Generic multi-prog API, tcx links and meta device for BPF

Daniel Borkmann (Isovalent)

LSF/MM/BPF 2023

Generic multi-attach API and tcx BPF layer

Goal

- Having a **generic reusable multi-program management API** that is fit for long-term
    - With popularity of BPF, more projects are using it in the wild
    - Therefore more users competing in case of old-style single-program attach hooks
- Being able to express dependencies between programs
- Same “look and feel“ for different attachment points

Work which led up to here

- LPC’22 proposal: **Cilium's BPF kernel datapath revamped**
- Corresponding patch set on BPF mailing list: https://lore.kernel.org/bpf/20221004231143.19190-1-daniel@iogearbox.net/

- tl;dr on patchset:
    - Rework of tc BPF (fast-path & management API) with addition of links for tc BPF
    - Attach/detach/query/link-create API via bpf() with tuple (prog/link fd + priority)
- Feedback was that i) to name the layer tcx and ii) priorities are hard to use due to collisions, can we challenge status quo?

```
That's all theory. Your cover letter example proves that in
real life different service pick the same priority.
They simply don't know any better.
prio is an unnecessary magic that apps _have_ to pick,
so they just copy-paste and everyone ends up using the same.
```

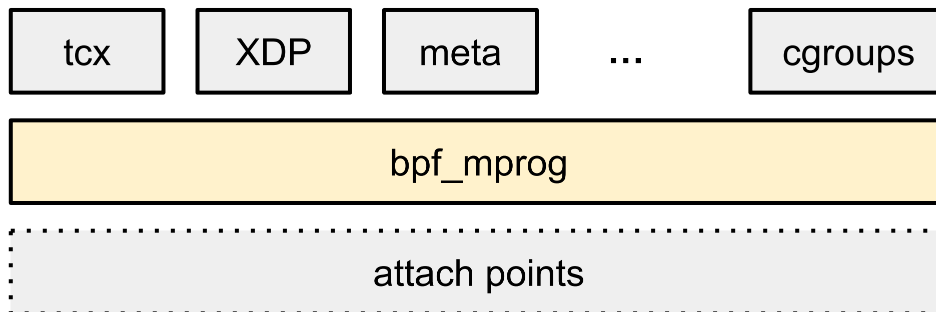
Alternative directions to express dependencies

- systemd has Before=/After= dependency directives on unit files
- BPF could have something similar which would be ideal for management daemons
  - Node-central management daemon may not be suitable for every environment (e.g. K8s), but a new API should definitely make their lives easier
- Collected requirements from Meta and Cilium side with Andrii for initial design and we converged with the following...

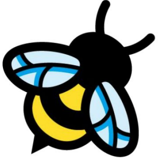
Requirements for generic multi-attach API

- Dependency directives (can also be combined):
  - BPF_F_{BEFORE,AFTER} with relative_{fd,id} which can be {prog,link}
    - BPF_F_ID flag as {fd,id} toggle
    - BPF_F_LINK flag as {prog,link} toggle
  - BPF_F_{FIRST,LAST}
- Support prog-based attach/detach and link API
- Internal revision counter and optionally being able to pass expected_revision
  - Daemon can query current state with revision, and pass it along for attachment to assert current state
- Common layer/API which deals which deals with all the details for state update
  - Must be easy to integrate/reuse

Requirements for generic multi-attach API



Examples for generic multi-attach API

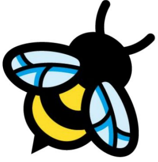


- Case: Simple append attach via link to tc BPF ingress of given ifindex

```
__u32 flags = 0, relative_obj = 0;
struct bpf_link *link;

[...]

link = bpf_program__attach_tcx(skel->progs.tc, ifindex, flags, relative_obj);
if (!link)
    goto [...];
```

```
prog section: tcx/ingress
```

