Mobilyzer: An Open Platform for Controllable Mobile Network Measurements

Ashkan Nikravesh* Hongyi Yao* Shichang Xu* David Choffnes† Z. Morley Mao*
*University of Michigan †Northeastern University
{ashnik,hyyao,xsc,zmao}@umich.edu choffnes@ccs.neu.edu

ABSTRACT

Mobile Internet availability, performance and reliability have remained stubbornly opaque since the rise of cellular data access. Conducting network measurements can give us insight into user-perceived network conditions, but doing so requires careful consideration of device state and efficient use of scarce resources. Existing approaches address these concerns in ad-hoc ways.

In this work we propose *Mobilyzer*, a platform for conducting mobile network measurement experiments in a principled manner. Our system is designed around three key principles: network measurements from mobile devices require tightly controlled access to the network interface to provide isolation; these measurements can be performed efficiently using a global view of available device resources and experiments; and distributing the platform as a library to existing apps provides the incentives and low barrier to adoption necessary for large-scale deployments. We describe our current design and implementation, and illustrate how it provides measurement isolation for applications, efficiently manages measurement experiments and enables a new class of experiments for the mobile environment.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Measurement techniques, Performance attributes]; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*

General Terms

Design, Measurement, Performance

Keywords

Measurement Tool; Cellular Networks; Network Performance; Video; Mobile Web

1. INTRODUCTION

Given the tremendous growth in cellular data traffic, it is increasingly important to improve the availability, performance and reliability of the mobile Internet. To adequately address these challenges, we ideally would be able to collect network measurement data from any mobile device on any network at any time. With this information, users could evaluate the service they are paying for, carriers could study the factors impacting performance and detect problems causing degradation, and application developers could tune and improve their service. Many of these optimizations can be performed dynamically when data are available in real time.

Despite a need for performance improvement and policy transparency in this space [27, 45], researchers currently still struggle to measure, analyze and optimize mobile networks. Mobile Internet performance characterization is a challenging problem due to a wide variety of factors that interact and jointly affect user experience. For example, the performance that a user receives from an application can depend on the device's available hardware resources, radio's network state, the network access technology used, contention for the wireless medium, distance from cell towers, session initiation overhead, congestion at various gateways and the overhead of executing code at the communicating endpoints. Further, this performance can change over time and as the user moves with the mobile device. Other challenges include resource constraints (data and power) on mobile platforms in addition to interference affecting measurements.

This problem has not gone unnoticed by researchers, operators and providers. A number of small testbeds and user studies have enabled progress in the face of these challenges [13, 16, 22, 35, 60], but with limited scope, duration, coverage and generality. We argue that previous work suffers from three key limitations that hamper their success. First, these individual solutions do not *scale*: each individual app or measurement platform is inherently limited to the population of participating users running a single piece of software. Second, each solution is *inconsistent* and *inflexible* in the set of network measurements it supports and the contextual information describing the experimental environment, making it difficult to ensure scientific rigor and to merge disparate datasets. Third, these solutions are *uncoordinated* in how they conduct network measurements: multiple apps can wastefully measure the same property independently or, worse, interfere with each other by running measurements at the same time from the same device.

Instead of proliferating apps that conduct independent network measurements in inconsistent ways, we argue that there should be a common measurement service that apps include in their code. Toward this goal, we designed and built *Mobilyzer*, a unified platform for conducting network measurements in the mobile environment. Our system is designed around three key principles:

1. **Measurement isolation:** Network measurements from mobile devices require tightly controlled access to the network

MobiSys'15, May 18–22, 2015, Florence, Italy.

http://dx.doi.org/10.1145/2742647.2742670.

interface to provide isolation from competing flows that impact measurement results.

2. **Global coordination:** Uncoordinated measurements from individual devices have limited scalability and effectiveness, so we provide a global view of available devices and their resources as a first-class service to support efficient and flexible measurements without overloading any device or network.

3. **Incentives for researchers and developers:** Instead of focusing on a single "killer app" to obtain a large user base, we distribute the platform as a library to include in any new or existing app. Developers and researchers who write apps needing network measurement benefit from reduced operational cost and coding/debugging effort when using our library, and are given network measurement resources across *all Mobilyzer* devices in proportion to the number of users they convince to install their *Mobilyzer*-enabled apps.

This paper answers key questions of how to (1) design a system that efficiently provides controllable and accurate network measurements in the mobile environment, and (2) leverage crowdsourcing to enable measurement studies previously infeasible in mobile networks.

Mobilyzer provides an API for issuing network measurements using a standard suite of tools, and manages measurement deployment and data collection from participating devices. Each device runs a measurement scheduler that receives measurement requests and gives experiments explicit control over the context in which a measurement runs. A cloud-based global manager dynamically deploys measurement experiments to devices according to available resources, device properties and prior measurement results.

Mobilyzer is currently deployed as a library that can be included in Android apps, and a cloud-based Google App Engine service for managing measurements across participating devices. *Mobilyzer* supports standard implementations of useful measurement primitives such as ping, traceroute, DNS lookups, HTTP GET, TCP throughput, and the like. These rich set of measurement primitives support experiments traditionally popular in fixed-line networks, such as mapping Internet paths (via traceroute), measuring and comparing performance for different destinations, and understanding broadband availability. We also support advanced application-layer and cellular-specific measurements, such as inferring RRC timers, measuring Video QoE, and breaking down Web page load time into its constituent dominant components.

Our design supports a constrained form of programmable measurements: an experiment may consist of sequential and/or concurrent measurements, where the execution of subsequent measurements can depend on the results of prior results and contextual information (*e.g.,* signal strength or location). We demonstrate the flexibility of these *compound measurements* using several experiments as case studies. For instance, we use this feature to trigger diagnosis measurements when anomalies in network performance are detected, or when we predict that a handover might occur.

We evaluate our deployed system in terms of our design goals. We demonstrate that it effectively provides measurement isolation, and failure to do so can lead to unpredictable and large measurement errors. Further, we show that *Mobilyzer* manages measurement scheduling efficiently, adapts to available resources in the system, is easy to use, and reduces development effort for measurement tasks.

We further evaluate our system in terms of new measurement studies that it enables, using *Mobilyzer*'s support for coordinated measurements among large numbers of vantage points to evaluate the performance of Internet-scale systems. We use *Mobilyzer* to identify and diagnose cases of CDN inefficiency, characterize and decompose page load time delays for Web browsing, and evaluate alternative video bitrate adaptation schemes in the mobile environment. For example, we find that (1) poor CDN replica selection adds 100 ms or more latency in 10% of our measurements, (2) limited mobile CPU power is a critical bottleneck in Web page load times (typically between 40-50% of load times) and doubling CPU speed can reduce load times by half, and (3) buffer-based adaptive video streaming improves the average delivered bitrate by 50% compared to commonly used capacity-based adaptation for low bandwidth clients.

2. BACKGROUND AND RELATED WORK

A variety of existing approaches attempt to shed light on mobile networks, but each exhibits limitations that we address with our work.

Existing research platforms. Our work shares many of the same goals of successful testbeds such as PlanetLab [47], RIPE Atlas [52], and BISMark [63], and our work uses M-Lab [38] servers as targets for many measurement tests. These are general-purpose experimentation platforms that require deployment of infrastructure and do not operate in the mobile environment. These systems are insufficient alone for understanding mobile networks because mobile networks generally use firewall/NATs [66]. We also share goals with Dasu [55], but differ fundamentally in that properly characterizing network performance in mobile environment requires different strategies than in the wired/broadband environment due to scarce resources and measurement interference. Seattle [20] is a general-purpose platform that supports deploying code on Android devices, and it shares many goals with *Mobilyzer*. It does not provide network measurement isolation, but many of its device management and security features can be integrated into *Mobilyzer*.

View from operators. Operators and manufacturers such as AT&T and Alcatel-Lucent have deployed in-network monitoring devices that passively capture a detailed view of network flows traversing their mobile infrastructure [18, 24]. Several studies use these passive measurements to understand network traffic generated by subscriber devices, with important applications to traffic engineering, performance characterization and security [18, 28, 37, 49, 53, 64, 68]. However, this approach does not allow the carrier to understand the performance experienced at the mobile device. For example, when the throughput for a network flow suddenly changes, it is difficult to infer from passive measurements alone whether it is due to wireless congestion, signal strength, and/or simply normal application behavior.

Further, most interesting behavior occurs at or near the edge of mobile networks, making an infrastructure-based deployment costly. For example, Xu et al. point out that monitors located near a cellular provider's core cannot account for detailed user mobility, and adding the infrastructure to support these measurements would require a deployment at least two orders of magnitude larger [67].

Existing measurement apps. Motivating the need for a mobile measurement library, several academic, governmental and commercial mobile performance measurement tools have been released recently. The FCC released a network measurement app [27] to characterize mobile broadband in the US but the system is closed and data is not publicly available. Several commercial apps measure mobile Internet performance by collecting measurements such as ping latency and throughput [21,44,61]. However, because the source code is closed, the methodology is under-specified,

and the datasets tightly controlled, it is difficult for the research community and end-users to draw sound conclusions. Further, these tests are generated on-demand by end-users, creating a strong selection bias in the data, as users may be more likely to run tests when they experience problems.

A number of research projects use apps to gather network measurements from hundreds or thousands of mobile devices [31, 36, 40, 41, 48, 57–59]. Such measurements reveal information not visible from passive in-network measurements, such as radio state, signal strength and the interaction between content providers and network infrastructure [34,59,67,69]. Further, these measurements reveal information across carriers, allowing researchers to understand different ISP network policies and performance [66]. Such one-off studies provide a useful snapshot of mobile networks, but they do not support longitudinal studies that inform how mobile network performance, reliability and policy changes over time, nor do they support targeted measurements that are required to understand the underlying reasons for observed network behavior.

Each of the above projects uses a separate codebase, a different set of collected contextual information and different privacy/data sharing policies. As such, experiments are conducted in an ad-hoc manner, with limited or no ability to compare and combine measurements from each limited deployment. As an example pitfall from the current approach, consider the case of interpreting the result of a simple ping measurement. Because mobile device radios enter a low power state during idle periods, a ping measurement that wakes the radio (and experiences additional delay during the wakeup period) can affect a subsequent ping measurement that occurs while the radio is in full power mode. Without contextual information, a researcher might falsely attribute network delays to the carrier instead of the device. Worse, if there are two measurement apps running on the same device, one app's measurements can interfere with the other's.

