Container Networking: The Play of BPF & Network NS with different Virtual Devices

Takshak Chahande & Martin KaFai Lau

Meta

Agenda

01 L3-Level networking (cgroup-bpf)

02 Building solution with Network Namespaces

03 How BPF helped to work around challenges

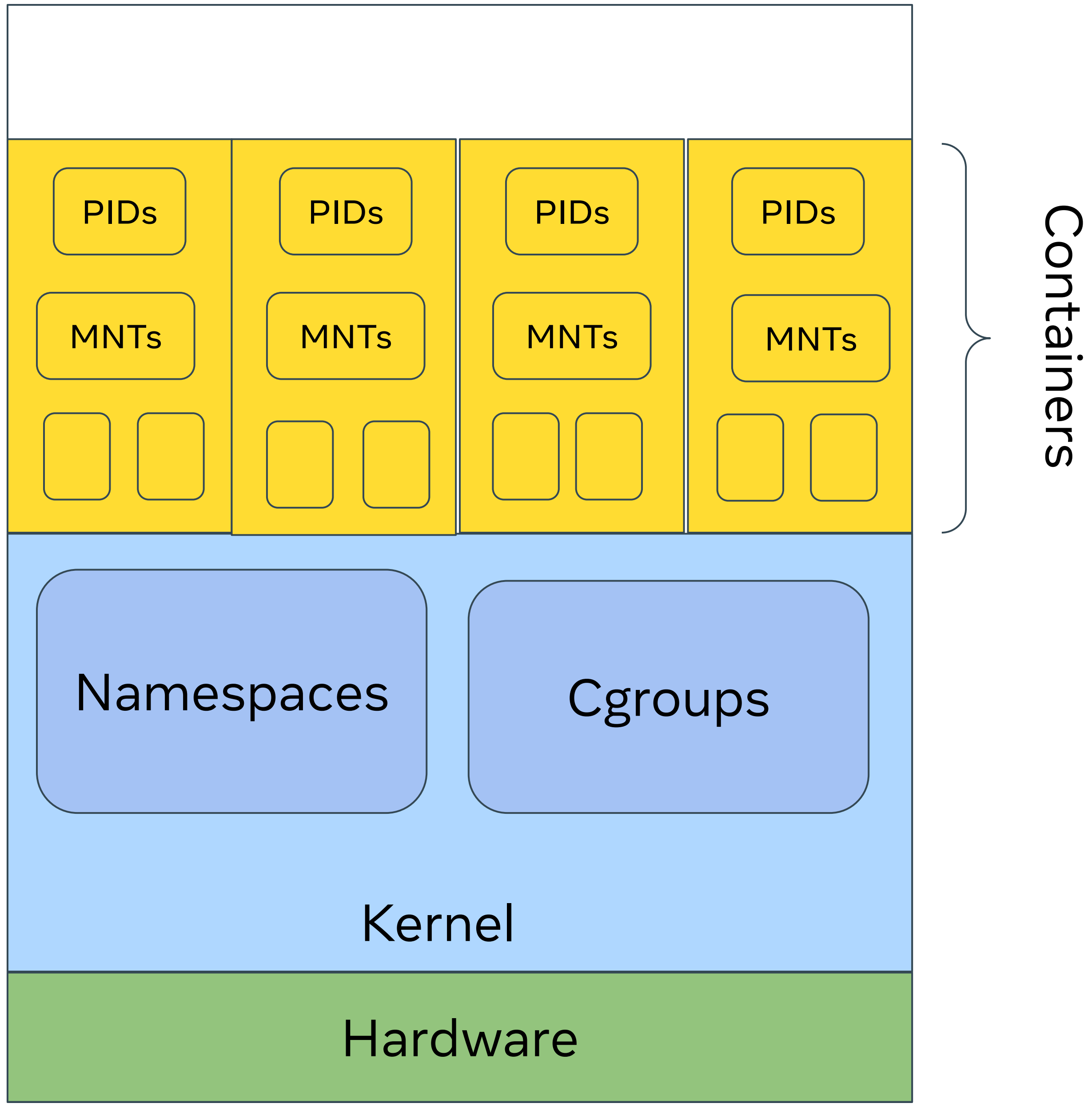
04 Performance data on different net devices solution

Meta

01 Building solution without Network Namespaces

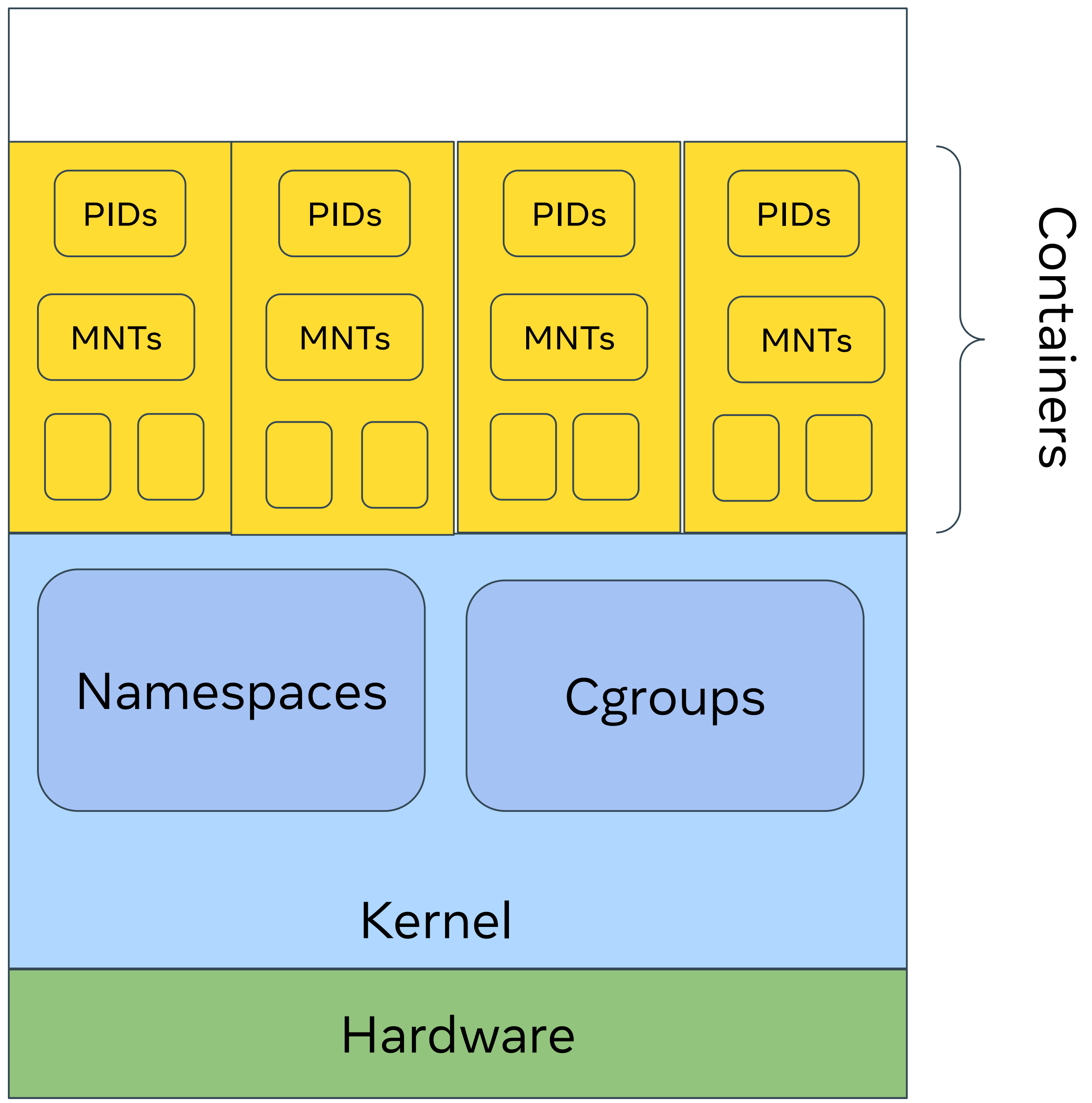
Meta

Linux Containers ?



Meta

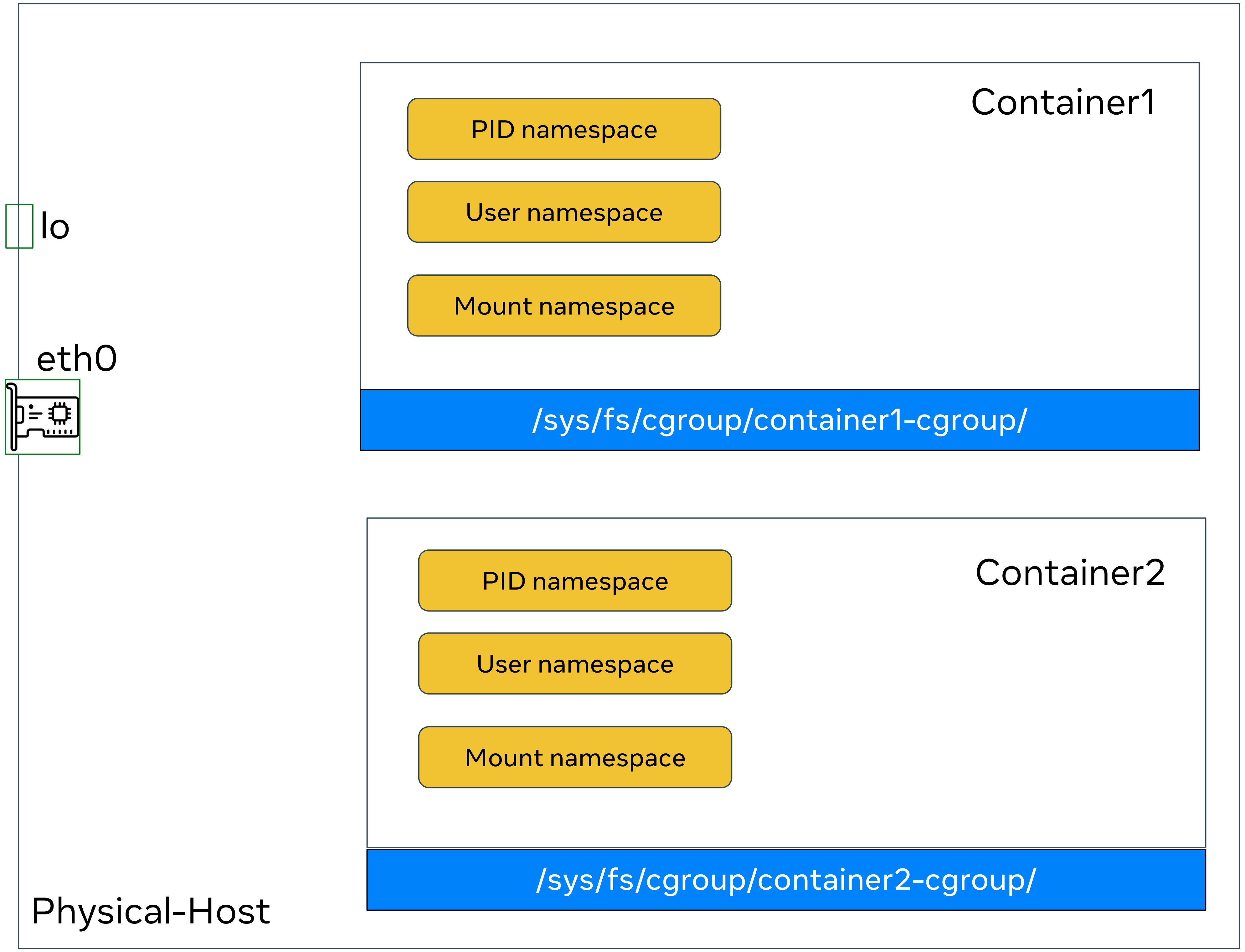
Linux Containers



- Container shapes
  - Square Shapes
  - L-Shape
- Host Accessibility
  - Single Tenant
  - Multi-Tenant Host
- Container Isolation
  - Resource Isolations
  - Shared Resources

