Schooling NOOBs with eBPF

Joel Sommers
jsommers@colgate.edu
Colgate University

Nolan Rudolph
ngr@uoregon.edu
University of Oregon

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

ABSTRACT

While networks have evolved in profound ways, the tools to measure them from end hosts have not kept pace. State-of-the-art tools are ill-suited for elucidating observed network performance impairments and path dynamics, and are susceptible to operational policies of the network. Consequently, the *semantic gap* between the application-view of network performance vs. actual conditions has resulted in network oblivious (NOOB) systems and applications.

To address this NOOB problem, we examine the Extended Berkeley Packet Filter (eBPF) as a new way to improve the practice of gathering fine-grained network telemetry from the edge. More specifically, by leveraging the safe and efficient in-kernel programming mechanism of eBPF, we design a high-performance telemetry framework called `nooBpf` with two tools—namely `noobprobe` and `noobflow`—to quantify the actual network performance from end hosts and offer unprecedented insights into the flow-level performance, including in-network queuing and routing-induced delays. We illustrate the potential of these two tools to address the NOOB problem through a variety of experiments. The results of our experiments strongly suggest eBPF as a promising foundation for high-performance telemetry and for addressing the NOOB problem.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; **Network measurement**.

KEYWORDS

Extended Berkeley packet filter, network measurement

ACM Reference Format:
Joel Sommers, Nolan Rudolph, and Ramakrishnan Durairajan. 2023. Schooling NOOBs with eBPF. In *1st Workshop on eBPF and Kernel Extensions (eBPF '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609021.3609302

1 INTRODUCTION

Ensuring the end-to-end performance of applications is one of the key goals of distributed systems research. End-to-end performance, in turn, depends on the (a) end hosts or data center servers in which the applications are deployed and (b) wide-area networks (WANs) including the protocols and infrastructures connecting those hosts. Traditionally, the performance of WAN paths is characterized by emitting active measurements from end hosts and gathering telemetry information to infer or directly assess end-to-end performance metrics, forwarding paths, configurations, among others (*e.g.*, [15, 22, 31, 32, 38]).

Unfortunately, the tools for elucidating WAN performance impairments and path dynamics have not kept pace with the technological evolution of WAN and edge infrastructures [29, 41]. For one, state-of-the-art measurement tools (*e.g.*, [30, 34]) are based on user-space approaches (*e.g.*, using `libpcap`) and result in telemetry information with suboptimal precision and high measurement overheads, obscuring the network performance characteristics. Second, the user-space nature of these tools warrants a non-trivial amount of changes to the applications and end systems during integration. Third, the tools emit measurement traffic which is susceptible to operational policies of WANs including blocked measurements, traffic shaping, etc. These issues, in sum, have resulted in a *semantic gap* between the application-view of network performance vs. actual performance, making applications and end systems network oblivious (NOOB) [12, 16, 39, 40, 42].

In this paper, we examine the Extended Berkeley Packet Filter (eBPF) as a new way to address the NOOB problem and develop two measurement tools. First is a high-performance in-kernel and in-band active measurement tool called `noobprobe`, which provides three key opportunities for improving the practice of network measurements from end hosts. First, `noobprobe` takes an *in-band* approach and injects probes into an existing application flow, ensuring that those probes will follow the same router-level path as packets in an ongoing application flow. While our prior work [30, 34] has largely 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 [27], they have used `libpcap` (for packet injection and capture) or similar means. As shown in prior work [36], `libpcap` introduces undesired variability in the probe emission process due to context switching between the kernel and the user spaces, among other overheads [28]. Besides, in-band measurements have been shown to be more resilient to the operational policies, *e.g.*, filtering, of providers [11, 30, 34].

