Efficient Continuous Latency Monitoring with eBPF

Simon Sundberg[1](✉), Anna Brunstrom[1], Simone Ferlin-Reiter[1,2], Toke Høiland-Jørgensen[3], and Jesper Dangaard Brouer[3]

[1] Karlstad University, Karlstad, Sweden
{simon.sundberg,anna.brunstrom}@kau.se
[2] Red Hat, Stockholm, Sweden
sferlinr@redhat.com
[3] Red Hat, Copenhagen, Denmark
{toke,brouer}@redhat.com

Abstract. Network latency is a critical factor for the perceived quality of experience for many applications. With an increasing focus on interactive and real-time applications, which require reliable and low latency, the ability to continuously and efficiently monitor latency is becoming more important than ever. Always-on passive monitoring of latency can provide continuous latency metrics without injecting any traffic into the network. However, software-based monitoring tools often struggle to keep up with traffic as packet rates increase, especially on contemporary multi-Gbps interfaces. We investigate the feasibility of using eBPF to enable efficient passive network latency monitoring by implementing an evolved Passive Ping (ePPing). Our evaluation shows that ePPing delivers accurate RTT measurements and can handle over 1 Mpps, or correspondingly over 10 Gbps, on a single core, greatly improving on state-of-the-art software based solutions, such as PPing.

Keywords: Passive monitoring · Network latency · eBPF

1 Introduction

That network latency is an important factor of network performance has long been known [8]. Various studies have shown that users' Quality of Experience (QoE) for many different applications, such as web searches [3], live video [32] and video games [31], is strongly related to end-to-end latency, where network latency can be a major component. For highly interactive applications envisioned for the Tactile Internet or Augmented and Virtual Reality (AR/VR), reliable low latency will be even more crucial [24]. It is therefore of great interest to Internet Service Providers (ISPs) to be able to monitor their customers' network latency at large. Furthermore, network latency monitoring has a wide range of other use cases like: verifying Service Level Agreements (SLAs), finding and troubleshooting network issues such as bufferbloat [28], making routing decisions [34], IP geolocation [12] and detecting IP spoofing [18] and BGP routing attacks [4].

There exists many tools for actively measuring network latency by sending out network probes, such as `ping` [16], `IRTT` [11], and RIPE Atlas [22]. While active monitoring is useful for measuring connectivity and idle network latency in a controlled manner, it is unable to directly infer the latency application traffic experience. The network probes may be treated differently from application traffic by the network, due to for example active queue management and load balancing, and therefore their latency may also differ. Furthermore, many active monitoring tools require agents to be deployed directly on the monitored target, which is not feasible for an ISP wishing to monitor the latency of its customers.

Passive monitoring techniques avoid these issues by observing existing application traffic instead of probing the network. Additionally, passive monitoring can often run on any device on the path that sees the traffic, not limited to end hosts. Several tools for passively inferring TCP round trip times (RTTs) already exist: `Tcptrace` [25] can compute TCP RTTs from packet traces, but is unable to operate on live traffic. `Wireshark` and the related `tshark` [9] can operate on live traffic, but are unsuitable for continuous monitoring over longer periods of time, due to keeping a record of all packets in memory. On the other hand, `PPing` [21] uses a streaming algorithm, which allows for continuous monitoring of live traffic. However, like most other software based passive network monitoring solutions, `PPing` relies on traditional packet capturing techniques such as `libpcap`. Packet capturing imposes a high overhead and is unable to keep up with the high packet rates encountered on modern network links [17].

To enable passive network monitoring at higher packet rates, several recent works [7,10,23,35] propose solutions based on P4 [6]. While these P4-based solutions can achieve high performance, they require hardware support for P4, commonly found in Tofino switches. It could be possible to modify such P4 programs to compile with Data Plane Development Kit (DPDK), however, this would compromise on the guaranteed performance provided by the hardware. Beyond DPDK and P4, there are many more Linux devices relying on kernel network stacks that could still benefit from monitoring network latency. Examples include commodity web servers, routers, traffic shapers and Network Intrusion Detection Systems (NIDS), which use the Linux network stack for their normal operation.

In recent years, the introduction of eBPF [29] in the Linux kernel added the ability to attach small programs to various hooks that run in the kernel. This makes it possible to inspect and modify kernel behavior in a safe and performant manner, without having to recompile a custom kernel. eBPF is in general well suited for monitoring processes in the kernel, and the BPF Compiler Collection (BCC) repository already contains two tools to passively monitor TCP RTT: `tcpconnlat` and `tcprtt`. While these tools expose RTT metrics in an efficient manner, they rely on the RTT estimations from the kernel's own TCP stack, and can therefore only run on end hosts.

While retrieving statistics from the kernel certainly has its uses, Linux Traffic Controller (tc) BPF and eXpress Data Path (XDP) [13] hooks go a step further and essentially enable a programmable data plane in the Linux kernel [30]. eBPF programs attached to tc and XDP hooks can process and take actions on each packet early in the Linux network stack, without the overhead from cloning the packet and exposing it to a user space process like packet capturing does. XDP and tc-BPF have been used to implement for example efficient flow monitoring [1], load balancers [19] and a Kubernetes Container Network Interface (CNI) [14]. Of particular relevance for this work, [33] proposes an in-band network telemetry approach for measuring one-way latency. It uses eBPF to add timestamps to a fraction of the packet headers. However, this approach requires full control over the part of the network that should be monitored as well as synchronized clocks between source and sink nodes.

