Automatic Metadata Generation for Active Measurement

Joel Sommers
Colgate University
jsommers@colgate.edu

Ramakrishnan Durairajan
University of Oregon
ram@cs.uoregon.edu

Paul Barford
University of Wisconsin-Madison
comScore, Inc.
pb@cs.wisc.edu

ABSTRACT

Empirical research in the Internet is fraught with challenges. Among these is the possibility that local environmental conditions (*e.g.*, CPU load or network load) introduce unexpected bias or artifacts in measurements that lead to erroneous conclusions. In this paper, we describe a framework for local environment monitoring that is designed to be used during Internet measurement experiments. The goals of our work are to provide a critical, expanded perspective on measurement results and to improve the opportunity for reproducibility of results. We instantiate our framework in a tool we call SoMeta, which monitors the local environment during active probe-based measurement experiments. We evaluate the runtime costs of SoMeta and conduct a series of experiments in which we intentionally perturb different aspects of the local environment during active probe-based measurements. Our experiments show how simple local monitoring can readily expose conditions that bias active probe-based measurement results. We conclude with a discussion of how our framework can be expanded to provide metadata for a broad range of Internet measurement experiments.

CCS CONCEPTS

• **Networks** → **Network experimentation**; **Network measurement**;

KEYWORDS

network measurement, metadata

ACM Reference Format:
Joel Sommers, Ramakrishnan Durairajan, and Paul Barford. 2017. Automatic Metadata Generation for Active Measurement. In *Proceedings of IMC '17, London, United Kingdom, November 1–3, 2017,* 7 pages.
https://doi.org/10.1145/3131365.3131400

1 INTRODUCTION

Active probe-based measurements have been widely used to elucidate Internet characteristics and behavior. Typical objectives for active probe-based measurement include end-to-end path properties (*e.g.*, reachability, latency, loss, throughput), hop-by-hop routing configurations and end-host performance. In each case, a sequence of packets is sent from one or more measurement hosts to remote targets and responses are measured either at the sending host or at a target host. One of the benefits of active probe-based measurement is that it enables broad and diverse assessment of Internet characteristics without the need for permission or authorized access.

Despite the benefits and the availability of data sets through ongoing collection efforts (*e.g.,* [10, 14, 27]), conducting active probe-based measurement studies is fraught with challenges. Among these is the possibility that the *local environment* can introduce unexpected bias or artifacts in measurements. We define the *local environment* as the host emitting probe packets plus other systems in the local area that can materially alter the behavior of probe packets but are intended to be outside of the scope of the measurement objectives. In particular, the host emitting probe packets is assumed to do so in a manner consistent with a measurement protocol. However, prior work has shown that variable CPU load can alter probe sequences [15, 30]. Similarly, hosts that share local connectivity can disrupt probe packet streams by sending bursts of traffic. So, how can we know if measurement fidelity has been affected by the local environment? The answer we advocate is to collect *metadata about the local environment* when measurements are being conducted.

In this paper, we describe a framework for collection of metadata about the measurement environment. Inspired by calls from the community to collect metadata during experiments, our high-level goals are to make the process easy and thereby improve the quality and reproducibility of active probe-based measurement studies. To that end, our design goals are to create a capability that will *(i)* measure the local environment when an active probe tool is being used, *(ii)* not perturb probe traffic, and *(iii)* work seamlessly with different systems and measurement platforms. To the best of our knowledge, this is the first attempt to address these meta-measurement issues.

We develop a tool for metadata collection called *SoMeta*, which addresses our core design goals. SoMeta is activated on initiation of a probe-based measurement campaign. It collects key performance metrics on the measurement host (*e.g.,* CPU and memory utilization) and performs simple probe-based measurements to hosts in the local environment. SoMeta has been implemented in Python, which makes it simple to run on diverse hosts. It has also been implemented to be lightweight in terms of its demands on a measurement host. SoMeta produces simple log files that can be analyzed for indications of high load or other events that provide perspective on unexpected observations in target measurement data.

We demonstrate the capabilities of SoMeta by deploying it on two versions of the Raspberry Pi, which is used in the Ark [10] active probe-based measurement project, and on large server-class systems. We begin by examining the load imposed by SoMeta on the host systems. We find that SoMeta imposes about 12% CPU load on a Pi model 1, 3% on a Pi model 3, and only about 1% on the

IMC '17, November 1–3, 2017, London, United Kingdom

ACM ISBN 978-1-4503-5118-8/17/11…$15.00
https://doi.org/10.1145/3131365.3131400

servers in what we expect will be a typical configuration. Next, we conduct a series of experiments in which we introduce artificial load in the local environment while conducting active probe-based measurements to a remote target using scamper [20]. The results from these experiments show how SoMeta measurements highlight the disturbances caused by the artificial load and how this could be used to point out artifacts in the scamper measurements.