Second, `noobprobe` leverages the safe, in-kernel programming mechanism of eBPF. eBPF programs, after safety verification, can be installed at different kernel hooks or tracepoints, and have been envisioned as a flexible and powerful means for system performance tracing and measurement collection [5, 20]. Unlike current user-space measurement tools, `noobprobe` brings the power of eBPF to network measurements and provides visibility into high-precise telemetry information (*e.g.*, flow-level performance, routing-induced delays, etc.) to end hosts. Third, `noobprobe` relies on two eBPF program types: `tc`/*cls-bpf* for egress packet processing [14], and *XDP* for ingress processing [21]. 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,

eBPF '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0293-8/23/09…$15.00
https://doi.org/10.1145/3609021.3609302

payloads of ICMP time exceeded messages are matched with probes, and, importantly, measurement traffic is dropped *prior to entering the OS networking stack*. Thus, the load imposed on the host system on packet ingress is minimal—unlike any prior system described in the literature (see § 2 in [36]). A Python controller program (in userspace) coordinates these two eBPF tracepoints and extracts latency measurements from in-kernel BPF maps.

We evaluate the efficacy of `noobprobe` in a WAN setting. Concretely, we use `noobprobe` to launch Network Diagnostic Tool (NDT) [10] throughput measurements from 5 Cloudlab [18] locations to 12 measurementlab.net (M-Lab) destinations around the world. We collect additional measurements using `noobprobe` with Netflix's fast.com as well as measurements from additional M-Lab destinations. Using probe rates of no more than 100 probes per second, we observe that at least 70% 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 our tool, while noisy, nonetheless reveal the evolution of router queues along a path. 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 `noobprobe` provides a new level of insight into network flow behavior without requiring access to intermediate routers, and can go a long way toward addressing the problem of application network-obliviousness.

The second tool is an XDP-based passive flow collector called `noobflow`. Network flow measurement has a long history within networking and flow records have been used for a variety of management and monitoring tasks such as DDoS detection and detection, capacity planning, traffic engineering, etc. [25]. We describe the implementation of `noobflow` in detail and evaluate it in Cloudlab [18]. Our experiments show that it scales with available CPU cores to perform 10 Gb/s lossless capture on 5 cores, with 60 byte packets and *without* any special hardware acceleration or storage configuration. We find that the main current limitation has to do with maximum map sizes in eBPF, and thus the number of flows that can be concurrently stored.

2 MOTIVATION AND SCOPE

Our research is motivated by the *network obliviousness* (NOOB) of today's distributed systems [12, 16] and key-value stores [39, 40]. This NOOB problem stems from three key issues with state-of-the-art: (a) lack of tools to gather fine-grained telemetry information about the network infrastructures from end hosts, (b) lack of capabilities to collate the gathered telemetry information to understand and infer the state of network and end-host infrastructures in a unified manner, and (c) lack of interfaces to expose the inferred state to the applications. Recent efforts such as MicroMon [24] seek to address the second issue and efforts on application-network interfaces [23] (and see references therein) focus on the third issue. To the best of our knowledge, however, the problem of gathering fine-grained telemetry information about the network path performance characteristics from end hosts is not adequately addressed. It is indeed this problem of gathering fine-grained measurements from end hosts that we focus on in this paper.

To illustrate, consider the widely-used Cassandra [2][1]—a NoSQL database—that replicates data for availability across geographically distributed data centers. Similar to LogDevice [9] and Zookeeper [3], Cassandra uses consistent hashing for replica selection. Also, in an attempt to accommodate network path dynamics, Cassandra uses the *Snitch* mechanism for monitoring network topology and detecting failures. While snitch offers the ability to route requests efficiently, it is completely oblivious to wide-area network (WAN) path dynamics—including routing changes, routing-induced delays, load balancing, link outages, queuing delays at routers, among others—that are known to affect application throughput [33].

Aggravating the problem further is the *semantic gap* between telemetry information measurable by today's Internet measurement tools on WAN paths vs. telemetry information needed for *e.g.*, enhancing the performance of applications. For example, the N replicas of Cassandra deployed at M data center locations connected by a WAN are susceptible to performance variability caused by the aforementioned WAN path dynamics. Unfortunately, despite the usefulness of end-to-end active measurement tools for illuminating path-oriented performance characteristics of a WAN, explaining or diagnosing the performance of *individual application flows* is generally not possible using today's active measurement tools. For example, an end-to-end router-level path identified using `traceroute` may not be the same path used by an application flow from Cassandra between the same endpoints due to load-balancing and other effects [13]. Similarly, latency measurements derived from tools such as `ping` may be quite different from latencies that an application flow of Cassandra experiences along the same end-to-end path [32]. In short, state-of-the-art measurement tools (a) provide coarse-grained information such as latency and bandwidth, making applications unaware of transient network congestion at certain hops, load balancing issues, etc.; (b) warrant changes to internal mechanisms of applications; and (c) are prone to operational policies (*e.g.*, occasionally blocked, traffic shaping) of WAN providers.

