects one of the servers as the measurement target and performs the six sub-tests one after the other. In order to mitigate the potential impact of network churn, the execution sequence of the six sub-tests is randomized. A 4-second sleep period is also inserted between adjacent sub-tests to allow the smartphone to return to a low-power RRC state. The entire set of experiments can be completed within one minute. We believe that the Internet path quality is mostly stationary for such a short time period [235]. Volunteers were recruited to perform the experiments with their smartphones. Besides the measurement results, the test app also collects the phone model, OS version, ISP, network types, IP address, geo-location information, and signal strength.

5.3.2.1 The overall results

From November 13, 2015 to December 3, 2015, our Internet experiments have been conducted on 28 smartphones, and we have collected 289 sets of measurement results. These smartphones, which are of 26 different models, run on 11 Android versions (ranged from version 4.0 to 5.1), and connect with 9 ISPs (one in Singapore, two in China, and six in Hong Kong). We observe that sometimes there is change in network types during the experiments. For example, the network type switches from WiFi to cellular network, or 3G to 4G. In our analysis, we consider only the results with consistent network type throughout all the six sub-tests as valid results. We have totally obtained 252 sets of valid results, of which 146 sets were performed in cellular networks.

We first analyze the number of cases that the nRTTs obtained by AMH and AMS are less than the other four reference methods. Let n be the total number of valid result sets, and n_M the number of them that the RTT reported by AMH or AMS is less than reference method M, where $M \in \{$SP, MPH, MPS, MPI$\}$. Hence, we can compute the percentage of results that AcuteMon gives lower nRTTs by

$$P_M = n_M/n \times 100\%. \tag{5.1}$$

Table 5.10 presents P_M for AMH and AMS in 2/3/4G networks. For MPS and MPI, we include the number of valid results used for computing P_M inside parentheses after P_M, because some Web servers do not support these measurement methods. More than 88%, 63%, 82%, and 80% of the AMH (TCP data packets) measurement results are less than SP, MPH, MPS, and MPI, respectively. The use of AMS (TCP control messages), P_M is further increased to more than 97%, 85%, 95%, and 89%. In more details, P_M for AMH can reach 70%-80% for both 3G and 4G networks, except the case of MPH in 3G (53.8%). For AMS, P_M is higher than 90% for most of the cases. Even for the worse case of MPH in 3G network, p_M is still over 75%.

Table 5.10: P_M for AMH and AMS in cellular networks.

Net. type	# of results	P_M for AMH				P_M for AMS			
		SP	MPH	MPS	MPI	SP	MPH	MPS	MPI
4G	47	85.1	72.3	83.9 (31)	87.8 (41)	95.7	95.7	90.3 (31)	92.7 (41)
3G	65	84.6	53.8	87.0 (46)	78.7 (61)	96.9	75.4	95.7 (46)	86.9 (61)
2G	34	100	70.6	70.8 (24)	73.5	100	91.2	100 (24)	91.2
Total	146	88.4	63.7	82.2 (101)	80.1 (136)	97.3	85.6	95.0 (101)	89.7 (136)

We employ another ratio R_M to measure how much the nRTTs obtained by method

M deviate from AcuteMon:

$$R_M = \frac{1}{n} \times \sum_{i=1}^{n} \frac{nRTT_{Mi}}{nRTT_{Ai}}, \tag{5.2}$$

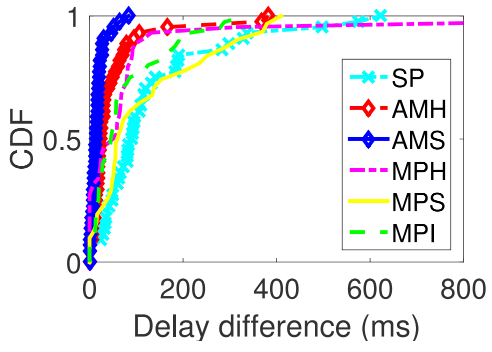
where $nRTT_{Mi}$ and $nRTT_{Ai}$ are the ith measurement result obtained by method M and AcuteMon, respectively. As shown in Table 5.11, the nRTTs measured by SP are extremely high in the 3/4G network. They can be 5 and 11 times more than AMH, and 12 and 27 times than AMS, respectively. The three MobiPerf methods perform better than SP, but their R_Ms are still 2 to 7 times more than AMH and AMS. Moreover, all the four reference methods perform better in the 2G networks than the 3/4G networks. Their R_Ms range from 1.2 to 2.4, and SP is still the worst performer. This is because the nRTTs in 2G network are usually larger than 3/4G networks. As a result, the contribution of the delay overhead to the overall delay becomes less significant. Moreover, the short promotion delay in 2G (compared to 3G) could be another reason for a lower R_M in 2G networks.

Table 5.11: R_M for AMH and AMS in cellular networks.

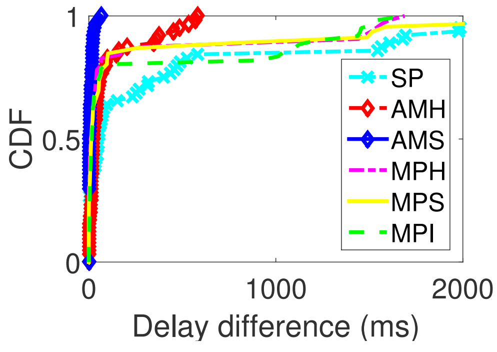
Network type	R_M for AMH				R_M for AMS			
	SP	MPH	MPS	MPI	SP	MPH	MPS	MPI
4G	11.60	7.00	2.54	3.20	27.5	1.74	3.35	3.50
3G	5.01	2.27	5.77	6.96	12.19	2.73	4.93	5.56
2G	1.95	1.19	1.62	1.47	2.40	1.49	1.99	1.91

