BPF as a fundamentally better dataplane

Daniel Borkmann (Isovalent), co-maintainer BPF

eBPF Summit

October 28, 2020

BPF, a general purpose engine

BPF, a general purpose execution engine

Minimal instruction set architecture

with two major design goals:

BPF, a general purpose execution engine

1) Low overhead when mapping to native code, in particular: x86-64, arm64

BPF, a general purpose execution engine

2) Must be verifiable for safety by the kernel at program load time.

BPF, a general purpose execution engine

Kernel provides framework of building blocks and attachment points.

Is BPF a generic virtual machine? No.

Is BPF a fully generic instruction set? No.

BPF, a general purpose execution engine

It’s an instruction set with C calling convention in mind.

BPF, a general purpose execution engine

Why? Kernel is written in C and BPF needs to efficiently interact with the kernel. Approx 150 BPF kernel helpers, 30 maps.

BPF, a general purpose execution engine

BPF: 114 instructions, 11 registers

x86-64: 2000+ instructions, 16 registers

LLVM is able to translate “BPF-restricted” C into BPF instructions.

How far off is “BPF” C from generic C?

BPF, a general purpose execution engine

BPF has seen major advances such as
BPF-2-BPF function calls, bounded loops,
global variables, static linking, BTF,
up to 1 Mio instructions / program, ...

This already allows for solving a *lot* of interesting production issues.

Few hand-picked examples next ...

Reducing kernel’s attack surface with BPF

Example: DoS via packet of death

Packet hash in network stack’s RX path used for many things (e.g. to steer traffic among CPUs).

Packet hash from NIC might have less entropy, not covering L4 headers or encaps. Kernel recalculates in SW.

Reducing kernel’s attack surface with BPF

net: flow_dissector: fail on evil iph->ihl

```
We don't validate iph->ihl which may lead a dead loop if we meet a IPIP
skb whose iph->ihl is zero. Fix this by failing immediately when iph->ihl
is evil (less than 5).

This issue were introduced by commit ec5efe7946280d1e84603389a1030ccec0a767ae
(rps: support IPIP encapsulation).

Cc: Eric Dumazet <edumazet@google.com>
Cc: Petr Matousek <pmatouse@redhat.com>
Cc: Michael S. Tsirkin <mst@redhat.com>
Cc: Daniel Borkmann <dborkman@redhat.com>
Signed-off-by: Jason Wang <jasowang@redhat.com>
Acked-by: Eric Dumazet <edumazet@google.com>
Signed-off-by: David S. Miller <davem@davemloft.net>
```

(Bug exposed for 2 years, hit many production kernels in meantime.)

Typical (traditional) workflow:

Wait until the fix hits stable kernels.

Then wait until distros backport it to their own kernels. Test it and ...

… then successively rollout new kernel into production, steering traffic away from live nodes for reboot.

Traditional tooling like netfilter would not have protected nodes since hash can be retrieved much earlier.

A BPF-based workflow has 2 options:

1) Drop these bogus packets right in the driver at XDP layer with BPF

Reducing kernel’s attack surface with BPF

BPF program can be atomically added in XDP while live traffic is flowing.

No need to wait for distro kernels.

No reboot / service disruption required.

Kernel upgrade can be planned without pressure later.

2) Replace kernel’s flow dissector with BPF

Reducing kernel’s attack surface with BPF

BPF program doing the packet parsing.

Tailored to the production use case, that is, only parse what is *really* needed.

No bogus packet access, loops, etc, given safety checked by BPF verifier. Also rolled out atomically & without service disruption.

