Passive Monitoring of Network Latency at High Line Rates

Simon Sundberg
Karlstad University, Sweden
simon.sundberg@kau.se

Anna Brunstrom
Karlstad University, Sweden
anna.brunstrom@kau.se

Simone Ferlin-Reiter
Ericsson, Sweden
simone.ferlin-reiter@ericsson.com

Toke Høiland-Jørgensen
Red Hat, Denmark
toke@redhat.com

Jesper Dangaard Brouer
Red Hat, Denmark
jbrouer@redhat.com

***Abstract*—Network latency plays a crucial role for many applications and their perceived quality of experience. With an increasing focus on high network speeds and real time, interactive applications relying on reliable and low latency, the ability to effectively monitor latency is becoming more important than ever. While many available tools rely on active monitoring, this approach relies on traffic injection in the network, which can be a source of latency in itself and have a negative overall network performance impact. This paper presents evolved Passive Ping (ePPing), a tool that leverages eBPF to passively monitor latency of existing network traffic. Preliminary evaluation shows that ePPing delivers RTT reports more reliably and at a lower overhead than other state-of-the-art tools, such as PPing.**

***Index Terms*—monitoring, network latency, eBPF, XDP**

I. Introduction

Network latency is an important factor of network performance, even if focus for a long time has been on throughput instead [1]. Being able to monitor network latency, typically in the form of Round Trip Times (RTTs), has a wide variety of use cases: Service providers may wish to monitor latency to ensure a good Quality of Experience (QoE), network providers and Internet Service Providers (ISPs) may want to ensure that they fulfill Service Level Agreements (SLAs) with latency requirements, and network operators may monitor network latency to find upcoming issues in their networks. Yet effectively monitoring network latency remains challenging, especially as the networks evolve to support ever higher packet rates. There are two fundamentally different approaches to monitor network latency: active or passive monitoring. Active monitoring injects traffic by sending out its own network probes, whereas passive monitoring instead inspects the already existing network traffic from other applications.

There exists a plethora of active monitoring tools such as ping [2], hping [3], nping [4], IRTT [5], Netlatency [6] and many others. While such tools have their merit measuring network latency in a very controlled manner, they have a number of shortcomings, mainly: 1) Active monitoring adds overhead in the form of additional traffic in the network. While an occasional probe may not be problematic, getting wide coverage and fine grained latency measurements require a non-negligible amount of probes, which may in turn impact other traffic in the network. 2) Measuring latency of a wide range of hosts requires sending probes covering possibly all paths between them. Having each host probe every other host does not scale well in large networks. Instead, careful consideration must be taken of which hosts should probe each other, see Pingmesh [7]. It also requires that the network operator is able to run agents on all hosts of interest, which may not be feasible e.g. for an ISP to monitor latency of its customers. 3) The latency measured by the network probes may not reflect that of the real application traffic. Due to for example *bufferbloat*, active queue management (AQM), load balancing and various types of middleboxes treating traffic differently, these probes may see different latency measurements.

By inferring the latency from existing application traffic instead of sending their own probes, passive monitoring techniques avoid these issues. As passive approaches only need to be able to observe network traffic, they do not necessarily need to run at the sender or receiver and can more easily run on any machine which sees the traffic of interest, ideally in both directions. Several tools for passively monitoring TCP latency using packet captures exist [8]–[10]. However, packet capturing techniques impose a lot of overhead and struggle to keep up with the growing rates encountered in modern networks. To extend network monitoring to high rates, several recent works [11]–[13] propose P4 [14] solutions. While P4 programs can achieve high performance, they typically require either special hardware support, like Tofino switches, or the use of the Data Plane Development Kit (DPDK) [15]. There are, however, many Linux devices, such as servers, routers and Network Intrusion Detection Systems (NIDS), that do not fulfill these requirements, but still could benefit from monitoring network latency.