What is critically lacking is a system for collecting fine-grained telemetry information from end hosts and exposing that information to applications. The telemetry information should go beyond what is measurable with state-of-the-art network measurement tools and offer new insights about network path dynamics to applications. One example of insight could be to infer queuing delays at different hops between a source and a destination. Another example is to identify the flow-level effects and/or load balancing effects that may have an immense impact on application performance. That is, we should be able to infer whether the application flows are end-host (`rwnd`) or in-network (`cwnd`) limited, and possibly even where in the network the impairment comes from.

Scope. Note that collecting telemetry information using programmable switches (*e.g.*, a path that a packet takes, number of unique flows per second, heavy hitters) is limited to an intra-domain setting (*e.g.*, enterprise network) and does not apply to WAN paths, which are our main focus. Similarly, in this work, we examine a novel high-performance in-band active measurement tool for tracing

[1]While we use Cassandra as an illustrative example, several applications (e.g., from Zookeeper to LogDevice to key-value stores) suffer from the NOOB problem.

application flow performance. The goal of this tracer is to provide unprecedented insights into the flow-level performance telemetry information to address issue (a), pushing the state-of-the-art further beyond what is possible today with user-level in-band measurement implementations (*e.g.*, `libpcap`). Integrating the tool to applications (*e.g.*, Cassandra) is beyond the scope of this work.

3 ACTIVE MONITORING WITH `NOOBPROBE`

In this section, we illustrate the use of eBPF for active measurement by presenting and evaluating a tool for in-band flow measurement.

3.1 Design Approach: In-band Measurement

To tackle the pitfalls of state-of-the-art measurement tools (described in § 2), we take an *in-band* measurement approach to design `noobprobe`. The main idea of this approach is to inject packets (probes) into an existing flow such that the probes 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.

Note that prior work [30, 34] 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 [27]. Although latency measures can also be gathered using these tools, they have used `libpcap` or similar means which may, as shown in [36], introduce undesired variability in the probe emission process due to context switching between the kernel and the user spaces, among other overheads [28]. Similarly, the approach of [11] requires a hybrid user-/kernel-space approach which introduces variability into the measurement process and undesirable context-switching overhead.

Our approach follows the novel approach of ELF [36]: `noobprobe` 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. 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 `noobprobe`, and why we argue that it is well-suited for addressing the problem of application network-obliviousness.

3.2 `noobprobe` Implementation Details

`noobprobe` is implemented using bcc [6]. Given a network interface to use, destination hostnames of interest and an optional program/application to run, 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. When `noobprobe` 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 `noobprobe` 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 toward the destination to probe. Also stored is a bitmap of hops toward a destination for which ICMP time exceeded messages have been received. This bitmap is used to detect non-responsive hops and to avoid probing them repeatedly. 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.

The probe egress and ingress programs are organized using "program maps": an initial program checks whether the packet is IP, and jumps to separate programs for IPv4 or IPv6 processing. On egress, these programs perform a lookup in the destination IP address BPF map and check whether a probe should be emitted toward a destination to satisfy a configured probe rate. If a probe should be emitted, the integer BPF array index associated with the destination is stored in the `skb` metadata and control is handed off to a third program that handles protocol-specific (ICMP, TCP, UDP) packet operations. Within this third program, the `skb` metadata is consulted to avoid another destination address lookup and the packet is *cloned* and emitted. As a result, the original application packet is sent on its way soon after identifying that a probe should be created. The packet clone is then truncated in size to be of minimum length (*e.g.*, 40 bytes for IPv4 TCP), 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 the first program invoked in the `tc` handler checks whether the `skb` metadata has been set with a destination (integer) identifier; if it has, this packet is ignored to avoid re-cloning a packet that has already been cloned. Program maps are also used on packet ingress in XDP in a similar way, providing some code modularity.