While we believe that SoMeta is an important step toward accountability and reproducibility in active probe-based measurement studies, it has certain limitations that we are addressing as part of our ongoing work, including further reducing its performance overhead, considering usage scenarios in shared measurement environments, and broadening the types of metadata that can be captured, *e.g.*, through in-network monitors. Moreover, there are two additional ways in which the concept should be expanded. First, the community needs to continue conversations about the importance of metadata collection and the kinds of metadata that should be collected to improve experiment reproducibility. Second, deploying SoMeta (or something similar) in an existing infrastructure would enable better understanding of its performance and how measurement artifacts and bias may be identified and possibly corrected. To that end, all code and documentation for SoMeta is readily available[1].

2 RELATED WORK

Over the years, there have been a number of calls from within the Internet measurement community to promote *sound* [25], *hygenic* [19], and *ethical* [24] processes when carrying out Internet measurement studies to improve confidence in the results derived from measurement data, to facilitate data sharing, replication, and reappraisal, and to carefully consider any harm that may be caused [1, 2, 19, 24, 25]. Our study finds inspiration in these prior works, in particular with regard to collecting metadata to assist a researcher with assessing the quality of the collected measurements, and for scientific replication [19, 25]. Particularly related to our work is Paxson's suggestion to measure facets of the same phenomena using different methods as a way to calibrate, assess measurement quality, and to identify or possibly correct for bias [25]. This method was used in prior work to evaluate the fidelity of measurements collected in RIPE Atlas [15], and a related analysis of latency measurement quality on early-generation RIPE Atlas nodes was done by Bajpai *et al.* [5, 6]. Holterbach *et al.* suggest providing a "confidence index" along with reported measurement results, indicating some measure of concurrent load on an Atlas probe [15]; providing such an index could be facilitated through the types of metadata gathered by SoMeta.

There have been a number of specific suggestions in prior work regarding the scope of metadata that should be captured for future reference, *e.g.*, [12, 19, 25] and that metadata should be easily machine-readable [25]. Examples of available metadata (and the associated data) can be found on Internet measurement data repositories such as CRAWDAD [17, 18], IMDC [11, 28], and M-Lab [13, 21]. Most of the metadata found through these platforms are *descriptive*, *e.g.*, where, when and how the measurements were collected, data format(s), etc., and some of these types of metadata are implicit in the data file naming convention (*e.g.*, time of measurement, measurement node name, measurement tool used). While some metadata collected through SoMeta are descriptive of the environment on which measurement tools are run, its main focus is on gathering system performance data while measurement takes place. In this regard, SoMeta bears some similarity to M-Lab, in which limited measures of Planetlab slice performance metadata gathered through Nagios [23] are available in a JSON format, such as CPU and memory utilization.

[1] See https://github.com/jsommers/metameasurement. In an effort to aid reproducibility, scripts and data used to generate plots can be found by clicking on them.

3 DESIGN AND IMPLEMENTATION

3.1 Design Goals

The design of SoMeta is based on three objectives. First, metadata should be collected by profiling various system resources at discrete intervals during the time in which an active measurement tool executes. In particular, CPU, storage/IO performance, and other system measures should be gathered, and the access network should be monitored, *e.g.*, the first k hops of the network path. Profiling should continue as long as the active measurement tool runs. When the active measurement tool completes, metadata should be stored in a simple and computer-readable format, *e.g.*, JSON, in order to facilitate analysis. Basic tools for analysis and visualization of metadata should be provided to show, *e.g.*, timeseries of idle CPU cycles, packet drops on an interface, RTT to first router, etc.

The next design objective is the lightweight operation of SoMeta, which we have designed to be configurably adaptable to a range of target compute and network settings. Nonetheless, we are also motivated by the fact that CPU power and network bandwidth to the edge has increased to the point that the networking and compute environment in which active network measurement is performed can sustain additional traffic and processing activity from metadata capture. For example, even low-cost computer systems (*e.g.,* the Raspberry Pi Model 3) have significant CPU power in the way of multiple cores (4 in the case of the Pi 3).

The last design objective of SoMeta is the ability to work seamlessly with different systems and measurement platforms. This objective is imperative to accommodate diverse measurement efforts *e.g.*, CAIDA uses Raspberry Pi- and 1U server-based Ark monitors [9], Yarrp uses Ubuntu VM [7], BGPmon uses sites with high-end multicore processors [8], RIPE Atlas currently uses a low-cost wireless router (TP-Link model TL-MR 3020) with custom firmware based on OpenWRT [27], etc.

