This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

Toward Accurate Network Delay Measurement on Android Phones

Weichao Li*, *Member, IEEE,* Daoyuan Wu†, *Student Member, IEEE,* Rocky K. C. Chang‡, *Member, IEEE,* Ricky K. P. Mok§, *Member, IEEE*

Abstract—Measuring and understanding the performance of mobile networks is becoming very important for end users and operators. Despite the availability of many measurement apps, their measurement accuracy has not received sufficient scrutiny. In this paper, we appraise the accuracy of smartphone-based network performance measurement using the Android platform and the network round-trip time (RTT) as the metric. We show that two of the most popular measurement apps—Ookla Speedtest and MobiPerf—have their RTT measurements inflated. We build three test apps for 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 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 the 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. We propose and implement a native measurement app to mitigate the delay overhead in the Android runtime, and the resulted delay inflation in the user space can be kept under 1.5ms for almost all cases.

Index Terms—Network measurement, mobile phone, Android, accuracy.

1 Introduction

Mobile devices, notably smartphones and tablets, have already become essential parts of our daily lives because of their mobility and rich functionalities. The popularity of mobile devices and mobile applications have changed the way users access and utilize the network. According to [1], 84% of apps require permission of Internet access from a pool of 55K Android apps randomly picked from the official Android app market. An ITU (International Telecommunication Union) report also shows that the penetration rate of mobile-broadband networks (3G or above) reaches 84% of the global population [2]. Therefore, understanding mobile network performance is crucial for providing good quality of experience to users. For example, some recent performance studies characterize the performance of LTE networks [3] and optimize mobile application performance [4]. The data collected by `Speedtest.net` are used for comparing the performance between cellular and WiFi networks [5].

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 measurement apps in Android [6], [7], [8], [9], iOS [10], [11], and Windows Phone [12], [13]. In particular, the Ookla Speedtest app [9] 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 [7].

Despite the availability of many measurement apps, their measurement accuracy has not received sufficient scrutiny. Since these measurement apps are implemented in the user space, they usually measure user-level performance. While they are useful for characterizing the user experience, they cannot reliably infer the network-level performance. Accurate measurement of the network condition is an important first step towards diagnosing and optimizing network performance. Measurement results with too much user-level noise can lead to wrong conclusions about the network conditions. A typical example is that when a measurement app reports a large network delay, users usually assume it is the network problem, and this is obviously not always true. Moreover, the network delay today is getting smaller. For example, FCC reported in 2016 that the median latency of broadband service for each monitored ISP in the US ranges from only 12 ms to 58 ms [14]. Another data source from Ookla shows that the mean RTT for the US in January 2015 is only 38ms [15]. Our own crowdsourcing measurement also confirms that the network delays experienced by the current apps are usually small (cf. §2.2). Considering the trend of continuous improvement in the network latency, the additional delay incurred in the phone is going to have more impact on the network latency measurement.

In this paper, we appraise the accuracy of smartphone-based network performance measurement. We focus on the RTT measurement, because it is the most fundamental metric, from which many other performance metrics can be derived, such as delay variation, capacity, and available bandwidth. Moreover, we consider only Android smartphones, because the source code of the measurement apps is available to us. In particular, we have evaluated the accuracy of Ookla Speedtest app and MobiPerf for measuring network RTT and discover that the 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

* *Huawei Future Network Theory Lab, Hong Kong*
† *School of Information Systems, Singapore Management University*
‡ *Department of Computing, The Hong Kong Polytechnic University*
§ *University of California, San Diego/CAIDA*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

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 six 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. Since the delay inflations depend on the device models and length of network path, different smartphones can report totally different results for the same network condition. Furthermore, even the same smartphone can incur different degrees of delay overheads for different network paths. Our further analysis reveals that the delay inflations occur in both user space and kernel space. We also discover that the delay inflations introduced in the user space is asymmetric in the outgoing and incoming direction.
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 the inefficient long path of subfunction 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.
3) (Mitigating the delay inflation) Our approach to mitigating the user-space delay inflation is to eliminate the runtime overhead. We implement the core measurement logic into a native C program and invoke it through an external system call. Experiment results show that the user-space delay inflation can be kept under 1.5ms for most cases. For the kernel-space delay inflation, both sending packets in small interval and injecting warm-up and background traffic can keep the WNIC in the active states and consequently remove the delay overhead.

The remainder of the paper is organized as follows. In §2, we summarize the implementation details of the existing measurement apps and introduce our approach to measuring the accuracy of three main methods in a testbed. In §3, we detail the different aspects of our testbed setup, including the use of multiple sniffers to obtain a complete trace for acquiring timestamp information. §4 and §5 report evaluation results obtained from the Internet experiments for Ookla Speedtest and MobiPerf, and the controlled testbed, respectively. Meanwhile, the root-cause analysis is also included in §5. In §6, we then propose a measurement method to mitigate the delay inflation and discuss how to apply our study to other scenarios. After highlighting the related works in §7, we conclude this paper in §8.

2 Background

2.1 Measurement apps in Android

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 the RTT measurement methods employed by a number of Android apps by inspecting their code and the packets exchanged between the Android phone and servers. Table 1 documents the implementation details for some popular measurement apps in Android. We summarize 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, directly executing the 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 nanosecond to millisecond. Even for the result reporting, different apps may have their own choices. They can output mean, min, or both.

2.2 Importance of accurate delay measurement

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 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.

In this paper, 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 30ms. 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 [18], 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

TABLE 1: Implementation details of the existing network measurement apps in Android.

App	Supported probe type	Core method	Timing function	# of samples	Results
Ookla Speedtest [9]	HTTP GET	`java.net.URLConnection`	`SystemClock.uptimeMillis()`	6	Min
MobiPerf [6]	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 [7]	UDP	`java.net.DatagramPacket`	`Date().getTime()`	200	Mean
Speedchecker [8]	TCP SYN/ACK**	`java.net.HttpURLConnection`	`Date().getTime()`	1	N/A
V-SPEED Internet Speed Test [16]	TCP SYN/ACK**	`java.net.HttpURLConnection`	`System.currentTimeMillis()`	50/30/20*	Mean
FCC Speed Test [17]	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.

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).

Accurate network latency measurement is essential for making accurate inferences from network measurement data. A classic example is analytical TCP throughput modelling with RTT, packet loss rate and TCP parameters as inputs [19], [20]. With negligible packet loss, the throughput is known to be inversely proportional to the RTT. Therefore, the actual throughput will be underestimated as a result of the inflated delay by the same amount (e.g., 30% throughput underestimation for 30% delay inflation). This inaccurate inference could result in selecting a suboptimal initial video bitrate in [21] which uses a decision tree to determine the best bitrate. Moreover, recently a growing number of works apply various machine learning techniques to build models based on network measurement data for network operational problems. The delay measurement inaccuracy could similarly affect their inference accuracy. To just name a few of these works, Liao et al. [22] employ machine learning techniques to infer and predict the network performance of unmeasured network based on the known network performance. In [23], Ahmed et al. build models based on measurement results and use it to detect and localize the network performance degradation problems in cellular networks.

