COMPUTER SCIENCE
DATAVETENSKAP

Building the next generation of programmable networking - powered by Linux

YEAR 2 REPORT, OCTOBER 2022

Frey Alfredsson, Simon Sundberg, Per Hurtig and Anna Brunstrom

1 Introduction

Programmable networking has the potential to enable new applications, as well as increase the flexibility of existing ones. Over the last years, the performance of general purpose computers has reached the point where it has become practical to perform high-speed packet processing in software, thus enabling programmable networking. Several frameworks have emerged to enable this, such as the DataPlane Developer Kit (DPDK). These frameworks have adapted a clean-slate design to maximize performance, which however means that existing mature network management tools are harder to integrate with them. On the other hand, networking stacks in modern operating systems are featureful and well-integrated into the ecosystem, but lack the performance to keep up with the specialized frameworks. Finally, networking hardware is starting to become ever more programmable, leading to a desire to integrate programmable hardware features with the software stack.

The Linux networking community has reacted to these challenges by integrating a new data path into the Linux kernel, called the eXpress Data Path (XDP). This runs inline with the regular data path, allowing flexible high-performance programmable networking to function in concert with the regular networking stack. In addition, some network adapters have adopted the BPF byte code format used by XDP as an option for offloading programmable processing to the hardware. This makes XDP a promising technology for solving the problems of integration between existing stacks, high-speed packet programming in software, and hardware offloading of programmable features.

While XDP shows promise, there are several open problems that need to be resolved before the vision of an integrated architecture for programmable networking can be achieved. To explore these problems and offer solutions at both the architectural and technical implementation levels, Red Hat (RH) and Computer Science at Karlstad University (KAU), Sweden, have engaged in a joint research project funded by RH Research. Below we report on the progress of this project during its second year. We summarize the technical work, describe the outreach activities that have been carried out, and describe the project organization.

2 Technical Work

The first year activities established the relevant technical background and narrowed down the scope of the project to focus on two main areas: enhancing XDP with support for queuing and utilizing BPF/XDP for efficient latency monitoring. Activities during the second year of the project have continued the technical development in these two areas as further summarized below.

2.1 Queueing in XDP

XDP provides a high-performance programmable network data path and allows programmers to process packets early out of the driver and even bypass the kernel's network stack for increased performance. While XDP excels in forwarding packets, it currently has no mechanism for queuing or reordering packets and can not implement traffic scheduling policies, pacing, or shaping. Packet scheduling is a common task on network equipment, and most devices that provide packet scheduling algorithms provide only a handful of predefined packet schedulers. The Linux kernel's network stack provides a traffic control system called Queuing discipline (Qdisc) capable of packet scheduling. However, it does not provide the means of implementing complete packet schedulers in BPF, nor can it be used in XDP when bypassing the kernel's networking stack. Therefore, a new extension to XDP is needed.

Below we present our work on adding programmable packet scheduling to XDP. We have designed a programmable packet scheduling framework in BPF using recently proposed schemes

for programmable queues. This extension allows programmers to define their packet schedulers using BPF while benefiting from the XDP fast data path.

2.1.1 Programmable Queues

Traditionally, network equipment and operating systems provide a handful of packet scheduling algorithms. This limited number of schedulers forced operators to tune the parameters of these algorithms to meet their requirements. However, modern networking equipment has started to provide programmable network capabilities. This new offering allows network designers to create custom logic and customized packet scheduling solutions optimized for their unique use case.

Consequently, the building blocks of a programmable packet scheduling framework need to be straightforward, fast, and easy to understand. Therefore, the essential part of creating a programmable packet scheduling framework is to provide a flexible data structure for programmable queues. Subsequently, this has become an active research area to develop a flexible data structure that programmers can leverage to implement their packet schedulers. However, due to the vast speeds of modern network interfaces, severe time limitations are put on these data structures, which tend to make generic priority queues, such as red-black trees and binary heaps, unsuitable choices.

