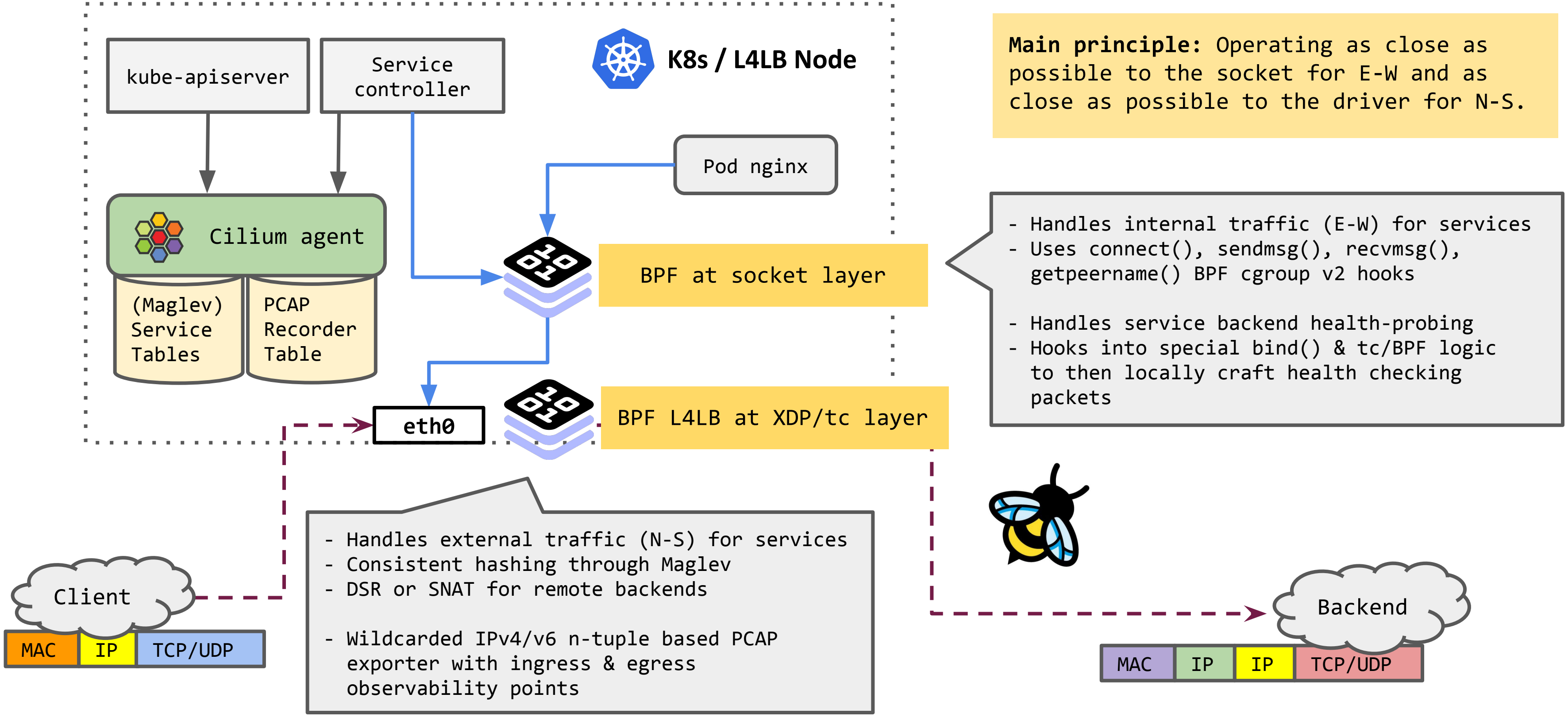
LINUX PLUMBERS CONFERENCE September 20-24, 2021

BPF datapath extensions for K8s workloads

Daniel Borkmann & Martynas Pumputis, Cilium.io

Cilium’s Load Balancer in one picture

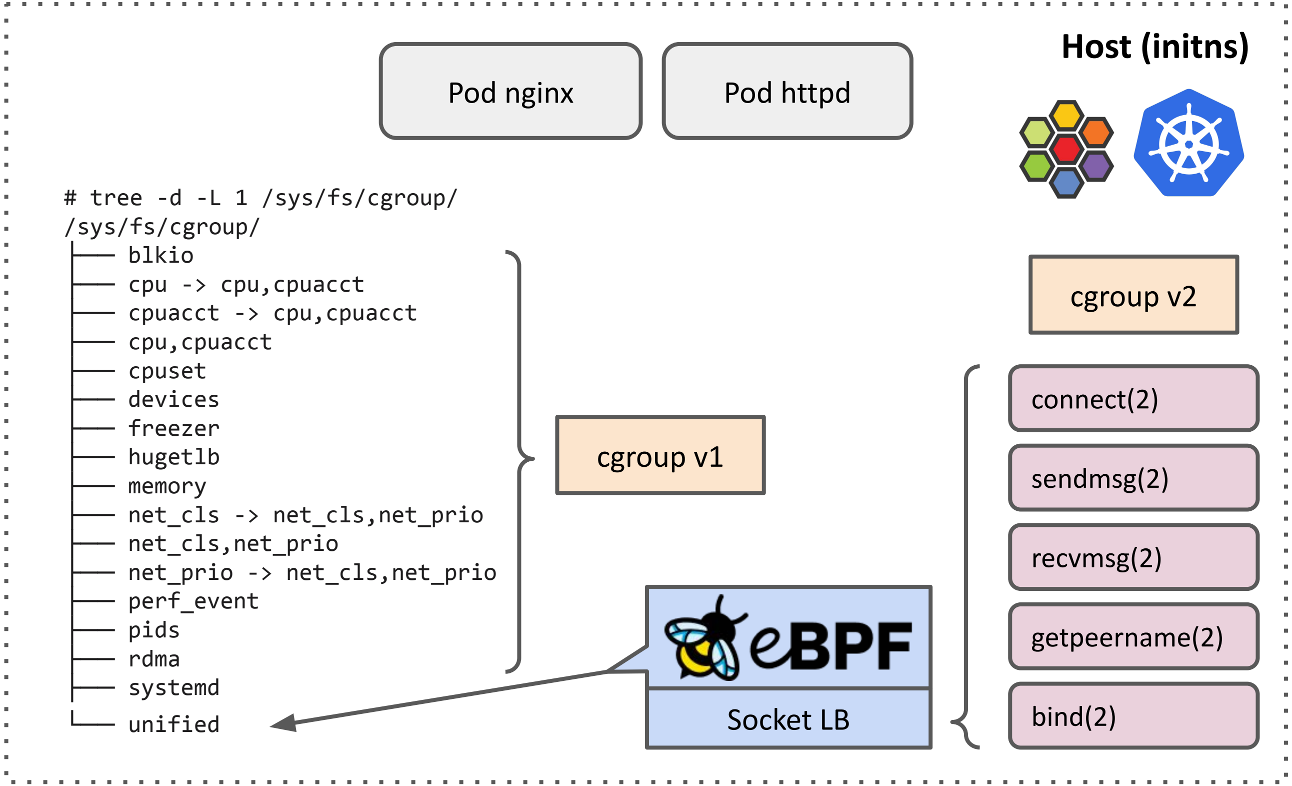


Agenda: Ongoing development items

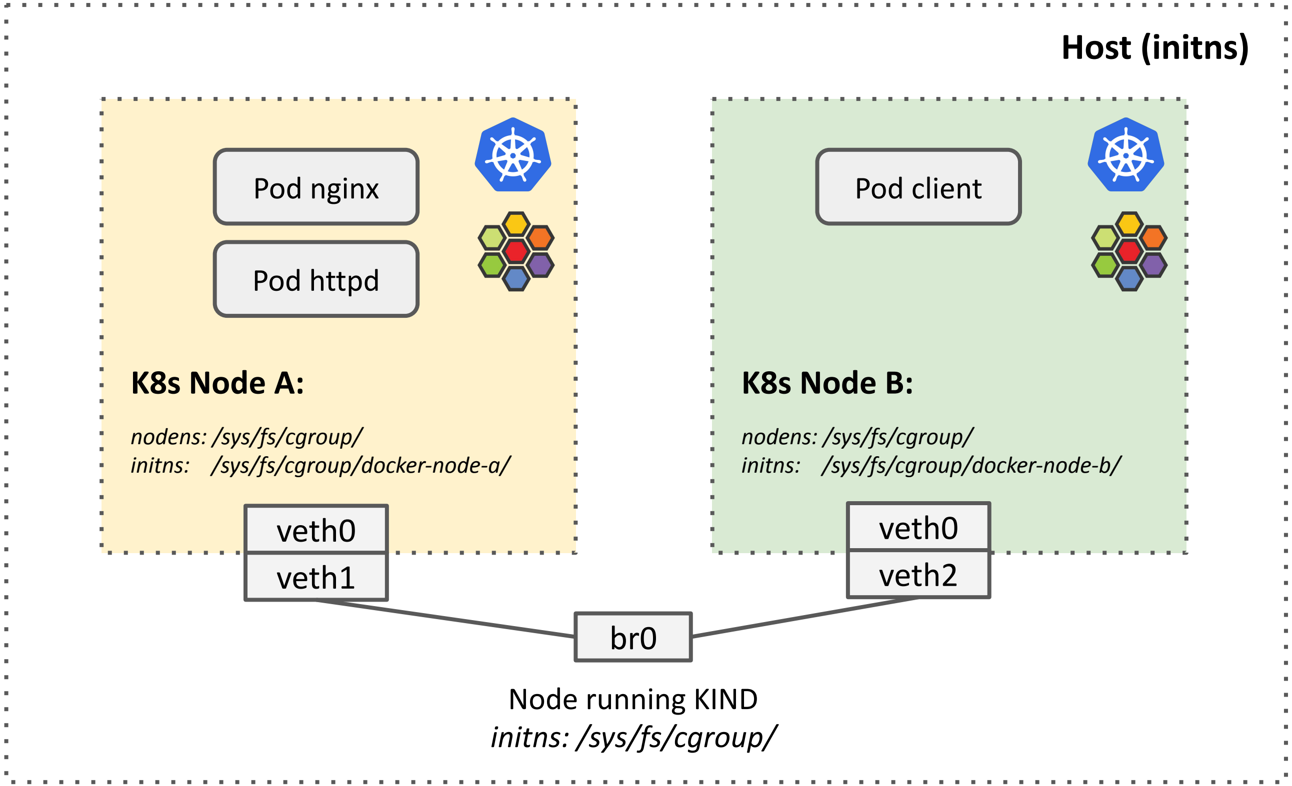
- ➔ Part 1: The cgroup v1/v2 interference problem
- ➔ Part 2: TCP pacing for Pods from initns
- ➔ Part 3: Managed neighbor entries and fib extensions
- ➔ Part 4: Wildcarded BPF map lookups

Part 1: The cgroup v1/v2 interference

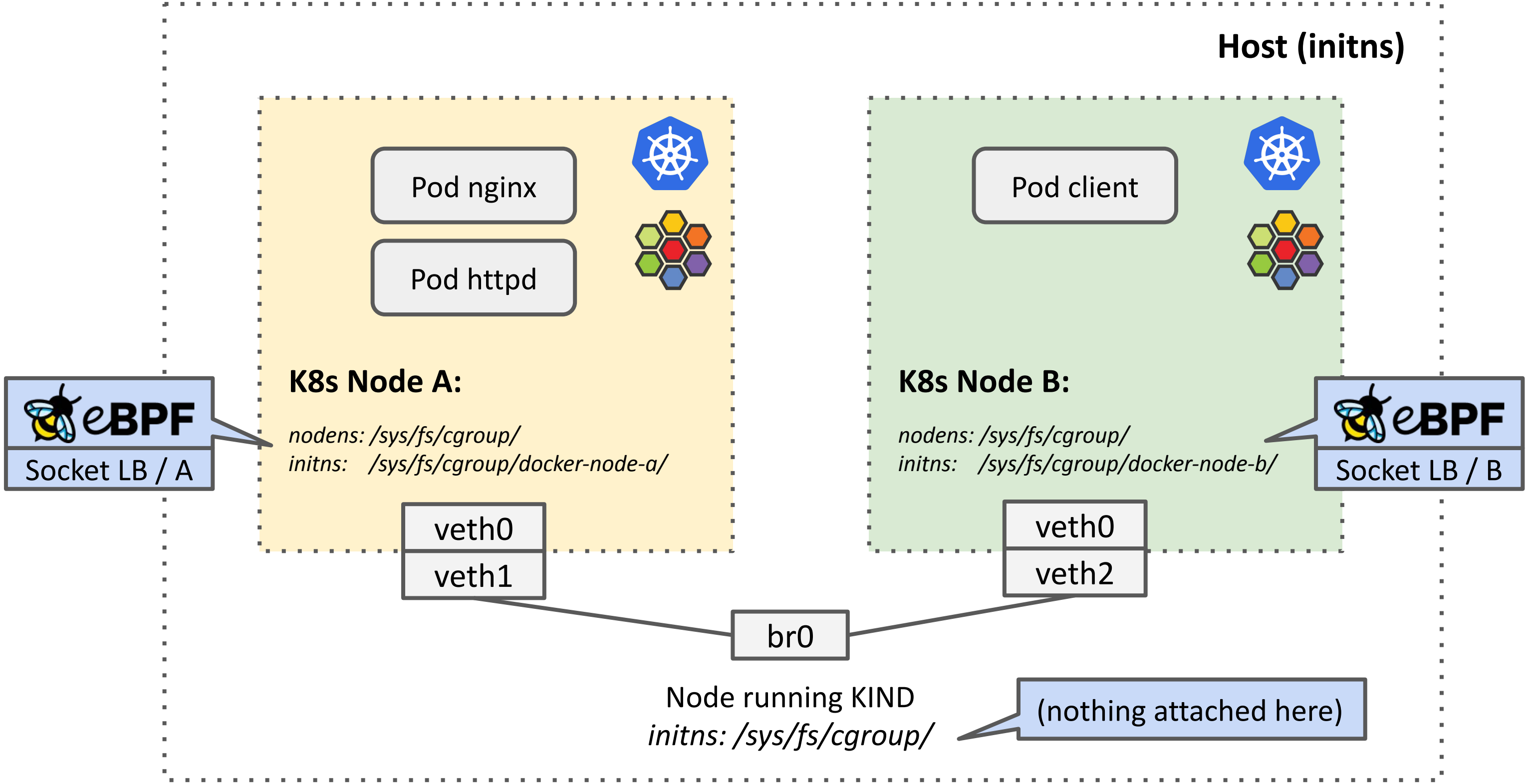
Cgroup v2 layout on (bare metal) K8s node



