ELF: High-Performance In-band Network Measurement

Joel Sommers
Colgate University
jsommers@colgate.edu

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

***Abstract*—Over the past decades, active measurements have been used to gain a deep and broad understanding of routing, latency, packet loss, etc. Unfortunately, typical active measurements are ill-suited for elucidating the performance of individual application flows due to route changes, load balancing, transient queues, and other dynamic effects. Recent efforts have identified in-band measurement, in which probes are injected into an existing application flow, as a promising approach for gaining insight into network behaviors that affect application flows. However, the use of libpcap by these efforts poses significant performance bottlenecks and is at odds with high-fidelity measurements.**

In this paper, we explore a new implementation pathway for in-band application flow monitoring: the extended Berkeley Packet Filter (eBPF), which enables safe programs to be run within the OS kernel. We develop an eBPF-based in-band flow monitoring tool called ELF that sends hop-limited probes within an existing flow. We compare the performance of our eBPF-based approach with the use of libpcap, finding that libpcap introduces undesirable high variability into the probe emission process. We illustrate the potential of ELF by monitoring hourly Network Diagnostic Tool (NDT) throughput measurements to 12 Measurement Lab destinations for one week. We observe that at least 90% of routers traversed by the in-band probes respond positively, with no apparent rate limiting. We examine how the hop-by-hop evolution of network queues is exposed using ELF in-band probes, illustrate the impact of mid-flow route changes, and show that load balancing may inequitably affect throughput.

***Index Terms*—eBPF, application flow performance**

I. Introduction

Active probe-based measurements provide the foundation for understanding latency, packet loss, routing, available bandwidth, throughput, and other characteristics of the Internet. A great deal of work over the past decades has gone into developing a variety of active measurement techniques. Ongoing projects such as RIPE Atlas [14], CAIDA Ark [6] and SLAC PINGer [13] publish active measurement data that are widely used to derive insights into Internet performance and behavior.

Unfortunately, explaining or diagnosing the performance of *individual application flows* is generally not possible using typical path-oriented active measurements. For example, an end-to-end router-level path identified using traceroute may not be the same path used by an application flow between the same endpoints due to load-balancing and other effects [16]. Similarly, latency measurements derived from tools (*e.g.*, ping) may be quite different from latencies that an application flow experiences along the same end-to-end path [36].

An approach taken by prior efforts to monitor network properties from the perspective of application flows is known as *in-band measurement*, in which probes are injected into an existing flow. For example, TCP sidecar [38] and service traceroute [35] introduce into an existing application flow packets with the same essential characteristics, ensuring that those packets will follow the same router-level path and will experience similar network conditions as application packets. These prior works have focused on measuring the specific router-level paths of application flows by setting TTLs in outgoing measurement probes (henceforth referred to as *probes*) in a similar manner as the traceroute method and collecting ICMP time exceeded messages [31]. Although latency measures can also be gathered using these tools, they have used libpcap (for packet injection and capture) which may introduce undesired variability in the probe emission process due to context switching between the kernel and the user spaces, among other overheads [33]. An alternative approach of direct implementation within the OS kernel is neither portable nor sustainable.

In this paper, we explore the use of a new implementation pathway for in-band measurement: *the extended Berkeley Packet Filter* (eBPF) [9]. eBPF is a recent innovation with good support in the Linux kernel and makes it possible to embed safe programs at pre-defined kernel tracepoints (see § II). Specifically, eBPF makes it possible to perform in-band flow monitoring safely within the OS kernel, enabling probes and application packets to be emitted closely spaced in time. Consequently, the performance (*e.g.*, latency and loss) experienced by application traffic is highly likely to be experienced by probes—a key insight used in this work.

Using this insight, we develop a tool called ELF[1] (eBPF teLemetry Framework) for in-band flow monitoring using the bcc (BPF compiler collection) toolchain [4]. The tool relies on two eBPF programs in the OS kernel: one uses *cls-bpf* for egress packet processing and the other uses *XDP* for ingress processing [3], [28]. On the egress path, packets destined to addresses of interest are periodically *cloned*, truncated in size to be of minimum length, and the TTL modified. On the ingress path, payloads of ICMP time exceeded messages are matched with probes. Moreover, these probe responses

[1]Source code for ELF is publicly available at https://github.com/jsommers/ELF.

978-3-903176-40-9 ©2021 IFIP

are discarded within the XDP component, thus they do not impose any further load on the measurement host; this is a distinctive feature of `ELF`. A Python controller program (in userspace) coordinates these two eBPF programs and extracts latency measurements from in-kernel BPF maps.

We show in a set of benchmark experiments that `ELF` emits probes (packet clones) closely spaced in time with application packets, generally less than 20 microseconds apart on Gigabit Ethernet. We compare these spacings with a libpcap-based program, which emits probes with higher average spacings and much higher variability. We also use `ELF` to trace `iperf3` flows in a controlled setting, comparing `ELF`-gathered latency measures with those gathered with the standard `mtr` tool. We find that `ELF` accurately captures dynamic hop-by-hop latency. Although we find that our eBPF-based tool enables much higher-fidelity measurement than has been previously possible, the use of eBPF imposes new constraints. For example, probes cannot be emitted at arbitrary points in time because eBPF programs at the cls-bpf and XDP tracepoints can only be invoked in response to the arrival or departure of a packet.

