K8s Service Load Balancing with BPF & XDP

Daniel Borkmann & Martynas Pumputis, Cilium.io

Linux Plumbers 2020

Agenda

→ Part 1: Kubernetes networking 101

→ Part 2: Cilium’s service LB & lessons learned

→ Part 3: New BPF kernel extensions

Part 1: K8s networking 101

K8s networking 101

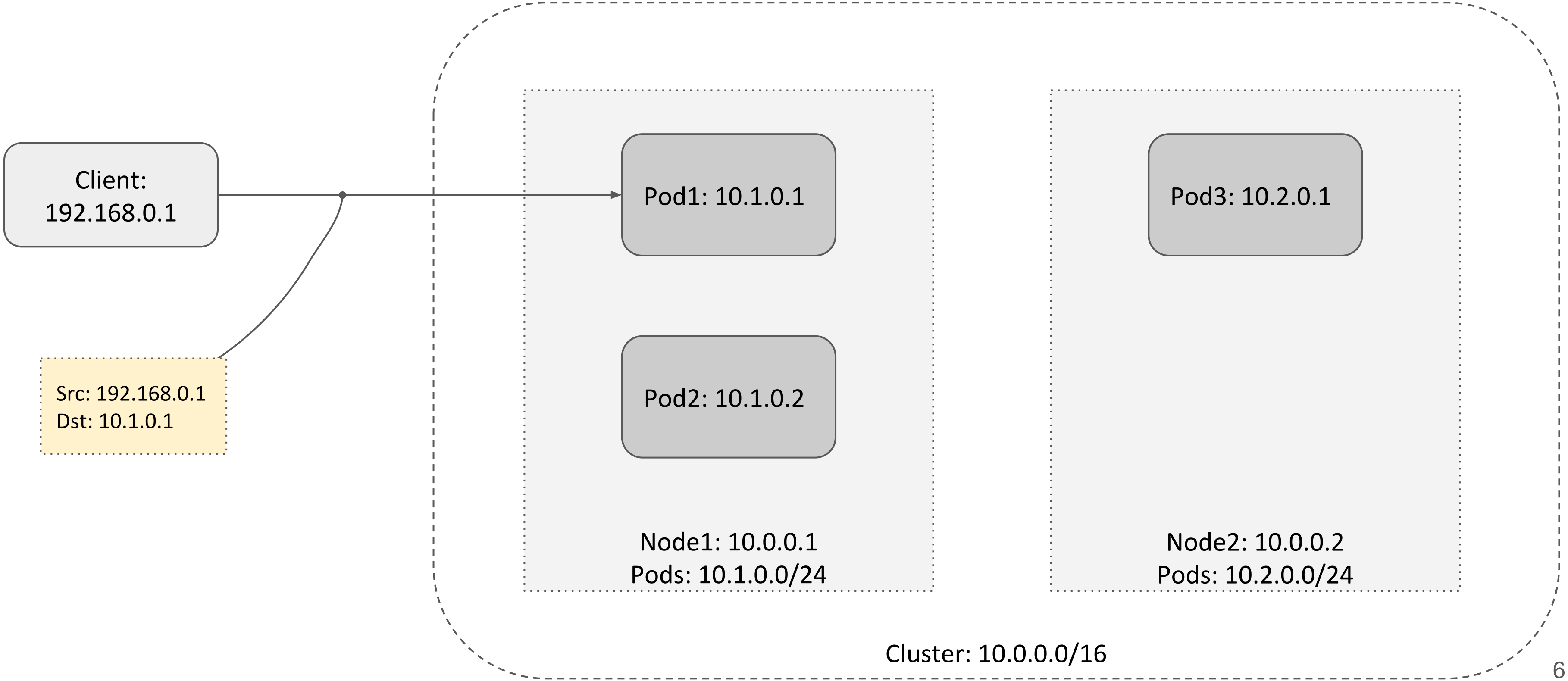
Each Pod must be reachable by its IP addr within a cluster:

- Handled by K8s CNI (e.g. Cilium)
- IP allocation/management and networking
- **Doesn’t say anything about access from outside**

K8s networking 101

1) Pod IP

K8s networking 101



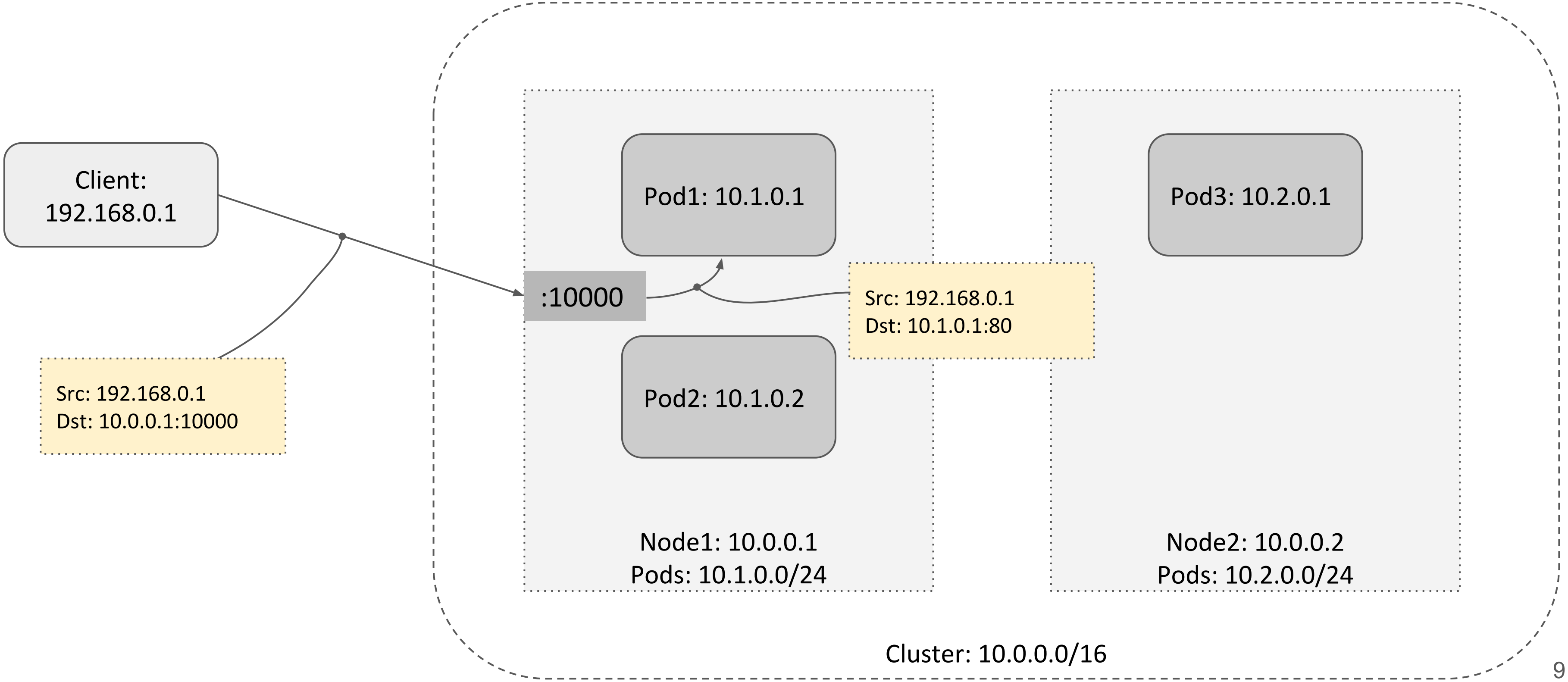
K8s networking 101

Downside: Pods come and go, no guarantee a Pod IP will ever be preserved.

K8s networking 101

2) “HostPort”

K8s networking 101



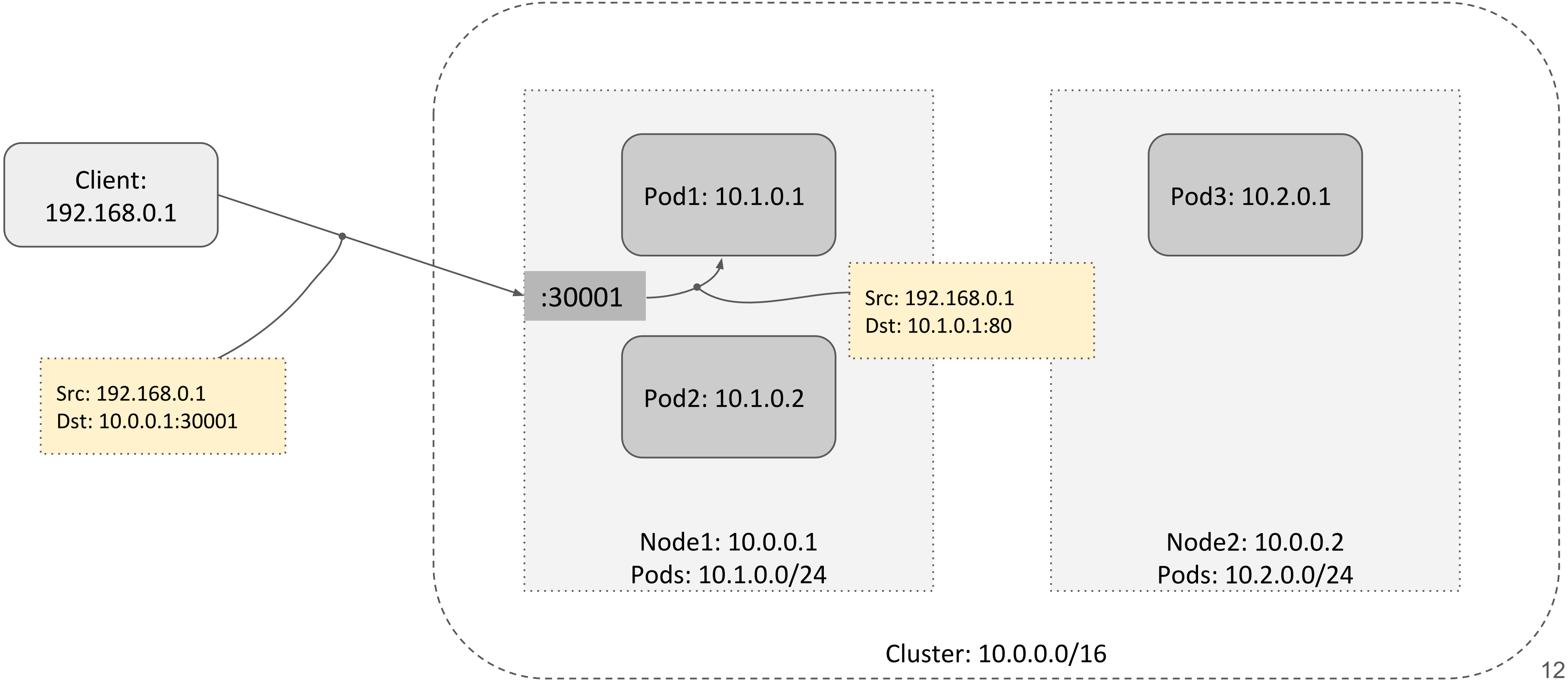
K8s networking 101

Downside: HostPort to local Pod is 1:1 mapping, that is, only 1 Pod can back-up the HostPort on a node. Use disadviced.

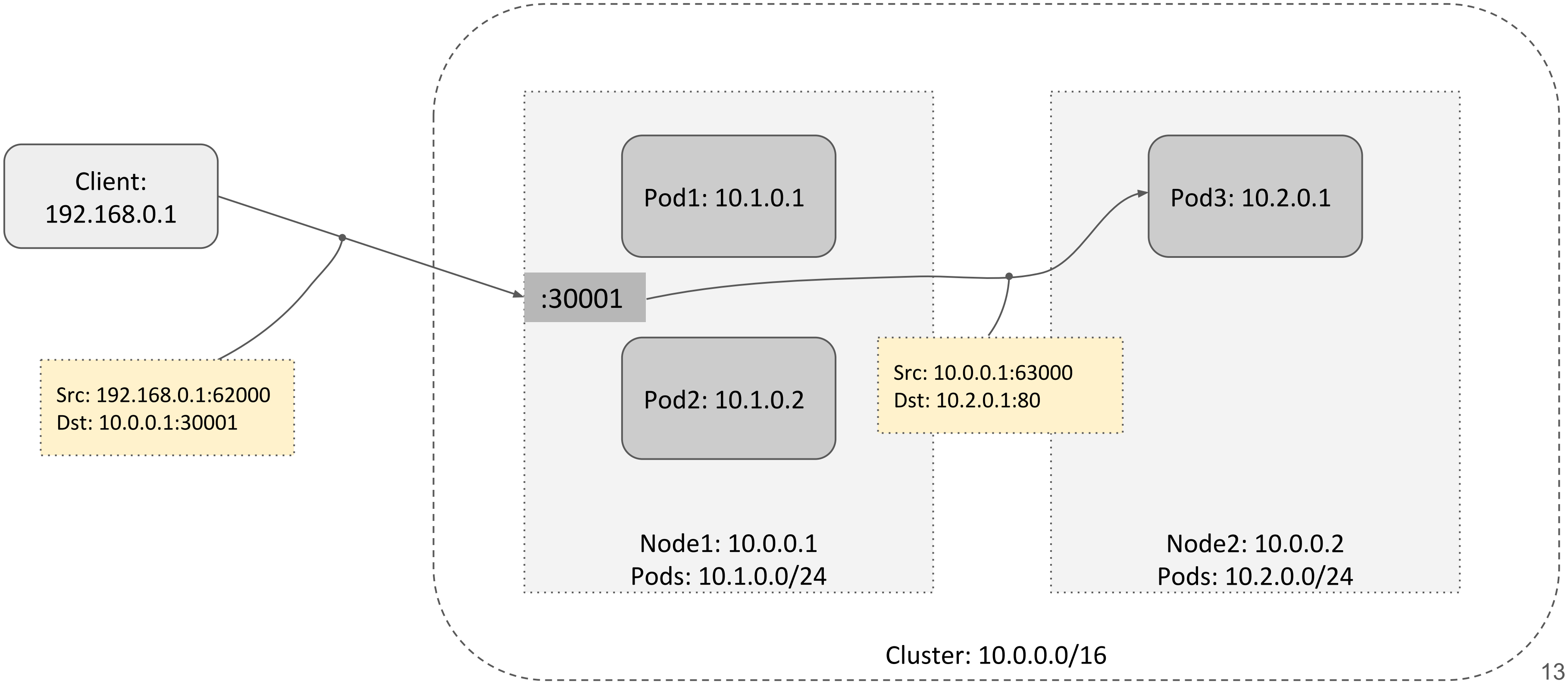
K8s networking 101

3) “NodePort service”

K8s networking 101



K8s networking 101



K8s networking 101

Advantage: Multiple Pods can back-up NodePort service. Pods can be local or remote to the node.