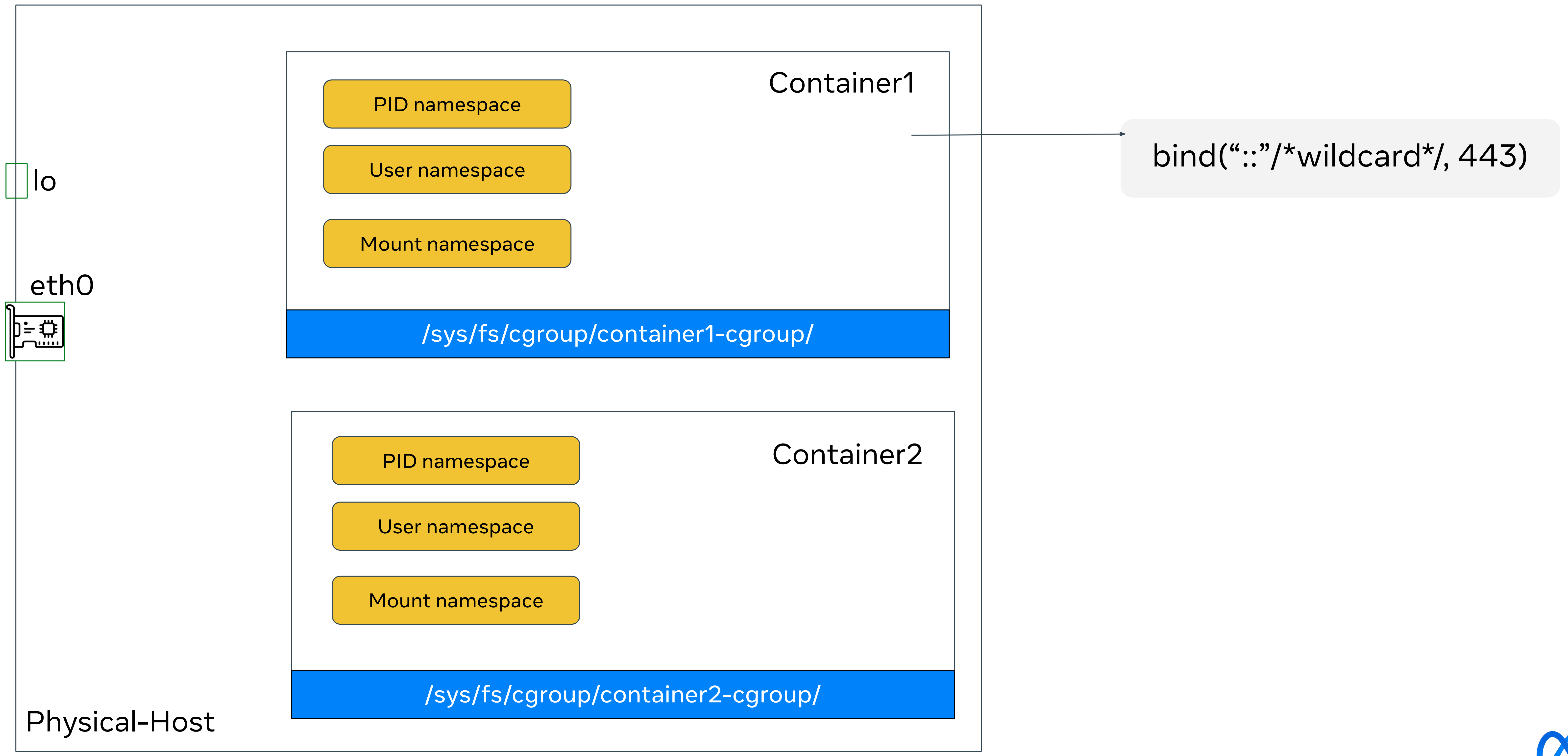
Meta

Linux Containers : Shared-Network Resource

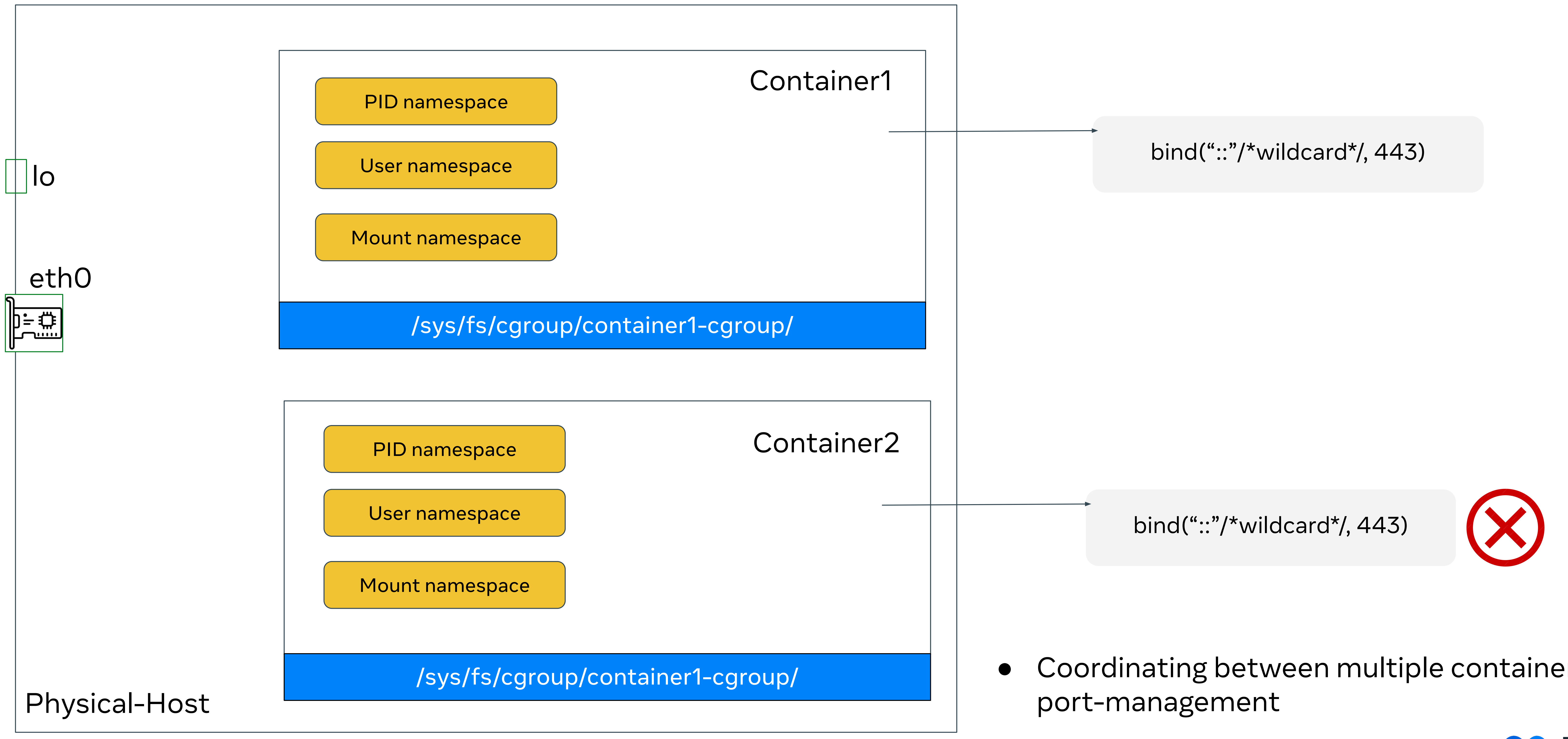


- Container1 & Container2 both shares the host-network namespace
- No extra-network configuration setup

Shared-Network Resource: Port mgmt



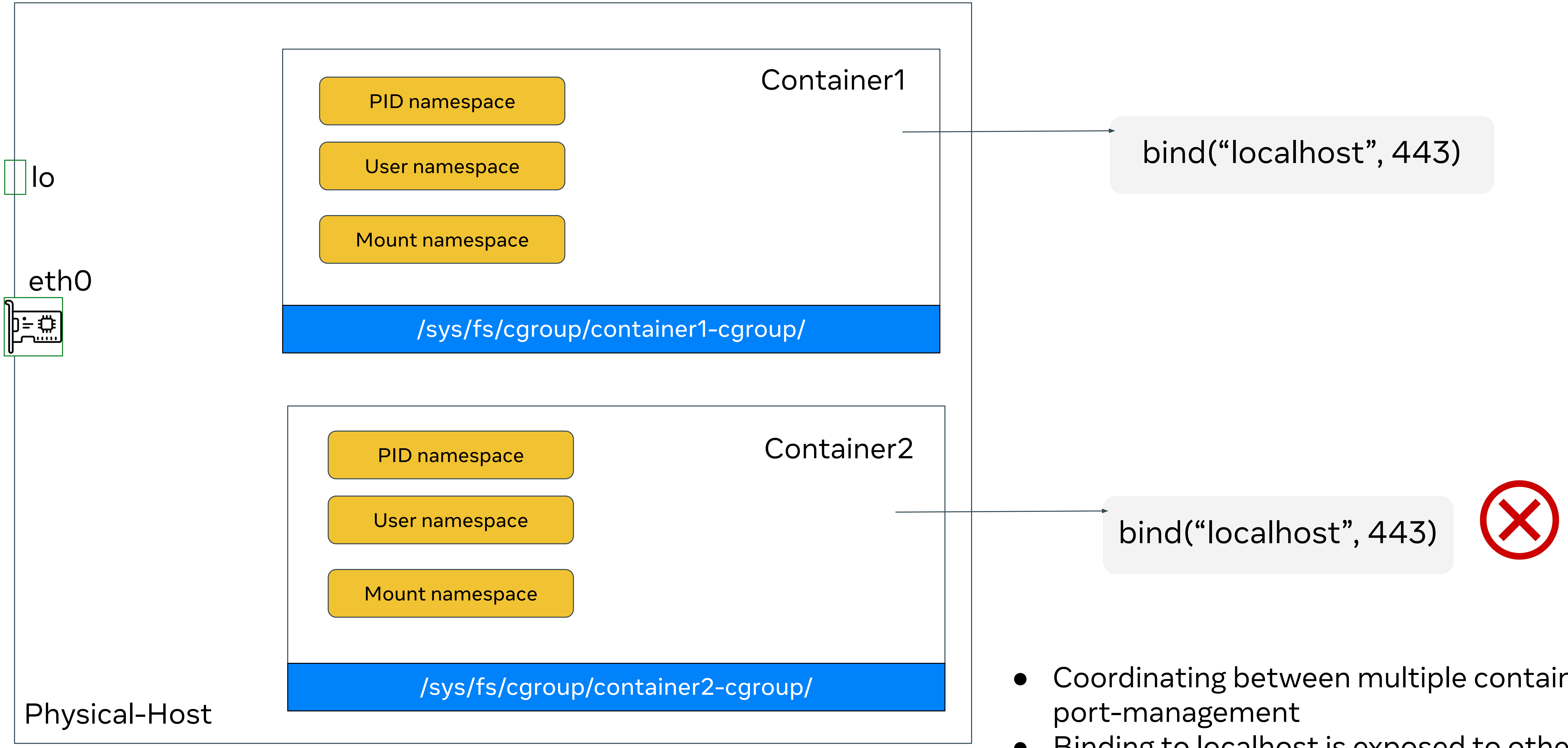
Shared-Network Resource: Port mgmt



- Coordinating between multiple containers for port-management

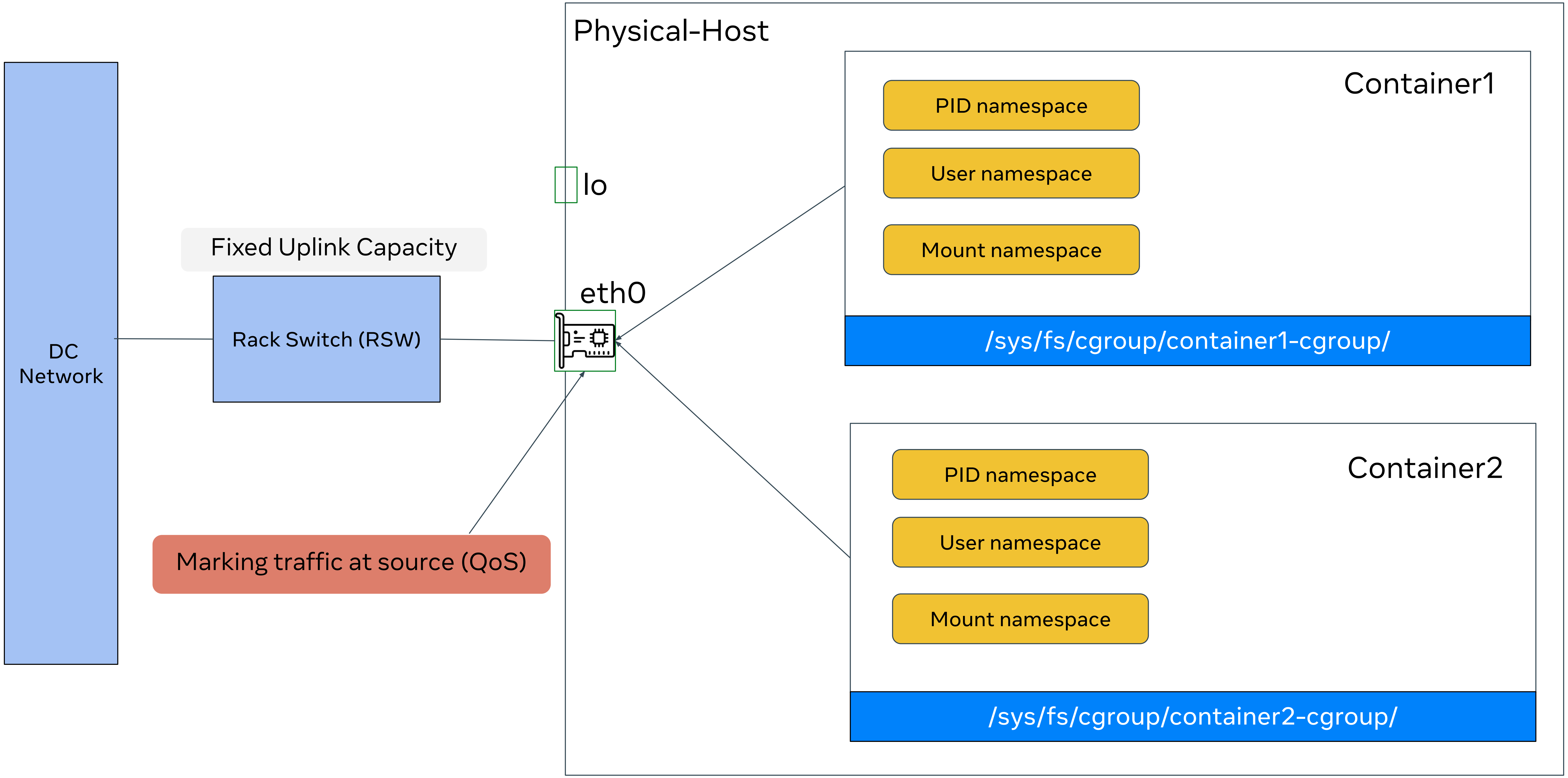
Meta

Shared-Network Resource : Port mgmt



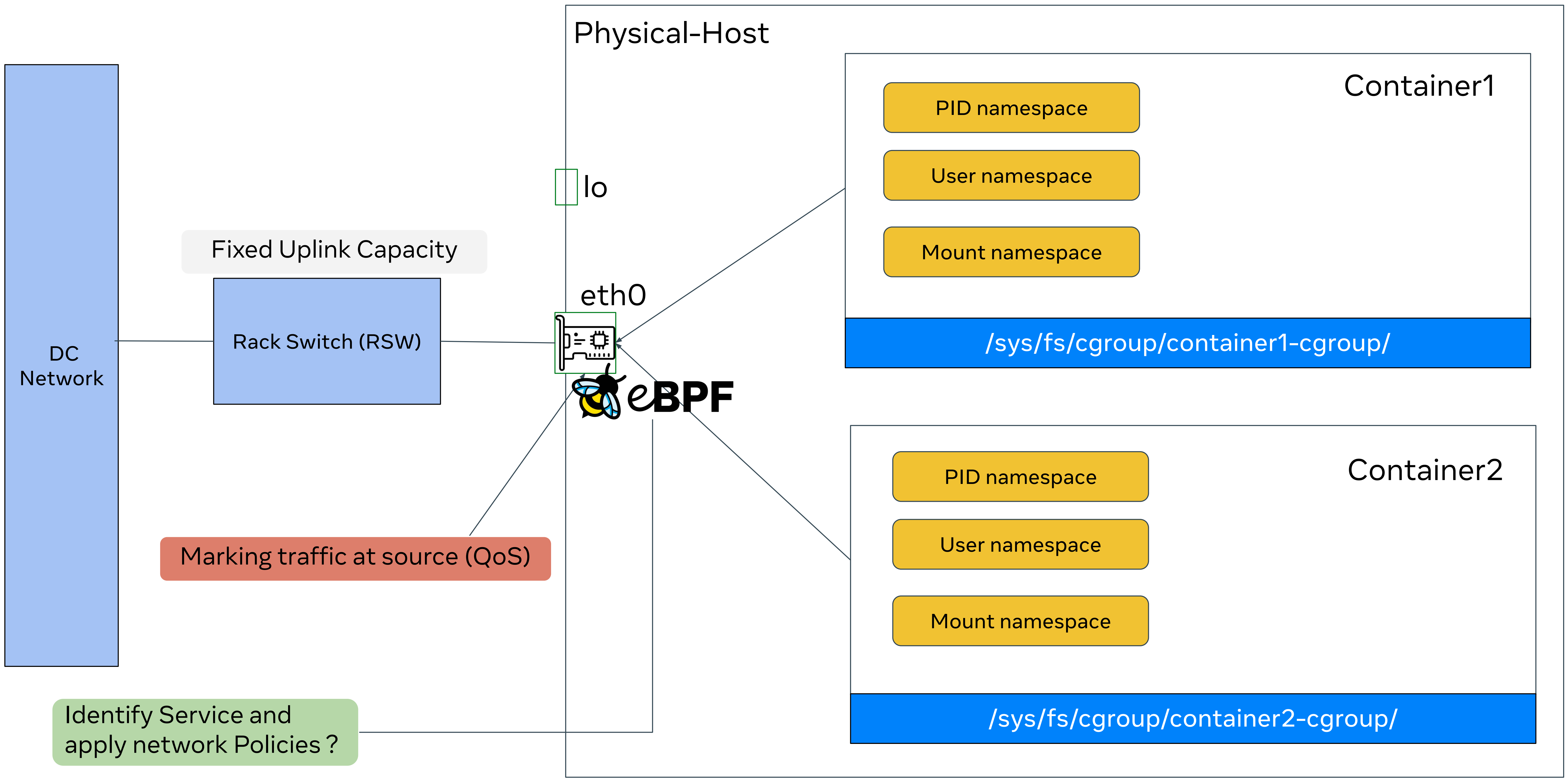
- Coordinating between multiple containers for port-management
- Binding to localhost is exposed to other containers & host