We use `ELF` to launch Network Diagnostic Tool (NDT) [12] throughput measurements from 5 Cloudlab [20] locations to 12 measurementlab.net (M-Lab) destinations around the world. Using probe rates of no more than 100 probes per second, we observe that at least 90% of routers traversed with our probes do no apparent rate-limiting or throttling of ICMP time exceeded responses. In addition, we find that the latency measurements gathered using `ELF`, while noisy, nonetheless reveal the evolution of router queues along a path and evidence for congestion. In particular, we observe evidence for congestion both *between* service providers and *within* service providers. Moreover, due to the relatively high-resolution measurements, we observe several instances of flow disruption due to route changes. The nature of our measurements enables us to show that, depending on the path, load balancing does not necessarily result in equal treatment of application flows. Overall, our results indicate that the in-band flow monitoring capability enabled by `ELF` can provide a new level of insight into network flow behavior without requiring access to intermediate routers.

II. Background and Related Work

In this section, we review work related to our efforts and provide some background on eBPF its features that are relevant for network measurement.

A. Prior Efforts and their Limitations

A packet filter (PF) is a kernel agent which provides unfettered access to raw (network) packets by exposing them to the userspace. Several PFs were proposed in the 1980s including Network Interface Tap [27], DEC's Ultrix PF [8], and CMU/Stanford PF (CSPF) [29]. In 1993, McCanne *et al.* identified several performance bottlenecks with these efforts and proposed BSD PF (BPF) [33]. For example, `tcpdump` uses BPF-based filtering and uses the libpcap library to attach to a network tap/interface to capture and analyze the network traffic. Building on these earlier efforts, in 2013 extended BPF (eBPF) [9] was proposed with a new virtual machine model, a vastly expanded instruction set, new kernel hooks, and other new facilities. The programmability enabled by eBPF resembles other efforts in on-device computation and software-defined networking, *e.g.* [21], [19], [42].

Many complementary approaches have been taken to reduce interrupt load and overheads of kernel-user-mode context switches that arise from using libraries like libpcap. For example, *coalescing interrupts* and "batching" packet arrivals [34] and *zero-copy buffering* can effectively reduce system load [22]. A more extreme approach is *kernel-bypass*, in which a special kernel-level driver passes packets directly to a user-mode program, avoiding the packet processing pipeline in the OS kernel and overheads of its generality. The recent DPDK [7] system takes this approach. Interestingly, the eXpress Data Path (XDP) component of eBPF has been described as an in-kernel competitor with DPDK in terms of high-throughput packet performance [28], [3].

Related to the in-band measurement approach taken in `ELF`, the notion of injecting measurement probes into an existing application traffic stream has been explored primarily in the context of identifying the interface-level network paths of application flows [38], [35]. The recent work of Ahmed *et al.* uses in-band measurement in a more general way to implement available bandwidth estimation within packets of the application flow [15] but their approach shares the drawbacks of approaches using libpcap since probe packets are emitted from userspace. Lastly, the per-hop latency measurements collected using `ELF` are similar to the time-series latency probe (TLSP) of Luckie et al. [30]. The key difference is that our measurements necessarily follow the same interface path as an application flow, thus any load-balancing effects are explicit and known. Our work also has similarities to Tulip [32]. The key differences are that Tulip measurements are "out-of-band" and measurement packets may be blocked and/or not follow the same link-level path as application packets, and that it is implemented in userspace with libpcap.

To the best of our knowledge, no prior research efforts leverage eBPF to perform active network measurement, although there are example programs that perform basic passive measurement tasks [2], [5].

B. eBPF Overview

The Extended Berkeley Packet Filter (eBPF) is based on the earlier Berkeley Packet Filter, which was designed as a simple virtual machine instruction set specifically for packet filtering [33]. The earlier BPF is now being referred to as "classic" BPF to distinguish it from its modern successor [3]. Development of Linux kernel eBPF facilities has continued at a steady pace since its first appearance in December 2014; a history of versions and main features added is maintained by the BPF compiler collection (bcc) project [4].

eBPF expands greatly on the notion of an in-kernel virtual machine instruction set. In particular, the VM instruction set has been broadened and generalized for a variety of tasks (*i.e.*, not just packet filtering). Upon loading into the kernel, programs are verified to ensure that they (1) cannot have invalid memory references and (2) all code paths terminate. Termination is verified by disallowing loops that cannot be fully unrolled and by evaluating the control flow graph. The instruction set resembles that of modern processors, which facilitates just-in-time compilation to the host architecture which is done by default.

eBPF programs are invoked in response to different kernel events. The eBPF *program type* restricts the range of kernel hooks where a program can be installed. For active network measurement, the two most relevant program types are eXpress Data Path (XDP) [28] and Linux's `tc` (traffic control) layer [3]. The XDP program type can *only* be invoked upon packet ingress; the program is invoked in the device driver *before* any parsing or host processing. `tc` eBPF programs, on the other hand, can be involved either on packet ingress or egress and operate on parsed packets (Linux `sk_buff`). For both program types, eBPF helper functions facilitate packet manipulation, obtaining timestamps, etc. These calls are translated into direct in-kernel function calls upon JIT compilation. Figure 1 shows the general architecture of where XDP and `tc` eBPF programs reside in the kernel.