K8s networking 101

Advantage: Each node in the cluster reserves the same NodePort port and has same view of backends. Every node becomes a LB.

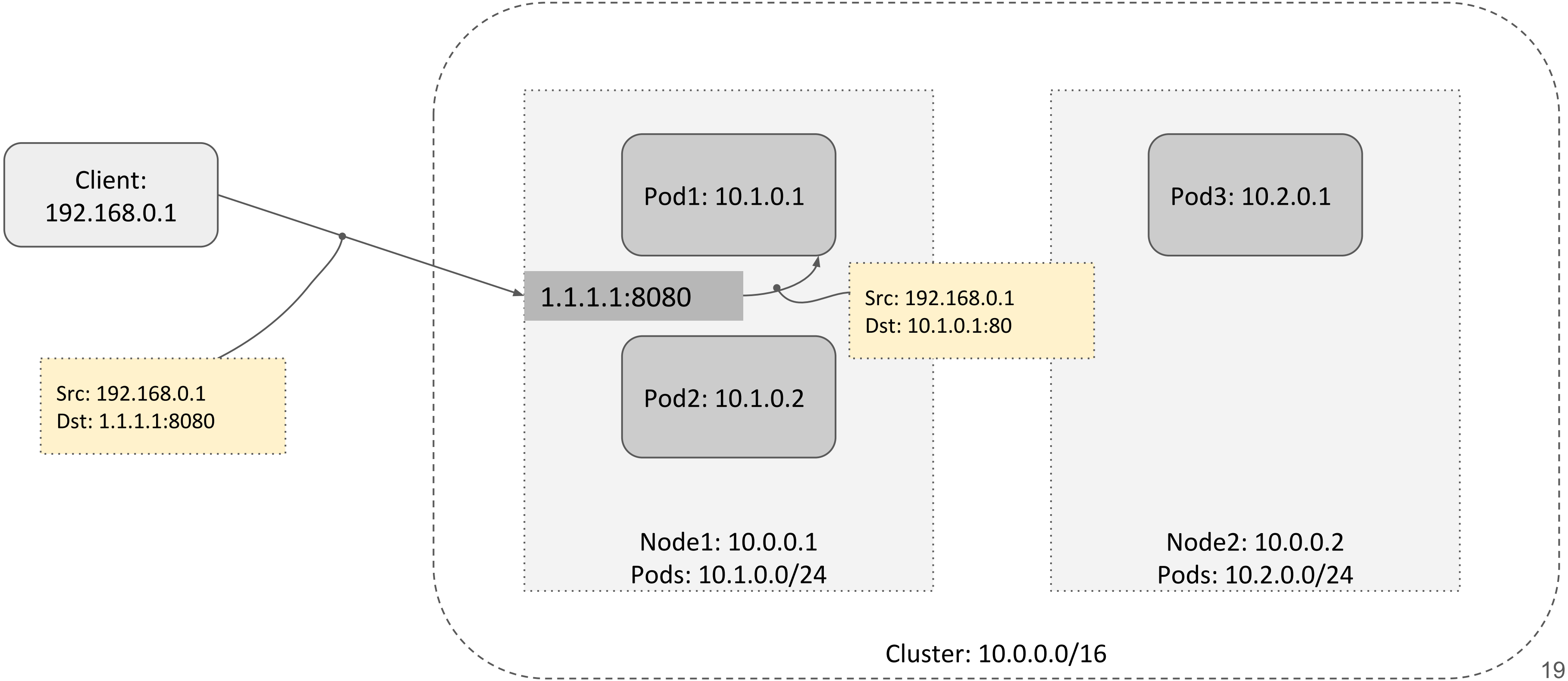
K8s networking 101

Advantage: Connectivity from host ns on every node w/o DNS through any local address, e.g. 127.0.0.1:NodePort.

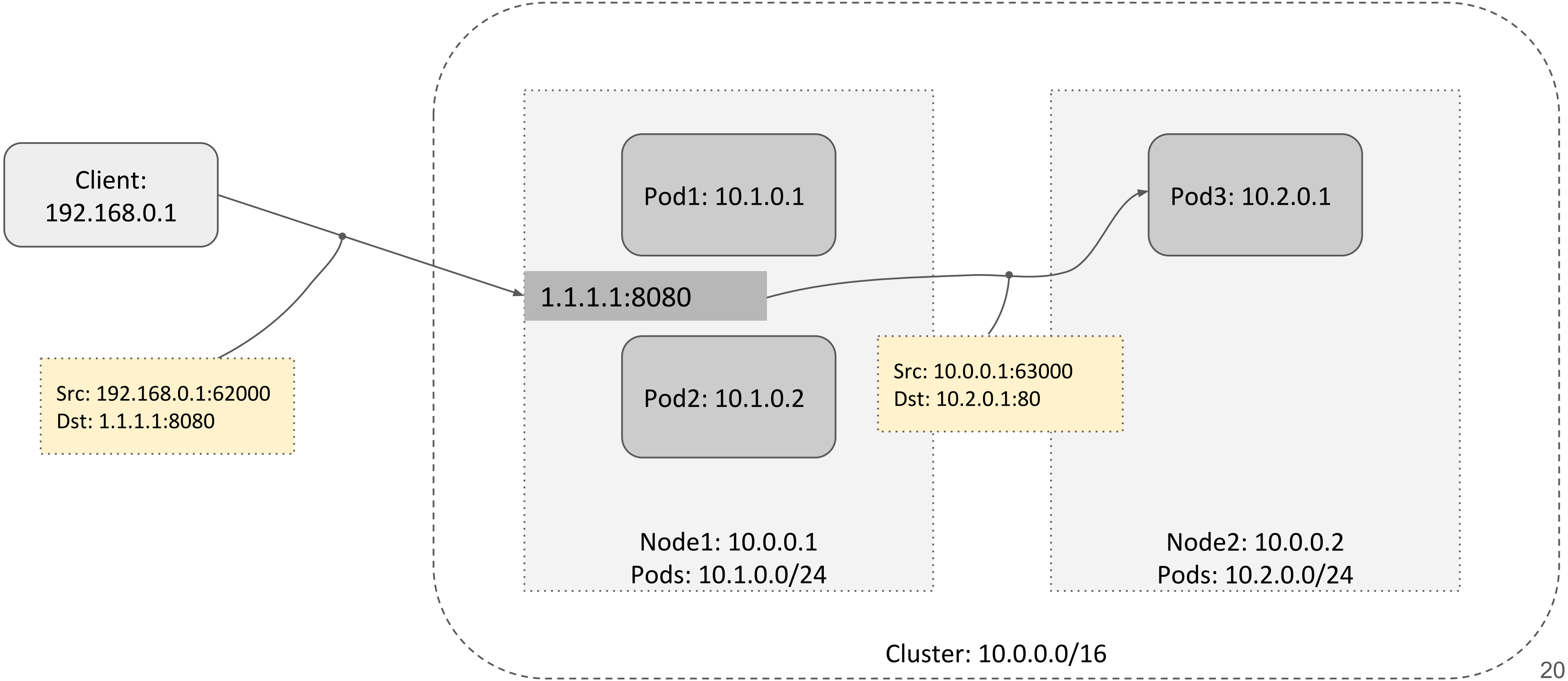
Disadvantage: SNAT-based implementations (common) hide client IP addr and introduce extra hop for replies if backend is remote.

4) “Services with external IPs”

K8s networking 101



K8s networking 101



K8s networking 101

Advantage: Can impersonate any public IP inside the cluster as long as network routes to these nodes.

K8s networking 101

Downside: External IPs are not managed by K8s. Need to be announced (e.g. via BGP) to route traffic to node.

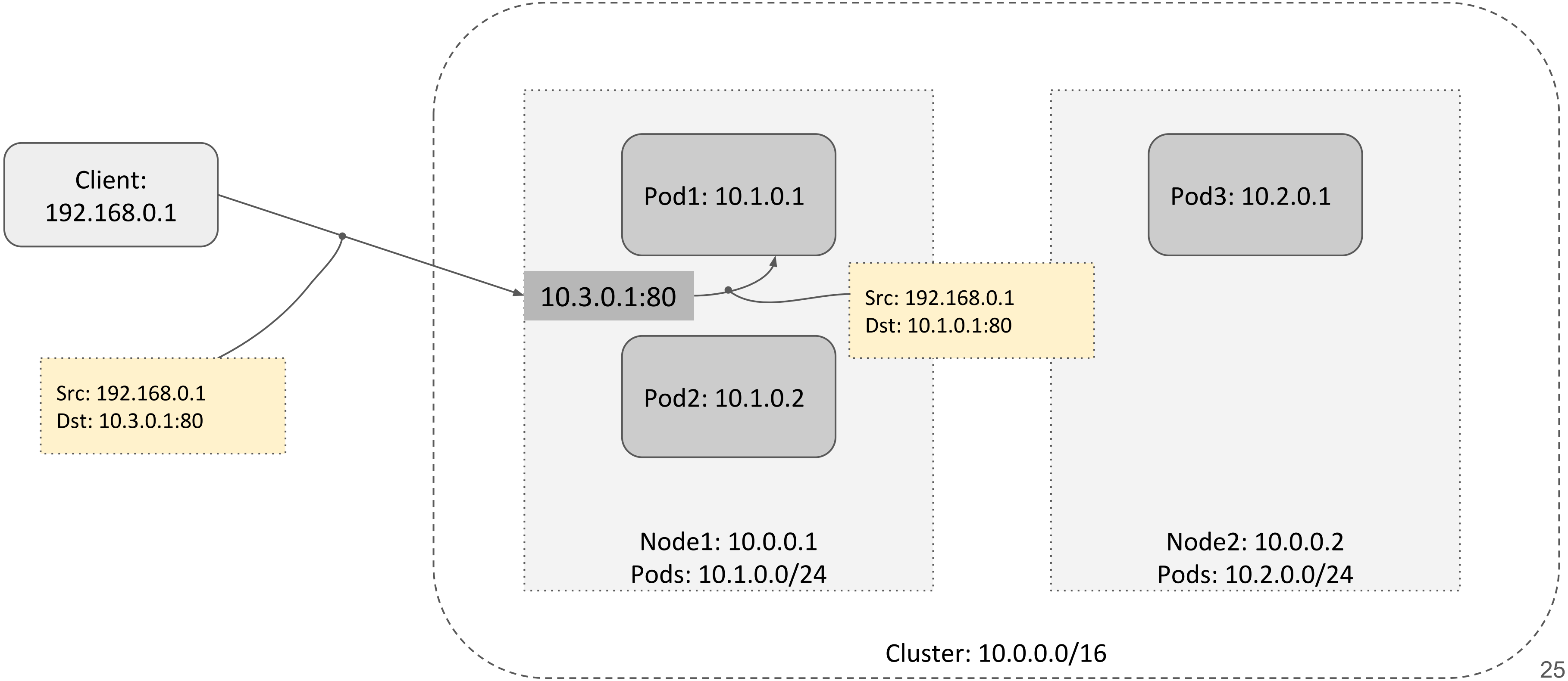
K8s networking 101

Downside: No in-cluster access to service due to potential of traffic spoofing.

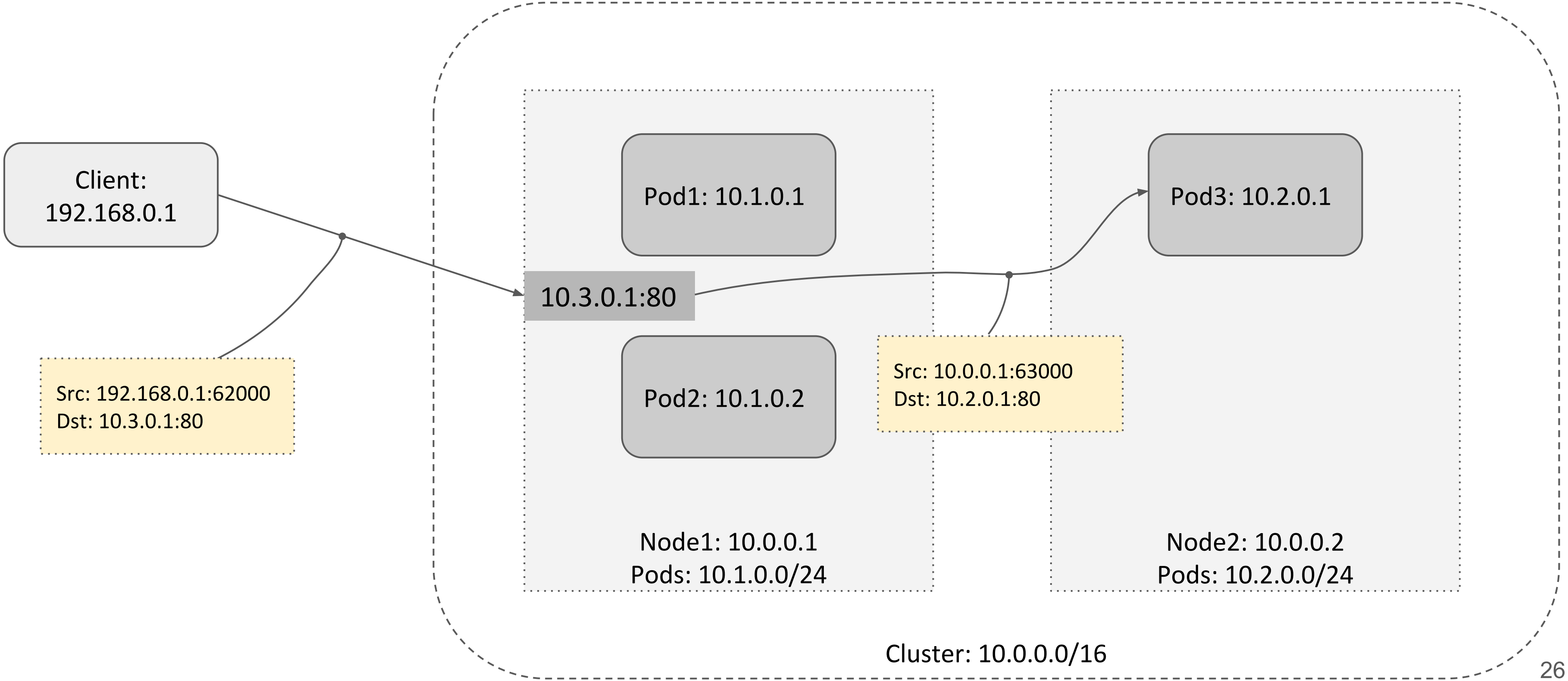
K8s networking 101

5.1) “LoadBalancer service” (on-prem)

K8s networking 101



K8s networking 101



K8s networking 101

Advantage: LoadBalancer IPs managed via K8s. Not via CNI plugin, but LoadBalancer implementation.