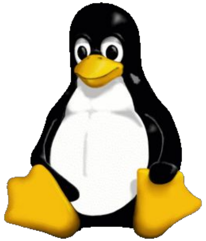
Cgroup v2 layout on KIND (K8s in Docker)



Cgroup v2 layout on KIND (K8s in Docker)



Cgroup v1/v2 interference: Context



The case for saving 8 byte in the socket structure

- ➔ Assumption back in 2015: "no reason to mix cgroup v1/v2"
  - struct sock_cgroup_data is a union with v1/v2 data
  - cgroup v1 net_cls/net_prio tags vs cgroup v2 pointer
- ➔ Reality check: Environments today have both flavors mounted

Cgroup v1/v2 interference: Context

Retrieving socket's cgroup v2 pointer in fast-path:

```
static inline struct cgroup *sock_cgroup_ptr(struct sock_cgroup_data *skcd)
{
#if defined(CONFIG_CGROUP_NET_PRIO) || defined(CONFIG_CGROUP_NET_CLASSID)
        unsigned long v;

        /*
         * @skcd->val is 64bit but the following is safe on 32bit too as we
         * just need the lower ulong to be written and read atomically.
         */
        v = READ_ONCE(skcd->val);

        if (v & 3)
                return &cgrp_dfl_root.cgrp;

        return (struct cgroup *)(unsigned long)v ?: &cgrp_dfl_root.cgrp;
#else
        return (struct cgroup *)(unsigned long)skcd->val;
#endif
}
```

Cgroup v1/v2 interference: Context

Retrieving socket's cgroup v2 pointer in fast-path:

```
static inline struct cgroup *sock_cgroup_ptr(struct sock_cgroup_data *skcd)
{
#if defined(CONFIG_CGROUP_NET_PRIO) || defined(CONFIG_CGROUP_NET_CLASSID)
        unsigned long v;

        /*
         * @skcd->val is 64bit but the following is safe on 32bit too as we
         * just need the lower ulong to be written and read atomically.
         */
        v = READ_ONCE(skcd->val);

        if (v & 3)
                return &cgrp_dfl_root.cgrp;

        return (struct cgroup *)(unsigned long)v ?: &cgrp_dfl_root.cgrp;
#else
        return (struct cgroup *)(unsigned long)skcd->val;
#endif
}
```

If cgroup v1 tagging is used on the socket, fallback to cgroup v2 root.

Cgroup v1/v2 interference: Context

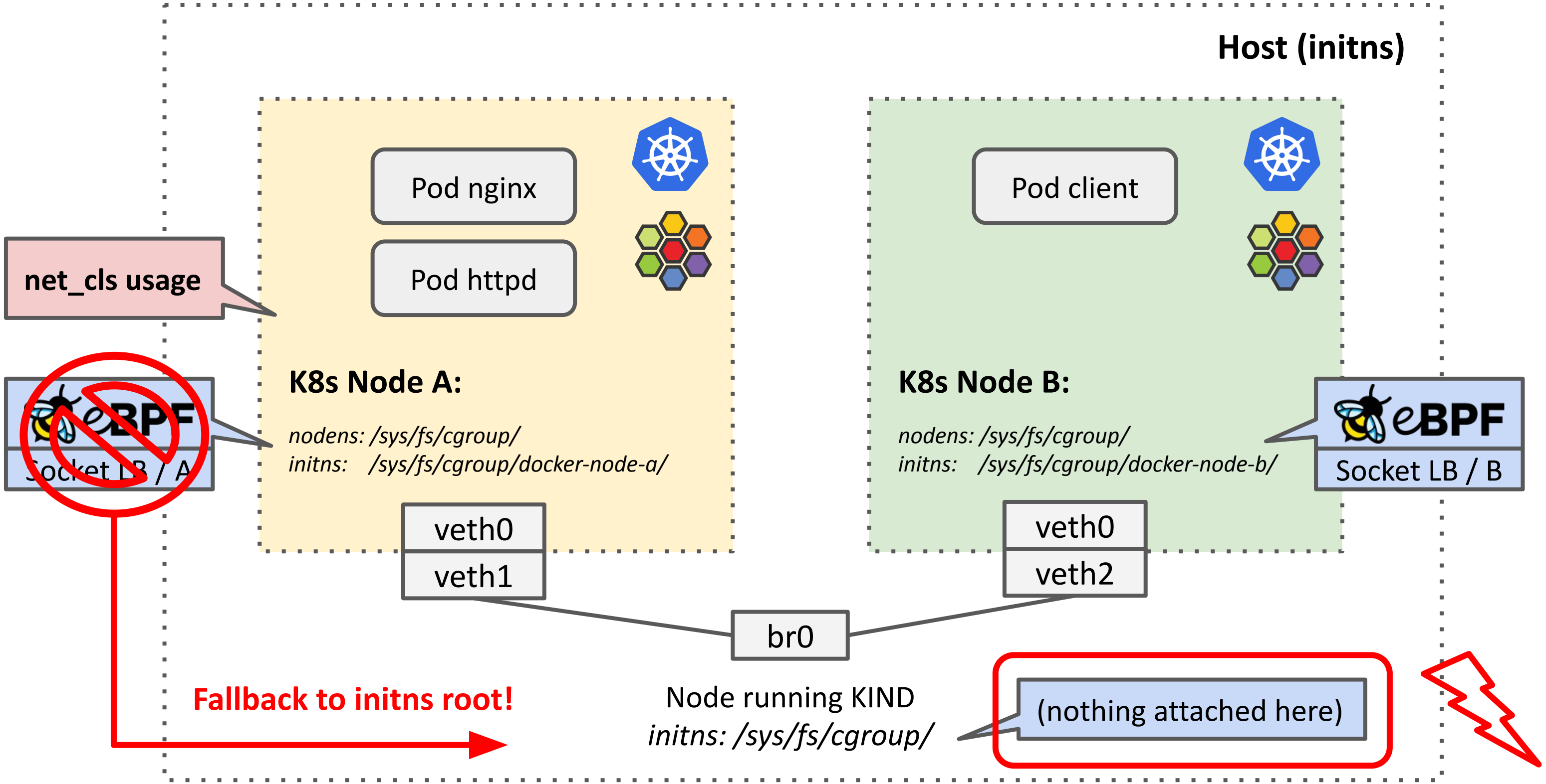
Retrieving socket's cgroup v2 pointer in fast-path:

```c
static inline struct cgroup *sock_cgroup_ptr(struct sock_cgroup_data *skcd)
{
#if defined(CONFIG_CGROUP_NET_PRIO) || defined(CONFIG_CGROUP_NET_CLASSID)
	unsigned long v;

	/*
	 * @skcd->val is 64bit but the following is safe on 32bit too as we
	 * just need the lower ulong to be written and read atomically.
	 */
	v = READ_ONCE(skcd->val);

	if (v & 3)
		return &cgrp_dfl_root.cgrp;

	return (struct cgroup *)(unsigned long)v ?: &cgrp_dfl_root.cgrp;
#else
	return (struct cgroup *)(unsigned long)skcd->val;
#endif
}
```

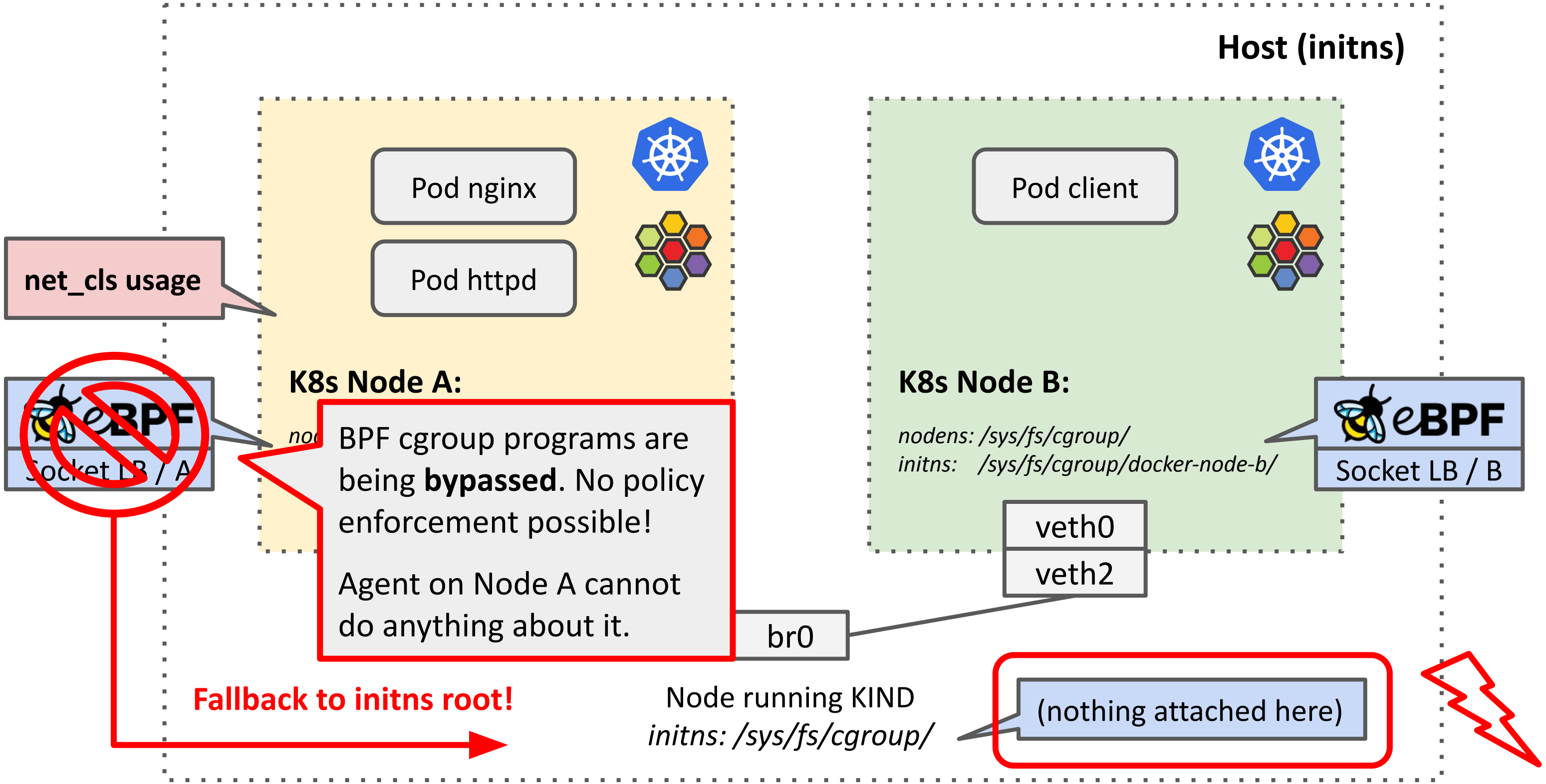
If cgroup v1 tagging is used on the socket, fallback to cgroup v2 root.

Problematic for today's environments!

Cgroup v2 layout on KIND (K8s in Docker)



Cgroup v2 layout on KIND (K8s in Docker)



Cgroup v1/v2 interference: Recap

v2 cgroup management complex and cumbersome

- ➔ Incompatible to cgroup namespaces or non-root cgroup paths
- ➔ v2-to-v1 switch on the socket leaks v2 object references
- ➔ Unreliable v2 invocation hinders adoption of BPF cgroup programs
  - Independent 3rd party agents inevitably step on each other
  - Distros usually enable everything for max compatibility

Approach to fixing the cgroup v1/v2 interference

Fix: biting the bullet and detangle the two ...

```
static inline struct cgroup *sock_cgroup_ptr(struct sock_cgroup_data *skcd)
{
        return skcd->cgroup;
}
```

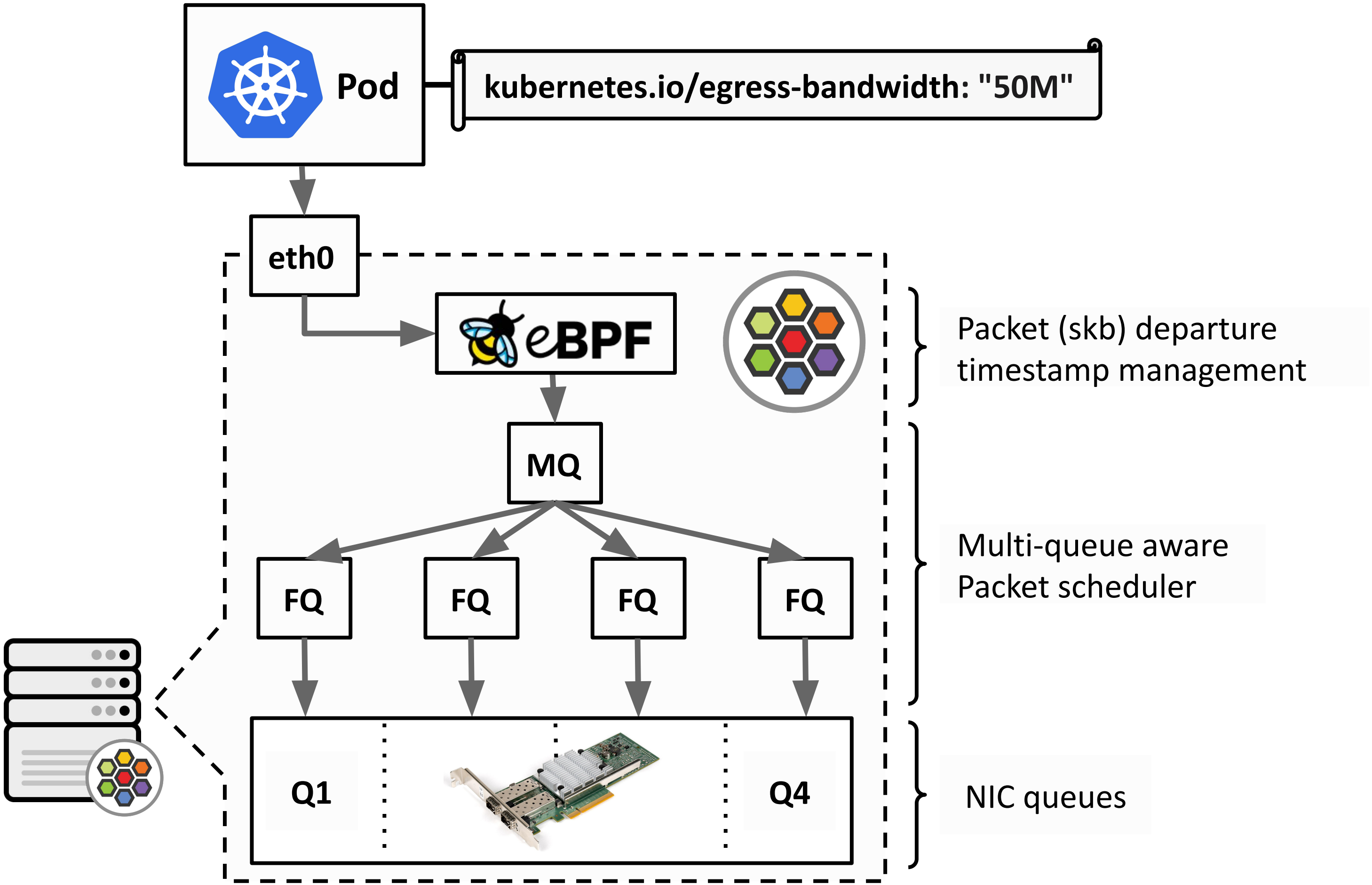
- ➔ struct sock_cgroup_data always holds reliable cgroup pointer
- ➔ Implicitly also addresses the v2 reference count leaks
- ➔ Fix along with test cases has been upstreamed recently