Improving kernel's scalability and extensibility with BPF

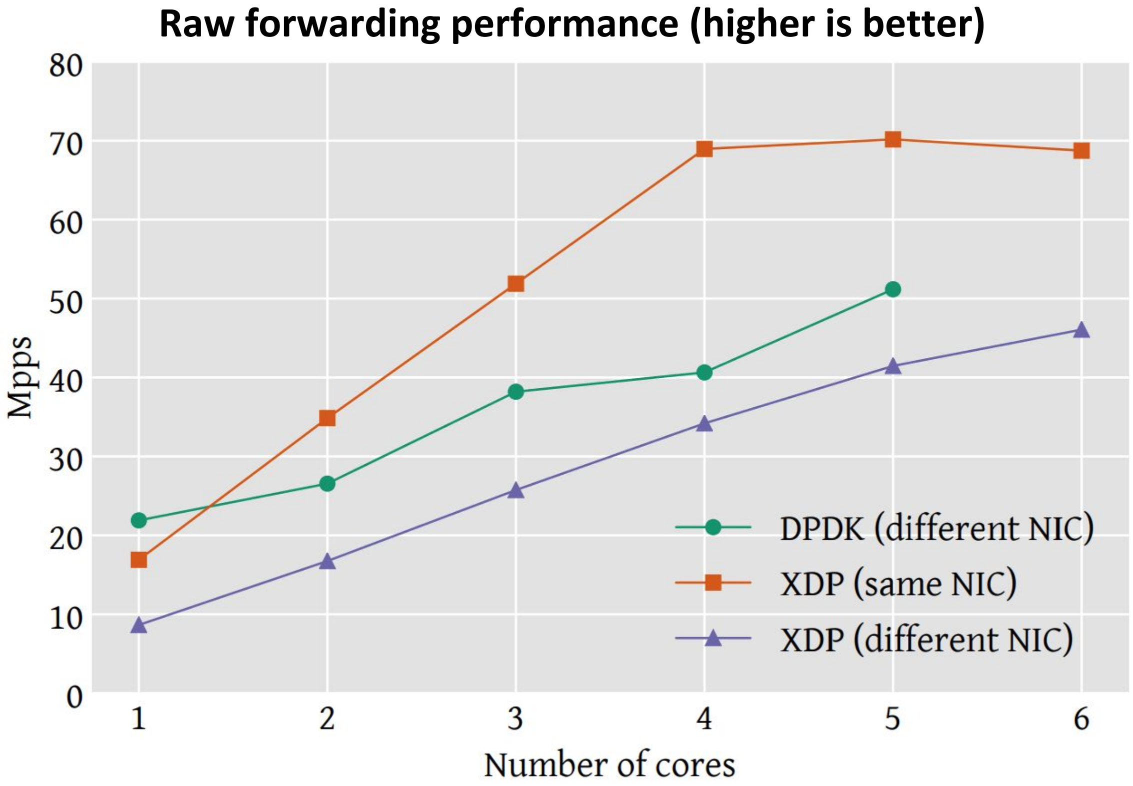
Cloud native example: co-location of service load balancer with regular user workloads on every node.

Improving kernel's scalability/extensibility with BPF

Moving load balancing from legacy subsystems to BPF at XDP layer reduces CPU cost significantly, and achieves DPDK speeds.

https://dl.acm.org/doi/pdf/10.1145/3281411.3281443

Improving kernel's scalability/extensibility with BPF

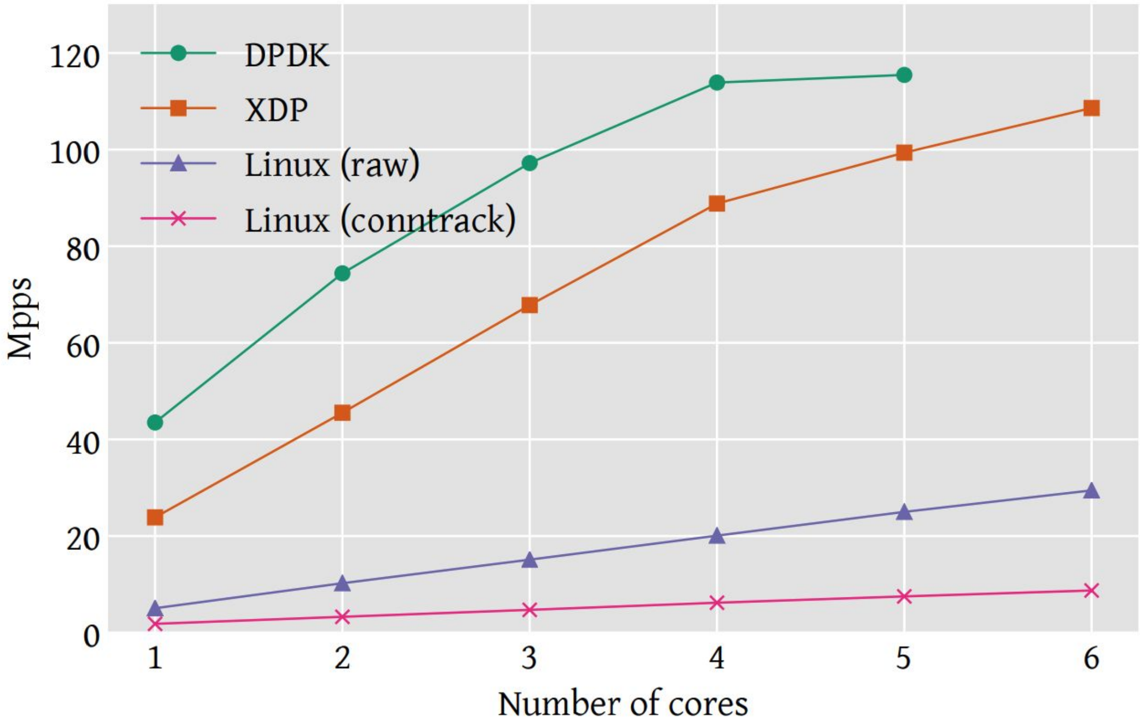


Improving kernel’s scalability/extensibility with BPF

Same for firewalling policies for inbound packets. Freed up CPU cycles can then be spent on user workloads instead.