Our work is also motivated by other efforts in building measurement libraries, especially on mobile platforms. For example, Insight [46] and AppInsight [51] offer platforms for instrumenting and measuring application performance, rather than network measurements.

3. GOALS

The goal of *Mobilyzer* is to provide a scalable platform for conducting meaningful network measurements in the mobile environment. This is the first open-source platform[1] to support mobile measurements in the form of a library. It is designed to achieve:

3.1 Standard, Easy-to-use Measurements

There is a tension between arbitrary flexibility and standardization in a platform for network measurements. While allowing experimenters to execute arbitrary code within a sandbox provides flexibility, a lack of standards means that these experiments are difficult to incorporate into existing and future datasets. In *Mobilyzer*, we opt for standard measurements, facilitating comparative and longitudinal studies atop our platform (see §4.2).

3.2 Measurement Isolation

The mobile environment poses unique challenges for controlling how network measurements are isolated from each other, and from network traffic generated by other apps and services. *Mobilyzer* accounts for these properties and gives experimenters explicit control over device state and concurrent network activity during measurement. *Mobilyzer* achieves the goal of measurement isolation via a local measurement scheduler (see §4.3).

[1] *Mobilyzer* source code is publicly available at http://mobilyzer-project.mobi/

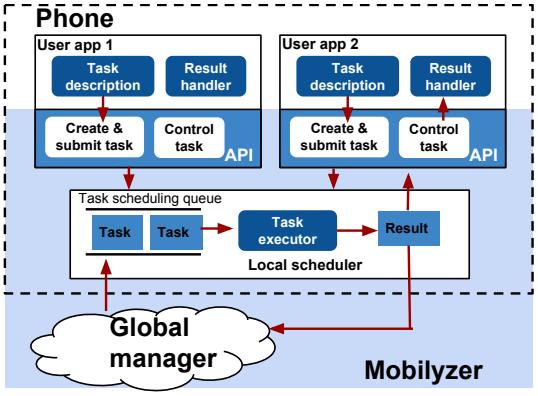


Figure 1: *Mobilyzer* architecture (shaded region). Apps (top) include the *Mobilyzer* library, which provides an API to issue network measurements, and a scheduler service (middle) that provides measurement management and isolation. The scheduler communicates with a global manager (bottom) to fetch measurement requests and report measurement results.

3.3 Global Coordination

Data quota and power are scarce resources for mobile networks, requiring that researchers use more principled approaches than "shotgun" measurement. With a global view of available network resources and the ability to coordinate measurements from multiple devices, we can coalesce redundant measurements and allow researchers to specify targeted experiments that use device resources efficiently (see §4.4). For instance, coordination among the *Mobilyzer*'s clients provides opportunities to (1) evaluate the performance of Internet-scale services and (2) schedule measurements in an interactive fashion, where a set of measurements scheduled on one device may depend on the results *Mobilyzer* receives from other devices.

3.4 Incentives for Adoption

As a crowdsourcing approach, *Mobilyzer* requires a crowd of mobile users to adopt our platform. We argue that no single app will provide the necessary crowd. Instead, we opt for a "bring your own app" deployment model, where researchers/developers develop *Mobilyzer*-enabled apps with user incentives.

We propose the following incentives for researchers and developers to use our library.

- *Incentive for researchers.* The incentive for researchers to develop *Mobilyzer*-enabled apps is that they can conduct measurements on *any* of the devices in the system (including those not running their app), with measurement quota proportional to the number of devices their app(s) bring to the system. This is analogous to the PlanetLab and RIPE Atlas models, except using a software (instead of hardware) deployment.
- *Incentive for developers.* One incentive for adopting *Mobilyzer* is reduced operational costs. As an example, an app that provides "speed tests" of peak transfer rates can incur significant bandwidth charges for the servers hosting transferred content. Our system uses resources donated by M-Lab, and this was the main reason that MySpeedTest adopted our tool.

  A second incentive is reduced coding/debugging effort. Supporting any app that requires network measurement (and pos-

Measurement Type	Supported Measurements	Usage
Basic	DNS lookups, ping, traceroute, HTTP GET, TCP throughput, and UDP burst	Supports experiments traditionally popular in fixed-line networks, such as mapping Internet paths, measuring and comparing performance for different destinations and carriers, and understanding broadband availability
Composed	Sequential and parallel	Combines multiple measurements programmatically to support new measurements, such as diagnosis measurements
Complex	RRC timer inference, Video QoE, and Page load time measurements	Supports application-layer and cellular-specific active measurements

Table 1: Measurement types supported by *Mobilyzer*.

sibly access to measurements from multiple devices) entails writing and maintaining software for measurement and data storage/retrieval. Our experience is that correct implementations of such functionality can take man-months to man-years of work. We provide this functionality out-of-the-box with *Mobilyzer*.

We also consider altruism and relationships with developers as incentives. For example, the Insight [46] project successfully convinced developers of a popular app to instrument and share data with researchers. Similarly, altruism has attracted users to support platforms such as RIPE Atlas and DIMES.

Comparison with ad libraries. Our library-based deployment model is inspired in part from the successful adoption of advertising (ad) libraries in mobile apps. We believe that ad libraries are successful in large part because they have direct incentives for developers (payment for click-throughs) and a limited impact on apps. Our direct incentive for developers is that they "earn" measurement resources proportional to the size of the *Mobilyzer*-enabled app deployment. This is analogous to the RIPE Atlas credit model, which has helped grow a hardware-based platform to more than 7,000 active hosts. Like ad libraries, which periodically pull advertisements for display, *Mobilyzer* primarily uses a pull-based model to fetch measurements.

However, ad libraries and *Mobilyzer* differ from the perspective of privacy and resource usage. Ad libraries can be used to track arbitrary information about users, often without any explicit privacy policy or user awareness. *Mobilyzer* is explicit about the data it collects, was designed with privacy concerns in mind, and requires explicit user consent. Further, it is unclear if or how ad libraries limit the resources consumed by their service. *Mobilyzer* provides controllable, limited impact on apps by enforcing hard limits on resource consumption (in terms of energy and data quota) and avoiding interference with network traffic from other apps. While *Mobilyzer* provides direct incentives for developers in terms of measurement resources, unlike ad libraries it will not directly subsidize app development costs.

3.5 Nongoals

First, we do not provide a "PlanetLab for mobile"; i.e., we do not provide a platform for deploying arbitrary distributed system code on mobile devices. Instead, we focus on the more constrained problem of providing a platform for principled network measurements in lieu of one-off, nonstandard measurement studies. Second, we do not provide a service for arbitrary data-collection from mobile devices; rather, we focus on active and passive network measurements annotated with anonymized contextual information. Collecting arbitrary data from users is a potential privacy risk that we avoid in *Mobilyzer*. Third, we do not propose any specific incentives for device owners to adopt *Mobilyzer*-enabled apps. Rather, we urge experimenters to either develop apps with user incentives in mind, or convince maintainers of existing popular app codebases to include our library. Currently, there are hundreds of apps [6] that simply do network measurements (e.g., speed tests) or network diagnosis (signal strength/coverage maps) and have large user bases. We believe this is strong evidence that users will install a *Mobilyzer*-enabled tool as long as it is useful. We also believe that *Mobilyzer*'s limited data collection is not a strong impediment to user adoption, as ad libraries are known to collect more intrusive information with no known effect on adoption.

4. DESIGN AND IMPLEMENTATION

4.1 Overview

Figure 1 provides a high-level view of the Mobilyzer platform. It consists of an app library, a local measurement scheduler, and a cloud-based global manager.

Measurement Library. *Mobilyzer* is designed to be easy to use and integrate into existing apps, while providing incentives for doing so. It is deployed to apps as a library with a well-defined API for issuing/controlling measurements and gathering results locally. This facilitates development of new apps that use network measurements. For example, an app that includes our library can issue a measurement and retrieve results with under 10 lines of code. Existing apps can incorporate our library to take advantage of our validated measurement tools and server-side infrastructure.

Local measurement scheduler. *Mobilyzer* achieves the goal of measurement isolation via the local measurement scheduler. The scheduler listens for measurement requests and ensures that any submitted measurements are conducted in the intended environment (*e.g.,* radio state, state of other apps using the network, and contextual information such as location, signal strength and carrier). Regardless of how many apps on a device use it, *Mobilyzer* ensures that there is exactly one scheduler running. If apps with multiple versions of the scheduler are present, the latest version is guaranteed to be used. The scheduler enforces user-specified limits on resource (data/power) consumption.

Global manager. The *Mobilyzer* global manager provides coordination for the following services: dynamic measurement deployment, resource cap enforcement, and data collection. This support is quite unique to our platform, as global centralized coordination improves the design of mobile experiments despite limited resources. Experimenters use the global manager to deploy measurement experiments to devices. For example, an experiment may contain a number of network measurements that should be conditionally deployed and executed according to device properties. The global manager deploys these measurements to devices based on device information reported periodically, and ensures that measurements are not deployed to devices that have exceeded user-specified resource caps. The manager maintains a datastore of anonymized data collected from all devices and

API Function	Description
`MeasurementTask` **`createTask`**`(TaskType type, Date startTime, Date endTime, double intervalSec, long count, long priority, Map<String,String> params)`	Create a task with the specified input parameters including task execution frequency task priority, etc.
`String` **`submitTask`**`( MeasurementTask task )`	Submit the task to the scheduler and return `taskId`.
`void` **`cancelTask`**`(String taskId)`	Cancel the submitted task, only allowed by the application that created this task.
`void` **`setBatteryThreshold`**`(int threshold)`	Set the battery threshold for check-in and running the global manager scheduled tasks.
`void` **`setCheckinInterval`**`(long interval)`	Set how frequently the scheduler checks in.
`void` **`setDataUsage`**`(DataUsageProfile profile)`	Set a limit for *Mobilyzer*'s cellular data usage.

Table 2: *Mobilyzer* key API functions to support issuing and controlling measurement tasks.

makes this available for interactive measurement experiments and for offline analysis. The data collection and scheduling support at the global manager enable dynamically triggered measurements based on observed network behavior, achieving a feedback loop of measurement, analysis, and updated measurements.

4.2 Measurement Library and API

Design. A key design principle for *Mobilyzer* is that the platform should remove barriers to widespread adoption so that it facilitates a large-scale deployment. We chose to implement *Mobilyzer* as a network measurement library that app developers include in their code. This code provides an API for issuing measurements using a standard suite of tools (Table 2), and a device-wide measurement scheduler.