In this paper we instead propose using eBPF to efficiently inspect packets and use a streaming algorithm, such as the one used by `PPing`, to calculate the RTT for the packets as they traverse the kernel. Such a solution can continuously monitor network latency from any Linux-based device that is able to see the traffic in both directions of a flow. Also, our proposal does not require the control of any other device in the network or end hosts. Furthermore, it avoids the overhead of packet capturing, and it does not require any modifications to the Linux kernel or special hardware support. To show the feasibility of this approach, we make the following contributions:

- We implement an evolved Passive Ping (`ePPing`), inspired by `PPing`, but using eBPF instead of traditional packet capturing.
- We evaluate the accuracy and overhead of `ePPing`, demonstrating that it provides accurate RTTs and can operate at high packet rates with considerably lower overhead than `PPing`, being able to process upwards of 16x as many packets at a third of the CPU overhead.
- We identify that reporting a large number of RTT values makes up a significant part of the overhead of `ePPing`, and implement simple in-kernel sampling and aggregation to mitigate it.

The design and implementation of `ePPing` is covered in Sect. 2, while the accuracy and performance of `ePPing` is evaluated in Sect. 3. Finally, we summarize our conclusions in Sect. 4.

Ethical Considerations This work does not raise any ethical issues as all experiments have been performed in a controlled testbed with no real user traffic. However, the presented `ePPing` tool reports IP addresses and ports, which in other contexts may contain sensitive information. Like any tool that can collect and report IP addresses, great care should therefore be taken to ensure that such information does not leak to unauthorized parties before deploying `ePPing` in a public network.

2 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 Fig. 1. First, each incoming or outgoing packet is parsed for a packet-identifier that can be used to match the packet against a future reply ①. 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 ②. Then the program checks if the packet contains a suitable reply identifier, which it can use to match with a previously seen packet in the reverse direction, and queries the hash map ③. If a match is found, the RTT is calculated by subtracting the stored timestamp from the current time ④. Finally, the RTT report is pushed to user space ⑤, which prints it out ⑥. Additionally, ePPing also keeps track of some state for each flow, e.g., number of packets sent and minimum RTT observed.

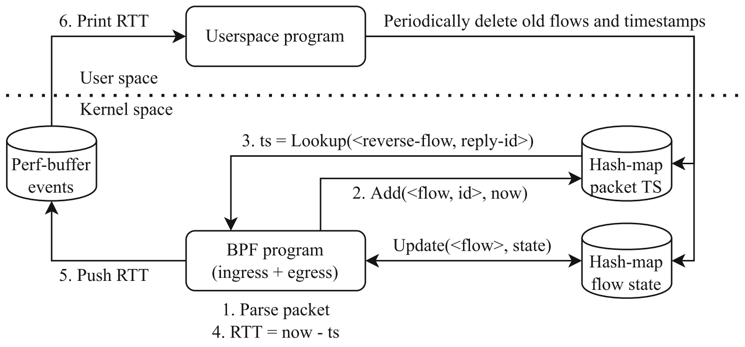


Fig. 1. Overview of ePPing design.

Both ePPing and PPing use the TCP timestamp option [5] as identifiers. With TCP timestamps, each TCP header 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. Thus, multiple consecutive packets may share the same TSval, which is therefore, especially at high rates, not a reliable unique identifier. To avoid mismatching replies to packets and getting underestimated RTTs, we only timestamp the first packet for each unique TSval in a flow and match it against the first TSecr echoing it. By only using the edge when TCP timestamps shift, the frequency rather than the accuracy of the RTT samples is limited to the update rate of TCP timestamps. Note that matching the first instance of a TSval against the first matching TSecr, combined with the algorithm for how the receiver sets the TSecr, means that the calculated RTT will always include a delay component of delayed ACKs [5]. We further discuss the implications of using TCP timestamps as identifiers to passively monitor the RTT in Appendix A.

Although primarily designed for TCP traffic, the fundamental mechanism `ePPing` is based on, to match replies of previously timestamped packets, is not limited to TCP. As a way to demonstrate this possibility, we have also implemented support for ICMP echo request sequence numbers as identifiers. This means that `ePPing` can also passively monitor latency for common `ping` utilities. Other possible extensions for future work include the DNS transaction ID [20] or the QUIC spin bit [15].

While the underlying logic for passively calculating RTTs is very similar between `ePPing` and `PPing`, 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 packets from kernel to user space. Once copied to user space, `PPing` can parse the packet headers to retrieve the necessary packet identifiers, e.g., the TCP timestamps. In contrast, `ePPing` implements most of its logic in eBPF programs running in kernel space, as shown by Fig. 1. By attaching its eBPF programs to the tc-BPF and XDP hooks, `ePPing` can parse packet headers directly from the kernel buffers, without any copying. The logic for parsing and timestamping packets, matching replies and calculating RTTs is implemented in the eBPF programs. The user space component is only responsible for loading and attaching the eBPF programs, printing out RTTs pushed by the eBPF programs, and periodically flushing stale entries in the hash maps.

Therefore, by moving most of the logic to kernel space and thereby avoiding the costly packet capturing and related copying of packets, `ePPing` is able to operate with lower overhead, significantly outperforming `PPing` at high packet rates. `ePPing` is available as open source [27], and the exact build used in this work together with the experiment scripts and measurement data is archived at [26].

3 Results

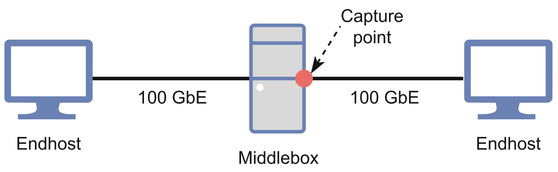


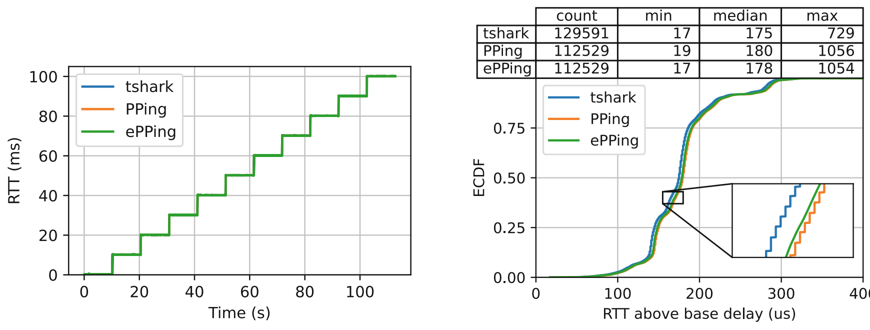
Fig. 2. Testbed setup.