K8s networking 101

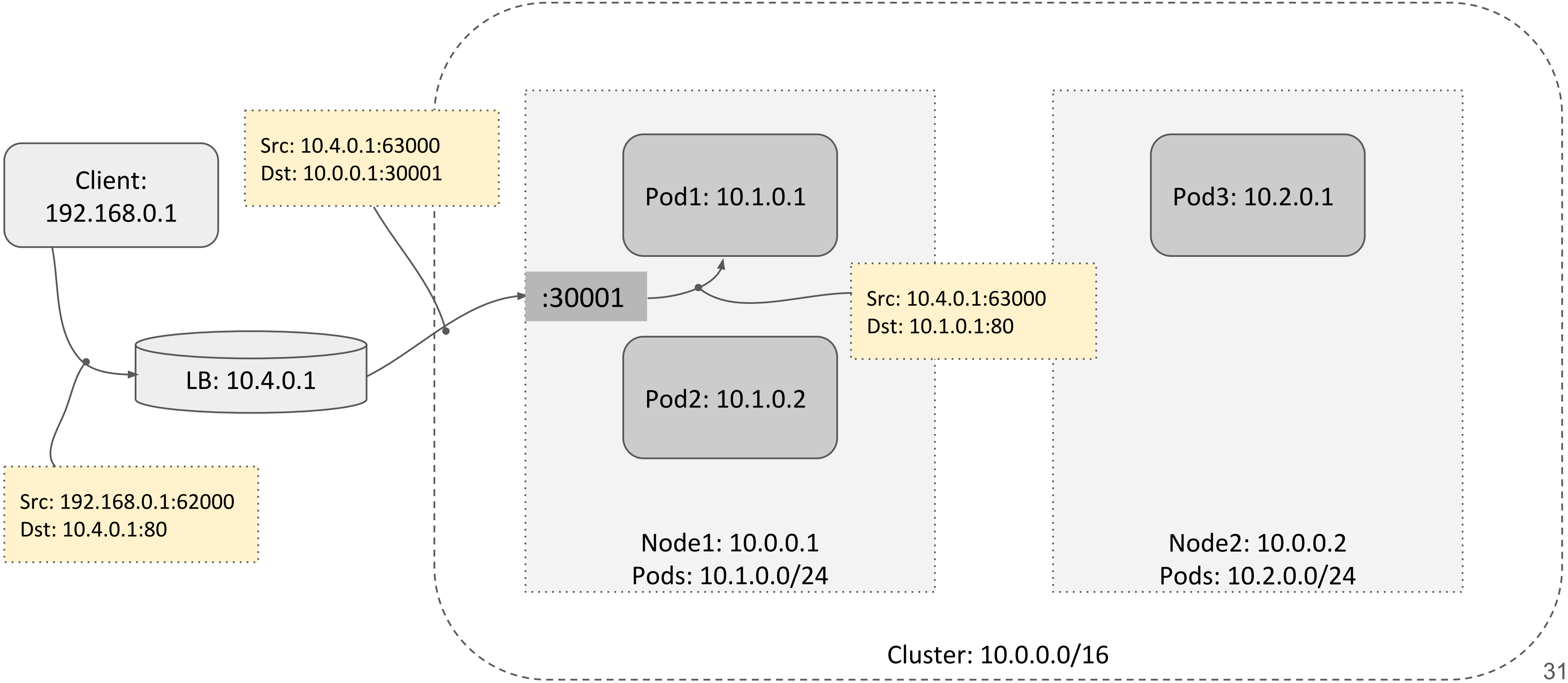
LoadBalancer implementation done by Cloud providers or MetalLB for on-prem. MetalLB can announce via ARP/NDP or BGP.

K8s networking 101

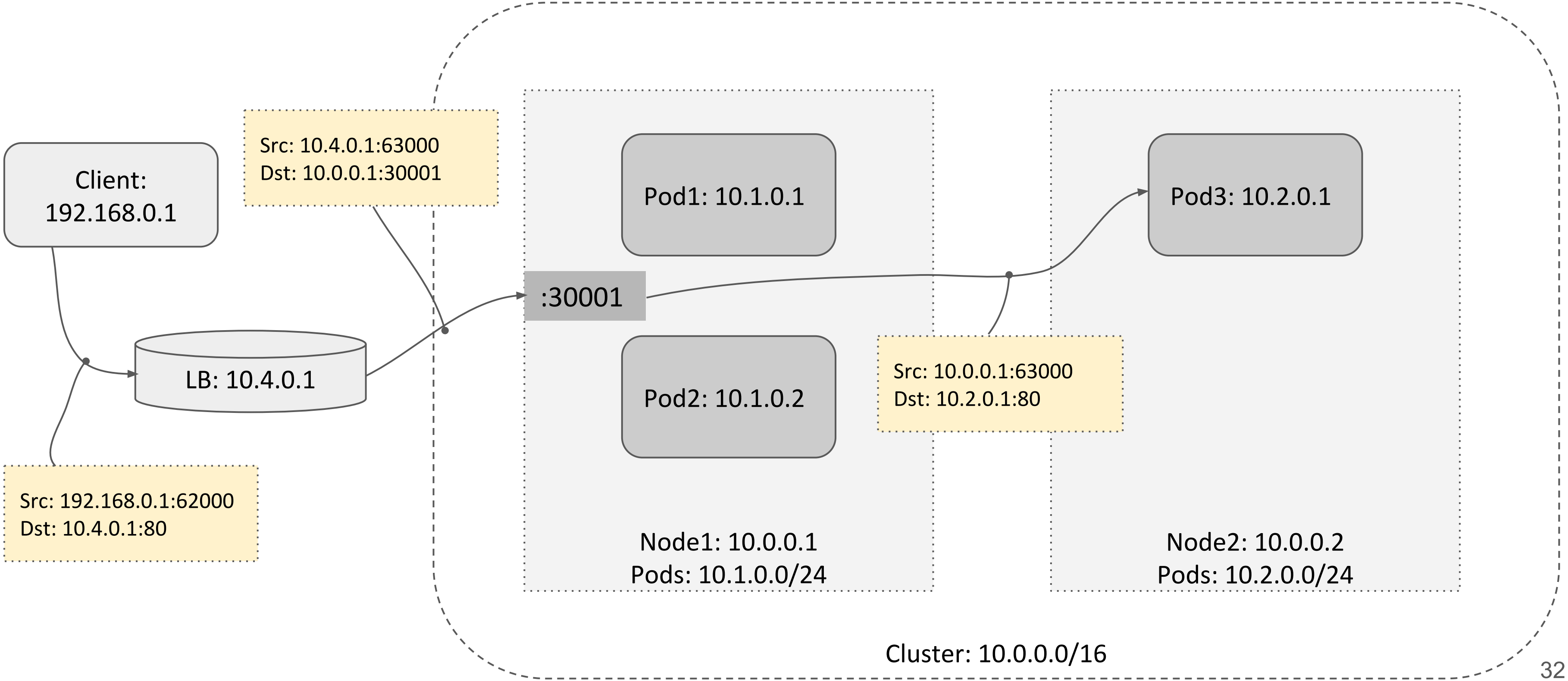
MetalLB does IP address allocation and external announcement, but does not sit in critical fast path (hence works with XDP).

5.2) “LoadBalancer service” (cloud)

K8s networking 101



K8s networking 101



K8s networking 101

Advantage: No additional user setup wrt BGP etc. All major cloud providers offer this for their managed K8s (EKS, GKE, AKS).

K8s networking 101

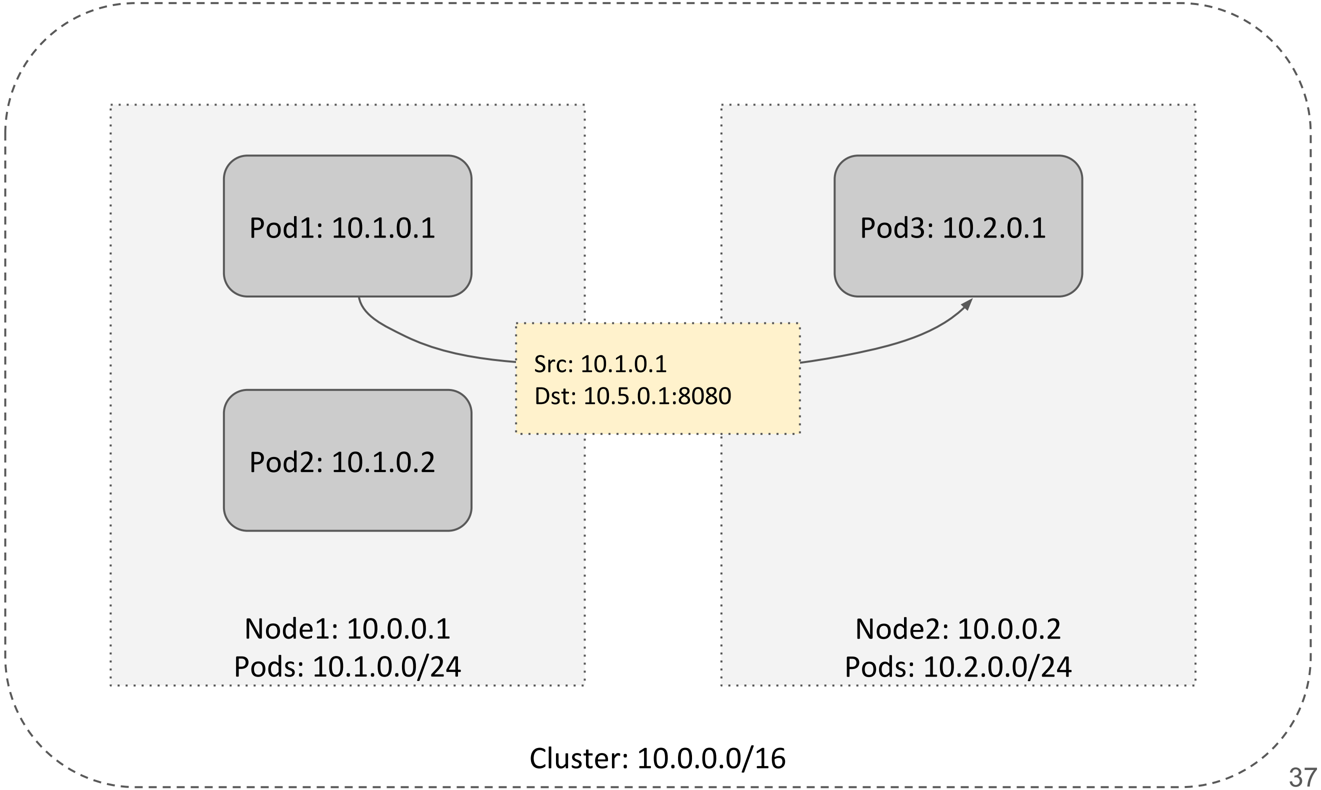
Cloud LB performs health checks to probe individual backend nodes from its LB whether they respond.

K8s networking 101

Downside: Two layers of LB, vendor specific LB annotations which are not yet standardised in K8s. Cloud LB programming time can be slow.

6) “ClusterIP service”

K8s networking 101



K8s networking 101

Dedicated IP range for ClusterIP, non-routable (always translated locally to backend). For in-cluster access only.

K8s networking 101

For one LoadBalancer service K8s creates:
LoadBalancer, NodePort, ClusterIP services
with same set of backends.

K8s networking 101

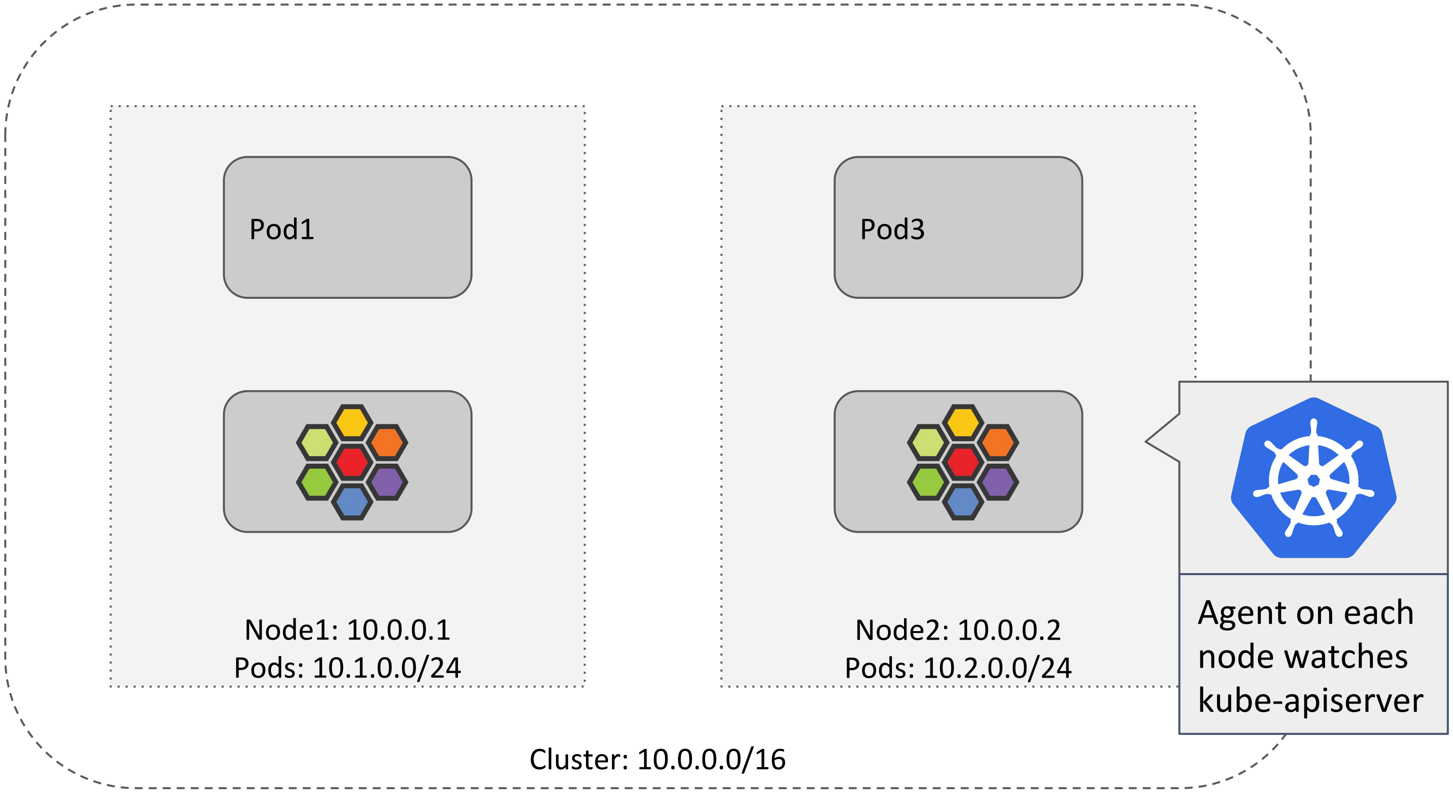
There are also various K8s features for services like sessionAffinity or externalTrafficPolicy (out of scope here).