5.3.2.2 Effect of the warm-up phase

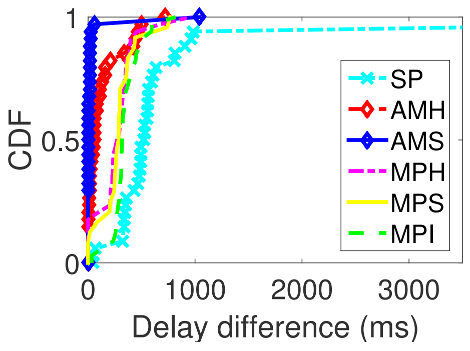
We investigate whether the measurement results are affected by the RRC state promotion. To this end, we compute the delay difference, denoted by $\Delta nRTT$, between the nRTT measured by the *first* probe packet in the MT and the minimal nRTT of all probes packets: $\Delta nRTT = nRTT_1 - \min\{nRTT_j\}$, where $j \in \{1, ..., m\}$ and m is the number of probes. Moreover, SP always includes the TCP connection establishment time for the first probe[2]. We therefore remove it by $nRTT_1 - \min\{nRTT_j\} \times 2$. A large $\Delta nRTT$ means that the first probe packet is delayed by the process of promoting the RRC state to a high-power state.



(a) 4G.



(b) 3G.



(c) 2G.

Figure 5.8: CDF plots of the delay difference $\Delta nRTT$.

Figure 5.8 plots the CDF of $\Delta nRTT$s. It shows that AMH and AMS hardly experience any state promotion delay, because their $\Delta nRTT$s are very close to zero. With the help of the warm-up phase, ∼90% and ∼95% of $\Delta nRTT$s in AMS are less than 30ms in the 4G and 3G network, respectively. For the 2G cases, although there is one large $\Delta nRTT$ (>1000ms), 95% of the remaining samples are less than 25ms. Moreover, AMH has relatively large $\Delta nRTT$s, but the values are still much less than d_{prom} for most of the cases. In contrast, the four reference methods have very large $\Delta nRTT$s with 5% to 15% of them higher than d_{prom}. Therefore, the warm-up phase in AcuteMon can help mitigate the effect of the state promotion delay.

[2]This is based on the design of Ookla Speedtest.

5.3.2.3 Effect of the background traffic

We next evaluate whether the measurement results are affected by state demotion. We still employ the metric R_M and Eqn. (5.2) to measure how method M differs from AcuteMon. We re-calculate $nRTT_{Mi}$ and $nRTT_{Ai}$ by removing the results obtained by the first probe. We use $nRTT'_{Mi}$ and $nRTT'_{Ai}$ to denote the re-calculated nRTTs, and denote the mean ratio as R'_M. In this way, the possible promotion delay caused by the first probe can be eliminated for the four reference methods. As summarized in Table 5.12, AMH and AMS report lower nRTTs than the other four methods for all network types. Compared to R_Ms presented in Table 5.11, R'_Ms for the four methods are decreased to between 1.15 and 4.72. Note that both Speedtest and MobiPerf do not consider the packet sending interval. Therefore, they can send a new probe packet only after receiving the response packet triggered by the previous probe. If the measured network path has a large nRTT, the smartphone will have a high probability to demote its RRC state, thus delaying the recipient of the response packet. AcuteMon, on the other hand, sends periodic background traffic to prevent state demotion during the delay measurement.

Table 5.12: R'_M for AMH and AMS in cellular networks.

Network	R'_M for AMH				R'_M for AMS			
type	SP	MPH	MPS	MPI	SP	MPH	MPS	MPI
4G	1.24	1.37	2.85	3.29	1.58	1.53	3.13	3.61
3G	1.42	1.65	3.25	3.44	2.04	2.24	4.06	4.72
2G	1.29	1.15	1.55	1.41	1.58	1.39	1.85	1.78

To sum up, our Internet experiment results support the finding that AcuteMon can achieve much lower nRTTs than Ookla Speedtest and MobiPerf, especially when TCP control messages are employed as measurement probes. Our warm-up and background traffic can effectively mitigate the effect of RRC state promotion and demotion during the delay measurement.

5.4 Summary

In this chapter we considered the problem of measuring network RTT from smartphones. We presented AcuteMon, a network measurement app in Android, to mitigate all major sources of delay inflation for WiFi and cellular networks. We first reported and demonstrated that various energy-saving mechanisms employed in these mobile networks and smartphones are the main sources of the delay inflation, including SDIO bus sleeping, IEEE 802.11 PSM, and RRC state transitions. Based on our driver code analysis and empirical evaluations, we proposed to keep the phone in the wake-up/high-power state during the delay measurement through a carefully timed sending of warm-up and periodic background traffic. We validated AcuteMon in a test bed for WiFi networks and performed experiments in real users' phones to evaluate the approach for cellular networks. Our experiment results show that AcuteMon can effectively mitigate the delay overheads and achieve very accurate network RTT.

Chapter 6

Conclusions and Future Work

The goal of this thesis is to appraise and improve the accuracy of smartphone-based and browser-based delay measurement. To quantify the accuracy of delay measurement, we defined “delay overhead” as the difference between the value measured by the tools and the actual network delay. We conducted an in-depth research to investigate the core methods employed by those measurement tools. For each method, we implemented a test page or an app so that we can evaluate its performance. Through controlled testbed experiments and Internet experiments, we found that each method exhibits different degrees of delay inflation. We then performed root-cause analysis to explain where and how the network delays were inflated. Based on our analysis, we further proposed a novel and effective method to mitigate the delay overheads in smartphone-based measurements. The testbed evaluation and Internet experiment indicated that our method can produce more accurate measurement results than the existing tools.