Shared-Network Resource : Service traffic mgmt



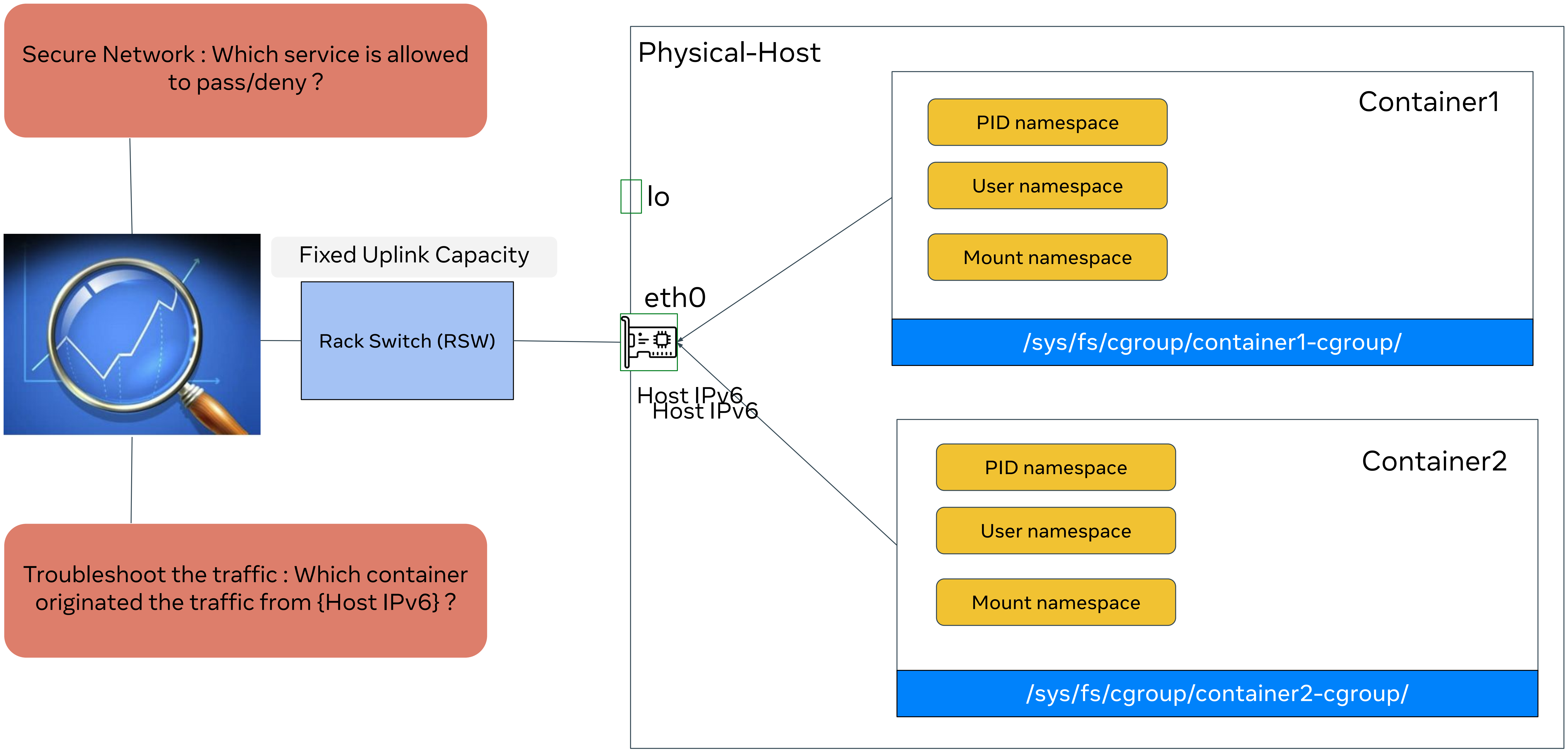
Meta

Shared-Network Resource : Service traffic mgmt



Meta

Shared-Network Resource : Service traffic mgmt



Need network identifier to each container and some level of isolation ?

Image credit: Internet sources

We decided to give unique IPv6 identity to each container

We decided to give unique IPv6 identity to each container

How to tie this IPv6 identity to the container ?

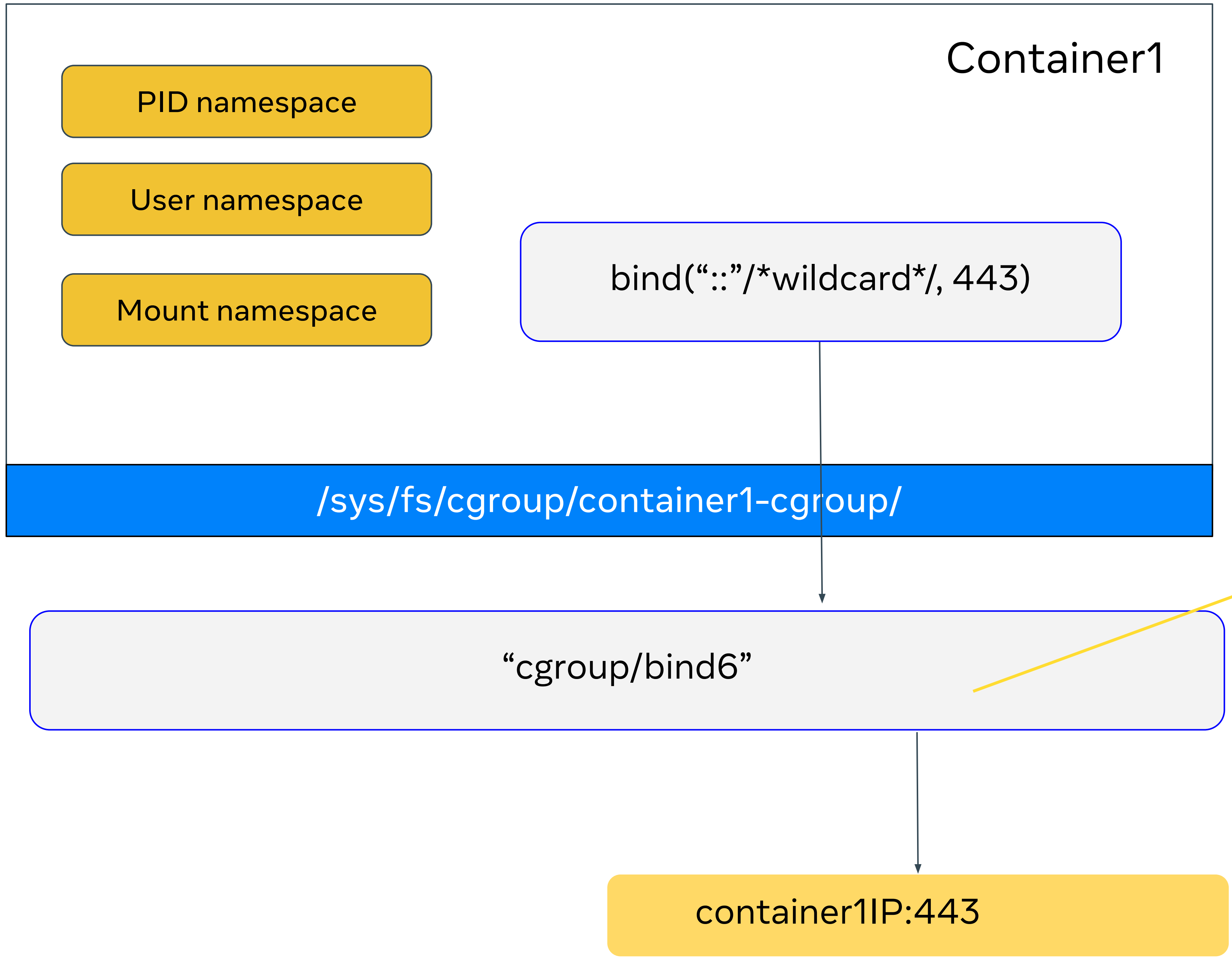
We decided to give unique IPv6 identity to each container

How to tie this IPv6 identity to the container ?

eBPF cgroup-bpf : bind/connect/sendmsg

Meta

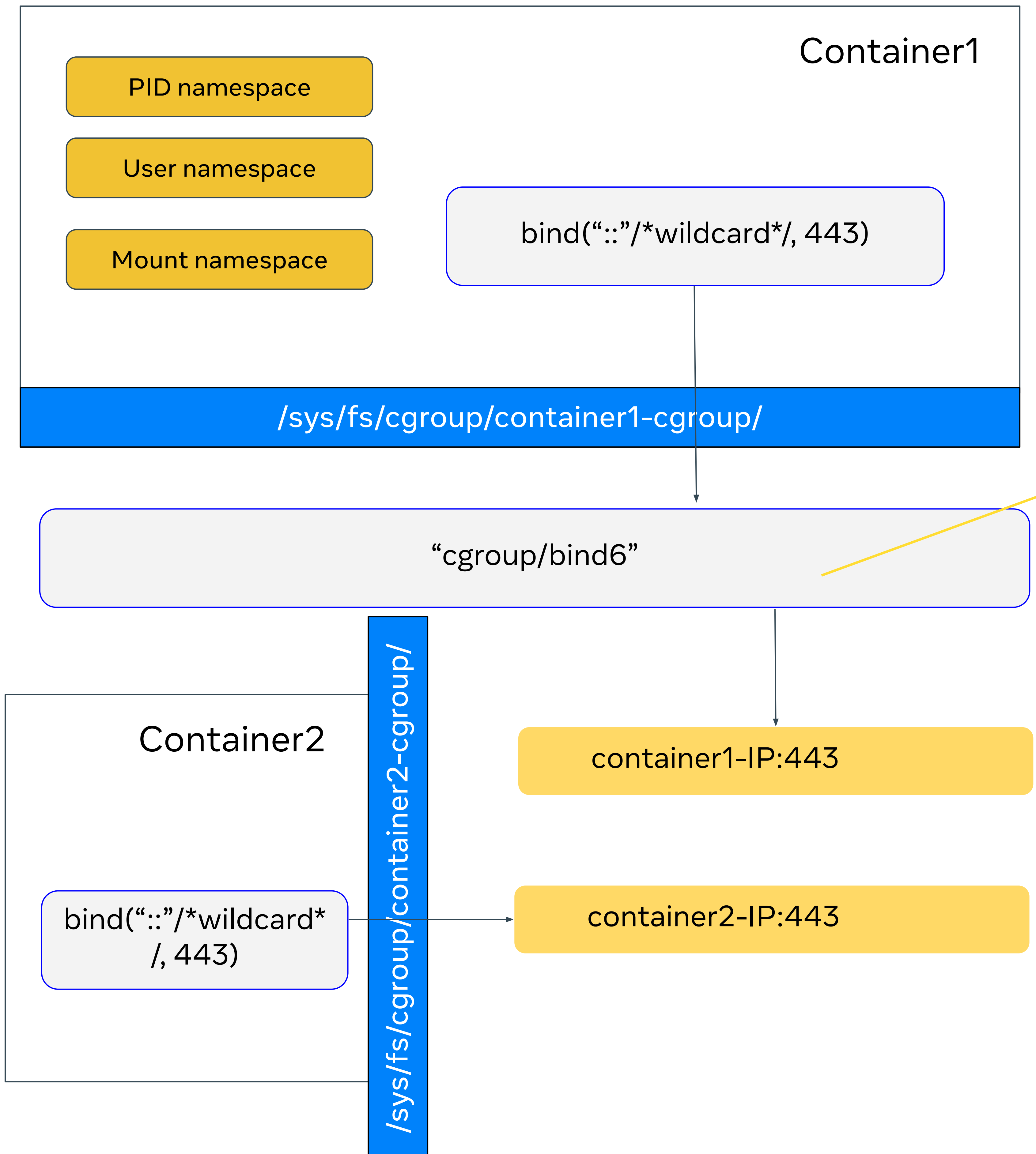
eBPF Play : Same Host-Network-namespace




```
/* User bpf_sock_addr struct to access socket fields and sockaddr
struct passed
 * by user and intended to be used by socket (e.g. to bind to,
depends on
 * attach type).
 */
struct bpf_sock_addr {
    __u32 user_family;  /* Allows 4-byte read, but no write. */
    __u32 user_ip4;     /* Allows 1,2,4-byte read and 4-byte write.
                 * Stored in network byte order.
                 */
    __u32 user_ip6[4];  /* Allows 1,2,4,8-byte read and 4,8-byte
write.
                 * Stored in network byte order.
                 */
    __u32 user_port;    /* Allows 1,2,4-byte read and 4-byte write.
                 * Stored in network byte order
                 */
    __u32 family;       /* Allows 4-byte read, but no write */
    __u32 type;     /* Allows 4-byte read, but no write */
    __u32 protocol;     /* Allows 4-byte read, but no write */
    __u32 msg_src_ip4;  /* Allows 1,2,4-byte read and 4-byte write.
                 * Stored in network byte order.
                 */
    __u32 msg_src_ip6[4]; /* Allows 1,2,4,8-byte read and 4,8-byte
write.
                 * Stored in network byte order.
                 */
    __bpf_md_ptr(struct bpf_sock *, sk);
 };
```