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. Furthermore, the sequence and original destination are matched, a timestamp is recorded, and the data are added to the per-cpu BPF map containing measurement results. For 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.

An alternative design we considered is direct implementation in the kernel, which has been employed in prior work (*e.g.*, [35]). 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.* [11, 30]) using `libpcap` or similar would be portable, but shown in [36] 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].

4 EVALUATION

4.1 Internet Experiments

In this section, we evaluate the performance of `noobprobe` with Internet-wide experiments [3]. Concretely, we describe a week-long experiment in which we monitored flows created using NDT[4], which is in use by M-Lab [10] and has been used in recent studies of the Internet congestion [37, 38]. We focus on particular examples that illustrate how high-fidelity in-band measurements can help in interpreting and contextualizing flow performance.

4.1.1 Data collection. We collected a week-long data set 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 [18] (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.

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 `noobprobe` 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, `noobprobe` was configured with a maximum probe rate of 100 probes/sec 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, as well as with throughput tests using Netflix's fast.com [8]. Moreover, we collected data from experiments in which we ran the NDT client with `noobprobe` 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 [30] and flowtrace [11].

4.1.2 Evaluation of Hop-by-Hop Latencies. We first comment on how queuing dynamics can be revealed through `noobprobe`. To this end, we show in Figure 1 the results from an experiment in which we traced flows generated through a throughput test from the university site using Netflix's `fast.com`. Unlike NDT, `fast.com` uses the standard best practice of multiple TCP flows as part of its throughput test to saturate links between the client and a set of dynamically chosen Netflix edge servers. In this particular test, there were 5 parallel flows created; the figure shows results for one of these flows. We see that there is substantial queuing oscillation at hop 4 (some hops are elided from the plot for readability), suggesting congestion at that point in the network path. Interestingly, hops 3–5 are all within the same provider, indicating that congestion for these flows does *not* take place at the border between providers which has been shown in prior work to be a common site of network congestion [17, 26]. We also observe that at 13.5–14 seconds there are similar effects at hop 2 as in the middle plot of Figure 2 in [36].

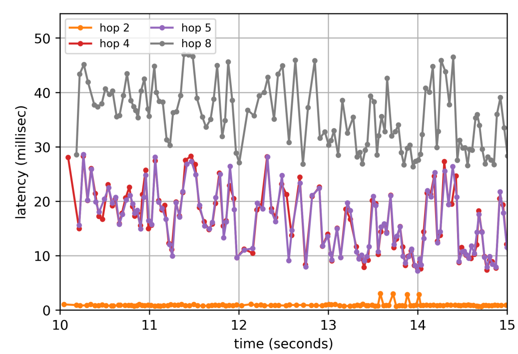

Figure 1: Evolution of hop-by-hop round-trip times measured by using `noobprobe` to monitor a `fast.com` throughput test from the university site.

Lastly, we note again that the RTT measurements collected using in-band probes bear similarity to the Time Series Latency Probe (TLSP) method of [26], 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 application flow behavior due to load-balancing effects and the possibility that ICMP probes take a different path than ordinary application flows.

4.1.3 Observation of Route Changes. Figure 2 depicts hop-by-hop latencies of one flow between the university client and an M-Lab server in Vancouver, Canada, and captures the effect of an inter-domain route change while the flow is in progress. We observe that at about 12 sec., there are major level-shifts in latency for hops 7–9

[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]In all experiments, we used a Linux host running kernel version 5.15 along with bcc commit `c65446b765c9f7df7e357ee9343192de8419234a` (from 28-3-2022).
[4]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.

and responses from hop 6 cease. We also see that very few probes are generated between 12–13 seconds. Since probes are cloned from packets generated from the application, we infer that the application flow went through a major slowdown in its sending rate (likely TCP timeout) at the time of the route change. In all the data we collected, we observed 9 interdomain route changes between the university client and the Vancouver, Canada M-Lab site. Currently, an application experiencing a route change would potentially just observe some change in throughput, latency, or packet loss and be oblivious to *why* the observed changes were occurring. With information from a system like `noobprobe`, however, applications could potentially make adjustments to avoid or appropriately react to an impairment.