The measurement model supported by *Mobilyzer* represents a trade-off between complete flexibility to run arbitrary code and complete safety in that all resources consumed by a measurement can be predicted in advance. Specifically, *Mobilyzer* supports a small set of commonly used *measurement primitives*, e.g., ping, traceroute, DNS lookup, HTTP get and throughput tests. These measurements can be performed independently in isolation, or they can be chained (do a DNS lookup for google.com, then ping the IP address) or executed in parallel (e.g., ping google.com during a TCP throughput test) in arbitrary ways.

Using this model, the amount of data and power consumed by each task (simple or complex) is predictable with few exceptions (e.g., downloading an arbitrary Web page) that can be mitigated via scheduler-enforced constraints on the data and power consumption of a task. Like Dasu, this approach avoids concerns from running arbitrary code; however, unlike Dasu this allows us to strictly control resource consumption (which is not a primary goal of Dasu).

Implementation. The *Mobilyzer* measurement library is implemented as Android code that is added by reference to an existing app's source code. *Mobilyzer* supports three types of measurements: *Basic*, *Composed*, and *Complex Measurements (Table 1).*

Basic Measurements Supported. Mobilyzer supports both passive, contextual measurements and active network measurements. Currently, the active measurements supported are DNS lookups, ping, traceroute, HTTP GET, and a TCP throughput and UDP burst test. For each measurement task, users can specify general task parameters (*e.g.,* the time at which to run) and task specific parameters (*e.g.,* TTL value for ping measurement). Further, each task can specify a *pre-condition* which must evaluate to true before a task can be executed (*e.g.,* location is Boston, signal strength is greater than 50, network type is cellular).

Mobilyzer also collects passive measurements on both network performance and device state, and associates a set of passive measurements with every active test run. It collects the signal strength (RSSI values) and the device's battery level, battery charging state, coarse-grained location, the network technology, the total number of bytes and packets sent and received by the device, as well as static context information such as the carrier, OS, and support for IPv6. Our set of measurements and monitored device properties do not require special privileges; however, we can support measurements that require rooted phones (if available).

Measurement composition. Mobilyzer supports combining multiple measurements into a single experiment, where sets of measurement tasks can run sequentially and/or in parallel. Further, the result of each sequential task can be used as a pre-condition for executing a subsequent task. This feature improves experiment flexibility—we use this feature to trigger diagnosis measurements in response to detecting anomalies in network performance. Figure 2 shows how this feature can help in building a simple diagnosis task, where the client will run traceroute, if latency and signal strength is above a threshold.

Supporting complex measurements. Mobilyzer is designed to provide extensible support for application-layer and cellular-specific measurements. These include inferring RRC timers, measuring Video QoE, and breaking down Web page load time into its constituent dominant components. Further, our system can incorporate new complex measurements as they become available. Each new task must have predictable power consumption, data usage, and duration, so that *Mobilyzer* can ensure that the measurement task will not exceed a device's battery and data usage limits.

We now describe how long-running, complex tasks motivate the need for additional scheduler features to ensure measurements complete successfully. Mobile devices change between different power states, called RRC states, in response to network traffic [26, 33]. The device's RRC state is not exposed to the OS due to a lack of an open API, so identifying how long devices stay in each power state entails sending a large number of individual packets with long gaps between them, and inferring power states based on observed packet delivery delays.

This type of measurement poses two key challenges. First, it is a long-running, low-rate measurement that should run only when connected to a cellular network. With no other tasks running on the device, the test can easily take half an hour. Given our pre-emptive priority scheduler (described in the next section), the RRC inference task will either block other tasks for long durations or will be constantly interrupted by higher priority tasks. The latter situation can lead to a large volume of wasted measurement bandwidth (as interrupted measurements must be discarded and restarted), and with sufficient interruptions the RRC task may not have a chance to complete.

changing the radio power state. To detect whether the network isolation property holds, *Mobilyzer* collects information about device-wide background traffic through the *proc* file system and discards results taken when there is background traffic.

These challenges motivate the following approach for the RRC state task. First, it runs with low priority so it does not block other measurements for unreasonable periods. Second, because it may be frequently pre-empted or unable to run due to the device using WiFi, we support suspending and resuming this task. Third, we upload partial measurement results when the task fails to complete, so the data collected is not wasted. We used this measurement to understand the impact of RRC states on latency and packet loss for various network protocols and applications, and found the presence of unexpected application-layer delays [54].

More generally, we include these features for any task that logically supports suspend, resume and partial measurements. For example, traceroute includes these features but atomic measurements such as ping do not.

4.3 Local Measurement Scheduler

Design. The local measurement scheduler is a service running on a device that manages the execution of measurement tasks. It can be implemented in the operating system or run as a user-level background service. The scheduler provides a measurement execution environment that can enforce network isolation from other measurements and apps running on the device. It also enforces resource-usage, priority, and fairness constraints.

Task priorities. The scheduler supports measurement task priorities with pre-emption. It executes tasks with the highest priority first and will pre-empt ongoing measurements if they are of lower priority. Certain measurement tasks cannot produce correct results if pre-empted (*e.g.,* a ping task pre-empted between sending an echo request and receiving a response), so pre-emption will lead to wasted measurement. To balance the trade-off between this waste and scheduler responsiveness to high-priority tasks, the scheduler waits for a short time (*e.g.,* one second) before pre-empting a measurement.

This is sufficient time for a ping measurement to complete, but not necessarily for other tasks (*e.g.,* traceroute). To minimize wasted measurement resources for pre-empted tasks, we define a `pause()` method that signals to measurement tasks that they should save their state due to an imminent pre-emption. For traceroute, this means that the traceroute measurement can save its current progress and continue to measure a path once it is rescheduled.

Not all measurements can (or should) be pre-emptible. In addition to the traceroute task mentioned above, we have also implemented a pre-emptible long-running task for inferring RRC state timers on mobile devices [26, 33]. Measurements like DNS lookup and ping are not pre-emptible (i.e., they are killed at pre-emption time) because they are short-lived tasks with minimal (or no) state to cache. In general, network measurement tasks (e.g., throughput measurement) to characterize time-varying properties of the network cannot benefit from saved state when pre-empted; this is in contrast to measurements of more stable properties (e.g., RRC state machine and NAT policies).

Traffic Interference. To avoid inference from (and with) traffic from other apps, *Mobilyzer* monitors traffic generated by other apps (using the TrafficStats API in Android) and pauses/kills measurement tasks that are subject to interference. For example, the TCP throughput task, which attempts to saturate bandwidth to measure throughput, will be stopped if scheduler detects any concurrent network traffic; failing to do so may adversely affect other applications' performance and lead to inconclusive measurement results.

Resource constraints and fairness. The scheduler accounts for how much data and power is consumed by network measurements and ensures that they stay within user-specified limits. Users specify hard limits on how much data *Mobilyzer* consume, and at what battery level the library should stop conducting measurements.

We enforce limits as follows.

- The scheduler does not execute a task if its data or energy consumption estimates exceed available resources. Such tasks are suspended until the constrained resource(s) is replenished. For energy, this happens when a device is recharged; for data plans, this is typically done according to the billing cycle.
- The scheduler monitors data and power consumption for ongoing measurements, and terminates a task if it consumes more data than expected, and/or exceeds a device's data/power quota. Because measurement tasks cannot execute arbitrary code, we can always predict in advance the maximum resources they may consume, and use that as a conservative estimate for scheduling purposes.
- To ensure liveness and fair access to resources, the scheduler enforces an upper bound on task duration by terminating execution of a measurement task exceeding that limit. This limit must be specified in the task description.

All measurement tasks generated by apps running on a device are assigned the same high priority value. To ensure fairness among multiple apps on the same device competing for resources, we ensure that each app receives an equal share of the constrained resource.

Implementation. The scheduler is currently implemented using an Android service, which guarantees that at most one scheduler is present regardless of the number of *Mobilyzer*-enabled apps running on a device. The library communicates with the scheduler service via IPC calls.

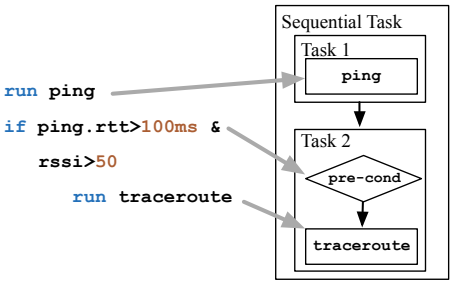
Forward compatibility. Our library-based deployment model means that different *Mobilyzer*-enabled apps running on the same device may use different versions of the measurement library and scheduler. We want to ensure that all apps on the device bind to the *newest* version of the scheduler service as soon as it is installed by an app. A key challenge for providing this functionality is that by default on Android, apps will bind only to the scheduler of the app that is first installed, which is unlikely to be the latest version.

We address this as follows. First, we exploit the Android priority tag in the *bind* Intent when declaring the scheduler in the manifest file. The scheduler with the highest priority is bound if there is no other bound scheduler, so we increment the priority value with each new *Mobilyzer* version. However, if an older scheduler has already been bound, apps bound to an older version do not switch until the device is rebooted. To address this, we deliver the scheduler version information via the *start* Intent. When the scheduler receives a *start* Intent, it compares its version with the one in the intent and terminates if it sees a newer scheduler version.

4.4 Global Manager

Design. The *Mobilyzer* manager maintains a global view of measurement resources, efficiently dispatches experiment tasks to available devices and coordinates measurements from multiple devices to meet various experiment dependencies. Atop the manager is a Web interface that allows experimenters to specify measurement experiments that are deployed to devices.

Specifically, researchers can submit a measurement schedule, along with information about the properties of devices that should participate in the experiment (e.g., location, device type, mobile provider). Devices periodically check in with the manager and




```
params = new HashMap<String, String>();
params.put("target", "www.google.com");
MeasurementTask pingTask = api.createTask(PingTask.TYPE,
    params);
MeasurementTask tracerouteTask =
    api.createTask(TracerouteTask.TYPE, params);
tracerouteTask.setPreCondition(AndCriteria(
      Criteria(Criteria.RESULT, 0 ,"mean_rtt_ms",">",100),
      //0 is index of prior task
      Criteria(Criteria.RSSI,">",50)));
MeasurementTask seqTask = api.createTask(SequentialTask.TYPE,
    Config.DEFAULT_MEASUREMENT_COUNT,
    Config.DEFAULT_MEASUREMENT_INTERVAL_SEC,
    API.USER_PRIORITY);
seqTask.setSubTasks([pingTask,tracerouteTask])
api.submitTask(seqTask);
```

Figure 2: Using sequential tasks to build a simple diagnosis task.

receive a list of experiments to run. When complete, devices report the results of the measurements to the manager.