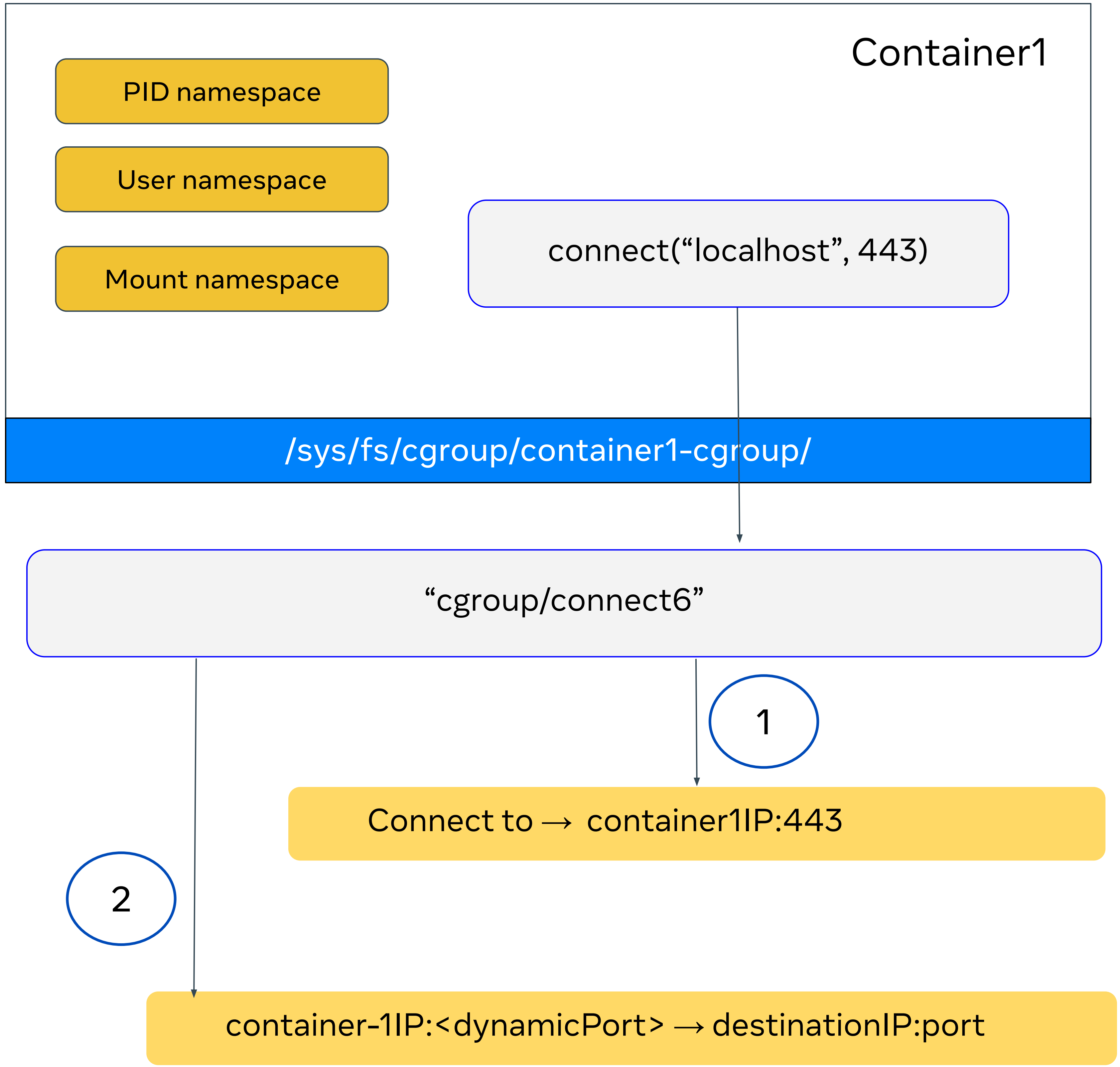
Meta

Play : Same Host-Network-namespace




```c
/* User bpf_sock_addr struct to access socket fields and sockaddr
struct passed
 * by user and intended to be used by socket (e.g. to bind to,
depends on
 * attach type).
 */
struct bpf_sock_addr {
    __u32 user_family;  /* Allows 4-byte read, but no write. */
    __u32 user_ip4;     /* Allows 1,2,4-byte read and 4-byte write.
                 * Stored in network byte order.
                 */
    __u32 user_ip6[4];  /* Allows 1,2,4,8-byte read and 4,8-byte
write.
                 * Stored in network byte order.
                 */
    __u32 user_port;    /* Allows 1,2,4-byte read and 4-byte write.
                 * Stored in network byte order
                 */
    __u32 family;       /* Allows 4-byte read, but no write */
    __u32 type;     /* Allows 4-byte read, but no write */
    __u32 protocol;     /* Allows 4-byte read, but no write */
    __u32 msg_src_ip4;  /* Allows 1,2,4-byte read and 4-byte write.
                 * Stored in network byte order.
                 */
    __u32 msg_src_ip6[4]; /* Allows 1,2,4,8-byte read and 4,8-byte
write.
                 * Stored in network byte order.
                 */
    __bpf_md_ptr(struct bpf_sock *, sk);
};
```

eBPF Play : Same Host-Network-namespace



- Re-write the destination to container IP
- Set the source identity for the outgoing traffic

```
sa.sin6_family = AF_INET6;
sa.sin6_port = bpf_htons(0);

in6cpy(&sa.sin6_addr, task_ip);

/* Rewrite source IP. */
if (bpf_bind(ctx, (struct sockaddr*)&sa, sizeof(sa)) != 0)
  return FAIL_OPEN;
```

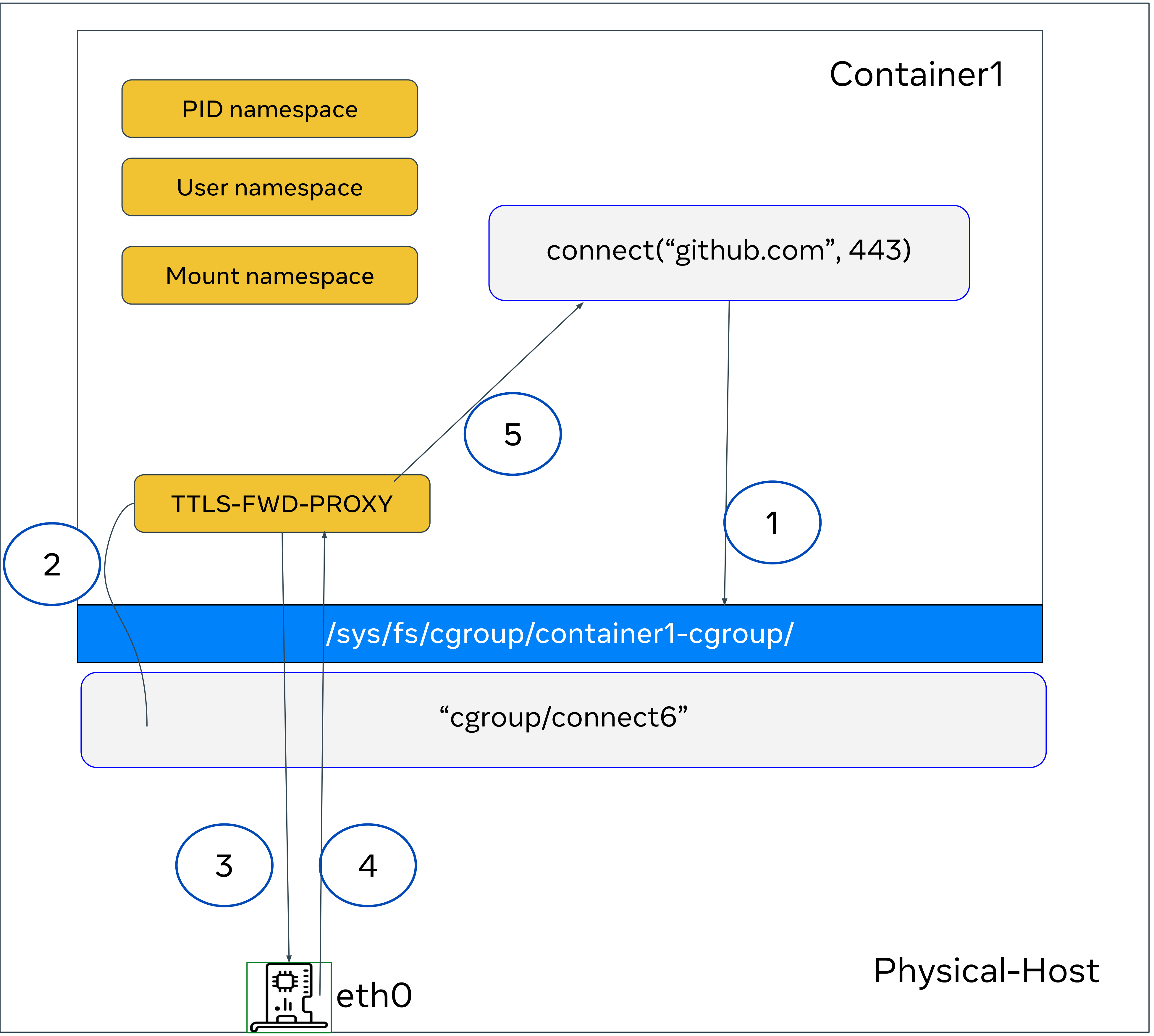
Acknowledgement : Andrey Ignatov and Alexei S

Shared host-network space : Re-look challenges

- ➔ Two containers can not start service on same fixed port
  - ◆ Unique IP per container helps at certain extent
  - ◆ Fails if the same port binds on the wildcard by other host-based services
- ➔ Container localhost service get exposed to host & whole meta
  - ◆ Services binds on container-IP which is routable in Meta fleet
  - ◆ eBPF helps but stills adds an additional overhead to handle it
- ➔ Does not allow wildcard binding inside the container (hacks for additional VIPs)
  - ◆ Hard to share same port among container IP and BGP VIPs

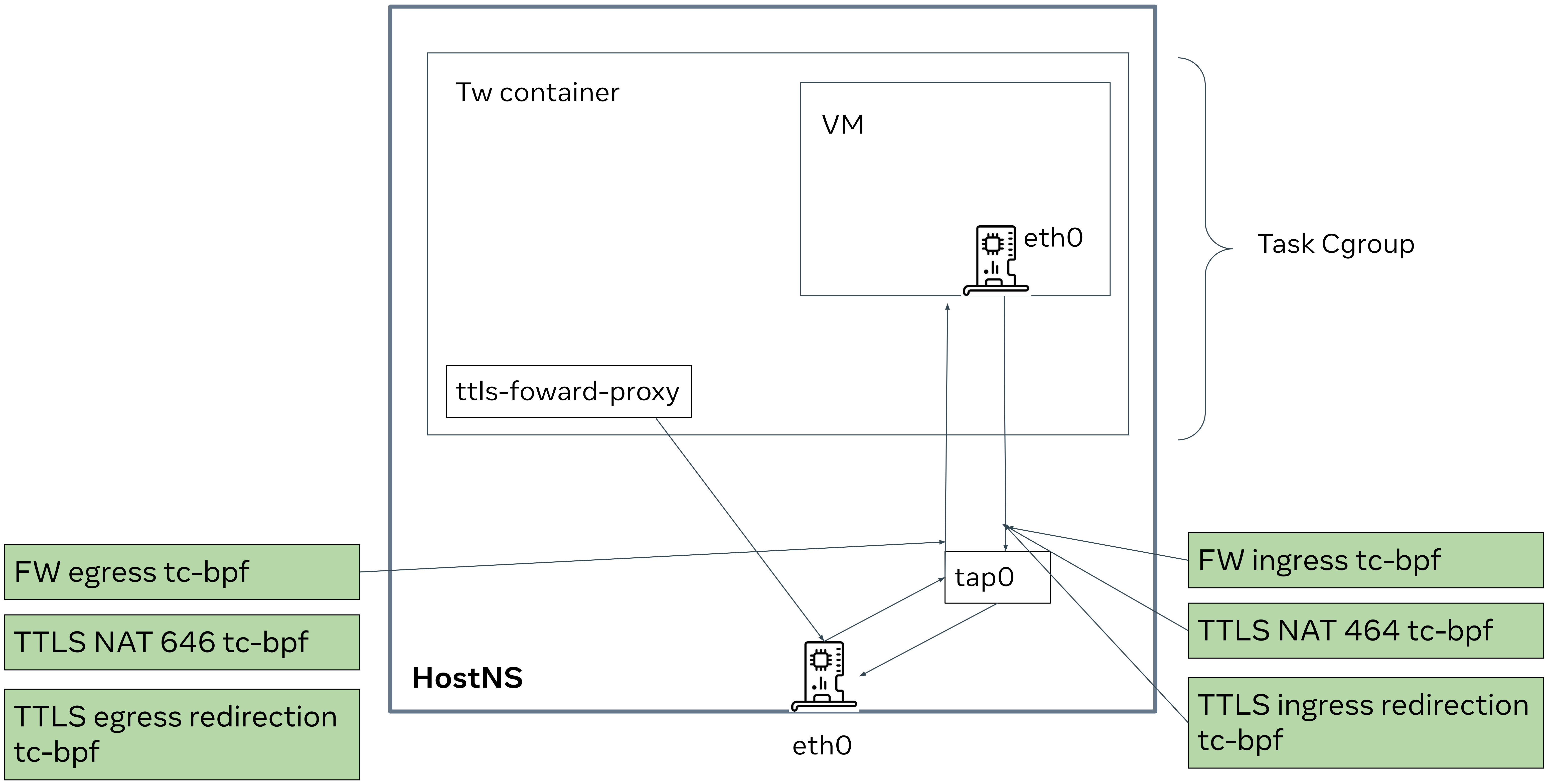
Meta

Traffic Redirection over TLS



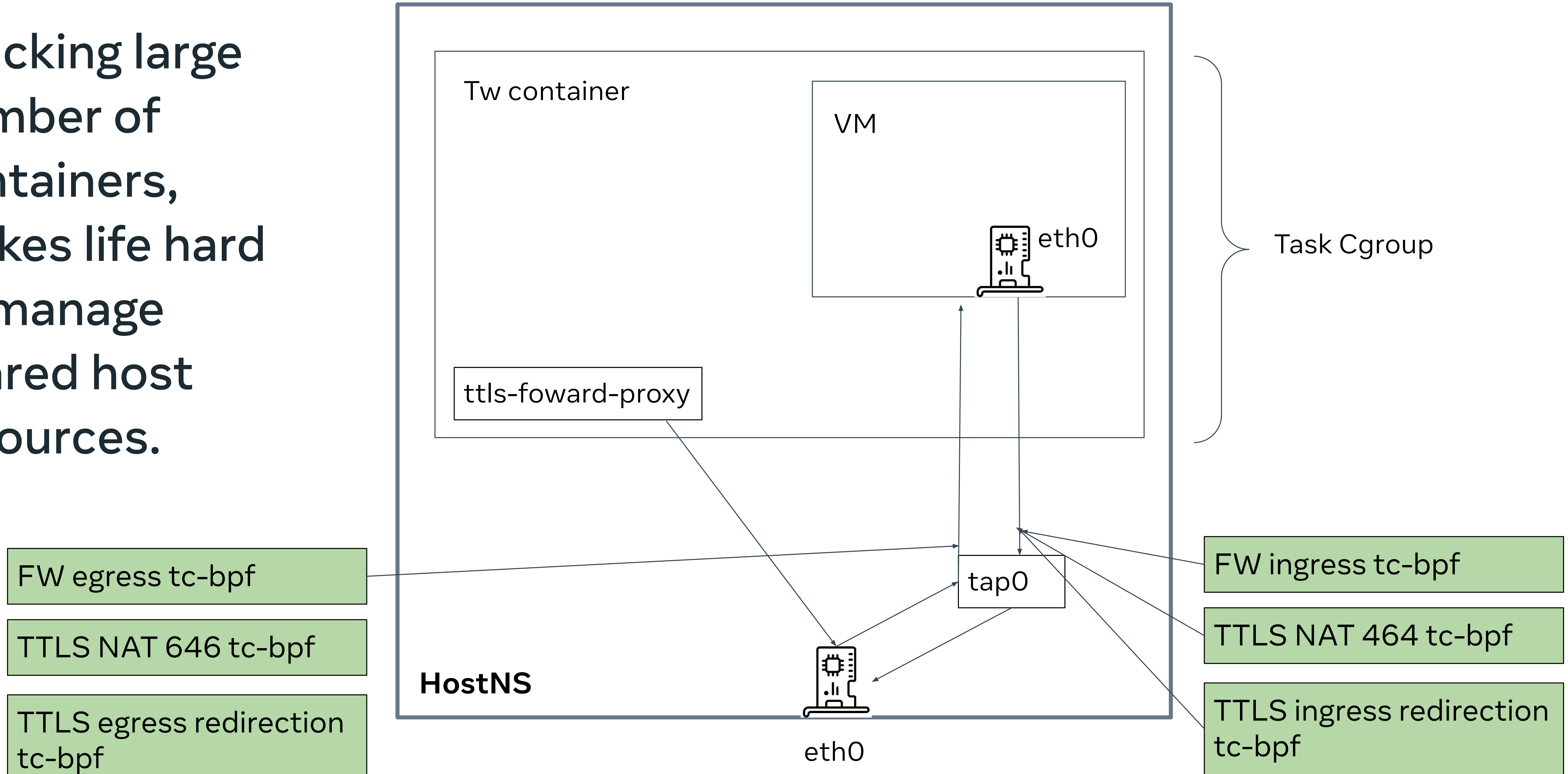
- Using eBPF hooks with socket cookies, it is easy to track TCP connections
- For UDP sockets, where the same source IP:port can be used for multiple destinations; proxy can’t track the connections
  - Packet encapsulation helps to solve this but that requires tc-bpf based solution
  - Moving every container’s UDP traffic tracking at host-eth0 is again a challenge in multi-tenant host