Examples for generic multi-attach API

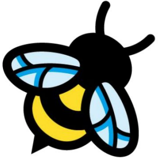


- Case: Attach before prog2 id via link to tc BPF ingress of given ifindex

```
__u32 flags = BPF_F_BEFORE | BPF_F_ID;
__u32 relative_obj = prog2_id;
struct bpf_link *link;

[...]

link = bpf_program__attach_tcx(skel->progs.tc, ifindex, flags, relative_obj);
if (!link)
     goto [...];
```

```
prog section: tcx/ingress
```

Examples for generic multi-attach API

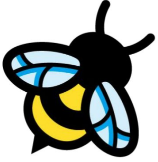


- Case: Attach before prog2 fd via link to tc BPF ingress of given ifindex and ensure it remains first

```
__u32 flags = BPF_F_FIRST | BPF_F_BEFORE;
__u32 relative_obj = prog2_fd;
struct bpf_link *link;

[...]

link = bpf_program__attach_tcx(skel->progs.tc, ifindex, flags, relative_obj);
if (!link)
    goto [...];
```

`prog section: tcx/ingress`

Examples for generic multi-attach API

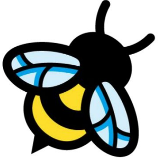


- Case: Attach before link1 id via link to tc BPF ingress of given ifindex and ensure it remains first

```
__u32 flags = BPF_F_FIRST | BPF_F_BEFORE | BPF_F_ID | BPF_F_LINK;
__u32 relative_obj = link1_id;
struct bpf_link *link;

[...]

link = bpf_program__attach_tcx(skel->progs.tc, ifindex, flags, relative_obj);
if (!link)
     goto [...];
```

```
prog section: tcx/ingress
```

Examples for generic multi-attach API

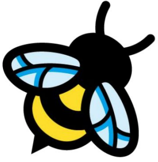


- Case: Attach after link1 fd via link to tc BPF ingress of given ifindex

```
__u32 flags = BPF_F_AFTER | BPF_F_LINK;
__u32 relative_obj = link1_fd;
struct bpf_link *link;

[...]

link = bpf_program__attach_tcx(skel->progs.tc, ifindex, flags, relative_obj);
if (!link)
     goto [...];
```

```
prog section: tcx/ingress
```

Examples for generic multi-attach API

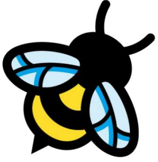


- Case: Attach via link to tc BPF ingress of given ifindex and ensure it remains first and last

```
__u32 flags = BPF_F_FIRST | BPF_F_LAST;
__u32 relative_obj = 0;
struct bpf_link *link;

[...]

link = bpf_program__attach_tcx(skel->progs.tc, ifindex, flags, relative_obj);
if (!link)
     goto [...];
```

`prog section: tcx/ingress`

Examples for generic multi-attach API

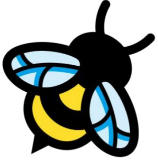


- Case: Attach after prog1 id via link to tc BPF ingress of given ifindex and ensure it remains last

```
__u32 flags = BPF_F_LAST | BPF_F_AFTER | BPF_F_ID;
__u32 relative_obj = prog1_id;
struct bpf_link *link;

[...]

link = bpf_program__attach_tcx(skel->progs.tc, ifindex, flags, relative_obj);
if (!link)
    goto [...];
```

```
prog section: tcx/ingress
```

Examples for generic multi-attach API

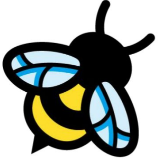


- Case: Attach via link to tc BPF ingress of given ifindex and ensure it remains last, bail out if internal revision is not 42

```
__u32 flags = BPF_F_LAST;
__u32 relative_obj = 0;
__u32 revision = 42;
struct bpf_link *link;

[...]

link = bpf_program__attach_tcx_revision(skel->progs.tc, ifindex, flags,
                                        relative_obj, revision);
if (!link)
     goto [...];
```