3.2 SoMeta Overview and Implementation

SoMeta has been implemented in a lightweight and extensible way to meet the design objectives described above. It is written in Python and uses the `asyncio` framework as the basis for structuring and handling asynchronous events associated with monitoring the host system and network. Use of `asyncio` allows SoMeta to be single-threaded, which helps to limit its performance impact on the host system on which it runs. Although `asyncio`'s asynchronous task scheduling is by no means perfect, we argue that it is sufficient for the purpose of collecting the types of metadata we envision.

The task of metadata collection is delegated to a set of *monitors* built in to SoMeta, which we describe below. When SoMeta is

started up, a user must configure some number of monitors, and also supplies the command line for executing an external measurement tool, such as scamper [20]. For as long as the external tool executes, SoMeta's monitors periodically collect system performance measures or other metadata. When the external tool completes, SoMeta writes out the metadata to a JSON file along with a variety of invocation and runtime information such as the full command line used to start SoMeta, the active measurement command, exit status, and any text written to the console by the external measurement tool.

3.2.1 Monitoring capabilities. The core functionality of SoMeta is to monitor, at discrete intervals, host system resources and network round-trip time (RTT) latencies to specific target hosts. SoMeta can be configured to collect CPU (*e.g.*, per-core utilization), I/O (*e.g.*, read/write activity counters), memory (*e.g.*, utilization), and network interface byte, packet, and drop counters. In addition, a monitor module can be configured to collect RTT samples using hop-limited probes, or by emitting ICMP echo request (*ping*) packets toward specific hosts.

There are five built-in monitor modules available, named `cpu`, `mem`, `io`, `netstat`, and `rtt`. Each of these modules can be separately configured, and each executes independently within SoMeta's `asyncio` event loop as a separate coroutine. The discrete interval at which measurements are collected is configurable, and defaults to one second. When SoMeta is started, each configured monitor is initiated after a small random amount of time in order to avoid synchronization. The `cpu`, `mem`, `io` and `netstat` monitors leverage the widely-used `psutil`[2] module for collecting a variety of system performance measures. Use of this third-party module ensures that SoMeta can perform effectively on a wide selection of operating systems, and facilitates the future creation of special-purpose monitors such as battery status or CPU and/or environment temperature, which can be accessed through existing `psutil` APIs. Lastly, we note that the monitoring framework has been designed to be easily extensible in order to permit new types of metadata to be collected and stored, or to be able to use an OS-specific performance monitoring subsystem such as System Activity Reporter (`sar`).

The `rtt` monitor module can be configured to emit either hop-limited probes (with configurable TTL) using ICMP, UDP, or TCP toward a destination address, or to emit ICMP echo request probes. The first four bytes of the transport header can be made constant to avoid measurement anomalies caused by load balancing [3, 26]. The `rtt` module uses the Switchyard framework to access `libpcap` for sending and receiving packets, and for constructing and parsing packet headers [29]. Although the system ARP table is used to bootstrap its operation, the `rtt` module contains ARP functionality. Moreover, the system forwarding table is retrieved on startup in order to determine the correct interface out which packets should be emitted. Lastly, probes are sent according to an Erlang (Gamma) process [4], and the probe rate is configurable. By default, probes are emitted, on average, every second.

Within the `rtt` module, we configure a `libpcap` filter on a pcap device so that only packets of interest to the module are received. In particular, the module only ever receives ARP responses or probe request and response packets. Since SoMeta's `rtt` module uses `libpcap`, timestamps for both sent and received packets are of relatively high quality: at worst they come from the OS, and at best they may come directly from the NIC. Unfortunately, however, use of `libpcap` does not imply that operating system differences can be ignored. In particular, on the Linux platform, `libpcap` uses a special socket (`PF_PACKET` socket) for sending and receiving raw frames rather than `/dev/bpf` or a similar type of device on BSD-derived platforms (*e.g.*, macOS and FreeBSD). A limitation with Linux's `PF_PACKET` socket is that packets that are emitted through the device *cannot also be received on that same device*. In order to obtain kernel-level timestamps on packets sent on a Linux system, we create a *separate* `PF_PACKET` socket for sending packets. There are yet other (more minor) quirks that are handled within the `rtt` module to smooth out platform differences.

