COMPUTER SCIENCE
DATAVETENSKAP

Building the next generation of programmable networking - powered by Linux

YEAR 1 REPORT, SEPTEMBER 2021

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 first year. We summarize the technical work, describe the outreach activities that have been carried out, and describe the project organization.

2 Technical Work

Initial discussions in the project focused on establishing the relevant technical background and narrowing down the scope of the project. As a result, the project partners agreed to focus the technical work on two main areas: enhancing XDP with support for queuing and utilizing BPF/XDP for efficient latency monitoring.

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. While XDP excels in forwarding packets, it currently has no mechanism for queuing or reordering of packets and cannot implement traffic scheduling policies. Packet scheduling is a common task on network equipment, and like for other aspects of networking, there is a growing interest for bringing programmability to this domain. For the Linux kernel, making packet scheduling fully programmable through BPF is the obvious answer to this trend.

Below we introduce our work on adding programmable packet scheduling to XDP. We are designing a programmable packet scheduling framework in BPF using recently proposed schemes for programmable queues. This extension will allow programmers to define their packet schedulers using BPF while benefiting from the XDP fast data path. Our new programmable packet scheduling framework design introduces new BPF hooks and a new BPF map type. We base our new map type on a data structure called Push-In First-Out[1] (PIFO). PIFO allows the program-

[1] `http://web.mit.edu/pifo/`

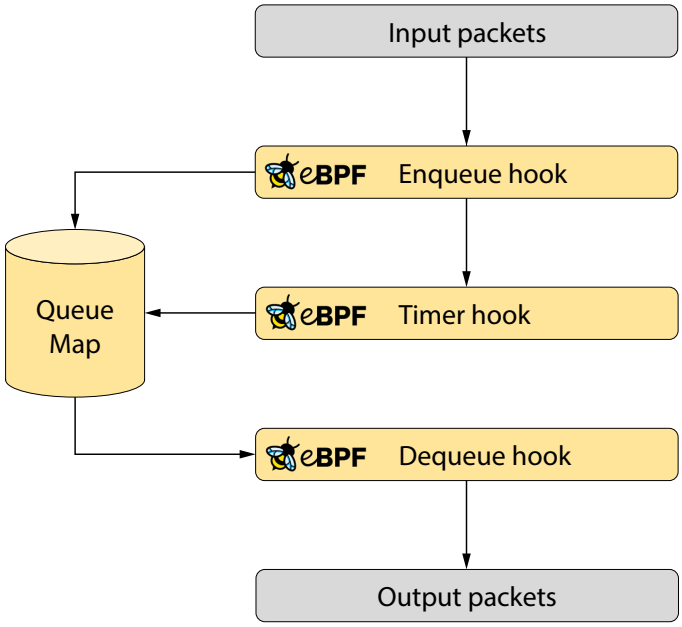

Figure 1: Depicted are the BPF hooks needed to implement programmable packet scheduling using our new proposed framework. The main hooks are the enqueue and dequeue hooks, with the optional timer hook. Queueing works the same in both XDP and the Qdisc. However, they do not use the same enqueue and dequeue hooks.

mer to order the packets on enqueue, and packets are always dequeued at the head of the queue. This data structure allows the programmer to express many practical scheduling algorithms and complex algorithms using a hierarchy of PIFOs.

2.1.1 Programmable packet scheduling framework design

Our vision is to create an BPF based programmable packet scheduling framework that is flexible enough to share between the XDP and kernel Qdisc layers. The preliminary design is depicted in Figure 1, which shows the basic building blocks for our design without the specifics of XDP nor the Qdisc subsystem. These building blocks are then further broken down as follows:

- Queue map: This new BPF map implements a PIFO and is the main building block for the programmer to create packet schedulers.
- Enqueue hook: This BPF hook is responsible for redirecting packets to BPF queue maps. This is the standard XDP hook in the case of XDP. However, it is a new hook in Qdiscs.
- Timer hook: This is an optional hook and is only required for algorithms that delay packets (for e.g., packet pacing or traffic shaping). It is responsible for dequeuing and enqueuing packets from different PIFOs to introduce delayed packets. We implement timers using the newly introduced BPF timer API. This API allows the programmer to schedule BPF programs to be run at a specified time in the future, which can be used to release held packets.
- Dequeue hook: This hook is responsible for delivering the packet from the packet scheduling algorithm. In XDP, this is a new hook that each driver calls to transmit a packet. It can deliver bulking by repeatedly calling the hook to dequeue multiple packets before it transmits them.