prog section: tcx/ingress

Generic multi-attach API: Query UAPI



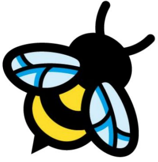
```
        struct { /* anonymous struct used by BPF_PROG_QUERY command */
-               __u32           target_fd;      /* container object to query */
+               union {
+                       __u32   target_fd;      /* target object to query or ... */
+                       __u32   target_ifindex; /* target ifindex */
+               };
                __u32           attach_type;
                __u32           query_flags;
                __u32           attach_flags;
                __aligned_u64   prog_ids;
-               __u32           prog_cnt;
+               union {
+                       __u32   prog_cnt;
+                       __u32   count;
+               };
+               __u32           revision;
                /* output: per-program attach_flags.
                 * not allowed to be set during effective query.
                 */
                __aligned_u64   prog_attach_flags;
+               __aligned_u64   link_ids;
+               __aligned_u64   link_attach_flags;
        } query;
```

Example:

target_ifindex = 1
attach_type = bpf_tcx_{ingress,egress}
revision = 12
count = 4

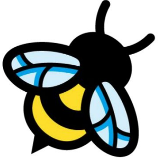
prog_ids =	1	5	2	3
link_ids =		8		1
prog_attach_flags =	BPF_F_ FIRST			
link_attach_flags =				BPF_F_ LAST

Generic multi-attach API: UAPI flag extensions



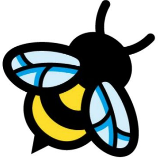
```
  #define BPF_F_ALLOW_OVERRIDE    (1U << 0)
  #define BPF_F_ALLOW_MULTI       (1U << 1)
+ /* Generic attachment flags. */
  #define BPF_F_REPLACE           (1U << 2)
+ #define BPF_F_BEFORE            (1U << 3)
+ #define BPF_F_AFTER             (1U << 4)
+ #define BPF_F_FIRST             (1U << 5)
+ #define BPF_F_LAST              (1U << 6)
+ #define BPF_F_ID                (1U << 7)
+ #define BPF_F_LINK              BPF_F_LINK /* 1 << 13 */
```

Generic multi-attach API: Attach / Detach UAPI



```diff
 struct { /* anonymous struct used by BPF_PROG_ATTACH/DETACH commands */
-        __u32           target_fd;      /* container object to attach to */
-        __u32           attach_bpf_fd;  /* eBPF program to attach */
+        union {
+                __u32   target_fd;      /* target object to attach to or ... */
+                __u32   target_ifindex; /* target ifindex */
+        };
+        __u32           attach_bpf_fd;
         __u32           attach_type;
         __u32           attach_flags;
-        __u32           replace_bpf_fd; /* previously attached eBPF
-                                         * program to replace if
-                                         * BPF_F_REPLACE is used
-                                         */
+        union {
+                __u32   relative_fd;
+                __u32   relative_id;
+                __u32   replace_bpf_fd;
+        };
+        __u32           expected_revision;
 };
```

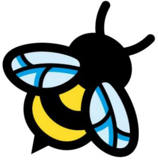
Generic multi-attach API: Link Create UAPI



- target_ifindex etc already there, but given we cannot add common fields for link creation at the end, we need to move these into link-specific section (here: .tcx):

```
+                       struct {
+                               union {
+                                       __u32   relative_fd;
+                                       __u32   relative_id;
+                               };
+                               __u32           expected_revision;
+                       } tcx;
```

Generic multi-attach API: Internals



```
struct bpf_prog_item {
        struct bpf_prog *prog;
        u32 flags;
        u32 id;
};

struct bpf_mprog_entry {
        struct bpf_prog_item            items[BPF_MPROG_MAX] ____cacheline_aligned;
        struct bpf_mprog_entry_pair     *parent;
};

struct bpf_mprog_entry_pair {
        struct bpf_mprog_entry          a;
        struct bpf_mprog_entry          b;
        struct rcu_head                 rcu;
        struct bpf_prog *               ref;
        atomic_t                        revision;
};
```

minimal for better cacheline fit

array for cache locality

a/b swap, so detach does not fail

tcx (aka “tc express” for BPF)