In this paper, we therefore instead propose using eBPF [16] to enable efficient network latency monitoring on Linux devices using the standard kernel network stack. eBPF is a technology that allows attaching small programs to numerous hooks in the Linux kernel. In particular, the tc [17] and XDP [18] hooks allow for running eBPF programs that can inspect and modify each packet early in kernel's network stack, and thus enable a programmable data plane in Linux. We

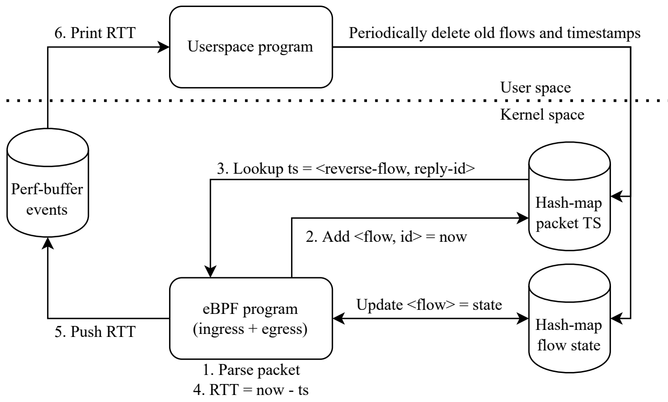


Fig. 1: Overview of ePPing design. The eBPF program runs in kernel space on every received and sent packet of a specified interface, and contains all the logic for parsing packets, matching replies with previously sent packets and calculate the RTTs. The userspace component only loads the eBPF program at startup and then prints out the RTTs the eBPF program pushes to it.

demonstrate the feasibility of this approach by implementing an evolved Passive Ping (ePPing), inspired by the packet capture based PPing [10]. Our evaluation shows that ePPing has much lower overhead than PPing and can handle over 10 Gbps on a single core.

II. ePPing design and implementation

The principle behind ePPing and most other passive latency monitoring tools, is to match replies to previously observed packets and to calculate the RTT as the time difference between these. How ePPing performs this task is illustrated in Figure 1. ① First each incoming or outgoing packet is parsed for some identifier that can be used to match the packet against a future reply. ② If such an identifier is found, the current time is saved in a hash map using a combination of the flow tuple and the identifier as a key to uniquely identify the packet. ③ Then the program checks if the packet is a reply to a previously timestamped packet by querying the hash map for an entry in the reverse flow direction with a matching identifier. ④ If such an entry is found, the RTT is calculated by subtracting the stored timestamp from the current time. ⑤ - ⑥ Finally the RTT can be reported to the user. Additionally, ePPing also keeps track of some state for each flow, for example number of packets sent and minimum RTT observed.

Both ePPing and PPing use the TCP timestamp option [19] as identifiers. Then, each TCP packet will contain two timestamps, TSval and TSecr. The TSval field will contain a timestamp from the sender, and the receiver will then echo that timestamp back in the TSecr field. One can thus use the TSval value as an identifier for an observed packet and later match it against the TSecr value in a reply. It should be noted that TCP timestamps are updated at a limited frequency, typically once every millisecond, and thus multiple consecutive packets may share the same TSval. To avoid mismatching replies to packets with duplicate TSval values and getting underestimated RTTs, we therefore only timestamp the first packet with each TSval in a flow and match it against the first TSecr echoing it. TCP timestamps are also optional and may not be available in all TCP traffic. Some tools, like Wireshark [8] and the P4 solutions [11]–[13] match sequence and ACK numbers instead. While it is possible to extend ePPing to operate on sequence and ACK numbers, we opted for TCP timestamps for simplicity as it avoids the TCP retransmission ambiguity, which may cause incorrect RTT samples for sequence and ACK matching if not handled.

The fundamental mechanism for matching replies of previously timestamped packets to calculate RTTs is, however, not limited to TCP, and ePPing can be extended to work with additional protocols. As a way to demonstrate this, we have implemented support for ICMP echo request sequence numbers as identifiers, which means that ePPing can also passively monitor latency for common ping utilities. Other possible extensions include the DNS transaction ID or the QUIC spin bit.