Part 2: TCP pacing for Pods from initns

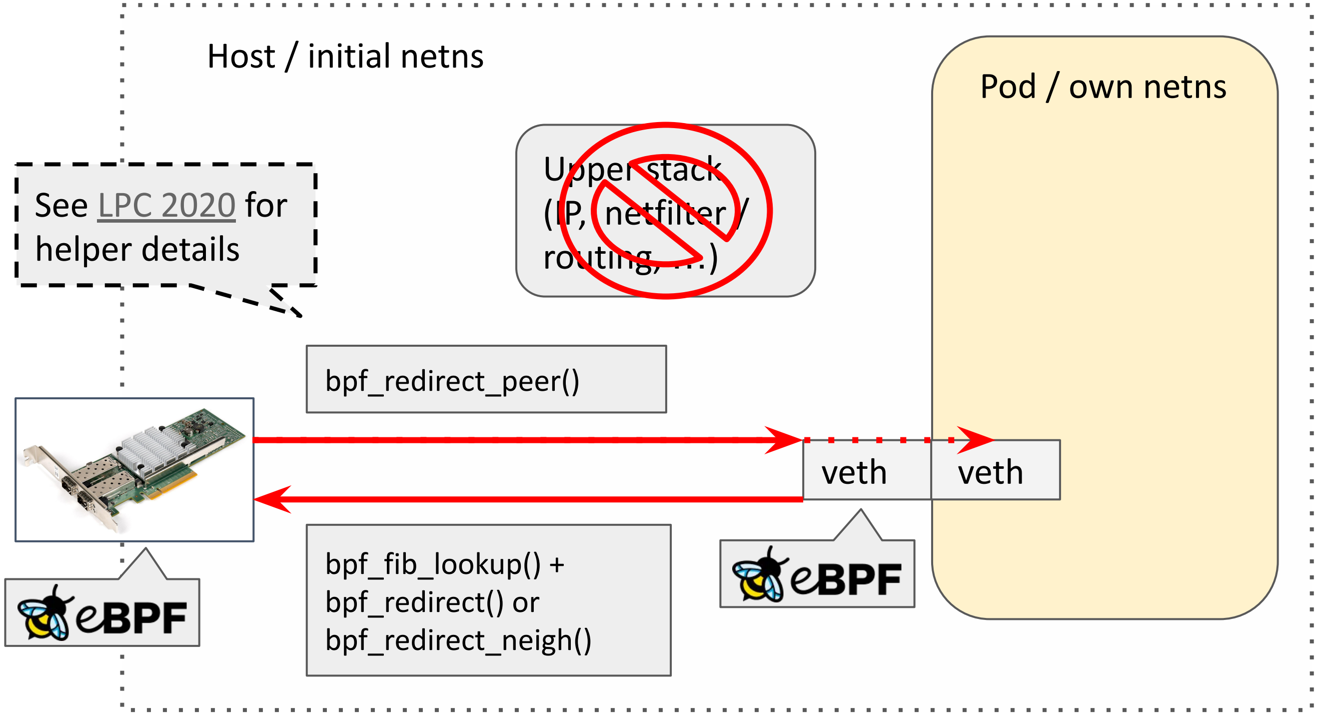
Current state: Cilium & K8s

K8s Pod-specific ingress/egress bandwidth annotation:

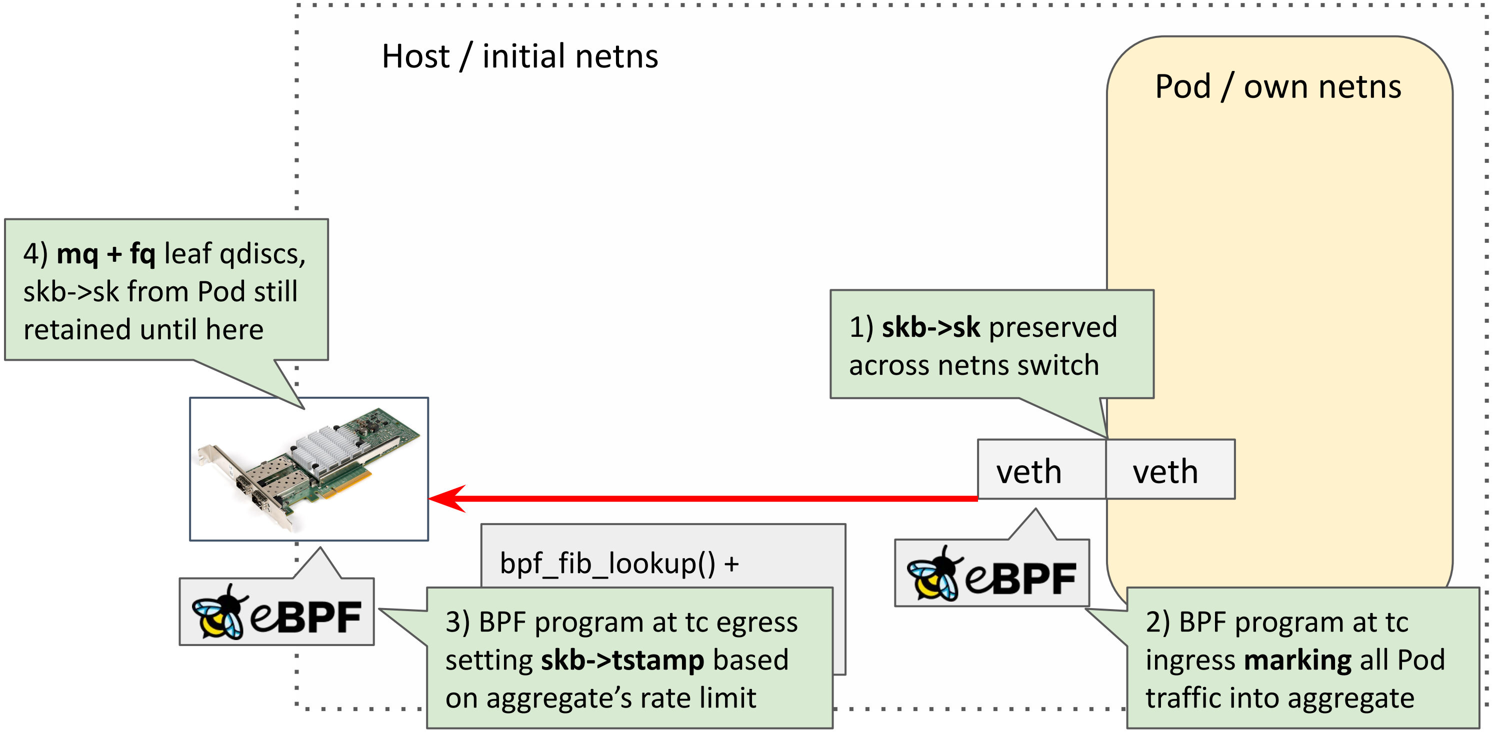
- ➔ Handled by K8s CNI plugins (e.g. Cilium or bandwidth plugin)
- ➔ Semantics for rate enforcement points defined by plugin:
  - K8s bandwidth plugin uses combination of ifb & tbf qdisc
  - Cilium natively implements EDT via BPF & fq qdisc for egress


Pod
kubernetes.io/egress-bandwidth: "50M"
eth0
eBPF
MQ
FQ
FQ
FQ
FQ
Q1
Q4
Packet (skb) departure
timestamp management
Multi-queue aware
Packet scheduler
NIC queues

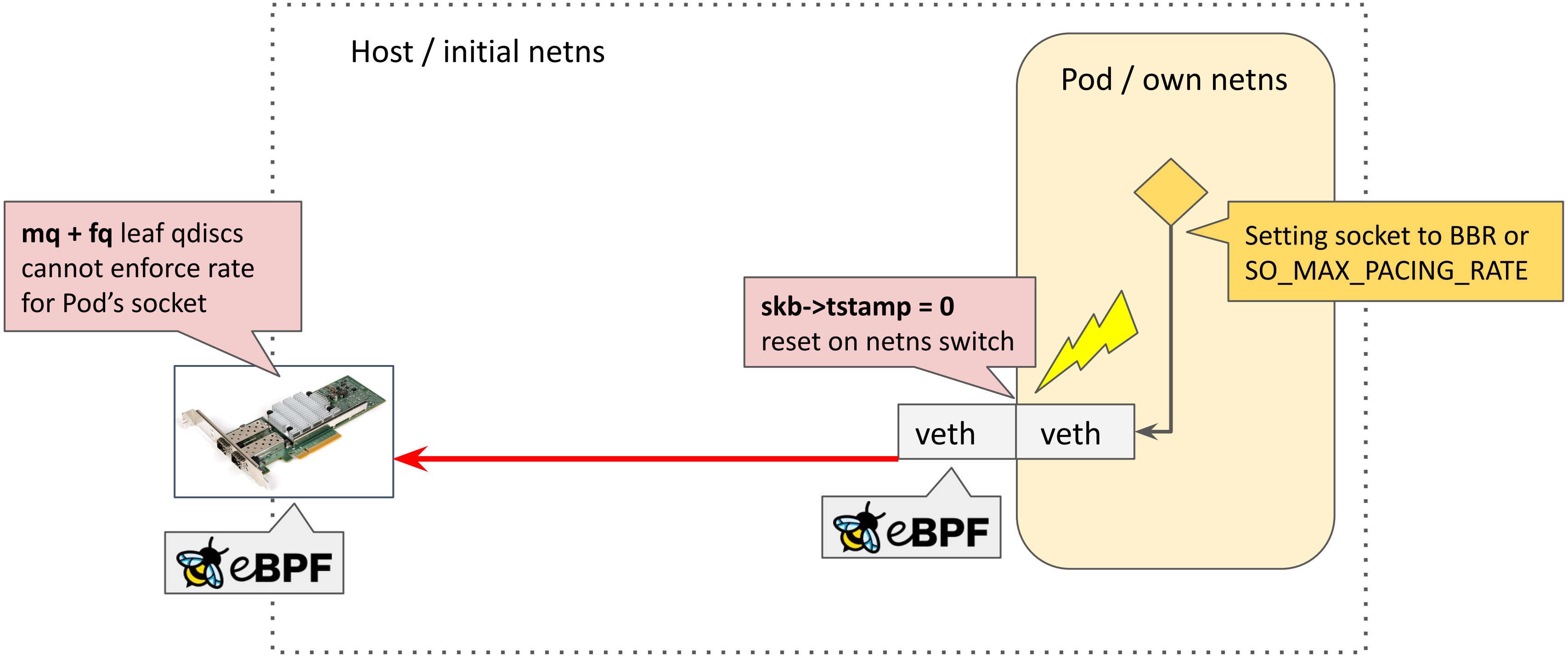
BPF datapath walk-through: Overview forwarding