- tc BPF will be first consumer of this API
    - See LPC’22 talk on the datapath revamp: **Cilium's BPF kernel datapath revamped**
    - For tc link use case this needs to be outside of qdisc but as part of tc layer
    - Cooperative with classic tc BPF for successive migration
    - New future-proof tc fast-path aka tcx (“tc express”)
    - Given cache-friendly array and minimal indirections it cuts cycles for entering into BPF program in half

tcx (aka “tc express” for BPF)

```
static __always_inline enum tc_action_base
tcx_run(const struct bpf_mprog_entry *entry, struct sk_buff *skb,
        const bool needs_mac)
{
        const struct bpf_prog_item *item;
        const struct bpf_prog *prog;
        int ret = TC_NEXT;

        if (needs_mac)
                __skb_push(skb, skb->mac_len);
        item = &entry->items[0];
        while ((prog = READ_ONCE(item->prog))) {
                bpf_compute_data_pointers(skb);
                ret = bpf_prog_run(prog, skb);
                if (ret != TC_NEXT)
                        break;
                item++;
        }
        if (needs_mac)
                __skb_pull(skb, skb->mac_len);
        return tcx_action_code(skb, ret);
}
```

tcx (aka “tc express” for BPF)

```
static __always_inline struct sk_buff *
sch_handle_ingress(struct sk_buff *skb, struct packet_type **pt_prev, int *ret,
                   struct net_device *orig_dev, bool *another)
{
        struct bpf_mprog_entry *entry = rcu_dereference_bh(skb->dev->tcx_ingress);
        int sch_ret;

        if (!entry)
                return skb;
        if (*pt_prev) {
                *ret = deliver_skb(skb, *pt_prev, orig_dev);
                *pt_prev = NULL;
        }

        qdisc_skb_cb(skb)->pkt_len = skb->len;
        tcx_set_ingress(skb, true);

        if (static_branch_unlikely(&tcx_needed_key)) {
                sch_ret = tcx_run(entry, skb, true);
                if (sch_ret != TC_ACT_UNSPEC)
                        goto ingress_verdict;
        }
        sch_ret = tc_run(container_of(entry->parent, struct tcx_entry, pair), skb);
ingress_verdict:
        switch (sch_ret) {
        case TC_ACT_REDIRECT:
                /* skb_mac_header check was done by BPF, so we can safely
                 * push the L2 header back before redirecting to another
```

meta device for BPF

Goal

- Achieve **same performance** for application inside K8s Pod (netns) compared to application residing inside host namespace

- Just because we move them into netns should not incur performance penalty, but it currently still does

veth + stack vs BPF host routing case, results:

TCP stream single flow Pod to Pod over wire, 8k MTU (higher is better)

Mbps	veth + upper stack forwarding	veth + BPF host routing	host (baseline/best case)
Throughput	63,429	89,464	98,541
Cost	101 usec/MB	92 usec/MB	85 usec/MB