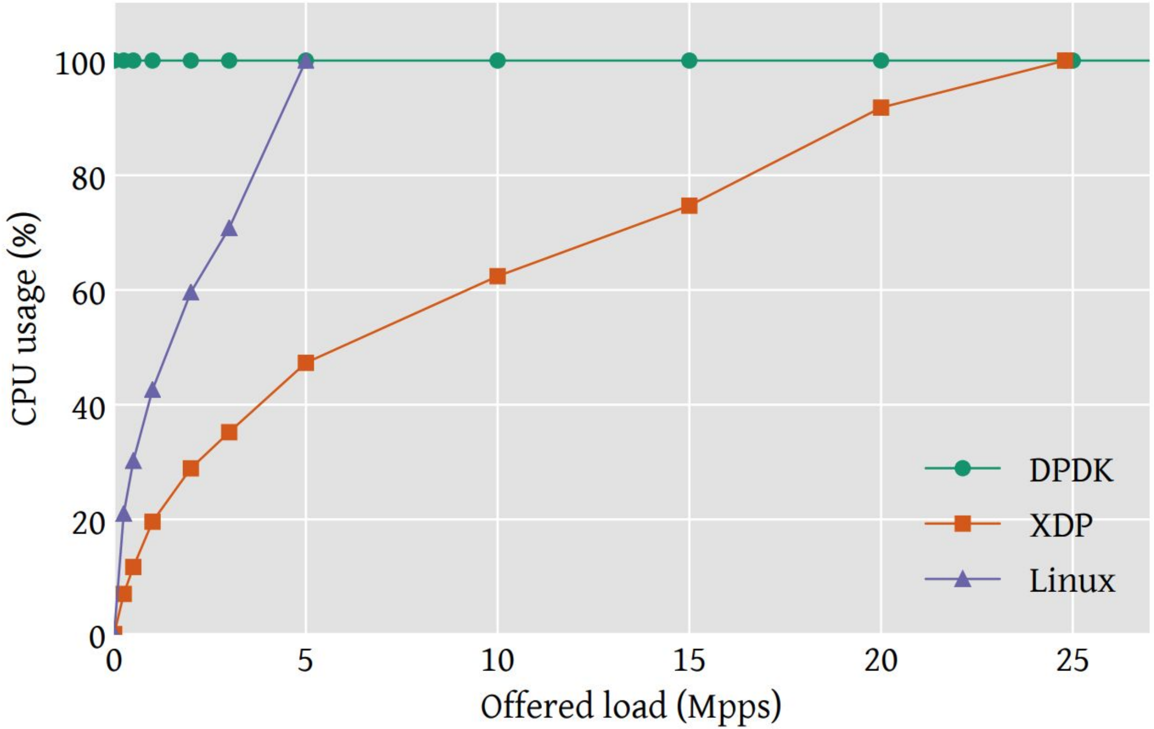
Improving kernel's scalability/extensibility with BPF

DDoS packet drop performance (higher is better)



Improving kernel's scalability/extensibility with BPF

CPU utilization under drop (lower is better)



Improving kernel’s scalability/extensibility with BPF

... all with BPF on upstream kernel drivers. No busy-polling CPUs. No user-kernel boundary crossing.

And with reusing kernel infrastructure such as fib tables via BPF. (Instead of bypassing everything.)

Improving kernel’s scalability/extensibility with BPF

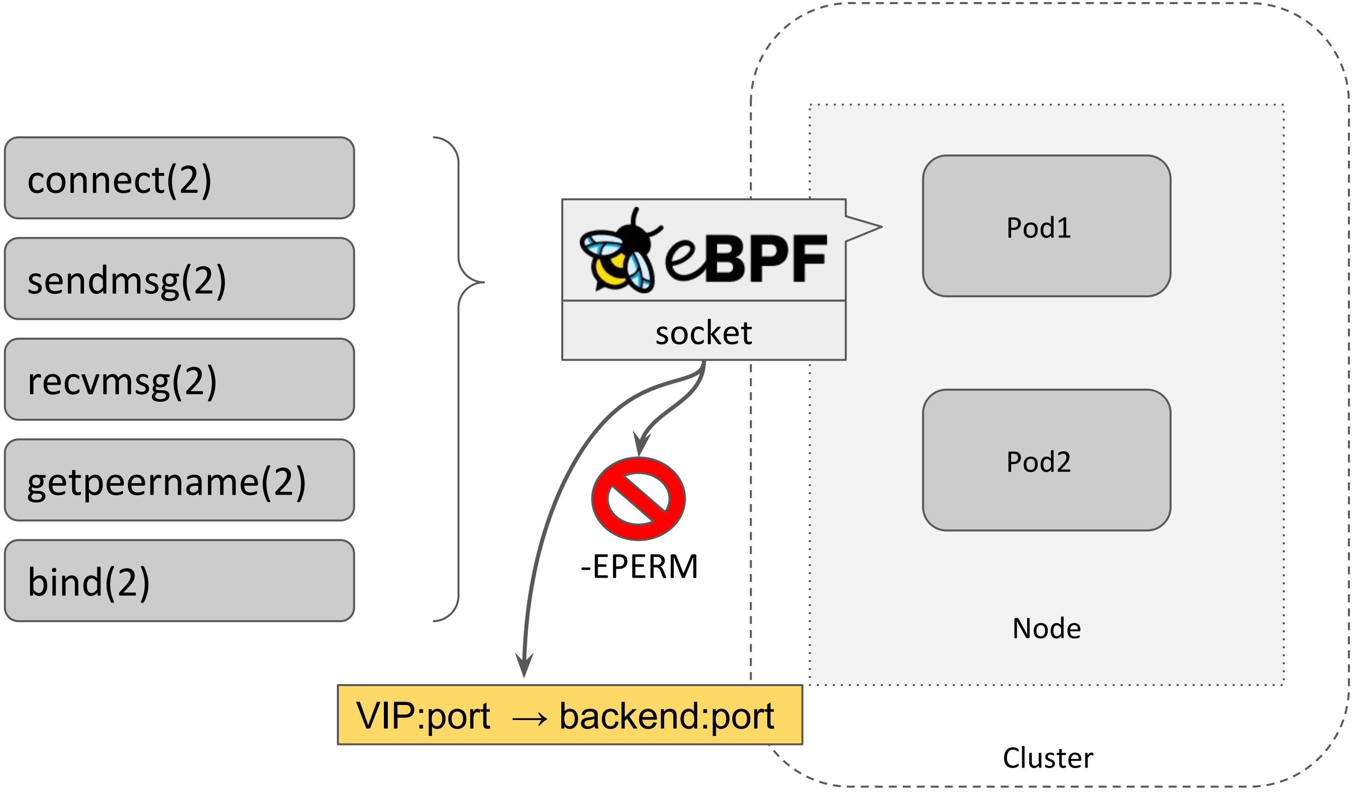
BPF / XDP service load balancers:

Katran, Cilium, Unimog

https://engineering.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/
https://cilium.io/blog/2020/06/22/cilium-18#kube-proxy-replacement-at-the-xdp-layer
https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/

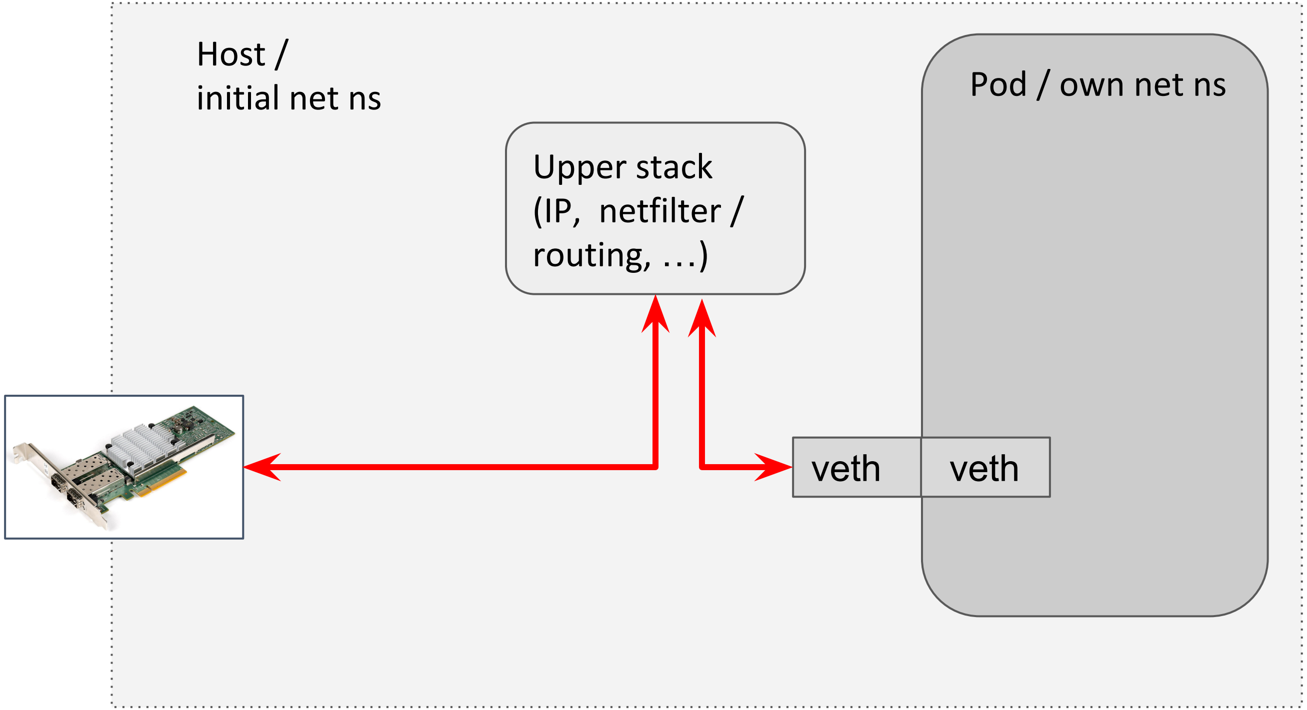
A BPF-based dataplane also helps for policy enforcement or moving traffic in and out of containers or Pods.