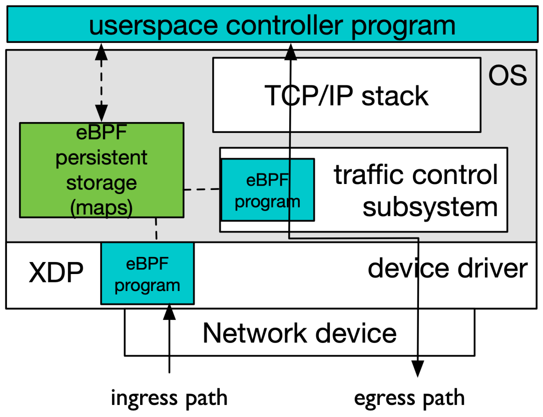


Fig. 1. Diagram showing primary locations where eBPF programs can be executed (XDP and `tc`), as well as how a user-space control program (via bcc) interacts with eBPF programs and in-kernel data storage (BPF maps).

Since eBPF programs are invoked only in response to packet arrivals (on ingress for XDP or ingress/egress for `tc`), there are additional facilities for persistent data storage and access within the kernel: *BFP maps*. In-kernel maps such as hashtables, arrays, and longest-prefix match tries can be accessed by programs running in user mode, and there are also per-cpu variants of some BPF map structures, enabling lock-free access. Low-level eBPF programming can be difficult, largely due to the complex API and safety requirements. There are several projects to create simplified front-ends for the "raw" eBPF C-based API, such as bcc [24]. In our work, we use the bcc (BPF Compiler Collection) tools, which enable the creation of a user-space control program in Python. bcc has interfaces for compiling and installing eBPF programs (including XDP and `tc` programs), interacting with in-kernel BPF maps, and collecting kernel debug information.

Because of the opportunities afforded by safe, in-kernel packet processing, eBPF is already making an impact in operational networks. For example, it has been reported that Facebook has 40 eBPF programs active on *every server in their infrastructure* and that Netflix runs 14 eBPF programs on *every one of their Amazon EC2 instances* [25]. Moreover, a number of networking projects leverage eBPF facilities, *e.g.*, [18], [17], [44]. An excellent compilation of eBPF resources and projects that use eBPF is available on GitHub [1].

III. Design and Implementation of ELF

In this section, we describe the design and implementation of an eBPF-based in-band active measurement tool called ELF.

A. *Design Approach: In-kernel In-band Measurement*

The main idea of our approach is to create measurement probes by periodically cloning application packets, modifying the TTL/hop count, truncating them in size, and injecting them into the application flow. The probes thus have the same 5-tuple (source/destination addresses, protocol, and source/destination port numbers), ensuring that they will follow the same router-level path as the application flow and bypass operational and management policies of WAN operators.

Prior work in in-band measurement [38], [35] has focused on measuring the specific router-level paths of application flows by setting TTLs in outgoing probes in a similar manner as the traceroute method and collecting ICMP time exceeded messages [31]. Although latency measures can also be gathered using these tools, they have used libpcap (for packet injection and capture) or similar means which introduces undesired variability in the probe emission process due to context switching between the kernel and the user spaces, among other overheads [33]. Similarly, the approach of [15] requires a hybrid user-/kernel-space approach which introduces variability into the measurement process and undesirable context-switching overhead. In our approach, we aim to solve these problems as well as to improve both measurement scalability and programmability.

Novelty of our Approach. Compared to prior in-band measurement efforts, our approach is novel: ELF is implemented with eBPF and thus its core functionality resides in the OS kernel, enabling probes and application packets to be emitted closely spaced in time, *e.g.*, within 20 μsec of application packets. Consequently, the performance (*e.g.*, latency, and loss) experienced by application traffic is highly likely to be experienced by probes—a key insight and feature of ELF. Furthermore, as we discuss below, ICMP time exceeded responses are eliminated within the XDP component *before any processing by the OS kernel*, thus reducing the impact on the measurement host. These reasons are why we argue that ELF is well-suited for measurement and diagnosis of application flow performance.

B. ELF *Implementation Details*

1) General Operation: ELF is implemented using bcc [4]. Given a network interface to use and destination hostnames of interest, a Python control program uses bcc to compile and install eBPF programs for egress and ingress packet processing. Egress handling and sending probes are done within the `tc` subsystem and ingress handling is done within XDP. Because `tc` eBPF programs can only be invoked in response to outgoing or incoming packets, probe emission is dependent on the presence of application-level traffic. In our ongoing work, we are examining the possibility of using more generic events to trigger probe emission, which will require modifications to the Linux kernel eBPF subsystem.

When `ELF` is started, the user can specify a probing rate, but this is, in effect, a *maximum* probe rate. The probe rate r can be specified as *per-hop*, in which case `ELF` attempts to meter probes out so that the rate observed at each hop along a path $\approx r$; it can also be specified as *global*, in which case the probe rate observed at an individual hop will be $\approx \frac{r}{pathlen}$.

There are several BPF maps used to track destinations of interest, manage information about destinations, and provide temporary storage for measurements. One map associates each destination IP address (v4 or v6) of interest with a unique integer identifier; this integer is used as an index into a BPF array of structs, where each struct contains a nanosecond timestamp of the most recent probe sent to a given destination, an estimated number of hops to the destination (inferred from the TTL/hop count in packets received from the destination), the next probe sequence to use, and the next hop number toward the destination to probe. A separate BPF map stores information about probes that have been sent, but for which responses have not yet been received. Lastly, another map stores information about received probes, including timestamps, IP address of the responding host, received TTL/hop count, etc. This last map is a *per-cpu* map so that no locks are required to update the map as probe responses are received.