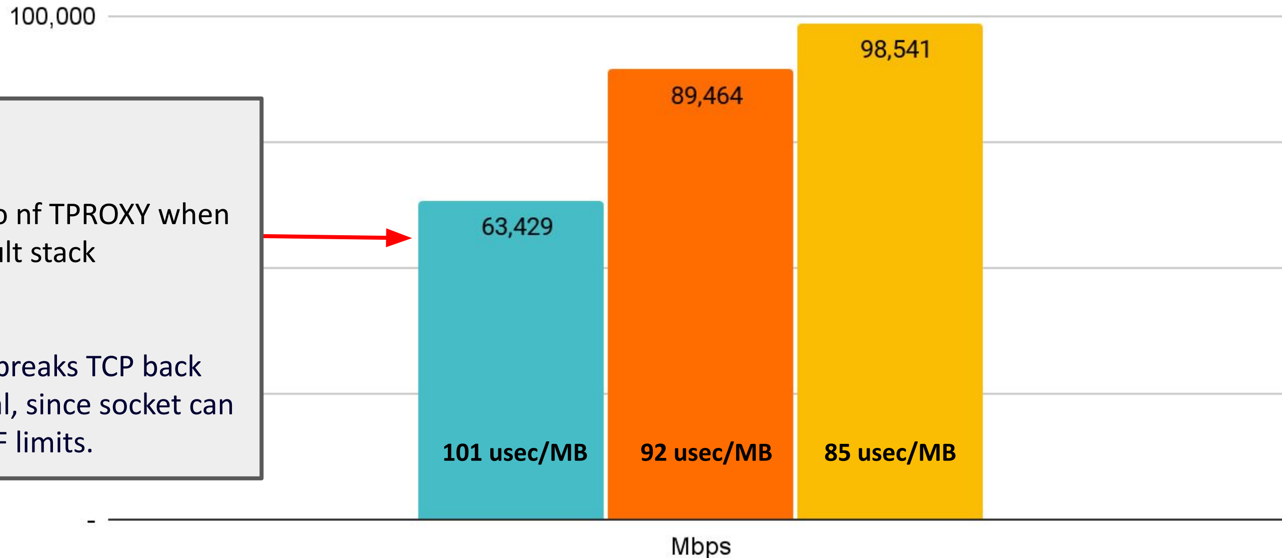
* 8264 MTU for data page alignment in GRO

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU

Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

veth + stack vs BPF host routing case, results:

TCP stream single flow Pod to Pod over wire, 8k MTU (higher is better)



Upper Stack:

skb_orphan due to nf TPROXY when packet takes default stack forwarding path.

Doing it too soon breaks TCP back pressure in general, since socket can evade SO_SNDBUF limits.

* 8264 MTU for data page alignment in GRO

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU

Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

veth + stack vs BPF host routing case, results:

TCP stream single flow Pod to Pod over wire, 8k MTU (higher is better)

	veth + upper stack forwarding	veth + BPF host routing	host (baseline/best case)
Mbps	63,429	89,464	98,541
	101 usec/MB	92 usec/MB	85 usec/MB

Cilium with BPF host routing:

Combining BPF fib lookup with redirect_peer and redirect_neigh helpers.

* 8264 MTU for data page alignment in GRO

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU

Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

veth + stack vs BPF host routing case, results:

TCP stream single flow Pod to Pod over wire, 8k MTU (higher is better)

* 8264 MTU for data page alignment in GRO

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU

Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

meta netdevices 1/2:

- “meta” for lack of better and short device type name :)
  - Derives from the Greek μετά, encompassing a wide array of meanings such as "on top of", "beyond". Given business logic is defined by BPF, this device can have many meanings.

- **Core Idea:**
  - BPF shifted from tc into the device driver, so business logic is part of device xmit itself
  - Performance: no per-CPU backlog when BPF redirects traffic from Pod to outside the node

- What about XDP support for meta virtual device?
  - No, just use veth - side node on complexity: ¾ of veth code is just for XDP today
  - For local Pod traffic batching right after native XDP@phys device with GRO’ed skb is preferred

- Program management: reuse of BPF multi-prog attach API (bpf_mprog)

meta netdevices 2/2:

- main/peer device: only main device can control BPF program management (typically leg in host)
    - Later step: option to configure as single device as well (e.g. collect_md with encapsulation/encryption via logic implemented in BPF)

- L3 mode (noarp) by default, L2 mode configurable (useful for testing IPv6 ND/ARP/LLDP/VRRP)
- Configurable traffic blackholing for main/peer dev if no BPF attached to avoid leaking traffic
- Maximum tc BPF compatibility to ease migration from tc+veth into meta device