To evaluate `ePPing`, we run a number of experiments to evaluate the accuracy of the reported RTT values as well as the runtime overhead. All experiments are performed on a testbed setup as depicted in Fig. 2. The testbed consists of two end hosts (Intel i7 7700, 16 GB RAM, kernel 5.16) connected via 100 Gbps links to a middlebox (Intel Xeon E5-1650, 32 GB RAM, kernel 5.19), which forwards traffic between the end hosts. In all experiments, the (partial) RTT between the middlebox and receiver end host is passively monitored from the interface on the middlebox facing the receiver, unless otherwise specified.

The network offloads Generic Receive Offload (GRO), Generic Segmentation Offload (GSO) and TCP Segmentation Offload (TSO) are disabled on the middlebox, but left enabled on the end hosts. With this, we force the middlebox to process every packet. This is not necessary for `PPing` or `ePPing`, however, it provides a more accurate view of how packets traverse the wire. Furthermore, disabling the offloads makes it easier to fairly compare performance across a varying amount of concurrent flows, as the offloads tend to become less effective as the rate per flow decreases. With the offloads left enabled, the middlebox would have inherently performed much better for a few flows with very high packet rates compared to if the same packet rate is distributed across many flows, even without passive monitoring.

Section 3.1 focuses on the accuracy of the RTTs reported by `ePPing` by comparing them to the RTTs reported by `PPing`, which also relies on TCP timestamps, and `tshark`, which instead calculates the RTTs from the sequence and acknowledgement numbers. Section 3.2 covers the overhead `ePPing` incurs on the system compared to `PPing`, thereby evaluating if implementing a similar algorithm in eBPF programs instead of relying on packet capturing is a feasible way to extend passive latency monitoring to higher packet rates.

3.1 RTT Accuracy



	count	min	median	max
tshark	129591	17	175	729
PPing	112529	19	180	1056
ePPing	112529	17	178	1054

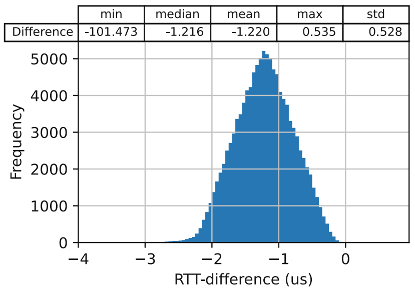
(a) RTTs reported over the duration of the test.

(b) The distribution of RTTs after subtracting configured delay.

Fig. 3. RTT values reported by `tshark`, `PPing` and `ePPing` for a single TCP flow with 0 to 100 ms of additional latency added in 10 ms steps.

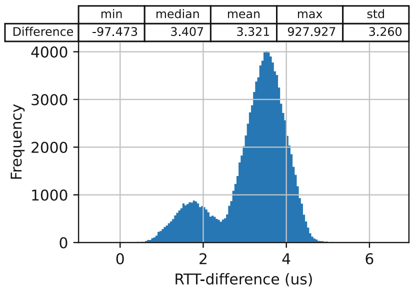
To evaluate the accuracy of the RTT values `ePPing` reports, we use `iperf3` to send data at a paced rate of 100 Mbps over a single flow from the sender to the receiver end host. To test that `ePPing` is able to accurately track changes in RTT, we apply a fixed `netem` delay, which is increased in 10 ms steps every 10 s, going from 0 to 100 ms, see Fig. 3a. In addition to running `ePPing` at the capture point, we capture the headers of all packets by running `tcpdump` on the same interface. `PPing`, `tshark` and `tcptrace` calculate the TCP RTT values from the capture file, but `tcptrace` is omitted from the results as it yields identical RTT values as `tshark`. To avoid small latency variations from the CPU aggressively entering different sleep states, we use the `tuned-adm` profile *latency-performance* on the middlebox during these tests.

Figure 3a shows a timeseries of the RTT values calculated by each tool. All tools provide RTT values closely following the configured `netem` delay. Figure 3b instead shows the distribution of how much higher the reported RTT values are compared to the configured netem delay, to avoid the scale of RTT values to dwarf the variation. However, in both Figs. 3a and 3b the magnitude of the RTT values and their variation are much larger than the differences between the tools. Therefore, Fig. 4 shows the pairwise difference between each RTT value for `ePPing` compared to `PPing` and `tshark`, respectively. Note that `tshark` reports an RTT value for every ACK, whereas `PPing` and `ePPing` only produce an RTT for ACKs with a new TSecr value, thus providing 13 % fewer RTT samples than `tshark` in this experiment (see the count field in Fig. 3b). Therefore, Fig. 4b only includes the RTT values from `tshark` that correspond to those from `PPing` and `ePPing`, i.e. the ones from the first ACK with each TSecr value. Furthermore, differences below 1 µs may be due to rounding as the RTT values from `tshark` and `PPing` have microsecond resolution.



	min	median	mean	max	std
Difference	-101.473	-1.216	-1.220	0.535	0.528

(a) Difference between `ePPing` and `PPing`.



	min	median	mean	max	std
Difference	-97.473	3.407	3.321	927.927	3.260

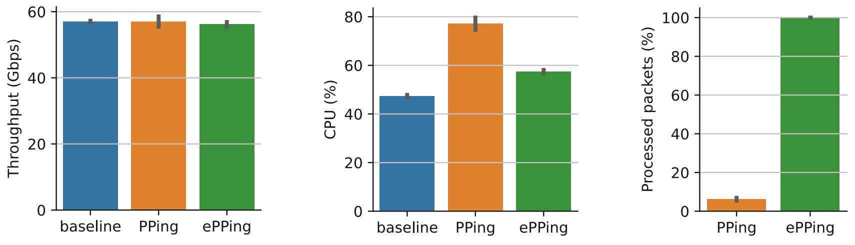
(b) Difference between `ePPing` and `tshark`.