3.2.2 Metadata analysis and visualization. Along with SoMeta's metadata collection capabilities, we have created simple tools to bootstrap analysis and visualization of metadata. An analysis tool can produce summary statistics of round-trip times, flag whether any probes were dropped at an interface or by `libpcap`, and whether there were periods of full CPU utilization, among other capabilities. A plotting tool can produce timeseries or empirical cumulative distribution function plots of any metadata collected, facilitating qualitative analysis to identify time periods during which measurements may have been disturbed due to host or network interference. The plotting capability is based on `matplotlib` [16] and provides functionality for plotting individual monitor metrics, *all* metrics collected by a monitor, or *all* metrics across *all* monitors.

4 EVALUATION

In this section we describe a set of experiments to evaluate SoMeta. We begin by assessing the performance cost of running SoMeta. We follow that by examining how SoMeta might be used in practice.

4.1 SoMeta Performance and Overhead

To evaluate the performance costs of running SoMeta, we created a simple laboratory testbed, which was also connected to a campus network and the Internet. The platforms on which we ran SoMeta are two versions of the Raspberry Pi—specifically, a Model 1 B Rev. 2 (Pi1) and a Model 3 B (Pi3)—and two server-class systems. The Pi1 has a 700 MHz single-core ARM1176JZF-S with 512 MB RAM and ran the Linux 4.1.19 kernel (Raspbian 7). The Pi3 has a 1.2GHz 64-bit quad-core ARM Cortex-A53 CPU, 1 GB RAM and ran the Linux 4.4.50 kernel (Raspbian 8). The two server-class systems are identical Octocore Intel(R) Xeon(R) CPU E5530 @2.40GHz with 16 GB RAM. One was installed with Linux 3.13 (Ubuntu server 14.04) and the other with FreeBSD 10.3. These four systems were connected through a switch via 100 Mb/s Ethernet (Pi1 and Pi3) or 1 Gb/s Ethernet (two server systems) to a series of two Linux-based routers, the second of which was connected to the campus network via a Cisco 6500.

Using the two Pi and two server systems, we ran SoMeta in a series of configurations to test its resource consumption. In particular, there were ten configurations we used: (1) only collect CPU performance measurements every second, (2) only collect RTT measures to the closest Linux router using ICMP echo requests, (3) only collect RTT measures to the closest Linux router using a hop-limited

[2] https://github.com/giampaolo/psutil

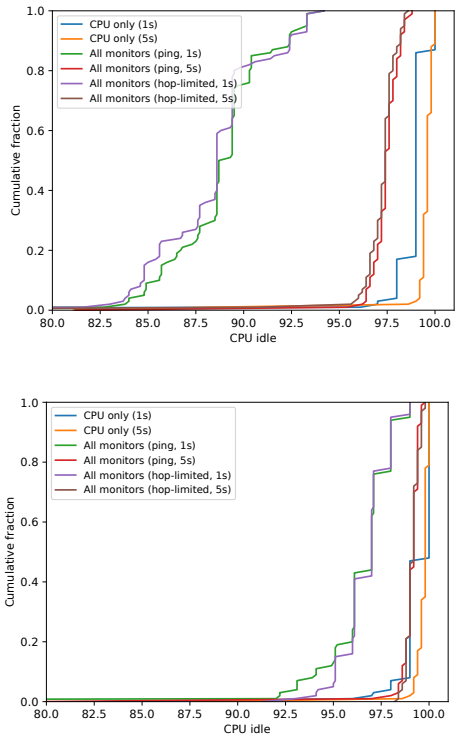

Figure 1: CPU performance overhead results (top: Raspberry Pi Model 1 B; bottom: Raspberry Pi Model 3 B).

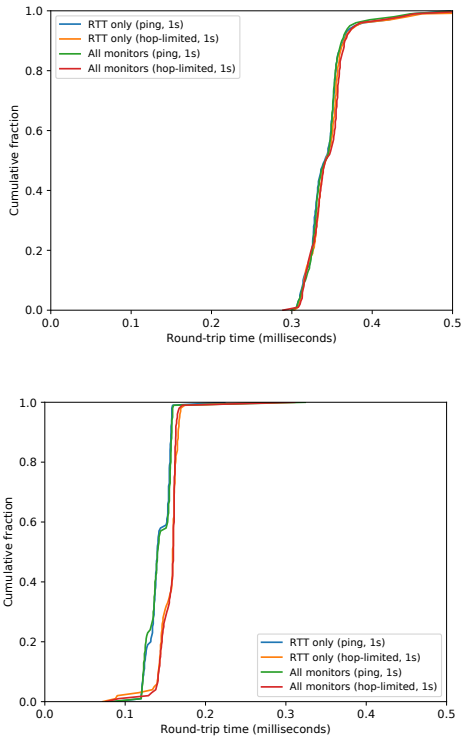

Figure 2: RTT performance overhead results (top: Raspberry Pi Model 1 B; bottom: FreeBSD server).