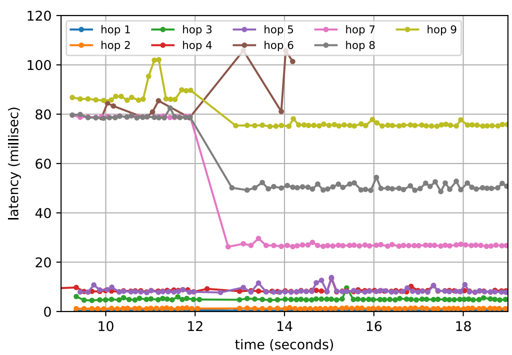

Figure 2: Evolution of hop-by-hop round-trip times measured using `noobprobe` in-band probes in the presence of a mid-flow interdomain route change. Plot shows one flow of an NDT test to an M-Lab server in Vancouver, Canada.

4.1.4 Load Balancing Effects. Finally, because the in-band flow measurements enable 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. Figure 3 shows empirical CDFs of outbound throughput measured by NDT across four load-balanced paths between the Clemson CloudLab client and the Dallas-Fort Worth, TX M-Lab site. The number of test instances that were identified to follow a given route is shown in the plot legend, and we note that the length of each route (in hops) was identical (8).

We see in the figure that download throughputs for two pairs of the routes are similar (1/2 and 3/4), but that performance for routes 1 and 2 was frequently measured to be substantially lower than for routes 3 and 4. We also note that a Kolmogorov-Smirnov 2-sample test indicates that the null hypothesis, that measurements from routes 1/2 and 3/4 are drawn from the same distribution, must be rejected. Examining other load-balanced paths (not shown due to space limits), we found that while there are instances of similar performance for flows following different load-balanced paths, there are also more examples similar to that shown in Figure 3 of substantial performance disparity.

In our data, many of the instances of load balancing we observed were of *flow-level* load balancing: all packets of the same flow follow the same interface-level path. For most clients and one particular destination (Amsterdam, Netherlands) we observed interface-level paths consistent with per-packet load balancing. Interestingly, however, there appeared to be fundamental route changes also taking place due to changes in the number of hops within particular autonomous systems.

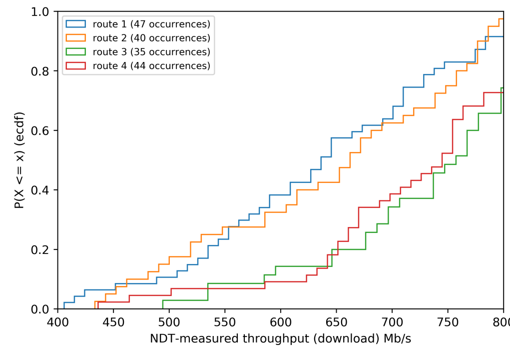

Figure 3: Empirical CDFs of inbound (download) throughput between the Clemson cloudlab site and the Dallas-Fort Worth, TX M-Lab server for four load-balanced paths.

5 PASSIVE MONITORING WITH `NOOBFLOW`

Passively collected network measurements are critical for monitoring local network dynamics and performance from the perspective of individual nodes within the edge-cloud environment. Traditional passive monitoring approaches like Simple Network Management Protocol (SNMP), while useful, are not granular or flexible enough to gain the kind of deep insight needed in a complex multi-layered system. In this section, we describe the design and evaluation of `noobflow`, a passive network measurement agent. `noobflow` is a concrete example of how eBPF-facilitated in-network computing can be exploited for lightweight, flexible, easily-deployed edge-cloud system monitoring.

5.1 `noobflow` Implementation Overview

`noobflow` comprises two components: a user-space program and an XDP-based in-kernel/in-network component. The user-space portion is implemented using the BPF Compiler Collection (bcc) library [7] and acts as a manager of the in-kernel component. In our prototype, this component is written in Python and can be used to dynamically load, configure, and re-configure the kernel component. In a system-wide edge-cloud setting, this component would be configured through a logically centralized controller.