Improving kernel's scalability/extensibility with BPF

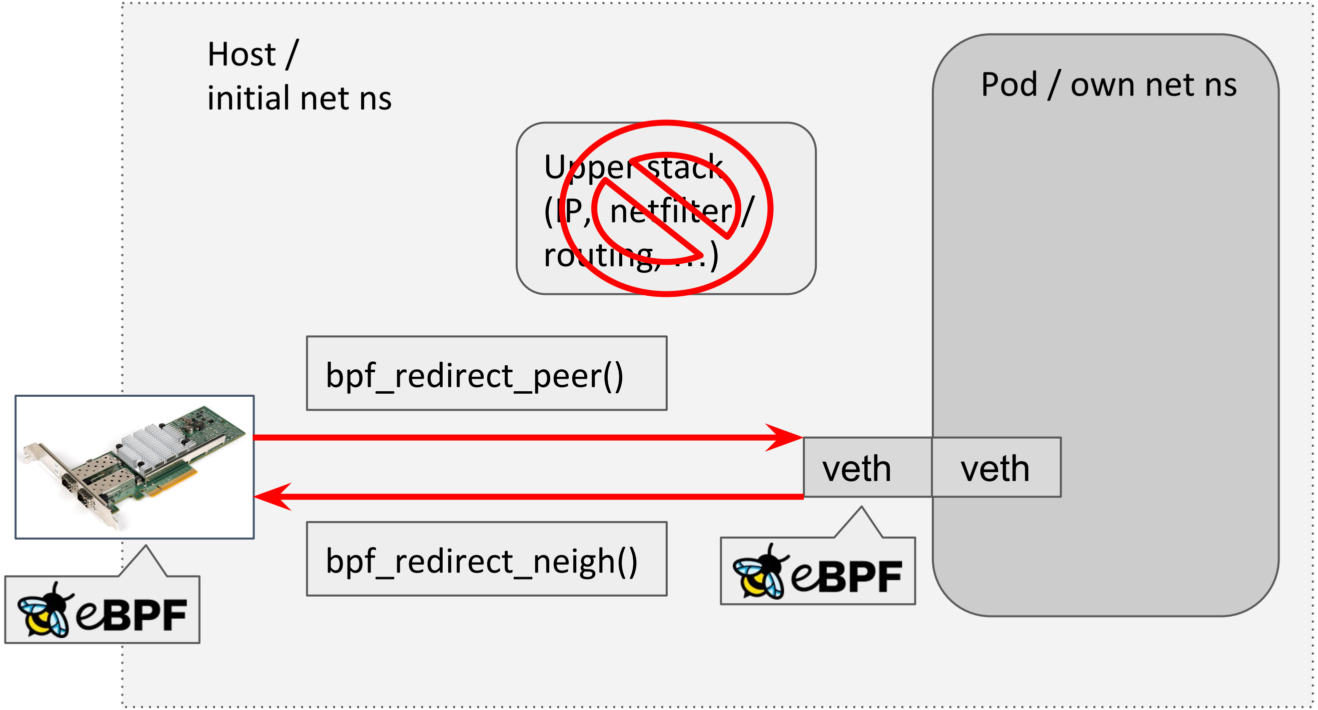


BPF program attached to cgroup v2 in this case. Enables fine-grained, scalable per Pod policies. No packet NAT for load balancing.