For browser-based measurements, we studied the impact of application-level delay overheads on a set of browser techniques which have already been, or can potentially be, applied to network delay measurements. Our study covered the traditional HTTP-based and socket-based techniques, such as XHR, WebSocket, Flash, and Java applet, as well as the more recent ones, such as Navigation Timing and WebRTC. We examined the delay overheads introduced by these methods with the major browsers on Windows and Ubuntu. With 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. In particular, we revealed that the socket-based methods incur much lower delay overhead than the HTTP-based methods in general. On the other hand, the Navigation Timing and WebRTC methods will also inflate the RTT measurement, especially for the measurement performed immediately after the the connection or data channel establishment. However, if this round of measurement is excluded, these two methods can achieve relatively consistent delay overheads, meaning that the results can be more easily calibrated. Considering the facts that i) Navigation Timing is supported by browsers natively without the need of installing third-party plug-ins, and ii) no additional socket server needs to be deployed compared to WebSocket, it is a good replacement for the traditional HTTP-based and socket-based methods. As for WebRTC, it is the only choice to conduct client-to-client measurements in browsers.

For smartphone-based measurements, we appraised the accuracy of measurement apps in Android phones. We first overcame the main challenge of obtaining actual network delays from the wireless medium by setting up a reliable wireless testbed. We then performed Internet experiment and testbed evaluation. In the Internet experiments, two popular apps, Ookla Speedtest and MobiPerf, have been tested. While in the testbed experiments, we built three test apps, each of which implemented one of the popular measurement methods. Both Internet experiments and testbed evaluation showed that the RTTs measured by the apps are significantly inflated. Moreover, the delay overhead introduced by the runtime virtual machine is significant and asymmetric in the send and receive directions. Our further multi-layer analysis showed that the long path of subfunction invocations in runtime accounts for the overhead in the user space. Through driver code analysis and empirical evaluations, we also demonstrated that the various energy-saving mechanisms employed by mobile networks and smartphones, including SDIO bus sleeping, IEEE 802.11 PSM, and RRC state transitions, caused the kernel-phy and external delay inflation.

We next considered the problem of improving the delay measurement accuracy for smartphone-based measurements. For the delay overhead in the user space, we proposed a method to mitigate it by implementing the measurement function into a native binary. Our evaluation showed that the new method can reduce its user-kernel delay overhead to less than 1.5ms. For the delay overhead caused by the energy-saving machanisms, we proposed to keep the smartphone in the wake-up/high-power state during the delay measurement. We finally presented AcuteMon, a network measurement app in Android, to mitigate all major sources of delay inflation for both WiFi and cellular networks. Through a carefully timed sending of warm-up and periodic background traffic, we could prevent the additional delay introduced by the state transition without rooting the smartphone or modifying the kernel and driver. We validated AcuteMon in a test bed for WiFi networks and performed experiments in real users' phones to evaluate the approach for cellular networks. Our experiment results showed that AcuteMon can effectively mitigate the delay overheads and achieved very accurate network RTT.

6.1 Future work

There are a few directions to extend tor improve he works performed in this thesis.

1. This thesis focuses only on the accuracy problem in smartphone-based and browser-based delay measurements. However, accurately measuring the network throughput (available bandwidth) is also very important. In fact, most network performance benchmarking tools support the symmetric network throughput mea-

surements. Although there are a lot of works to evaluate or compare the accuracy of the existing throughput measurement tools [96–98, 187, 210, 215], few of them pay attention to the benchmarking tools. When measuring the throughput, the benchmarking tools usually employ *flooding-based probing* method, which floods the access link in one or more parallel connections. Since the network layer information is invisible to them, they cannot implement *optimized-probing* methods, such as the Probe Gap Model [106, 215] and the Probe Rate Model [99, 121, 187, 211]. The estimation methods used by these tools are very simple. They just divide the total transferred bytes by the time duration. In particular, Ookla Speedtest aggregates the download into 20 slices, and ignore the fastest 10% and slowest 30% of the slices in calculation [57]. However, whether these methods are accurate and effective is still unknown to us. In the context of throughput measurement, we will evaluate the accuracy and efficiency (e.g., the measurement cost) of these tools. Another possible direction is trying to employ other more advanced methods, e.g., QUIC [27, 101].

2. We will implement a unified solution for accurate browser-based delay measurement. Although we have tested and compared twelve methods in the thesis, we only provide suggestions or recommendations on how to avoid the most significant parts of delay inflations, such as the best timestamping function, and excluding the delay of TCP connection or data channel establishment. Moreover, we have not provided applicable calibration methods. For this reason, we will try to reduce the performance gap among different browser and OSes, as well as apply statistical analysis to effectively calibrate the measurement results. We will also build a unified platform which enables both client-to-server and client-to-client measurements within the browsers.

3. The scale of our Internet evaluation for AcuteMon was small, which covers a

small portion of smartphone models and ISPs. Since our design is based on empirical parameters, we need to collect more samples and build a database to determine the best configuration. Another possible workaround is to introduce a more intelligent method to determine the optimal values by training the program after installation. Besides the parameter selection, our next step is to perform large-scale mobile broadband measurement through crowdsourcing. We are planning to combine MopEye [228], another tool that enables per-app network performance monitoring, to attract more users and provide more useful information.

Bibliography

[1] http://tinyurl.com/MarsRTTCode/.

[2] http://tinyurl.com/MarsFig/.

[3] http://tinyurl.com/MarsSDTCode/.

[4] ART and Dalvik. https://source.android.com/devices/tech/dalvik/index.html.

[5] Can I use High Resolution Time API? http://caniuse.com/#feat=high-resolution-time.

[6] Can I use Navigation Timing API? - Compatibility table for support of Navigation Timing API in desktop and mobile browsers. http://caniuse.com/nav-timing.

[7] Can I use WebRTC peer-to-peer connections? http://caniuse.com/#feat=rtcpeerconnection.

[8] Cedexis Radar. http://www.cedexis.com/radar/index.html.