In our work, we focus on using the Push-In First-Out[1] (PIFO) priority queue and the Eiffel[2] extension to PIFO in our BPF scheduling framework. The PIFO is a simple data structure with the following qualities: (i) it uses an integer-based ranking function to queue packets by their priority and relies on a fixed range of ranks known at initialization time; and (ii) it dequeues packets according to the scheduled rank order. Furthermore, it has the flexibility to create complex packet scheduling algorithms by using hierarchies. The Eiffel extension allows scheduling decisions on dequeue, while a traditional PIFO only makes scheduling decisions on enqueue. This distinction stems from the fact that PIFOs are well suited for hardware and software. However, Eiffel is software-only and is capable of more complex operations, such as flow-based scheduling, which requires scheduling on dequeue.

PIFO: The Push-In First-Out (PIFO) data structure is a priority queue where the programmer can queue packets based on their rank, where each rank is queued in a FIFO order, as shown in Figure 1. However, the programmer can only dequeue from the head of the PIFO and only order packets on enqueue. Also, the PIFO does not allow reordering the packets within a FIFO. This limitation inhibits the PIFO from implementing some packet scheduling algorithms.

Despite its simplicity, the PIFO data structure is quite versatile and allows the programmer to implement various packet schedulers. From a programmer's perspective, the programmer only needs to decide what order to schedule packets and when to schedule them for non-work conserving algorithms. However, one limitation of the PIFO is that each FIFO, and therefore, each rank, consumes memory. Therefore, the more granular ranks, the more resources the PIFO needs. This limitation limits the size of the PIFO in hardware and software implementations. A mitigation to this limitation is the SP-PIFO [3], which approximates a large PIFO using a smaller PIFO.

Eiffel extension to PIFO: Eiffel is a software-only extension to the PIFO data structure and adds a couple of alterations to the traditional PIFO. First, with Eiffel, the programmer can schedule flows and packets, depending on the algorithm, where each flow is a separate FIFO that contains only the flow's packets. The Eiffel extended data structure can schedule references to those FIFOs instead of the individual packets. Second, the programmer can do on-dequeue scheduling; this allows the algorithm to virtually rearrange all the packets from the same flow

[1] `http://web.mit.edu/pifo/`
[2] `https://www.usenix.org/conference/nsdi19/presentation/saeed`
[3] `https://sp-pifo.ethz.ch/`

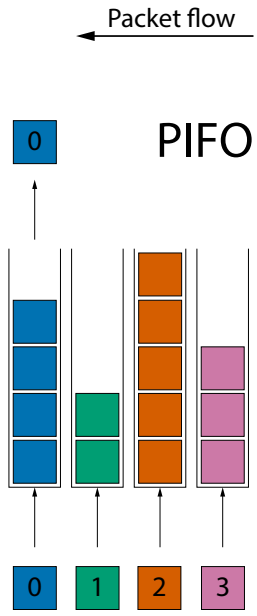


Figure 1: The PIFO data structure is a priority queue where the programmer enqueues packets by rank. However, the dequeue operation can only dequeue packets from the top of the structure. Therefore, the lower priority queues will starve if the highest priority queue keeps getting packets without being emptied.

by simply requeueing the reference to that flow back into the Eiffel extended PIFO. These two additions allow the Eiffel PIFO to represent more scheduling algorithms than a traditional PIFO.

Also, because Eiffel is software-only, the Eiffel paper proposes a few software-based optimizations to improve the performance of the Eiffel data structure. One such optimization is to use a bit-field to keep track of queues containing packets and use the CPU's Find-First-Set [4] instruction to accelerate the lookup in the bit lookup table. The lookup table can be further segmented into a tree-based lookup table when Eiffel has many ranks.

2.1.2 A Programmable Packet Scheduling Framework in BPF

While XDP excels at forwarding packets, it does not support packet reordering or scheduling. Our contribution is the extension of XDP to support packet scheduling in BPF using PIFO data structures with Eiffel capabilities. This support includes adding helper functions, a new PIFO BPF map data type, and an extra hook for dequeuing packets from the PIFO data structure. Figure 2 show the design and basic building blocks of our new programmable packet scheduling framework without the specifics of XDP or the Qdisc subsystem.