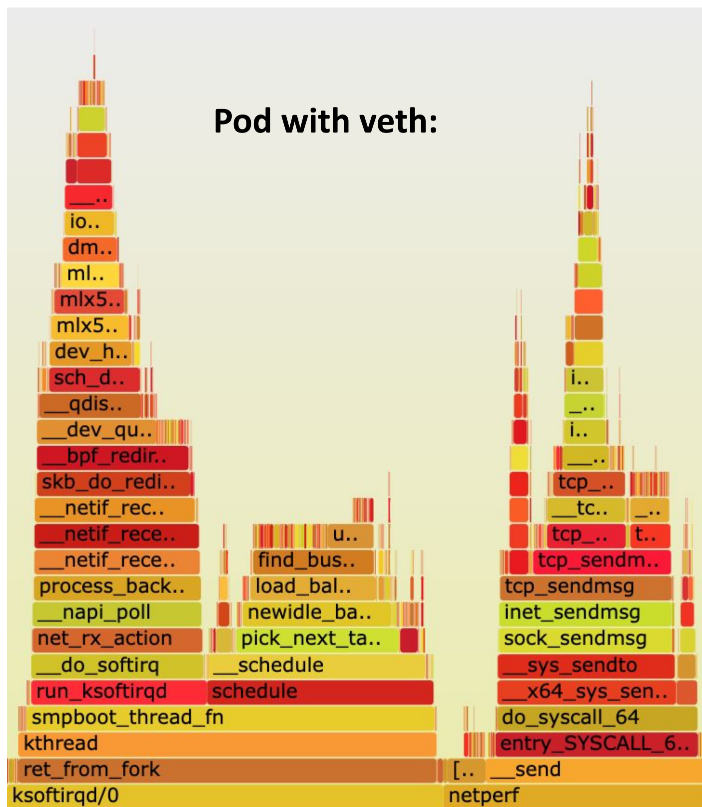
```c
static netdev_tx_t meta_xmit(struct sk_buff *skb, struct net_device *dev)
{
	struct meta *meta = netdev_priv(dev);
	struct net_device *peer;
	struct bpf_prog *prog;

	rcu_read_lock();
	peer = rcu_dereference(meta->peer);
	if (unlikely(!peer || skb_orphan_frags(skb, GFP_ATOMIC)))
		goto drop;

	meta_scrub_minimum(skb);
	skb->dev = peer;

	prog = rcu_dereference(meta->prog);
	if (unlikely(!prog))
		goto drop;
	switch (bpf_prog_run(prog, skb)) {
	case META_OKAY:
		skb->protocol = eth_type_trans(skb, skb->dev);
		skb_postpull_rcsum(skb, eth_hdr(skb), ETH_HLEN);
		__netif_rx(skb);
		break;
	case META_REDIRECT:
		skb_do_redirect(skb);
		break;
	case META_DROP:
	default:
drop:
		kfree_skb(skb);
		break;
	}
	rcu_read_unlock();
	return NETDEV_TX_OK;
}
```

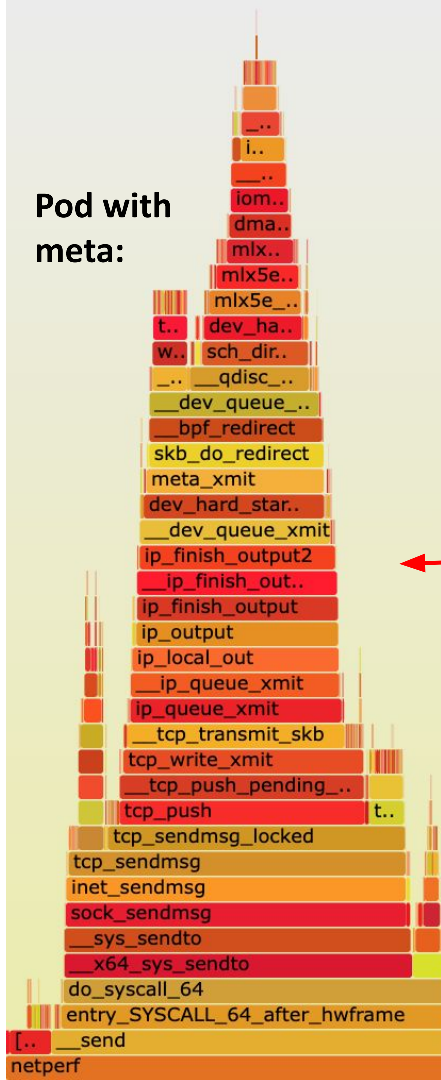
(switch netns to host)