Another example of illustrating the importance of accurate delay measurement tool can be found in [24]. This work proposes a framework for measuring and characterizing WiFi latency for a large campus WiFi network. A critical component in this framework is using `ping2` to obtain accurate delay measurement. Instead of sending one ICMP echo request packet, `ping2` sends two consecutive packets, with the first one to wake up the measured devices which may be in the energy saving mode and the second to obtain the delay measurement. In their evaluation, they found that the regular ping is unusable in practice, because the measurement is inflated by energy-saving schemes in various degrees. `Ping2`, on the other hand, gives consistently lower and more accurate delay measurement. However, they did not investigate and characterize the delay inflation as we do in this paper.

2.3 Measuring the delay overhead

To evaluate the accuracy of Android measurement apps, we use the *delay overhead* defined in [25], which is the difference between the measured and the actual network delay. Given a simple probe-response scenario in Fig. 1, 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, arriving 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

$$\Delta d = d_u - d_n = (t_u^i - t_u^o) - (t_n^i - t_n^o). \quad (1)$$

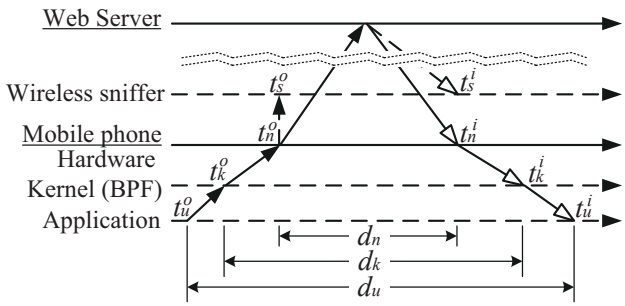


Fig. 1: Measurement flow for Android apps.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

`clock_gettime()` through vsyscall according to POSIX.1-2008 [26]. Giucastro tested the granularity and performance of these two functions on some Android phone, and found that the average cost for executing the timing function is about 1μs [27]. 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 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 diverted from a network tap. However, it is not reliable to capture all the packets in the air using a single wireless sniffer. We will explain how we tackle the issue in §3.

2.4 Multi-layer analysis

To locate where the overheads are introduced, we perform multi-layer analysis by dissecting the delay overheads into several components. Back to the packet sending and receiving processes in Fig. 1, 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

$$\Delta d_k = d_k - d_n = (t_k^i - t_k^o) - (t_n^i - t_n^o). \quad (2)$$

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

$$\Delta d_u = d_u - d_k = (t_u^i - t_u^o) - (t_k^i - t_k^o). \quad (3)$$

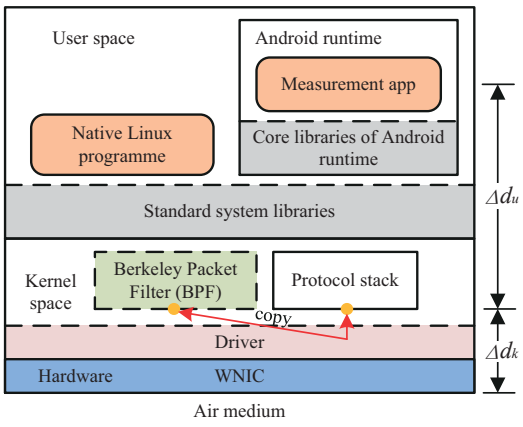


Fig. 2: The user-kernel delay overhead and kernel-phy delay overhead.

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 Fig. 2, 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 the WNIC's hardware and driver, whereas Δd_u on the performance of the user space and part of the kernel space. Although our evaluation in this paper is based on IEEE 802.11g network, the analysis of Δd_u is still valid for cellular networks, such as HSPA and LTE.

2.5 Runtime environment in Android

Although built on Linux, Android differs from other Linux distributions by employing Java as the official programming language. Java achieves the platform-independency 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) is 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 [28], [29].

In this paper, we also take into account the performance differences between DVM and ART, because a faster Android runtime could lead to shorter Δd_u according to Fig. 2. 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 [30]. Therefore, our experiment design (see §5) 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.

3 A MULTIPLE-SNIFFER TESTBED

To evaluate the accuracy of measurement apps, we build a multiple-sniffer testbed in Fig. 3. 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 data rate of the WLAN is configured to 54Mbps. 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 run 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.

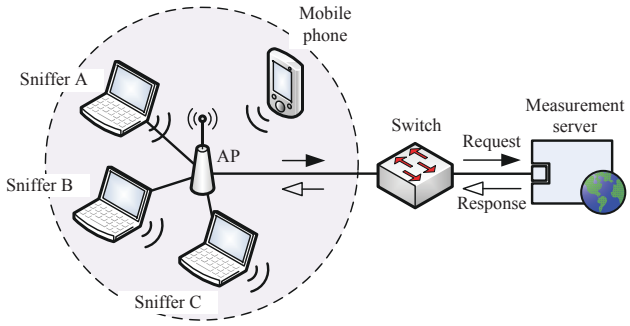


Fig. 3: The testbed setup where the packet sniffers, mobile phone, and wireless AP are placed within a distance of 0.5m.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

3.1 Wireless packet capturing

A simple way to passively monitor the wireless network traffic is to listen to the WNIC of the AP [31], [32]. 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, and retry [32]. It is costly to solve the problem through, for example, employing APs that support WNIC hardware timestamp, modifying the driver, and rebuilding kernel. We therefore use the packet capturing method described in [33] for its easy deployment and low cost. 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 §3.2 to evade the clock drift offline.

We employ three sniffers, because a single sniffer will miss many packets [33], [34]. Although we put the AP, mobile phone, and the sniffers very close to one another (within a distance of 0.5m and on the same level), we still find random frame losses and duplications in the captured traces. Such frame losses, which are unpredictable and independent across sniffers, can be successfully transmitted between the AP and smartphone but missed by the sniffers. To ensure the completeness of a packet trace, Serrano et al. propose to use multiple sniffers to merge the individual traces [34], 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%.

3.2 Trace merging and clock skew handling

The basic idea of trace merging is to identify the missing frames and copy them to the incomplete trace. We first randomly assign a data trace as the main trace and others as reference traces. The missing data frames in the main trace can be identified after comparing all traces. Finally, we insert the missing frames to the correct locations in the main trace and adjust their timestamps, so that they are coherent to the local frames.

Due to the existence of clock skew between two sniffers, the most challenging part in this procedure is to accurately recover the timestamps of the missing frames. A most straightforward approach is to synchronize the sniffers [35], 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."