BPF datapath walk-through: Works today

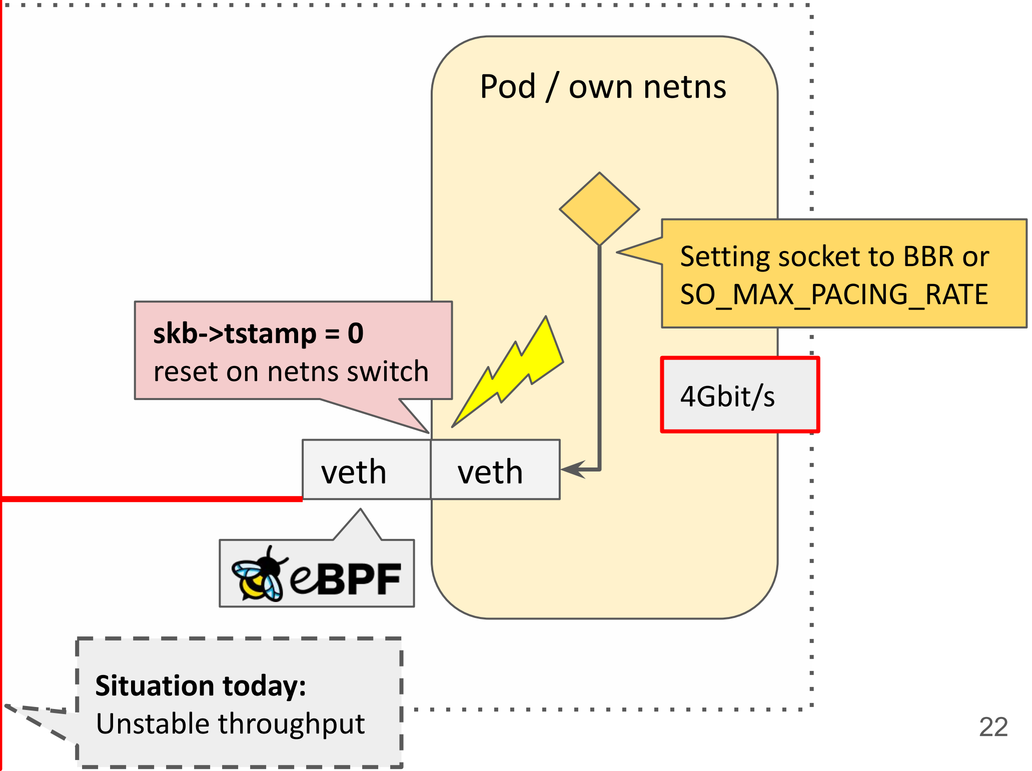


BPF datapath walk-through: Next steps



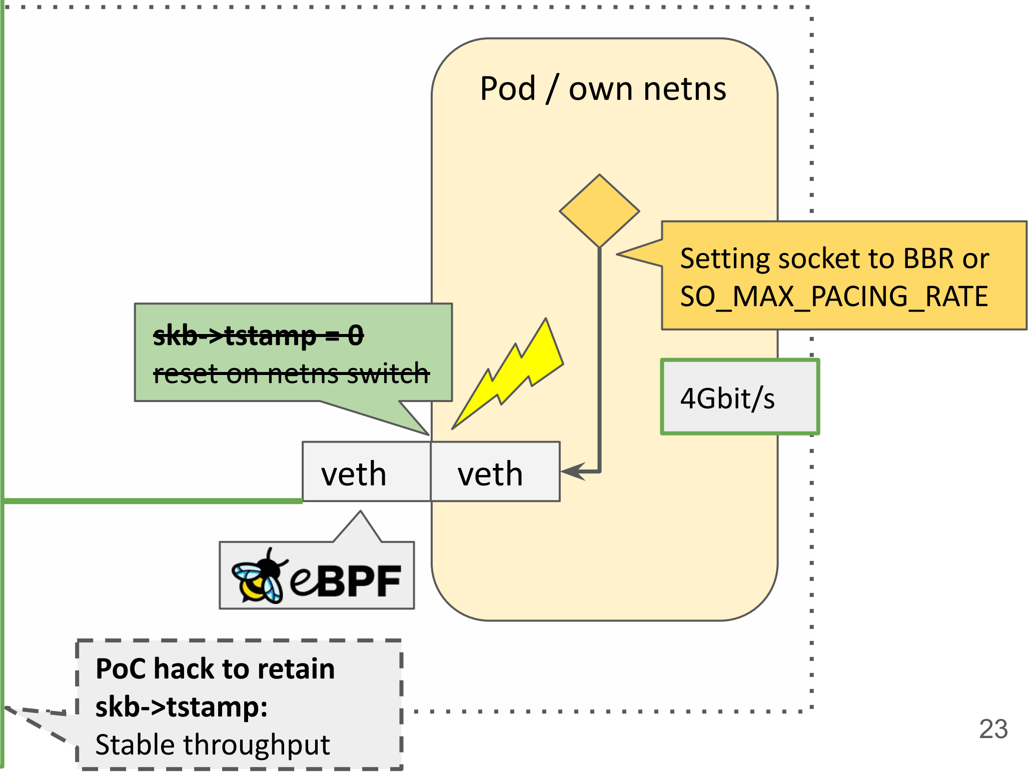
BPF datapath walk-through: Next steps

```
root@apoc:~/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
 87380  16384  16384    40.04     655.52

root@apoc:~/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
 87380  16384  16384    40.07    1274.70

root@apoc:~/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
 87380  16384  16384    40.07    1519.32

root@apoc:~/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
 87380  16384  16384    40.06     849.96
```



BPF datapath walk-through: Next steps

```
root@apoc:~/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
 87380  16384  16384    40.01    3976.04

root@apoc:~/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
 87380  16384  16384    40.01    3961.40

root@apoc:~/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
 87380  16384  16384    40.01    3957.66

root@apoc:~/go/src/github.com/cilium/cilium# netperf
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
 87380  16384  16384    40.01    3977.37
```



Rationale on today’s timestamp reset

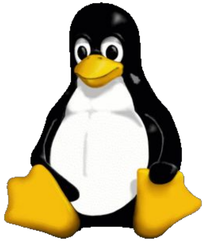
Kernel uses different clock bases for skb->tstamp:

➔ Ingress is CLOCK_TAI, egress is CLOCK_MONOTONIC (as is fq)

➔ Forwarding from RX to TX would cause drop in fq due to overreaching fq’s drop horizon (given clock’s offsets)

➔ No means to figure out clock base from skb->tstamp, hence reset

Rationale on today's timestamp reset



Can skb->tstamp be normalized to a single base?

- ➔ Initially TCP EDT was based on CLOCK_TAI as well
- ➔ Nodes were seen where improper RTC setup caused clock discontinuities of +50yrs during boot
- ➔ Confused fq which lead to drops, thus broke TCP
  - Hence CLOCK_MONOTONIC & reset on direction switch

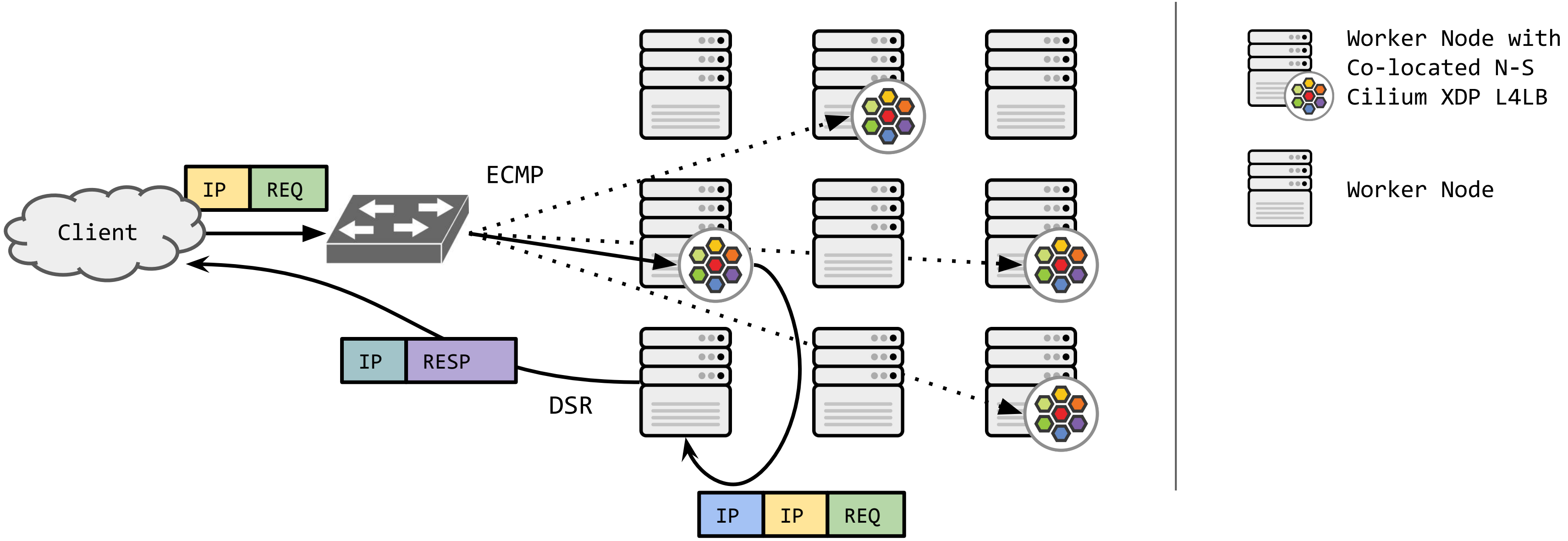
Approach to fixing the timestamp reset

Adding new skb->tstamp_base bit (defines: 0 ➔ TAI, 1 ➔ MONO)

- ➔ skb_set_tstamp_{mono,tai}(skb, ktime) helper used by RX and TX
- ➔ fq_enqueue() detects TAI clock and resets skb->tstamp
- ➔ All skb->tstamp = 0 due to forwarding are then removed
  - skb_mstamp_ns union could be removed as well
- ➔ net_timestamp_check() must be deferred in RX after tc ingress

Part 3: Managed neighbor/fib extensions

Use case: Cilium's XDP L4LB



Current state: Cilium's XDP L4LB

XDP LB receives packet to svcIP/port, forwards to backendIP/port:

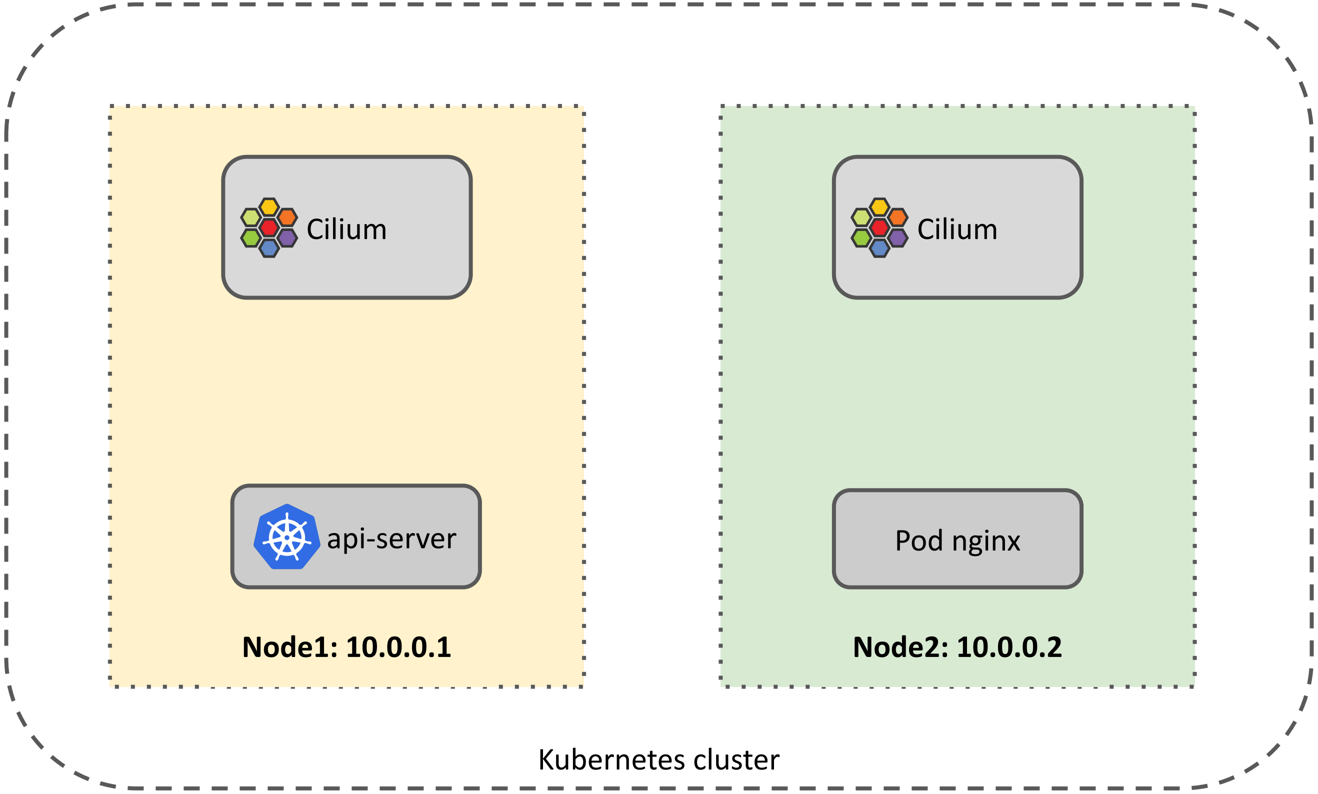
- ➔ BPF: Either DNAT & SNAT or DSR with IPIP/IP6IP6 encapsulation
- ➔ In both cases outer header has backendIP as destination
- ➔ bpf_fib_lookup() used to piggyback on neighbor resolution
- ➔ Pushed back out via XDP_TX (transparent of phys/bond device)

Current state: Cilium’s XDP L4LB

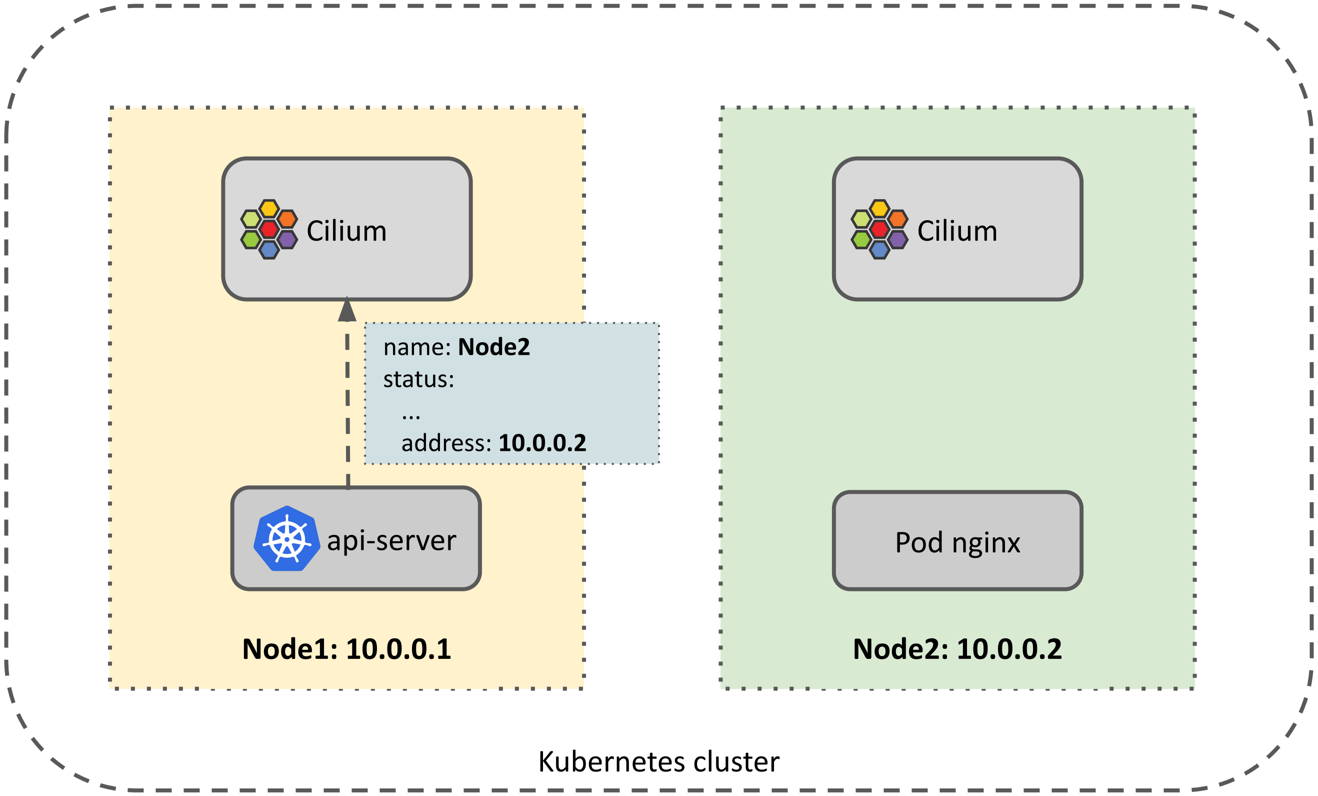
Neighbor resolution under XDP:

- ➔ Neighbor entry must be present in table, cannot resolve from XDP
- ➔ Agent currently resolves entries manually which is a pain point
- ➔ Pushes resolution as NUD_PERMANENT into neighbor table

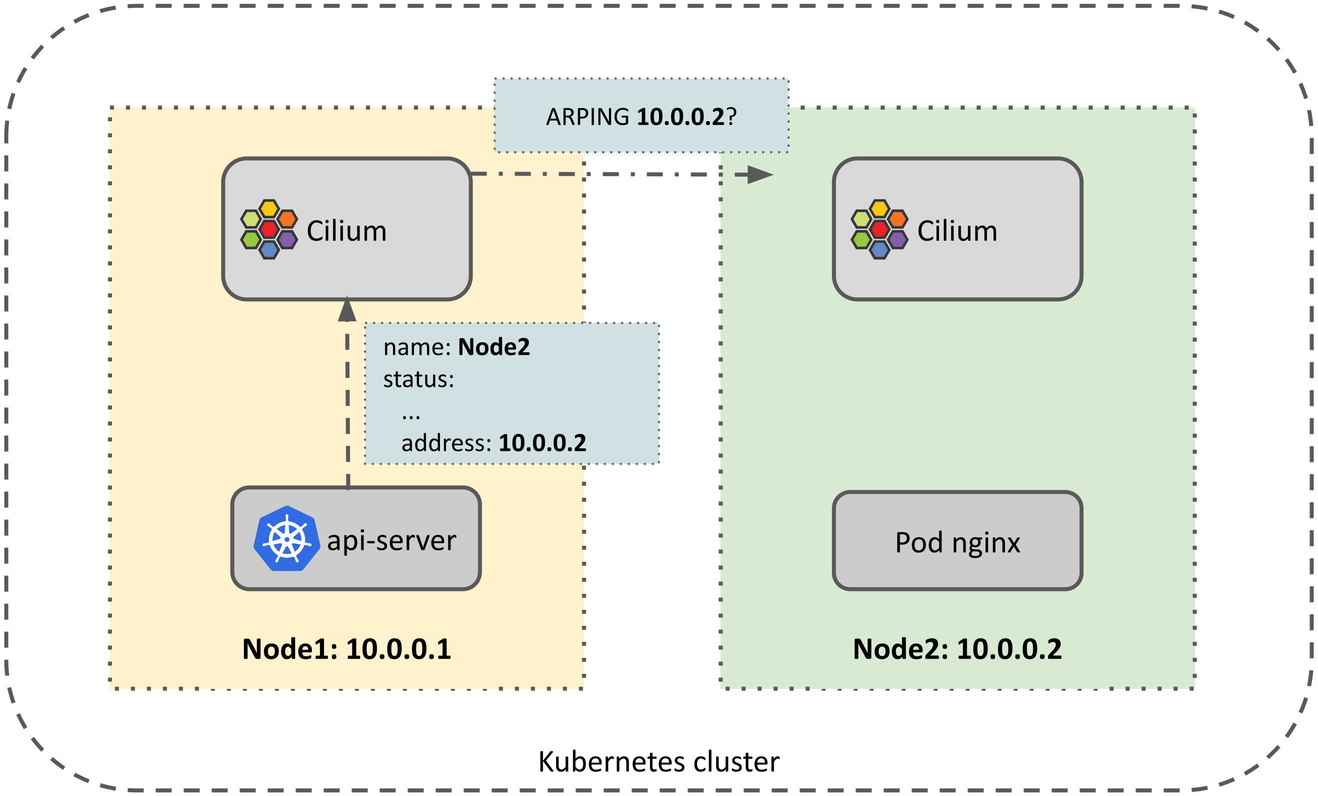
Current state: Neighbor entry management



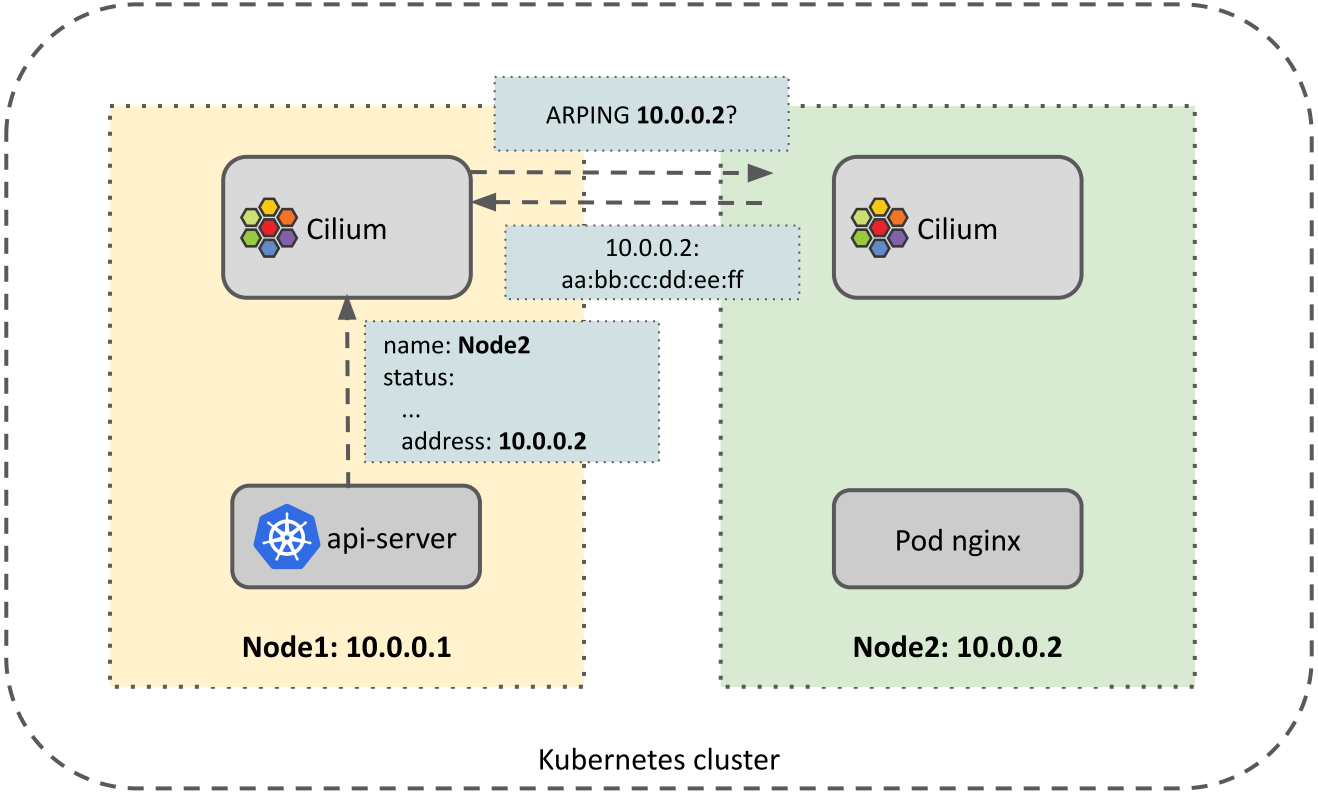
Current state: Neighbor entry management



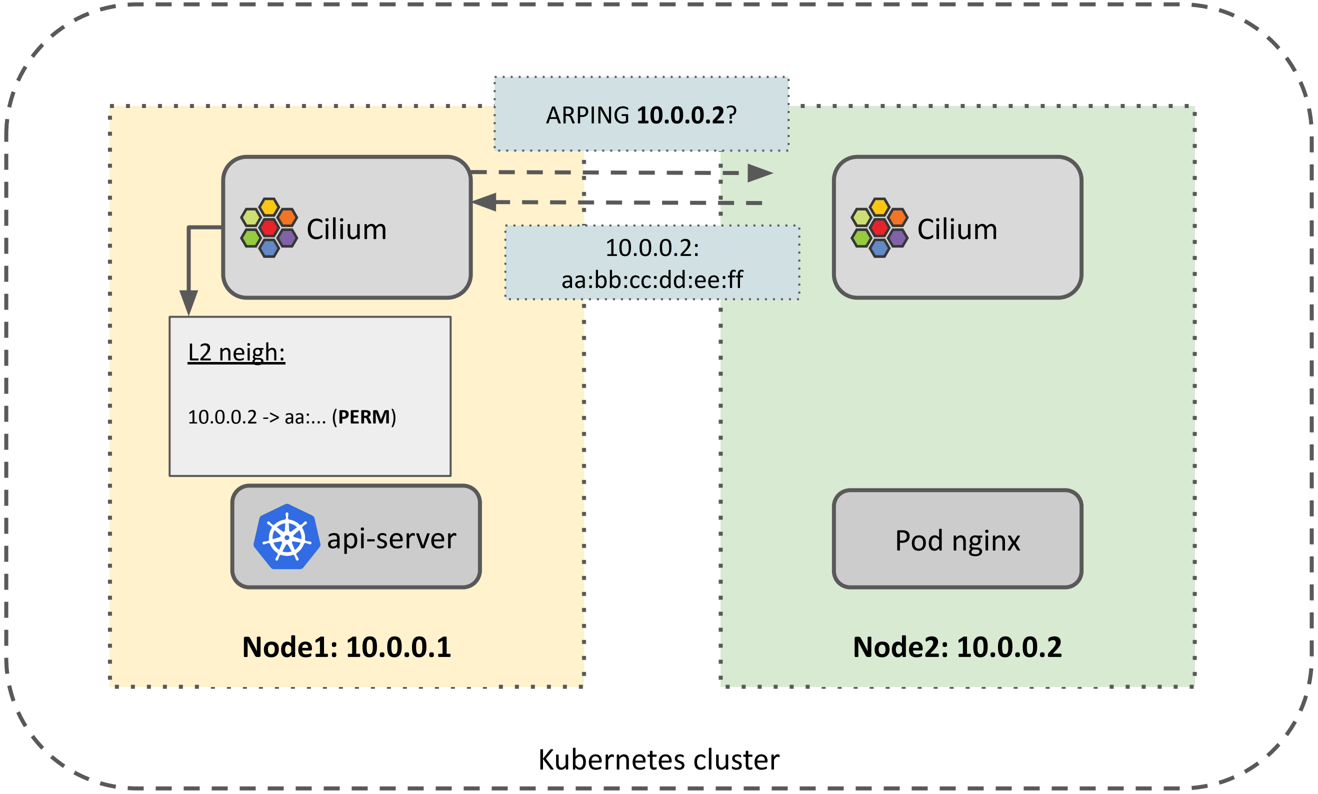
Current state: Neighbor entry management



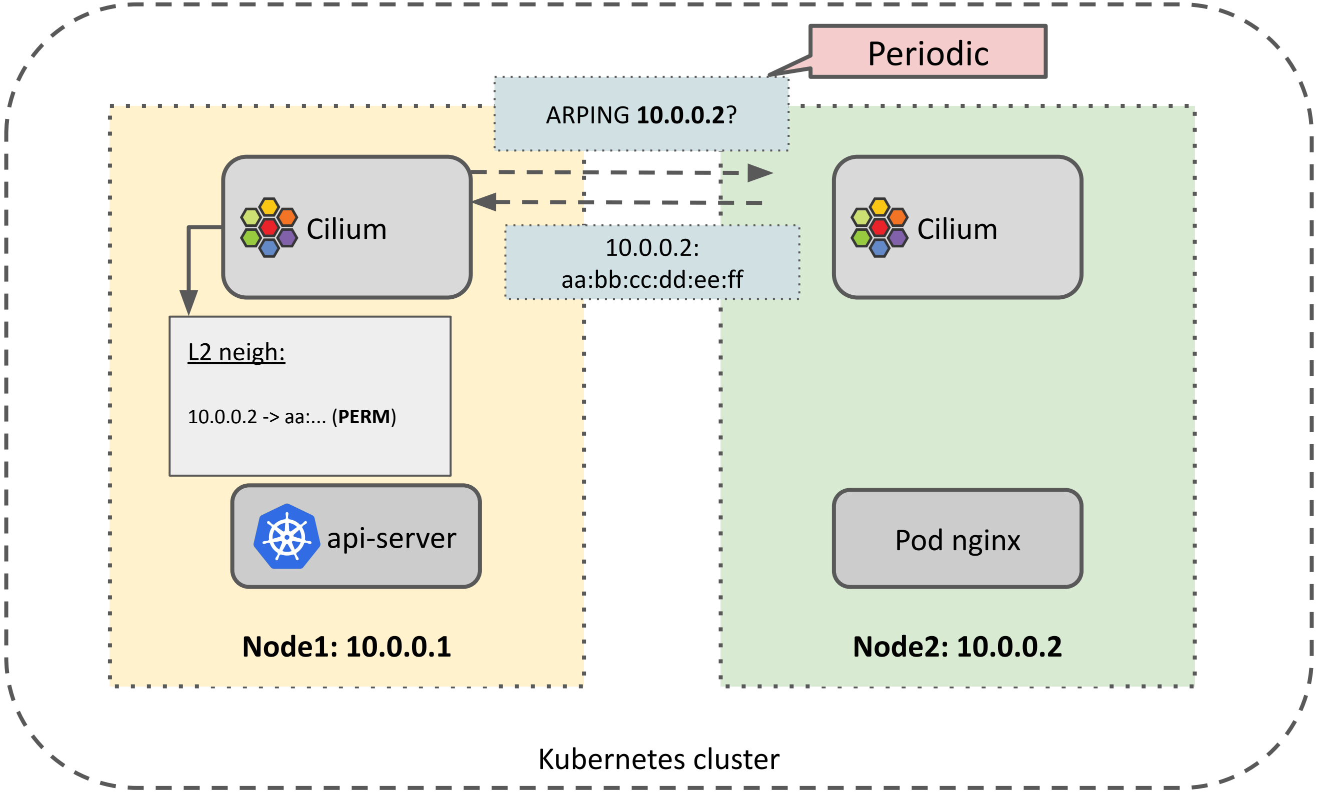
Current state: Neighbor entry management



Current state: Neighbor entry management



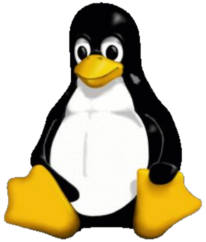
Current state: Neighbor entry management