probe (with maximum TTL of 1), (4) collect performance measures using *all* monitors every second, including measuring RTT to the closest Linux router using ICMP echo requests, and (5) using all monitors to collect performance measurements every second but collecting RTT measures using a hop-limited probe, again to the closest Linux router. Configurations 6–10 were identical to configurations 1–5 except that instead of collecting measurements every second, SoMeta was configured to collect measurements every 5 seconds. Each of these 10 experiments was run for 900 seconds by having SoMeta run the command `sleep 900` (*i.e.*, the `sleep` command is used as the external "measurement" tool). For these experiments we pinned SoMeta to a single CPU core which, while not strictly necessary, simplified analysis of CPU usage and system overhead since 3 out of 4 systems we used were multicore.

Figures 1 and 2 show selected results for the performance overhead experiments. Figure 1 shows empirical CDFs for CPU idle percent for the Pi1 and Pi3, and Figure 2 shows empirical CDFs for RTT for the Pi1 and the FreeBSD server. In the top plot of Figure 1, we observe that enabling all monitors with a 1 second average sampling interval incurs the greatest CPU load. Still, the 50th percentile CPU idle percentage is about 88%, which we view as promising since SoMeta is written in a very high-level language and has not undergone (at this point) any performance optimization. We also observe that when reducing the sampling rate to an average of once every 5 seconds, the 50th percentile CPU idle percentage goes up to 97%, which is only 1–2 percent lower than only measuring CPU usage every second. In the lower plot of Figure 1 we observe similar trends, but with different specific CPU idle percentage values. The higher-powered Pi3 is about 97% idle at the 50th percentile when running all monitors and collecting measurements, on average, every second. For the two server-class systems (not shown), the idle percentage is even higher than the Pi3, at about 98–99% for all monitors with a 1 second sampling interval. Although SoMeta's performance overhead is modest, it may yet be too high in shared measurement environments where either multiple instances of SoMeta may need to be run or its metadata collection architecture may need to be adapted. We are addressing this issue in our ongoing work.

In Figure 2 we show RTT results when running all monitors or only the RTT monitor, and for both ICMP echo request and hop-limited probes, each for a 1 second average measurement interval. We observe that the RTTs are, in general, small and distributed across a narrow range. Given the Pi1's lower performing CPU, it is not surprising that the 90th percentile RTT (≈ 0.35 milliseconds) is more than twice that of the 90th percentile RTT for the FreeBSD server (≈0.15 milliseconds). The 90th percentile RTT for the Pi3 is between the two (≈0.25 milliseconds) and results for the Linux server are similar to the FreeBSD server. We note that none of these systems were configured to perform hardware timestamping on the NIC (it is not supported on the Pi devices, and the two server

systems were not configured to do so). Thus all timestamps used to generate these results were generated in software, in the OS kernel.

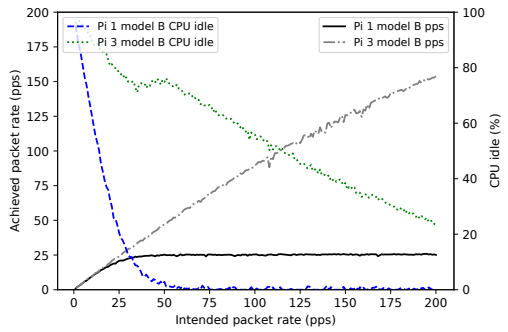

Figure 3: Performance of the Pi1 and Pi3 when configured for a range of probing rates.

Finally, to better understand SoMeta's scheduling accuracy and to examine finer probing intervals which may be needed to monitor certain experiments, we examine its performance when configured to emit probes in an increasingly rapid manner, from 1 probe/s up to 200 probes/s. We ran each of these experiments for 60 sec by having SoMeta execute the command `sleep 60`, and we configured SoMeta with only the CPU and RTT monitors (in hop-limited mode). Figure 3 shows the results for these experiments for the Pi1 and Pi3. We observe that for the Pi1 the CPU is fully utilized at about 25 probes/s, and that the Pi3 can support 150 probes/s with about 25% idle CPU. (CPU utilization on the Pi3 hits 100% around 250 probes/s). We note again that no special performance tuning has yet been done on SoMeta, including any attempt to compensate for scheduling inaccuracies with the `asyncio` module. Importantly, there were zero packet drops for all experiments on both Pi devices, and the RTTs measured were statistically identical to those in the overhead experiments described above. This observation implies that even when the host system is under significant load, metadata collected by SoMeta remain accurate, even if they are not gathered according to the intended schedule. We also examined probe send time accuracy, which we found to be generally accurate on the Pi3 when the system was lightly loaded, but poorer on the Pi1, similar to the findings of [22]. In summary, our experiments show that using `asyncio` for task scheduling appears to be sufficient for modest probe rates but we are nonetheless continuing to examine how to improve scheduling fidelity.