The user-space component creates per-CPU maps that are used to store passively collected network flow records. Using per-CPU maps enables `noobflow` to scale well with the computational resources available on VMs deployed across the edge/cloud. *Which* flows to capture information for can be dynamically reconfigured, and standard flow information is stored, such as source/destination IPs and ports, the number of bytes and packets received, timestamps, etc. A double-buffering scheme is used with *two* per-CPU maps; while one map accumulates newly arriving packets, data from the other can be retrieved and cleared. Periodically, the user-space program atomically swaps the map roles, enabling continuous, lock-free flow collection.

5.2 Evaluation of **noobflow**

5.2.1 Configuration. We design our experiment in the CloudLab infrastructure [18], where two nodes are connected via 25 Gb/s network interfaces [5]. One node acts as the flow collector, running `noobflow`, and the other uses `pktgen`, a high-speed packet generation tool packaged with the Linux kernel. We configured `pktgen` to emit a varying rate of 60 byte UDP packets, starting at 1 Mpps (1 Million packets/sec). We also randomized the destination address and source port to produce 25K new flows/sec. Lastly, we utilized a number of cores on the host running `pktgen` to provide offered loads from 1 Mpps up to 20 Mpps (just over 10 Gb/s, at maximum).

5.2.2 Results. We evaluated the performance of `noobflow` on two criteria: performance and accuracy. In particular, our main metric was the maximum offered packet rate that could be sustained with zero or negligible packet loss (*i.e.*, less than 0.0001% loss) at the flow collector. By simply comparing the number of packets emitted from `pktgen` with the number of packets received by `noobflow`, we could assess whether a given packet rate could be sustained. In our experiments, we varied the number of CPU cores available on the host running `noobflow` as we also vary the offered packet rate from the `pktgen` host.

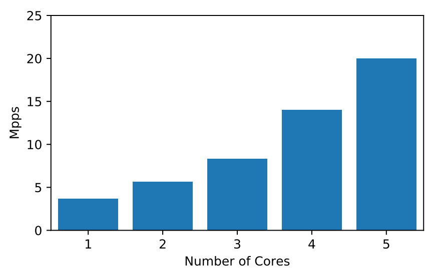

Figure 4: Maximum sustainable packet rate (Mpps) by `noobflow` for an increasing number of CPU cores.

Figure 4 shows results of the number of CPU cores required (x-axis) to sustain a packet rate (y-axis, in Mpps). We observe in the plot that a single core can handle approximately 3.66 Mpps and that this rate scales well with additional cores. The maximum rate we considered in our experiments, 20 Mpps, was achieved with 5 cores (this limit is due to `pktgen` and is not inherent to `noobflow`). Although not shown here, we found that the main performance limitation with `noobflow` has to do with keeping up with a high rate of new flow arrivals, mainly due to limits on the size of BPF maps. We conclude from these experiments that our eBPF-based passive measurement agent is able to scale well with available computational resources at the edge.

6 DISCUSSION AND FUTURE WORK

Limitations. Overall, our experiences and the results of our experiments described in this paper strongly suggest eBPF as a promising high-fidelity vehicle for implementing measurement methods. There are, however, some limitations that we have encountered. In particular, one needs to take care in regards to the movement of data from kernel to userspace. There are facilities for *pushing* data from kernel to userspace (perf buffers [4]) which cause callbacks in the control program (*e.g.*, in Python), but data can also be *pulled* through direct access to eBPF maps from a control program. While the perf interface is simple, the version of bcc we employed used a fixed-size ring buffer which could not scale to the data rate. Direct map access does not suffer from that issue but requires careful data management. Determining which interface is best for a given application can be non-trivial. Another challenge has to do with debugging, which largely relies on `printf`-style tracing. Silent failures can happen (particularly, in our experience, with XDP), which can be difficult to identify, and memory safety checks can sometimes be difficult to understand and fix. We do, however, expect debugging tools to improve as the eBPF subsystem matures. Lastly, there are limitations regarding map sizes which may pose difficulties for flow capture if the rate of new flow creation is too high. While we intend to examine alternative flow storage architectures to address such limitations, we also expect that future eBPF versions may relax current constraints.