A more specific description of each component of this new addition is as follows:

- PIFO map: This new BPF map implements a PIFO and is the main building block for the programmer to create packet schedulers. The framework provides two variants of this map, a PIFO that can store network packets and a PIFO that can store any data type, including other PIFOs. The second data type gives the programmer a convenient way of creating PIFO hierarchies and scheduling flows instead of packets. These maps come with new helper functions to enqueue and dequeue new packets, and a peek function for the generic PIFO map.

- Enqueue hook: This is the standard XDP hook, and in addition to the original capabilities of XDP, it is now capable of redirecting packets to our new PIFO maps.

[4] The Find-First-Set machine instruction finds the first set bit in a CPU register.

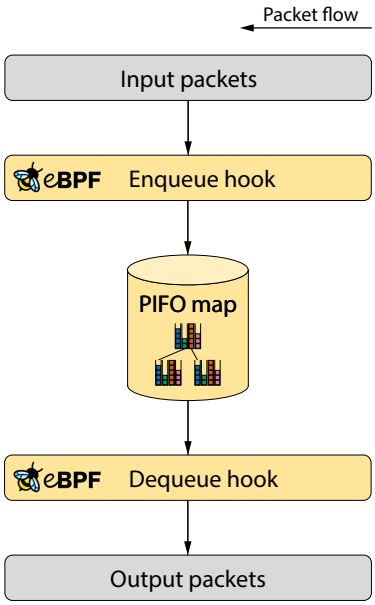


Figure 2: The diagram shows the flow of packets through our proposed BPF-based programmable packet scheduling framework. The framework adds a new dequeue hook to the Linux kernel and a PIFO map. For a fully functional scheduler, the XDP hook must redirect packets to a PIFO, and the dequeue hook must dequeue packets to an interface.

- Dequeue hook: This new hook is responsible for delivering the packet from the packet scheduling algorithm. The hook can be dequeued multiple times for added performance before the interface transmits the packets.

Representing the PIFO data structure as a BPF map presents programmers with a familiar map interface that they have come accustomed to in BPF. From a programmer's perspective, the BPF hooks reference the queues like any other map type. The enqueue hook decides which queue to direct the packet to by a map reference. Similarly, the dequeue hook picks which queue map to dequeue from and returns a reference to the dequeued packet to the kernel for transmission.

2.1.3 Status of the framework

The implementation of the new XDP scheduling framework involves two separate development efforts. One is the implementation of the new BPF hooks and queue map in the kernel, and the second is the implementation of several real-world queueing algorithms using the new framework.

We currently have a working version of the new BPF scheduling API, which includes the PIFO data type and the deqeue BPF hook. As detailed in Section 3, the scheduling API has been presented to the Linux kernel community on several occasions during 2022. In response, the kernel community has given us feedback that will impact the framework's design before the maintainers are ready to include it in the upstream kernel.

Parallel to working on the new BPF scheduling API, we have created a new prototyping framework that allows us to develop new scheduling algorithms quickly. The framework loads the XDP schedulers into the kernel without attaching them to network interfaces. The framework's Lua scripting language enables us to write testing scripts capable of enqueueing and dequeuing individual packets to test the correctness of our schedulers.

2.2 Latency Monitoring

Network latency has a large impact on the Quality of Experience (QoE) for interactive applications running over the network. As networks evolve to support highly interactive real-time applications

which require reliable and low latency, monitoring network latency to ensure a high QoE is becoming more important than ever. Furthermore, network latency monitoring is also useful to validate Service Level Agreements (SLAs), identifying and troubleshooting network issues and detecting some man-in-the-middle types of attacks. However, continuously monitoring network latency is challenging and current tools for network latency monitoring have several limitations. In this project we therefore investigate the feasibility of using eBPF to enable efficient and continuous network latency monitoring by implementing an evolved Passive Ping (ePPing).