The PIFO data structure is represented as an BPF map in our design. This allows the queues to be straight-forwardly referenced by the BPF programs making up the scheduling algorithm.

From a programmer's perspective, the BPF hooks reference the queues like any other map type. The enqueue hook can decide which queue to direct the packet to by a map reference. Similarly, the dequeue hook can pick which queue map to dequeue from, and return a reference to the dequeued packet to the kernel for transmission.

2.1.2 Status of the framework

Implementation of the framework involves two separate development efforts. One is the implementation of the new BPF hooks and queue map in the kernel. The second is the implementation of several real-world queueing algorithms using those hooks, to demonstrate the viability of the new API.

The Linux BPF implementation is currently being prototyped, following the design outlined above. Parallel to this, we have implemented a prototyping framework that allows us to implement different scheduling algorithms quickly and try different API constructs. The framework allows us to experiment with different possible ways to construct packet scheduling algorithms and how convenient they are for programmers to use. This prototyping framework is an ongoing effort and allows us to fine tune the API before finalizing the Linux kernel code.

2.2 Latency Monitoring

For many applications, latency is the most important metric in determining user experience. Being able to measure network latency is therefore essential for understanding network performance, and has also proven invaluable for troubleshooting applications or network miss-configurations. The most well known tool for measuring network latency is probably ping, which reports a Round Trip Time (RTT) to a target node by sending a message and measuring the time until it gets a response. Due to being standardized as part of the ICMP protocol, ping is universally available and usually a good first choice to determine the idle latency between two specific nodes. But the fundamental approach of actively sending out additional network traffic to measure network latency has several problems.

1. It introduces additional network overhead. While a single ICMP packet every second is negligible on most links, increasing the granularity of RTT reports requires sending packets at a higher rate, which could add up to considerable overhead on slower links.

2. It only reports the RTT between a single pair of nodes. To get an overview of the latency in a large network would require running ping between every possible node pairings which is cumbersome and does not scale well.

3. It only provides the latency experienced by the ICMP echos. This latency does not necessarily correspond with the latency experienced by traffic from other applications. Running ping in isolation from other traffic will likely fail to capture latency spikes that are caused by bloated network queues. Even if ping is run concurrently with other traffic, the ping traffic may be treated differently due to for example active queue management, or even routed differently because of e.g. a load balancer.

Passive ping (pping) uses a different approach that avoids these issues. Instead of sending out additional network traffic, pping looks at existing traffic and reports the RTT experienced by this traffic. This means that pping adds no network overhead, can report RTTs to any hosts for which regular network traffic is passing through, regardless of if it's being run on an end host or a middlebox, and the reported latency corresponds to the network latency experienced by the real application traffic.

2.2.1 BPF pping design

Kathleen Nichols proved the feasibility of this approach by implementing pping for TCP traffic, based on the TCP timestamp option. Kathie's C++ implementation (referred to as k-pping below), like most user space programs, uses the traditional but rather inefficient technique of copying packet headers to user space and parsing them there. At high line rates copying all packet headers to user space is very resource demanding, and it may not be possible for the program to keep up with the network traffic, leading to missed packets.

With the development of BPF pping (e-pping) in the project, we want to leverage the power of BPF to fix this inefficiency. Using BPF, the packets can be parsed directly in kernel space while they pass through the network stack, without ever being copied to user space. This approach allows pping to keep up with higher line rates and imposes less overhead. Furthermore, e-pping adds some additional features, such as JSON output, and extends pping beyond TCP so it can be used to monitor a wider range of traffic. Currently it works for TCP traffic that uses the TCP timestamp option and ICMP echo messages, but could be extended to also work with for example TCP seq/ACK numbers, the QUIC spinbit and DNS queries.