2) Probe Send Path: On packet egress, a lookup in the destination IP address BPF map is performed to check whether a probe should be emitted to satisfy a configured probe rate. If a probe should be emitted, `ELF` egress-handling code stores the integer BPF array index associated with the destination in the `skb` metadata and control is handed to `ELF` protocol-specific (ICMP, TCP, UDP) code. The packet is then *cloned* and emitted. As a result, the original unmodified application packet is sent on its way soon after identifying that a probe should be created. `ELF` code then truncates the packet clone in size to be of minimal length (*e.g.*, 40 bytes for IPv4 TCP) in order to minimize measurement overhead, and the IP TTL/hop count is modified in such a way as to cycle over each hop between the source and destination. Also, checksums are recomputed, and a sequence number and a nanosecond-scale timestamp are recorded and stored on another map. Note that `ELF` code inspects the `skb` metadata early on to determine whether it has been set with a destination identifier; if so, this packet is ignored to avoid re-cloning. Although `ELF` currently truncates all packets by default, we plan to make this capability optional for users who wish to use the full application packet as a measurement probe (albeit with an appropriately set TTL/hop count). To enhance programmability, we are also creating the capability for a user to provide additional eBPF code to modify the probe packet contents in any desired way, thus overriding built-in functionality from `ELF`.

3) Probe Receive Path: Within the network, ICMP time exceeded messages are typically generated at router interfaces where TTLs expire, and these messages are received in the ingress (XDP) component where a timestamp is recorded and measurement results are added to the per-CPU BPF map. For incoming ICMP time exceeded messages that match an outgoing probe, the message is, by default, *dropped* within the XDP processing path so as not to impose any additional processing load on the host. This behavior (to drop incoming ICMP time exceeded messages) is configurable; as a debugging aid, it can be helpful to observe those packets from other programs. Moreover, it is important to note that this ability to eliminate measurement traffic *before it enters the OS networking stack* is a distinct advantage of eBPF-based measurement over using libpcap and similar approaches. As long as the application that was given to the Python control program continues to run or until interrupted, paths to destinations of interest will continue to be monitored. Periodically, the Python control program reads result data from BPF maps and appends these data to a CSV file. To further enhance programmability, we are creating the capability for a user to provide additional eBPF code to select packet contents of interest for extraction from the received ICMP time exceeded message.

4) Overheads: Lastly, we note that eBPF programs, before being installed at a kernel hook, are compiled to eBPF bytecode then JIT translated into machine code for execution. The bcc framework enables the machine code to be shown at installation time for debugging purposes. On the packet egress path, there are fewer than 100 instructions executed to determine whether a packet should be cloned, and a total of 814 instructions in the egress path-handling code (some of these instructions are calls to access/update BPF maps or are calls to BPF helper functions). On the ingress path, there are 50 instructions necessary to determine whether an incoming packet is an ICMP time exceeded response of interest; a total of 657 instructions are on the ingress code path. These program sizes (which have not been carefully optimized) — along with the safety guarantees that come with eBPF — provide concrete support for the use of eBPF in network measurement and give context for results from experiments we describe below.

An alternative design we considered is direct implementation in the kernel, which has been employed in prior work (*e.g.*, [39]). While such an approach has lower overhead by virtue of avoiding user/kernel boundary crossings, the implementation would be tightly coupled to specific kernel versions and fundamentally non-portable. A user-space implementation (*cf.* [15], [35]) using libpcap or similar would be portable, but as we discuss in the next section, the performance cost is too high for even modest packet rates. The eBPF approach we chose and advocate strikes a balance between high-performance and implementation using a stable and, in theory, portable API.[2]

IV. Evaluation

A. Laboratory Experiments

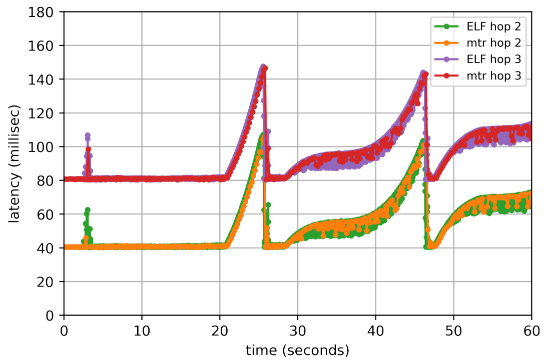
In this section, we evaluate the performance of ELF in a controlled laboratory setting. We first describe the results of experiments to compare timing overheads of using eBPF for in-band active measurement with a baseline libpcap-based approach. We then examine hop-by-hop latency measurements collected using ELF.

1) Evaluation of Overheads: As noted above, the typical approach taken in prior work on in-band active measurement has been to create a userspace program that uses libpcap to inject packets into an existing flow [38], [35]. To compare timing overheads, we created a simple experiment setup in which we could control input packet rates and observe the performance of libpcap and eBPF. We wrote programs using libpcap and eBPF/bcc to clone *every* $k = 100^{th}$ UDP packet[3]. Specifically, each program clones an incoming UDP packet, sets the TTL in the clone to a fixed low value, and recomputes the IP checksum before forwarding the clone to the destination. We directly connected three hosts in a linear A–B–C arrangement using Gigabit Ethernet. Host A was configured to generate small packets (60 bytes, including Ethernet header) at a configurable rate using the Linux pktgen kernel module [43]. Host B was configured to use either the libpcap or eBPF cloning program. Host C was configured with an XDP program that received incoming packets and timestamped them (at nanosecond scale).