During the first year an initial prototype of ePPing was developed. During the second year several refinements to the implementation were made, including several optimizations to the core eBPF components as well as improvements to the setup and teardown process of the program as the surrounding eBPF landscape has evolved. Additionally both the performance and accuracy of ePPing have been evaluated through a series of experiments. During the experiments further limitations of the original Passive PPing (PPing), which ePPing aims to improve upon, were discovered and measured. The experiments also revealed opportunities to further improve ePPing which will be further evaluated in future work. The work with ePPing is summarized in subsections 2.2.1 - 2.2.4.

2.2.1 Limitations with current latency monitoring solutions

Among current solutions to monitor network latency, the simple ping utility is often used to measure the Round Trip Time (RTT). Ping works by sending an ICMP packet to the desired target and measuring the time until it receives an echo reply. This is a form of active network monitoring, where a probe is injected on the network and then monitored. In addition to ping there exists many other tools that can actively monitor network latency, such as IRTT[5] and RIPE Atlas[6].

While active network monitoring works well for measuring the idle network latency in a controlled manner, it is not well suited to infer the network latency for real application traffic. The network probes add overhead in the form of additional network traffic, which may interfere with the normal application traffic. Active monitoring may also require control over the monitored targets in order to be able to deploy monitoring clients and servers, which may for example not be feasible for ISPs who wish to monitor the latency between their customers and popular web services. Furthermore, active latency monitoring only reports the latency experienced by the network probe, which may not reflect the latency experienced by various application traffic. Network probes and application traffic may be treated differently by any network devices on the path, for example load balancers, firewalls and Active Queue Management (AQM), and may therefore experience very different network latency.

Passive network latency monitoring avoids these issues by observing existing network traffic instead of sending out separate probes. By matching packets in one direction with replies in the opposite direction, passive latency monitoring can directly infer the network latency that actual application traffic experiences. Additionally, such passive latency monitoring can be performed on any device on the path that is able to observe packets in both directions, such as gateway routers or firewalls, not just the endhosts.

Unfortunately the selection of tools that can passively monitor network latency is far more limited and have various other limitations. Tcptrace can only run on packet capture files, and while Wireshark/tshark can run on live traffic it keeps records of every packet in memory, making it unsuitable for continuous monitoring over a longer period of time. In 2018 Kathleen Nichols developed Passive Ping (PPing)[7] which can continuously monitor TCP RTTs on live traffic without needing to keep records of all packets. However, like most software based passive network monitoring, PPing still relies on traditional packet capturing to observe the packets, and packet

[5] https://github.com/heistp/irtt
[6] https://atlas.ripe.net/
[7] https://github.com/pollere/pping

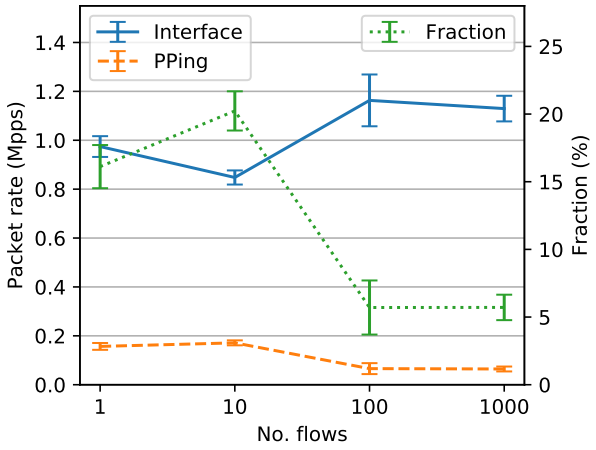


Figure 3: Packet processed by PPing compared to all packets traversing the monitored interface

capturing has high overhead and fails to keep up with the high packet rates encountered on modern multi-Gbps links. This is demonstrated in Figure 3, where PPing only sees between 5 - 20 % of all packets during an experiment. By missing packets PPing not only misses potentially valuable RTT samples, but may also get less accurate RTTs as its algorithm requires seeing every packet in a flow to ensure that it does not mismatch packets.