Fig. 4 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, since 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 $\delta_{\{A,B\}}$. In our case, the clock skew is ∼15 ppm (parts per million) between sniffer B and A, and ∼9 ppm between sniffer B and C. To recover the timestamp $C_A(t_2)$, we make use of the reference frame *pkt1* in Fig. 4 in both traces:

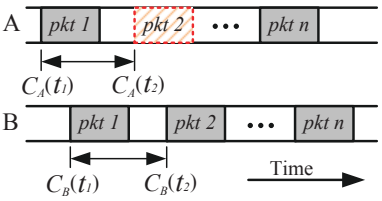


Fig. 4: Procedure of trace merging and time recovery.

$$C_A(t_2) = C_A(t_1) + (C_B(t_2) - C_B(t_1)) + \int_{t_1}^{t_2} \delta_{\{A,B\}}(t)dt. \tag{4}$$

We employ the closest beacon frame before the lost frame as the reference frame for two reasons: i) beacon frames are seldom missing in our traces; and ii) beacon frames are observed with the smallest fluctuations when a sniffer reports the capturing time. Accordingly, every lost frame can be bounded within a specific beacon interval (102.4ms by default). Given such short period of time and the very small clock skew, the clock offset between two sniffers can be ignored (smaller than 1.5μs in our case). Moreover, t_2 and t_1 can be further replaced by $C_B(t_2)$ and $C_B(t_1)$. Therefore, we can recover $C_A(t_2)$ by

$$C_A(t_2) \approx C_A(t_1) + (C_B(t_2) - C_B(t_1)). \tag{5}$$

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'. Let $C_p(t)$ and $C_s(t)$ be the times reported by the phone and sniffer at time t, respectively, 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

$$\Delta D_{\{p,s\}} = (C_p(t_2) - C_p(t_1)) - (C_s(t_2) - C_s(t_1)). \tag{6}$$

We have tested several Android phones and wireless sniffers. The clock skews among them are all within the range of ±100 ppm. For an end-to-end network path, the RTT is usually tens to hundreds milliseconds [36]. 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 approximated by

$$\Delta d \approx (t_u^i - t_u^o) - (t_s^i - t_s^o). \tag{7}$$

4 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 [37], [38], [39], [40]. 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).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

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 sample ($min(d_u)$) 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.

4.1 Experiment setup

We conduct our experiments in the testbed described in §3, 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 4. 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 will change to TCP SYN/ACK. In our 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 Evaluation

4.2.1 Ookla Speedtest

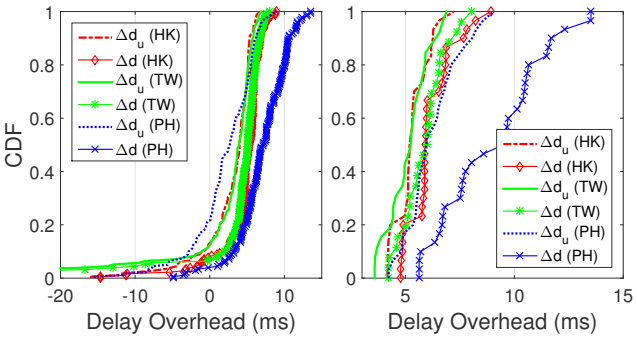
Capturing packets both internally (through `tcpdump`) and externally (through sniffers) allows us to compare the RTTs measured by the apps (d_u), in the kernel (d_k) and in the air (d_n). As shown in Table 2, although Ookla Speedtest reports the smallest value as the final RTT for each measurement ($min(d_u)$), this value can be around 3−7ms larger than the actual network RTT. Even compared to d_k measured by `tcpdump`, it still overestimates the delays by 1−3ms.

TABLE 2: RTTs measured by Ookla Speedtest ($min(d_u)$), in the kernel (d_k) and in the air (d_n) (mean with 95% confidence interval, in ms). Here d_k and d_n are calculated with all samples.

Target Server	$min(d_u)$	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 Fig. 5. We consider two cases: i) comparing all 6 samples of d_k and d_n with the reported RTT for each measurement, and ii) using the minimum d_k and d_n for a fairer comparison. For case (i) (as shown in Fig. 5(a)), although Speedtest has already returned 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 ∼14ms. Due to the network fluctuation during the measurement, it is still possible that the smallest d_u is smaller than d_k and d_n of other samples. Therefore, RTT underestimation events can be found for this case, even though they account for only a small portion.

For case (ii), since smaller d_n usually corresponds to smaller d_k and d_u, they are not affected by the performance fluctuation during the measurement. Fig. 5(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.



(a) With all samples. (b) With the smallest d_k and d_n.

Fig. 5: CDF plots of Δd and Δd_u for Ookla Speedtest.

4.2.2 MobiPerf

Table 3 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 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 ∼17ms).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

TABLE 3: 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 min	d_u mean	d_u max	d_k	d_n
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

5 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 conduct the experiments. Their detailed hardware configurations and OS versions are listed in Table 4. 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 §5.1) 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.

TABLE 4: 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

5.1 Building test measurement apps

Employing existing apps for systematic evaluation is difficult, because we cannot easily switch the measurement target to our measurement server and control the actual 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 1. 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 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. The class `java.net.HttpURLConnection` is employed to implement this app. Here the outgoing and incoming packets are complete HTTP GET request and response messages. We limit the size of the 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()`.

5.2 Overview

Table 5 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 can be as high as 2.9ms. Moreover, the overheads are device and network length dependent, which means that they cannot be easily modeled and calibrated through statistical methods.

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, with mean values usually larger than 10ms. For some extreme cases, the

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

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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

TABLE 5: 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

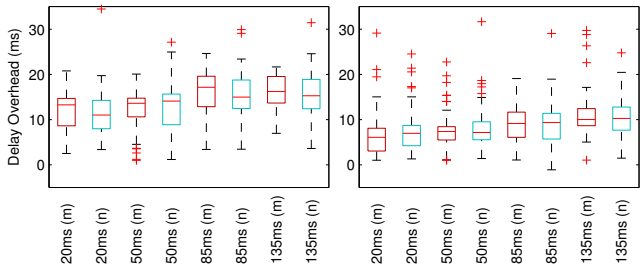
Note *: G for Google Nexus 5, H for HTC One, S for Sony Xperia J, W1 for Huawei G7 Plus, W2 for Huawei Honor 7, and W3 for Huawei Mate 8.

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 also be 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, for 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 ∼7ms, ∼12ms, and ∼7ms for Native ping, Inet ping, and HTTP ping, respectively. But when the RTT increases to 85ms and 135ms (long RTTs), the mean Δds increase to ∼14ms, ∼16ms, and ∼10ms, with additional values of 3-7ms.

5.3 Effect of timing functions

The results presented in Table 5 are measured when `System.currentTimeMillis()` is used. Since it is reported that this function could have coarse granularity (such as ∼15ms) in some OS [25], 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 §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 Fig. 6. 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.

Fig. 6: 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()` are 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.