While the logic for timestamping packets, matching replies and calculating RTTs is very similar between PPing and ePPing, the main difference between them is where this logic runs, i.e., how it is implemented. PPing is a user space application and relies on traditional packet capturing, i.e., copying the packets from kernel space to user space, which incurs a lot of overhead at high packet rates. Once copied to user space, PPing can parse the packet headers to retrieve the necessary packet identifiers. In contrast, ePPing implements most of its logic in an eBPF program running in kernel space, as shown by Figure 1. By attaching its eBPF program to the tc and XDP hooks, ePPing can parse the packet headers directly from the kernel buffers, without any copying. All the logic for parsing and timestamping packets, matching replies and calculating RTTs is implemented in the eBPF program. The user space component is only responsible for loading and attaching the eBPF program, printing out RTTs pushed by the eBPF program, and periodically trigger the cleanup of stale entries in the hash maps.

By moving most of the logic to kernel space and thereby avoiding the costly copying of packets from kernel to user space, ePPing is able to operate with lower overhead and outperfom PPing at high rates. ePPing is still under development, but it is open source and can be found at https://github.com/xdp-project/bpf-examples/tree/master/pping

III. Results

The motivation behind implementing ePPing in eBPF was to reduce overhead and allow it to work at higher rates. Therefore, we measure what impact ePPing has on a system under high load. A testbed was set up consisting of two hosts (Intel i7 7700, 16 GB RAM) connected via 100 Gbps links to a middlebox (Intel Xeon E5-1650, 32 GB RAM) forwarding the traffic between the hosts. To ensure that the middlebox was the bottleneck, we configured it to only use a single CPU core for all packet processing and any latency monitoring

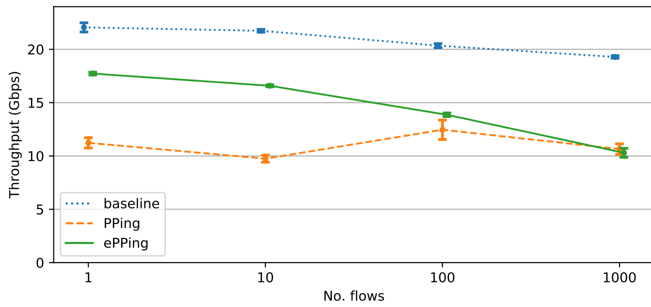
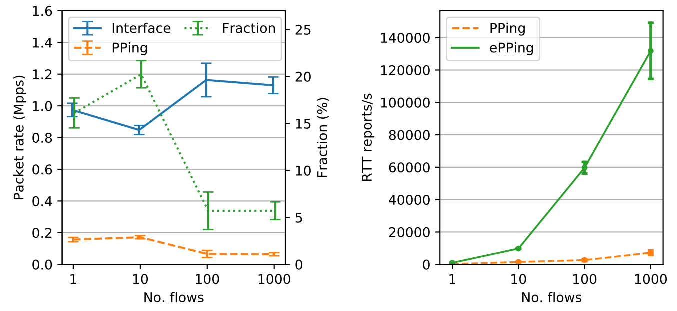


Fig. 2: Throughput achieved while just forwarding (baseline), running PPing or running ePPing. The error bars show standard deviation. Note that PPing does not run on every packet at these rates, while ePPing does.



(a) Packets seen by PPing compared to packets traversing the interface.

(b) RTTs reported per second by PPing and ePPing

Fig. 3: PPing missing packets and its impact on RTT reports.

tool running on it. Additionally, various network offloads such as Generic Receive Offload (GRO), Generic Segmentation Offload (GSO) and TCP Segmentation Offload (TSO) were disabled on the middlebox, but left enabled on the hosts. One of the hosts generated a varying number (1, 10, 100 or 1000) TCP flows using iperf3, and either PPing or ePPing was used to passively monitor the (partial) RTT of the traffic from the interface on the middlebox facing the receiver. We measured the total throughput achieved while running PPing or ePPing and compared to a baseline, i.e., not using any latency monitoring. The results have been summarized in Figure 2.