[9] Dtrace for Linux on Github. https://github.com/dtrace4linux/linux.

[10] Facebook/network-connection-class at Github. https://github.com/facebook/network-connection-class.

[11] FCC Speed Test. https://play.google.com/store/apps/details?id=com.samknows.fcc.

[12] Internet Speed Test 3G, 4G, Wifi on Google Play. https://play.google.com/store/apps/details?id=uk.co.broadbandspeedchecker.

[13] Internet Speed Test on Google Play. https://play.google.com/store/apps/details?id=pl.speedtest.android.

[14] IP Performance Metrics (ippm). http://datatracker.ietf.org/wg/ippm/.

[15] Iperf. http://sourceforge.net/projects/iperf.

[16] Mobile Speed Test.com. http://www.mobilespeedtest.com/.

[17] MobiPerf on Google Play. https://play.google.com/store/apps/details?id=com.mobiperf.

[18] Netalyzr on Google Play. https://play.google.com/store/apps/details?id=edu.berkeley.icsi.netalyzr.android.

[19] Netperf. http://www.netperf.org/.

[20] Network Connection Class. https://code.facebook.com/projects/1547113495553528/network-connection-class/.

[21] Network monitoring tools. http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html.

[22] Network Speed Test on Windows Store. http://www.windowsphone.com/en-us/store/app/network-speed-test/9b9ae06b-2961-41ef-987d-b09567cffe70.

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

[24] PeerJS - Simple peer-to-peer with WebRTC. http://peerjs.com/.

[25] Platform versions, Dashboards — Android Developers. https://source.android.com/devices/tech/dalvik/index.html.

[26] Profiling with Traceview and dmtracedump. http://developer.android.com/tools/debugging/debugging-tracing.html.

[27] QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic.

[28] Rtcweb status pages. https://tools.ietf.org/wg/rtcweb/.

[29] SamKnows. https://www.samknows.com/.

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

[31] SpeedOf.Me Lite. http://speedof.me/m/.

[32] Speedtest X HD WiFi & Mobile Speed Test on App Store. https://itunes.apple.com/us/app/speedtest-x-hd-wifi-mobile/id366593092.

[33] Speedtest.net on App Store. https://itunes.apple.com/us/app/speedtest.net-mobile-speed/id300704847.

[34] Speedtest.net on Google Play. https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest.

[35] Speedtest.net on Windows Store. http://www.windowsphone.com/en-us/store/app/speedtest-net/4fcd4de1-050b-44dc-b123-a786808eb49b.

[36] strace. http://strace.sourceforge.net/.

[37] Tencent Mars at Github. https://github.com/Tencent/mars/.

[38] The Internet Engineering Task Force (IETF). http://www.ietf.org/.

[39] University of Oregon Route Views project. http://www.routeviews.org/.

[40] W3C Web Performance Working Group. http://www.w3.org/2010/webperf/.

[41] World Internet users statistics and 2016 world population stats. http://www.internetworldstats.com/stats.htm.

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

[43] 3GPP TS 36.321: Evolved Universal Terrestrial Radio Access (E-UTRA); Medium Access Control (MAC) protocol specification (V12.7.0), 2015.

[44] 3GPP TS 36.331: Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification (V12.7.0), 2015.

[45] Adobe. Statistics: PC penetration. http://www.adobe.com/hk_en/products/flashplatformruntimes/statistics.html, 2011.

[46] V. Aggarwal, E. Halepovic, J. Pang, S. Venkataraman, and H. Yan. Prometheus: Toward quality-of-experience estimation for mobile apps from passive network measurements. In *Proc. HotMobile*, HotMobile '14, pages 18:1–18:6, New York, NY, USA, 2014. ACM.

[47] M. Aida, N. Miyoshi, and K. Ishibashi. A scalable and lightweight QoS monitoring technique combining passive and active approaches. In *Proc. IEEE INFOCOM*, volume 1, pages 125–133 vol.1, March 2003.

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

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

[50] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. ACM SOSP*, 2001.

[51] D. Antoniades, M. Athanatos, A. Papadogiannakis, E. Markatos, and C. Dovrolis. Available bandwidth measurement as simple as running wget. In *Proc. PAM*, 2006.

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

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

[54] F. Baccelli, S. Machiraju, D. Veitch, and J. C. Bolot. The role of PASTA in network measurement. In *Proc. ACM SIGCOMM*, 2006.

[55] M. D. Bailey. *A Scalable Hybrid Network Monitoring Architecture for Measuring, Characterizing, and Tracking Internet Threat Dynamics*. PhD thesis, Ann Arbor, MI, USA, 2006.

[56] L. Batyuk, A.-D. Schmidt, H.-G. Schmidt, A. Camtepe, and S. Albayrak. Developing and benchmarking native Linux applications on Android. In *Proc. Mobilware*, 2009.

[57] S. Bauer, D. D. Clark, and W. Lehr. Understanding broadband speed measurements. Technical Report TPRC 2010, Massachusetts Institute of Technology, August 2010.

[58] Y. Bejerano, J. Ferragut, K. Guo, V. Gupta, C. Gutterman, T. Nandagopal, and G. Zussman. Scalable WiFi multicast services for very large groups. In *Proc. ICNP*, 2013.

[59] J. Bellardo and S. Savage. Measuring packet reordering. In *Proc. ACM SIGCOMM IMW*, 2002.

[60] R. Beverly, W. Brinkmeyer, M. Luckie, and J. P. Rohrer. IPv6 alias resolution via induced fragmentation. In *Proc. PAM*, 2013.

[61] C. J. Bovy, H. T. Mertodimedjo, G. Hooghiemstra, H. Uijtervaal, and P. V. Mieghem. Analysis of end-to-end delay measurements in Internet. In *Proc. PAM*, 2002.