Problems with current approach

- ➔ How often to arping? (Currently once every 5 min)
- ➔ Buggy, for example:
  - ○ An obsolete NUD_PERMANENT entry for the api-server node is fatal after agent restart if the former’s L2 address changed
  - ○ No auto-updates from active traffic processed by the local stack
- ➔ Duplicating logic of net/ipv4/arp.c
- ➔ Need an equivalent for IPv6’s ND

Managed neighbor entry: Rationale



- ➔ Control plane (here: Cilium agent) requirements
  - ○ Netlink route lookup: backendIP in same L2 or via GW IP?
  - ○ Pushes L3 (without L2) addresses into neighbor table
- ➔ Neighboring subsystem auto-resolves them
- ➔ Periodically keeps them in REACHABLE state
- ➔ Option to avoid GC eviction
- ➔ Visibility for agent restart to resync/clean obsolete L3 entries

Managed neighbor entry: Design

➔ We can piggyback on NTF_USE | NTF_EXT_LEARNED neigh flag
  - Gets us quite close already:
    - Triggers one-time resolution via neigh_event_send()
    - Updates STALE state upon external/internal traffic events
    - Ensures that neigh entries are not added to GC list

Managed neighbor entry: Design

- ➔ We can piggyback on NTF_USE | NTF_EXT_LEARNED neigh flag
  - ○ What it does not do:
    - No self-managed auto-refresh to get back to REACHABLE from STALE state due to inactivity
    - Creation flags not propagated back to user space
    - Not retained upon carrier-down events (like NUD_PERMANENT)

Managed neighbor entry: Design

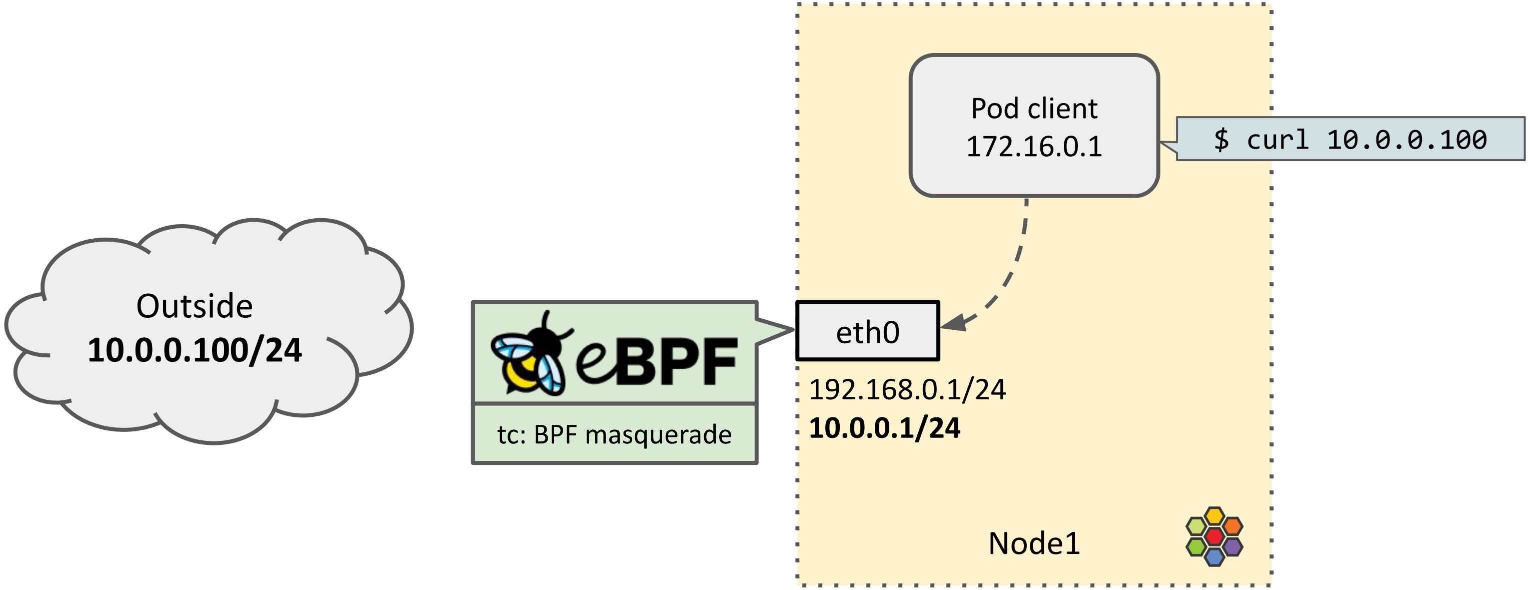
- ➔ Proposal: New NUD_MANAGED state for neigh entry creation
  - ○ Volatile pseudo-state (not fixed as in NUD_PERMANENT):
    - Implies NTF_USE and adds entry to a per-neigh table list
  - ○ Uses delayed system-wq to trigger neigh_event_send() for entries
  - ○ Triggered on BASE_REACHABLE_TIME/2 with slack
  - ○ NUD_MANAGED can be combined with NTF_EXT_LEARNED
  - ○ Retained upon carrier-down & refreshed once up again

Managed neighbor entry: iproute2 example

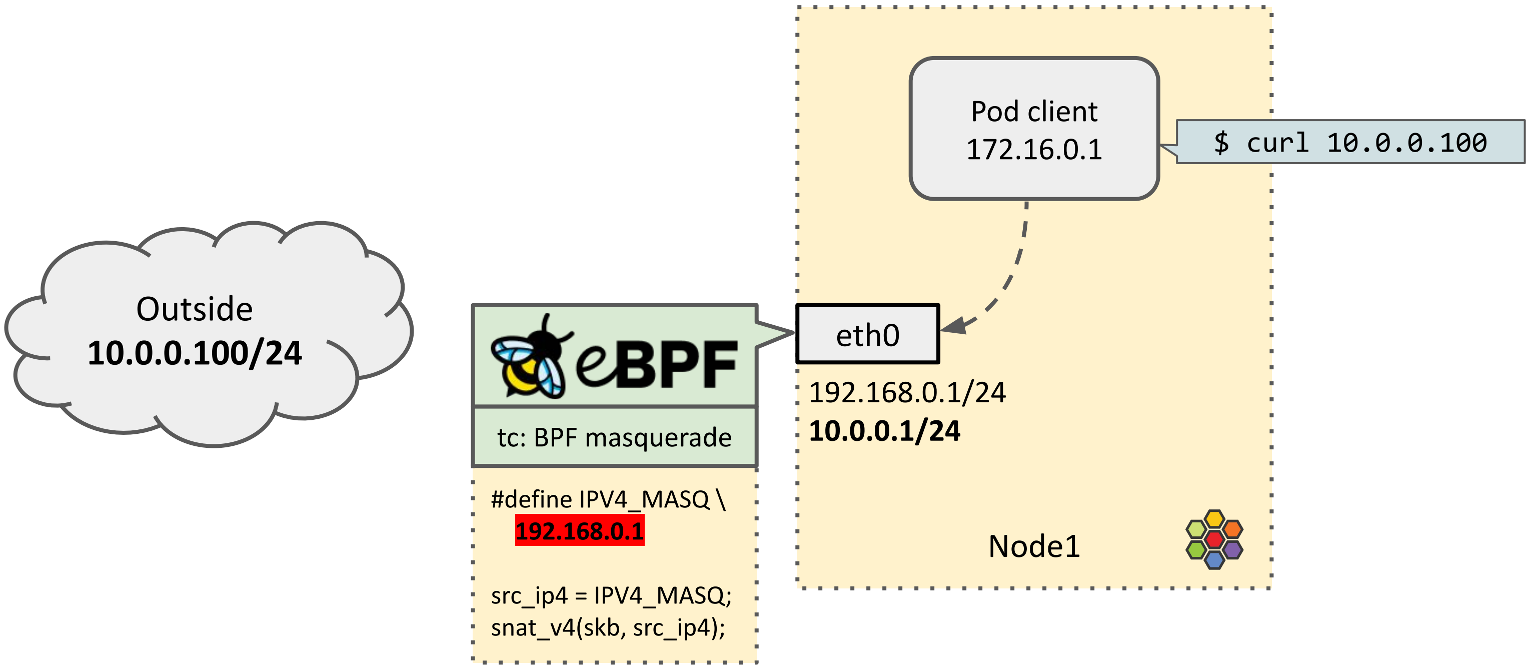
- Entry creation via ‘nud managed’:
  - `ip neigh replace 192.168.1.99 dev enp5s0 extern_learn nud managed`

- Entry dump (including flag propagation fix):
  - `192.168.1.99 dev enp5s0 lladdr 98:9b:cb:05:2e:ae use extern_learn REACHABLE`

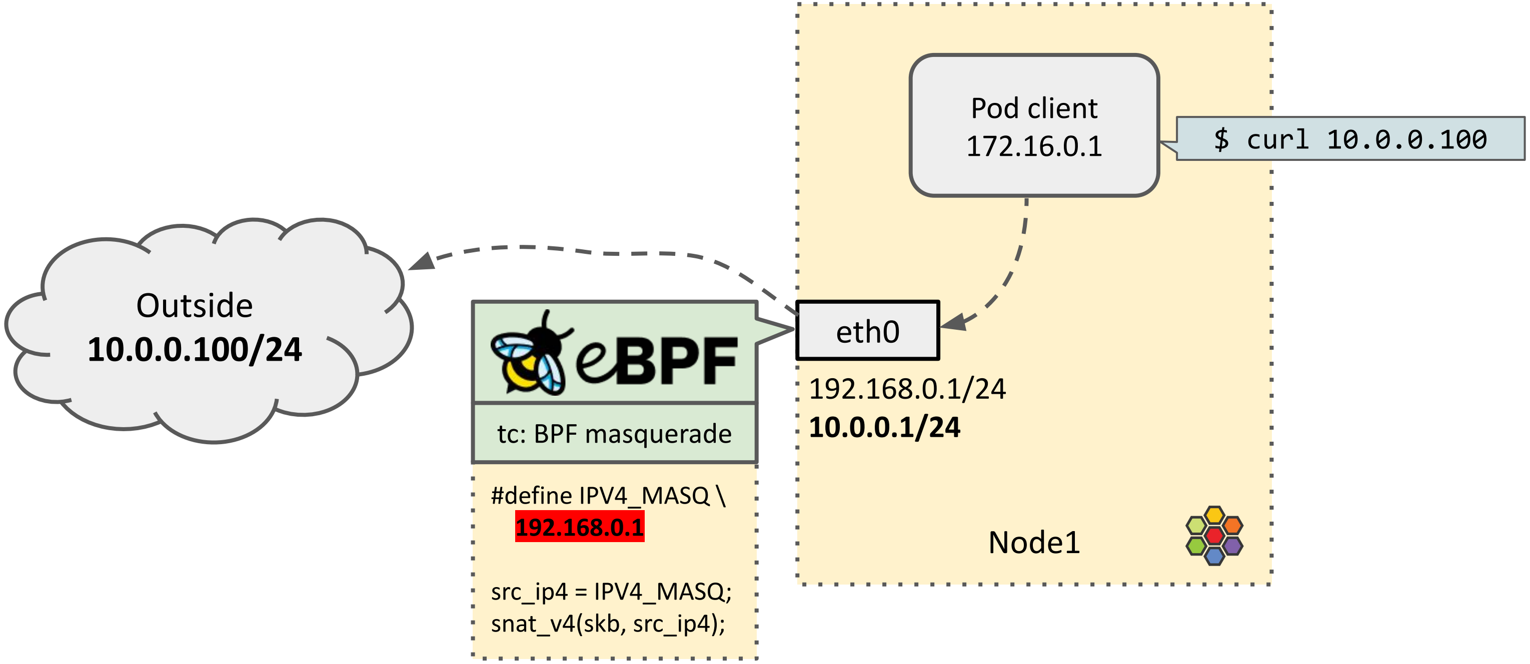
FIB extensions 1/2: Source IP address selection for SNAT



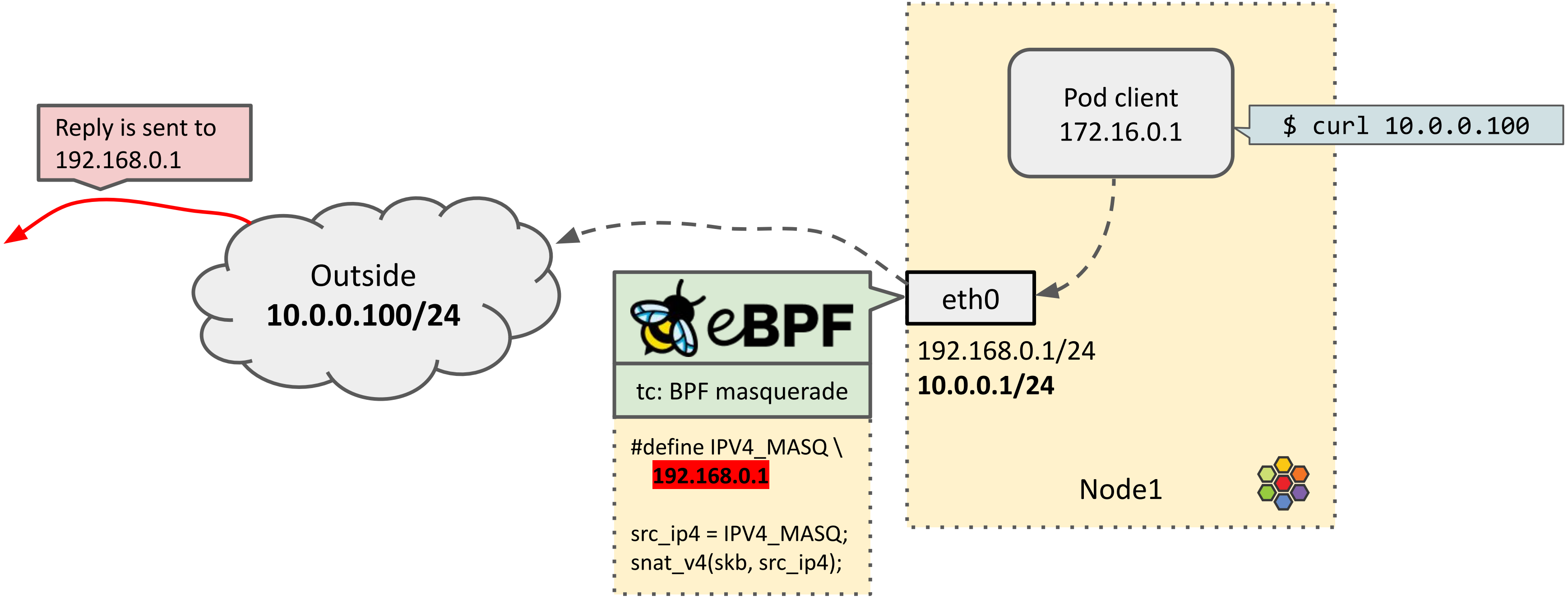
FIB extensions 1/2: Source IP address selection for SNAT