Our results from these experiments (not shown in detail due to space constraints) reveal that for the eBPF program, the median time difference between a packet and its clone is below 20 μsec for all offered loads and there is little variability in the differences. For the libpcap-based program and the *lowest* offered packet rates (below 32 kpps) the median time difference between a packet and its clone is also low ($\approx$ 20 μsec), but things change at 32 kpps and above. With modest offered packet rates ($\geq$ 32 kpps) for the libpcap-based program, although the median spacing difference between a packet and its clone remains low, there are increasing levels of packet loss, and spacing variability also significantly increases, *e.g.*, at 32 kpps the 90^{th} percentile is 90 μsec and the variance is 55 μsec for libpcap compared with 21 μsec and 0.115 μsec, respectively, for eBPF. Since these programs were designed to clone every 100^{th} packet, the 32 kpps offered load should result in a probe rate of merely 320 packets/sec on an otherwise idle host, thus *any* packet loss is surprising. Even more surprising is that for the libpcap experiments with offered loads of 32 kpps and above, we observed some cloned packets to arrive *before* the original packet. In no instances did this occur with the eBPF program since it was not subject to the same scheduling effects as a userspace program. Moreover, our observations were similar for multiple different host systems and networking hardware setups, thus these limitations appear to be fundamental to libpcap. We conclude that while libpcap is sufficient for collecting router-level path information about application flows, it is inadequate for measurement of delay at hops along a path or collection of other performance measures.

[2]We note that although the Linux kernel is the context for many of the advances in the eBPF landscape, there is also an implementation for the FreeBSD kernel as well as platform-independent implementations in userspace [1].

[3]This was an arbitrary choice; our results were the same for other k.



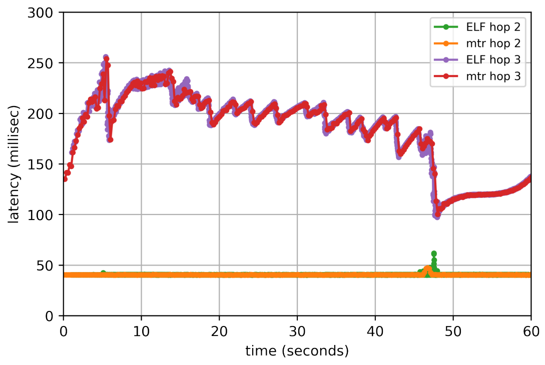


Fig. 2. Hop-by-hop latency measurements for two configurations of controlled lab experiments. Latency measurements computed from ELF and mtr are shown for two intermediate hops of each experiment.

2) Evaluation of Hop-by-Hop Latencies: In another set of controlled laboratory experiments, we evaluated the per-hop latency measures collected by ELF. We used a linear topology, with Linux hosts directly connected in an A–B–C–D–E arrangement. On links B–C and C–D we used the Linux traffic control (`tc`) subsystem to impose one-way delays (in each direction) of 20 milliseconds, leading to a round-trip time (RTT) of 80 milliseconds between hosts A and E. On host E we ran an `iperf3` [10] server, and on host A we ran an `iperf3` client along with ELF to inject measurement probes in the `iperf3` packet stream. For `iperf3`, we used 4 parallel flows and ran 60-second experiments with the primary flow of traffic from A$\rightarrow$E ("upload") as well as with the primary flow of traffic from E$\rightarrow$A. Additionally, we introduced cross-traffic in 3 different configurations: (1) on link B-C, (2) on link C-D, and (3) on both links B-C and C-D. Lastly, we also launched the standard `mtr` tool [11], which combines functionalities of traceroute and ping, on host A to collect continuous (every second) ICMP-based traceroute measurements along the path to host E. We note also that the Linux hosts in this setup used the default TCP CUBIC congestion control.

Figure 2 shows results from two of the 8 experiments we ran. In these experiments, `ELF` was configured to emit probes at most every 10 milliseconds (100 probes/sec)[4]. Note that with this choice of probe rate, the measurement overhead is an extremely low 32 kb/s[5]. The top plot of Figure 2 shows latency measurements collected from `ELF` and `mtr` for hops 2 (host C) and 3 (host D) without cross-traffic and with the primary direction of traffic flow being A$\rightarrow$E. We observe in the figure the baseline 40 milliseconds RTT to host C (hop 2) and 80 milliseconds RTT to host D (hop 3). We also observe substantial additional latency due to packet queuing which happens overwhelmingly at hop 2. Interestingly, we observe the cumulative effect of latencies on probes that expire at hop 3, *i.e.*, latencies measured to hop 3 $\approx$ 80 milliseconds + queuing delay at hop 2. Lastly, we also observe that the `ELF` probes, which are TCP packets because of the transport protocol used with `iperf3`, and the `mtr` (ICMP) probes are nearly identical.