The manager ensures: (i) no experiment uses more than its fair share of available measurement capacity across *Mobilyzer* devices; (ii) experiments are scheduled in a way that maximizes the likelihood that the targeted devices will complete their measurements during the period of interest; and (iii) it enforces limits that prevent harm to the network or hosts (*e.g.,* via a DDoS).

Fairness under contention. In a successful *Mobilyzer* deployment, there are likely to be cases where measurement requests exceed available measurement resources. In periods of contention, we must dispatch measurement tasks according to some fairness metric.

We assign measurement tasks to devices such that the fraction of assigned tasks for an experimenter is proportional to the number of measurement-enabled devices that the experimenter has contributed to *Mobilyzer*. When measurement demand from an experimenter is lower than their fair share, we backfill available resources from remaining pending tasks to ensure that our system is work conserving. Similar to previous work [19, 42], we can use an auction-based approach as a building block to provide such fairness.

Availability prediction. Available devices in *Mobilyzer* are subject to high churn and mobility, meaning that simply deploying an experiment to some random fraction of devices may lead to a low rate of successful measurements. To improve this, the global manager incorporates knowledge of experiment dependencies, prediction for resource availability and accounting for failures due to issues such as disconnections and loss of power. For example, we use prediction to prevent wasted measurement resources by preventing experiments from being dispatched if they are likely to fail (*e.g.,* due to unavailable measurement quota or due to measurement preconditions not being met on the device).

Other features. *Mobilyzer* supports a variety of other features, described in detail in the technical report [43]. Briefly, we support live measurement experiments, which allows researchers to specify measurement experiments that are driven by results from ongoing measurements; i.e., experiments that cannot be specified a priori (demonstrated in §5.4.1). The global manager can prevent harm to the network and to other hosts by accounting for all measurement activity in the system and ensure that no host or network is overloaded by measurement traffic. Last, users can access their data by logging into our dashboard [7], and collected data is anonymized and publicly accessible online [8].

Implementation. The *Mobilyzer* manager is currently implemented using a Google App Engine (GAE) instance, providing a highly scalable platform for managing thousands or millions of devices. The manager currently uses a pull model for experiment deployment, where devices check in with the manager periodically to retrieve updated experiments and to update their contextual information (coarse location, remaining battery, access technology). We also support Google Cloud Messaging (GCM) services to provide a push-based model for experiments with tighter timing constraints, which is subject to the same resource constraints as pull-based measurements.

Experimenters currently can specify simple measurement schedules using an interface that permits control over measurement parameters, experiment durations and periodicity. We have also developed several dynamic measurement experiments as described in Section 5.4, where the measurements issued to a device depend on the results of prior measurements (from potentially many other devices).

The manager deploys measurements based on contextual information gathered from devices. For example, the manager will reduce the measurement rate for periodic tasks that would otherwise exceed device quota. Similarly, measurement tasks for devices in a specific geographic region are not deployed to devices outside that region.

4.5 Security

Security is paramount in any large, controllable distributed system. Our design addresses the following threat model: an attacker who subverts the system by taking over our control channel, hacking our global manager, participating in the system to launch as DDoS, or control/drain resources on devices. We address these attacks as follows:

- **Subverting the control channel:** We use Google App Engine (GAE) to control our devices, which relies on HTTPS to communicate with the trusted GAE hosts. Subverting the control channel requires subverting the PKI for Google certificates, which is difficult.
- **Subverting the GAE controller:** We use Google account authentication to control access to our global manager and limit access only to authorized users. This does not address account compromise, which we cannot rule out. However, we monitor system usage, which should allow us to react quickly should this occur.
- **Insider attack:** A malicious developer could try to use our system to launch "measurements" that actually consist of a DDoS attack or resource drain. To prevent the former, we account for all measurement requests scheduled either by check-in or push-

Measurement App	Measurement LoC
FCC SpeedTest [27]	12550
MySpeedTest [40]	9545
Mobiperf	8976

Table 3: Lines of code (LoC) for open-source measurement apps. By integrating *Mobilyzer* into existing apps, developers can save thousands of lines of code.

based mechanism and conducted by all participating devices, and place limits on how many measurements can target a given domain or network. To prevent the latter, we use the hard limits on resource constraints mentioned above.

- **App compromise:** An attacker may wish to take control of apps/devices by executing code that compromises the app/device OS. We support executing only measurements we validate, limiting the attack surface for compromise. We further rely on the Android/Java sandboxes to prevent OS compromise.

5. EVALUATION

We now evaluate *Mobilyzer* in terms of deployment experience, performance, and applications.

5.1 Deployment Experience

One key advantage of *Mobilyzer* is that apps requiring network measurements are easier to write and maintain. Our platform provides validated measurement code, prevents interference from other *Mobilyzer*-enabled apps, collects and stores measurement data and utilizes existing infrastructure (via M-Lab) for bandwidth testing. We believe that by separating measurement management and data collection code from app development, *Mobilyzer* helps researchers focus on interesting measurement experiments and compelling incentives for user adoption.

Table 3 lists the lines of code (LoC) used for network measurements in three popular measurement apps. In Mobiperf, 80% of the code was for measurement. Using a library-based model, we simplify the app codebase, making it easier to focus on UI and other elements to encourage user adoption. With *Mobilyzer* one can issue a measurement task and retrieve the results with fewer than 10 LoC.

Mobilyzer has been adopted by the developers of MySpeedTest, a throughput-testing app. The experience from the main developer was overall quite positive, and identified cases where instructions for use were unclear. While it took several e-mail exchanges to clarify these issues, the developer reported that writing the code to port MySpeedTest to *Mobilyzer* took "about an hour or less" and they have used our library since May, 2014. We are currently in discussions with other researchers about integrating *Mobilyzer* into their apps.

5.2 Measurement Isolation

An important feature of *Mobilyzer* is that it schedules measurements to provide applications with control over if and when measurements are run in isolation. We now use this feature to demonstrate the value that isolation brings. In particular, we use a compound measurement task that consists of a TCP throughput test run in parallel with a ping measurement. We vary the start time for the ping measurement such that it occurs before, during, and after the throughput test and plot the results. We also control whether the cellular radio is in a low-power state before measurement. Each experiment is repeated 40 times; we plot a CDF of ping latencies for each configuration. We omit throughput results because they are unaffected by ping measurement cross traffic.

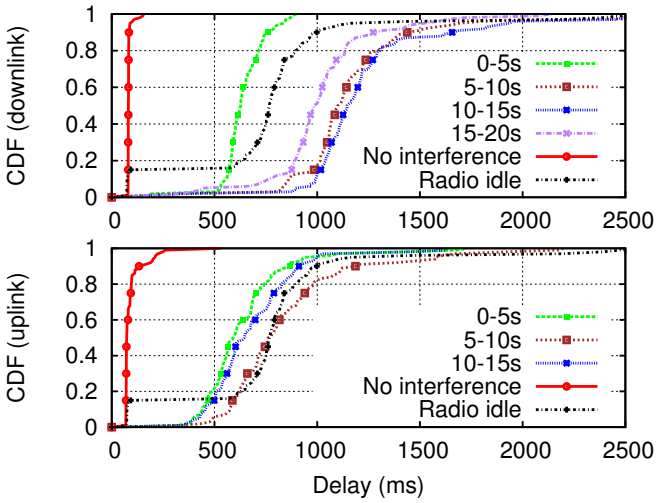


Figure 3: Impact of throughput measurements (grouped into time periods) and radio state on measured RTT. Results vary with radio state, and upstream or downstream cross-traffic with varying throughput. *Mobilyzer* provides strict control over these kinds of dependencies.

Figure 3 shows that ping measurements are significantly affected by cross traffic, and the difference in latency compared to the case with no interference can be hundreds of milliseconds or seconds. The additional delay, presumably from queuing behind the TCP flow from the throughput test, also varies depending on how much cross traffic occurs. As a result, interference from cross traffic essentially renders the latency measurement meaningless. Note that the downlink interference is more severe likely due to higher throughput (2.5 Mbps down vs. 0.35 Mbps uplink), which leads to more queuing inside the network and on the device.

The figures also show differences in measured latencies when the radio is active compared to when it is idle before measurement. Similar to the above scenario, not accounting for this effect can cause misleading or incorrect conclusions. With *Mobilyzer*, experimenters can tightly control the impact of these sources of noise in measurement experiments.

5.3 Microbenchmarks

This section presents results from controlled experiments evaluating *Mobilyzer* overhead in terms of measurement-scheduling delay, power usage, and data consumption.

5.3.1 Scheduling Delays

The scheduling delay introduced by *Mobilyzer* consists of the delay from using IPC (interprocess communication) between an app and the scheduler, and the delay introduced by the scheduler. As we show, these delays are reasonably low for all measurements and under significant load.

The IPC delay (Fig. 4) is the sum of (i) the delay between when the client submits a task and when the server receives it (IPC part I in the figure), and (ii) the delay between when the scheduler sends the result to the client, and the client receives it (IPC part II).

We use an HTC One (Android 4.1.1, using LTE) to run measurement tasks for characterizing the IPC overhead. Each task is run 100 times. Fig. 6 presents a CDF of the delay; for all tasks, the maximum delay is below 100ms, while most delays are within 20ms. These values match the performance analysis for IPC latency using Intents [30]. Note that this delay affects the time until a measurement runs but does not interfere with measurement execution.

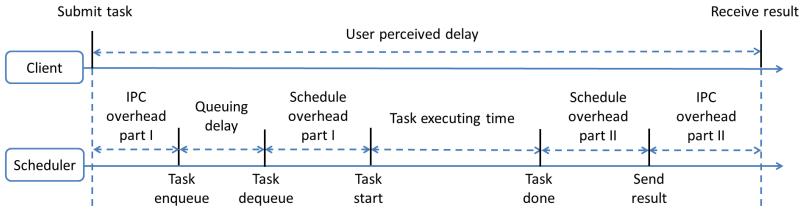


Figure 4: Scheduling overhead for a *Mobilyzer* measurement task.

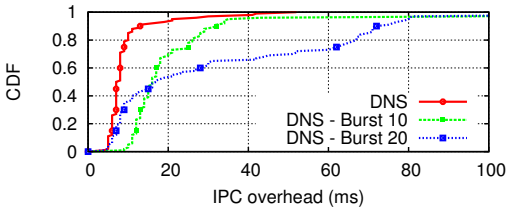


Figure 5: IPC overhead for DNS burst sizes.