Part 2: Cilium’s service LB & lessons learned

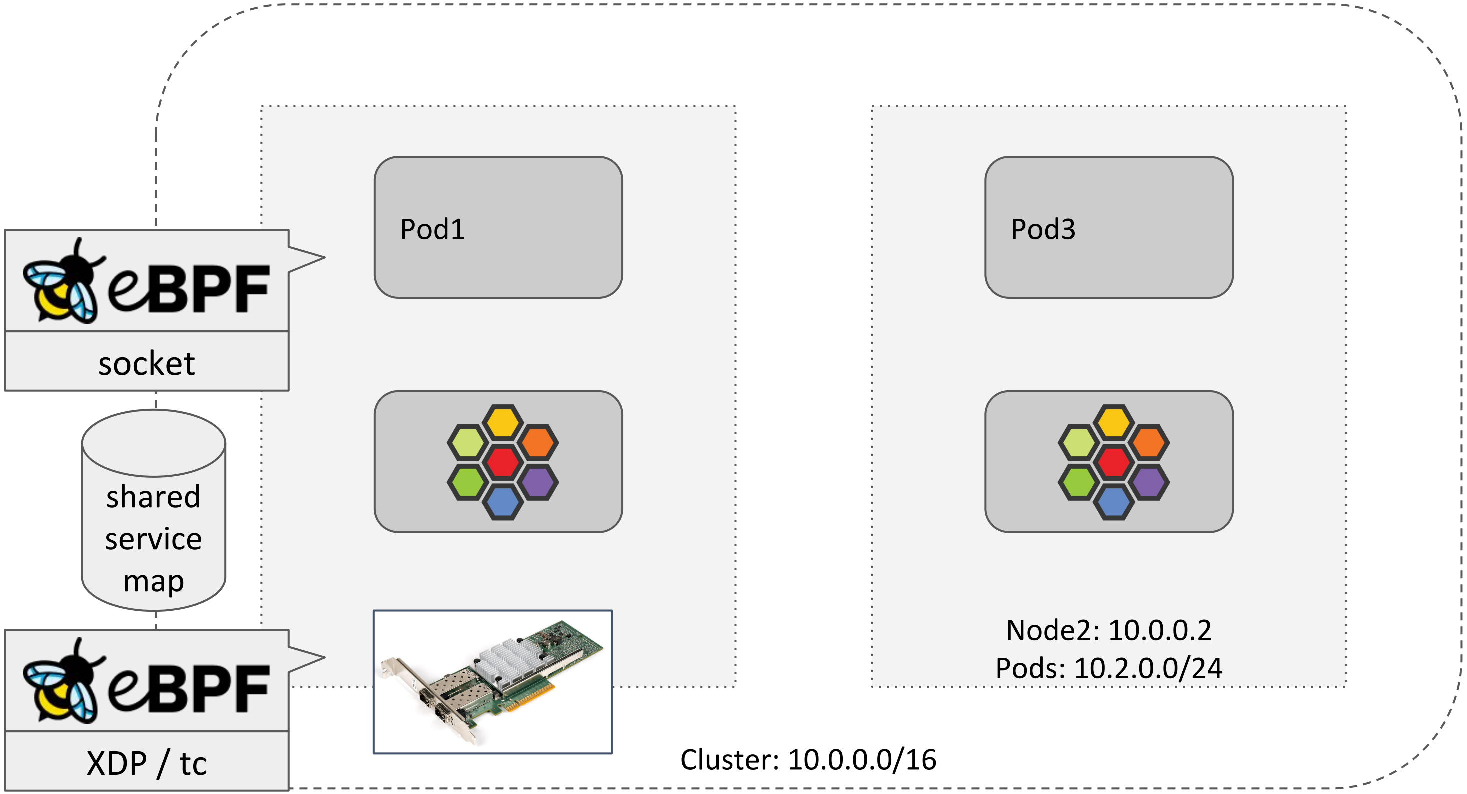
Cilium’s service LB & lessons learned

Cilium service LB implements **data path** for **all** K8s service types via **BPF**.

Cilium’s service LB & lessons learned



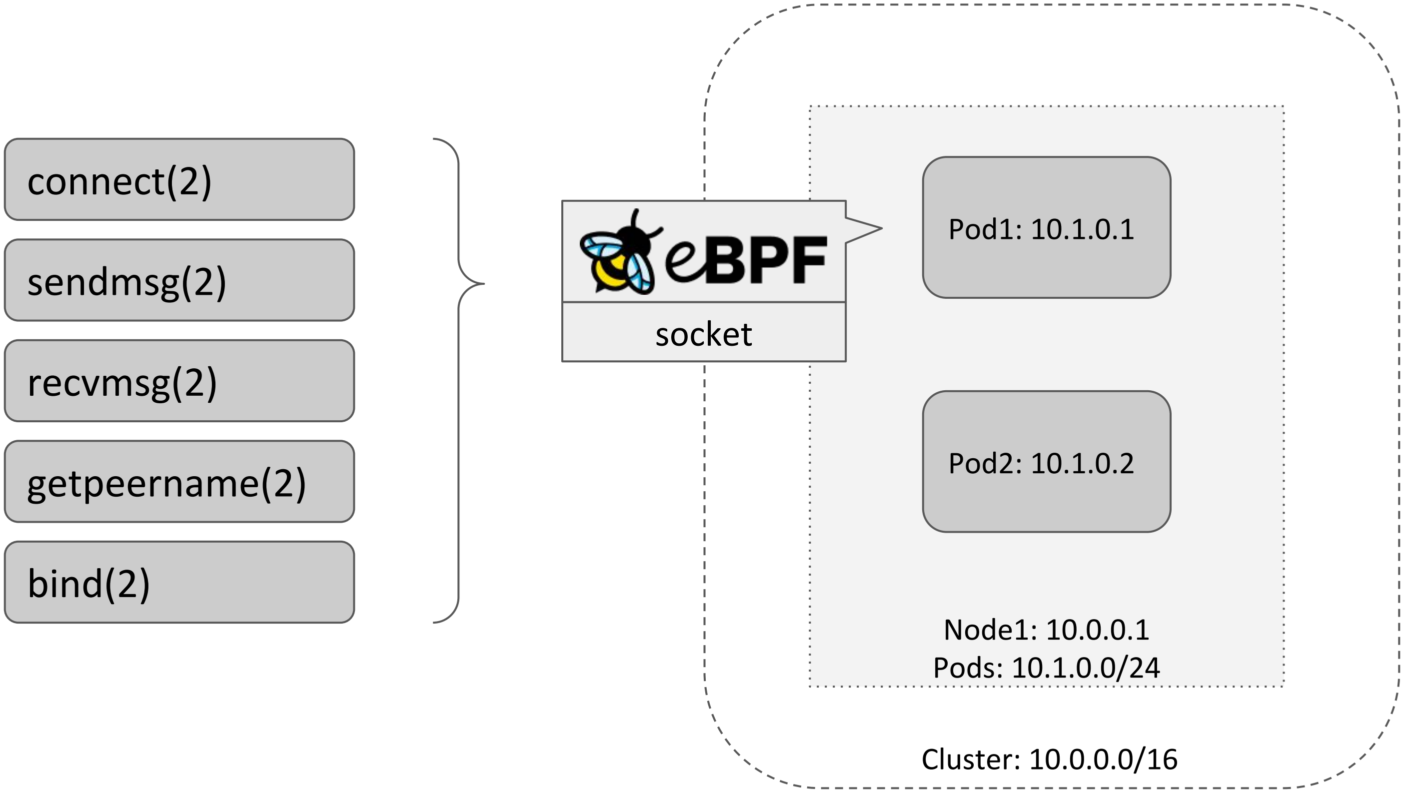
Cilium's service LB & lessons learned



Cilium’s service LB & lessons learned

1) E-W (in-cluster): BPF socket LB

Cilium’s service LB & lessons learned



K8s Pods are still cgroup v1. Cilium mounts cgroup v2, attaches BPF to root cgroup. Hybrid use works well for root v2.

Cilium’s service LB & lessons learned

connect + sendmsg BPF progs do fwd xlation of struct sockaddr. recvmsg + getpeername BPF progs rev xlation. No packet-based NAT! Done for TCP + UDP on v4, v6, v4-in-v6.

Scoped lookup in service map for sock LB vs tc / XDP in order to permit different backends depending on node internal/external traffic.

Wildcarded lookup in service map for sock LB in order to expose service on local or loopback addresses.

Approach to xlate all services on every cluster node at socket layer also faster than kube-proxy: no additional hops in network.

Cilium’s service LB & lessons learned

bind BPF prog rejects application requests from binding to NodePort.

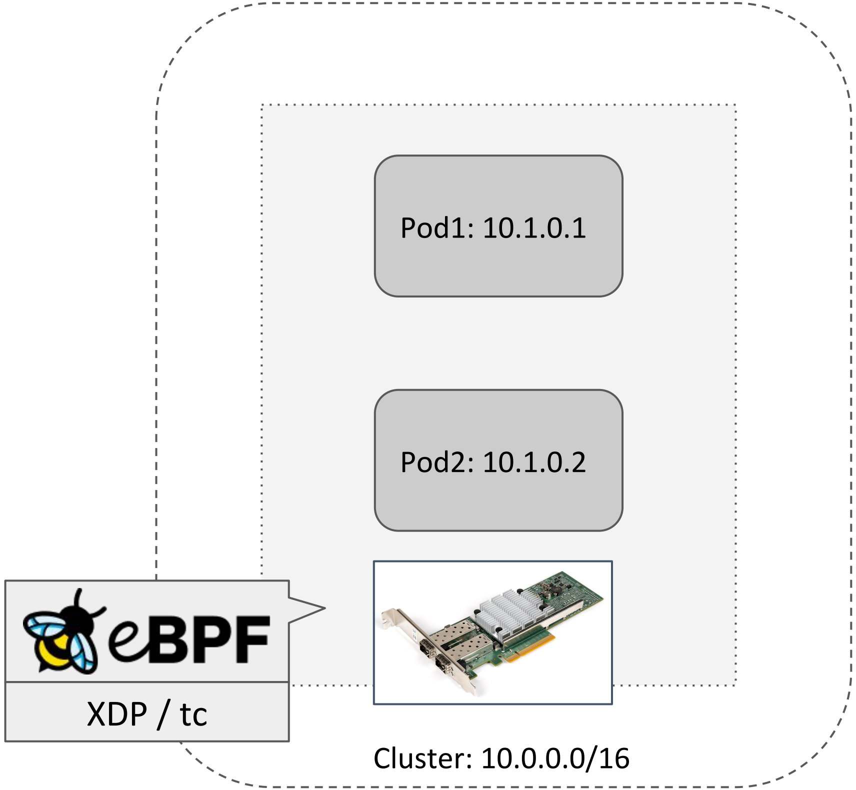
Globally unique bpf_get_socket_cookie() and bpf_get_netns_cookie() crucial helpers.

Cilium’s service LB & lessons learned

2) N-S: BPF via tc or XDP

Cilium’s service LB & lessons learned

- BPF DNAT
- BPF SNAT/DSR
- Multi-dev support
- BPF ctx abstraction



Cilium’s service LB & lessons learned

D/SNAT engine, DSR, conntrack, etc all implemented in tc BPF. For XDP support Q was whether to abstract context or reimplement ...

Ended up refactoring almost all parts of our BPF code base to make it ctx generic.

Cilium’s service LB & lessons learned

Rationale: avoids bit rot, optimizations & fixes in generic code apply to both XDP + tc.

Example, generic code:

```
if (ctx_adjust_room(ctx, 0x8, BPF_ADJ_ROOM_NET,
                    ctx_adjust_room_dsr_flags()))
        return DROP_INVALID;

if (ctx_store_bytes(ctx, ETH_HLEN + sizeof(*ip4),
                    &opt, sizeof(opt), 0) < 0)
        return DROP_INVALID;
```

skb specific implementation

xdp specific implementation

Example, context specifics:

```
#define __ctx_buff                                  xdp_md
#define __ctx_is                                    __ctx_xdp

#include "common.h"
#include "../helpers_xdp.h"
#include "../builtins.h"
#include "../section.h"
#include "../loader.h"
#include "../csum.h"

#define CTX_ACT_OK                                  XDP_PASS
#define CTX_ACT_DROP                                XDP_DROP
#define CTX_ACT_TX                                  XDP_TX  /* hairpin only */
```

Cilium’s service LB & lessons learned

Most helpers in skb context need inline equivalents for XDP. LLVM tends to optimise which then fails verifier. Inline asm as rescue.