4.2 Artificial System Load Experiments

For the results described in this section, our goal is to illustrate how metadata collected by SoMeta could be used to identify and, to a certain extent, evaluate the impact of system and network interference on network measurements. For these experiments, we configured SoMeta to use all built-in monitors and to gather measurements every second. We also configured it to start the measurement tool scamper [20] and perform latency measurement using ICMP echo requests to the Google IPv4 Anycast DNS server 8.8.8.8. Scamper was configured to store its RTT samples in an external file for later analysis. SoMeta was configured to collect RTT samples from the first three network hops using either a hop-limited probe or ICMP echo request probes, and to also measure RTT to 8.8.8.8. Simultaneous to running SoMeta, we introduced artificial system and network load in four different experiment types: 100% CPU load on all cores, memory system load by cycling reads and writes over a large array, I/O load by repeatedly writing to a file on the same filesystem as the output files generated by scamper and SoMeta, and upload and download transfers of a range of file sizes (100KB, 1MB, and 10MB). CPU, memory, and I/O artificial load were introduced in an on-off cycle of 60 seconds per cycle, starting with 60 seconds of no artificial load, followed by 60 seconds of load. The file transfers to introduce artificial network load were initiated every 10 seconds from a separate host, but using the same network path from the first router up through several network hops.

For these experiments, we used the same testbed as in our overhead experiments described above, and we also deployed a Pi3 in a home network connected to the Internet via a large cable modem provider. The Pi3 deployed in the home network was either attached to the home router using a wired 100 Mb/s connection or via 802.11n WiFi. Figure 4 shows results for the home network-deployed Pi3 connected via WiFi to the router, and with artificial CPU load. The plots show empirical CDFs of RTTs between the Pi3 and the first hop (home router) (left), between the Pi3 and the second hop (center), and RTTs to the scamper target IP address 8.8.8.8. For each plot, we show CDFs for *all* RTT measurements, as well as separate CDFs for the no artificial load time periods (off) and when artificial CPU load is introduced (on). For each plot, we observe that the artificial CPU load has a clear skewing effect on the latency measurements. An experimenter using SoMeta to detect subpar local conditions could (1) evaluate the CPU-related metadata to discover that there were time periods during which there was zero (or very little) idle CPU, and/or (2) compare (either visually, or in a more rigorous quantitative or statistical way) the latency data generated from the experiment with data previously collected during *known "good" time periods*. We note that the "off" curve from the left-hand plot of Figure 4 is virtually identical to a curve generated from a separate experiment in which no artificial load is introduced (not shown).

Lastly, in Figure 5 we show results for the artificial network load experiment with the Pi3 again in the home network environment. Results are shown for file transfer sizes of 100KB, with the Pi3 connected to the home router via 1 Gb/s wired Ethernet. File transfers were initiated from a separate host connected via WiFi. The network path of the file transfers shared the same network path as the probes toward 8.8.8.8 for the first 6 hops. In Figure 5, we show CDFs from measurements in which *no* artificial load was introduced, and CDFs from the artificial network load experiment; note the log scale on the x axis. First, we observe that the first network hop is largely undisturbed. Recall that the access technologies for the Pi3 (wired Ethernet) and the host from which file transfers are initiated (WiFi) differ, thus we do not observe any obvious interference at this hop. For subsequent hops, however, we observe significant skewing even with the transfer of a relatively small file. An experimenter might become aware of the local network disturbance in a similar way as described above, by comparing (as shown in the figure) newly collected data with measurements collected during known quiescent time periods. We note that in a similar experiment in which the

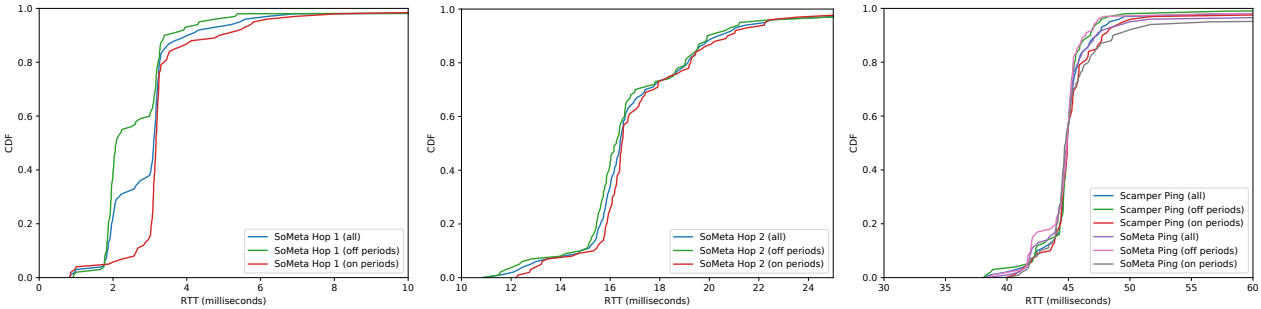

Figure 4: Artificial CPU load experiment.