[62] J. But, U. Keller, D. Kennedy, and G. Armitage. Passive TCP stream estimation of RTT and jitter parameters. In *Proc. IEEE LCN*, 2005.

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

[64] 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.

[65] E. W. W. Chan, X. Luo, and R. K. C. Chang. A minimum-delay-difference method for mitigating cross-traffic impact on capacity measurement. In *Proc. ACM CoNEXT*, 2009.

[66] M. C. Chan, Y.-J. Lin, and X. Wang. A scalable monitoring approach for service level agreements validation. In *Proc. IEEE ICNP*, pages 37–48, 2000.

[67] B. Chandrasekaran, G. Smaragdakis, A. Berger, M. Luckie, and K. Ng. A server-to-server view of the Internet. In *Proc. ACM CoNEXT*, 2015.

[68] K. Chen, Y. Xue, S. H. Shah, and K. Nahrstedt. Understanding bandwidth-delay product in mobile ad hoc networks. *Computer Communications*, 27(10):923 – 934, 2004. Protocol Engineering for Wired and Wireless Networks.

[69] X. Chen, R. Jin, K. Suh, B. Wang, and W. Wei. Network performance of smart mobile handhelds in a university campus WiFi network. In *Proc. ACM/USENIX IMC*, 2012.

[70] P. Chimento and J. Ishac. Defining network capacity. RFC 5136, IETF, Dec. 2002.

[71] L. Ciavattone, A. Morton, and G. Ramachandran. Standardized active measurements on a tier 1 IP backbone. *IEEE Communications Magazine*, 41(6):90–97, June 2003.

[72] K. Claffy. Internet measurement and data analysis: topology, workload, performance and routing statistics. In *Proc. National Academy of Engineering (NAE) Workshop*, 1999.

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

[74] L. Colittia, G. D. Battista, M. Patrignania, M. Pizzonia, and M. Rimondini. Investigating prefix propagation through active BGP probing. *Microprocessors and Microsystems*, 31(7):460–474, 2007.

[75] C. Demichelis and P. Chimento. IP packet delay variation metric for IP performance metrics (IPPM). RFC 3393, IETF, Nov. 2002.

[76] 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.

[77] H. Ding and M. Rabinovich. TCP stretch acknowledgements and timestamps: Findings and implications for passive RTT measurement. *SIGCOMM Comput. Commun. Rev.*, 45(3):20–27, July 2015.

[78] N. Ding, A. Pathak, D. Koutsonikolas, C. Shepard, Y. Hu, and L. Zhong. Realizing the full potential of PSM using proxying. In *Proc. IEEE INFOCOM*, 2012.

[79] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing residential broadband networks. In *Proc. ACM/USENIX IMC*, 2007.

[80] C. Dovrolis, P. Ramanathan, and D. Moore. Packet dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Trans. Networking*, 12(6):963–977, 2004.

[81] A. Downey. Using pathchar to estimate internet link characteristics. In *Proc. ACM SIGCOMM*, 1999.

[82] S. Dutton. Measuring page load speed with Navigation Timing. `http://www.html5rocks.com/en/tutorials/webperformance/basics/`.

[83] J. Fabini and T. Zseby. The right time: Reducing effective end-to-end delay in time-slotted packet-switched networks. *IEEE/ACM Trans. Netw.*, In press.

[84] J. Fabini, T. Zseby, and M. Hirschbichler. Representative delay measurements (RDM): Facing the challenge of modern networks. In *Proc. ACM VALUETOOLS*, 2014.

[85] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proc. ACM/USENIX IMC*, 2010.

[86] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proc. ACM MobiSys*, 2010.

[87] FCC. Measuring fixed broadband report - 2016. `https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-report-2016`, 2016.

[88] W. Fok, X. Luo, R. Mok, W. Li, Y. Liu, E. Chan, and R. Chang. Monoscope: Automated network faults diagnosis based on active measurements. In *Proc. IFIP/IEEE IM*, 2013.

[89] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. C. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 17(6):6–16, Nov 2003.

[90] A. Frumusanu. A closer look at Android RunTime (ART) in Android L. `http://anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/`, 2014.

[91] F. Fund, C. Wang, Y. Liu, T. Korakis, M. Zink, and S. S. Panwar. Performance of DASH and WebRTC video services for mobile users. In *Proc. 20th International Packet Video Workshop*, pages 1–8, Dec 2013.

[92] E. Gavaletz, D. Hamon, and J. Kaur. Comparing in-browser methods of measuring resource load times. In *Proc. W3C Workshop on Web Performance 8*, 2012.

[93] A. Gerber, J. Pang, O. Spatscheck, and S. Venkataraman. Speed testing without speed tests: Estimating achievable download speed from passive measurements. In *Proc. ACM IMC*, IMC '10, pages 424–430, New York, NY, USA, 2010. ACM.

[94] L. Gharai, C. Perkins, and T. Lehman. Packet reordering, high speed networks and transport protocol performance. In *Proc. IEEE ICCCN*, pages 73–78, Oct 2004.

[95] S. Giucastro. Getting high precision timing on Android. `http://www.gamasutra.com/view/feature/171774/getting_high_precision_timing_on_.php`.

[96] O. Goga and R. Teixeira. Speed measurements of residential Internet access. In *Proc. PAM*, 2012.

[97] E. Goldoni, G. Rossi, and A. Torelli. Assolo, a new method for available bandwidth estimation. In *Proc. ICIMP*, 2009.

[98] E. Goldoni and M. Schivi. End-to-end available bandwidth estimation tools, an experimental comparison. In *Proc. TMA*, 2010.

[99] C. D. Guerrero and M. A. Labrador. On the applicability of available bandwidth estimation techniques and tools. *Computer Communications*, 33(1):11 – 22, 2010.

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

[101] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. QUIC: A UDP-based secure and reliable transport for HTTP/2. `https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02`, January 2016.