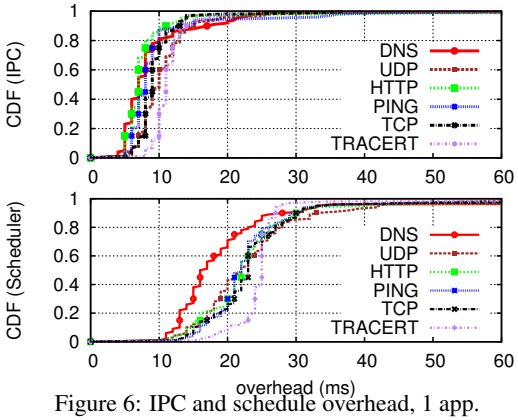


Figure 6: IPC and schedule overhead, 1 app.

Task	Power consumption under WiFi (mAh)	Power consumption under LTE (mAh)
DNS	0.03 (0.01)	0.09 (0.01)
HTTP	0.05 (0.01)	0.12 (0.01)
UDP (Down)	0.06 (0.02)	0.17 (0.02)
UDP (Up)	0.08 (0.03)	0.17 (0.04)
PING	0.21 (0.01)	0.52 (0.02)
TCP (Down)	0.11 (0.28)	2.88 (0.24)
TCP (Up)	1.33 (0.04)	2.36 (0.20)
Traceroute	0.75 (0.25)	2.34 (0.02)
Sum(unbatched)	3.51 (0.38)	8.66 (0.32)
Sum(batched)	3.75 (0.61)	5.06 (0.25)

Table 4: Power consumption of *Mobilyzer*: each cell lists the average, then the standard deviation in parentheses.

Note that the IPC overhead increases when many tasks are submitted back to back, due to the way Android implements Intent-based IPC. Specifically, there are two IPCs for each Intent-based IPC – one from sender to the Intent manager, and then one from the manager to the receiver [30].

To test the impact of this, we submit bursts of 1, 10 and 20 tasks with 15 ms between each burst. If several IPCs are sent to the Intent manager at once, there may be a delay in redirecting them to the receiving app. Fig. 5 shows the IPC overhead under high load for the DNS task (other tasks exhibit a similar pattern). Although larger burst sizes lead to longer delays, with a burst of 10 the delay is mostly within 50 ms. For a burst of 20, over 90% are below 100 ms, which we believe is acceptable for scheduling measurement tasks. It is unlikely that user-generated activity will create such a large burst of measurements, so we do not expect these delays to affect user-perceived responsiveness.

We similarly measure the scheduling delay introduced by *Mobilyzer*, *i.e.,* the delays in starting the measurement and sending the result (schedule overhead part I and II in Fig. 4). The scheduling delay distribution for a single client is shown in Fig. 6. It is slightly higher than the IPC overhead, but still on the order of tens of milliseconds. Again, we believe this is acceptable for scheduling and reporting results from measurement tasks.

5.3.2 *Power Usage*

We now estimate the power consumed by *Mobilyzer*. In general, it is difficult to distinguish power consumption of our library from the application that runs it. To address this, we measure the power consumption of an app using the *Mobilyzer* service as it runs in the background executing server scheduled measurement tasks and compare it to power consumed by the same app before integrating *Mobilyzer*, with the overall functionality kept the same.

We measure the power consumption of measurement tasks using a Samsung Galaxy Note 3 as the test device, and a Monsoon power monitor [9] to measure the device power usage.We run each task 5 times with a 15-second delay between tasks to determine the power drain in isolation (unbatched). All experiments are repeated 3 times to identify possible outliers, and we record the average power consumption. Additionally, we run a batched task consisting of all 8 tasks 5 times, and measure the power consumption during the entire measurement period to better reflect that most tasks in *Mobilyzer* run in batches. For comparison, we also sum the average power consumption for all 8 tasks, denoted as Sum(unbatched) based on individual experiments. Table 4 shows the results for WiFi and LTE.

Of the unbatched experiments, we can see that the TCP throughput test has the highest power consumption. This is consistent with prior results showing that power drain is roughly linear with the throughput and duration that the network interface is in a high power state [33]. The power drain from remaining tasks is proportional to the measurement duration because their throughput is low. Note that traceroute has higher power consumption under LTE than under WiFi, since it sends a large number of ICMP packets with an interval of roughly 0.5s. While on LTE, the device will stay in the high power state between these packets.

For batched tasks, we found that on WiFi the power consumption for batched and unbatched tasks are similar. However, on LTE the total power consumption of the batched measurements is much smaller than that of the sum of the individual measurements, by 41.5%. This is because of the tail energy effect on LTE, where the device remains in a high power radio state for several seconds after a network transmission. In each individual task, there is a contribution from the tail time of an average of 48.2% to the total energy consumed [33]. When the tasks are batched, there is only one tail time, demonstrating the effect of batching in saving energy in cellular networks.

These microbenchmarks identify the energy for individual measurements, but not the relative cost compared to other services running on devices. We evaluate this using the built-in Android battery consumption interface on a Samsung Galaxy S4 device actively running a *Mobilyzer*-enabled app with a 250 MB monthly measurement quota. We find that the power consumed by *Mobilyzer* in this scenario is nearly identical to the Android OS itself, with each comprising about 5% of the total energy drain. We believe this power consumption to be reasonably low for most

users, and emphasize that *Mobilyzer* allows users to specify limits on the maximum energy our measurements consume.

5.3.3 Data Usage

We now evaluate adaptive measurement scheduling in response to per-device caps of *Mobilyzer* data consumption. Our system consumes data quota from (i) fetching measurement requests from the global manager, (ii) uploading measurement results, and (iii) running the measurement tasks. The last category constitutes the vast majority of data consumption, so we set the frequency of periodic measurements according to each device's data cap. We now demonstrate how well this works in practice.

For this experiment, we monitor the data consumption on a HTC One device by reading the proc files under `/proc/uid_stat/<app uid>` once per minute to monitor the app-specific data usage for *Mobilyzer*. Figure 7 shows *Mobilyzer*'s data usage on a cellular network during 14 hours for a 250 MB and 50 MB data cap, along with the corresponding ideal data consumption. The server adjusts the measurement frequency for tasks based on estimates of the data consumed by each task and the device's data cap. Data is consumed at a slightly higher rate than expected, as the server estimates are not precise. For more precise data consumption control, a client-side data monitor measures all data consumed by *Mobilyzer* and stops running server-scheduled tasks as soon as the limit is reached.

5.4 Server Scheduling

The *Mobilyzer* global scheduler enables dynamic, interactive measurement experiments where tasks assigned to devices vary in response to results reported from prior measurements. We present two applications of this feature: measuring the *CDN redirection effectiveness* of major CDNs, and *network diagnosis with adaptive scheduling*, which dynamically schedules diagnosis measurements in response to observed performance issues.

5.4.1 CDN Redirection Effectiveness

In previous work [62], Su et al. used PlanetLab-based experiments to show that CDNs typically send clients to replica servers on low latency paths, instead of optimizing for factors such as load balancing. This requires measuring paths not only to replica servers returned by a DNS redirection, but also testing the latency to other replica servers that could have been selected. Using a dynamic measurement experiment, we repeat this study for the mobile environment to test the extent to which this holds true.

Specifically, we select DNS names served by five large CDNs (Akamai, LimeLight, Google, Amazon CloudFront, and EdgeCast) and measure the latency to the servers that mobile clients are directed toward via DNS lookups. In addition, we measure the latency to servers (5 servers with lowest latency from the recent device measurements and 5 randomly selected servers in distinct /24 prefixes) that *other* devices are directed toward.

For each round of measurement, we find the delay difference between the lowest-latency server and the one returned by the DNS lookup to determine the quality of the mapping. In addition, we sort the latencies to determine the *rank* of the server. For example, if the server returned by the CDN is the second-fastest of ten, its rank is 2.

We summarize our rank results in Figure 8 for 476 devices which used either WiFi (860K measurements) or cellular (393K measurements). Each data point represents the ranking result for one round of measurement.

The figures show that CDNs do not pick the best server half of the time, and as we show below, a this leads to a significant performance penalty for many of our measurements. We calculate the latency difference between the lowest-latency CDN replica and the CDN-selected replica for the same devices in each round of ping measurements. In more than 40% of cases, the latency to the CDN-selected server is optimal. Figure 9 plots the latency difference for cases where the CDN-selected replica is not optimal. We find that for cellular users, 20% of the cases lead to a latency difference of higher than 100ms, which is particularly bad for small downloads common in Web pages. In the worst 10% of cases, there is a noticeable difference between WiFi and cellular; for some CDNs (*e.g.,* Akamai and EdgeCast), the latency difference for WiFi is 50 ms, compared to 500 ms for cellular. While we do not identify the reason for each of these cases, previous work identifies a variety of cases caused by path inflation [69]. Importantly, in contrast to previous work in 2011 showing little opportunity for CDN optimization in mobile networks [68], our study indicates that mobile carriers have more egress points today and CDNs have multiple options for serving clients, but they do not always make optimal decisions for mapping clients to CDN replicas, which can have a significant impact on performance.

5.4.2 Network Diagnosis with Adaptive Scheduling

Dynamic scheduling not only enables new measurements in the mobile environment, it also provides an opportunity to improve measurement efficiency. For example, consider a measurement experiment that measures performance periodically and issues diagnosis measurements in response to anomalous conditions, *e.g.,* to isolate whether there is a problem with a specific server, endpoint or network. Without a priori knowledge about when and where network problems will manifest, the only way to ensure diagnosis information is always available is to issue diagnosis measurements even when there is no problem detected. We now use two examples to show how *Mobilyzer* allows us to issue diagnosis measurements on demand in response to observed problems. For both cases, the global manager scheduled a diagnosis traceroute task only after observing an increase in ping round trip time.

Data Roaming. When roaming, a subscriber's IP traffic may egress into the public Internet directly from the roaming carrier, or it can be forwarded to the home carrier [15], to facilitate data-usage accounting and provide access to services located only in the home carrier. Our diagnosis measurements identified the cost of the latter approach. When a Verizon (US) user roamed on five Europe carriers, the that latency to google.com (server in US) increased from 198±46ms to 613±34ms for the same cellular access technology. Our traceroutes reveal that the first hop of the path belongs to Verizon network for all the measurements from five roaming carriers and the first hop round trip time increased from 159±47ms (non-roaming) to 469±26ms (roaming), indicating that *traffic from Europe was tunneled to the US and back for this user, incurring hundreds of additional milliseconds of delay*.