5.4 Effect of runtime

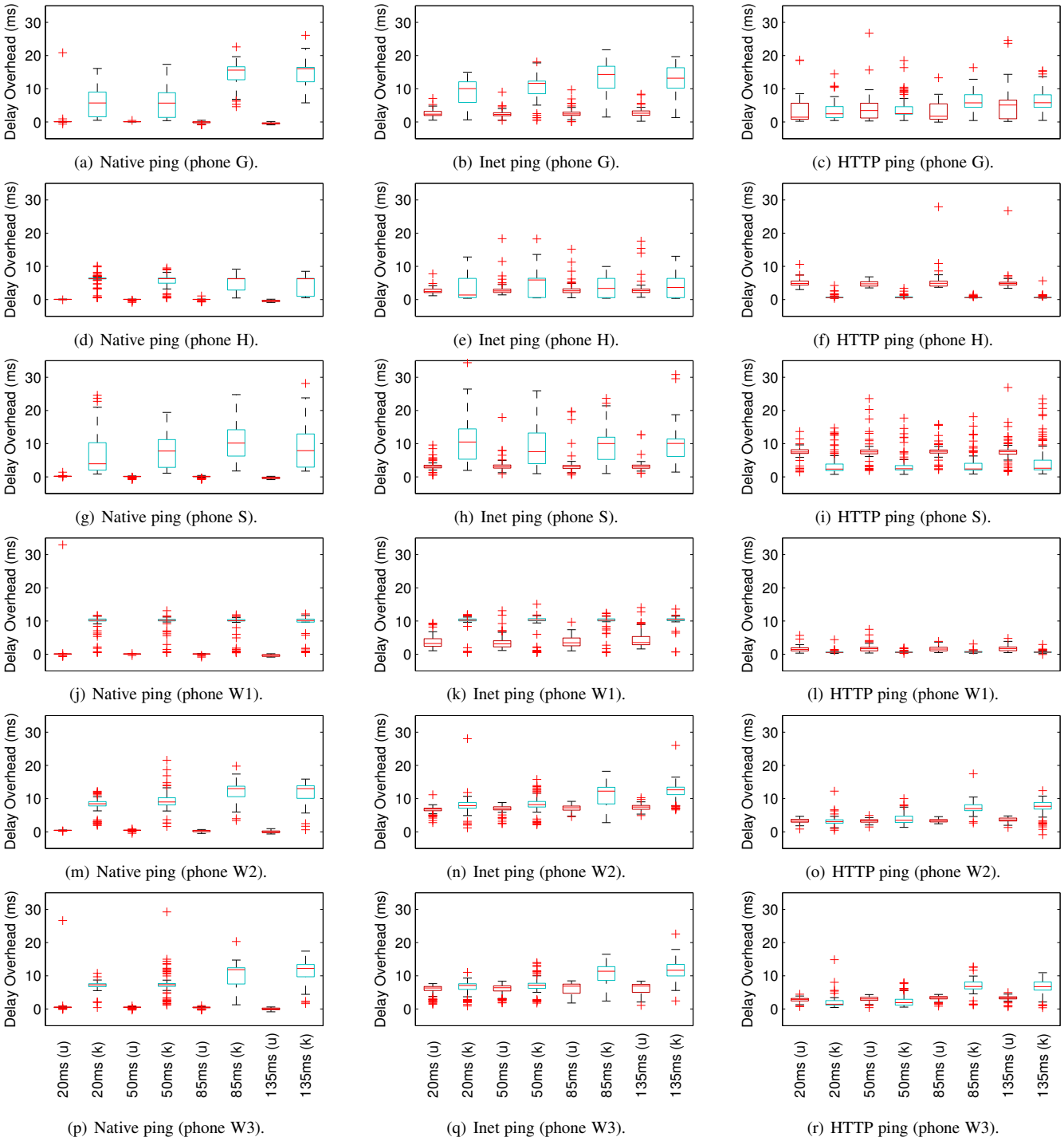
Android 4.4.2 allows us to switch runtime between DVM and ART in the developer options. Therefore, we run Inet ping and HTTP ping on phone G with the same experiment settings to examine their delay overheads in ART. We link the results with those obtained in DVM on the same phone (described in §5.2) in box plots, as shown in Fig. 7. Here we do not consider Native ping, because the `ping` program is not executed in the runtime but runs as a native Linux program.

Fig. 7(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 Δd 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 Fig. 7(a), Inet ping has higher Δd with ART. This observation can also be confirmed by Table 5, where the delay overheads measured by W1-W3 are usually higher than the other three phones. We will discuss the reasons in §5.8.

5.5 User-space and kernel-space overheads

As described in §2.3, during our previous experiments, we also run `tcpdump` in the background on those three test phones to obtain

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing



(a) Native ping (phone G). (b) Inet ping (phone G). (c) HTTP ping (phone G).

(d) Native ping (phone H). (e) Inet ping (phone H). (f) HTTP ping (phone H).

(g) Native ping (phone S). (h) Inet ping (phone S). (i) HTTP ping (phone S).

(j) Native ping (phone W1). (k) Inet ping (phone W1). (l) HTTP ping (phone W1).

(m) Native ping (phone W2). (n) Inet ping (phone W2). (o) HTTP ping (phone W2).

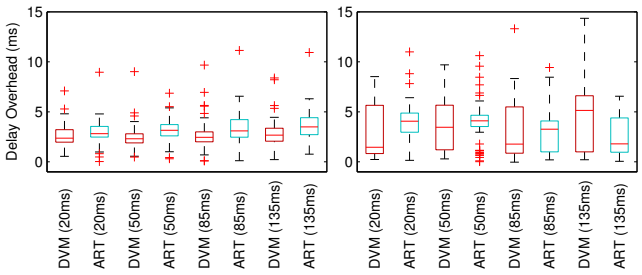
(p) Native ping (phone W3). (q) Inet ping (phone W3). (r) HTTP ping (phone W3).

Fig. 8: Box plots for the user-kernel delay overheads (Δd_u, red) and kernel-phy delay overheads (Δd_k, cyan).

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 Fig. 8. Although we use median Δd_u and Δd_k for comparison in the following, the delay overheads are in fact within a wide range. For example, Δd_k of Inet ping introduced by phone S can range from ∼2ms to ∼26ms (as shown in Fig. 8(h)).

We first focus on Δd_u experienced by the three test apps. In general, Δd_u can be considered as RTT-independent, because each test app experiences very close Δd_u in the same phone no matter what the emulated network RTT is. Fig. 8(a), 8(d), 8(g), 8(j), 8(m), and 8(p) 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 running DVM (G, H, and S), Δd_k contributes the majority of the total delay overheads, as shown in Fig. 8(b), 8(e), and 8(h), which is similar to Native ping except that the layer above the kernel space adds 2-4ms extra delay. But for the rest of the phones, Inet ping encounters much larger Δd_u (see Fig. 8(k), 8(n), and 8(q)). For W2 and W3 in particular, Δd_u can be around 6-7ms. As for HTTP

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing



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

Fig. 7: Delay overhead comparison in box plot for phone G when different runtimes are adopted (red for DVM, and cyan for ART).

ping, the phones 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 a Java API usually involves several more function calls (see §5.8). 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 phone 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 the overall delay overheads in §5.2.