[102] A. Hanemann, J. W. Boote, E. L. Boyd, J. Durand, L. Kudarimoti, R. Lapacz, D. M. Swany, S. Trocha, and J. Zurawski. PerfSONAR: A service oriented architecture for multi-domain network monitoring. In *Proc. Service-Oriented Computing - ICSOC*, 2005.

[103] Y. He and R. Yuan. A novel scheduled power saving mechanism for 802.11 wireless LANs. *IEEE Transactions on Mobile Computing*, 8(10):1368–1383, Oct 2009.

[104] K. Hedayat, B. Networks, R. Krzanowski, A. Morton, and K. Yum. A two-way active measurement protocol (TWAMP). RFC 5357, IETF, Oct. 2008.

[105] F. Heusden. httping. `http://www.vanheusden.com/httping/`.

[106] N. Hu and P. Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE Journal on Selected Areas in Communications*, 21(6):879–894, Aug 2003.

[107] C. Huang, A. Wang, J. Li, and K. W. Ross. Measuring and evaluating large-scale CDNs. In *Proc. ACM IMC*, 2008.

[108] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4G LTE networks. In *Proc. ACM MobiSys*, 2012.

[109] J. Huang, F. Qian, Q. Xu, Z. Qian, Z. M. Mao, and A. Rayes. Uncovering cellular network characteristics: Performance, infrastructure, and policies. Tech. Rep. MSU-CSE-00-2, University of Michigan and Cisco, 2013.

[110] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proc. ACM MobiSys*, 2010.

[111] B. Huffaker, D. Plummer, D. Moore, and K. Claffy. Topology discovery by active probing. In *Proc. Symposium on Applications and the Internet (SAINT) Workshops*, 2002.

[112] IDC. Smartphone OS market share, Q2 2016. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`.

[113] C. Ihrig. Profiling page loads with the Navigation Timing API. `http://www.sitepoint.com/profiling-page-loads-with-the-navigation-timing-api/`.

[114] E. International. ECMAScript language specification. `https://tc39.github.io/ecma262/`.

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

[116] ITU. ICT facts and figures 2016. `http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2016.pdf`.

[117] ITU. International Telecommunication Union. `http://www.itu.int/`.

[118] ITU-T Recommendation Y.1540. Internet protocol data communication service - IP packet transfer and availability performance parameters, 2011.

[119] ITU-T Recommendation Y.1541. Network performance objectives for IP-based services, 2011.

[120] V. Jacobson and R. Braden. TCP extensions for long-delay paths. RFC 1072, IETF, October 1988.

[121] M. Jain and C. Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. In *Proc. PAM*, 2002.

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

[123] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: reliable multicasting with on overlay network. In *Proc. USENIX OSDI*, 2000.

[124] H. Jiang and C. Dovrolis. Passive estimation of TCP round-trip times. *SIGCOMM Comput. Commun. Rev.*, 32(3):75–88, 2002.

[125] H. Jiang, Z. Liu, Y. Wang, K. Lee, and I. Rhee. Understanding bufferbloat in cellular networks. In *Proc. ACM CellNet*, 2012.

[126] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling bufferbloat in 3G/4G networks. In *Proc. ACM/USENIX IMC*, 2012.

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

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

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

[130] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu. Kernel probes (Kprobes). `https://www.kernel.org/doc/Documentation/kprobes.txt`.

[131] C. Kilinc and K. Andersson. A congestion avoidance mechanism for WebRTC interactive video sessions in LTE networks. *Wireless Personal Communications*, 77(4):2417–2443, 2014.

[132] R. Koodli and R. Ravikanth. One-way loss pattern sample metrics. RFC 3357, IETF, Aug. 2002.

[133] R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. *Wireless Network*, 11:135–148, Jan. 2005.

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

[135] 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.

[136] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao. Moving beyond end-to-end path information to optimize CDN performance. In *Proc. ACM IMC*, 2009.

[137] J.-C. Kuester and A. Bauer. Monitoring real Android malware. In *Proc. Runtime Verification*, 2015.

[138] K. Lai and M. Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *Proc. USENIX Symposium on Internet Technologies and Systems*, 2001.

[139] D. Lee, B. Carpenter, and N. Brownlee. Media streaming observations: Trends in UDP to TCP Ratio. *International Journal On Advances in Systems and Measurements*, 3(3 and 4):147–162, 2011.

[140] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong. Mobile data offloading: How much can WiFi deliver? In *Proc. ACM CoNEXT*, 2010.

[141] S. Lee and J. W. Jeon. Evaluating performance of Android platform using native C for embedded systems. In *Proc. IEEE ICCAS*, 2010.

[142] W. Li, W. Fok, E. Chan, X. Luo, and R. Chang. Planetopus: A system for facilitating collaborative network monitoring. In *Proc. IEEE/IFIP IM (Application Session)*, 2011.

[143] W. Li, R. Mok, R. Chang, and W. Fok. Appraising the delay accuracy in browser-based network measurement. In *Proc. ACM/USENIX IMC*, 2013.

[144] W. Li, R. Mok, D. Wu, and R. Chang. On the accuracy of smartphone-based mobile network measurement. In *Proc. IEEE INFOCOM*, 2015.

[145] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: A partition scheme. In *Proc. ACM CASES*, 2001.

[146] X. Luo, E. Chan, and R. Chang. Design and implementation of TCP data probes for reliable network path monitoring. In *Proc. USENIX ATC*, 2009.

[147] X. Luo and R. Chang. Novel approaches to end-to-end packet reordering measurement. In *Proc. ACM/USENIX IMC*, 2005.

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

[149] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proc. USENIX OSDI*, 2006.

[150] B. Mah. pchar: A tool for measuring Internet path characteristics. http://www.kitchenlab.org/www/bmah/Software/pchar/.