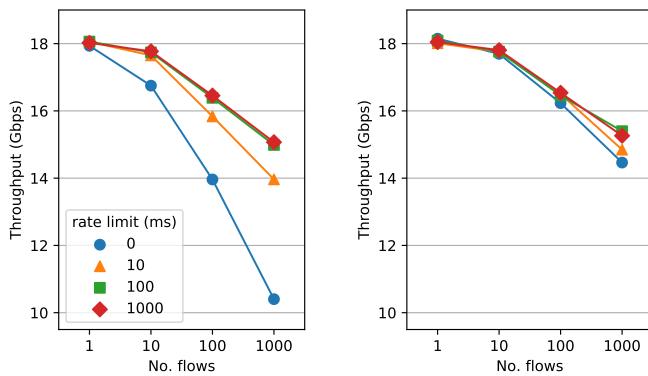
From Figure 2 it is clear that both PPing and ePPing have considerable overhead, and may reduce throughput substantially if they are deployed on a system under heavy load. At the same time, it is worth noting that ePPing is able to reach 10+ Gbps on a single core with offloading mechanisms turned off, which may be sufficient for many use cases. PPing can seemingly also handle 10+ Gbps, but it only does so by missing a lot of packets. While ePPing is generally able to sustain a much higher throughput than PPing, the throughput decreases with an increasing number of flows, where both tools show similar performance at 1000 flows. The overhead increase for ePPing with more flows is expected, as the limited update frequency of TCP timestamps means more flows will result in an increase of potential RTT samples that ePPing has to process and report. Additionally, even the baseline, which just forwards packets and should be largely oblivious to the number of flows, also decreases throughput as the numbers of flows increase. This is due to an increasing number of sent ACKs that need to be forwarded. In other words, even though throughput decreases with more flows, the number of packets the middlebox has to process remains fairly constant for the baseline.

What is surprising in Figure 2 is the performance of PPing as the number of flows increases. One could expect that the workload for PPing would increase in a similar manner as for ePPing, or at least show a similar slope or decrease in throughput as the baseline. Instead the throughput with PPing has no clear trend, and even increases between 10 and 100 flows. The reason for this behavior is likely that the packet capturing library (libtins) used by PPing is unable to keep up with the high rates (10+ Gbps). Thus, PPing actually sees only a fraction of the packets. This is confirmed by Figure 3a, which shows the number of packets PPing handles related to the total number of packets actually traversing the interface. Between 10 and 100 flows the fraction of packets PPing sees drops from $20\,\%$ to just over $5\,\%$, which corresponds well with the increase in overall throughput seen in Figure 2. By missing many packets, PPing also misses many RTT samples, which is clearly seen in the number of RTTs reported by ePPing and PPing shown in Figure 3b. Additionally, when missing packets, PPing cannot be sure that it is matching the first unique TSval with the first TSecr, which may also impact the accuracy of its RTT estimates.

As shown in Figure 3b, ePPing ends up reporting a very large amount of RTTs as the number of flows increase (around 130000 per second at 1000 flows). A considerable part of the overhead from ePPing consists of printing out the reports at this point. In many cases, such a large amount of RTT samples may not be required. To reduce the number of reported RTTs, and the overhead that comes with them, we therefore implemented a simple rate limit. The rate limit will only allow a single RTT sample every time interval t per flow. This is a more deterministic way to reduce the number of RTTs reported than to semi-randomly report a subset of RTTs due to missing packets as PPing does.

The obtained throughput when setting the rate limit t to $0\,\mathrm{ms}$, $10\,\mathrm{ms}$, $100\,\mathrm{ms}$ and $1000\,\mathrm{ms}$ (corresponding to 1000, 100, 10 and 1 potential RTT sample per second and flow) is shown in Figure 4a. By comparing Figure 4a to Figure 2, it is clear that limiting the amount of RTT samples drastically reduces the overhead of ePPing when dealing with many flows. While performance still clearly decreases with the number of flows even with $t = 1000\,\mathrm{ms}$, the drop in throughput is now similar to the one that can be observed for the baseline in Figure 2.

To evaluate the overhead of passing the RTTs to user space and printing them out, the experiment was also repeated with a modified ePPing that does not push the calculated RTTs to



(a) When reporting RTTs. (b) When not reporting RTTs.

Fig. 4: Throughput achieved when rate limiting RTT samples to various degrees.

user space. Figure 4b shows the throughput achieved with this modified ePPing. It seems that the majority of the performance improvement gained by rate limiting RTT samples comes from simply reporting fewer RTTs. This means that ePPing is capable of calculating RTTs at a relatively high rate in the kernel, but reporting all of them is costly. A feasible solution between granularity of RTT reports and performance overhead may therefore be to calculate a large number of RTT samples in the kernel, but only send aggregated reports to user space.