5.6 Delay overhead asymmetry

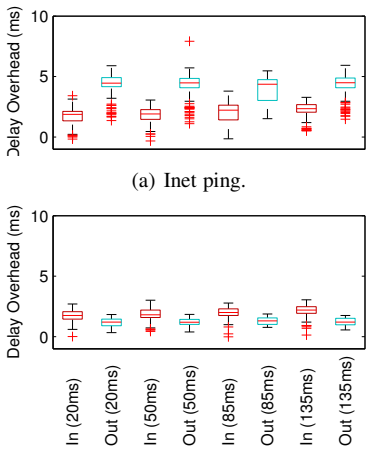
Running `tcpdump` also allows us to analyze the (a)symmetry of the delay overheads occurring in the app. Since Android uses the same clock source of the underlying Linux system, the timestamps recorded by the measurement apps and `tcpdump` are comparable. Therefore, we can measure the outgoing user-kernel 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 Fig. 9 for Inet ping and in Fig. 10 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 Fig. 9 and Fig. 10 show significant delay asymmetry. For example, for Inet ping, establishing a TCP connection costs more time in the outgoing direction. For phone W1, W2, and W3 in particular, the disparity can be larger than 3ms. On the 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, phone W1, W2, and W3 experience much smaller incoming delay overheads than phone G, H, and S. Our further analysis in §5.8 shows that the performance difference between Android 4 and 5/6 is due to the change in the Java I/O library.

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 phone H and S.

5.7 Other WiFi networks and issues

Our evaluations in both Internet and testbed so far are conducted under 802.11g network. To investigate whether the network delay is also inflated in other types of WiFi networks, we conduct similar experiments in the same testbed under 802.11n network on 5GHz band. Since our sniffers cannot capture any data packets with the network speed higher than 130Mbps, our experiments are performed with 54Mbps and 130Mbps. Fig. 11 shows the Δd_u and Δd_k for phone W3 (Huawei Mate 8) in 802.11n network when the transmission speed is set to 130Mbps. Compared to 802.11g network (Fig. 8(p), 8(q), and 8(r)), the delay overheads encountered in 802.11n network follow nearly the same distribution for all three test apps. In detail, Native ping still introduces nearly no overhead between the app and kernel, whereas Inet ping and HTTP ping do. When the emulated RTT increases to 85ms and 135ms, the Δd_ks also increase from ∼2ms to ∼7ms for HTTP ping and from ∼7ms to ∼11ms for Native ping and Inet ping. We also plot the distributions of Δd_u^i and Δd_u^o for Inet ping and HTTP ping in Fig. 12(a) and Fig. 12(b), respectively. Again, no significant difference can be observed between 802.11g and 802.11n. The other phones have similar results, therefore not shown here.



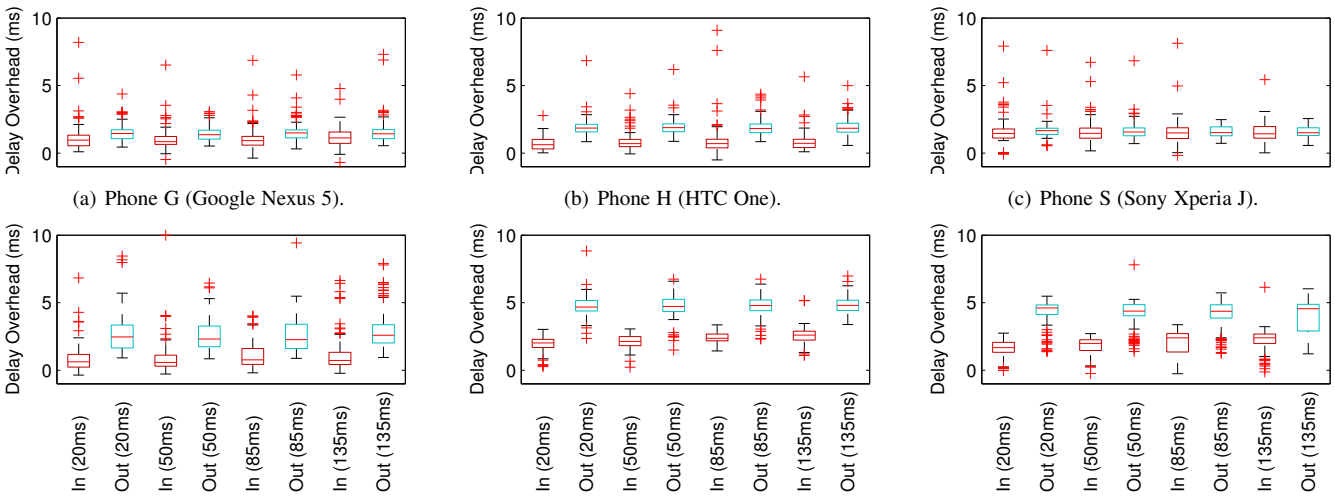
(a) Inet ping.

(b) HTTP ping.

Fig. 12: Box plots of the delay overhead asymmetry for HTTP ping and Inet ping (for phone W3).

Another source of delay overhead is due to the frame aggregation which has been available since 802.11n. The frame aggregation is designed to increase the throughput by sending multiple frames in a single transmission or a superframe. On the down side, frame aggregation can increase the packet delay in two ways. The first is the delay of forming a superframe. Instead of sending a frame immediately, it may have to wait for other

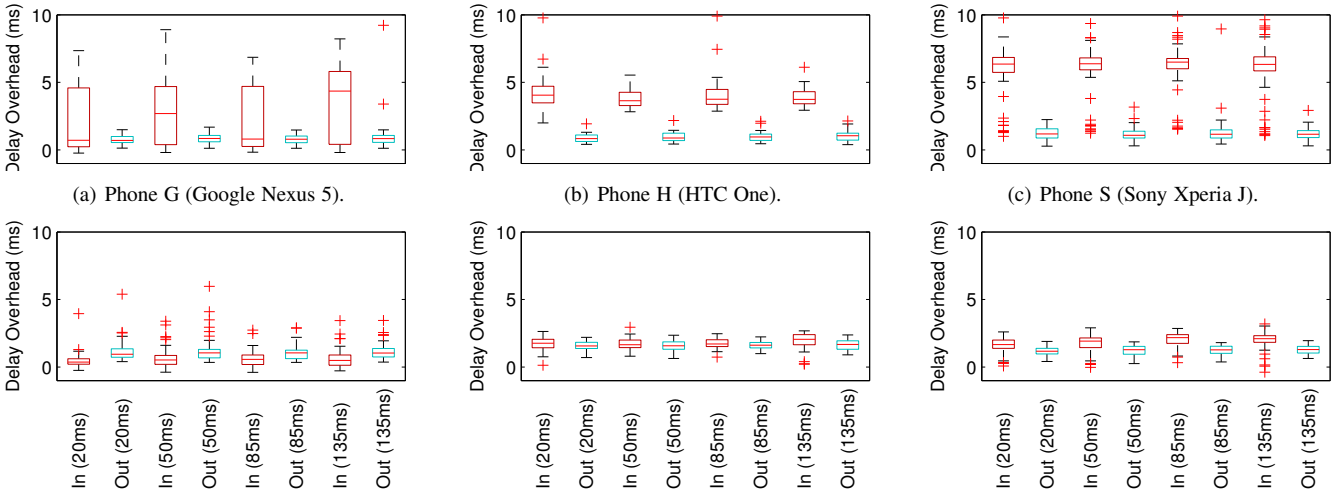
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing



(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).