Process VM inside the container

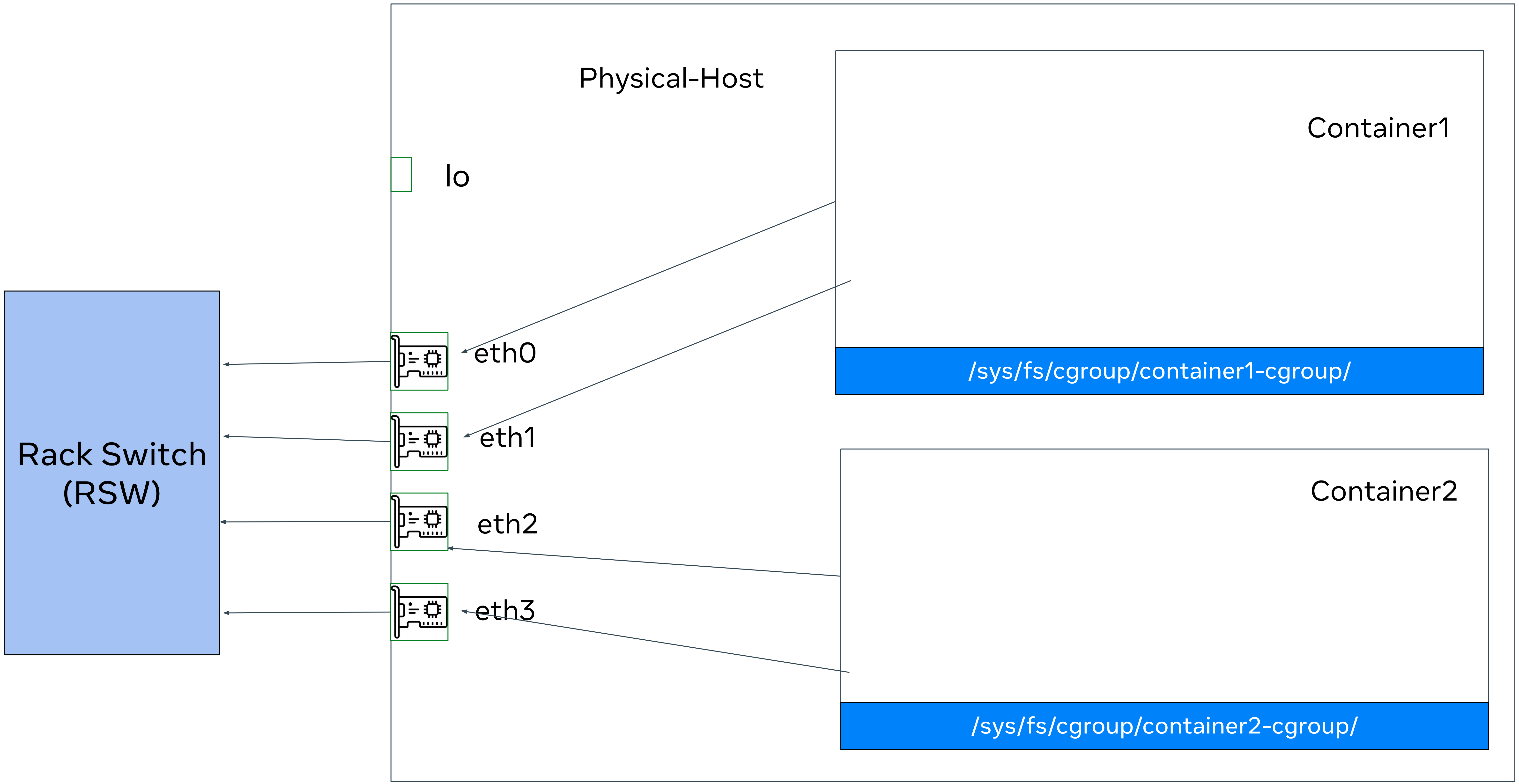


Process VM inside the container

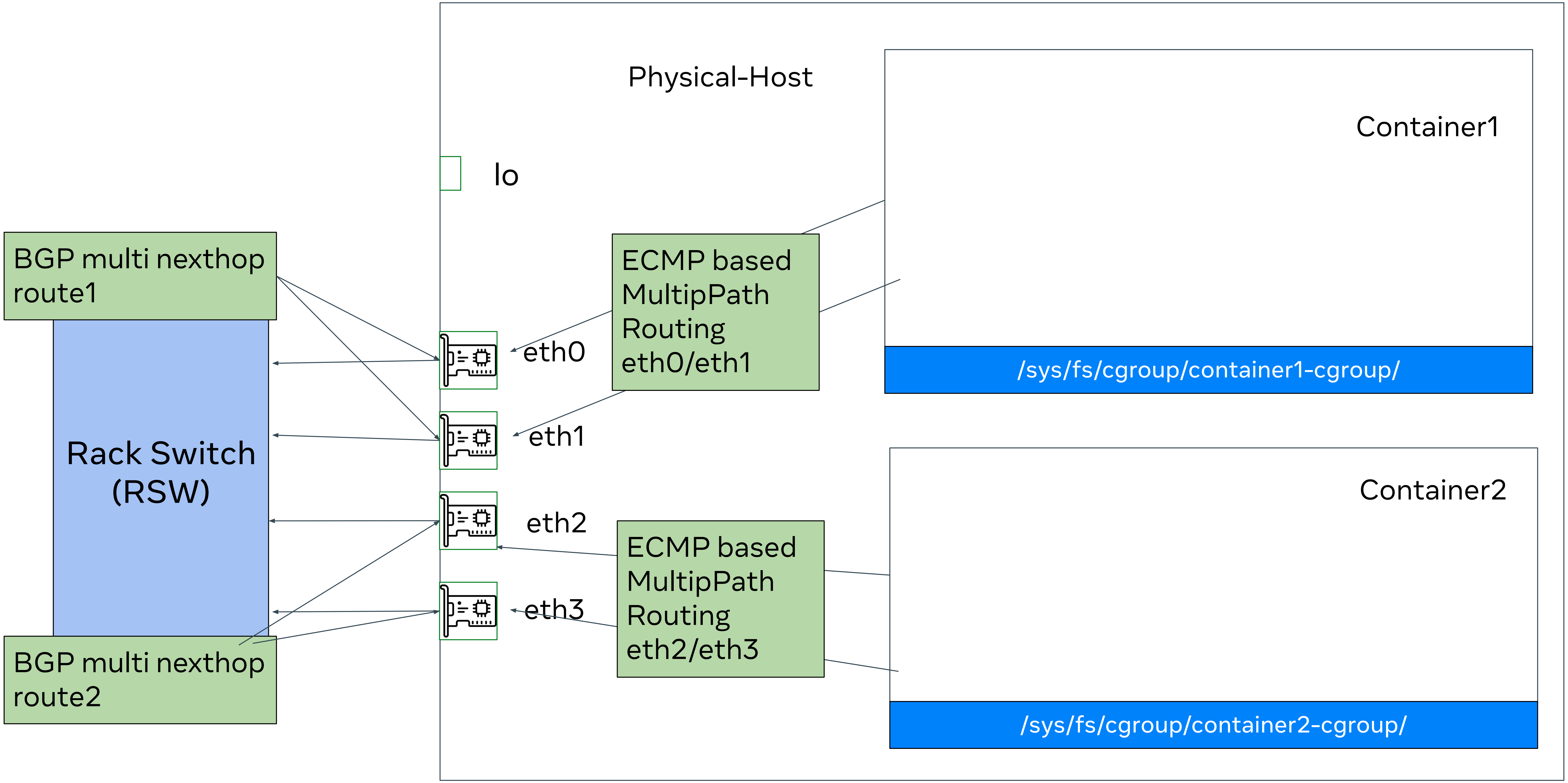
Stacking large number of containers, makes life hard to manage shared host resources.



Multi-NIC Host : Routing Mgmt (Ingress/Egress)



Multi-NIC Host : Routing Mgmt



Meta

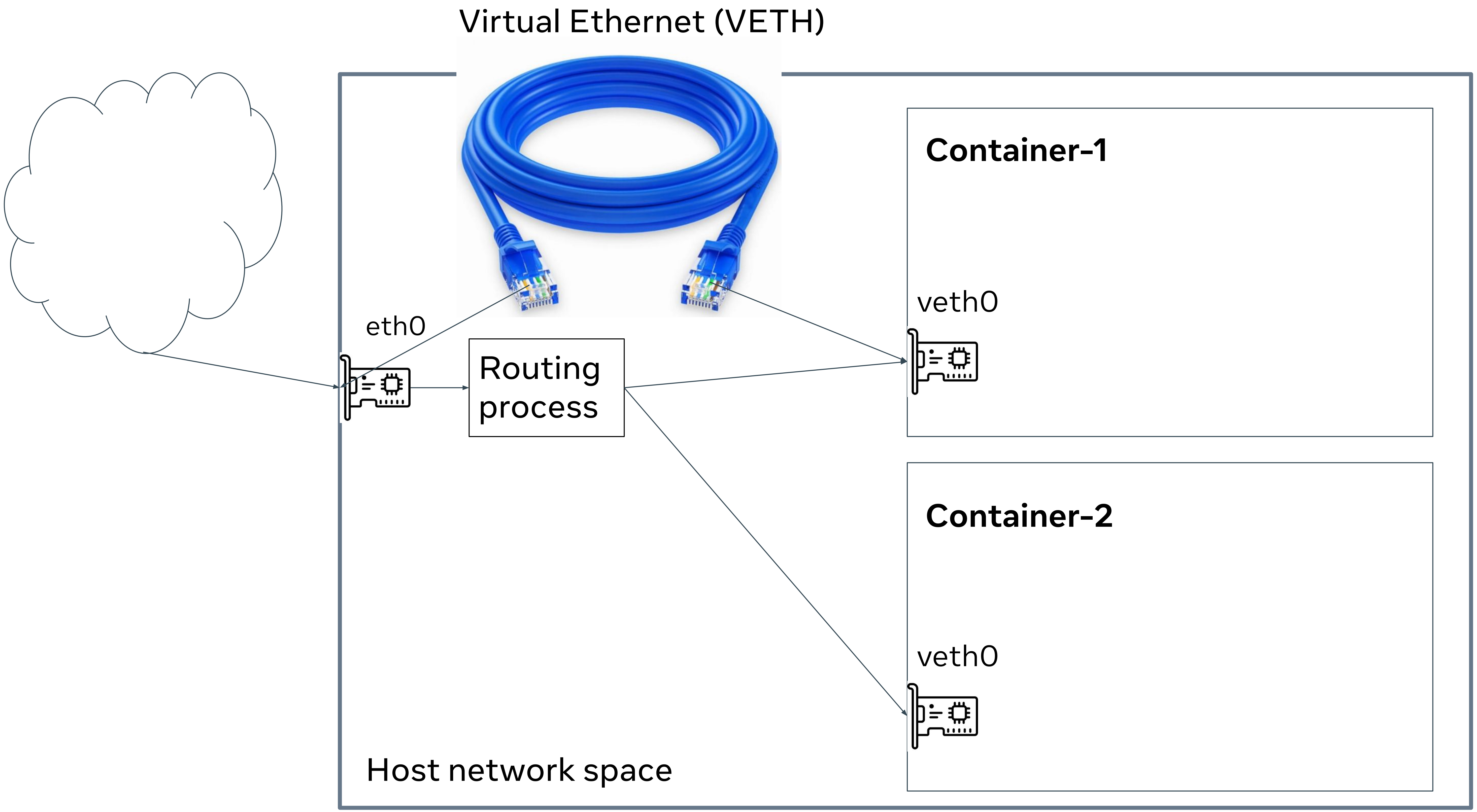
Apart from the gaps, other use-cases ?

- ➔ L2 level secure isolation to avoid all enforcement at host level
- ➔ per-container tc/XDP eBPF support again to avoid physical-eth0 a choking point
- ➔ Some of the emulated services need IPv4 support
- ➔ Running third-party services/applications with jailed environment
- ➔ Debug the container level traffic without having access to host

Meta

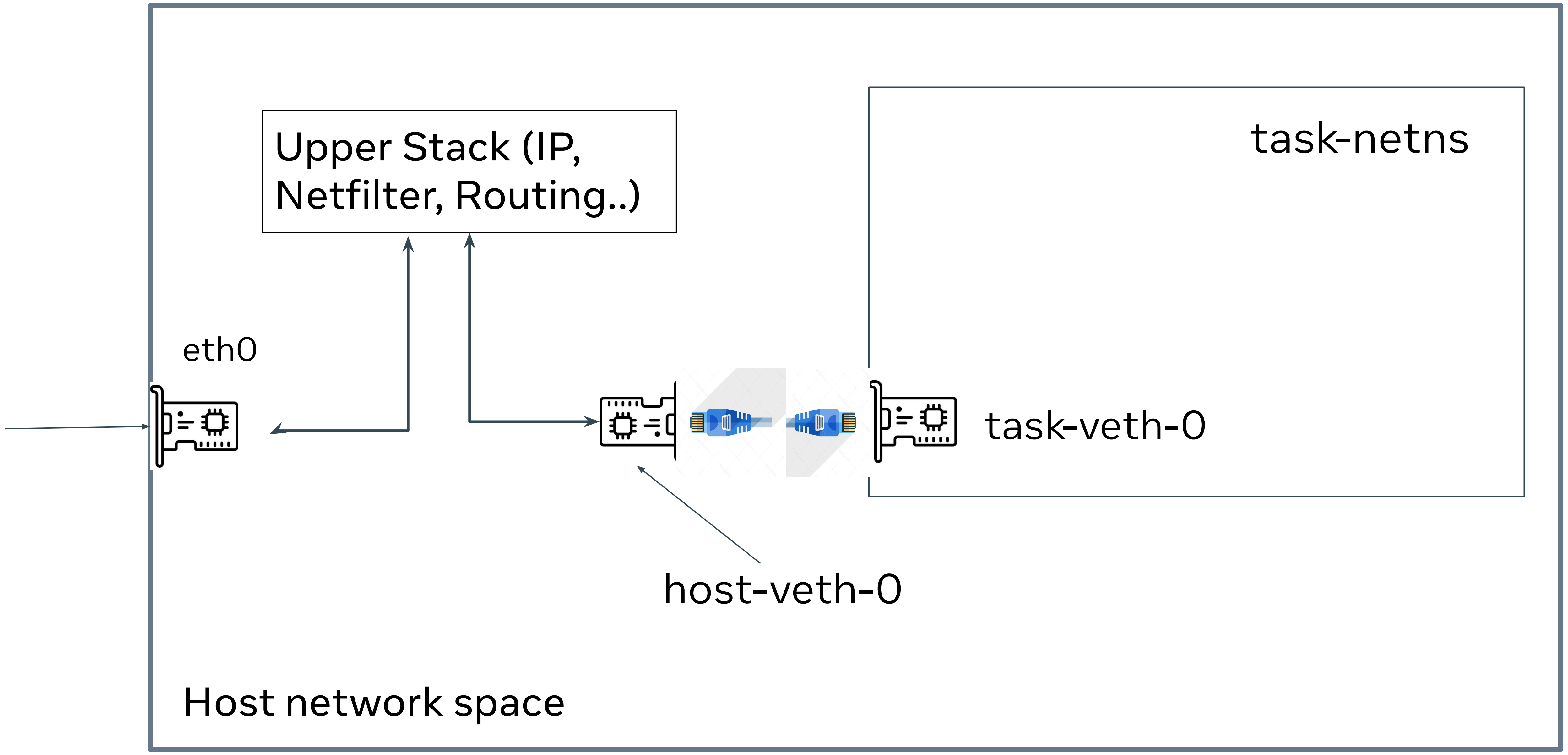
02 Building solution with Network Namespaces