Cilium’s service LB & lessons learned

```
static __always_inline __maybe_unused int
xdp_load_bytes(const struct xdp_md *ctx, __u64 off, void *to, const __u64 len)
{
	void *from;
	int ret;
	asm volatile("r1 = *(u32 *)(%[ctx] +0)\n\t"
		     "r2 = *(u32 *)(%[ctx] +4)\n\t"
		     "%[off] &= %[offmax]\n\t"
		     "r1 += %[off]\n\t"
		     "%[from] = r1\n\t"
		     "r1 += %[len]\n\t"
		     "if r1 > r2 goto +2\n\t"
		     "%[ret] = 0\n\t"
		     "goto +1\n\t"
		     "%[ret] = %[errno]\n\t"
		     : [ret]"=r"(ret), [from]"=r"(from)
		     : [ctx]"r"(ctx), [off]"r"(off), [len]"ri"(len),
		       [offmax]"i"(__CTX_OFF_MAX), [errno]"i"(-EINVAL)
		     : "r1", "r2");
	if (!ret)
		memcpy(to, from, len);
	return ret;
}
```

v5.6 kernel was a milestone on XDP side:
‘XDP for the masses’ on public cloud via
ena & hv_netvsc driver.

For max support on variety of drivers though, only bare minimum features must be assumed: XDP_PASS/DROP/TX

Cilium’s service LB & lessons learned

Cilium only supports native XDP on user side.

Generic XDP only utilized for CI purpose.

Reasons to avoid generic XDP two-fold: given this runs on every end node in the cluster, we cannot linearize every skb & bypass GRO.

Own optimised mem{cpy,zero,cmp,move}().

Compile error for LLVM builtin functions.

LLVM builtins end up as byte-wise ops for non-stack data. No context on efficient unaligned access.

bpf_ktime_get_ns() overhead noticeable under XDP. Made clock source selectable, switched to bpf_jiffies64(). Approx +1.1Mpps.

No cb[] for passing data between tail calls in XDP. Initially used xdp_adjust_meta().

Bad for 2 reasons: missing driver support, high rate of cache-misses. Switched to per-CPU scratch map, approx +1.2Mpps.

bpf_map_update_elem() in fast-path hitting bucket spinlock. If assumptions allow, can be converted to lock-free lookup first.

```diff
-        return map_update_elem(&LB4_REVERSE_NAT_SK_MAP, &rkey,
-                               &rval, 0);
+        tmp = map_lookup_elem(&LB4_REVERSE_NAT_SK_MAP, &rkey);
+        if (!tmp || memcmp(tmp, &rval, sizeof(rval)))
+                ret = map_update_elem(&LB4_REVERSE_NAT_SK_MAP, &rkey,
+                                      &rval, 0);
+        return ret;
```

Cilium’s service LB & lessons learned

bpf_fib_lookup() expensive, can be avoided and compiled out e.g. for hairpin LB. Approx +1.5Mpps in our test env.

Also don’t gamble with LLVM & enforce BPF’s tail call patching via text_poke for static slots.

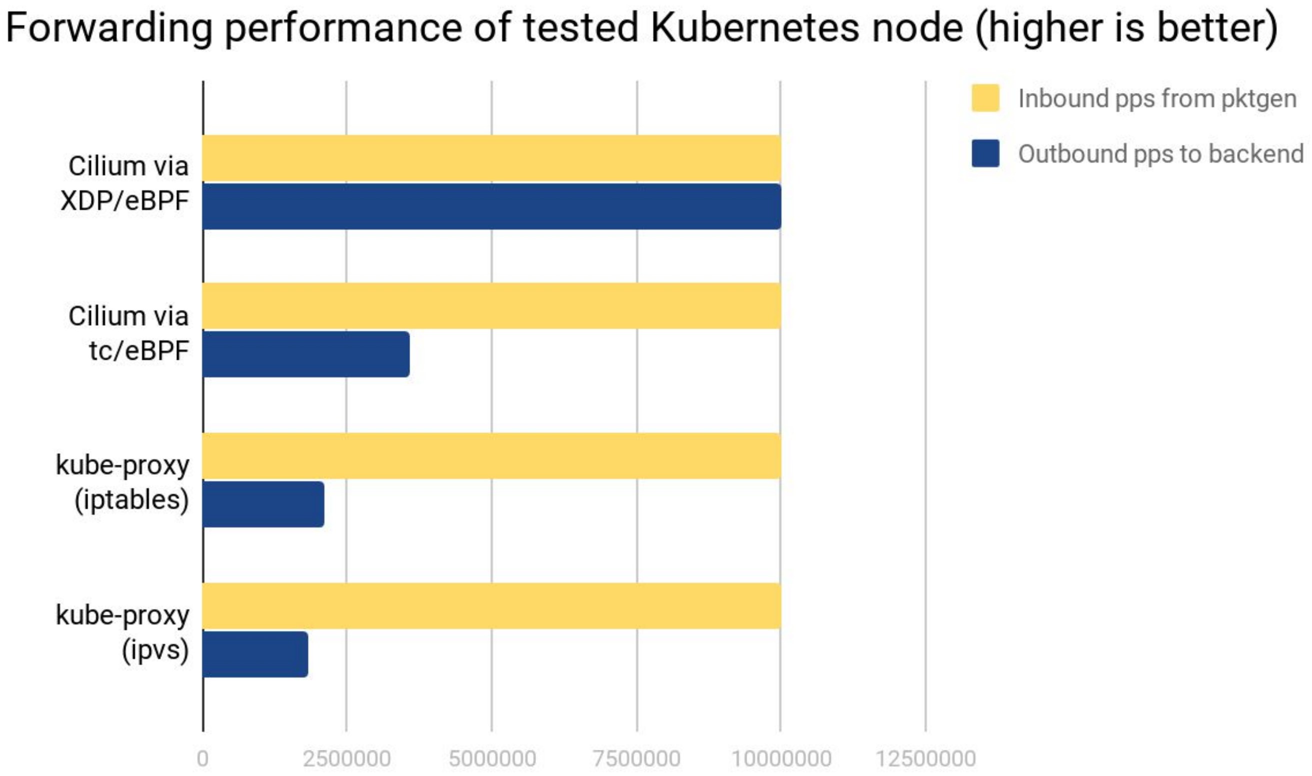
Cilium’s service LB & lessons learned

```
static __always_inline __maybe_unused void
tail_call_static(const struct __ctx_buff *ctx, const void *map,
                 const __u32 slot)
{
        if (!__builtin_constant_p(slot))
                __throw_build_bug();
        asm volatile("r1 = %[ctx]\n\t"
                     "r2 = %[map]\n\t"
                     "r3 = %[slot]\n\t"
                     "call 12\n\t"
                     :: [ctx]"r"(ctx), [map]"r"(map), [slot]"i"(slot)
                     : "r0", "r1", "r2", "r3", "r4", "r5");
}
```

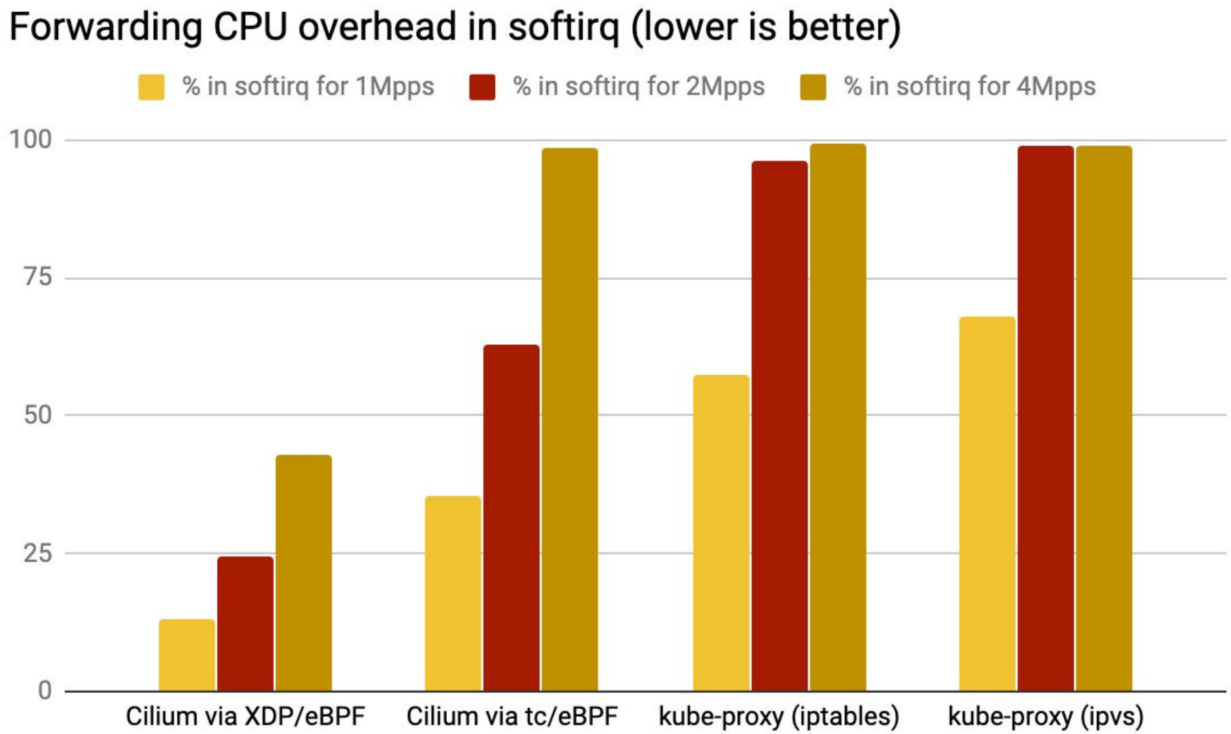
Cilium’s service LB & lessons learned

pktgen hairpin test on XDP layer for remote K8s service backend. 10Mpps inbound ...

Cilium's service LB & lessons learned

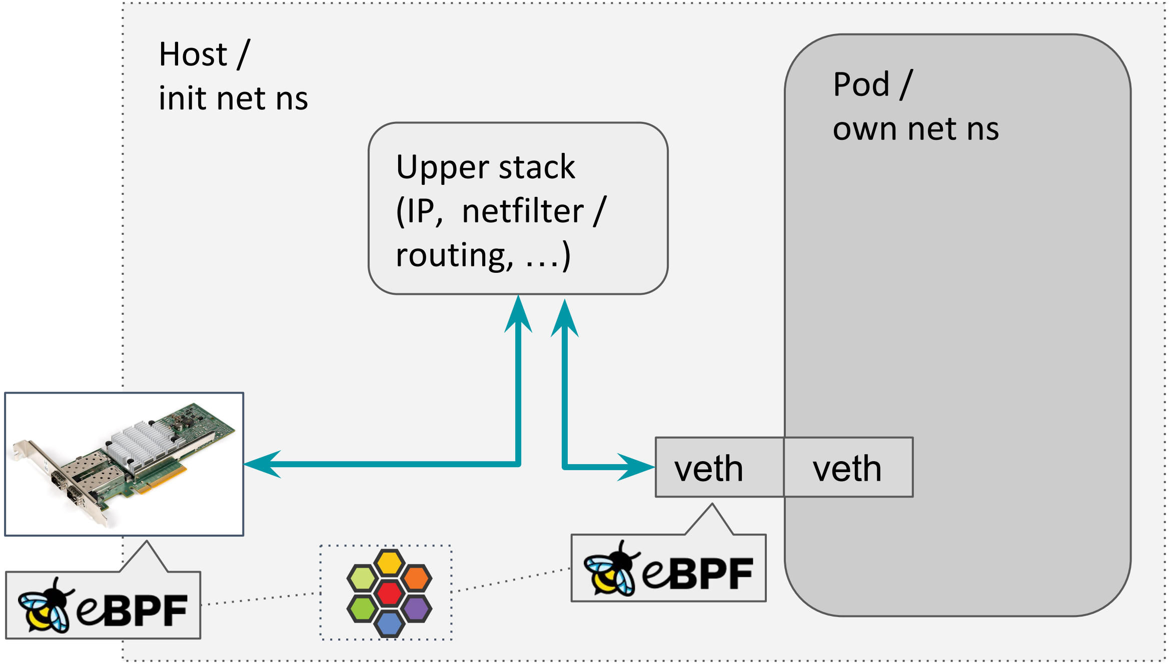


Cilium's service LB & lessons learned



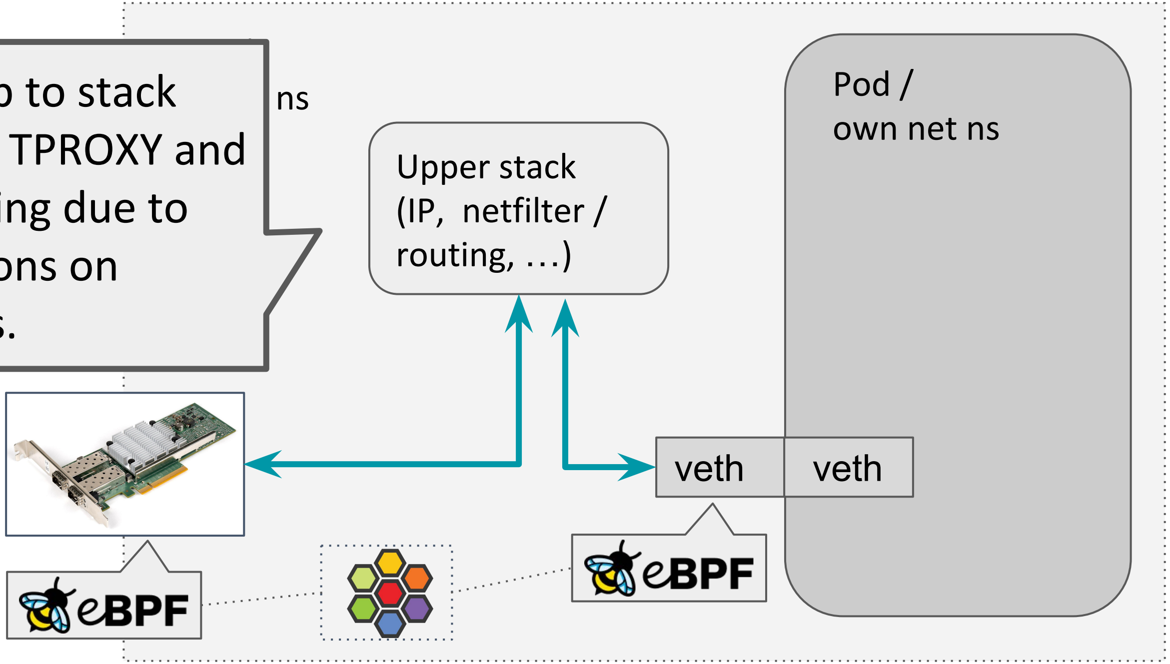
Part 3: New BPF kernel extensions (for Cilium use case)