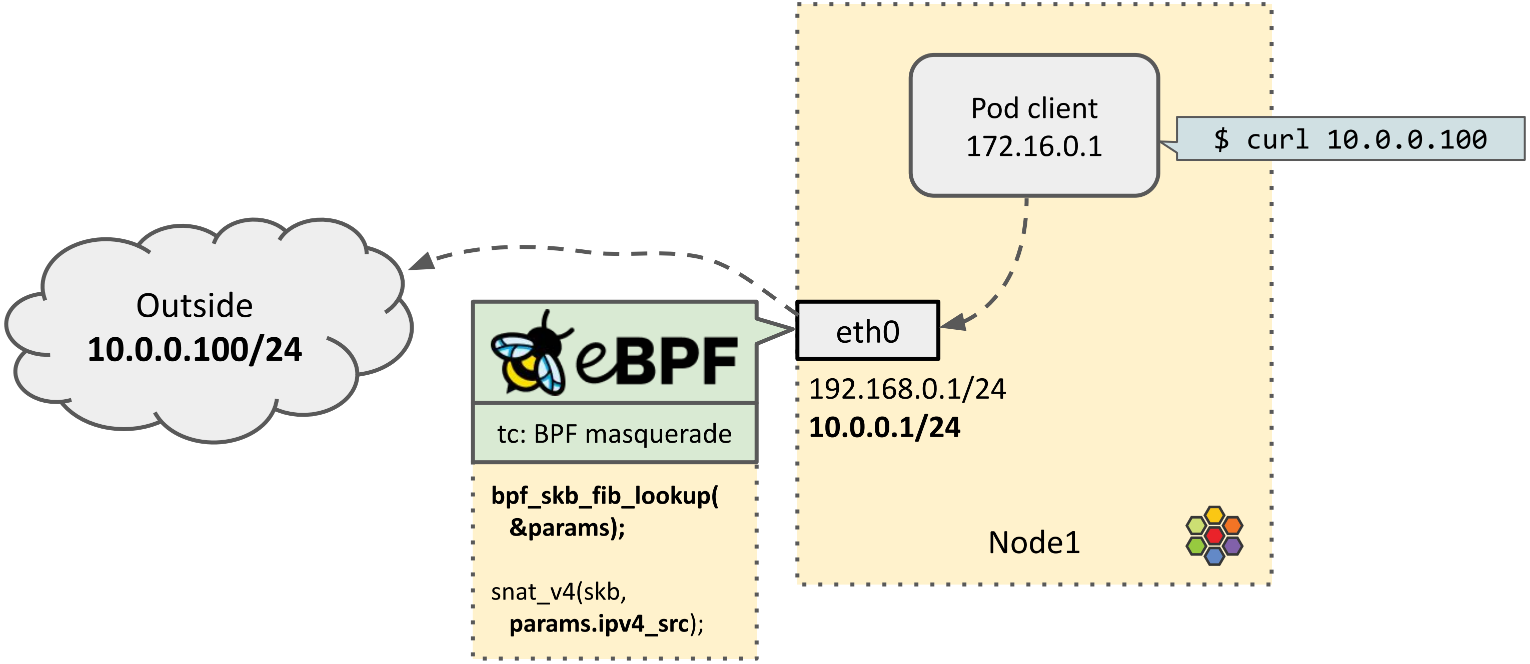
FIB extensions 1/2: Source IP address selection for SNAT



FIB extensions 1/2: Source IP address selection for SNAT



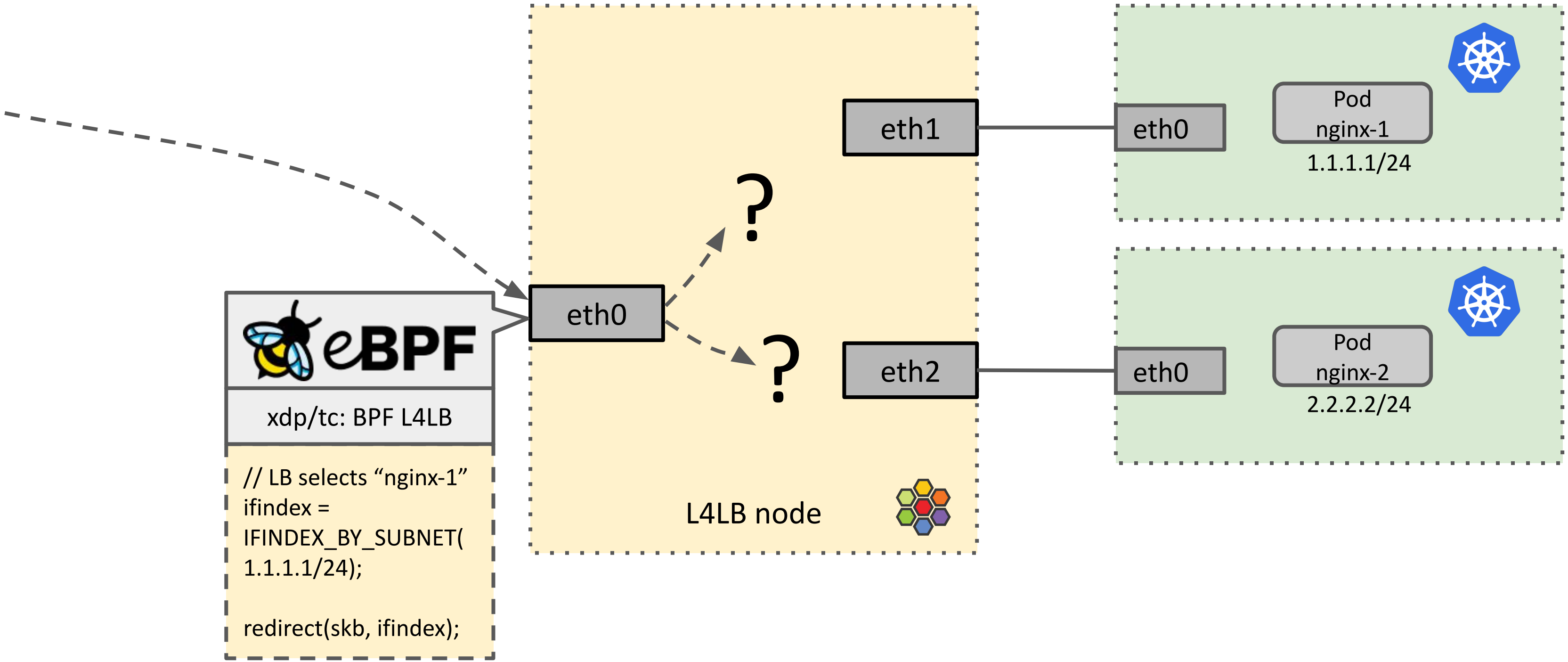
FIB extensions 1/2: Source IP address selection for SNAT



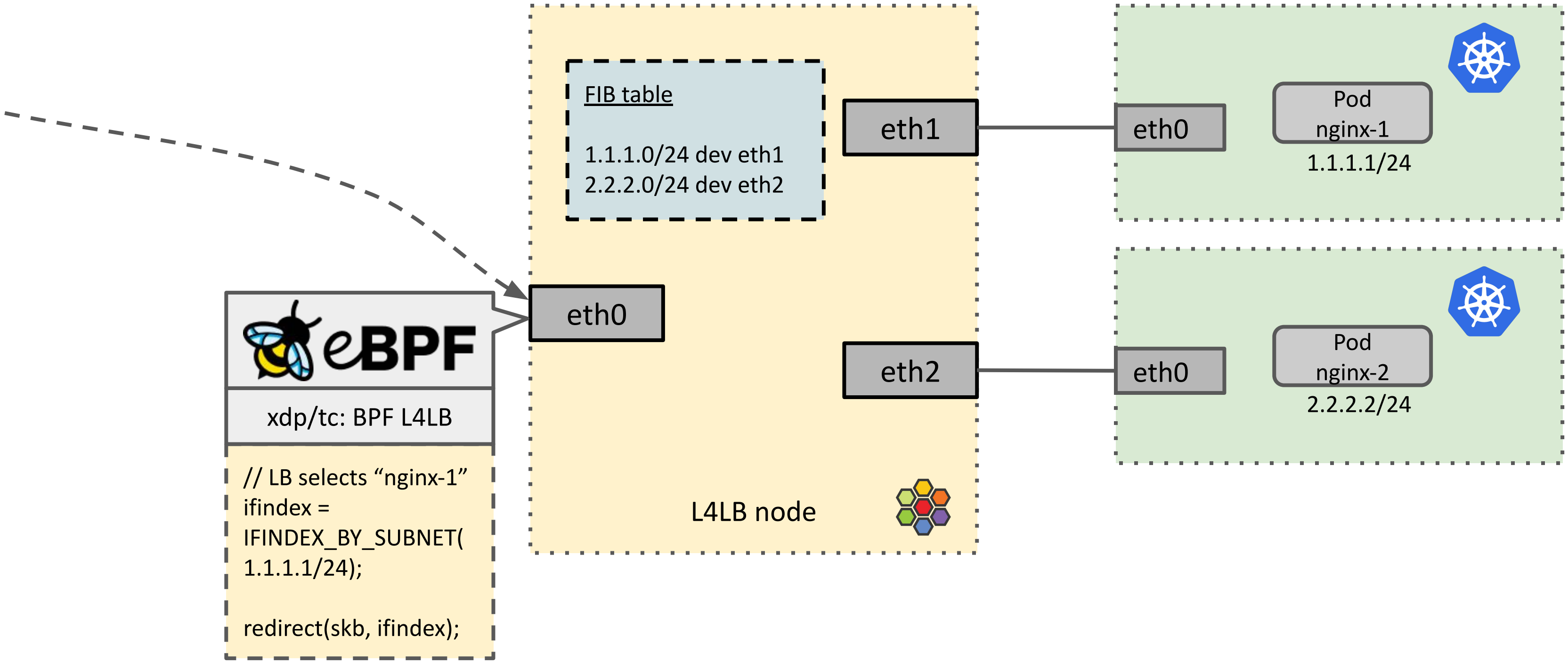
Proposed solution for source address selection

- ➔ Use bpf_{xdp,skb}_fib_lookup() for source IP address selection
  - ○ Requires changes to the BPF helper implementation
- ➔ Introduction of a new BPF_FIB_LOOKUP_SET_SRC flag
  - ○ Sets the fib_params.ipv{4,6}_src address to: fib_result_prefsrc() / fib6_info.fib6_src
- ➔ Another benefit: No need to hardcode IP addresses into the datapath

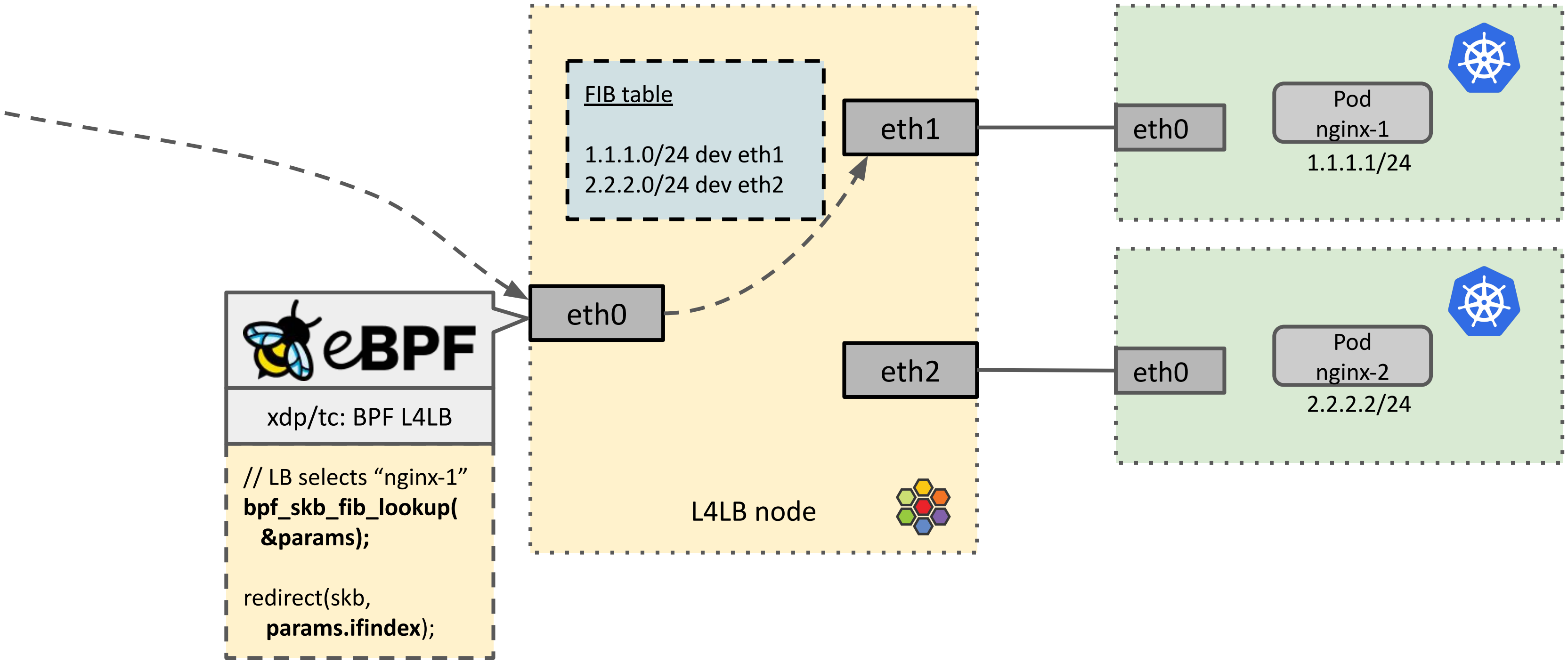
FIB extensions 2/2: Redirect in multi-homed network