The fundamental logic of pping is to timestamp a pseudo-unique identifier for packets, and then look for matches in the reply packets. If a match is found, the RTT is simply calculated as the time difference between the current time and the stored timestamp. The e-pping tool, just as Kathie's original pping implementation, uses TCP timestamps as identifiers for TCP traffic. The TSval (which is a timestamp in and off itself) is used as an identifier and timestamped. Reply packets in the reverse flow are then parsed for the TSecr, which are the echoed TSval values from the receiver. The TCP timestamps are not necessarily unique for every packet (they have a limited update frequency, normally 1000 Hz for modern Linux systems), so only the first instance of an identifier is timestamped, and matched against the first incoming packet with a matching reply identifier. For ICMP echo, e-pping uses the echo identifier as port numbers, and echo sequence number as identifier to match against.

The design of e-pping is illustrated in Figure 2. It consists of two major components, the

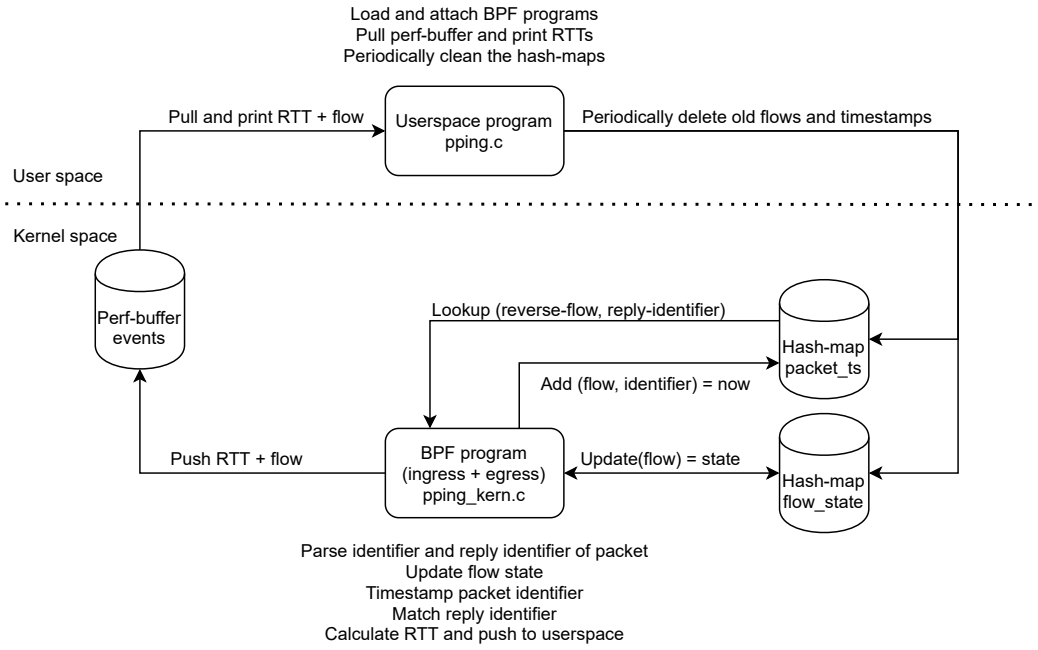

Figure 2: Design overview of e-pping tool.

user space program and the kernel space BPF program. Once the user space program has loaded and attached the kernel space BPF program, the BPF program parses incoming and outgoing

packets, and uses BPF maps to store packet identifiers as well as some state about each flow. When the BPF program can match a reply packet against one of the stored packet identifiers, it pushes the calculated RTT to the user space program which in turn prints it out.