New BPF kernel extensions



New BPF kernel extensions

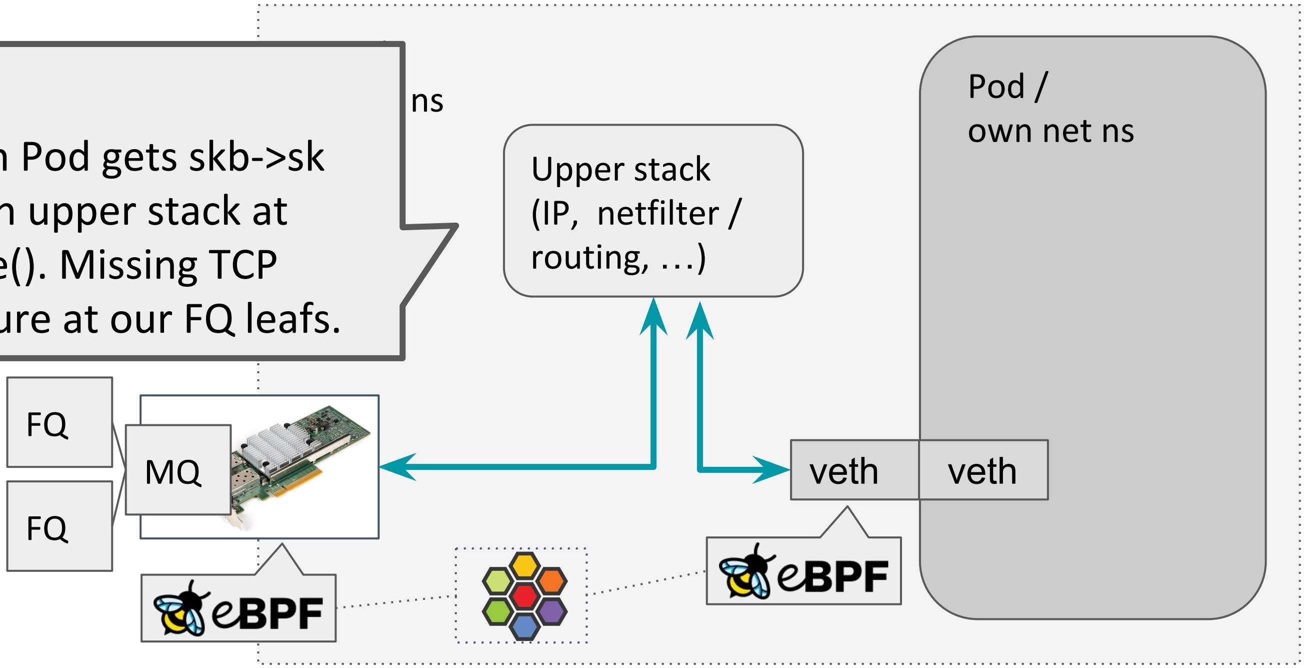
Packets passed up to stack due to need for i) TPROXY and ii) netfilter dropping due to 'invalid' connections on asymmetric paths.



New BPF kernel extensions

Issues:

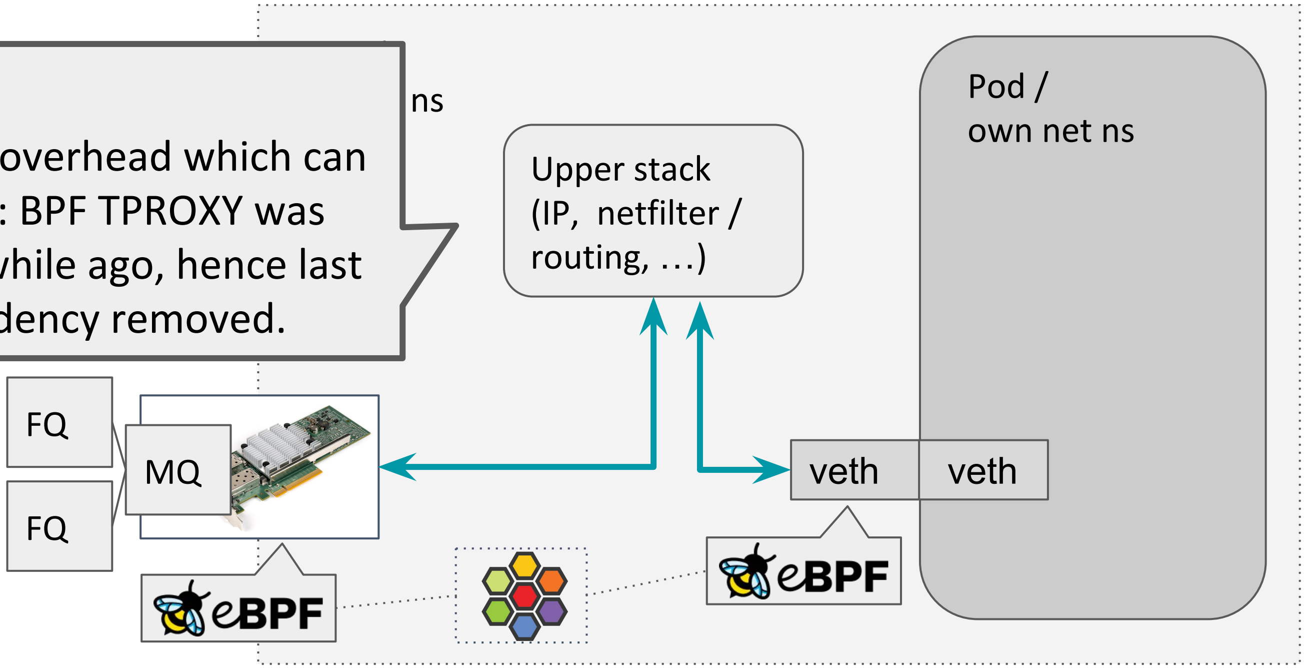
Traffic from Pod gets skb->sk orphaned in upper stack at ip_rcv_core(). Missing TCP back-pressure at our FQ leafs.



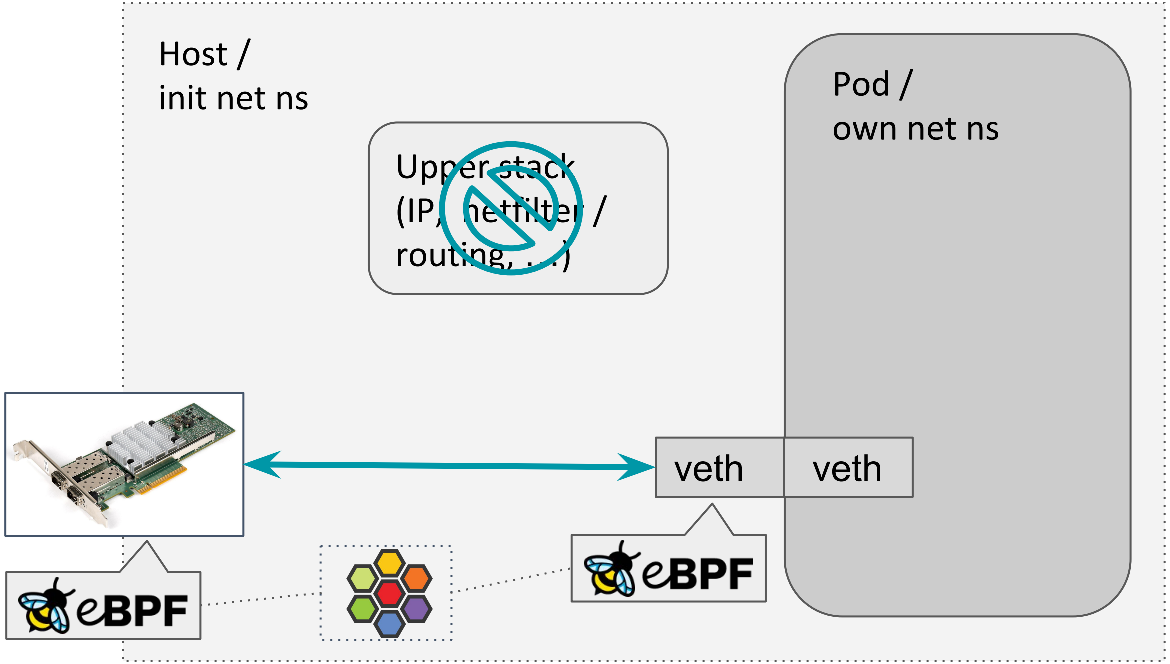
New BPF kernel extensions

Issues:

Per packet overhead which can be avoided: BPF TPROXY was merged a while ago, hence last real dependency removed.



New BPF kernel extensions



New BPF kernel extensions

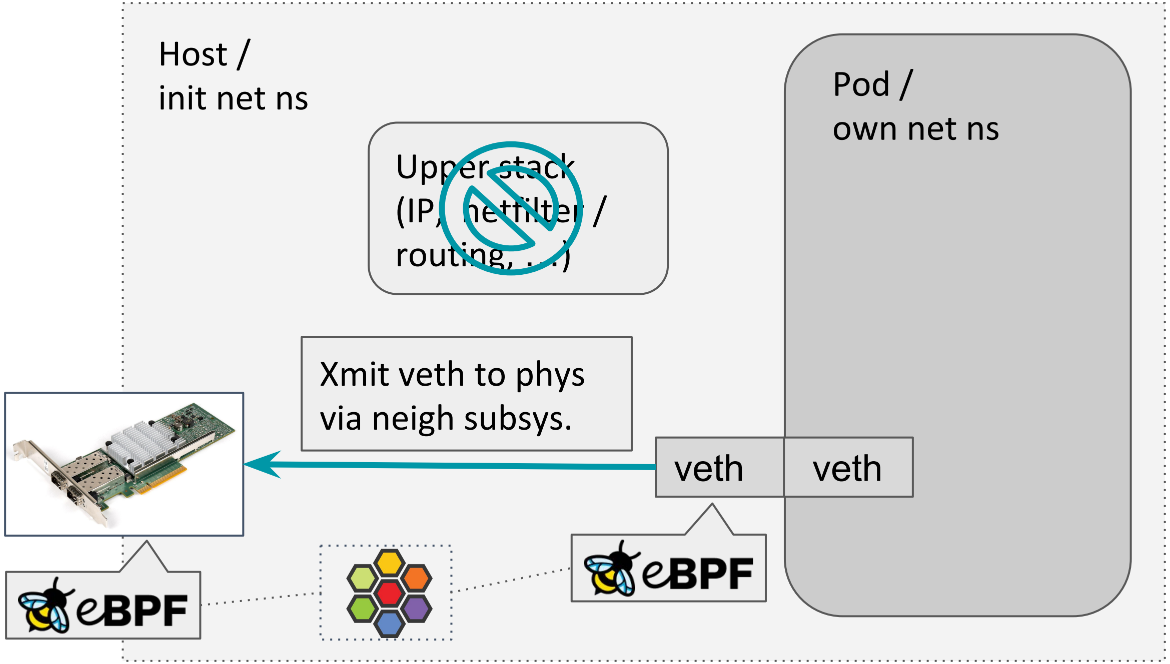
2 new BPF helpers for tc:

bpf_redirect_neigh() and

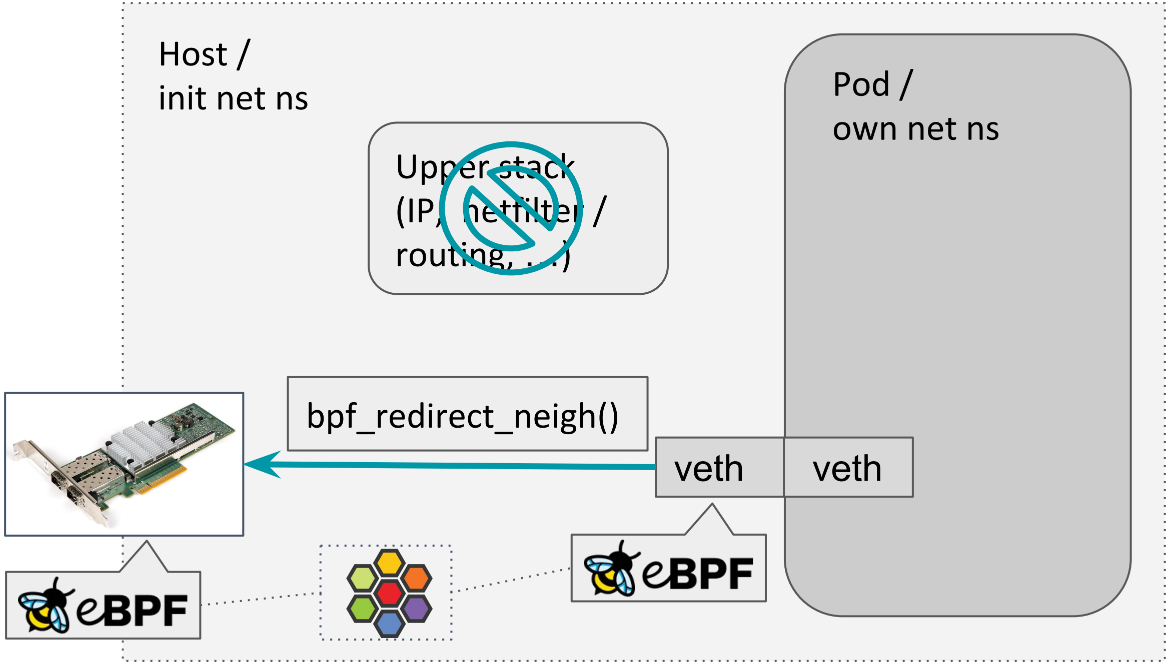
bpf_redirect_peer()...

borrowing ideas from ipvlan.