Finally Network NS : Network Connection model



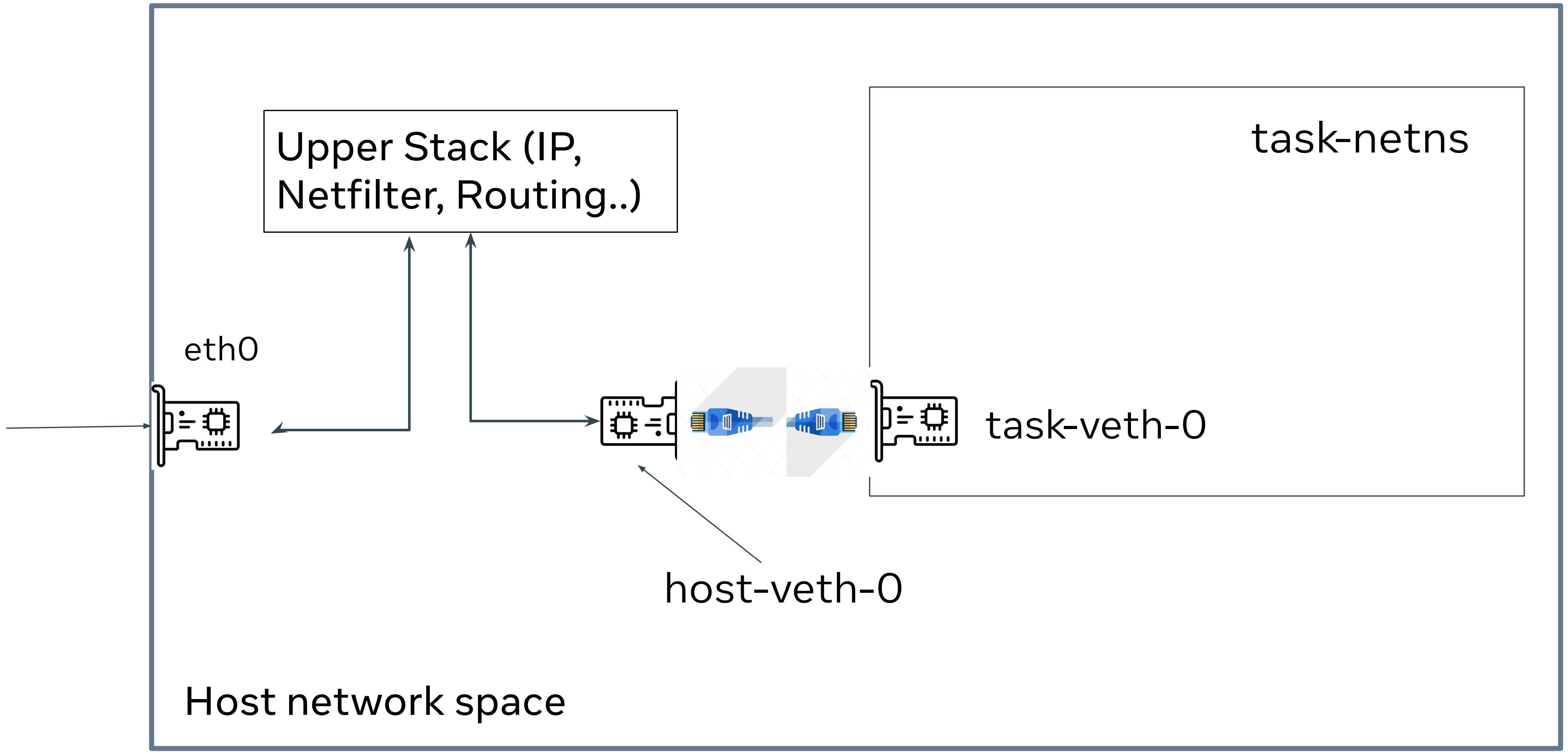
Meta

Network Namespaces : Build network connectivity



Meta

Network Namespaces : Build network connectivity



- ➢ Global IP Forwarding enablement
- ➢ veth(4) is slower due to additional traversal of network stack

03 How bpf helped to work around the issue

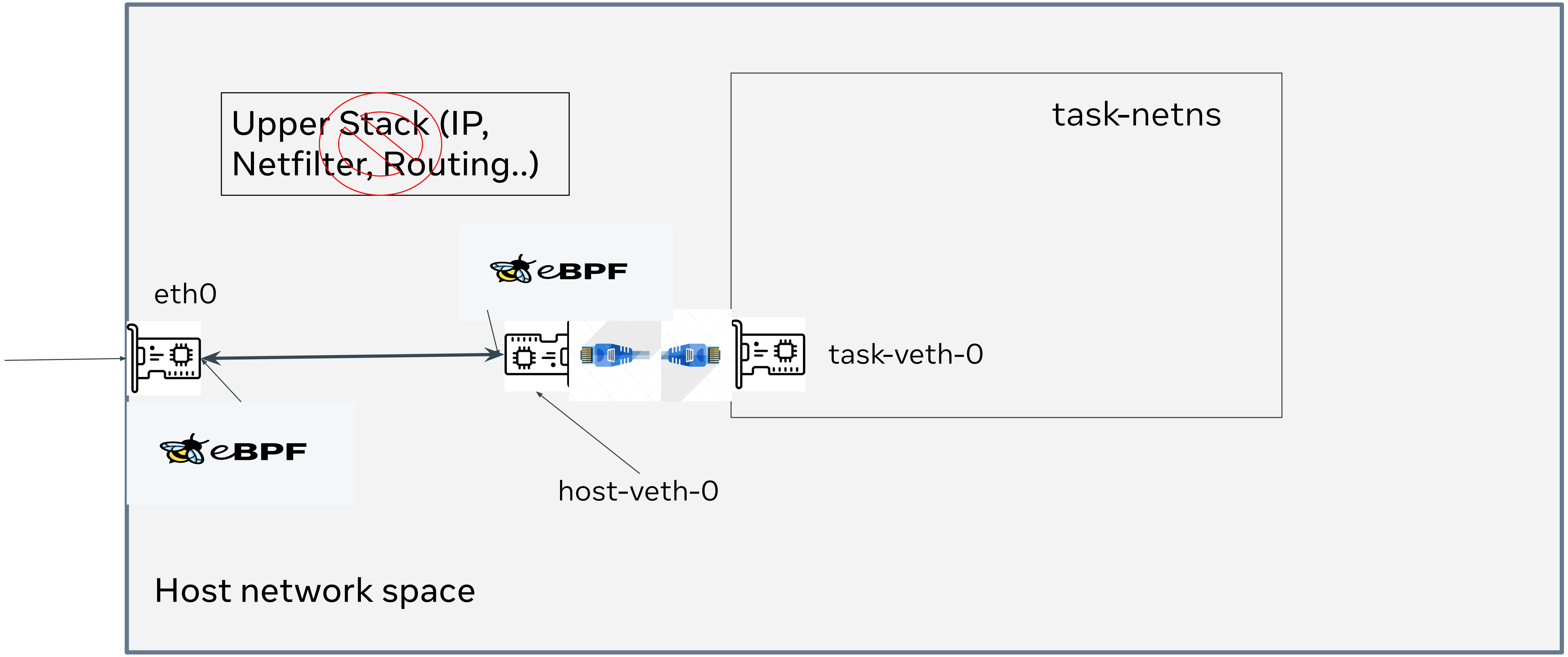
Network Namespaces : Use of eBPF Kernel extensions

author	Daniel Borkmann <daniel@iogearbox.net>	2020-09-30 17:18:17 +0200
committer	Alexei Starovoitov <ast@kernel.org>	2020-09-30 11:50:35 -0700
commit	b4ab31414970a7a03a5d55d75083f2c101a30592 (patch)	
tree	513b1e81e65d103278b626750a47644452bd7881	
parent	92acdc58ab11af66fcaef485433fde61b5e32fac (diff)	
download	linux-b4ab31414970a7a03a5d55d75083f2c101a30592.tar.gz	

bpf: Add redirect_neigh helper as redirect drop-in

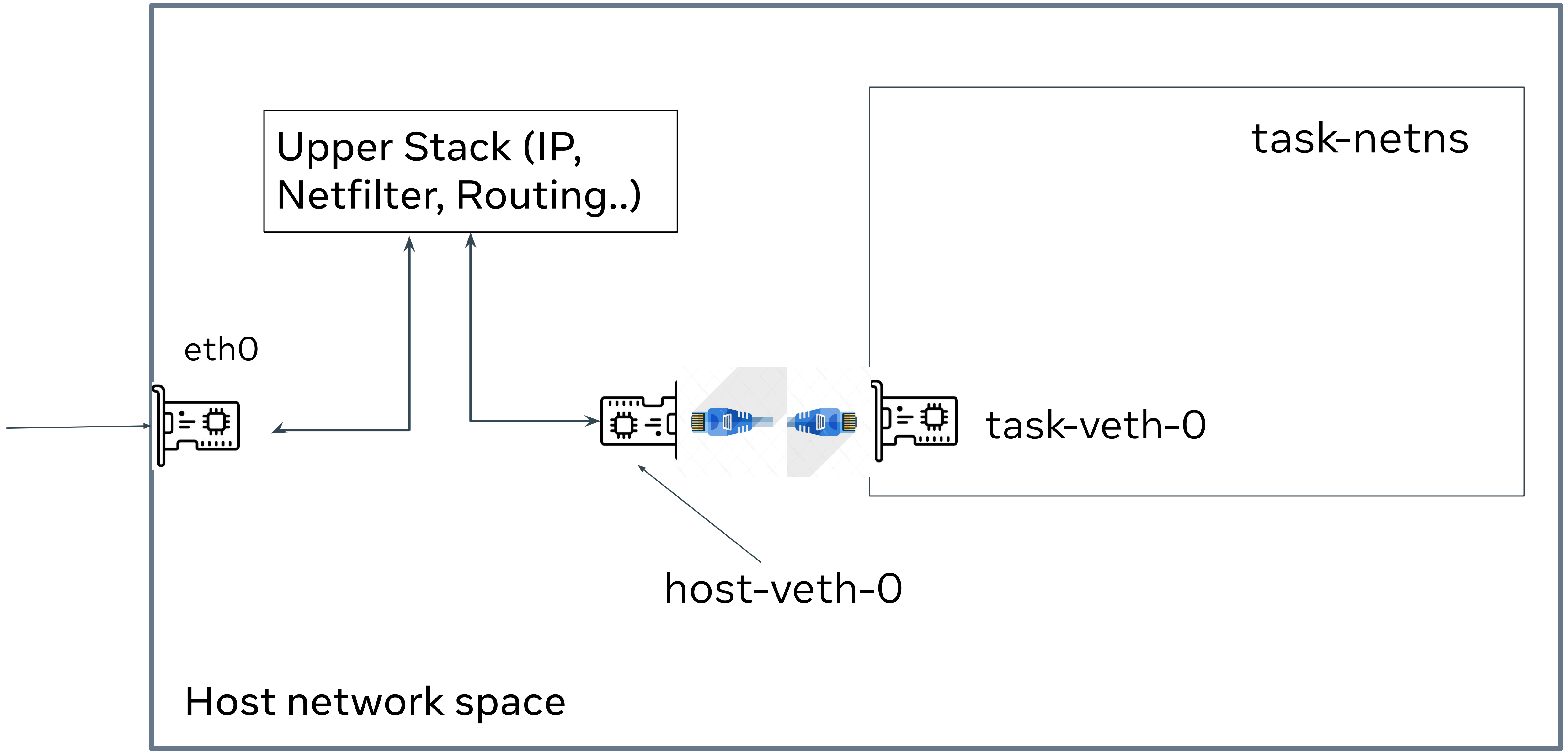
```
Add a redirect_neigh() helper as redirect() drop-in replacement
for the xmit side. Main idea for the helper is to be very similar
in semantics to the latter just that the skb gets injected into
the neighboring subsystem in order to let the stack do the work
it knows best anyway to populate the L2 addresses of the packet
and then hand over to dev_queue_xmit() as redirect() does.

This solves two bigger items: i) skbs don't need to go up to the
stack on the host facing veth ingress side for traffic egressing
the container to achieve the same for populating L2 which also
has the huge advantage that ii) the skb->sk won't get orphaned in
ip_rcv_core() when entering the IP routing layer on the host stack.
```

Network Namespaces : Use of eBPF Kernel extensions



Meta

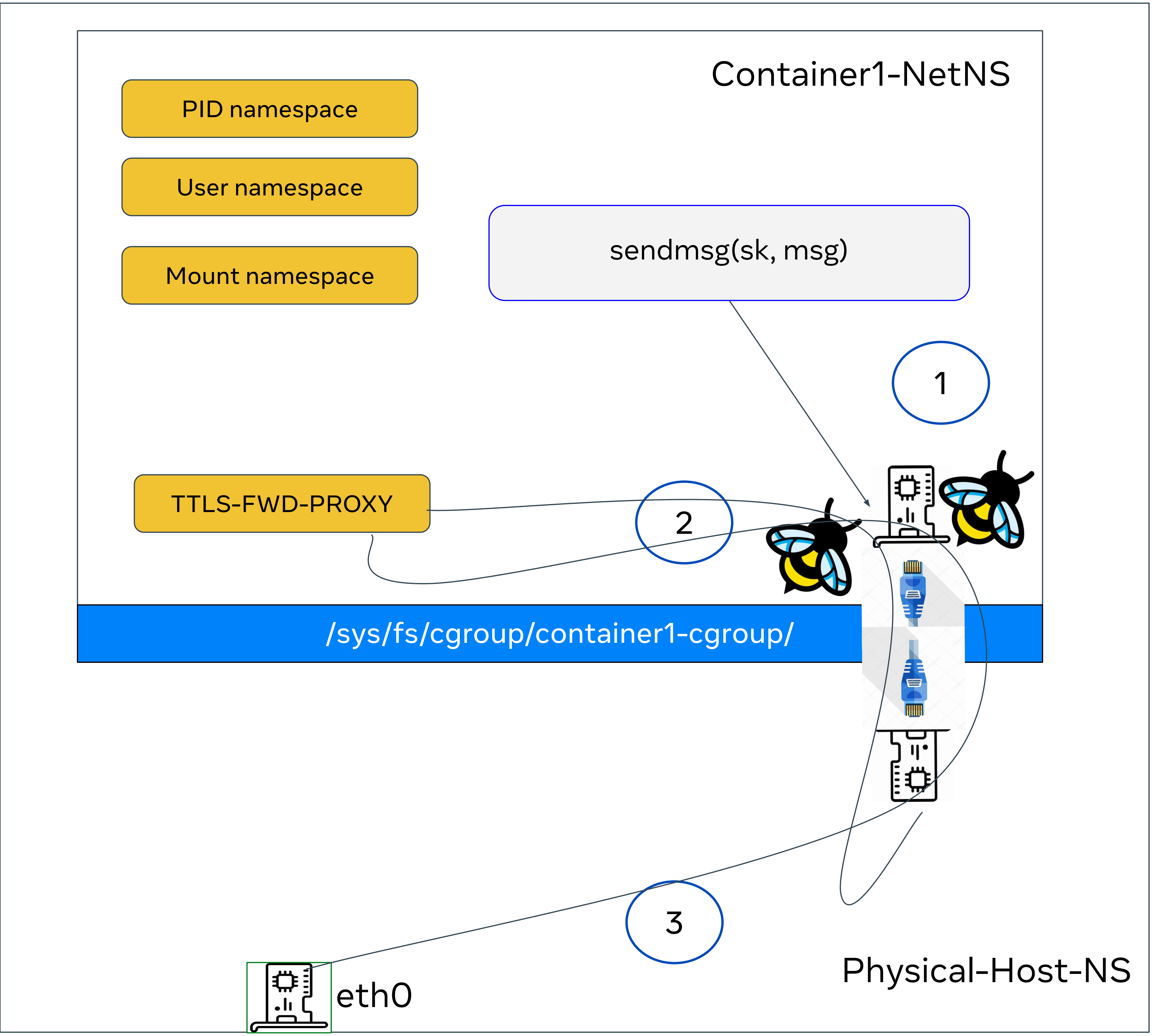
Network Namespaces : Build network connectivity