Improving kernel’s scalability/extensibility with BPF



Improving kernel’s scalability/extensibility with BPF

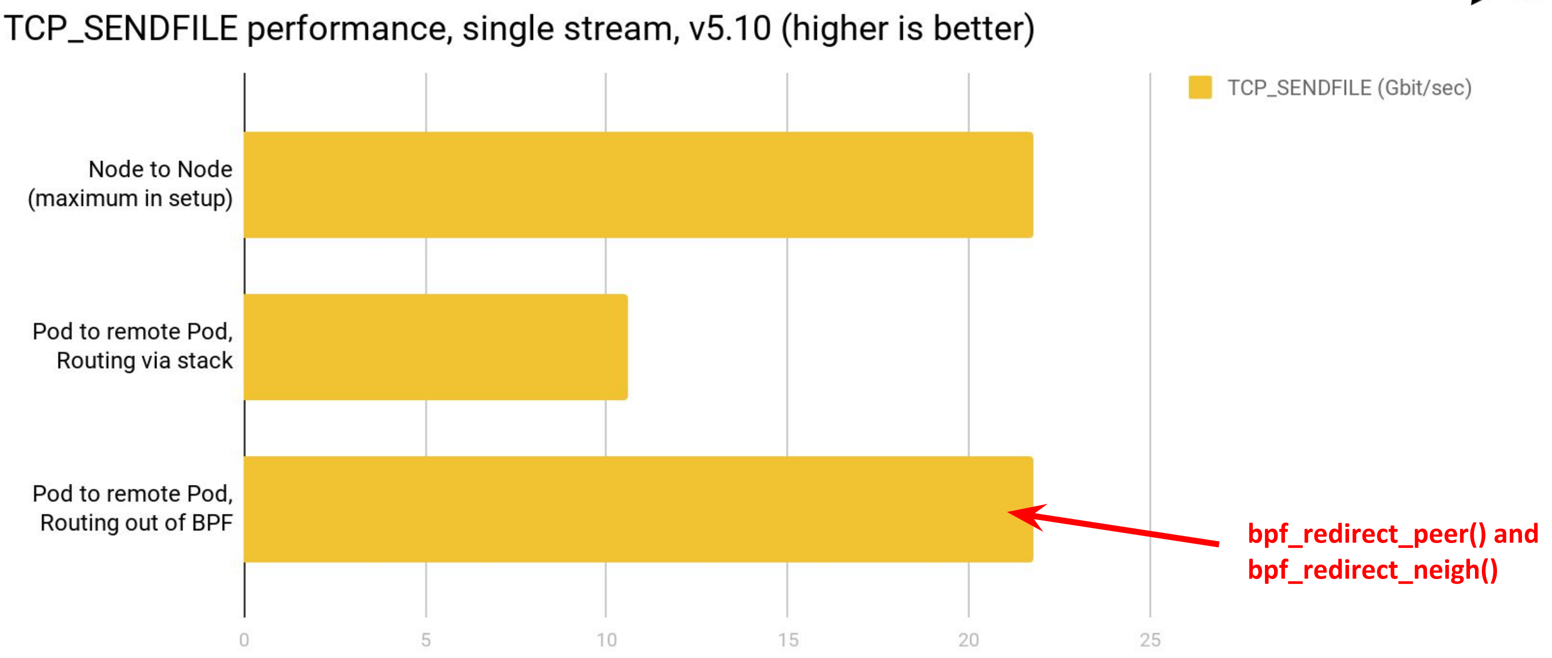


Improving kernel’s scalability/extensibility with BPF

Low-latency net ns switch into Pod from BPF. Automatic L2 resolution for traffic from Pod. Both directions now avoid host stack.

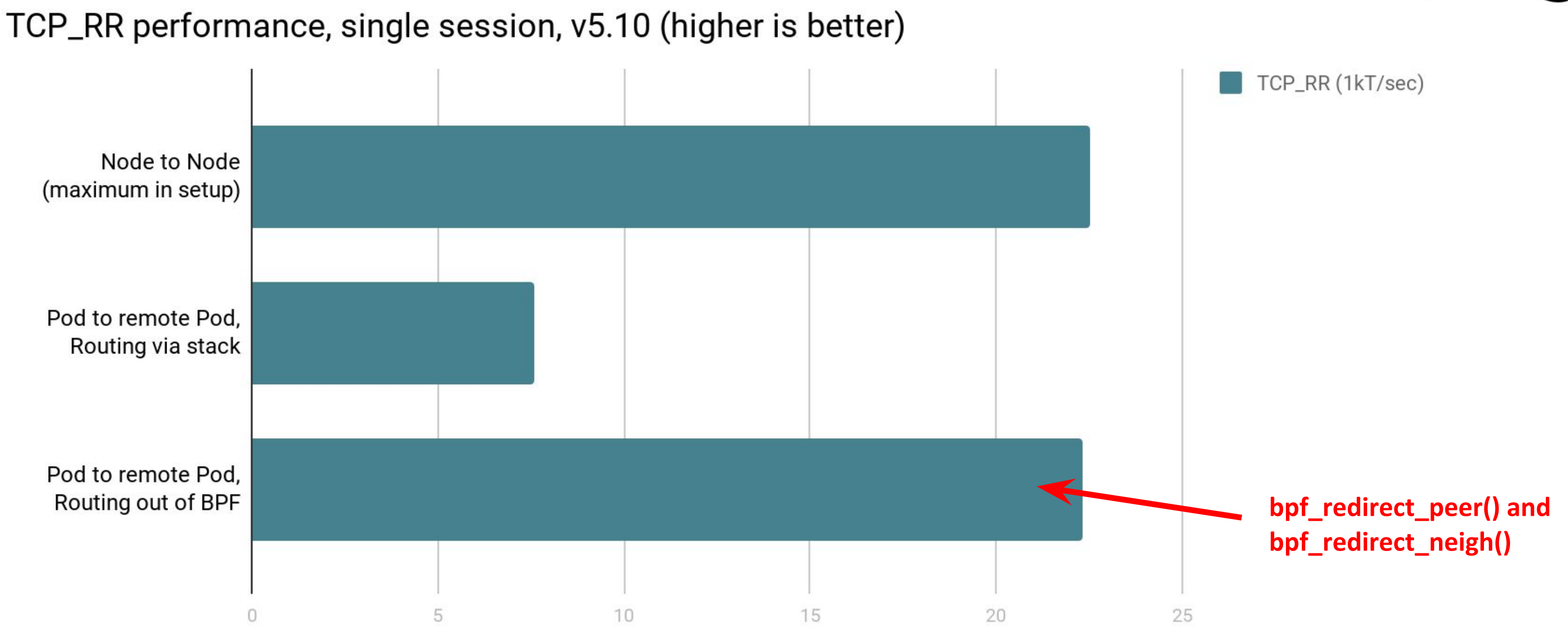
https://git.kernel.org/torvalds/c/9aa1206e8f48
https://git.kernel.org/torvalds/c/b4ab31414970