To support passive latency monitoring at high packet rates, researchers at Princeton University have instead proposed implementing continuous monitoring of TCP latency in P4[8]. While P4 solutions can support high packet rates, they require hardware with P4 support, such as Tofino switches. P4 based solutions are thus not possible to deploy on many existing network devices, such as routers and firewalls, that lack P4 support. As a purely software based alternative, eBPF can be used to passively monitor the network on Linux devices. Some solutions for monitoring network latency using eBPF already exists. The tools tcpconnlat and tcprtt in the BPF Compiler Collection (BCC) repository can passively monitor TCP RTTs, but only work directly on endhosts as they depend on the kernel's TCP stack.

In this work we are instead investigating if it is possible to use eBPF to implement continuous network latency monitoring by only observing the packets, similar to PPing, but without the overhead and potential to miss packets that packet capturing suffers from. Such a solution can be deployed on any Linux device which sees traffic in both directions, without any special hardware support and without requiring any control over the endhosts or any other network devices except the one the monitoring solution is deployed at.

2.2.2 Design and implementation of eBPF based Passive Ping

To evaluate the feasibility of using eBPF for efficient and continuous network latency monitoring, an evolved Passive Ping (ePPing) was implemented. The design of ePPing is heavily inspired by Kathleen Nichols PPing, but generalizes the design to support additional protocols while also employing some optimizations to further reduce the overhead of the eBPF based implementation.

The fundamental mechanic to passively monitor RTTs in ePPing is to observe when packets and their corresponding replies are seen. This works by observing the packets at a capture point, which can be one of the endhosts or any device between the endhosts that sees the packets in both directions, such as a gateway router or firewall. Once a packet is observed at the capture point it is parsed for two identifiers, a message identifier and a reply identifier. If the packet has a valid message identifier the time when the packet was observed is saved in a hashmap, using a combination of the flow tuple and the per-flow message identifier to get a globally unique key.

[8]Sengupta, S., Kim, H., Rexford, J.: Continuous in-network round-trip time monitoring. In: SIGCOMM 2022.

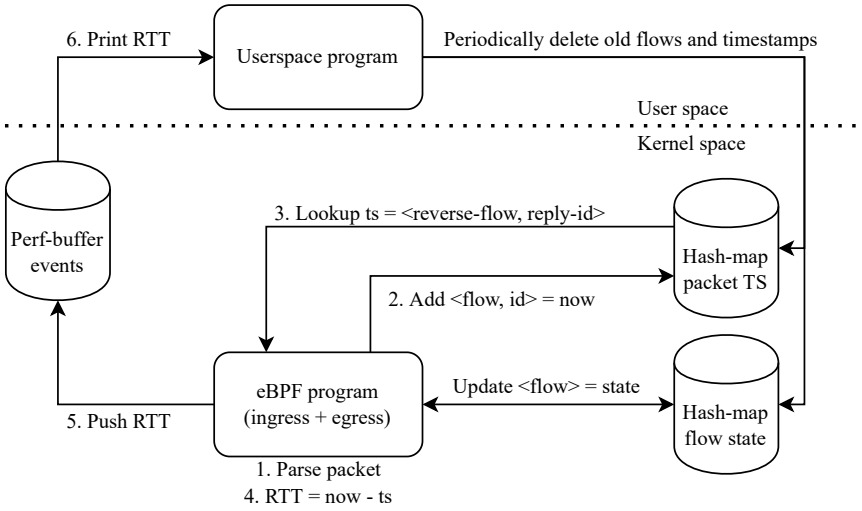


Figure 4: Design overview of ePPing.

If the packet contains a valid reply-identifier, a lookup for the corresponding previous message-identifier in the reverse direction is performed, and if a match is found the RTT between the previous message and the corresponding reply can be calculated by subtracting the saved time for when the original message was observed from the time when the reply was observed.