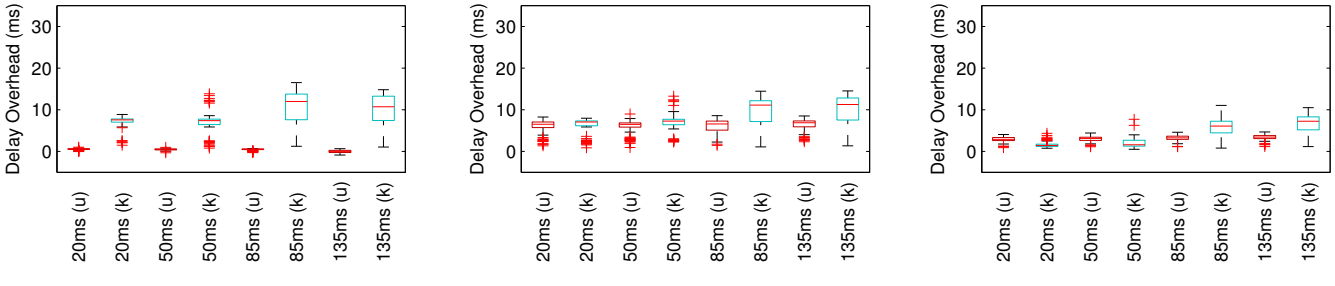
Fig. 9: Box plots of the delay overhead asymmetry for Inet ping.



(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).

Fig. 10: Box plots of the delay overhead asymmetry for HTTP ping.



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

Fig. 11: Box plots for the user-kernel delay overheads (Δd_u, red) and kernel-phy delay overheads (Δd_k, cyan) for phone W3 in 802.11n network.

frames for aggregation. The second is longer delay of transmitting a superframe. Unlike a single-frame transmission, a superframe is more prone to frame collision or corruption due to its length. Therefore, it could take a superframe a longer time to transmit successfully when the channel is congested or noisy.

There are some limitations for our testbed to capture the delay overhead caused by frame aggregation. As seen from §3, our testbed is noise free and hosts only one smartphone, therefore observing almost no retransmission events. Moreover, since our sniffers cannot capture any data packets with network speed higher

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

than 130Mbps, we cannot test higher data rate to facilitate frame aggregation. However, we maintain that our testbed can still obtain accurate measurement for the delay overhead should the frame aggregation be present, because our results will not be affected by the two aforementioned delays associated with frame aggregation.

For the delay of forming a superframe, frame aggregation will be performed only when the total size of the waiting packets reaches the size threshold (3,839/7,935 bytes for A-MSDU, 65,535 bytes for 802.11n A-MPDU, and 4,692,480 bytes for 802.11ac A-MPDU) or the age of the oldest packet reaches a pre-defined packet holding time. A-MSDU and A-MPDU are the two frame aggregation schemes for IEEE802.11. Therefore, a measurement probe could be delayed by at most the packet holding time if there are not enough packets to induce a frame aggregation upon its arrival. In lack of a standard, the choice of the packet holding time is vendor dependent. For example, according to [41], there is no waiting/holding time to form an A-MPDU, and the maximal delay can be set to 1s for A-MSDU. In [42], an aggregated frame is sent immediately without waiting for more frames if the transmission queue is empty. Our analysis of the WNIC driver source code also shows that the holding time is not large, e.g., 5ms for A-MPDU in Broadcom's "bcrm80211" driver. To sum up, the delay overhead due to forming a superframe is very minimal in practice.

For the second source of delay inflation, a longer delay overhead may be recorded because of the frame retransmissions. Since a sniffer can timestamp only when receiving a successfully transmitted frame, the time for frame retransmissions will be counted towards the delay overhead. However, our testbed experiments with 802.11n are performed with good channel quality, and there are no frame retransmissions in our dataset. Therefore, our delay overhead evaluation will not be affected by this type of delay overhead.

5.8 Diagnosis of delay overhead in user space

Android provides `Debug` class and `Traceview` tool to trace and profile function executions in runtime [43]. 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 the function tracing immediately before and after invoking the core measurement methods. As Traceview can analyze the function behavior only in the runtime, 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. To profile the performance of system-layer functions, we use `strace` [44]. Note that we cannot enable `Debug` feature and `strace` simultaneously, because `strace` could introduce significant system overhead for the `Debug` class to record timestamps.

We analyze three core functions, which are employed by Inet ping, MobiPerf's TCP SYN/ACK method, and HTTP ping. We trace their subfunction calls on Google Nexus 5 with DVM or ART enabled in an alternate manner. Our analysis shows that all these functions involve a long series of subfunction calls before they are bridged to the native network functions defined in the bionic libc library (`sys/socket.h`). The long path of subfunction invocation is very inefficient, because it introduces much unnecessary execution time. For example, `InetAddress.getByName().isReachable()` has to parse the address information and prepare a socket before sending out a TCP SYN packet to the remote endpoint, which takes a longer time in the outgoing direction than the incoming direction. That is why we observe larger Δd_u^o in §5.6. For `HttpURLConnection.connect()`, we find that both preparing the HTTP engine and processing the received packets (e.g., `Platform.getMtu()`) can introduce very large additional delay. This observation agrees with the fact that MobiPerf's TCP SYN/ACK method incurs very large delay inflation in §4.2.2. Finally, our analysis shows that the subfunctions decoding the HTTP response message also takes a very long time for `HttpURLConnection.getInputStream()`, which explains why the HTTP ping method introduces more delay overheads in the incoming direction in Fig. 10.

Our function tracing further reveals that there is no significant difference in subfunction calls between ART and DVM. Therefore, the increment of Δd_u for Inet ping could be due to the runtime inefficiency of ART. Another notable finding is that Android 5/6 replaces the default IO functions (`java.io`) with a third-party library (`okio`). This new IO library leads to a significant delay degradation of HTTP ping in the incoming direction.

5.9 Diagnosis of delay overhead in kernel space

By studying the source code 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` [45] 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 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 6. 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.

TABLE 6: 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()`	68
	`ip_rcv()`	86
	`netif_rx()`	73

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 (the details can be found in [46]). For the Broadcom WNIC chipsets, we find that the driver will put the SDIO bus into sleeping mode if there is no packet sending and receiving for a pre-defined period (50ms by default). Although the bus dormancy does reduce the energy consumption, it introduces additional delays for the driver to bring up the bus. Such state promotion delay can be large than 10ms. This explains why phone G, S, W2, and W3 will have a large Δd_k

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

(>10ms) when the emulated RTTs are increased to 85ms and 135ms (see §5.5): the SDIO bus has entered the sleeping mode before the response packet arrives (>50ms), and it takes more than 10ms to wake up to process the packet. And we can also infer that more frequent packet sending/receiving activities will result in smaller delay overheads. For Qualcomm WNIC chipsets, although they connect to the system via SMD interface instead of SDIO, they share similar energy-saving mechanism but with a shorter state demotion time.