Improving kernel's scalability/extensibility with BPF



Xeon E3-1240, 3.4 GHz each (4 cores, HT off), back to back via nfp, IRQs pinned to CPUs, tuned with profile network-throughput
From Pod ns: netperf -H <remote-pod-ip> -t TCP_SENDFILE -T0,0 -P0 -s2 -l 60 -D 2 -f g

Improving kernel's scalability/extensibility with BPF



Xeon E3-1240, 3.4 GHz each (4 cores, HT off), back to back via nfp, IRQs pinned to CPUs, tuned with profile network-latency
From Pod ns: percpu_netperf <remote-pod-ip> (https://github.com/borkmann/netperf_scripts/blob/master/percpu_netperf)

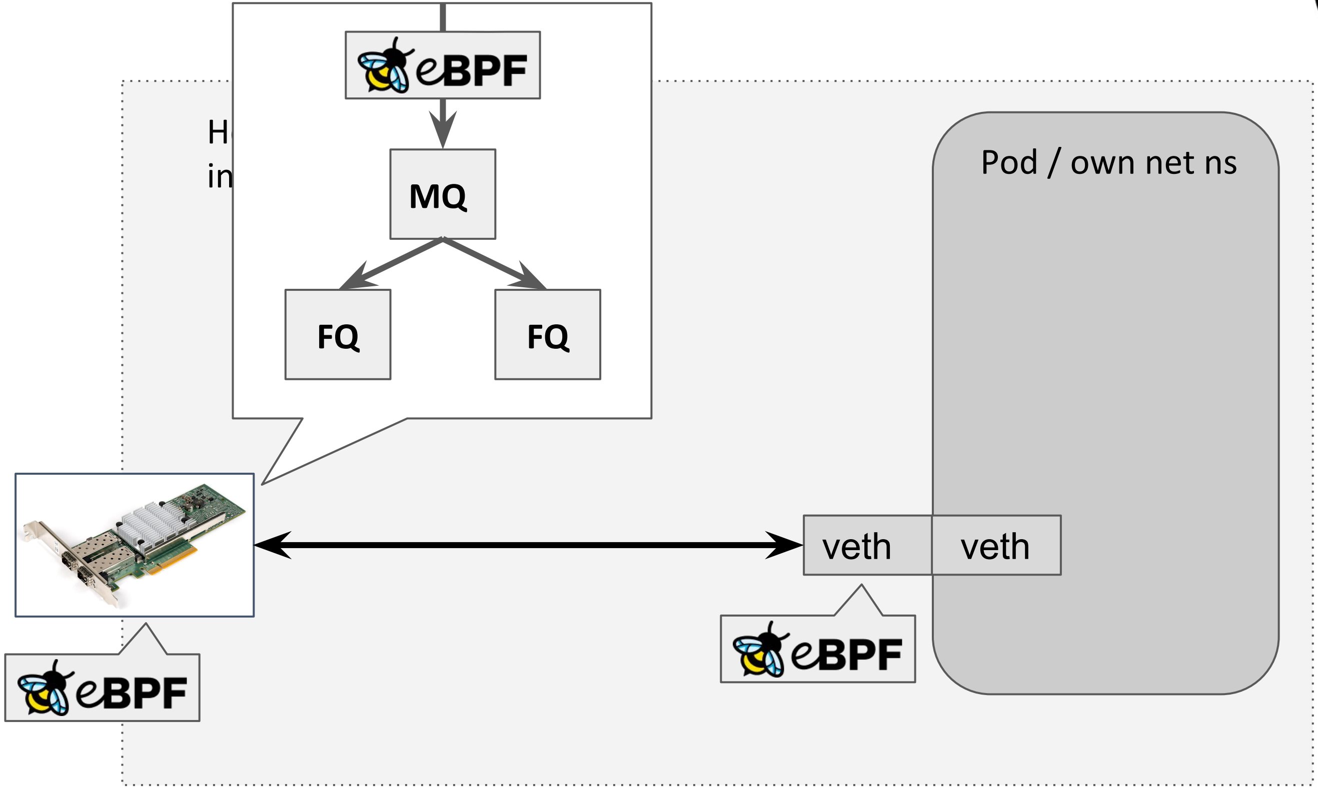
Lock-less rate-limiting on multi-queue via BPF and Earliest Departure Time (EDT).

Improving kernel's scalability/extensibility with BPF

BPF classifies network traffic to Pod and then sets packet departure time based on user defined bandwidth rate.