Fig. 4. Pairwise difference between RTT values reported by `ePPing` compared to other tools.

Overall, ePPing reports slightly lower RTT values than `PPing`. This is expected as the XDP hook used by `ePPing` for ingress traffic is triggered before the packet enters the rest of the Linux network stack, and can be captured by `tcpdump`. On the other hand, `ePPing` provides RTT values that are around 1 to 5 µs higher than those from `tshark`, which is explained by `tshark` calculating the RTT in a different way. Both `PPing` and `ePPing` use TCP timestamps, and will therefore always include the additional latency caused by delayed ACKs. Meanwhile, `tshark` instead matches sequence and acknowledgement numbers, which will often exclude this delay component. We have verified that the differences between `ePPing` and `tshark` correspond to the additional latency component from delayed ACKs. While the difference in how delayed ACKs are handled result in very small differences in Fig. 4b, it can create larger differences for some particular traffic patterns. In Appendix A we further discuss how relying on TCP timestamps affect the calculated RTT values.

3.2 Monitoring Overhead

The motivation behind implementing `ePPing` in eBPF was to reduce overhead and thus allow it to work at higher packet rates. Therefore, we measure what impact `ePPing` has on the forwarding performance when running on a machine that is under high packet processing load. This is done by measuring the throughput `iperf3` is able to achieve when sending TCP traffic from the sender to the receiver end host. The test is first performed without any passive monitoring on the middlebox to establish a baseline, and is then repeated with either `ePPing` or `PPing` running at the capture point. We run each test 10 times for 120 s, but discard the results from the first 20 s as a warm-up phase to let cache usage and CPU frequency scaling stabilize. We then repeat the tests using 1, 10, 100 or 1000 TCP flows to evaluate how performance is affected by the number of flows.



(a) Throughput (b) CPU (average across all 6 cores) (c) Packets processed

Fig. 5. Forwarding performance without monitoring (baseline), with PPing and with ePPing for 10 concurrent `iperf3` flows, when middlebox uses all CPU cores.

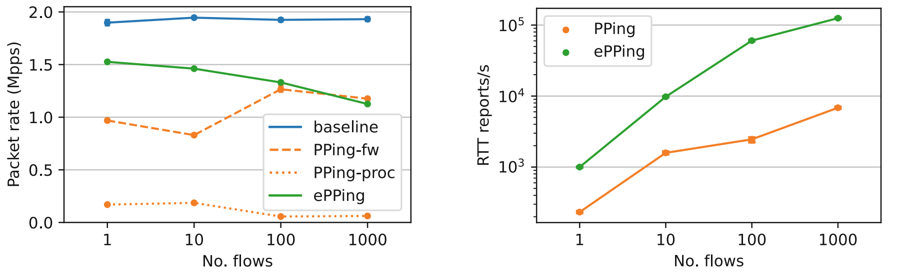
Figure 5 shows the performance that is achieved with 10 concurrent flows, which is when the end hosts are able to push the traffic at the highest rate in our experiments. While Fig. 5a shows that neither `PPing` or `ePPing` has a considerable impact on the forwarding throughput, Fig. 5b shows that `ePPing` has much lower CPU overhead. With a baseline utilization of 47 %, `ePPing` only increases it to 57 %, while `PPing` increases it all the way to 77 %. Meanwhile, Fig. 5c shows that despite `PPing` having roughly 3 times higher CPU overhead than `ePPing`, `PPing` is actually only processing just over 6 % of the packets. This is due to the packet capturing being unable to keep up with the high packet rate, and therefore missing the majority of the packets. In contrast, `ePPing` runs in line with the rest of the network stack, and sees every packet, meaning it processes roughly 16 times as many packets. While not apparent from Fig. 5, also note that `PPing` is implemented as a single-threaded user space application, and is therefore limited to how fast a single core can process all the logic. While the user space component reporting the RTT values in `ePPing` is also single-threaded, the eBPF programs that contain the logic for calculating the RTT values run on the cores that the kernel assigns to process each packet, thus distributing the load across multiple cores in the same manner as the normal network stack processing.

Table 1. Average packets per second processed on single core at capture point when only forwarding (baseline scenario).

No. flows	Packet rate (Mpps)		
	Tx	Rx	Total
1	1.86	0.04	1.90
10	1.86	0.09	1.95
100	1.72	0.21	1.92
1000	1.64	0.29	1.93

Although the results in Fig. 5 are promising, the end hosts are usually the bottleneck here, especially as we increase the number of flows. These experiments are consequently unable to push the middlebox and `ePPing` to their limits. We therefore constrain the middlebox to using a single CPU core in the remaining experiments, moving the bottleneck to the middlebox CPU. This means that the middlebox is already using all of its CPU capacity just forwarding the traffic, and any additional overhead from the passive monitoring results in decreased throughput. Furthermore, we emphasize the total packet rate (sum of transmitted and received packets) rather than the throughput. Packet rate is more relevant for the performance of `PPing` and `ePPing`, as their logic has to run per packet, and also stays more consistent across a varying number of flows as Table 1 shows. As the number of flows increases, the number of ACKs sent back by the receiver increases (seen by the increase in received packets at the middlebox). This results in less capacity to forward data packets by the middlebox (decrease in transmitted packets), and thereby a lower throughput, while the total packet rate handled remains similar.