With active measurement, in particular, metering out probes in the desired way is the main challenge. Specifically, arbitrary timing of packet emissions is not possible, especially considering eBPF programs installed in `tc` and/or XDP are on the critical path for packet processing thus making spin-waiting an unviable approach. At the same time, *any* packet arrival or departure can be used to trigger a probe within `tc` using the "clone and redirect" API call, which we use in `noobprobe`. The clone and redirect interface must be used with care, though, because it can lead to extended bursts of newly cloned packets. In some cases, bursts may be desired (and indeed, we experimented with bursts), but avoidance is also straightforward, as we discuss in § 3. Still, for active measurement methods that require precise spacing of packets according to some distribution or pattern, traditional approaches or approaches using kernel bypass (*e.g.*, DPDK) are more appropriate at present.

Rethinking Internet measurements with `noobprobe`. We believe that eBPF holds a lot of promise for both active and passive network measurement, and there are several directions we intend to investigate in future work. In particular, we plan to revisit the efforts of Govindan and Paxson to examine delays in ICMP packet generation at routers [19] to better understand how to reduce or eliminate noise in the hop-limited latency measurements. We also plan to collect a much broader set of measurements to better understand wide-area queuing dynamics and congestion. Moreover, we plan to examine alternative data organization for storing flows beyond current eBPF-imposed limits. Finally, we plan to explore the design and evaluation of interfaces to introduce network-awareness gained using `noobprobe` to distributed systems and applications.

Revisiting the NOOB problem. As mentioned in § 2, we need to expose the insights gathered from `noobprobe` and `noobflow` to an inference capability to effectively tackle the NOOB problem. The inference capability should not only have visibility into actual network performance gathered using `noobprobe`/`noobflow` but also performance information at the end. Moreover, as mentioned above the inference capability requires new interfaces to communicate the unified information to applications in (near) real-time to react to diverse network events. We plan to bring this network awareness to applications (*e.g.* Cassandra, ZooKeeper, etc.) and perform an end-to-end evaluation of the interplay between `noobprobe` and `noobflow`, inference capability, and applications as part of future work.

[5]We used Linux hosts running kernel version 5.15 along with bcc commit `c65446b765c9f7df7e357ee9343192de8419234a` (from 28-3-2022).