Path Dynamics. In this example, the path measurement revealed that 100 ms increase in latency for a Vinaphone user was due to a transient path change (perhaps due to a failure and/or BGP update): latency and hop counts increased by a factor of 3 and 1.5, respectively.

6. NEW MEASUREMENTS ENABLED

Mobilyzer enables us to crowdsource and measure[2] the performance of two popular Internet services on mobile devices: Web

[2]This study is IRB-approved and participating users are consented before participating via an in-app dialog. Users may report data via a Google account, which allows them to view and delete their data at any time, or they may report data anonymously. We strip all user information and IDs from the data we gather; further, we coarsen the location granularity to one square kilometer.

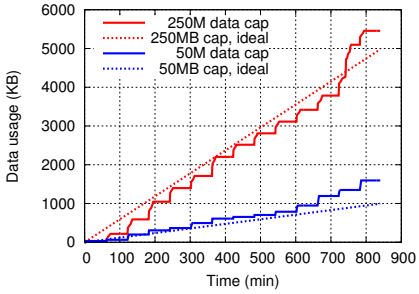


Figure 7: Data usage under different data caps.

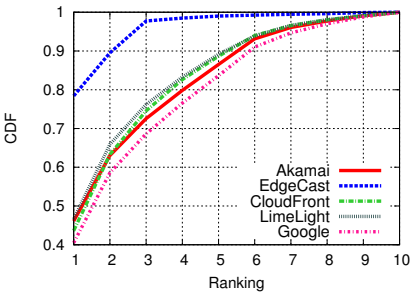


Figure 8: CDF of CDN-selected server rank (in terms of latency) compared to ten other CDN servers. WiFi and cellular redirections have similar distributions, so we show only the aggregated result here.

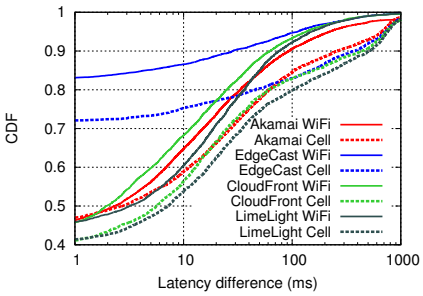


Figure 9: CDF of ping latency difference between the CDN-selected server and the CDN IP with lowest latency.

page load time (PLT) and Video QoE. We use the PLT measurement to show that mobile CPUs can be a significant bottleneck in browser performance, and simple optimizations can improve it. With our video QoE testing, we compare the performance of different bitrate adaptation schemes in the mobile environment.

6.1 Page Load Time Measurement

Recent studies show that a significant fraction of mobile Internet traffic uses HTTP [56], and many of those flows come from Web browsing. The wait time between requesting a Web page and the page being rendered by the browser has a significant impact on users' quality of experience (QoE). This is often measured using the PLT metric, defined to be the delay to fetch all the resources for a page. In addition to measuring PLT for mobile devices, our work is the first to characterize and compare it with page interactive time (PIT), which is the delay to first render the page by the browser so that a user can interact with it.

PLT and PIT are important performance metrics, useful for predicting user experiences. Large PLT and PIT values can lead to users abandoning the pages. As a result, there are several efforts to improve PLT measurements. Browsers collect and report fine-grained page performance and usage info using the Telemetry API [1, 14], without identifying the reasons for the measured performance. Systems such as WebPageTest [17] and others [39] provide similar information from a small set of dedicated hosts in multiple locations. The Wprof [65] approach helps explain the resource dependencies that affect page load times, but it requires a custom browser and has been used primarily in a lab environment.

Importantly, there is a poor understanding of PLTs in the *mobile environment*. The key challenge is that it is difficult to instrument existing mobile browsers (*e.g.,* because they do not support extensions). Huang et al. [31] studied and measured mobile Web browsing performance using controlled experiments and inferred PLT from packet traces. However, we still have a limited understanding of the key factors that affect PLT behavior in the mobile environment.

In this section, we use *Mobilyzer* to conduct the first crowdsourced measurements of mobile Web page performance. We use a novel methodology that combines empirical data from resource download timings with Wprof dependency graphs generated outside the mobile environment. This allows us not only to measure summary statistics like PLT and PIT, but also understand bottlenecks such as network and computation.

6.1.1 Methodology

We use a PLT measurement task to investigate page load times on mobile devices. Previous methods to measure PLT either instrument the browser to report resource load timings [65], or rely on packet traces to infer the page load time [31,50]. Some also used the change in layout to infer PLT [23]. Using crowdsourcing, we cannot modify a browser or infer from network traces. Instead, we use the Navigation Timing (NT) API as described below, which is arguably one of the most accurate ways of measuring PLT, as uses system clock time and measured by the browser itself.

Metrics. Our implementation loads a URL in a `WebView`, a browser commonly embedded in mobile apps. We measure the time for a `WebView` to load a particular page using the NT API [10], a W3C standard implemented in all browsers. This allows us to measure timings for DNS lookup, TCP handshake, transfer of the main HTML file, parsing and constructing the Document Object Model (DOM), render time, and the completion of the page load. After loading a URL using `WebView`, we use JavaScript to read these timings and report them in the measurement result.

A key limitation of the NT API is that it reports timings only for the main HTML page, not any other referenced resources. To get resource timings, we intercept resource requests from the `WebView` using an Android API that notifies our PLT task when each resource is requested, allowing us to capture all of a page's resources and their timings.

When parsing completes and the DOM is constructed, the browser can start rendering and painting the initial view of the page. At this time, called *above-the-fold render time*, page becomes interactive. In this paper, we refer to this as the *page interactive time* (PIT). Developers argue that PIT is a better metric to quantify the user perceived performance of the page than the total PLT, as it can reflect how quickly the user can begin interacting with the page [29].

Pages measured. To study the PLT of webpages on mobile devices, we selected ten popular webpages, one from each of the following categories in the top 100 Alexa pages: social networking, image and video hosting services, forums, wikis, shopping, and banking Web sites. For sites with a landing page that only shows a login screen or redirects the client to an Android app, we selected a custom profile page for testing to obtain a more meaningful result.

6.1.2 Summary results

We collected measurements from 80 mobile devices worldwide, providing resource timings for 2,000 measurements per page on average. We present summary results in Fig. 10, which separates page load time into TCP handshake, SSL handshake, time to first byte, parsing, PIT, and total PLT.

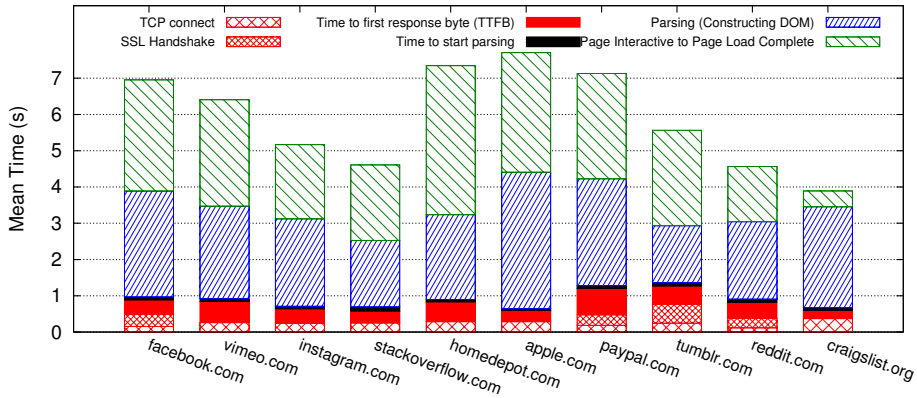


Figure 10: Navigation timing data from crowdsourced PLT measurements, comprising 80 users during six weeks (2000 measurements per page on average).

A key result is that total PLTs range from 4 (craiglist) to 7 (apple.com) seconds, which is substantially larger than reported in desktop environments (*e.g.,* most PLTs were less than 2 seconds in a previous study [65]). As expected, the pages with higher total PLTs tend to be more resource-rich (images, JavaScript). Fig. 10 shows that the initial network costs (TCP and SSL handshake, time to first byte) are a small fraction of total page load time. That said, the SSL handshake can add up to hundreds of milliseconds of delay in the PLT [25]. Importantly, the network delays for loading pages are nearly identical for all pages and quite small, suggesting that *improving network latencies alone is unlikely to significantly reduce PLTs in the mobile environment*.

The figure also shows that most of the page load time comes from rendering and resource loading. It is important to note the large gap between PIT (when the DOM construction is complete) and total PLT. This typically occurs due to network delays for downloading resources and processing delays for parsing them. In the next section, we disentangle these two factors to identify the bottlenecks for page rendering on mobile devices.

Based on the gap between PIT and PLT, the proportion of page interactive time to total page load time ranges from 45% for homedepot to 90% for craigslist, and the mean ratio of PIT to PLT is 0.62 (median 0.63) for the top 200 Alexa pages. This is in contrast to findings in previous work [50], which reported that for 60% of the pages, page render time is equal to page load time. Our results indicate that *it is important to consider both PIT and PLT because of their high variability across sites*.

6.1.3 Is the bottleneck computation or network?

The previous analysis describes that most of the user delays for loading Web pages result from parsing and resource loading, without further separating the time spent downloading objects from that processing them. We now disentangle these factors and demonstrate that processing delays are by far the largest bottleneck.

Inferring processing time. This requires understanding when a browser spends time downloading objects versus processing them — the NT API reports only the start the the end of the resource download, without detailing the processing time. The Wprof project provides this missing information, but requires running a custom browser that we could not distribute to our users. We propose a new methodology consisting of running Wprof analysis on a Web page loaded in our lab environment, followed by using the resulting dependency graph to infer processing times based on empirical resource timing gathered using *Mobilyzer*.

We argue that the dependency graph generated by Wprof for Chrome is likely identical to the one for the Android WebView, because of these reasons: (1) most of the dependencies in the pages are flow dependencies, which are imposed by all browsers [65], and (2) both Android WebView and Chrome use WebKit as their rendering engine. To help validate this assumption, we compared the download order of resources in Chrome (dependency graph) and Android WebView (resource timing), and found them to be consistent.