Figure 6a summarizes the impact `PPing` and `ePPing` have on the forwarding performance of the middlebox when it is constrained to a single core. Both `ePPing` and `PPing` now have a considerable impact on the forwarding performance, but `ePPing` clearly sustains a higher packet rate than `PPing`, at least at a limited number of flows. As the number of flows increases, the packet rate with ePPing drops from 1.53 to 1.13 Mpps. The reason for this drop in performance as the number of flows increases is that, due to the limited update rate of TCP timestamps, the number of potential RTT samples that ePPing has to process increases with number of flows. This is evident in Fig. 6b, which shows that while ePPing reports the expected 1000 RTT values per second for a single flow, at 1000 flows this increases to roughly 125,000 values per second.



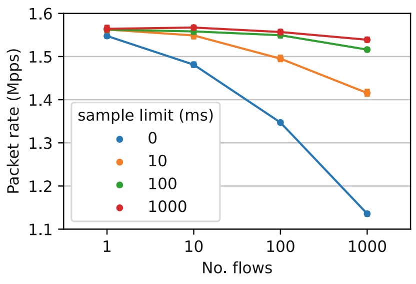
(a) Average packet rate (b) Average RTT report rate (log scale)

Fig. 6. Middlebox performance when just forwarding (baseline), with PPing or ePPing on a single CPU core. PPing misses most packets and thus processes (PPing-proc) packets at a much lower rate than they are forwarded (PPing-fw).

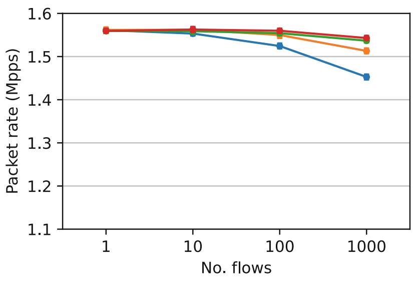
Meanwhile, the forwarded packet rate with PPing actually appears to increase slightly with number of flows (0.97 Mpps at one flow, 1.18 Mpps at 1000 flows), but this is merely due to the packet capturing missing a larger fraction of packets. The packet rate actually handled by PPing drops from approximately 170 kpps at 1 and 10 flows, to just 60 kpps at 100 and 1000 flows, meaning ePPing processes packets at approximately an 18 times higher rate than PPing at 1000 flows. PPing missing the majority of packets results in it also missing many RTT samples, which can be seen by the much fewer RTT values reported by PPing in Fig. 6b. Furthermore, the algorithm for matching packets to replies that PPing and ePPing uses, relies on matching the first instance of each TSval to the first matching TSecr. As PPing does not see every packet, it cannot guarantee this, and it may therefore introduce small errors in its RTT values.

However, limiting the load by sampling, as PPing in practice does by missing packets, can be a valid approach. The high rate of RTT values reported by ePPing may not be necessary, or even desirable, for many use cases. We therefore implement sampling for ePPing, and evaluate if it can be an effective way to reduce the overhead. While it would be possible to only process a random subset of the packets, similar to PPing, such an approach has several drawbacks. As already mentioned, missing packets may interfere with the algorithm for matching packets and replies, thus resulting in less accurate RTT values. Furthermore, a random subset of packets is likely to mainly yield RTT samples from elephant flows, and largely miss sparse flows. However, sparse flows often carry control traffic and other latency sensitive data, and being able to monitor their RTT may therefore be at least as important as the RTT of the elephant flows. Instead of eliminating `ePPing`'s advantage of being guaranteed to see every packet, we opt to implement a simple per-flow sample rate limit. With the sample rate limit, `ePPing` must wait a time period t after saving a timestamp entry for a packet before it can timestamp another packet from the same flow. This t may either be set to a static value, or it can be dynamically adjusted to the RTT of each flow, so that flows with shorter RTTs get more frequent samples than flows with longer RTTs.

We repeat the experiments from Fig. 6, setting the sample limit t to 0, 10, 100 and 1000 ms, in practice corresponding to at most 1000, 100, 10 and 1 RTT values per flow and second, respectively. Figure 7a summarizes the results, and clearly shows that less frequent sampling greatly reduces the overhead of `ePPing`. Already at a sample limit of 10 ms we see great improvements. When limiting it to a single sample every 1000 ms per flow (the default rate of `ping`), `ePPing` is able to sustain a packet rate of 1.54 Mpps for 1000 flows, compared to 1.14 Mpps without sampling. The drop in forwarding performance when going from 1 to 1000 flows thus decreases from 27 % without sampling, to just 1.6 % at $t = 1000$ ms. The drawback of such coarse sampling is that the granularity of the monitoring is reduced, and one might miss important RTT variations.



(a) Reporting individual RTTs



(b) Aggregated RTT reports

Fig. 7. Impact of different levels of per-flow sample limiting and aggregation. Note that the Y-axis does not start at 0.

As an alternative approach to sampling, aggregation can be used to reduce the overhead from frequent RTT reports. We therefore also implement a bare bones aggregation functionality to evaluate the feasibility of aggregating the RTT values directly in the kernel. When aggregation is enabled, the eBPF programs add each RTT value to a global histogram in a BPF map, instead of sending every RTT value directly to user space. Additionally, the minimum and maximum RTTs are tracked. The user space then pulls the aggregated RTT statistics once per second and prints them out. Figure 7b shows the results when repeating the experiment in Fig. 7a using the aggregation. As can be expected, with a high sample limit, and consequently few RTT values to aggregate, the aggregation yields a very modest improvement. However, for smaller sample limits the aggregation becomes more beneficial. In the scenario without any sampling, the aggregation increases the packet rate at 1000 flows from 1.14 to 1.45 Mpps. By combining sampling and aggregation, we therefore expect `ePPing` to be able to maintain a high level of performance, while still providing useful RTT metrics, at significantly more than 1000 concurrent flows. Due to limitations with the current testbed, we are however unable to validate performance beyond 1000 flows.