- ➢ Global IP Forwarding enablement
- ➢ veth(4) is slower due to additional traversal of network stack

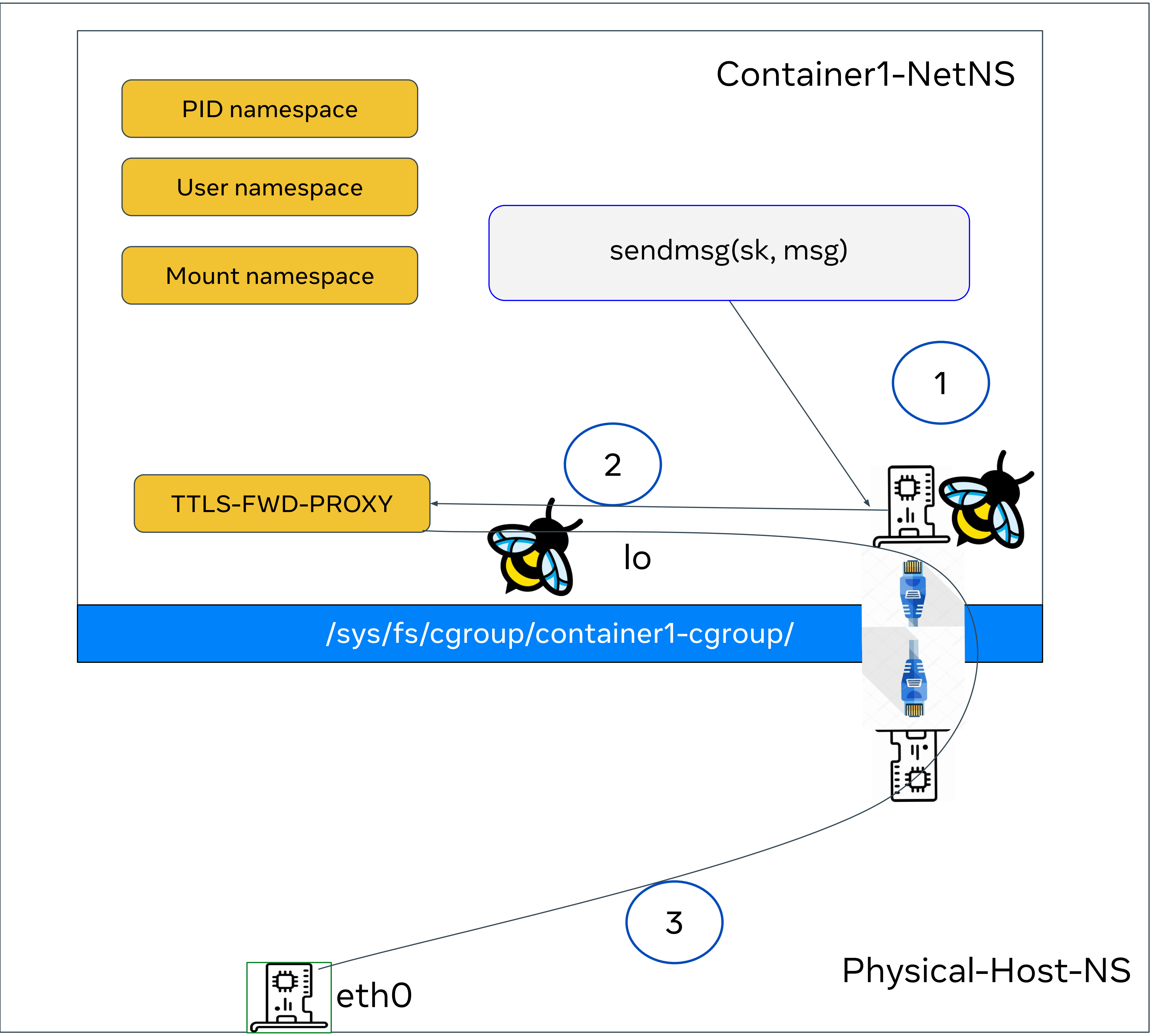
Meta

UDP Traffic Redirection over TLS



- Challenges
  - Packet makes a round trip from veth0-egress to host-end and back to task's netns
  - Ingress program at veth0 is in-effective

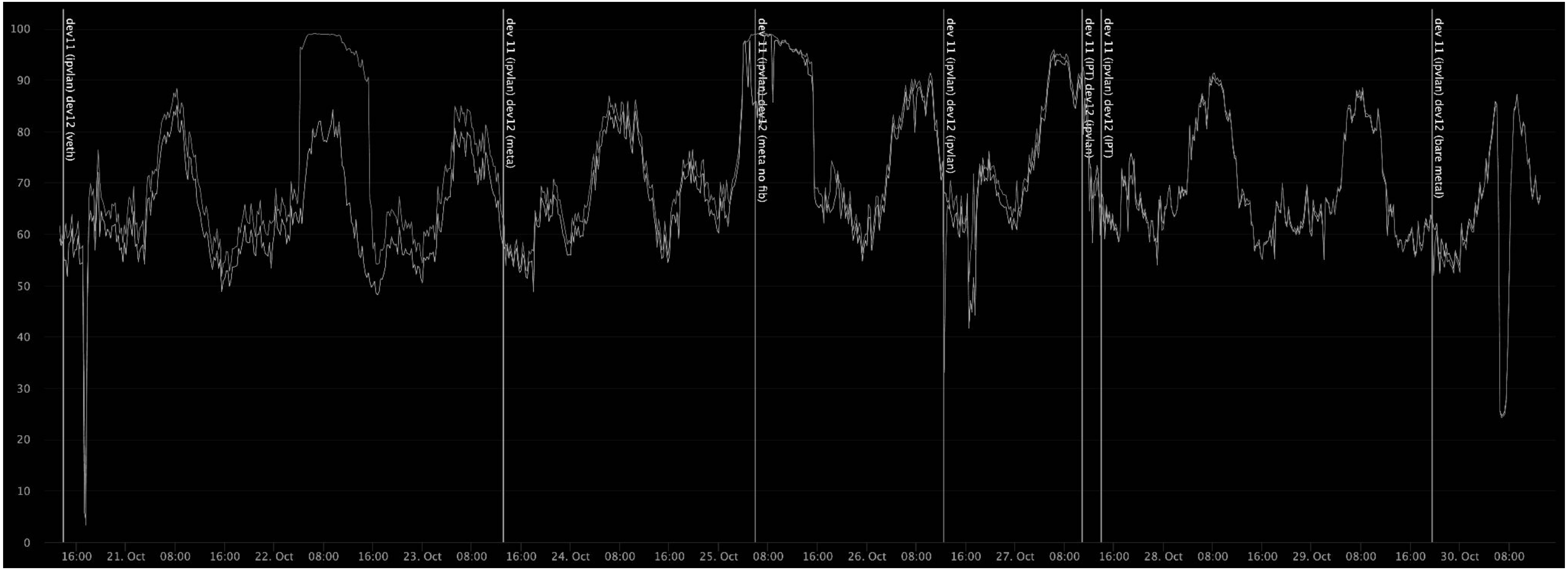
UDP Traffic Redirection over TLS



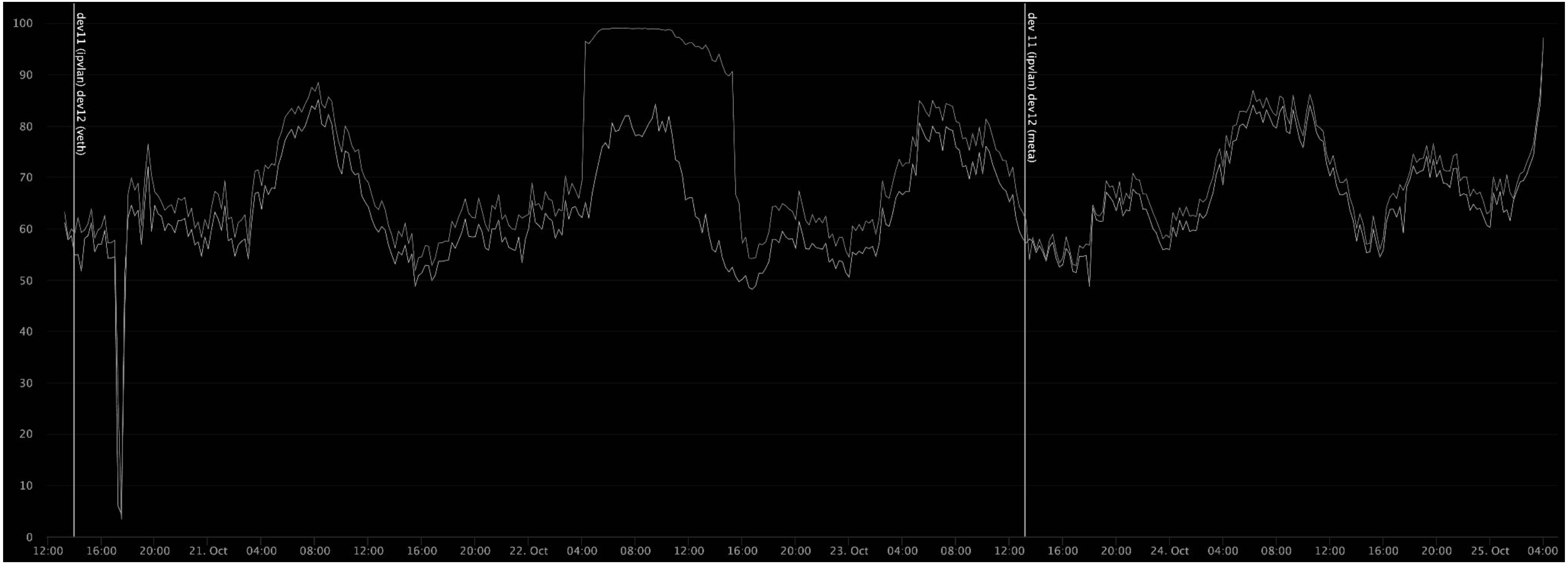
- Challenges
  - Packet makes a round trip from veth0-egress to host-end and back to task’s netns
  - Ingress program at veth0 is in-effective for ttls-fwd-proxy → user client.
- Current Solution:
  - Ingress program needed to attach at “lo” due to kernel optimizing the route.
  - bpf_redirect(“eth0”→ifindex, BPF_F_INGRESS) & update MAC
  - Change direction from EGRESS to INGRESS

04 Other Virtual Devices & Performance Improvements

ipvlan/veth/netkit/bare-metal



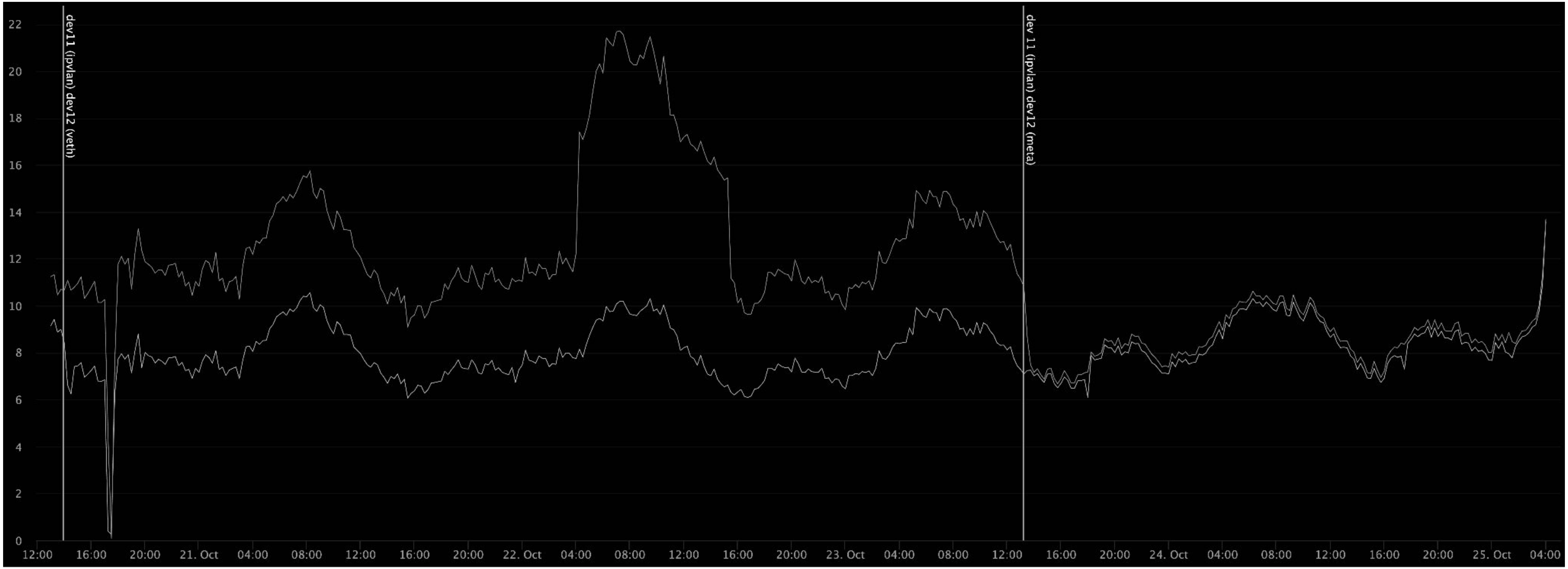
veth -> netkit (cpu-util)