What is used as the message and reply identifiers depend on the protocol. For TCP traffic, the TCP timestamp option is used. When TCP timestamps are enabled, each TCP header contains two additional fields, `TSval` and `TSecr`. The sender puts its own timestamp in the `TSval` field when sending a packet, and the receiver echos a recently seen `TSval` in the `TSecr` field. Thus the `TSval` can be used as a message identifier, and `TSecr` can be used to match a reply to the original packet. One important aspect of TCP timestamps to consider is that they are updated at a limited rate (once per millisecond on Linux), and thus multiple packets may have the same `TSval` and multiple replies may have the same `TSecr`. To avoid mismatching packets and compute faulty RTTs only the first instance of a particular `TSval` may be matched against the first corresponding `TSecr`. In addition to TCP, ePPing currently also supports ICMP traffic, where the echo request sequence number is used as the packet identifier and the echo reply sequence number is used to match the reply.

To avoid the overhead from packet capturing, the XDP and tc eBPF hooks are utilized. These hooks trigger whenever a packet is received or transmitted, and gives the attached eBPF program direct access to the contents of the packet inside the Linux kernel, without having to copy the packet. The logic for parsing the packets ①, saving the time the packets were observed ②, matching replies against saved timestamps ③ and calculating the RTTs ④ have therefore been implemented in an eBPF program as illustrated in Figure 4. Once the eBPF program has calculated an RTT it pushes it together with related data, such as the flow, to a user space process ⑤ which just prints out the information ⑥. One instance of the eBPF program is attached to egress and one to ingress, so that the program is triggered every time a packet is sent or received. The eBPF program uses BPF hash maps to store state between each invocation, such as the packet timestamps and some state for each flow, like number of packets and bytes per flow. The user space process is relatively simple, it loads and attaches the eBPF program at startup, prints out the RTT reports that the eBPF program pushes to it, and tears down and cleans up after the eBPF program on shutdown. Additionally, the user space program also periodically triggers a cleanup process which deletes old entries from the BPF hash maps.

While the overall design of ePPing is still similar to the initial prototype developed during the first year, several improvements have been made. Some examples include making ePPing symmetrical so that it can calculate RTTs in both directions of a flow, adding an option to use

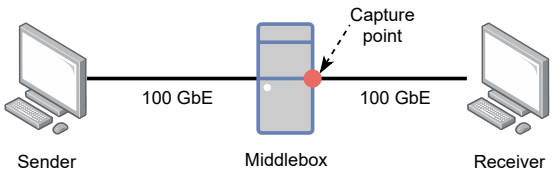
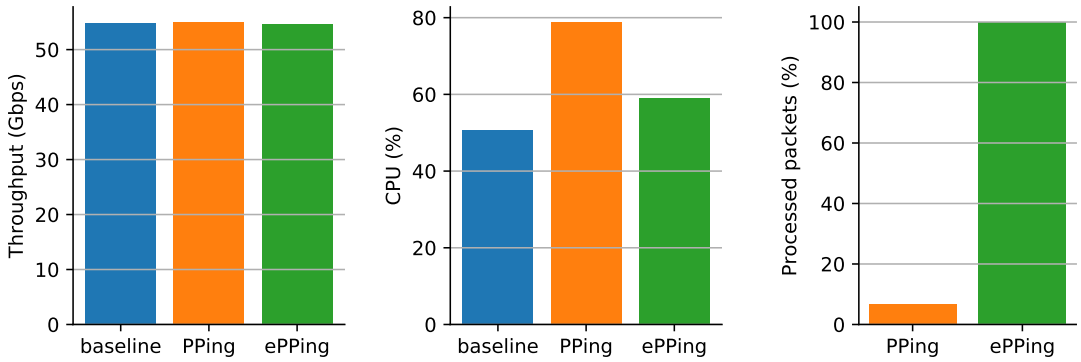


Figure 5: Testbed setup



(a) Throughput (b) CPU (c) Packets processed

Figure 6: Performance for 10 concurrent iperf3 flows

tc-ingress instead of XDP in case the NIC does not have XDP driver support and adding an option to dynamically adapt the number of RTT samples based on the current RTT of the flow. Additionally several optimizations have been made, such as reducing the number of required hash map lookups and earlier and more efficient removal of stale entries from the hash maps.

2.2.3 Results

To evaluate the performance and overhead of ePPing, a testbed was set up as depicted in Figure 5, where two endhosts are connected to a middlebox that forwards the traffic between them. Offloads such as GRO, GSO and TSO were disabled on the middlebox so that it had to process each individual packet. Iperf3 was then used to generate TCP traffic and the experiments were repeated while either running PPing or ePPing at the middlebox, as well as without any passive monitoring to establish a baseline.