4 Conclusion

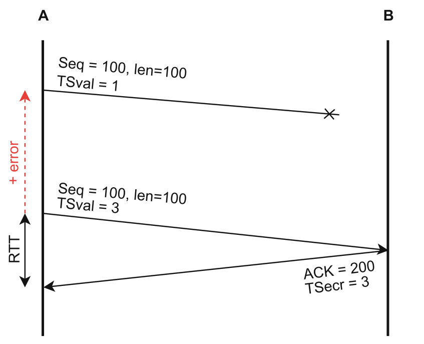
In this paper we propose using eBPF to passively monitor network latency, and demonstrate the feasibility of this by implementing evolved Passive Ping (`ePPing`). By using eBPF, `ePPing` is able to efficiently observe packets as they pass through the Linux network stack without the overhead associated with packet capturing. It does not require any modifications to the kernel or replacing the network stack with DPDK, nor any special hardware support. Our evaluation shows that `ePPing` delivers accurate RTTs and has much lower overhead than `PPing`, being able to handle over 1 Mpps on a single core, corresponding to more than 10 Gbps of throughput. We also demonstrate that sampling and aggregation of RTT values in the kernel can be used to further reduce the overhead from handling a large amount of RTT samples.

While `ePPing` overall performs well in our experiments, our evaluation is heavily based on bulk TCP flows generated by `iperf3`. In future work we intend to evaluate how `ePPing` fares with a more realistic workload by using traffic from an ISP vantage point. Another important aspect to consider is what impact the passive monitoring has on end-to-end latency. We are currently working on better understanding `ePPing`'s impact on end-to-end latency. Preliminary findings indicate that while `ePPing` only adds a couple of hundred nanoseconds of processing latency to each packet (99th percentile of approximately 350 ns), it may under certain scenarios increase end-to-end latency by hundreds of microseconds.

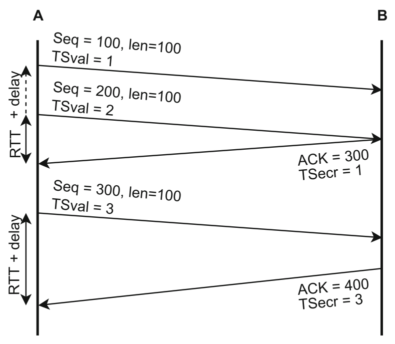
Furthermore, our current implementation of `ePPing` has some limitations. Limitations inherent to using TCP timestamps are further discussed in Appendix A, with one of the primary ones being the lack of ability to monitor flows where TCP timestamps are not enabled. Some of the these limitations could be avoided by using sequence and acknowledgement numbers instead, although that has its own set of limitations. We are also considering adding support for other protocols, such as DNS and QUIC. Additionally, the sampling and aggregation methods we employ in this work are relatively simple, and we are working on more sophisticated ways to sample, filter and aggregate RTTs in-kernel to provide enhanced RTT metrics while maintaining low overhead.

A Effects of Using TCP Timestamps to Infer RTT

As briefly explained in Sect. 3.1, using TCP timestamps to match packets to their corresponding ACKs may yield slightly different RTTs than when matching sequence and ACK numbers. We here discuss these differences in further detail, covering the pros and cons of each approach to passively monitor network latency.



(a) Retransmission. Sequence and acknowledgement matching suffer from the retransmission ambiguity, which may cause a large over- or underestimation of the RTT. TCP timestamps can separate the original from the retransmitted packet and do therefore not have the ambiguity.



(b) Delayed ACK. TCP timestamps will always calculate the RTT between the first packet being ACKed and the ACK, always including the additional latency from a delayed ACK. Matching sequence and acknowledgement numbers will only include the additional latency if the delayed ACK was triggered by a timeout.

Fig. 8. TCP timestamps and sequence and ACK numbers: Differences.

The decision to use TCP timestamps for `ePPing` was mainly based on having a simple algorithm that avoids the TCP transmission ambiguity. As illustrated in Fig. 8a, retransmissions can cause approaches that match sequence and ACK numbers to greatly overestimate or underestimate the RTT, unless they also detect retransmissions to filter out such spurious RTT samples. However, for TCP timestamps, the retransmission will typically have a newer TSval, and, therefore, no additional precautions are needed to calculate a correct RTT.

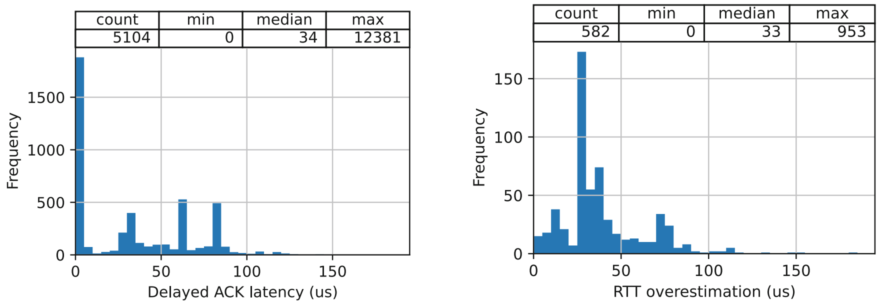
```
if $SEG.TSval \geq TS.Recent$ and $SEG.SEQ \leq Last.ACK.sent$ then
    $TS.Recent \leftarrow SEG.TSval$;
end
```
Algorithm 1: RFC 7323 algorithm for how to update *TS.Recent*, which is copied into the TSecr field when an ACK is sent.