Results. To determine the amount of computation time, we exclude the network transfer time; *i.e.,* we emulate the case where all resources are cached. Figure 13 presents the average and standard deviation for PLT, PIT and computation time. The results show that for most of the pages in our study, computation is responsible for about half of the PLT and an even larger fraction of PIT. Note that for pages such as Tumblr with fewer scripts or more static content (*e.g.,* images), the network is responsible for approximately 80% of total PLT. To compare, the Wprof study reported the median portion of computation time for the top 200 Alexa pages in the desktop environment to be 35%, higher than the computation time of our five pages (Fig. 13). This difference results from using the mobile version of pages in our experiments, which are simpler than the full version used in previous work.

We further used our crowdsourced measurements to understand the impact of mobile-device CPU speed on loading Web pages. We grouped the devices in our dataset into two clock frequency ranges of their processor: 1-1.5 GHz (17 distinct models) and 1.9-2.5 GHz (13 distinct models). We also loaded the pages on a desktop computer with a 2.6GHz CPU to compare the computation time in a device with a powerful processor. Figure 12 demonstrates that PLT in mobile devices can significantly benefit from faster CPUs. Specifically, the computation time of smartphones with 1.9-2.5GHz processor is around half of the smartphones with 1-1.5GHz processors on average. *Thus, Web site load times can significantly benefit from faster processors*.

Importantly, desktop load times are significantly faster than mobile devices. A potential pitfall of using desktops to simulate the mobile environment is clear: while Wprof finds that increasing the CPU frequency from 2GHz to 3GHz will decrease the computation time by only 5%, we find in the mobile environment that the benefit of using faster processors is significantly higher. Figure 13 indicates that, compared to a desktop computer with a 3 Mbps downlink (*i.e.,* similar to 3G's 2-3 Mbps), the CPU resources in smartphones is a bottleneck for Web page rendering, as it accounts for half of the total PLT.

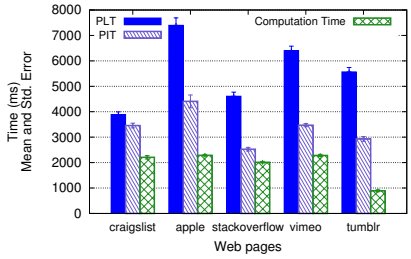


Figure 11: Browser computation time versus PLT and PIT. The fraction of load time due to computation is variable, but often a significant portion.

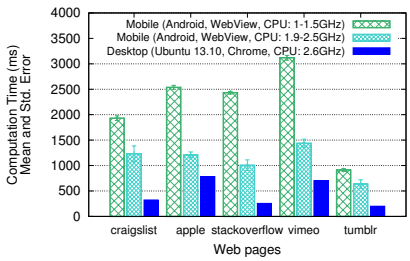


Figure 12: Computation time of different mobile processors (30 distinct device models) compared with desktop.

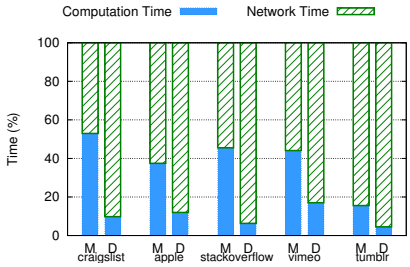


Figure 13: Computation time in mobile devices can account for more than half of the PLT. (Desktop was connected to a 3Mbps broadband internet service.)

6.1.4 *Critical Rendering Path and Render Time*

To shorten the PIT, developers attempt to optimize the critical rendering path (CRP) [4] — the sequence of steps the browser uses to construct the DOM and render the initial view of a Web page. During parsing, some network and computation processes can occur in parallel, but certain blocking resources prevent parallelism and are the main factors for the CRP time. Importantly, optimizing resources not on the CRP (*non-blocking* resources) cannot shorten the render time. We now use our Wprof-generated dependency graphs and empirical page load times to diagnose cases where the CRP is unduly long.

We compare the total parsing time and total computation time in Figures 10 and 11, and find that for some of the pages, total computation time differs from the total parsing time, implying that the parser is blocked by the network. For example, for craigslist and tumblr, the blocking time due to network in the CRP comprises 27% and 24% of total parsing time, respectively. By investigating the dependency graphs for tumblr and craigslist, we find that large blocking JavaScript files (150-260KB) are the culprit, since these resources may not have been downloaded yet when the parser needs to process them. In some cases, these problems can easily be remediated: for craiglist the 255 KB JavaScript file would require only 77 KB if the server supported gzip compression.

6.2 Video QoE Measurement

Video streaming over mobile devices is gaining popularity: YouTube [5] reorted recently that about 50% of views come from a mobile phone or tablet. Further, the average bandwidth achievable in cellular networks in 2014 is 11 Mbps [11], sufficient for high resolution content (1440p).

Despite the higher capacities in today's mobile networks, a key challenge for streaming video providers is how to ensure reliable and high quality video delivery in the face of variable network conditions common in the mobile environment. This requires avoiding stalls and rebuffering events, and adapting the video bitrate to available network bandwidth. This is commonly done via Dynamic Adaptive Streaming over HTTP (DASH), which tries to seamlessly adapt the requested video quality based on available bandwidth, device capability and other factors. In this section, we evaluate the relative performance of two alternative schemes for DASH, using crowdsourced measurements over mobile networks experiencing a diversity of network conditions.

Methodology. We use the ExoPlayer library [2], which provides a pre-built customizable video player for Android with DASH. ExoPlayer is currently used by YouTube and Google Play Movies [3], thus allowing us to directly apply our findings to these popular video services. This library allows us to record throughput, bitrates and rebuffering events over time during video playback.

Using ExoPlayer, we implement a recently proposed buffer-based adaptive (BBA) streaming algorithm [32], and compare it with capacity-based adaptive (CBA) streaming, implemented by YouTube. Both CBA and BBA attempt to minimize rebuffering during playback. BBA [32] dynamically adapts the video bitrate by monitoring the buffer occupancy. To avoid rebuffering, it downloads chunks with a lower bitrate when the buffer starts draining. During the startup phase, it uses the estimated bandwidth to select the appropriate bitrate. On the other hand, CBA only considers estimated bandwidth when selecting the proper bitrate. It always chooses a bitrate that is less than or equal to the estimated bandwidth.

We use a two-minute YouTube video[3] with 5 bitrates (144p to 720p) for our measurements, and streamed it using BBA and CBA on *Mobilyzer* clients. We collected 10K video measurements from 40 users during two weeks.

Results. We considered two QoE metrics to evaluate the performance of these algorithms: (1) Average bitrate of the video that is shown to user, which shows the average quality of video, and (2) rebuffering events, which reports if the video gets interrupted during the playback.

Average Bitrate. To investigate the performance of BBA and CBA under different conditions, we group the measurements based on their average bandwidth[4] into two groups: larger than highest video bitrate ($2193Kbps$) and smaller than the highest bitrate. In the former case, there should be no need for rate adaptation, and in the latter, the streaming algorithm must pick a bitrate lower than the highest quality.

Figure 14 shows the average throughput and bitrates achieved by the two algorithms in crowdsourced measurements. In the low bandwidth scenario, BBA displays videos with a higher average bitrate. To understand why, we focus on two BBA and CBA experiments with the same throughput during the measurement (Fig. 15). The figure shows that BBA switches to a higher bitrate when the buffer occupancy is high, while CBA does not adapt its rate because it considers only the instantaneous estimated throughput. Therefore, when available bandwidth falls between available bitrates (or switches between them), BBA will provide a higher bitrate than CBA on average. On the other hand, we observe that CBA adapts more quickly to sudden changes in throughput by switching to higher or lower bitrates, while BBA maintains its current bitrate based on its current buffer occupancy.

For devices with larger bandwidth, CBA provides a slightly higher average bitrate than BBA. This occurs because BBA is

[3]We pick this value to limit data consumption; further, a recent study indicates that most video views are less than 3 minutes [12].
[4]Bandwidth is estimated from the chunk download times.

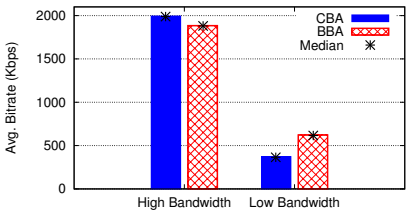


Figure 14: Comparing BBA with CBA bitrates and throughput for users with throughputs that are higher and lower than the maximum bitrate.

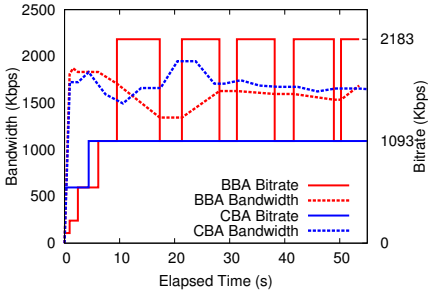


Figure 15: Timeline plots of typical throughputs and bitrate adaptations for BBA and CBA. BBA tends to achieve higher average bitrates than CBA with minimal impact on rebuffering.

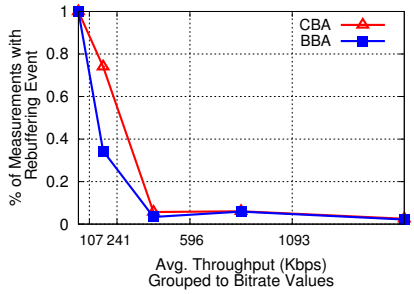


Figure 16: BBA leads to lower rebuffering rates compared with CBA, particularly for users with lower throughput.

more conservative and increases its bitrate incrementally during the startup phase.

Rebuffering. A significant additional component of video QoE is the rate of rebuffering events — the lower, the better. We investigate the performance of BBA and CBA using this metric in Figure 16, which shows the percentage of measurements with rebuffering events. We group our results into five bandwidth buckets based on the video bitrates. The figure shows that when the average bandwidth is lower than the smallest throughput, all the measurements experience at least some rebuffering. However, with higher bandwidths, BBA experiences fewer rebuffering events. This occurs because it avoids rebuffering by switching to the lowest bitrate when the buffer is nearly drained.

Competition under limited bandwidth. To understand how BBA and CBA behave when they compete for a limited bandwidth, we stream videos using a smartphone connected to an AP with constrained bandwidth. We played a BBA and CBA stream at the same time using a *Mobilyzer* parallel task. When using the *load control* ExoPlayer feature, streaming pauses when the playback buffer reaches a threshold; in this case, CBA will pauses for 22.9s on average, while BBA pauses only for 2.9s. This lower pause time occurs because BBA switches to higher bitrates when the buffer is nearly full. Thus, while the throughput is lower than the highest bitrate, BBA will not pause. In contrast, playback buffer size for CBA will grow continuously, due to choosing a bitrate lower than the estimated throughput.