(BPF program: policy, fib lookup, redirect, etc)

(directly redirect to phys dev, no backlog queue)

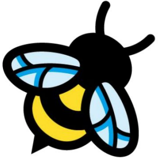
veth vs meta: backlog queue





Remains in process context all the way.

meta + BPF host routing case, results:



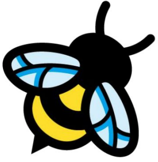
TCP stream single flow Pod to Pod over wire, 8k MTU (higher is better)

Mbps	veth + upper stack forwarding	veth + BPF host routing	meta + BPF host routing	host (baseline/best case)
Throughput	63,429	89,464	98,615	98,541
usec/MB	101 usec/MB	92 usec/MB	85 usec/MB	85 usec/MB

tput as high as host

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU
Receiver: taskset -a -c <core> tcp_mmap -s (non-zerocopy mode), Sender: taskset -a -c <core> tcp_mmap -H <dst host>

meta + BPF host routing case, results:



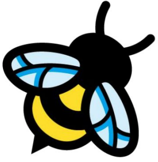
Latency in usec Pod to Pod over wire (lower is better)

	veth + BPF host routing	meta + BPF host routing	host (baseline/best case)
MIN	17	15	15
P90	21	19	19
P99	23	20	20

latency as low as host

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off
netperf -t TCP_RR -H <remote pod> -- -O MIN_LATENCY,P90_LATENCY,P99_LATENCY,THROUGHPUT

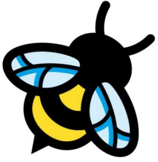
meta inside Cilium, architecture:



Building blocks in Cilium
for optimal Pod performance:

Bandwidth Manager
(fq/EDT/BBR) **+**
BPF Host Routing
(FIB lookup & redirect_peer/neigh) **+**
meta netdevices +
BIG TCP (IPv4/IPv6)

BIG TCP + veth vs meta, results:



Latency in usec Pod to Pod over wire (lower is better)

	veth + BPF host routing	meta + BPF host routing	host (baseline/best case)
MIN	15	14	14
P50	34	26	26
P90	56	42	42
P99	89	69	69

latency as low as host

(side note: BIG TCP + upper stack forwarding currently broken)

Back to back: AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0, ConnectX-6 Dx, mlx5 driver, striding mode, LRO off, 8264 MTU
netperf -t TCP_RR -H <remote pod> -- -O MIN_LATENCY,P90_LATENCY,P99_LATENCY,THROUGHPUT

meta netdevices, open questions:

- meta ships as module whereas BPF multi-prog attach API is built-in and has no dynamic registration right now. Options:
    - A: make meta Kconfig def_bool BPF_SYSCALL, bit similar to netfilter BPF
    - B: Expose bpf_mprog API to modules, and for meta make callbacks registerable
        - Ideally all logic can reside in the driver itself
        - Potentially ndo device callbacks to delegate

Next steps:

- Generic multi-attach API & tcx (wip code on Github)
    - Currently completing big test case batch to cover all corner cases
    - After that ready to submit to the list (planned right after conf)
    - Landing this is prereq for meta device as well
- meta netdevices (wip code on Github)
    - Implementation with multi-prog management API
    - BPF selftests, planned to land within May, max June
- XDP multi-attach support
    - Planning to take on next for native XDP after all of above lands