The bottom plot of Figure 2 shows results from an experiment where the primary direction of traffic is E$\rightarrow$A, with cross-traffic only on the C-D link. We again observe that both `ELF` and `mtr` measurements are nearly identical. We further observe that with cross-traffic only on the C-D link that latency measurements for the B-C link are largely at the minimum RTT of 40 milliseconds. There is also an interesting event at about 47 seconds into the trace when the end-to-end `iperf3` flows go into timeout and loss recovery and a burst of packets are subsequently released during slow start, causing a short-lived queue at hop 2. The results shown in these plots provide context for per-hop queuing that we observe in our wide-area experiments, described below, and relate directly to our goal of collecting fine-grained performance measures for diagnosing application flow performance.

B. Internet Experiments

In this section, we describe a week-long experiment in which we monitored flows created using NDT[6], which is in use by M-Lab [12] and has been used in recent studies of the Internet congestion [40], [41]. We focus on particular examples that illustrate how high-fidelity in-band measurements can help in interpreting and contextualizing flow performance.

1) Data collection: We collected a week-long data set between 2 March–9 March 2020 using NDT in conjunction with in-band flow measurements. For each data set, we launched the NDT client from a host in one of 4 CloudLab data centers [20] (Utah, Wisconsin, Clemson, and Paris) as well as a fixed university location in the USA. The NDT client connected to 12 M-Lab server locations distributed across the world. Specifically, we used M-Lab servers in Auckland, New Zealand; Amsterdam, Netherlands; Dallas-Fort Worth, TX, USA; Dublin, Ireland; Miami, FL, USA; New York, NY, USA; Nairobi, Kenya; Seattle, WA, USA; Singapore; Toronto, Canada; Vancouver, Canada; and Vienna, Austria. Measurements to each M-Lab location were collected in series every hour. For each experiment to each location (168 for each location and from each of the 5 launch sites), we stored the output of NDT as well as the in-band probe data from `ELF`.

The NDT application creates several flows over the span of an individual test. We note that many paths between the five client sites and the various M-Lab servers have some form of load balancing. Because `ELF` creates probes (packet clones) based on a destination IP address and a maximum probe rate, *all* flows for a given test are monitored. The CSV data file created by the Python control program indicates TCP ports, etc. to disambiguate specific flows that are monitored. For all experiments, `ELF` was configured with a maximum probe rate of 100 probes/sec per hop; as noted above this choice of probe rate leads to a measurement probe rate of 32 kb/s per hop.

In addition to the week-long data set, we collected measurements using different settings of maximum probe rate, and with a broader set of NDT servers. Moreover, we collected data from experiments in which we ran the NDT client with `ELF` followed by running NDT alone, to evaluate whether the probes had any measurable impact on throughput, flow completion times, etc. Although we do not show detailed results due to space limits, *we found no statistical differences between the NDT output with or without in-band monitoring*—an observation consistent with results observed by the authors of service traceroute [35] and flowtrace [15].

2) Router Responsiveness: We first comment on how responsive routers are to hop-limited in-band probes with ICMP time exceeded messages since `ELF` relies on receiving these packets. Our results (not shown in detail due to limited space) vary according to the site from which measurements are launched. For the Wisconsin CloudLab site, for example, only about 5% of emitted probes lack responses, whereas for the university site there are no responses for about 27% of emitted probes (the first-hop router, within the university network, rate-limited ICMP responses and was responsible for many of these non-responses). Across all sites, at least 90% of all routers observed respond without any apparent throttling of probes or probe responses.

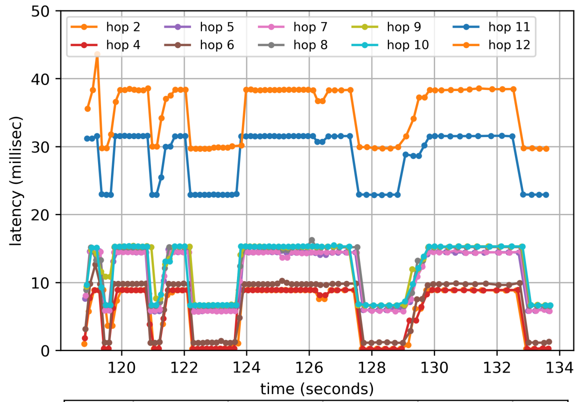
Overall, our results are similar to those of Ravaioli *et al.* [37]: low levels of rate-limiting with probe rates that we use. We expect that, similar to Ravaioli *et al.*, we would see higher rates of non-responses with higher probing rates and plan to investigate this in detail in future work. On the whole, our results indicate that an in-band measurement approach is viable in the wide area for collecting fine-grained performance measurements for application flow diagnosis.

3) Evaluation of Hop-by-Hop Latencies: In Figure 3 we show hop-by-hop RTT latencies measured using `ELF` between 2 different NDT client sites and different M-Lab servers. Interestingly, we observe increasing delays at different hops, which we attribute to the growth of router queues as the TCP flow increases its sending rate; *c.f.*, Figure 2. For the top plot

[4]Again, this was an arbitrary choice; we used it to be consistent with our wide-area Internet experiments described below.

[5]Each probe packet is 40 bytes, not including layer 2 headers.

[6]We do not claim that NDT is ideal for throughput measurement; we use this tool simply to generate longer-lived TCP flows across the wide-area.