These behaviors have a significant impact on energy and performance. In terms of energy, CBA improves efficiency by letting the radio go idle during pauses. However, when CBA pauses, BBA consumes the remaining bandwidth and achieves higher average throughput. In our experiments, BBA achieved 63% higher throughput and 57% higher bitrate than CBA, meaning that BBA will provide higher QoE.

In summary, our evaluation shows that the BBA algorithm performs better than CBA in the mobile environment based on common QoE metrics. Further investigation is needed to determine how frequent rate adaptation affects the user experience, and to understand the broader network impact of large numbers of clients using BBA instead of CBA.

7. CONCLUSION

This paper makes the case for network measurement as a service in the mobile environment, and demonstrates how we addressed several challenges toward making this service practical and efficient. *Mobilyzer* provides (i) network isolation to ensure valid measurement results, (ii) contextual information to ensure proper scheduling of measurements and interpretation of results, (iii) a global view of available resources to facilitate dynamic, distributed experiments and (iv) a deployment model with incentives for experimenters and app developers. We showed that our system is efficient, easy to use and supports new measurement experiments in the mobile environment. As part of our future work, we are designing an interface to facilitate new measurement experiments and testing new ideas for predicting resource availability and optimizing measurement deployment.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Rajesh Balan, for their helpful feedback. This work was supported in part by NSF under CNS-1059372, CNS-1345226, and CNS-1318396, a Google Faculty Research Award, and Measurement Lab.

8. REFERENCES

[1] Chrome's Telemetry performance testing framework. `http://www.chromium.org/developers/telemetry`.

[2] ExoPlayer. `http://developer.android.com/guide/topics/media/exoplayer.html`.

[3] ExoPlayer: Adaptive video streaming on Android - YouTube. `https://www.youtube.com/watch?v=6VjF638VObA`.

[4] Google Developers, Critical Rendering Path. `https://developers.google.com/web/fundamentals/performance/critical-rendering-path/`.

[5] Half of YouTube's traffic is now from mobile - CNBC.com. `http://www.cnbc.com/id/102128640`.

[6] Internet Speed - Android Apps on Google Play. `https://play.google.com/store/search?q=network%20tools&c=apps&hl=en`.

[7] Mobilyzer dashboard. `https://openmobiledata.appspot.com`.

[8] Mobilyzer data repository. `https://console.developers.google.com/storage/openmobiledata_public/`.

[9] Monsoon Power Monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`.

[10] Navigation Timing. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html.

[11] Ookla NET INDEX EXPLORER. http://explorer.netindex.com/maps.

[12] Ooyala Global Video Index Q2 2014. http://go.ooyala.com/rs/OOYALA/images/Ooyala-Global-Video-Index-Q2-2014.pdf.

[13] Phonelab testbed. http://www.phone-lab.org.

[14] Telemetry. https://wiki.mozilla.org/Telemetry.

[15] The LTE Network Architecture, A comprehensive tutorial. Alcatel-Lucent.

[16] University of Notre Dame NetSense Project: A study into the formation and evolution of social networks using mobile technology. http://netsense.nd.edu/.

[17] WebPagetest - Website Performance and Optimization Test. http://www.webpagetest.org/.

[18] Alcatel-Lucent. 9900 wireless network guardian, 2013. http://www.alcatel-lucent.com/products/9900-wireless-network-guardian.

[19] A. Auyoung, P. Buonadonna, B. N. Chun, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. *Two Auction-Based Resource Allocation Environments: Design and Experience*. John Wiley & Sons, Inc., 2009.

[20] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: A platform for educational cloud computing. In *SIGCSE*, 2009.

[21] Carrier Compare. Compare speed across networks. https://itunes.apple.com/us/app/carriercompare-compare-speed/id516075262?mt=8.

[22] Carrier IQ. Carrier IQ: What it is, what it isn't, and what you need to know. http://www.engadget.com/2011/12/01/carrier-iq-what-it-is-what-it-isnt-and-what-you-need-to/.

[23] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau. QoE Doctor: Diagnosing Mobile App QoE with Automated UI Control and Cross-layer Analysis. In *Proc. of IMC*, 2014.

[24] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.

[25] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafoo, K. Papagiannaki, and P. Steenkiste. The Cost of the "S" in HTTPS. In *Proc. of ACM CoNEXT*, 2014.

[26] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing Radio Resource Allocation for 3G Networks. In *Proc. of IMC*, 2010.

[27] FCC Announces "Measuring Mobile America" Program. http://www.fcc.gov/document/fcc-announces-measuring-mobile-america-program.

[28] A. Gerber, J. Pang, O. Spatscheck, and S. Venkataraman. Speed testing without speed tests: estimating achievable download speed from passive measurements. In *Proc. of IMC*, 2010.

[29] I. Grigorik. Deciphering the Critical Rendering Path. http://calendar.perfplanet.com/2012/deciphering-the-critical-rendering-path/.

[30] C.-K. Hsieh, H. Falaki, N. Ramanathan, H. Tangmunarunkit, and D. Estrin. Performance evaluation of android ipc for continuous sensing applications. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2012.

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

[32] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proc. of ACM SIGCOMM*, 2014.

[33] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Perfomance and Power Characteristics of 4G LTE Networks. In *Proc. of MobiSys*, 2012.

[34] K. Jang, M. Han, S. Cho, H. Ryu, J. Lee, Y. Lee, and S. Moon. 3G and 3.5 G Wireless Network Performance Measured from Moving Cars and High-Speed Trains. In *Proc. of Workshop on Mobile Internet through Cellular Networks*, 2009.

[35] P. Kortum, A. Rahmati, C. Shepard, C. Tossell, and L. Zhong. LiveLab: Measuring wireless networks and smartphone users in the field. http://livelab.recg.rice.edu/traces.html.

[36] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *IMC*, 2010.

[37] M. Laner, P. Svoboda, E. Hasenleithner, and M. Rupp. Dissecting 3G Uplink Delay by Measuring in an Operational HSPA Network. In *Proc. PAM*, 2011.

[38] Measurement Lab website. http://www.measurementlab.net/.

[39] V. S. Michael Butkiewicz, Harsha Madhyastha. Understanding Website Complexity: Measurements, Metrics, and Implications. In *Proc. of IMC*, 2011.

[40] MySpeedTest App. https://play.google.com/store/apps/details?id=com.num.

[41] NetRadar. https://www.netradar.org/en/about.

[42] C. Ng, P. Buonadonna, B. N. Chun, A. C. Snoeren, and A. Vahdat. Addressing strategic behavior in a deployed microeconomic resource allocator. In *Workshop on Economics of Peer-to-Peer Systems*, 2005.

[43] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao. Mobilyzer: An Open Platform for Principled Mobile Network Measurements. Technical report, University of Michigan, 2014. http://www.eecs.umich.edu/~zmao/Papers/Mobilyzer.pdf.

[44] Ookla. Ookla speedtest mobile apps. http://www.speedtest.net/mobile/.

[45] P802.16.3 - standard for mobile broadband network performance measurements. http://standards.ieee.org/develop/project/802.16.3.html.

[46] A. Patro, S. Rayanchu, M. Griepentrog, Y. Ma, and S. Banerjee. The anatomy of a large mobile massively multiplayer online game. In *Proc. of MobiGames*, 2012.

[47] PlanetLab website. http://www.planet-lab.org.

[48] Portolan. The portolan network sensing architecture. http://portolan.iet.unipi.it/.

[49] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Proc. of MobiSys*, 2010.

[50] Qian, Feng and Sen, Subhabrata and Spatscheck, Oliver. Characterizing Resource Usage for Mobile Web Browsing. In *Proc. of MobiSys*, 2014.

[51] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *Proc. of USENIX OSDI*, 2012.

[52] RIPE NCC. Ripe atlas. `https://atlas.ripe.net/`.

[53] P. Romirer-Maierhofer, F. Ricciato, A. D'Alconzo, R. Franzan, and W. Karner. Network-Wide Measurements of TCP RTT in 3G. In *Proc. of TMA*, 2009.

[54] 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. of MobiCom*, 2014.

[55] M. A. Sánchez, J. S. Otto, Z. S. Bischof, D. R. Choffnes, F. E. Bustamante, B. Krishnamurthy, and W. Willinger. Dasu: pushing experiments to the internet's edge. In *NSDI*, 2013.

[56] Sandvine. Global Internet Phenomena Report 1H 2014. `https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/1h-2014-global-internet-phenomena-report.pdf`, 2014.

[57] J. Sclamp and G. Carle. Measrdroid. `http://media.net.in.tum.de/videoarchive/SS13/ilab2/2013+05+02_1000+MeasrDroid/pub/slides.pdf`.

[58] S. Sen, J. Yoon, J. Hare, J. Ormont, and S. Banerjee. Can They Hear Me Now?: A Case for a Client-assisted Approach to Monitoring Wide-area Wireless Networks. In *Proc. ACM IMC*, 2011.

[59] J. Sommers and P. Barford. Cell vs. WiFi: on the performance of metro area mobile connections. In *Proc. of IMC*, 2012.

[60] Speedometer project source. `https://github.com/Mobiperf/Speedometer`.

[61] Speedtest.net. `http://www.speedtest.net/`.

[62] A.-J. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting behind Akamai: Travelocity-based detouring. In *Proc. of ACM SIGCOMM*, Pisa, Italy, September 2006.

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

[64] F. Vacirca, F. Ricciato, and R. Pilz. Large-Scale RTT Measurements from an Operational UMTS/GPRS Network. In *Proc. of WICON*, 2005.

[65] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proc. of USENIX NSDI*, 2013.

[66] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. of ACM SIGCOMM*, 2011.

[67] Q. Xu, A. Gerber, Z. M. Mao, and J. Pang. Acculoc: practical localization of performance measurements in 3g networks. In *Proc. of MobiSys*, 2011.

[68] Q. Xu, J. Huang, Z. Wang, F. Qian, A. Gerber, and Z. M. Mao. Cellular data network infrastructure characterization and implication on mobile content placement. In *Proceedings of SIGMETRICS*, 2011.

[69] K. Zarifis, T. Flach, S. Nori, D. Choffnes, R. Govindan, E. Katz-Bassett, Z. M. Mao, and M. Welsh. Diagnosing path inflation of mobile client traffic. In *Proc. PAM*, 2014.