6 Discussion

6.1 A better practice

Our analysis in both user space and kernel space shows that the long path of subfunction invocations in the Android runtime 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 bypassing the effect of runtime and migrating 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 §3 and compute Δd_u based on Eqn. (3).

TABLE 7: 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

We compare Δd_u measured by External ping to the other two in-DVM apps in Table 7. 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, the user-kernel delay inflation can be kept 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. For example, for those rooted phones, we can use the `ping` program with a small packet sending interval[4]. Another is to inject warm-up and background traffic to keep the WNIC in the active state [46].

6.2 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 also 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, there is no difference in the data path of the kernel and user space. Reducing the runtime overhead can definitely mitigate the user-kernel delay overhead.

As for the energy-saving mechanisms, our testbed experiment results for WiFi networks in [46] show that our approach of injecting warm-up and background traffic can cap the kernel-phy overhead to 3ms. On the other hand, cellular networks adopt RRC (Radio Resource Control) [3], [38], [47], which is very similar to the Power Saving Mode in WiFi networks. Therefore, a similar approach can also mitigate the delay overhead for cellular networks. Our another set of Internet experiment results, which are not reported in [46], show that $85-97\%$ of the delay measurement obtained by our approach are less than Ookla Speedtest and MobiPerf, especially when TCP control messages are employed as measurement probes.

Although our study focuses on network delay measurement, other performance metrics, such as delay variation and bottleneck capacity, will also be affected by the runtime overhead and the driver's energy-saving mechanisms. For capacity measurement, obtaining accurate network RTT is the prerequisite to packet-pair capacity measurement via minimum delay sum or minimum delay difference [48], [49]. Although the behavior of packet pair or packet train could be changed by the base station due to the proportional fair queue, Michelinakis et al. show that it is still possible to apply these techniques to measure the link capacity in cellular networks [50]. Besides obtaining accurate network RTT, our delay overhead mitigation methods can also improve the latency performance of non-measurement apps, especially delay-sensitive apps. These apps also suffer from the latency introduced by the inefficient Java function calls and various energy-saving mechanisms. The delay inflation in user space can be mitigated by implementing the network functions in native Linux program. In fact, some apps (e.g., Noroot Firewall [51]) have already implemented their network functions in native Linux program to reduce the packet forwarding delay. However, this approach is too costly for other less delay-sensitive apps. For the delay overhead due to energy efficiency mechanisms, our approach of keeping the WNIC awake can also improve the app's latency performance. Fig. 13 plots the delay overhead experienced by ping measurement for two packet sending intervals (10ms and 1s). As shown, if the packet sending interval is small, the delay overhead can be reduced

4. `Ping` requires root privilege when the packet sending interval is smaller than 200ms.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing

to close to 0. This inexpensive scheme can therefore benefit all non-measurement apps.

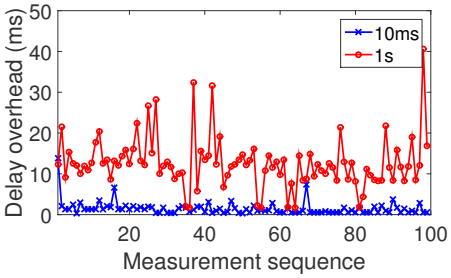


Fig. 13: Delay overheads in time series when using `ping` with packet sending intervals of 10ms and 1s.

7 Related Works

Our work shares similar analysis methodology in our previous work [25], which appraised the accuracy of browser-based measurement methods in fixed network. However, the methodology in [25] cannot be applied straightly to the mobile network measurement. We therefore design and build a new testbed to reliably capture packets in the air medium. Our evaluation results based on the three test apps developed by ourselves was previously reported in [52]. In this paper, we make several extensions: i) we summarize the implementation details for the existing measurement apps in Android; ii) we include the Internet experiment results for Ookla Speedtest and MobiPerf, showing that their RTT measurements are all inflated; iii) we investigate the performance difference between DVM and ART on network measurement by including three new Android phones with ART enabled, and iv) we perform a careful root-cause analysis in the runtime virtual machine, system layer, kernel layers, and device driver, revealing how the delay overheads are introduced.

The measurement studies based on smartphones users include [3], [53], [54], [55]. In particular, a simple logger was employed in [53] to collect the network usage information from Android and Windows Mobile users, whereas LiveLab [55] measured wireless networks in iOS. In [3] and [54], the performance of 4G LTE and 3G networks was evaluated using 4GTest and 3GTest. MobiPerf, the successor of 4GTest and 3GTest, has been deployed to uncover the RRC state dynamics in cellular networks [38], [40] and study the network performance from end users' perspectives [37], [39]. Netalyzr, another measurement app in Android, is used to characterize middlebox behavior and business relationships in cellular networks [56]. These existing apps are designed with more concern on privacy issues or 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 [57]. Batyuk et al. compared the performance between native C and Java applications for identical tasks [58], 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 [59] and found that JNI communication delays were about 0.15ms. These works focused mainly on the performance comparison of specific algorithms but do not study the relationship between system delay and network delay measurement. In [60], 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 [61], a tool based on `kprobes` was utilized to intercept system events in Android. However, these two works did not analyze the network behavior systematically, nor the device driver.

8 Conclusions

In this paper, 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 setup 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, which can reduce the user-kernel delay overhead to less than 1.5ms. In our future work, we will investigate how to mitigate the kernel-phy delay overhead on Android, as well as extend our work to cellular networks and iOS.

References

[1] A. Tongaonkar, S. Dai, A. Nucci, and D. Song, "Understanding mobile app usage patterns using in-app advertisements," in *Proc. PAM*, 2013.
[2] ITU, "ICT facts and figures 2016," http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2016.pdf.
[3] 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.
[4] Q. Xu, "Optimizing mobile application performance through network infrastructure aware adaptation," Ph.D. dissertation, University of Michigan, 2013.
[5] J. Sommers and P. Barford, "Cell vs. WiFi: On the performance of metro area mobile connections," in *Proc. ACM/USENIX IMC*, 2012.
[6] "MobiPerf on Google Play." https://play.google.com/store/apps/details?id=com.mobiperf.
[7] "Netalyzr on Google Play." https://play.google.com/store/apps/details?id=edu.berkeley.icsi.netalyzr.android.
[8] "SpeedChecker on Google Play." https://play.google.com/store/apps/details?id=uk.co.broadbandspeedchecker.
[9] "Speedtest.net on Google Play." https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest.
[10] "Speedtest X HD WiFi & Mobile Speed Test on App Store." https://itunes.apple.com/us/app/speedtest-x-hd-wifi-mobile/id366593092.
[11] "Speedtest.net on App Store." https://itunes.apple.com/us/app/speedtest.net-mobile-speed/id300704847.
[12] "Network Speed Test on Windows Store." http://www.windowsphone.com/en-us/store/app/network-speed-test/9b9ae06b-2961-41ef-987d-b09567cffe70.
[13] "Speedtest.net on Windows Store." http://www.windowsphone.com/en-us/store/app/speedtest-net/4fcd4de1-050b-44dc-b123-a786808eb49b.
[14] FCC, "Measuring fixed broadband report - 2016," https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-report-2016, 2016.
[15] Ookla, "Speedtest intelligence from Ookla — internet performance database," http://www.ookla.com/speedtest-intelligence.
[16] "Internet Speed Test," https://play.google.com/store/apps/details?id=pl.speedtest.android.
[17] "FCC Speed Test," https://play.google.com/store/apps/details?id=com.samknows.fcc.