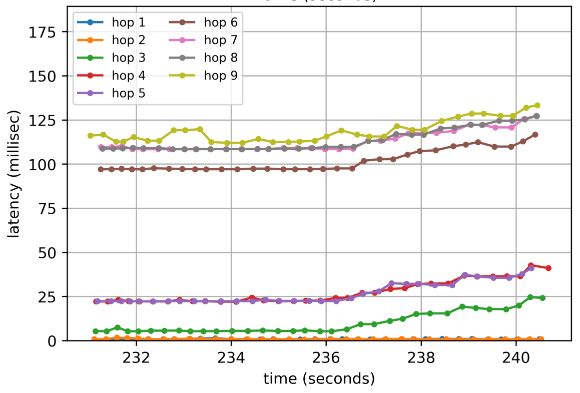


Fig. 3. Evolution of hop-by-hop round-trip times measured using ELF in-band probes. Top plot shows results for one NDT upload flow between the Paris Cloudlab site and the Vienna, AT M-Lab site; bottom plot shows one NDT download flow between the Wisconsin Cloudlab site and the Dublin, IE M-Lab site.

of Figure 3, which shows results from an outbound/upload flow between the Paris Cloudlab site and an M-Lab server in Vienna, AT, there are four networks traversed by the flow. We observe prominent queuing delays at hop 2 of ≈ 8 milliseconds at maximum, which is within Universite Pierre et Marie Curie, and the last hop before traffic enters the Parisian district network for research and education. We infer that this flow is largely constrained by congestion in the network, close to the traffic source. Similarly, the bottom plot shows results for an inbound/download flow between the Wisconsin CloudLab site and an M-Lab server in Dublin, IE. We observe in the plot that for the first 4 seconds of the flow's lifetime, it is likely constrained by TCP *rwnd* since per-hop latencies remain relatively flat. After time 236, however, we see a steady increase in delay at hop 3 (which is the last hop within the University of Wisconsin network before Internet2) which suggests an in-network constraint on throughput.

RTT measurements gathered using hop-limited probes are fairly well understood to be quite noisy (*e.g.*, see [23], [37], [26]) but details in the plots of Figure 3 strongly indicate that the measured latencies are not simply noise. In particular, we observe that inflated latencies at one hop have an accretive effect on latencies observed at later hops, similar to what we see in the controlled lab experiments in Figure 2. For example, both plots of Figure 3 show that inflated latencies observed at one hop cause similarly inflated latencies at later hops (*e.g.*, hop 2 on all later hops in the top plot).

Moreover, it has been pointed out in earlier work, *e.g.*, [23], that the timescale of noise relative to RTT is critical. In particular, both of these prior works observed that in practice, noise may not have a significant effect on conclusions drawn from tools such as `pathchar`. Nonetheless, we are investigating methods for quantifying the effects of noise in our ongoing work, and are also looking at methods for automatically identifying the location of congestion (*i.e.*, which hop along a path) over the lifetime of a flow, noting that that location may change over time. Further, we note again that the RTT measurements collected using ELF bear similarity to the Time Series Latency Probe (TLSP) method of [30], with two key differences: TLSP targets a particular pair of interfaces along a path, and the probes are emitted "out of band". Because of the out-of-band nature of TLSP, it may not be possible to relate the TLSP measurements with the behavior of an application flow due to load-balancing effects and the possibility that ICMP probes take a different path than application flows.

4) Observation of Route Changes and Load Balancing Effects: Although we do not show detailed results, detailed hop-by-hop measurements from our wide-area Internet experiments with ELF revealed other important performance impacts. First, we observed several interdomain route changes while NDT throughput experiments were in progress. These mid-flow route changes each caused statistically significant throughput drops. Second, because ELF in-band flow measurements enable the identification of the interface-level paths followed by individual flows, we can directly compare performance differences that may result from uneven load-balanced paths to the same destination. For example, between the Clemson CloudLab client and the Dallas-Fort Worth, TX M-Lab site, we found statistically significant performance differences across the four observed load-balanced paths (each of equal length in terms of hops). For each of these observations, an application would observe some change in throughput, latency, or packet loss but be oblivious as to *why* the observed changes were occurring. With information from ELF, however, applications could potentially make adjustments to avoid or appropriately react to network impairments.

V. Summary and Future Work

In this paper, we describe and evaluate ELF, an Extended Berkeley Packet Filter (eBPF)-based active measurement tool. ELF periodically clones application packets to produce hop-limited probes along the same path as application flows. We compare our in-kernel eBPF approach with a libpcap-based approach, which has been used in prior work, finding that the libpcap approach can introduce undesirable high variability into the probe emission process. We evaluate ELF in a laboratory environment and use it to monitor hourly NDT through tests to 12 Measurement Lab servers over the span of a week. We analyze the data collected and show that at least 90% of routers traversed by our probes respond to the hop-limited probes with no apparent rate limiting. We illustrate how the evolution of network queues can be tracked with our in-band measurements, and investigate the impacts of mid-flow route changes and load-balancing on throughput.

We believe that ELF and the new approach it takes opens up new areas of inquiry in active network measurement and in our ongoing work we are continuing to enhance and evaluate it. In particular, we intend to investigate the scaling properties of ELF as the number of destinations grows. We also plan to examine ways to enhance programmability and user control of ELF, as well as to revisit the efforts of Govindan and Paxson to examine delays in ICMP packet generation at routers [23] to better understand how to reduce or eliminate noise in the latency measurements.

Acknowledgments

We thank the anonymous reviewers for their feedback and suggestions. This material is supported by the National Science Foundation through awards CNS-1814537 and CNS-1850297.

References

[1] A curated list of awesome projects related to eBPF. https://github.com/zoidbergwill/awesome-ebpf. Accessed March 2021.