Meta

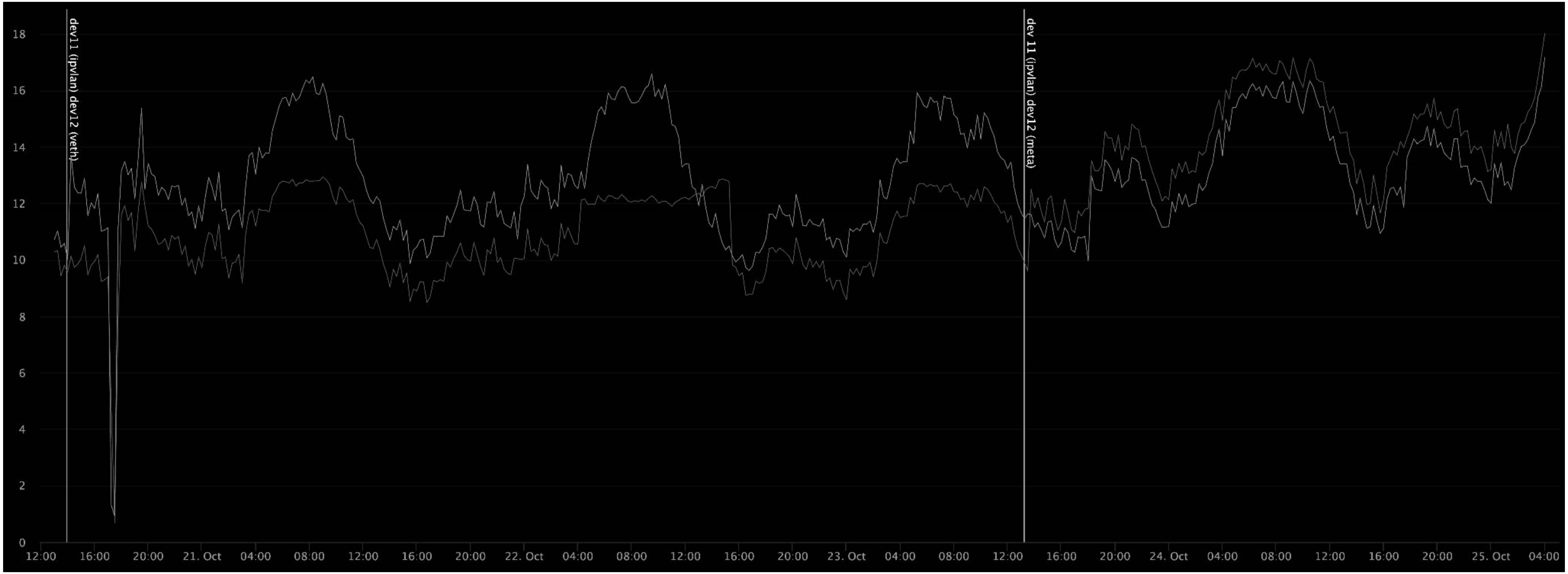
veth -> netkit (cpu-sys + cpu-softirq)

veth -> netkit (cpu-softirq)



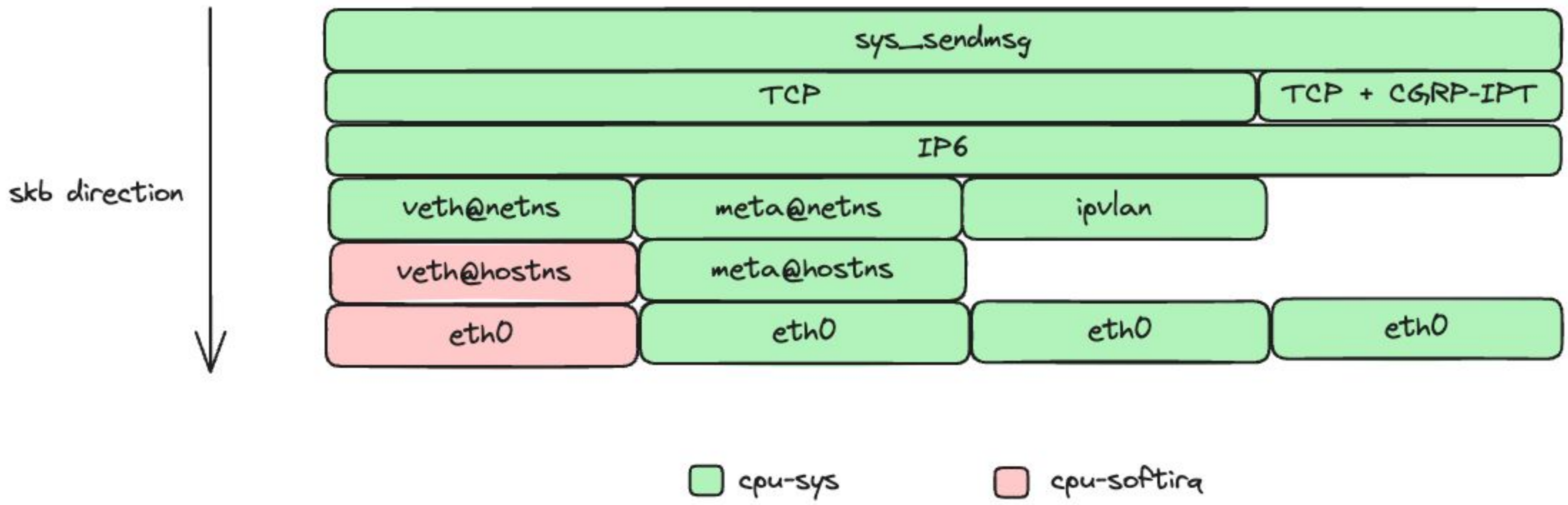
Meta

veth -> netkit (cpu-sys)



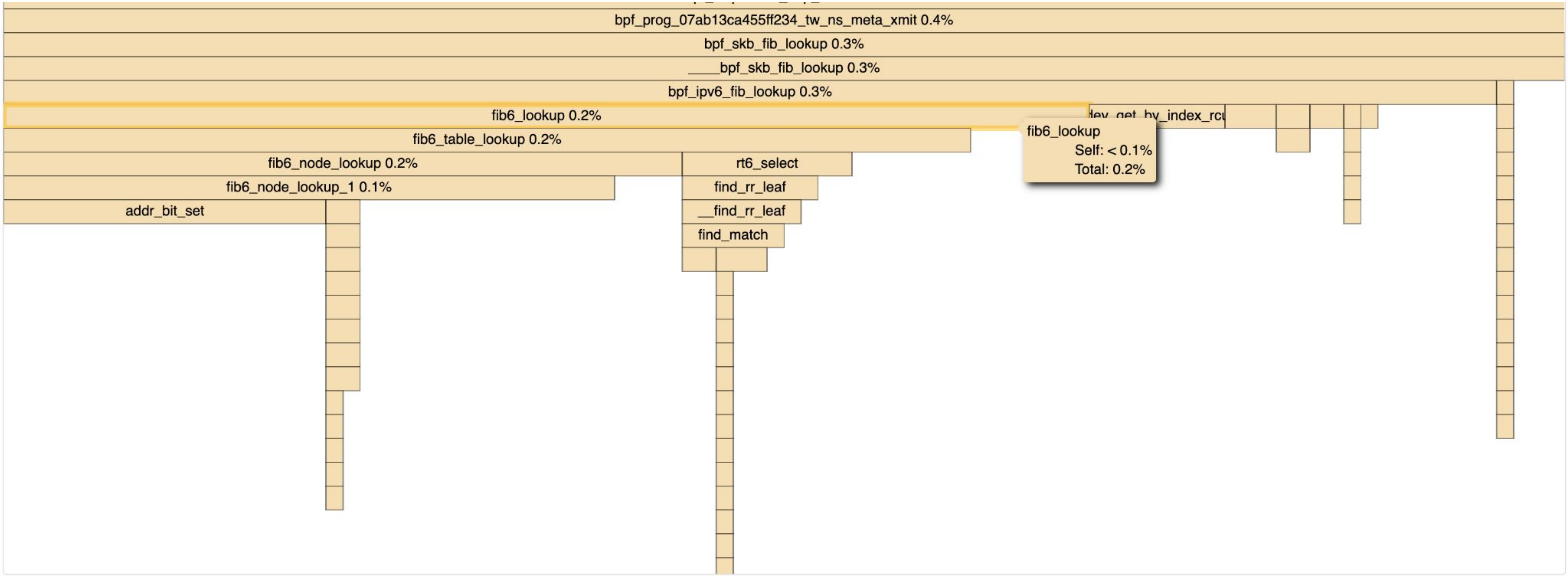
Meta

egress veth/netkit => phy-eth0

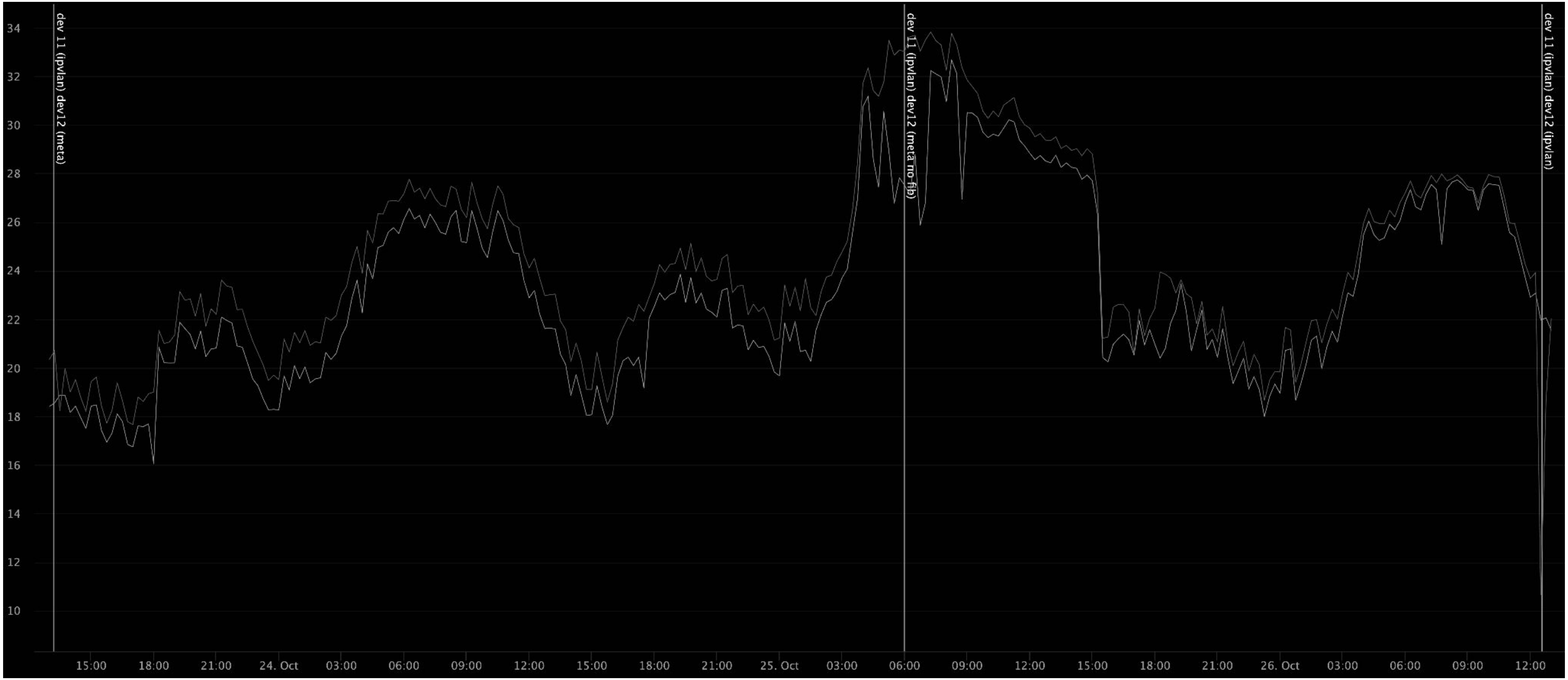


Meta

bpf prog at netkit



netkit at L2 mode (cpu-sys + softirq)



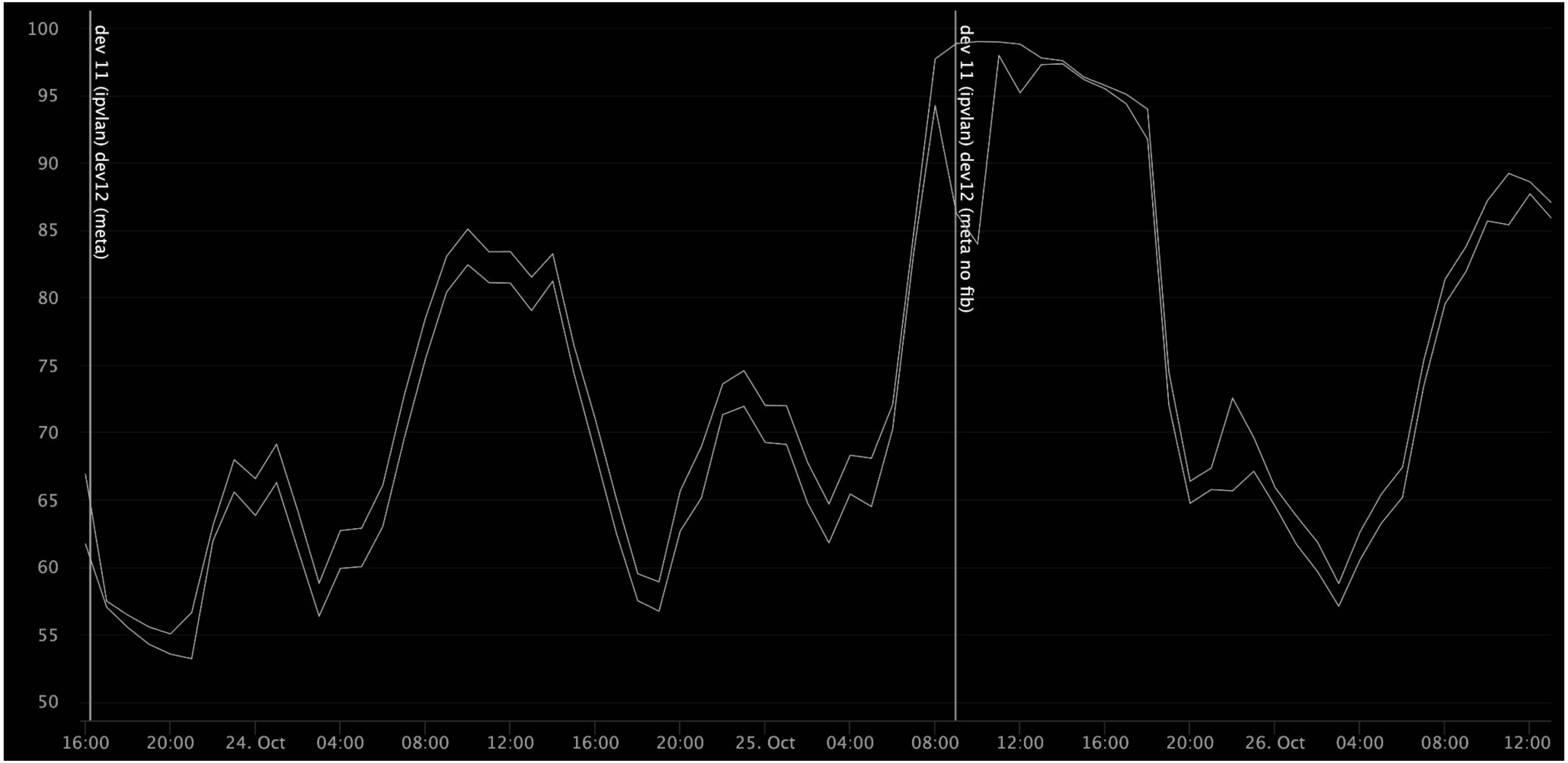
tcp_rr 500 flows 36 threads 1 byte req/rep

	A	B	C	D	E
1	1 x task	no netns (bare metal)	ipvlan L2	meta L2 (no fib lookup)	veth L2 (no fib lookup)
2	cpu-util	84.78	85.19	85.85	87.09
3	cpu-softirq	2.48	2.85	2.17	14.78
4	cpu-sys	63.24	63.73	64.79	53.89
5	cpu-user	18.46	18.09	18.38	17.89
6	#Transactions (M) / s	1.63	1.61	1.62	1.62
7	trans_per_s (K)/ cpu%	19.17	18.90	18.88	18.61

Meta

Meta

Ipvlan vs meta (L2)



Meta

ipvlan vs ipvlan (background difference)