FIB extensions 2/2: Redirect in multi-homed network



FIB extensions 2/2: Redirect in multi-homed network



Proposed solution for target ifindex selection

- ➔ Use bpf_{xdp,skb}_fib_lookup() to determine ifindex
  - Requires fixing the BPF helper implementation, too
- ➔ Do not require ifindex when !BPF_FIB_LOOKUP_DIRECT
  - “params->ifindex = dev->ifindex;” already exists
  - Is current behavior a bug?
- ➔ Commit for making params->ifindex optional (to be upstreamed)

Part 4: Wildcarded BPF map lookups

Current state: Cilium XDP L4LB use case

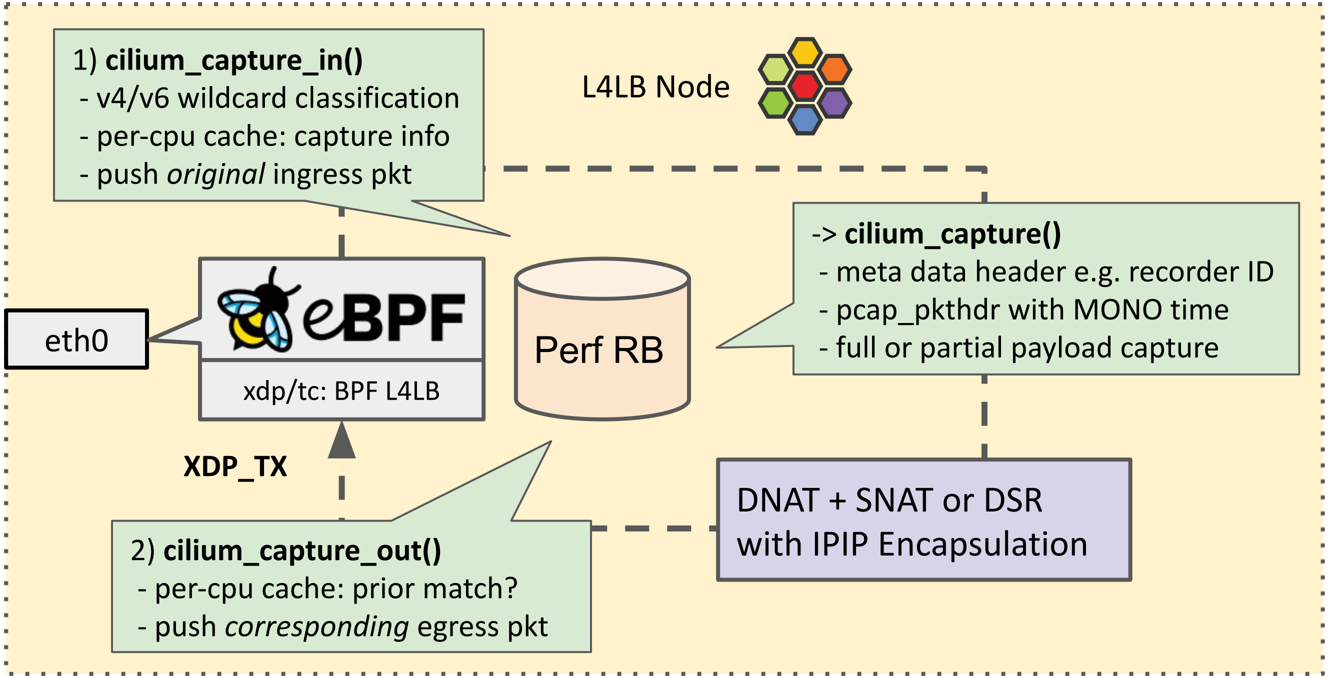
Flexible LB traffic recorder to correlate inbound/outbound pkts:

- ➔ Introspection on path taken from fabric to L4LBs to L7 proxies/backends
- ➔ Higher-level API for out-of-band programming of L4LB agents
- ➔ Hubble then constructing PCAP for offline troubleshooting

```
$ hubble record "0.0.0.0/0 0 192.168.33.11/32 80 TCP"
Started recording. Press CTRL+C to stop.
2021-05-19T10:54:07Z Status: 345 packets (27445 bytes) written
```

```
$ cilium recorder list
ID     Capture Length   Wildcard Filters
4241   full             0.0.0.0/0:0        ->   192.168.33.11/32:80   TCP

Users   Priority      Wildcard Masks
1       56            00000000:0      ->   ffffffff:ffff   ff
$
```

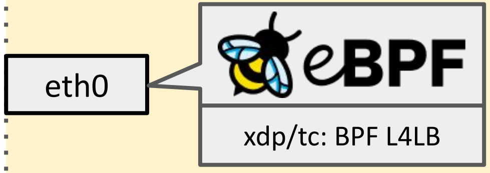
Cilium XDP L4LB: PCAP recorder overview



PCAP recorder: Classifier rules

1) **cilium_capture_in()**
- v4/v6 wildcard classification

L4LB Node




```
$ cilium recorder list
ID      Capture Length   Wildcard Filters
4241    full             0.0.0.0/0:0          ->    192.168.33.11/32:80    TCP

Users   Priority     Wildcard Masks
1       56           00000000:0        ->    ffffffff:ffff    ff
$ 
```

One 'Recorder' consists of:

- Source CIDR, destination CIDR
- Source Port, destination Port (0: any, n: direct match)
  - Currently unsupported: n-m range
- Protocol (0: any, n: direct match)

PCAP recorder: Classifier rules

1) **cilium_capture_in()**
- v4/v6 wildcard classification

L4LB Node

eth0

eBPF

xdp/tc: BPF L4LB

```
$ cilium recorder list
ID      Capture Length    Wildcard Filters
4241    full              0.0.0.0/0:0         ->    192.168.33.11/32:80    TCP

Users   Priority    Wildcard Masks
1       56          00000000:0        ->    ffffffff:ffff    ff
$ 
```

One 'Recorder' consists of:

- Source CIDR, destination CIDR
- Source Port, destination Port (0/n)
  - Currently unsupported: n-m range
- Protocol (0/n)

Agent:

- User API for programming recorders
- Tracking different masks from rules
- Regens datapath on mask set change
- v4/v6 hashtable each for rule lookup

PCAP recorder: Classifier rules

```
static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
	struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };

	[...]

_Pragma("unroll")
	for (i = 0; i < size; i++) {
		cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
		match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
		if (match)
			return match;
	}

	return NULL;
}
```

PCAP recorder: Classifier rules

```
static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
        struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };

        [...]

_Pragma("unroll")
        for (i = 0; i < size; i++) {
                cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
                match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
                if (match)
                        return match;
        }

        return NULL;
}
```

Dynamic, ordered mask set, regenerated by agent on the fly.

PCAP recorder: Classifier rules

```
static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
        struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };

        [ ... ]

_Pragma("unroll")
        for (i = 0; i < size; i++) {
                cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
                match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
                if (match)
                        return match;
        }

        return NULL;
}
```

Generating masked key (lkey) from original tuple (okey) and current mask.

PCAP recorder: Classifier rules

```
static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
        struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };

        [...]

        _Pragma("unroll")
        for (i = 0; i < size; i++) {
                cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
                match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
                if (match)
                        return match;
        }

        return NULL;
}
```

Using masked key (lkey) for the hashtable lookup.

PCAP recorder: Classifier rules

```
static __always_inline struct capture_rule *
cilium_capture4_classify_wcard(struct __ctx_buff *ctx)
{
        struct capture4_wcard prefix_masks[] = { PREFIX_MASKS4 };

        [...]

_Pragma("unroll")
        for (i = 0; i < size; i++) {
                cilium_capture4_masked_key(&okey, &prefix_masks[i], &lkey);
                match = map_lookup_elem(&CAPTURE4_RULES, &lkey);
                if (match)
                        return match;
        }

        return NULL;
}
```

Holds Recorder ID and capture length.

PCAP recorder: Classifier rules

```c
static __always_inline void
cilium_capture4_masked_key(const struct capture4_wcard *orig,
                           const struct capture4_wcard *mask,
                           struct capture4_wcard *out)
{
        out->daddr = orig->daddr & mask->daddr;
        out->saddr = orig->saddr & mask->saddr;
        out->dport = orig->dport & mask->dport;
        out->sport = orig->sport & mask->sport;
        out->nexthdr = orig->nexthdr & mask->nexthdr;
        out->dmask = mask->dmask;
        out->smask = mask->smask;
}
```

Masked key (lkey) generation for the map lookup.

Problems with current approach

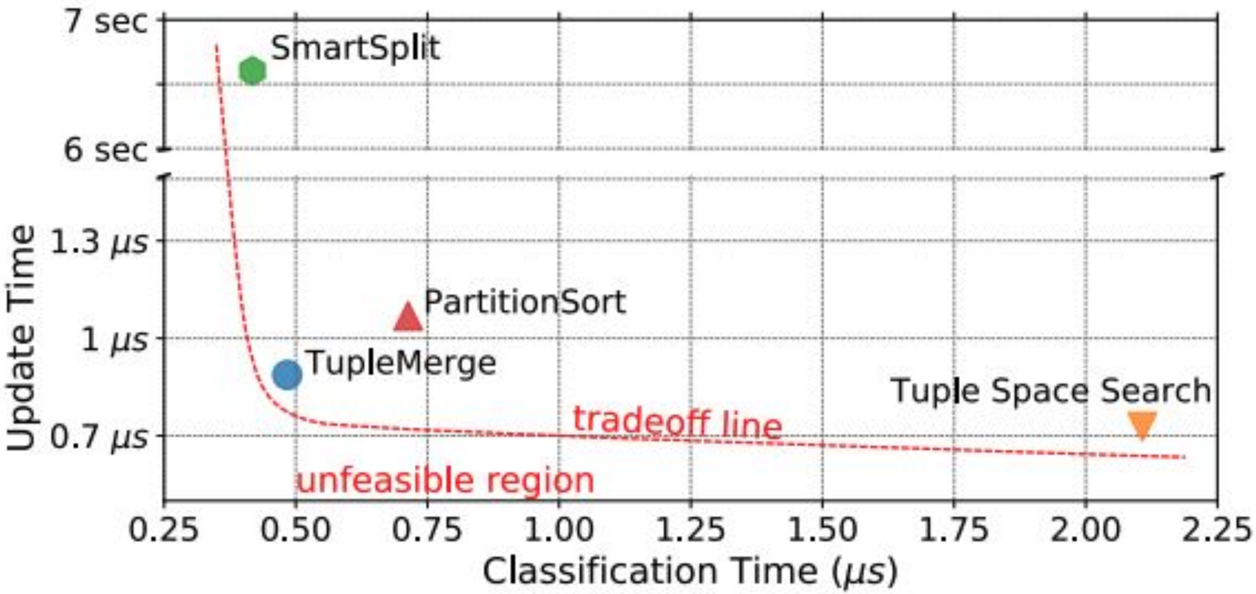
- ➔ “Poor man’s version” of wildcard match:
  - Assumes small number of masks, *but* allows for large number of matches within the mask set: acceptable for our use-case
  - Requires expensive on-the-fly recompilation on mask set change
  - Linearity for probing different masks
- ➔ Works on old kernels, but loop unrolling risks verifier complexity issues

Native wildcard-supported BPF map

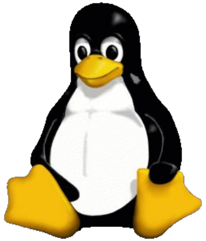
- ➔ Ideally native BPF map to avoid costly code regeneration:
  - ‘Very fast’ lookup time (Millions/sec)
  - ‘Reasonably fast’ update time (Thousands/sec)
- ➔ First use-case dates back to 2018 in context of BPF + OVS to implement Megaflows in BPF, effort stalled however

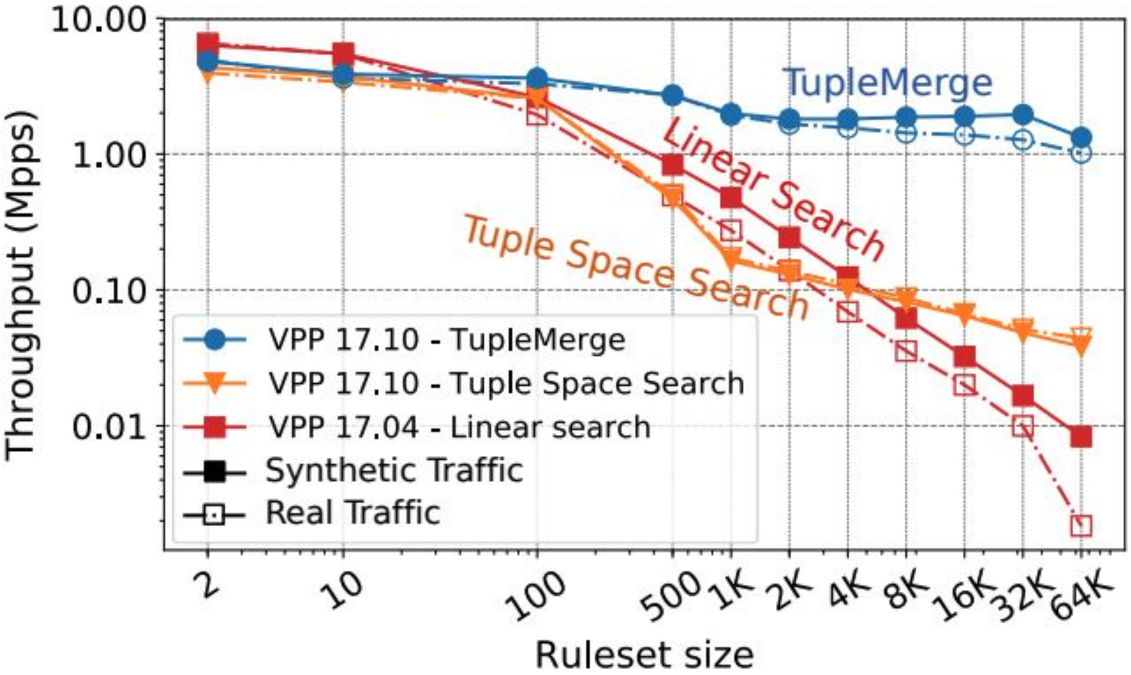
Native wildcard-supported BPF map

➔ Potential map candidate: TupleMerge (Eric Torng et al.)
  ○ Current state-of-the-art in classification algorithms



Native wildcard-supported BPF map





➔ Next step on our agenda: PoC implementation for BPF runtime

Thanks! Questions, feedback, comments?

- ➔ Try it out: https://cilium.link/kubeproxy-free
- ➔ Cilium: https://github.com/cilium/cilium
- ➔ PoC code: https://git.kernel.org/[...]/dborkman/bpf.git
  https://github.com/brb/linux