[2] BCC tools. https://github.com/iovisor/bcc/tree/master/tools. Accessed March 2021.

[3] BPF and XDP Reference Guide. http://docs.cilium.io/en/latest/bpf/. Accessed March 2021.

[4] BPF Compiler Collection (BCC). https://github.com/iovisor/bcc. Accessed March 2021.

[5] bpftrace. https://github.com/iovisor/bpftrace. Accessed March 2021.

[6] CAIDA Ark. http://www.caida.org/projects/ark.

[7] Data Plane Development Kit. https://www.dpdk.org/. Accessed March 2021.

[8] DEC Ultrix Packet Filter. https://web-docs.gsi.de/~kraemer/COLLECTION/ULTRIX/ultrix_faq.html.

[9] eBPF - extended Berkeley Packet Filter. https://www.iovisor.org/technology/ebpf. Accessed March 2021.

[10] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr. Accessed March 2021.

[11] MTR. http://www.bitwizard.nl/mtr/. Accessed March 2021.

[12] NDT (Network Diagnostic Tool). https://www.measurementlab.net/tests/ndt/. Accessed March 2021.

[13] PingER: Ping End-to-end Reporting. http://www-iepm.slac.stanford.edu/pinger/. Accessed March 2021.

[14] RIPE Atlas. https://atlas.ripe.net. Accessed March 2021.

[15] A. Ahmed, R. Mok, and Z. Shafiq. Flowtrace: A framework for active bandwidth measurements using in-band packet trains. In *International Conference on Passive and Active Network Measurement*, pages 37–51. Springer, 2020.

[16] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 153–158. ACM, 2006.

[17] G. Bertin. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *NetDev Conference*, 2017.

[18] O. Bonaventure. Making our networking stack truly extensible. In *2019 IEEE 44th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*, pages i–i. IEEE, 2019.

[19] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, and G. Varghese. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[20] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[21] N. Feamster, J. Rexford, and E. Zegura. The road to SDN. *Queue*, 11(12):20–40, 2013.

[22] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 218–224. ACM, 2010.

[23] R. Govindan and V. Paxson. Estimating router ICMP generation delays. In *Passive & Active Measurement (PAM)*, 2002.

[24] B. Gregg. Linux Extended BPF (eBPF) Tracing Tools. http://www.brendangregg.com/ebpf.html. Accessed March 2021.

[25] B. Gregg. Extended bpf: A new type of software. http://www.brendangregg.com/blog/2019-12-02/bpf-a-new-type-of-software.html, October 2019.

[26] M. Gunes and K. Sarac. Analyzing router responsiveness to active measurement probes. In *International Conference on Passive and Active Network Measurement*, pages 23–32. Springer, 2009.

[27] D. Hess, D. Safford, and U. Pooch. A Unix network protocol security study: Network Information Service. 1992.

[28] T. Høiland-Jørgensen, J. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 54–66. ACM, 2018.

[29] B. Lampson and R. Sproull. An Open Operating System for a Single-user Machine. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 98–105, 1979.

[30] M. Luckie, A. Dhamdhere, D. Clark, and B. Huffaker. Challenges in inferring internet interdomain congestion. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 15–22. ACM, 2014.

[31] M. Luckie, Y. Hyun, and B. Huffaker. Traceroute probe method and forward IP path inference. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 311–324. ACM, 2008.

[32] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level internet path diagnosis. *ACM SIGOPS Operating Systems Review*, 37(5):106–119, 2003.

[33] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter*, 1993.

[34] J. Mogul. Efficient use of workstations for passive monitoring of local area networks. *ACM SIGCOMM Computer Communication Review*, 20(4):253–263, 1990.

[35] I. Morandi, F. Bronzino, R. Teixeira, and S. Sundaresan. Service Traceroute: Tracing Paths of Application Flows. In *International Conference on Passive and Active Network Measurement*, pages 116–128. Springer, 2019.

[36] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush. From Paris to Tokyo: On the suitability of ping to measure latency. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 427–432. ACM, 2013.

[37] R. Ravaioli, G. Urvoy-Keller, and C. Barakat. Characterizing ICMP rate limitation on routers. In *2015 IEEE International Conference on Communications (ICC)*, pages 6043–6049. IEEE, 2015.

[38] R. Sherwood and N. Spring. Touring the Internet in a TCP sidecar. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 339–344. ACM, 2006.

[39] J. Sommers and P. Barford. An active measurement system for shared environments. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, October 2007.

[40] S. Sundaresan, M. Allman, A. Dhamdhere, and K Claffy. TCP congestion signatures. In *Proceedings of the 2017 Internet Measurement Conference*, pages 64–77. ACM, 2017.

[41] S. Sundaresan, X. Deng, Y. Feng, D. Lee, and A. Dhamdhere. Challenges in inferring internet congestion using throughput measurements. In *Proceedings of the 2017 Internet Measurement Conference*, pages 43–56. ACM, 2017.

[42] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE communications Magazine*, 35(1):80–86, 1997.

[43] D. Turull, P. Sjödin, and R. Olsson. Pktgen: Measuring performance on high speed networks. *Computer communications*, 82:39–48, 2016.

[44] M. Xhonneux and O. Bonaventure. Flexible failure detection and fast reroute using eBPF and SRv6. In *2018 14th International Conference on Network and Service Management (CNSM)*, pages 408–413. IEEE, 2018.