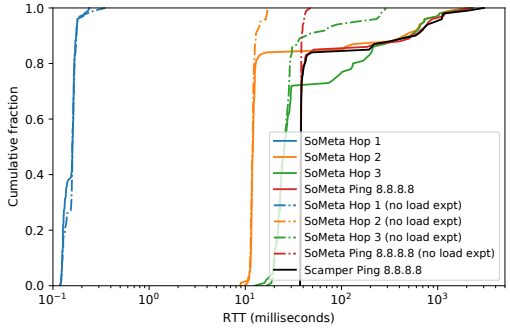

Figure 5: Artificial network load experiment.

Pi3 is connected to the home router via WiFi (not shown), first-hop latencies are observed to be skewed since the Pi3 and the host from which data transfers originate share the WiFi access.

5 SUMMARY AND FUTURE DIRECTIONS

We view the current monitoring capabilities and kinds of data that SoMeta can collect as a starting point toward our goal to simplify collection and analysis of certain types of metadata for Internet measurement experiments. Our results show that even on devices with limited resources, *e.g.*, the Pi1, the cost of collecting host performance and local network measures is low. In particular, about 12% CPU overhead on a Pi1 is incurred using a modest data sampling rate of once per second, or less than 3% with a sample rate of once every five seconds. The potential benefit of these measurements is high, as the results from our experiments show in which artificial system and network load is introduced. In particular, the differences in performance measures gathered during quiescent periods compared with measurements collected when artificial CPU, memory, I/O or network load is present make it apparent that something in the local environment has perturbed the measurement.

In our ongoing work, we are considering a number of directions which we think this work opens up. First, we intend to examine the idea of treating network latency measurements between a host and local network systems that are explicitly collected during quiescent conditions as a *baseline reference* to use for calibration of new measurements. For example, while it is straightforward to evaluate certain host system measures (*e.g.* CPU utilization) to determine periods during which system load *may* contribute to poor network measurement, we believe that one way to gain evidence of impaired measurement quality is to collect additional latency measures via SoMeta's RTT monitor and compare them with previously collected data. In other words, the previously collected measurements could be used as a reference against which to compare, *e.g.*, qualitatively, or using a statistical test such as the Kolmogorov-Smirnov two-sample test. Any shift away from the baseline reference could be detected and possibly even corrected in the network measurements, and at the very least, a measurement quality label could be applied to different time periods of a trace based on analysis of additional measurements collected through SoMeta.

Secondly, while SoMeta is, at present, decoupled from any particular measurement tool, we are considering an API through which measurement tools (or other data sources) could explicitly provide metadata to assist with creating a comprehensive, structured record of an experiment for documentation and scientific replication purposes. In addition to providing an avenue for metadata storage, status/health polling (*i.e.*, "heartbeats") could be done through such an API, and meta-API information (*i.e.*, response time) could be used for continuous assessment of system performance.

There are yet other extensions we are considering to SoMeta that we believe will make it appealing for a wide variety of usage scenarios, including automatic identification and dynamic monitoring of file systems/disks and network interfaces that are being used by a measurement tool, performance improvements for adapting to extremely constrained environments and to shared measurement platforms where there may be multiple experiments executing simultaneously. Whether or not SoMeta sees wide use within the network measurement community, we hope that our study provokes a renewed discussion on the role of metadata in assessing measurement quality and experiment reproduction.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Rocky K. C. Chang for their feedback. This material is based upon work supported by NSF grant CNS-1054985, by DHS grant BAA 11-01 and AFRL grant FA8750-12-2-0328. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, DHS or AFRL.

REFERENCES

[1] M Allman. 2013. On changing the culture of empirical Internet assessment. *ACM SIGCOMM Computer Communication Review* 43, 3 (2013), 78–83.

[2] M. Allman and V. Paxson. 2007. Issues and etiquette concerning use of shared measurement data. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*. 135–140.

[3] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. 2006. Avoiding traceroute anomalies with Paris traceroute. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*. 153–158.

[4] F. Baccelli, S. Machiraju, D. Veitch, and J. C. Bolot. 2007. On optimal probing for delay and loss measurement. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*. 291–302.