Due to how TSecr is updated, TCP timestamps also handle delayed ACKs a bit differently compared to sequence and ACK number matching: The echoed TSecr value is not necessarily the latest TSval. Rather, RFC 7323 [5] specifies that TSecr should be set to a recent Tsval, which is updated according to Algorithm 1, and essentially results in TSecr being set to the TSval from the oldest in-order unacknowledged segment. The effect of this is that RTTs based on TCP timestamps will, by design, always include the additional latency from delayed ACKs. On the other hand, matching sequence and ACK numbers will only include the delayed ACK if it is triggered by a timeout, as shown in Fig. 8b. Consequently, matching sequence and ACK numbers will usually result in RTTs that are a bit closer to the underlying network latency, whereas using TCP timestamps will result in RTTs more similar to those experienced by the TCP stack. Both methods are, however, prone to include RTT spikes caused by delayed ACKs timing out.

There are also two noteworthy drawbacks with relying on TCP timestamps: Firstly, TCP timestamps are optional, and `ePPing` can therefore only monitor TCP traffic with TCP timestamps enabled. A recent study [2] found that out of the most common operating systems (Android, iOS, Windows, MacOS and Linux), Windows was the only one not supporting TCP timestamps by default. A lot of traffic these days goes through mobile devices running Android and iOS, but Windows is still the dominant desktop OS, making this a noteworthy limitation. Secondly, the TCP timestamp update rate limits how frequently we can collect RTT samples within a flow. The study in [2] found that among servers for popular websites, the most common update rate was once per millisecond, which is what Linux uses since v4.13, but some updated at a slower rate of every 4 ms or every 10 ms. For most applications we deem that 1000–100 RTT samples per second per flow is plenty, but for very fine-grained analysis requiring an RTT sample for every ACK this could be problematic.

Furthermore, there are two edge cases in which matching TCP timestamps may result in slightly overestimating the RTT beyond the delayed ACK component: The first case is when a retransmission happens fast enough that the TSval is not updated from the original transmission. For example, consider if the retransmission in Fig. 8a would still use $TSval = 1$. This can only occur if the retransmission occurs faster than the TCP timestamp update rate, and may at most overestimate the RTT with the TCP timestamp update period. With TCP timestamps typically being updated every millisecond, this should be very rare in most environments outside of for example data center networks. The second case is when the TSval is updated during a delayed ACK and persists into packets being acknowledged by the next ACK. For example, consider if the third packet sent by A in Fig. 8b would still have $TSval = 2$. In that case, the RTT for the second ACK sent by B ($ACK = 400$) would incorrectly be calculated from the second packet sent by A ($Seq = 200$) instead of from the third packet sent by A ($Seq = 300$). This error can occur in the presence of delayed ACKs, and if multiple packets within a flow have the same TSval. Thus, this is also bounded to at most overestimate the RTT with one TCP timestamp period. This edge case is more likely to occur than retransmissions without updated timestamps, however, the magnitude of the error is still small compared to the spikes that can be caused by delayed ACKs.

Finally, Fig. 9 shows an example of how the different handling of delayed ACKs and overestimations due to the second edge case can impact RTTs based on TCP timestamps when compared to matching sequence and ACK numbers. Here, a modified version of the experiment from Fig. 4 is used, where the latency applied with `netem` is 50 ms and the traffic is sent in a burstier manner by using `iperf3`'s internal pacing at 50 ms intervals (-b 100M –pacing-timer 50000). Figure 9a shows the additional latency due to the handling of delayed ACKs, which is typically in the range between 0 to 100 µs, with three instances exceeding 1 ms, and one reaching 12.4 ms. Meanwhile, Fig. 9b shows the overestimation for the 582 out of the 5104 RTT samples where the second edge case occurs. These overestimations are of a similar scale as the difference due to delayed ACKs, although the maximum error is just under 1 ms. In contrast, both using TCP timestamps and matching sequence and ACK numbers result in a few RTT values of over 92 ms, exceeding the configured latency by over 40 ms, due to delayed ACKs timing out.



count	min	median	max
5104	0	34	12381

count	min	median	max
582	0	33	953

(a) Difference from including and excluding the additional latency component of delayed ACKs.

(b) Additional overestimation of RTT from TCP timestamps due to the second edge case.

Fig. 9. Difference between RTTs computed by matching TCP timestamps compared to sequence and acknowledgement numbers.

In summary, using TCP timestamps may result in slightly higher RTT values than matching sequence and ACK numbers, mainly due to different handling of delayed ACKs. While `ePPing` could be modified to instead operate on sequence and ACK numbers, it would then risk missing valid RTT samples, especially on lossy links, and would still capture the largest RTT spikes from delayed ACKs.