[151] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Analyzing the MAC-level behavior of wireless networks in the wild. In *Proc. ACM SIGCOMM*, 2006.

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

[153] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM*, 2002.

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

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

[156] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On dominant characteristics of residential broadband Internet traffic. In *Proc. ACM/USENIX IMC*, 2009.

[157] M. Mathis and M. Allman. A framework for defining empirical bulk transfer capacity metrics. RFC 3148, IETF, Jul. 2001.

[158] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the Internet. *SIGCOMM Comput. Commun. Rev.*, 35(2):37–52, Apr. 2005.

[159] F. Michelinakis, N. Bui, G. Fioravantti, J. Widmer, F. Kaup, and D. Hausheer. Lightweight capacity measurements for mobile networks. *Computer Communications*, 84:73–83, 2016.

[160] P. Mockapetris. Domain names - implementation and specification. RFC1035, IETF, November 1987.

[161] R. Mok, X. Luo, E. Chan, and R. Chang. QDASH: A QoE-Aware DASH System. In *Proc. ACM MMSys*, 2012.

[162] 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.

[163] D. Morrill. On Android compatibility. `http://android-developers.blogspot.hk/2010/05/on-android-compatibility.html`, May 2010.

[164] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser. Packet reordering metrics. RFC 4737, IETF, Nov. 2006.

[165] Netcraft. February 2016 web server survey. `https://news.netcraft.com/archives/2016/02/22/february-2016-web-server-survey.html`.

[166] A. Nikravesh, D. Choffnes, E. Katz-Bassett, Z. Mao, and M. Welsh. Mobile network performance from user devices: A longitudinal, multidimensional analysis. In *Proc. PAM*, 2014.

[167] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao. Mobilyzer: An open platform for controllable mobile network. In *Proc. ACM MobiSys*, 2015.

[168] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon. Evaluation of Android Dalvik virtual machine. In *Proc. JTRES*, 2012.

[169] R. V. Oliveira, B. Zhang, and L. Zhang. Observing the evolution of Internet AS topology. *SIGCOMM Comput. Commun. Rev.*, 37(4):313–324, Aug. 2007.

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

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

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

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

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

[175] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. Modeling tcp reno performance: A simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8(2):133–145, Apr. 2000.

[176] V. Paxson. Strategies for sound Internet measurement. In *Proc. ACM/USENIX IMC*, 2004.

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

[178] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale Internet measurement. *IEEE Communications*, 36(8):48–54, 1998.

[179] C. Pei, Y. Zhao, G. Chen, R. Tang, Y. Meng, M. Ma, K. Ling, and D. Pei. WiFi can be the weakest link of round trip network latency in the wild. In *Proc. IEEE INFOCOM*, pages 1–9, 2016.

[180] Y. Pei, H. Wang, and S. Cheng. A passive method to estimate TCP round trip time from nonsender-side. In *Proc. IEEE ICCSIT*, pages 43–47, Aug 2009.

[181] D. Pezaros, D. Hutchison, R. Gardner, F. Garcia, and J. Sventek. Inline measurements: a native measurement technique for IPv6 networks. In *Proc. INCC*, 2004.

[182] J. Postel. Internet control message protocol. RFC 792, IETF, Sept. 1981.

[183] A. J. Pyles, X. Qi, G. Zhou, M. Keally, and X. Liu. SAPSM: Smart adaptive 802.11 PSM for smartphones. In *Proc. ACM UbiComp*, 2012.

[184] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing radio resource allocation for 3G networks. In *Proc. ACM/USENIX IMC*, 2010.

[185] E. R. Stewart. Stream control transmission protocol. https://tools.ietf.org/html/rfc4960, September 2007.

[186] E. Rescorla and N. Modadugu. Datagram transport layer security. https://tools.ietf.org/html/rfc4347, April 2006.

[187] V. J. Ribeiro, R. H. Riedi, R. G. Baraniuky, J. Navratil, and L. Cottrell. pathChirp: Efficient available bandwidth estimation for network paths. In *Proc. PAM*, 2003.

[188] RIPE NCC. Routing Information Service (RIS). https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/routing-information-service-ris.

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

[190] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Discovering fine-grained RRC state dynamics and performance impacts in cellular networks. In *Proc. ACM MobiCom*, 2014.

[191] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Understanding RRC state dynamics through client measurements with Mobilyzer. In *Proc. the 6th Annual Workshop on Wireless of the Students, by the Students, for the Students (S3)*, 2014.

[192] M. Roughan. Fundamental bounds on the accuracy of network performance measurements. In *Proc. ACM SIGMETRICS*, 2005.

[193] M. Roughan. A comparison of Poisson and uniform sampling for active measurements. *IEEE Journal on Selected Areas in Communications*, 24(12):2299–2312, Dec 2006.

[194] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu. NAPman: Network-assisted power management for WiFi devices. In *Proc. ACM MobiSys*, 2010.

[195] 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.

[196] S. Savage. Sting: A tool for measuring one way packet loss. In *Proc. IEEE INFOCOM*, 2000.

[197] P. Serrano, M. Zink, and J. Kurose. Assessing the fidelity of COTS 802.11 sniffers. In *Proc. IEEE INFOCOM*, 2009.

[198] B. Shaffer. Broadcom and Qualcomm battle for WLAN IC leadership. `https://technology.ihs.com/517658/broadcom-and-qualcomm-battle-for-wlan-ic-leadership`, November 2014.

[199] S. Shalunov, B. Teitelbaum, A. Karp, J. Boote, and M. Zekauskas. A one-way active measurement protocol (OWAMP). RFC 4656, IETF, Sept. 2006.

[200] Y. Shavitt and E. Shir. DIMES: Let the Internet measure itself. *SIGCOMM Comput. Commun. Rev.*, 35(5):71–74, Oct. 2005.