2.2.2 Initial results

We are currently in the process of evaluating the performance of the e-pping tool as well as continuously enhancing its functionality and design. An initial performance result that illustrate the potential of using BPF to parse the packets directly in kernel space, as opposed to performing the parsing in user space, is shown in Figure 3. The results were obtained using a basic setup

	min	median	mean	max
no_pping	0.00	103.48	99.75	117.32
k_pping	2.00	203.08	195.63	210.96
e_pping	0.00	106.52	103.81	115.08

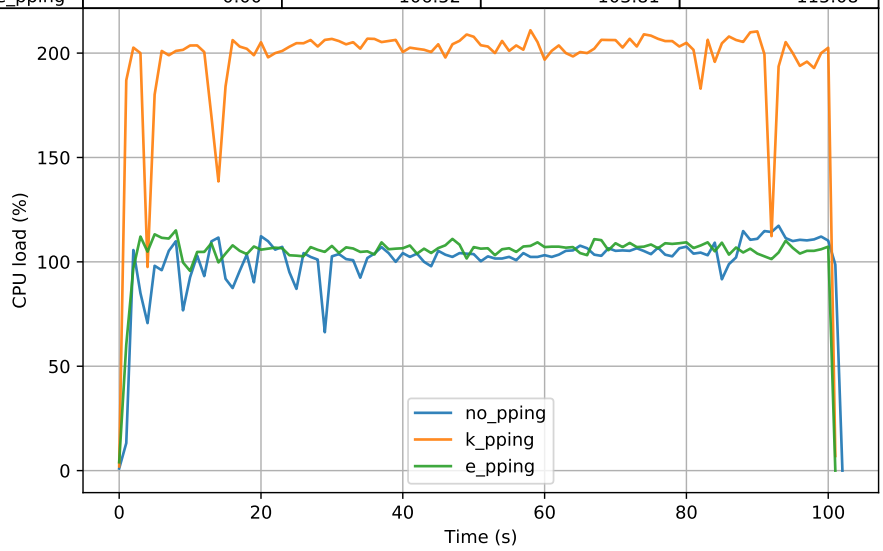

Figure 3: Initial performance results on the CPU overhead of e-pping as compared to k-pping and a baseline without any latency monitoring.

with three virtual machines in a string topology. Iperf3 servers were setup at VM3, e-pping or k-pping was set up on VM2, and iperf3 clients were started on VM1 uploading traffic to VM3 (with the traffic being routed through VM2). The graph displays the CPU load on VM2 when running e-pping, k-pping, or without any latency monitoring, respectively. A single bulk TCP flow is transferred in the experiment. As can be seen in the figure, e-pping introduces negligible CPU overhead as compared to the setup with no latency monitoring, whereas k-pping introduces a significant overhead.

3 Communication and Outreach

While the technical work in the project is still very much in progress, several activities to promote the project and connect with external partners has been carried out. At the start of the project, a news article on the collaboration was published on the KAU web site[2] and the project is also represented with a project page on both the RH and KAU web sites.

Two invited talks related to the project has been given during the first year. Freysteinn Alfredsson from KAU gave a talk on BPF and his work on scheduling for the Simbiosys Systems Discussion Group at Emroy University, US, in June and Simon Sundberg from KAU gave a talk

[2] `https://www.kau.se/en/cs/news/new-project-will-build-next-generation-programmable-networks`

on pping at the meeting group on latency organized by Dave That, also in June. We have also submitted a proposal to present the XDP queueing work at the Inria/Interdigital Workshop On Systems in Brittany, France this coming October. Additionally, the August issue of the Red Hat Research Quarterly magazine featured an interview with Professor Anna Brunstrom from KAU conducted by Toke Høiland-Jørgensen from RH[3]. The conversation covered programmable networking, other ongoing network trends, open source, industry-academia collaboration and more.

The project has as an explicit goal to make its results available as open access, open source software and open data. The developed code[4] and other project materials[5] 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, who started in September 2020. PhD student Simon Sundberg is funded by national Swedish funding sources, but collaborates in the project based on his research interests.

After a planning and recruitment phase, the technical work in the project started in September 2020, when Freysteinn Alfredsson arrived. Following a project kick-off meeting on September 8, the project team has had bi-weekly 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 as well as some representatives from Ericsson have taken part in the bi-weekly meetings on a per interest and availability basis.

While the pandemic has forced all meetings to be carried out online and the project team is still eagerly waiting to all meet for a joint physical workshop, 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.

[3] https://research.redhat.com/blog/article/the-right-idea-at-the-right-time-networking-researchers-use-open-source-for-real-world-results/

[4] pping source code: `https://github.com/xdp-project/bpf-examples`

[5] Other project materials: `https://github.com/xdp-project/bpf-research`