[5] V. Bajpai, A. W. Berger, P. Eardley, J. Ott, and J. Schönwälder. 2016. Global Measurements: Practice and Experience (Report on Dagstuhl Seminar# 16012). *ACM SIGCOMM Computer Communication Review* 46, 1 (2016), 32–39.

[6] V. Bajpai, S. Eravuchira, and J. Schönwälder. 2015. Lessons learned from using the RIPE Atlas platform for measurement research. *ACM SIGCOMM Computer Communication Review* 45, 3 (2015), 35–42.

[7] R. Beverly. 2016. Yarrp'ing the Internet: Randomized High-Speed Active Topology Discovery. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*.

[8] BGPmon [n. d.]. BGPmon Deployment. http://bgpmon.netsec.colostate.edu/about.html. ([n. d.]). Accessed May 2017.

[9] CAIDA [n. d.]. Ark Monitor Site Information. http://www.caida.org/projects/ark/siteinfo.xml. ([n. d.]). Accessed May 2017.

[10] CAIDA [n. d.]. CAIDA Ark project. http://www.caida.org/projects/ark/. ([n. d.]). Accessed May 2017.

[11] Center for Applied Internet Data Analysis (CAIDA). [n. d.]. DatCat: Internet Measurement Data Catalog. http://www.datcat.org. ([n. d.]). Accessed May 2017.

[12] Center for Applied Internet Data Analysis (CAIDA). 2009. How to Document a Data Collection. http://www.caida.org/data/how-to/how-to_document_data.xml. (March 2009). Accessed May 2017.

[13] C. Dovrolis, K. Gummadi, A. Kuzmanovic, and S. D. Meinrath. 2010. Measurement lab: Overview and an invitation to the research community. *ACM SIGCOMM Computer Communication Review* 40, 3 (2010), 53–56.

[14] J. Heidemann, Y. Pradkin, R. Govindan, C. Papadopoulos, G. Bartlett, and J. Bannister. 2008. Census and Survey of the Visible Internet (extended). *ISI-TR-2008-649* (2008).

[15] T. Holterbach, C. Pelsser, R. Bush, and L. Vanbever. 2015. Quantifying interference between measurements on the RIPE Atlas platform. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*. 437–443.

[16] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering* 9, 3 (2007), 90–95.

[17] D. Kotz and T. Henderson. 2005. CRAWDAD: A community resource for archiving wireless data at Dartmouth. *IEEE Pervasive Computing* 4, 4 (2005), 12–14.

[18] D. Kotz, T. Henderson, I. Abyzov, and J. Yeo. [n. d.]. CRAWDAD: A Community Resource for Archiving Wireless Data At Dartmouth. https://crawdad.cs.dartmouth.edu. ([n. d.]). Accessed May 2017.

[19] B. Krishnamurthy, W. Willinger, P. Gill, and M. Arlitt. 2011. A Socratic method for validation of measurement-based networking research. *Computer Communications* 34, 1 (2011), 43–53.

[20] M. Luckie. 2010. Scamper: a scalable and extensible packet prober for active measurement of the Internet. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*. 239–245.

[21] M-Lab [n. d.]. M-Lab. https://www.measurementlab.net. ([n. d.]). Accessed May 2017.

[22] R. K. P. Mok, W. Li, and R. K. C. Chang. 2015. Improving the packet send-time accuracy in embedded devices. In *Passive and Active Network Measurement Conference*. 332–344.

[23] Nagios [n. d.]. Nagios: The Industry Standard in IT Infrastructure Monitoring. https://www.nagios.org. ([n. d.]). Accessed May 2017.

[24] C. Partridge and M. Allman. 2016. Ethical considerations in network measurement papers. *Commun. ACM* 59, 10 (2016), 58–64.

[25] V. Paxson. 2004. Strategies for Sound Internet Measurement. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*. 263–271.

[26] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush. 2013. From Paris to Tokyo: On the suitability of ping to measure latency. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*. 427–432.

[27] RIPE [n. d.]. RIPE Atlas - RIPE Network Coordination Centre. https://atlas.ripe.net. ([n. d.]). Accessed May 2017.

[28] C. Shannon, D. Moore, K. Keys, M. Fomenkov, B. Huffaker, et al. 2005. The Internet Measurement Data Catalog. *ACM SIGCOMM Computer Communication Review* 35, 5 (2005), 97–100.

[29] J. Sommers. 2015. Lowering the Barrier to Systems-level Networking Projects. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE)*. 651–656.

[30] J. Sommers and P. Barford. 2007. An active measurement system for shared environments. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*. 303–314.