[201] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. Livelab: Measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev.*, 38(3):15–20, Jan. 2011.

[202] R. Sherwood and N. Spring. A platform for unobtrusive measurements on PlanetLab. In *Proc. USENIX WORLDS*, 2006.

[203] R. Sherwood and N. Spring. Touring the Internet in a TCP sidecar. In *Proc. ACM/USENIX IMC*, 2006.

[204] V. Shrivastava, S. Rayanchu, S. Banerjee, and K. Papagiannaki. PIE in the sky: Online passive interference estimation for enterprise WLANs. In *Proc. USENIX NSDI*, 2011.

[205] V. Singh, A. A. Lozano, and J. Ott. Performance analysis of receive-side real-time congestion control for WebRTC. In *Proc. 20th International Packet Video Workshop*, pages 1–8, Dec 2013.

[206] J. Sommers and P. Barford. Cell vs. WiFi: On the performance of metro area mobile connections. In *Proc. ACM/USENIX IMC*, 2012.

[207] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving accuracy in end-to-end packet loss measurement. In *Proc. ACM SIGCOMM*, 2005.

[208] J. Sommers, P. Barford, N. Duffield, and A. Ron. Accurate and efficient SLA compliance monitoring. In *Proc. ACM SIGCOMM*, 2007.

[209] J. Sommers, P. Barford, N. Duffield, and A. Ron. A geometric approach to improving active packet loss measurement. *IEEE/ACM Trans. Networking*, 18(2):652–665, April 2008.

[210] J. Sommers, P. Barford, and W. Willinger. A proposed framework for calibration of available bandwidth estimation tools. In *Proc. IEEE ISCC*, pages 709–718, June 2006.

[211] J. Sommers, P. Barford, and W. Willinger. Laboratory-based calibration of available bandwidth estimation tools. *Microprocessors and Microsystems*, 31(4):222 – 235, 2007. Special Issue with selected papers from the 11th IEEE Symposium on Computers and Communications (ISCC06).

[212] R. Spangler. Analysis of remote active operating system fingerprinting tools. `http://packetwatch.net/documents/papers/osdetection.pdf`, 2003.

[213] Speedchecker Limited. BandwidthPlace Speed Test. `http://www.bandwidthplace.com/`.

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

[215] J. Strauss, D. Katabi, and F. Kaashoek. A measurement study of available bandwidth estimation tools. In *Proc. ACM IMC*, 2003.

[216] K. Sui, M. Zhou, D. Liu, M. Ma, D. Pei, Y. Zhao, Z. Li, and T. Moscibroda. Characterizing and improving WiFi latency in large-scale operational networks. In *Proc. ACM MobiSys*, 2016.

[217] S. Sundaresan, S. Burnett, N. Feamster, and W. de Donato. BISmark: A testbed for deploying measurements and applications in broadband access networks. In *Proc. USENIX ATC*, 2014.

[218] 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.

[219] The IEEE and The Open Group. IEEE Std 1003.1-2008. `http://pubs.opengroup.org/onlinepubs/9699919799/`.

[220] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding mobile app usage patterns using in-app advertisements. In *Proc. PAM*, 2013.

[221] S. Traverso, E. Tego, E. Kowallik, S. Raffaglio, A. Fregosi, M. Mellia, and F. Matera. Exploiting hybrid measurements for network troubleshooting. In *Telecommunications Network Strategy and Planning Symposium (Networks), 2014 16th International*, pages 1–6, Sept 2014.

[222] N. Vallina-Rodriguez, N.Weaver, C. Kreibich, and V. Paxson. Netalyzr for Android: Challenges and opportunities. In *Proc. Workshop on Active Internet Measurements (AIMS)*, 2014.

[223] N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, N. Weaver, and V. Paxson. Beyond the radio: Illuminating the higher layers of mobile networks. In *Proc. ACM MobiSys*, 2015.

[224] W3C. High resolution time level 2. `https://www.w3.org/TR/hr-time-2/`.

[225] 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.

[226] Z. Wen, S. Triukose, and M. Rabinovich. Facilitating focused Internet measurements. In *Proc. ACM SIGMETRICS*, 2007.

[227] L. Wenwei, Z. Dafang, Y. Jinmin, and X. Gaogang. On evaluating the differences of TCP and ICMP in network measurement. *Computer Communications*, 30(2):428–439, 2007.

[228] D. Wu, W. Li, R. Chang, and D. Gao. MopEye: Monitoring per-app network performance with zero measurement traffic. In *Proc. CoNEXT Student Workshop*, 2015.

[229] Q. Xu. *Optimizing Mobile Application Performance through Network Infrastructure Aware Adaptation*. PhD thesis, University of Michigan, 2013.

[230] Y. Xu, Z. Wang, W. K. Leong, and B. Leong. An end-to-end measurement study of modern cellular data networks. In *Proc. PAM*, 2014.

[231] L. Xue, C. Qian, and X. Luo. Androidperf: A cross-layer profiling system for Android applications. In *Proc. IEEE IWQoS*, 2015.

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

[233] J. Yeo, M. Youssef, and A. Agrawala. A framework for wireless LAN monitoring and its applications. In *Proc. ACM WiSe*, 2004.

[234] J. Yoo, T. Huehn, and J. Kim. Active capture of wireless traces: Overcome the lack in protocol analysis. In *Proc. ACM WinTech*, 2008.

[235] Y. Zhang and N. Duffield. On the constancy of Internet path properties. In *Proc. ACM IMW*, 2001.

[236] X. Zhou and K. L. Calvert. Lightweight privacy-preserving passive measurement for home networks. In *Proc. IEEE ICC*, pages 1019–1024, June 2015.

[237] T. Zseby. Deployment of sampling methods for SLA validation with non-intrusive measurements. In *Proc. PAM*, 2001.