REFERENCES

[1] A curated list of awesome projects related to eBPF. https://github.com/zoidbergwill/awesome-ebpf. Accessed May 2020.
[2] Apache cassandra. https://cassandra.apache.org/.
[3] Apache zookeeper. https://zookeeper.apache.org/.
[4] bcc Reference Guide. https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md. Accessed May 2020.
[5] BPF and XDP Reference Guide. http://docs.cilium.io/en/latest/bpf/. Accessed May 2020.
[6] BPF Compiler Collection (BCC). https://github.com/iovisor/bcc. Accessed May 2020.
[7] ebpf - extended berkeley packet filter. https://www.iovisor.org/technology/ebpf.
[8] FAST. https://fast.com. Accessed May 2020.
[9] Logdevice. https://logdevice.io.
[10] Ndt (network diagnostic tool). https://www.measurementlab.net/tests/ndt/.
[11] Adnan Ahmed, Ricky Mok, and Zubair 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.
[12] Samer Al-Kiswany, Suli Yang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Nice: Network-integrated cluster-efficient storage. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 29–40, 2017.
[13] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clémence Magnien, and Renata Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 153–158. ACM, 2006.
[14] Daniel Borkmann. On getting tc classifier fully programmable with cls bpf. 2016.
[15] Mark Crovella and Balachander Krishnamurthy. *Internet measurement: infrastructure, traffic and applications*. John Wiley & Sons, Inc., 2006.
[16] Xu Cui, Michael Mior, Bernard Wong, Khuzaima Daudjee, and Sajjad Rizvi. Netstore: Leveraging network optimizations to improve distributed transaction processing performance. In *Proceedings of the Second International Workshop on Active Middleware on Modern Hardware*, pages 1–10, 2017.
[17] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 1–15, 2018.
[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.
[19] Ramesh Govindan and Vern Paxson. Estimating router ICMP generation delays. In *Passive & Active Measurement (PAM)*, 2002.
[20] Brendan Gregg. Linux Extended BPF (eBPF) Tracing Tools. http://www.brendangregg.com/ebpf.html. Accessed May 2020.
[21] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David 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.
[22] Van Jacobson. Pathchar: A tool to infer characteristics of Internet paths, 1997.
[23] Sangeetha Abdu Jyothi, Sayed Hadi Hashemi, Roy Campbell, and Brighten Godfrey. Towards an application objective-aware network interface. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.
[24] Baber Khalid, Nolan Rudolph, Ramakrishnan Durairajan, and Sudarsun Kannan. Micromon: A monitoring framework for tackling distributed heterogeneity. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.
[25] Bingdong Li, Jeff Springer, George Bebis, and Mehmet Hadi Gunes. A survey of network flow applications. *Journal of Network and Computer Applications*, 36(2):567–581, 2013.
[26] Matthew Luckie, Amogh Dhamdhere, David Clark, Bradley Huffaker, et al. Challenges in inferring internet interdomain congestion. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 15–22. ACM, 2014.
[27] Matthew Luckie, Young Hyun, and Bradley 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.
[28] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter*, 1993.
[29] Chris Misa, Ramakrishnan Durairajan, Reza Rejaie, and Walter Willinger. Revisiting network telemetry in coin: A case for runtime programmability. *IEEE Network*, 35(5):14–20, 2021.
[30] Ivan Morandi, Francesco Bronzino, Renata Teixeira, and Srikanth Sundaresan. Service Traceroute: Tracing Paths of Application Flows. In *International Conference on Passive and Active Network Measurement*, pages 116–128. Springer, 2019.
[31] Vern Paxson, Jamshid Mahdavi, Andrew Adams, and Matt Mathis. An architecture for large scale internet measurement. *IEEE Communications Magazine*, 36(8):48–54, 1998.
[32] Cristel Pelsser, Luca Cittadini, Stefano Vissicchio, and Randy 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.
[33] Diana Popescu, Noa Zilberman, and Andrew Moore. Characterizing the impact of network latency on cloud-based applications' performance. 2017.
[34] Rob Sherwood and Neil Spring. Touring the Internet in a TCP sidecar. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 339–344. ACM, 2006.
[35] J. Sommers and P. Barford. An active measurement system for shared environments. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, October 2007.
[36] Joel Sommers and Ramakrishnan Durairajan. Elf: High-performance in-band network measurement. In *IFIP Network Traffic Measurement and Analysis Conference*, 2021.
[37] Srikanth Sundaresan, Mark Allman, Amogh Dhamdhere, and Kc Claffy. TCP congestion signatures. In *Proceedings of the 2017 Internet Measurement Conference*, pages 64–77. ACM, 2017.
[38] Srikanth Sundaresan, Xiaohong Deng, Yun Feng, Danny Lee, and Amogh Dhamdhere. Challenges in inferring internet congestion using throughput measurements. In *Proceedings of the 2017 Internet Measurement Conference*, pages 43–56. ACM, 2017.
[39] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. {FLAIR}: Accelerating reads with consistency-aware network routing. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 723–737, 2020.
[40] Alex FR Trajano and Marcial P Fernandez. Two-phase load balancing of in-memory key-value storages using network functions virtualization (nfv). *Journal of Network and Computer Applications*, 69:1–13, 2016.
[41] Bahador Yeganeh, Ramakrishnan Durairajan, Reza Rejaie, and Walter Willinger. A first comparative characterization of multi-cloud connectivity in today's internet. In *Passive and Active Measurement: 21st International Conference, PAM 2020, Eugene, Oregon, USA, March 30–31, 2020, Proceedings 21*, pages 193–210. Springer, 2020.
[42] Bahador Yeganeh, Ramakrishnan Durairajan, Reza Rejaie, and Walter Willinger. A case for performance- and cost-aware multi-cloud overlays. In *In Proceedings of IEEE International Conference on Cloud Computing, Illinois, USA*, 2023.