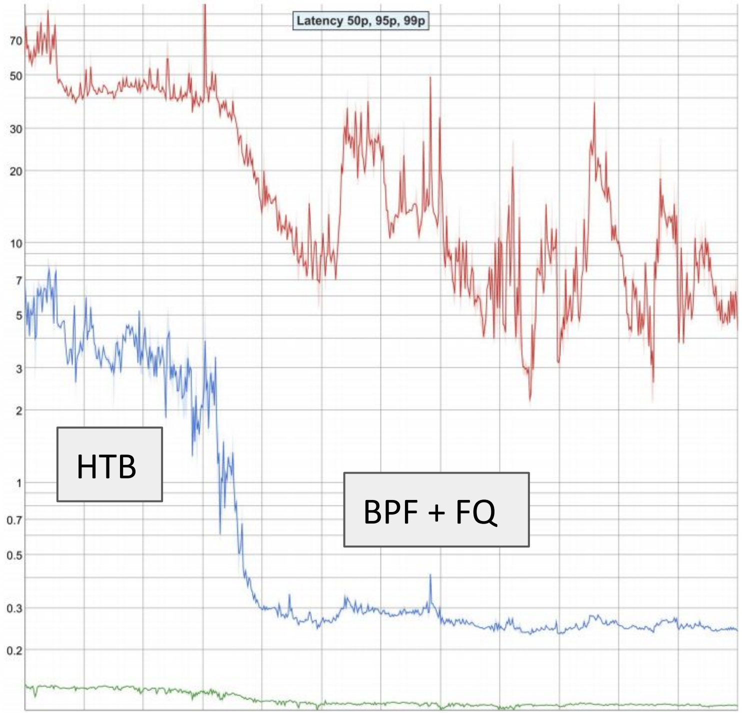
FQ qdisc schedules packet under this time constraint.

Improving kernel's scalability/extensibility with BPF



Improving kernel's scalability/extensibility with BPF

Transmission latency (lower is better)



99p ⟶ **10x latency reduction**

95p ⟶ **20x latency reduction**

50p

https://netdevconf.info/0x14/pub/papers/55/0x14-paper55-talk-paper.pdf

There are many more examples such as BPF implemented TCP congestion control, custom TCP header options, ...

https://linuxplumbersconf.org/event/7/contributions/687/attachments/537/1262/BPF_network_tcp-cc-hdr-sk-stg_LPC_2020.pdf

TCP CUBIC

```c
void BPF_STRUCT_OPS(bictcp_cong_avoid, struct sock *sk, __u32 ack, __u32 acked)
{
	struct tcp_sock *tp = tcp_sk(sk);
	struct bictcp *ca = inet_csk_ca(sk);

	if (!tcp_is_cwnd_limited(sk))
		return;

	if (tcp_in_slow_start(tp)) {
		if (hystart && after(ack, ca->end_seq))
			bictcp_hystart_reset(sk);
		acked = tcp_slow_start(tp, acked);
		if (!acked)
			return;
	}
	bictcp_update(ca, tp->snd_cwnd, acked);
	tcp_cong_avoid_ai(tp, ca->cnt, acked);
}

__u32 BPF_STRUCT_OPS(bictcp_recalc_ssthresh, struct sock *sk)
{
	const struct tcp_sock *tp = tcp_sk(sk);
	struct bictcp *ca = inet_csk_ca(sk);

	ca->epoch_start = 0;	/* end of epoch */

	/* Wmax and fast convergence */
	if (tp->snd_cwnd < ca->last_max_cwnd && fast_convergence)
		ca->last_max_cwnd = (tp->snd_cwnd * (BICTCP_BETA_SCALE + beta))
			/ (2 * BICTCP_BETA_SCALE);
	else
		ca->last_max_cwnd = tp->snd_cwnd;

	return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
}
```

```c
static void bictcp_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
	struct tcp_sock *tp = tcp_sk(sk);
	struct bictcp *ca = inet_csk_ca(sk);

	if (!tcp_is_cwnd_limited(sk))
		return;

	if (tcp_in_slow_start(tp)) {
		if (hystart && after(ack, ca->end_seq))
			bictcp_hystart_reset(sk);
		acked = tcp_slow_start(tp, acked);
		if (!acked)
			return;
	}
	bictcp_update(ca, tp->snd_cwnd, acked);
	tcp_cong_avoid_ai(tp, ca->cnt, acked);
}

static u32 bictcp_recalc_ssthresh(struct sock *sk)
{
	const struct tcp_sock *tp = tcp_sk(sk);
	struct bictcp *ca = inet_csk_ca(sk);

	ca->epoch_start = 0;	/* end of epoch */

	/* Wmax and fast convergence */
	if (tp->snd_cwnd < ca->last_max_cwnd && fast_convergence)
		ca->last_max_cwnd = (tp->snd_cwnd * (BICTCP_BETA_SCALE + beta))
			/ (2 * BICTCP_BETA_SCALE);
	else
		ca->last_max_cwnd = tp->snd_cwnd;

	return max((tp->snd_cwnd * beta) / BICTCP_BETA_SCALE, 2U);
}
```

Short turnaround time to experiment, develop and deploy changes; safety checked, stable API in contrast to kernel modules.

Allows to interoperate and extend feedback with data from other BPF programs.

And *simplifies* control plane too when BPF used in multiple subsystems. Less overall dependencies and moving parts.

Extended BPF has been around for 6 years by now, yet it feels the potential is so huge that we’re still just at the very beginning.

Future: Tiny core kernel enriched with BPF

Steven Rostedt
@srostedt

BPF will replace Linux #kr2019

11:06 am · 26 Sep 2019 · Twitter for Android

Thanks! Questions?

BPF as dataplane …

- fully programmable
- highly scalable
- safety verified
- solving production issues