New BPF kernel extensions



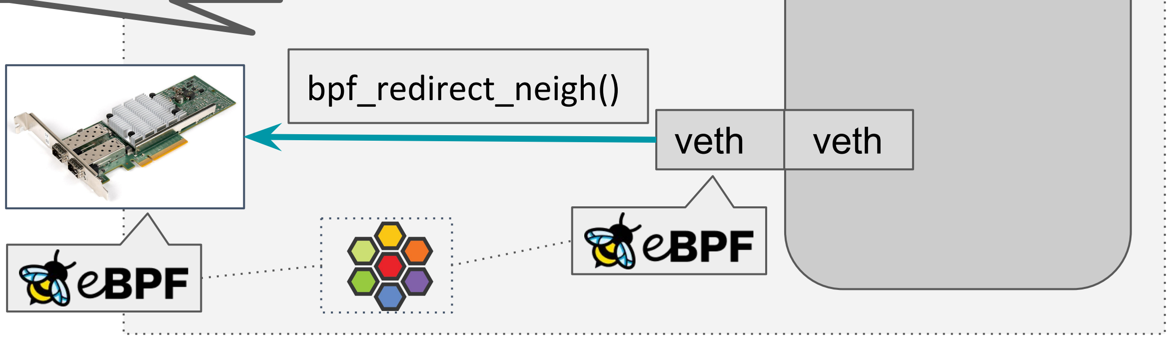
New BPF kernel extensions



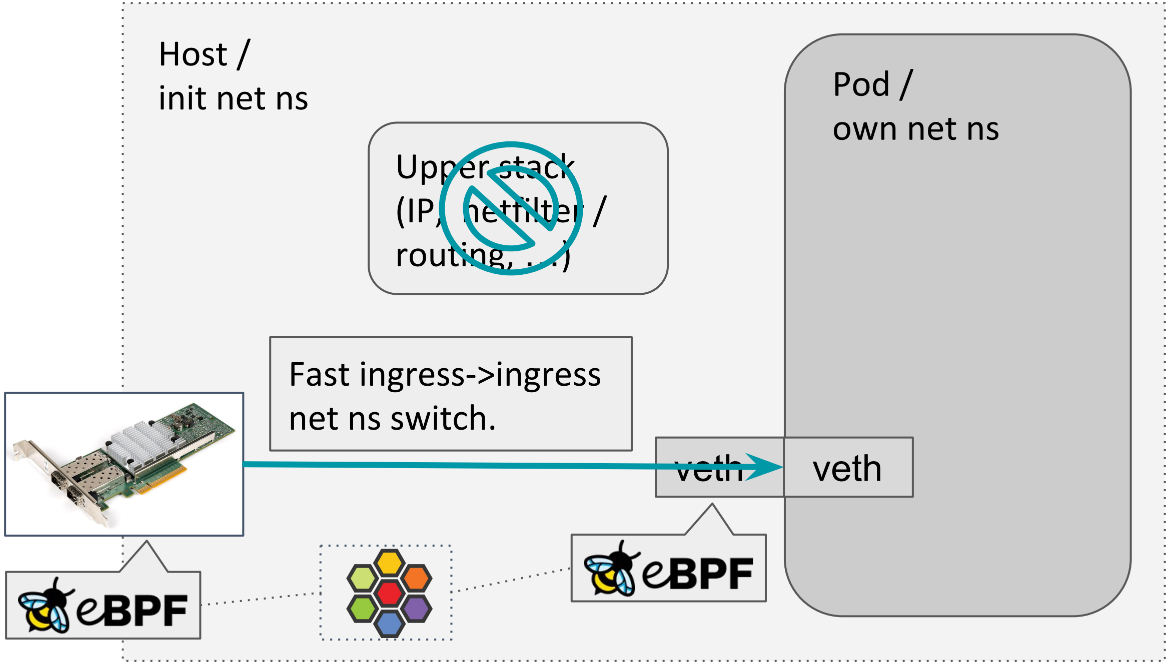
New BPF kernel extensions

PoC currently doing internally:

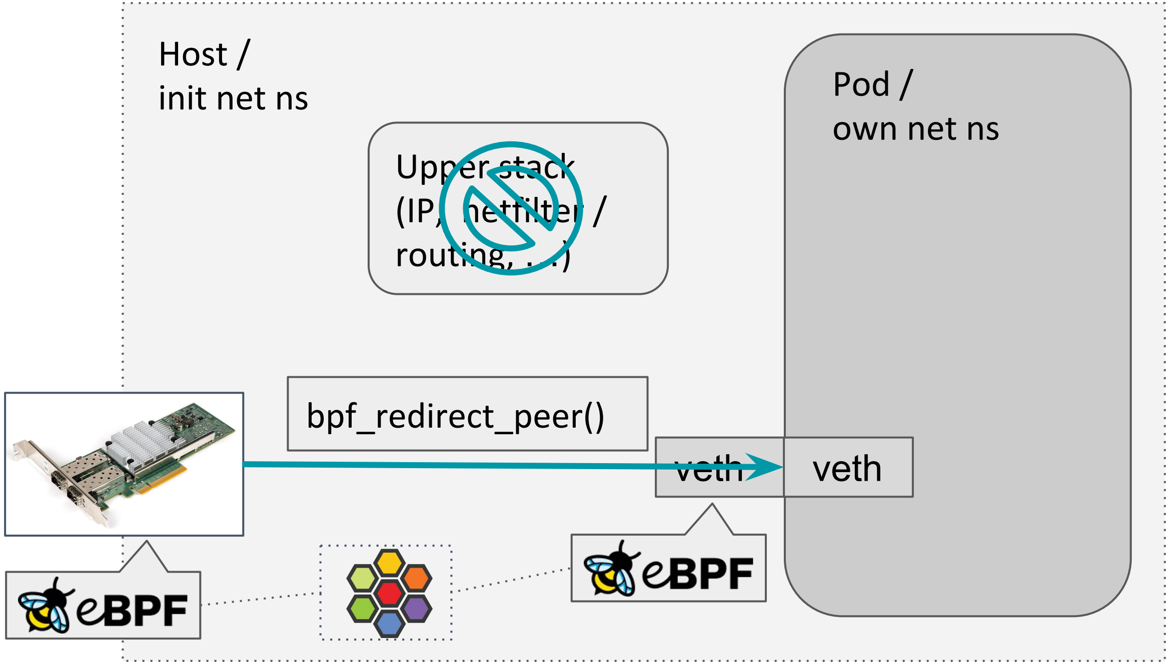
ip_route_output_flow()
skb_dst_set()
ip_finish_output2()
- fills in neigh info
- retains skb->sk till Qdisc on phys



New BPF kernel extensions



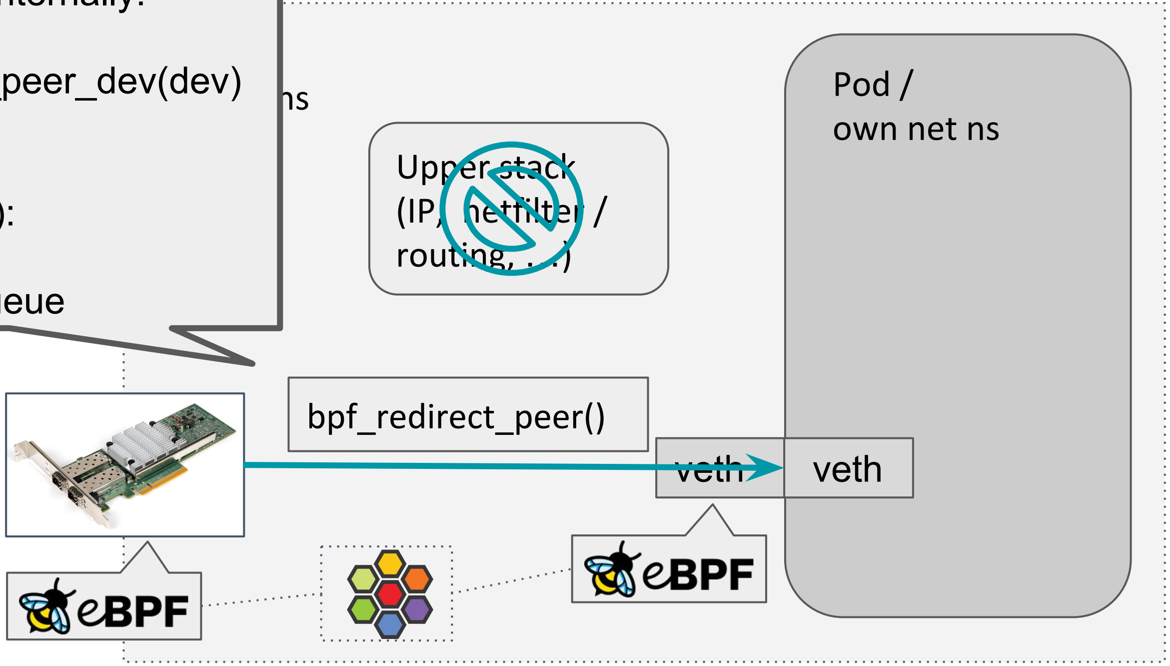
New BPF kernel extensions



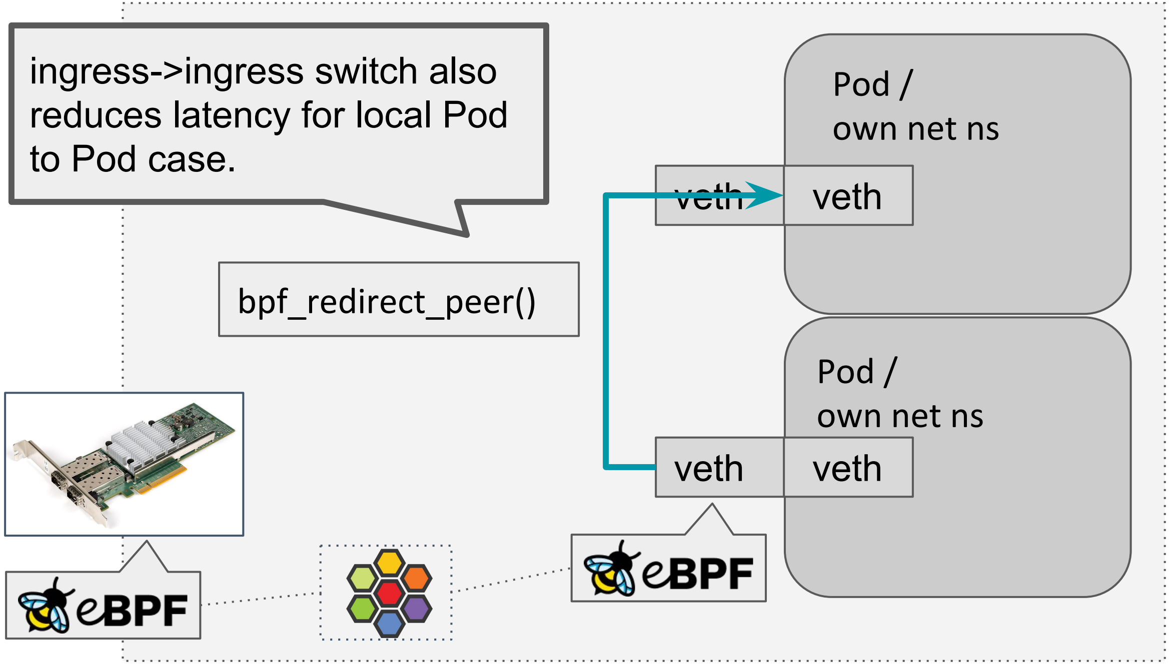
New BPF kernel extensions

PoC currently doing internally:

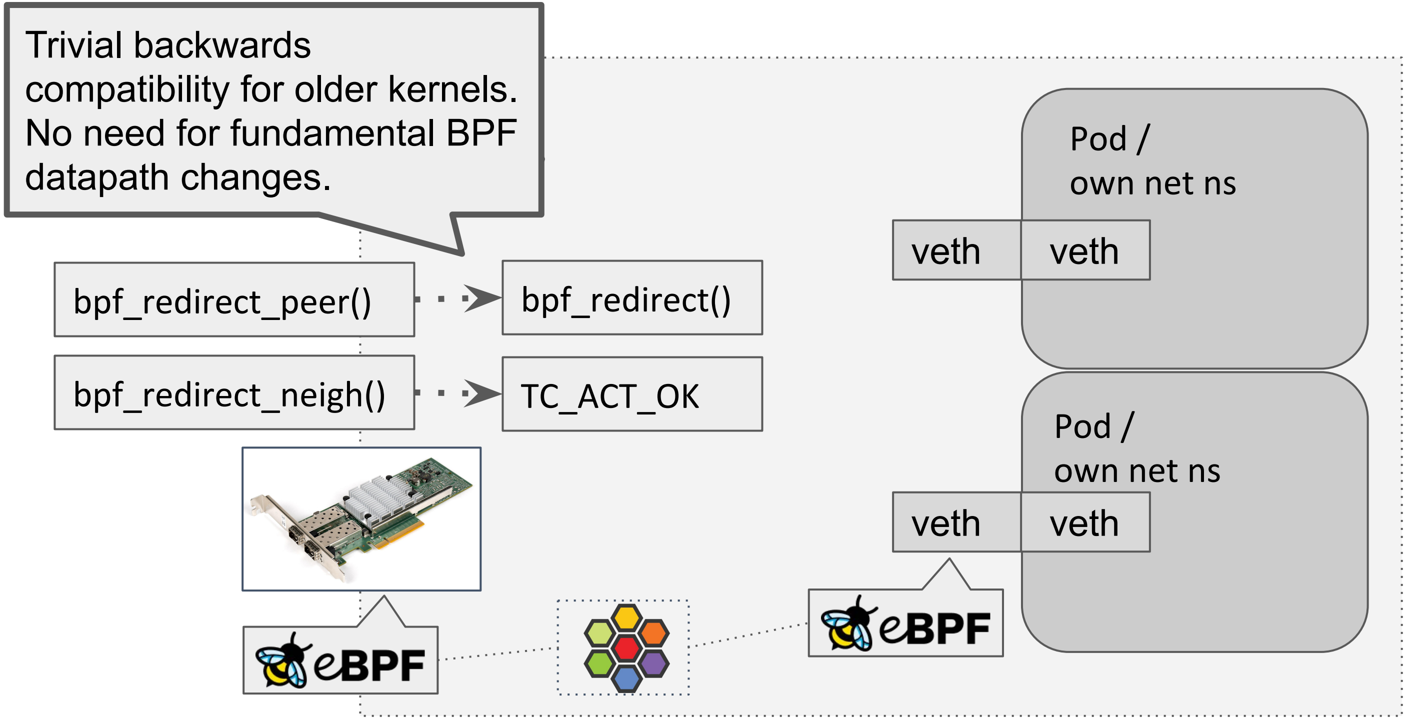
dev = ops->ndo_get_peer_dev(dev)
skb_scrub_packet()
skb->dev = dev
sch_handle_ingress():
- goto another_round
 - no CPU backlog queue