IV. Conclusion

In this paper we have presented evolved Passive Ping (ePPing), a tool using eBPF to passively monitor network latency at high rates. By using eBPF to inspect the packets directly in kernel space ePPing is able to avoid the large overhead associated with copying packets to user space that PPing suffers from. Our evaluation shows that ePPing has lower overhead than PPing. We also demonstrate that packet capturing techniques like PPing are unable to keep up with the high packet rates encountered in 10+ Gbps networks, missing many of the packets and as a consequence delivering fewer and less reliable RTT samples, issues that ePPing does not suffer from.

However, the evaluation also indicates that reporting the large amount of RTTs calculated in the presence of many flows incurs a lot of overhead for ePPing. A simple RTT rate limiting mechanic is able to substantially reduce this overhead at the cost of lower RTT granularity, but there is room for developing better ways to cope with the high rate of RTT samples in future work. Additionally, we also wish to extend ePPing to additional protocols, such as DNS and QUIC, in the future, making it a convenient general tool to monitor network latency for a wide range of traffic types.

V. Acknowledgements

This work is funded by the Swedish Knowledge Foundation (KKS) through the project AIDA: A Holistic AI-driven Networking and Processing Framework for Industrial IoT.

References

[1] S. Cheshire. “It’s the Latency, Stupid.” (2001), [Online]. Available: http://www.stuartcheshire.org/rants/Latency.html.

[2] *Iputils*, 2022. [Online]. Available: https://github.com/iputils/iputils.

[3] S. Sanfilippo, *Hping*, 2006. [Online]. Available: http://www.hping.org/.

[4] L. MartinGarcia and Fyodor, *Nping*, n.d. [Online]. Available: https://nmap.org/nping/.

[5] P. Heist, *IRTT (Isochronous Round-Trip Tester)*, 2021. [Online]. Available: https://github.com/heistp/irtt.

[6] H. Thiery, *Netlatency*, 2021. [Online]. Available: https://github.com/kontron/netlatency.

[7] C. Guo, L. Yuan, D. Xiang, *et al.*, “Pingmesh: A large-scale system for data center network latency measurement and analysis,” in *SIGCOMM ’15*, ACM, 2015, pp. 139–152.

[8] G. Combs, *Wireshark*, 2022. [Online]. Available: https://www.wireshark.org/.

[9] S. D. Strowes, “Passively measuring TCP round-trip times,” *Commun. ACM*, vol. 56, no. 10, pp. 57–64, 2013.

[10] K. Nichols, *Pping: Passive ping network monitoring utility*, 2018. [Online]. Available: https://github.com/pollere/pping.

[11] M. Ghasemi, T. Benson, and J. Rexford, “Dapper: Data plane performance diagnosis of TCP,” in *SOSR ’17*, ACM, 2017, pp. 61–74.

[12] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford, “Measuring TCP round-trip time in the data plane,” in *SPIN ’20*, ACM, 2020, pp. 35–41.

[13] Y. Zheng, X. Chen, M. Braverman, and J. Rexford, “Unbiased delay measurement in the data plane,” in *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2022, pp. 15–30.

[14] P. Bosshart, D. Daly, G. Gibb, *et al.*, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[15] DPDK Project. “Home - DPDK.” (n.d.), [Online]. Available: https://www.dpdk.org/.

[16] The Linux Foundation. “eBPF.io.” (2021), [Online]. Available: https://ebpf.io/.

[17] Q. Monnet. “Understanding tc direct action mode for BPF.” (2020), [Online]. Available: https://qmonnet.github.io/whirl-offload/2020/04/11/tc-bpf-direct-action/.

[18] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, *et al.*, “The EXpress data path: Fast programmable packet processing in the operating system kernel,” in *CoNEXT ’18*, ACM, 2018, pp. 54–66.

[19] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger, “TCP Extensions for High Performance,” IETF, RFC 7323, Sep. 2014. [Online]. Available: https://datatracker.ietf.org/doc/rfc7323.