Figure 6 shows an example of the performance that was achieved when running 10 concurrent flows. Neither PPing or ePPing has a noticeable impact on the throughput in this scenario. While both PPing and ePPing have some CPU overhead, the overhead for ePPing is clearly lower despite ePPing handling all packets whereas PPing only processes about 6 % of the packets.

However, in this initial test setup the endhosts are typically the bottleneck and are therefore unable to push the middlebox, PPing or ePPing to their limits. The bottleneck was therefore moved to the middlebox by limiting it to a single core, so that the CPU was fully utilized with just forwarding the traffic, and any overhead from PPing or ePPing resulted in a lower throughput. The impact PPing and ePPing had on the throughput across a varying number of flows in this bottlenecked scenario is summarized in Figure 7. Note that while PPing has been included for reference here, the comparison to it is somewhat skewed as PPing misses most of the packets as shown in Figure 3.

The results show that ePPing does have a considerable impact on the forwarding performance in this bottlenecked scenario, but is still able to handle over 10 Gbps on a single core (while unlike PPing processing all packets). The overhead of ePPing does however increase with the number of flows. The reason for this increase in overhead is that more flows result in more RTT samples

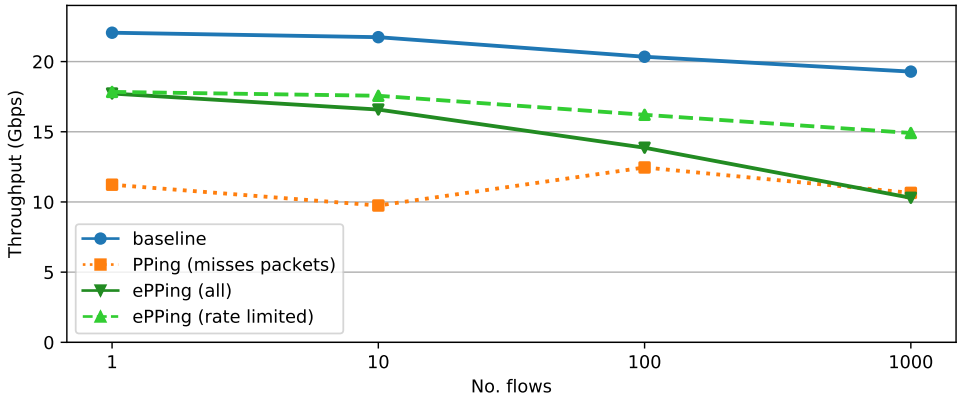


Figure 7: Impact of monitoring overhead in bottlenecked scenario

to calculate and report. At 1000 flows, over 130,000 RTTs per second were reported. Such a large amount of RTT samples is often not necessary or even desirable. Therefore, an optional RTT sample rate limit was added, which limits how often RTTs can be calculated per flow. The dashed green line in Figure 7 shows the performance when limiting ePPing to at most generate one RTT sample every 100 ms (or correspondingly 10 samples per second) per flow. The rate limiting substantially decreases the overhead at larger amounts of flows at the cost of reduced RTT sample granularity. The rate limit can also be automatically adapted to each flow's RTT, so that flows with short RTTs that are prone to more rapid changes can be monitored at a higher granularity than flows with higher RTTs.

Additional testing also revealed that most of the benefit from rate limiting the RTT samples did not come from the eBPF programs having to calculate fewer RTTs, but rather from not having to push as many RTT reports to user space and print them out. Therefore, future work will look at ways to filter and aggregate RTT samples in a better way to both reduce the overhead from reporting and make it easier to process the reported RTTs for valuable insights.