[18] D. Wu, R. Chang, W. Li, E. Cheng, and D. Gao, "MopEye: Opportunistic monitoring of per-app mobile network performance," in *Proc. USENIX ATC*, 2017.

[19] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: A simple model and its empirical validation," in *Proc. ACM SIGCOMM*, 1998.

[20] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the TCP congestion avoidance algorithm," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 3, pp. 67–82, Jul. 1997.

[21] R. Mok, W. Li, and R. Chang, "IRate: Initial video bitrate selection system for HTTP streaming," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 6, pp. 1914–1928, June 2016.

[22] Y. Liao, W. Du, P. Geurts, and G. Leduc, "Decentralized prediction of end-to-end network performance classes," in *Proc. ACM CoNEXT*, 2011.

[23] F. Ahmed, J. Erman, Z. Ge, A. X. Liu, J. Wang, and H. Yan, "Detecting and localizing end-to-end performance degradation for cellular data services," in *Proc. IEEE INFOCOM*, 2016.

[24] 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.

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

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

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

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

[29] 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.

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

[31] C. Pei, Y. Zhao, G. Chen, R. Tang, Y. Mengy, 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*, 2016.

[32] 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.

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

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

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

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

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

[38] 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.

[39] A. Nikravesh, D. Choffnes, E. Katz-Bassett, Z. Mao, and M. Welsh, "Mobile network performance from user devices: A longitudinal, multi-dimensional analysis," in *Proc. PAM*, 2014.

[40] 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.

[41] D. Skordoulis, Q. Ni, H. H. Chen, A. P. Stephens, C. Liu, and A. Jamalipour, "IEEE 802.11n MAC frame aggregation mechanisms for next-generation high-throughput WLANs," *IEEE Wireless Communications*, vol. 15, no. 1, pp. 40–47, February 2008.

[42] A. Saif and M. Othman, "A reliable A-MSDU frame aggregation scheme in 802.11n wireless networks," *Procedia Computer Science*, vol. 21, pp. 191 – 198, 2013.

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

[44] "strace," http://strace.sourceforge.net/.

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

[46] W. Li, D. Wu, R. K. Chang, and R. K. Mok, "Demystifying and puncturing the inflated delay in smartphone-based WiFi network measurement," in *Proc. ACM CoNEXT*, 2016.

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

[48] E. Chan, X. Luo, and R. Chang, "A minimum-delay-difference method for mitigating cross-traffic impact on capacity measurement," in *Proc. ACM CoNEXT*, 2009.

[49] 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.

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

[51] https://play.google.com/store/apps/details?id=app.greyshirts.firewall&hl=en.

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

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

[54] 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.

[55] 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.*, vol. 38, no. 3, pp. 15–20, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925019.1925023

[56] 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.

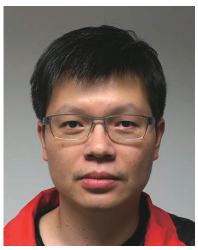
[57] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon, "Evaluation of Android Dalvik virtual machine," in *Proc. JTRES*, 2012.

[58] 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.

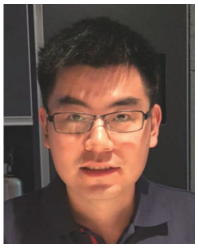
[59] S. Lee and J. W. Jeon, "Evaluating performance of Android platform using native C for embedded systems," in *Proc. IEEE ICCAS*, 2010.

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

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

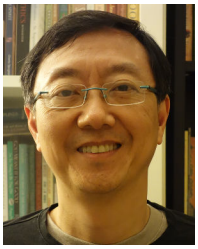


Weichao Li (S'15-M'16) received his B.E and M.E degrees from South China University of Technology, Guangzhou, China in 2002 and 2009, respectively, and Ph.D. degree in 2017 from Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China. He is currently a researcher at Huawei Future Network Theory Lab, Hong Kong. His research focuses on network measurement, especially on Internet measurement technologies, large-scale network performance monitoring and diagnosis, mobile network performance monitoring. He is also interested in areas including cloud computing, HTTP streaming system, quality of experience (QoE) measurement, and machine learning.



Daoyuan Wu (S'15) is currently a PhD candidate at School of Information Systems, Singapore Management University. He received his M.Phil degree from The Hong Kong Polytechnic University in 2015 and the B.E. degree from Nanjing University of Posts and Telecommunications in 2011. His research interests include smartphone security, vulnerability detection, and mobile network measurement.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMC.2017.2737990, IEEE Transactions on Mobile Computing



Rocky K. C. Chang (M'89) received the B.Sc. degrees from Virginia Tech, Blacksburg, VA, USA, and Vincennes University, Vincennes, IN, USA, in 1981 and 1983, respectvely, and the M.Sc. and Ph.D. degrees from the Rensselaer Polytechnic Institute, Troy, NY, USA, in 1985, 1987, and 1990, respectively. He is currently an Associate Professor with the Department of Computing, Hong Kong Polytechnic University (PolyU), Hong Kong, China. Prior to that, he received his postdoctoral training with the Computer Science Department, IBM Thomas J. Watson Research Center, Hawthorne, NY, USA. He is leading the Internet Infrastructure and Security Research Laboratory, addressing network measurement, network security, and network operations and management problems. He also leads a research group on Internet Services Monitoring and Diagnostics in the Division of Smart Cities under PolyUs Research Institute for Sustainable Urban Development.

Ricky K. P. Mok (S'11-M'14) received the B.Eng.(Hons.) degree in computer engineering from the Chinese University of Hong Kong, Hong Kong, China, in 2009, and the Ph.D. degree from the Department of Computing, Hong Kong Polytechnic University, Hong Kong, China, in 2016. He is currently serving as a Postdoctoral Scholar with the CAIDA/University of California at San Diego, San Diego, CA, USA. His research interests involve the QoE of video streaming systems, such as developing reliable QoE assessment methodologies, investigating the human factors influencing the QoE, and intergrading network path quality measurement into video streaming systems. He also conducts network measurement research, including packet send-time accuracy in networked embedded systems and delay measurement in mobile devices.