References

1. Abranches, M., Michel, O., Keller, E., Schmid, S.: Efficient network monitoring applications in the Kernel with eBPF and XDP. In: IEEE NFV-SDN 2021 (2021). https://doi.org/10.1109/NFV-SDN53031.2021.9665095
2. Barbette, T., Wu, E., Kostić, D., Maguire, G.Q., Papadimitratos, P., Chiesa, M.: Cheetah: A high-speed programmable load-balancer framework with guaranteed per-connection-consistency. IEEE/ACM Trans. Netw. **30**(1), 354–367 (2022). https://doi.org/10.1109/TNET.2021.3113370
3. Barreda-Ángeles, M., Arapakis, I., Bai, X., Cambazoglu, B.B., Pereda-Baños, A.: Unconscious physiological effects of search latency on users and their click behaviour. In: SIGIR 2015 (2015). https://doi.org/10.1145/2766462.2767719
4. Birge-Lee, H., Wang, L., Rexford, J., Mittal, P.: SICO: surgical interception attacks by manipulating BGP communities. In: CCS 2019 (2019). https://doi.org/10.1145/3319535.3363197
5. Borman, D., Braden, R.T., Jacobson, V., Scheffenegger, R.: TCP Extensions for High Performance. Technical report. RFC 7323, Section 3, Internet Engineering Task Force (2014). https://doi.org/10.17487/RFC7323
6. Bosshart, P., et al.: P4: programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev. **44**(3), 87–95 (2014). https://doi.org/10.1145/2656877.2656890
7. Chen, X., Kim, H., Aman, J.M., Chang, W., Lee, M., Rexford, J.: Measuring TCP round-trip time in the data plane. In: SPIN 2020 (2020). https://doi.org/10.1145/3405669.3405823
8. Cheshire, S.: It's the Latency, Stupid (2001). http://www.stuartcheshire.org/rants/Latency.html. Accessed 07 May 2022
9. Combs, G.: Tshark (2022). https://www.wireshark.org/docs/man-pages/tshark.html. Accessed 17 May 2022
10. Ghasemi, M., Benson, T., Rexford, J.: Dapper: data plane performance diagnosis of TCP. In: SOSR 2017 (2017). https://doi.org/10.1145/3050220.3050228
11. Heist, P.: IRTT (Isochronous Round-Trip Tester) (2021). https://github.com/heistp/irtt. Accessed 31 Oct 2022
12. Hillmann, P., Stiemert, L., Rodosek, G.D., Rose, O.: Dragoon: advanced modelling of IP geolocation by use of latency measurements. In: ICITST 2015 (2015). https://doi.org/10.1109/ICITST.2015.7412138
13. Høiland-Jørgensen, T., et al.: The eXpress data path: Fast programmable packet processing in the operating system kernel. In: CoNEXT 2018 (2018). https://doi.org/10.1145/3281411.3281443
14. Isovalent: Cilium - Linux Native, API-Aware Networking and Security for Containers (nd). https://cilium.io. Accessed 21 Oct 2022
15. Iyengar, J., Thomson, M.: QUIC: A UDP-Based Multiplexed and Secure Transport. Technical report, RFC 9000, Section 17.4, Internet Engineering Task Force (2021). https://doi.org/10.17487/RFC9000
16. Kuznetsov, A., Yoshifuji, H.: Iputils (2022). https://github.com/iputils/iputils. Accessed 03 May 2022
17. Li, J., Wu, C., Ye, J., Ding, J., Fu, Q., Huang, J.: The comparison and verification of some efficient packet capture and processing technologies. In: DASC/PiCom/CBDCom/CyberSciTech 2019 (2019). https://doi.org/10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00177

18. Maheshwari, R., Krishna, C.R., Brahma, M.S.: Defending network system against IP spoofing based distributed DoS attacks using DPHCF-RTT packet filtering technique. In: ICICT 2014 (2014). https://doi.org/10.1109/ICICICT.2014.6781280
19. Meta: Katran: A high performance layer 4 load balancer (2022). https://github.com/facebookincubator/katran. Accessed 21 Oct 2022
20. Mockapetris, P.: Domain names - implementation and specification. Technical report, RFC 1035, Section 4.1.1, Internet Engineering Task Force (1987). https://doi.org/10.17487/RFC1035
21. Nichols, K.: PPing: Passive ping network monitoring utility (2018). https://github.com/pollere/pping. Accessed 21 Sep 2021
22. RIPE NCC: Home—RIPE Atlas (nd). https://atlas.ripe.net/. Accessed 20 Oct 2022
23. Sengupta, S., Kim, H., Rexford, J.: Continuous in-network round-trip time monitoring. In: SIGCOMM 2022 (2022). https://doi.org/10.1145/3544216.3544222
24. Sharma, S., Woungang, I., Anpalagan, A., Chatzinotas, S.: Toward tactile internet in beyond 5G era: recent advances, current issues, and future directions. IEEE Access. **8**, 56948–56991 (2020). https://doi.org/10.1109/ACCESS.2020.2980369
25. Shawn Ostermann: Tcptrace (2013). https://github.com/blitz/tcptrace. Accessed 03 Apr 2022
26. Sundberg, S., Brunstrom, A., Ferlin-Reiter, S., Høiland-Jørgensen, T., Brouer, J.D.: Efficient continuous latency monitoring with eBPF - Resources (2023). https://doi.org/10.5281/zenodo.7555410
27. Sundberg, S., Høiland-Jørgensen, T.: BPF-examples: PPing using XDP and TC-BPF (2022). https://github.com/xdp-project/bpf-examples/tree/master/pping. Accessed 26 Jan 2023
28. The Bufferbloat community: Bufferbloat.net (nd). https://www.bufferbloat.net/projects/. Accessed 05 May 2022
29. The Linux Foundation: eBPF - Introduction, Tutorials & Community Resources (2021). https://ebpf.io/. Accessed 03 May 2022
30. Vieira, M.A.M., et al.: Fast packet processing with eBPF and XDP: concepts, code, challenges, and applications. ACM Comput. Surv. **53**(1), 1–36 (2020). https://doi.org/10.1145/3371038
31. Vlahovic, S., Suznjevic, M., Skorin-Kapov, L.: The impact of network latency on gaming QoE for an FPS VR game. In: QoMEX 2019 (2019). https://doi.org/10.1109/QoMEX.2019.8743193
32. Wang, H., Zhang, X., Chen, H., Xu, Y., Ma, Z.: Inferring end-to-end latency in live videos. IEEE Trans. Broadcast. **68**(2), 517–529 (2022). https://doi.org/10.1109/TBC.2021.3071060
33. Xhonneux, M., Duchene, F., Bonaventure, O.: Leveraging eBPF for programmable network functions with IPv6 segment routing. In: CoNEXT 2018 (2018). https://doi.org/10.1145/3281411.3281426
34. Zhao, Z., Gao, S., Dong, P.: Flexible routing strategy for low-latency transmission in software defined network. In: ICCBN 2021 (2021). https://doi.org/10.1145/3456415.3456444
35. Zheng, Y., Chen, X., Braverman, M., Rexford, J.: Unbiased delay measurement in the data plane. In: APOCS 2022 (2022). https://doi.org/10.1137/1.9781611977059.2