Experiments were also performed to determine the accuracy of the RTTs reported by ePPing, both for ICMP and TCP traffic. In general the RTTs were found to be very accurate and within a few microseconds of the RTTs reported by ping (for ICMP traffic) and PPing (for TCP) traffic, and largely consistent with the RTTs reported by tshark (Wireshark) which calculates the RTTs based on sequence and ACK numbers instead of TCP timestamps. However, a few edge cases have been found when calculating RTTs using TCP timestamps as done by PPing and ePPing may result in slight over-estimations. These edge cases are currently being further investigated to determine the extent of them and how to best handle them.

2.2.4 Ongoing and future work

While ePPing is currently functional and usable, work is still ongoing to further improve it. Testing has shown that reporting large amounts of RTT samples has considerable overhead, and the large amount of output ePPing may generate makes it challenging to deploy in practice. Therefore different options for how to filter and aggregate the RTT samples before reporting them are currently being investigated. In the future we also hope to extend ePPing to support additional protocols, such as DNS and QUIC. Furthermore the accuracy and performance of ePPing is continuously being evaluated as additional improvements and features are added.

Once a suitable approach for aggregating and filtering RTT samples has been implemented, the plan is to deploy ePPing in different environments to evaluate it for different use cases. Potential targets for deployment include smaller ISP gateways, where a Danish ISP operator as well as people from LibreQoS has shown some interest, and the AIDA edge observability framework being developed as part of the AIDA project at Karlstad University.

3 Communication and Outreach

In parallel to the technical work in the project, several activities to promote the project and connect with external partners from both the Linux community and from the networking research community has been carried out. Toke Høiland-Jørgensen has provided the Linux kernel community with an RFC and has presented the new scheduling framework at the Linux Plumbers 2022 conference in Dublin, Ireland. Furthermore, Freysteinn Alfredsson presented the scheduling framework at Lund Linux Con 2022 in Lund Sweden.

To spread information about the work to the research community, Freysteinn Alfredsson gave a talk on bringing packet queueing to XDP at the 10th Inria Workshop On Systems, held at Rennes, France, and online in 2021. Simon Sundberg presented a poster paper on the latency monitoring at ACM Internet Measurement Conference 2021, held as a virtual event. Freysteinn and Simon also presented their work on the scheduling framework and on latency monitoring, respectively, at the Swedish National Computer Networking Conference 2022 in Stockholm. The presentations were accompanied by short papers that summarized the work.

Simon Sundberg also presented the work on latency monitoring to Red Hat staff and partners at the Red Hat Research Day Europe 2022 in Brno. The project is also represented with a project page on both the RH and KAU web sites.

The project has as an explicit goal to make its results available as open access, open source software and open data. The developed code[9] and other project materials[10] are freely available on GitHub. The project embraces an open collaboration model and we would be very happy to receive comments, pull requests or other feedback on our work.

4 About the Project

As mentioned in the introduction, the project is a collaboration between RH and KAU. The core project team consists of Principal Kernel Engineer Toke Høiland-Jørgensen (RH), Senior Principal Kernel Engineer Jesper Dangaard Brouer (RH), Professor Anna Brunstrom (KAU), Associate Professor Per Hurtig (KAU), PhD student Freysteinn Alfredsson (KAU) and PhD student Simon Sundberg (KAU). The project funding from RH is being used to fund the PhD position for Freysteinn Alfredsson. PhD student Simon Sundberg is funded by national Swedish funding sources, but collaborates in the project based on his research interests.

As during the first year, the project team has had bi-weekly online project meetings to follow up on the work in the project and discuss technical issues. In addition to the core project team, additional staff members from RH and KAU have taken part in the bi-weekly meetings on a per interest and availability basis. In addition to the bi-weekly online meetings, a physical two-day project workshop was organized in Karlstad in early March 2022. As the first year project kick-off had to be virtual due to the pandemic, this was a great opportunity for the project team to finally all meet in person and for in depth technical discussions.

The combination of research expertise on the KAU side with the extensive development experience on the RH side has been very fruitful and the collaboration in the project has worked very well. In particular, the involved PhD students have greatly benefited from the detailed code reviews and other feedback on their work provide by the RH team.

[9] pping source code: `https://github.com/xdp-project/bpf-examples`
[10] Other project materials: `https://github.com/xdp-project/bpf-research`