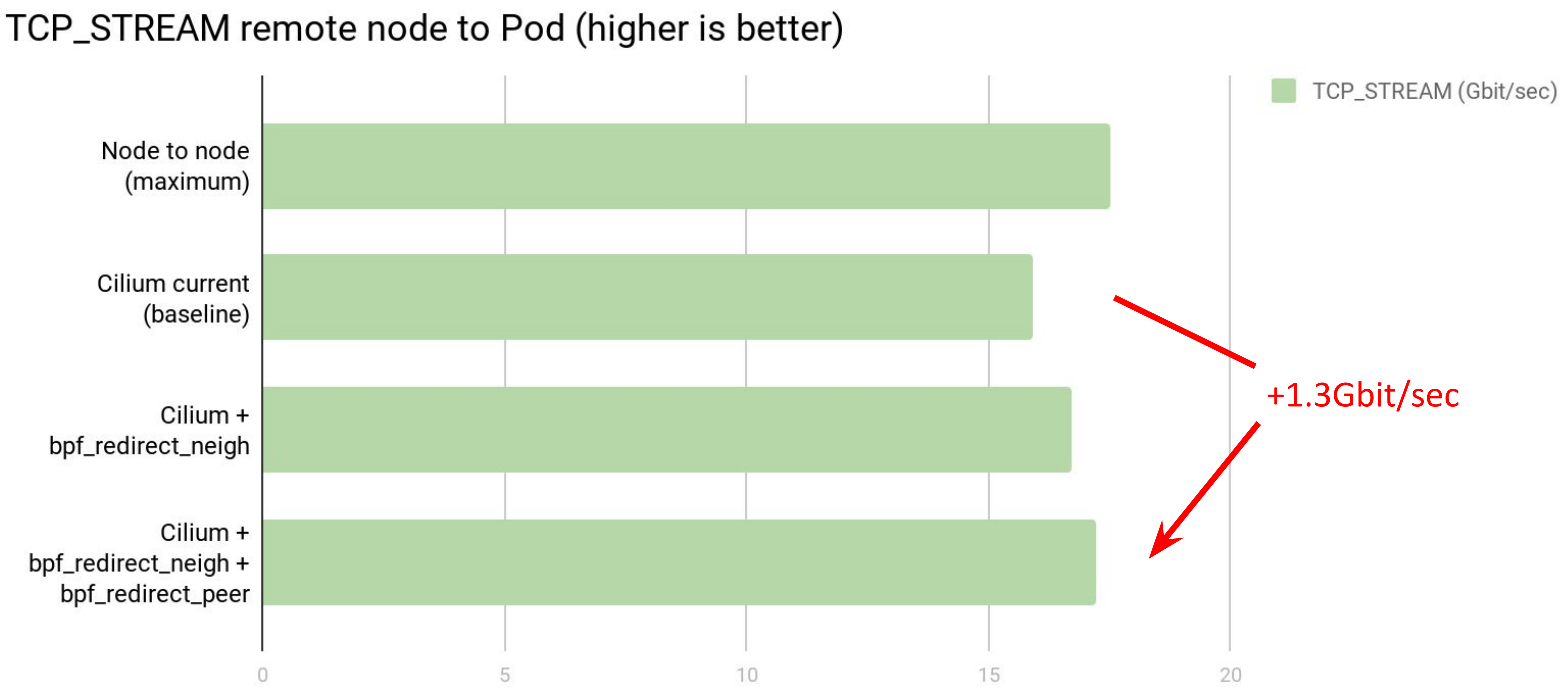
New BPF kernel extensions



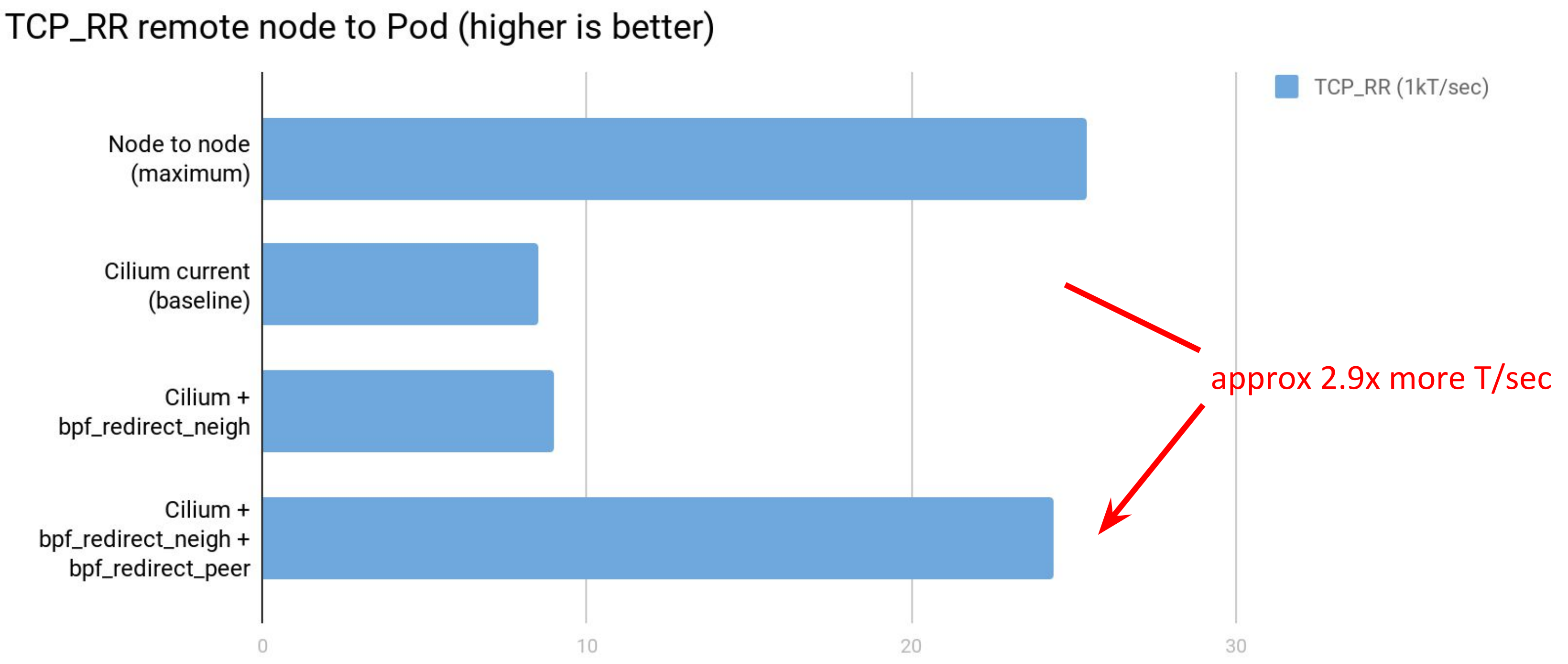
New BPF kernel extensions



New BPF kernel extensions



New BPF kernel extensions



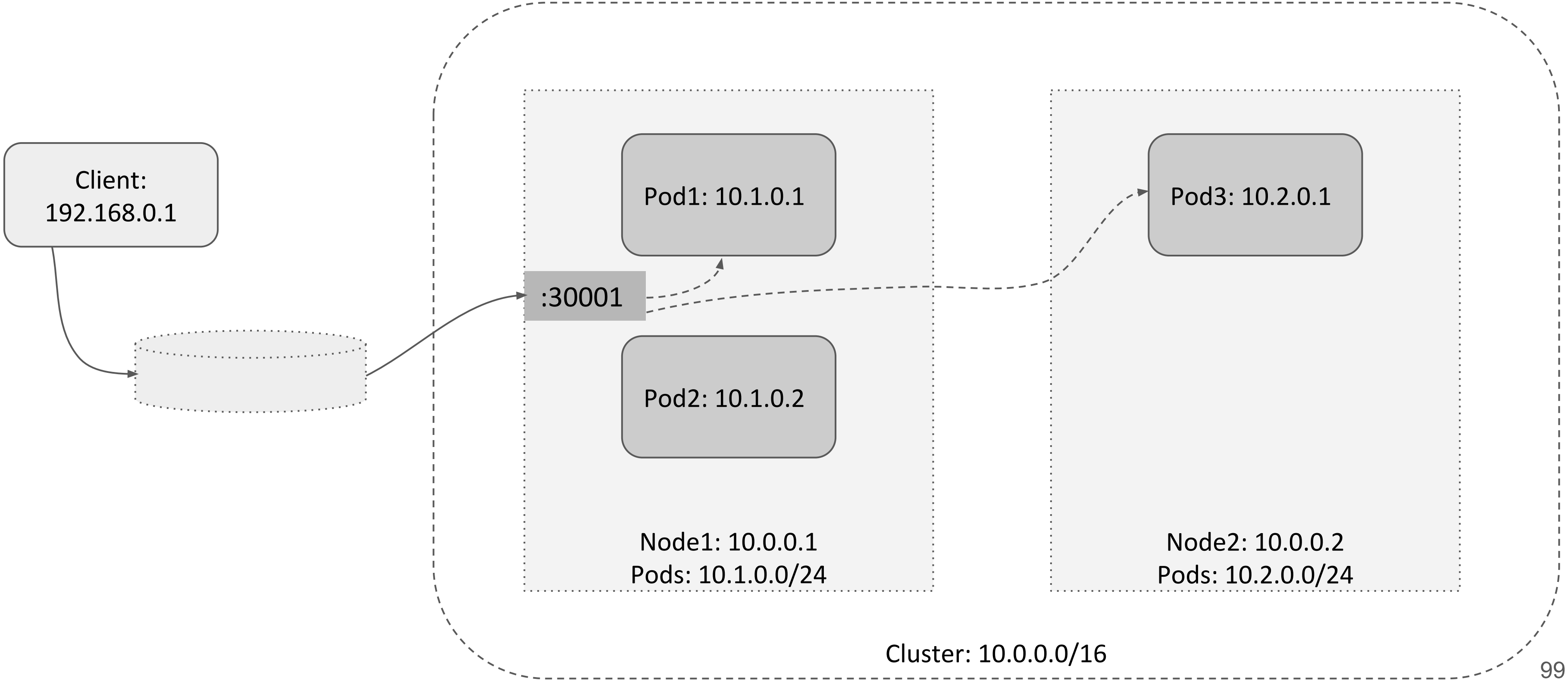
Thanks! Questions?

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

Appendix: K8s networking 101

3/4/5.a) “externalTrafficPolicy=Cluster”

K8s networking 101



K8s networking 101

Backends can be local or remote.

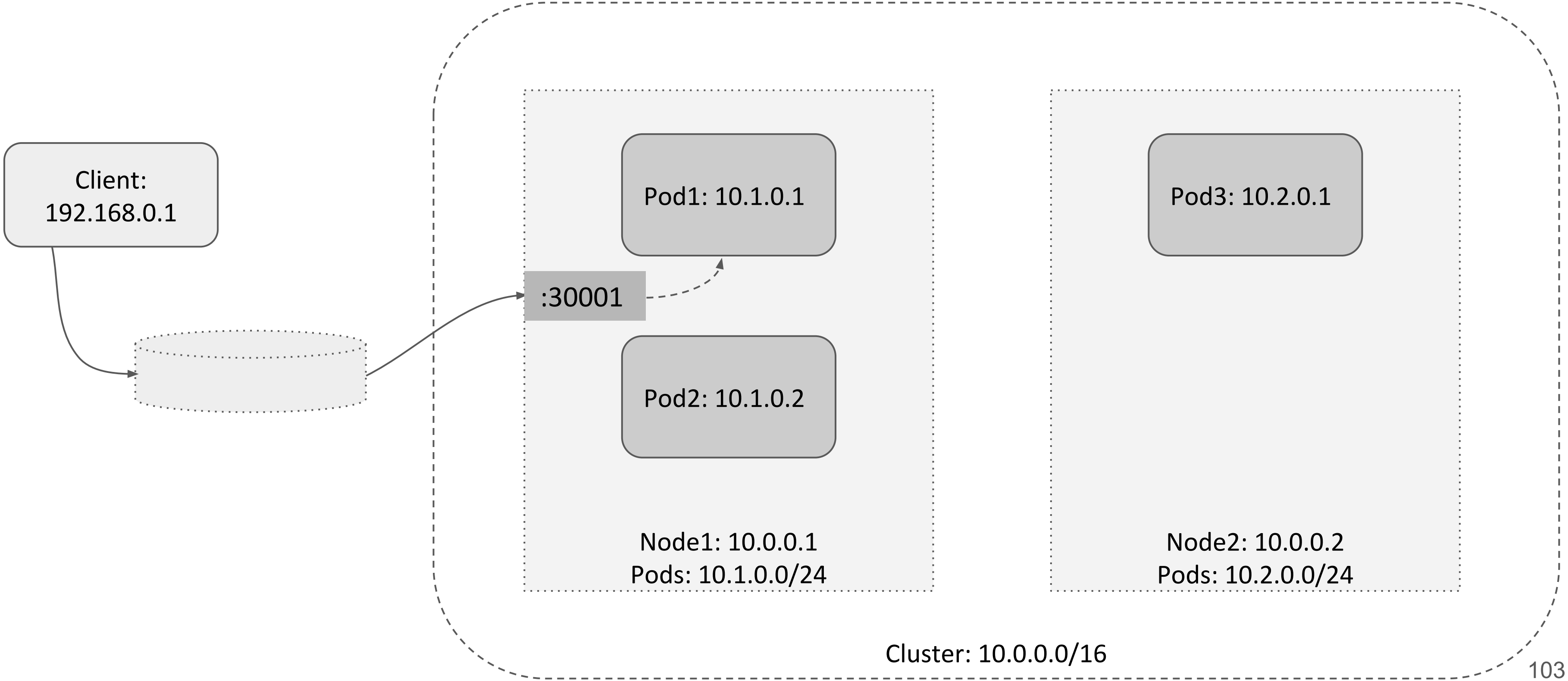
K8s networking 101

Traffic can be spread evenly across cluster, but for remote backends client source IP is lost unless DSR is available & used (Cilium).

K8s networking 101

3/4/5.b) “externalTrafficPolicy=Local”

K8s networking 101



K8s networking 101

Backends must only ever be local.

K8s networking 101

Client source IP is preserved. Potential of traffic imbalance though. Needs Pod anti-affinity against hostname.

K8s networking 101

Nodes w/o service backends drop requests. This is probed via cloud LB health check to update its backends.