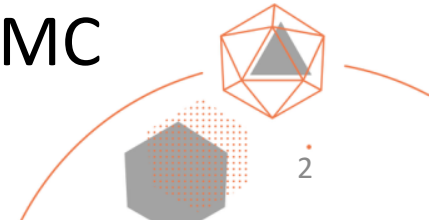
LINUX PITER

2019 OCTOBER 4-5
SAINT PETERSBURG

LINUX PITER
2019 OCTOBER 4-5
SAINT PETERSBURG

Linux networking stack in enterprise storage

Dmitry Krivenok
Dell EMC

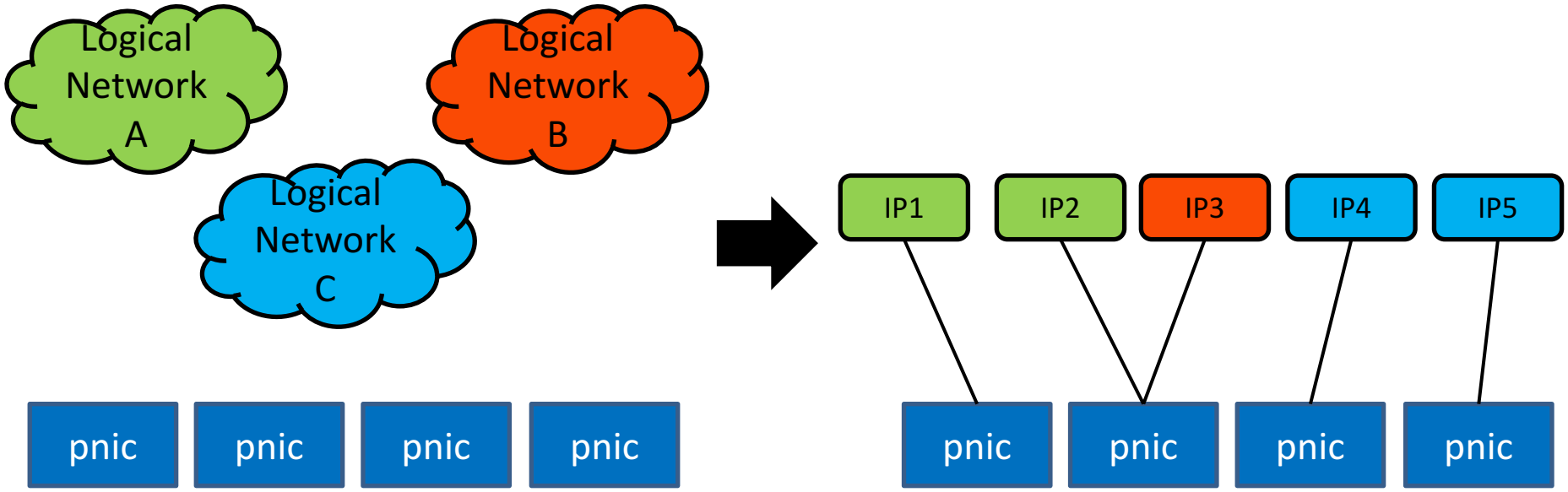


This talk

- Not about storage
- Not about kernel bypass technologies (DPDK, RDMA, etc.)
- It’s about our experience using Linux network stack in storage products

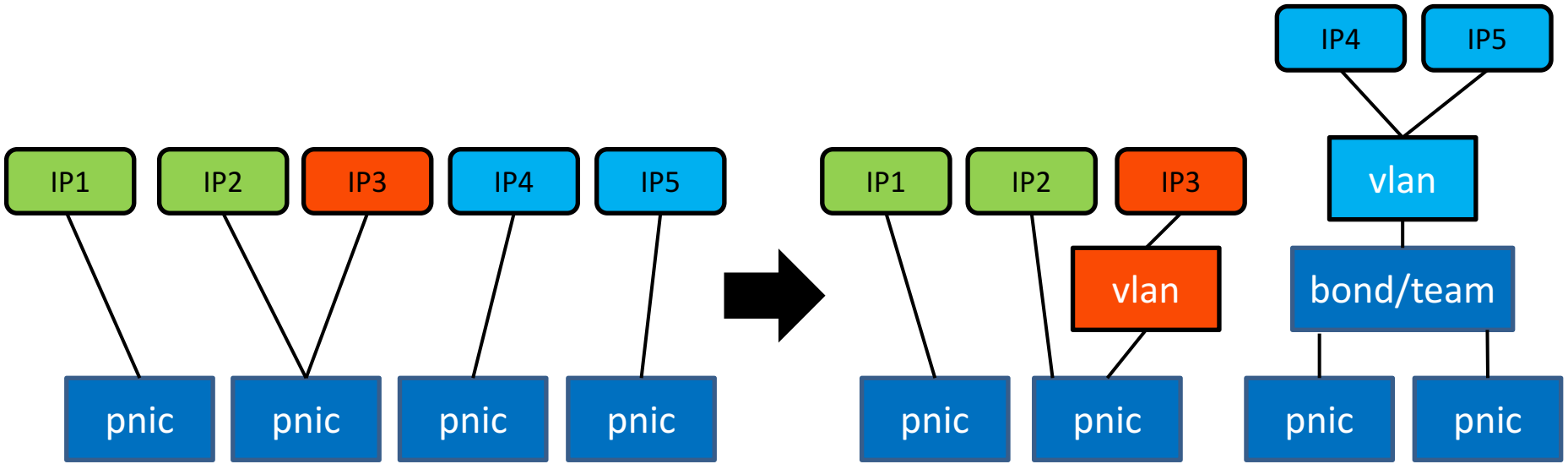
net_device story

Can we just use a flat network model?



Not really…

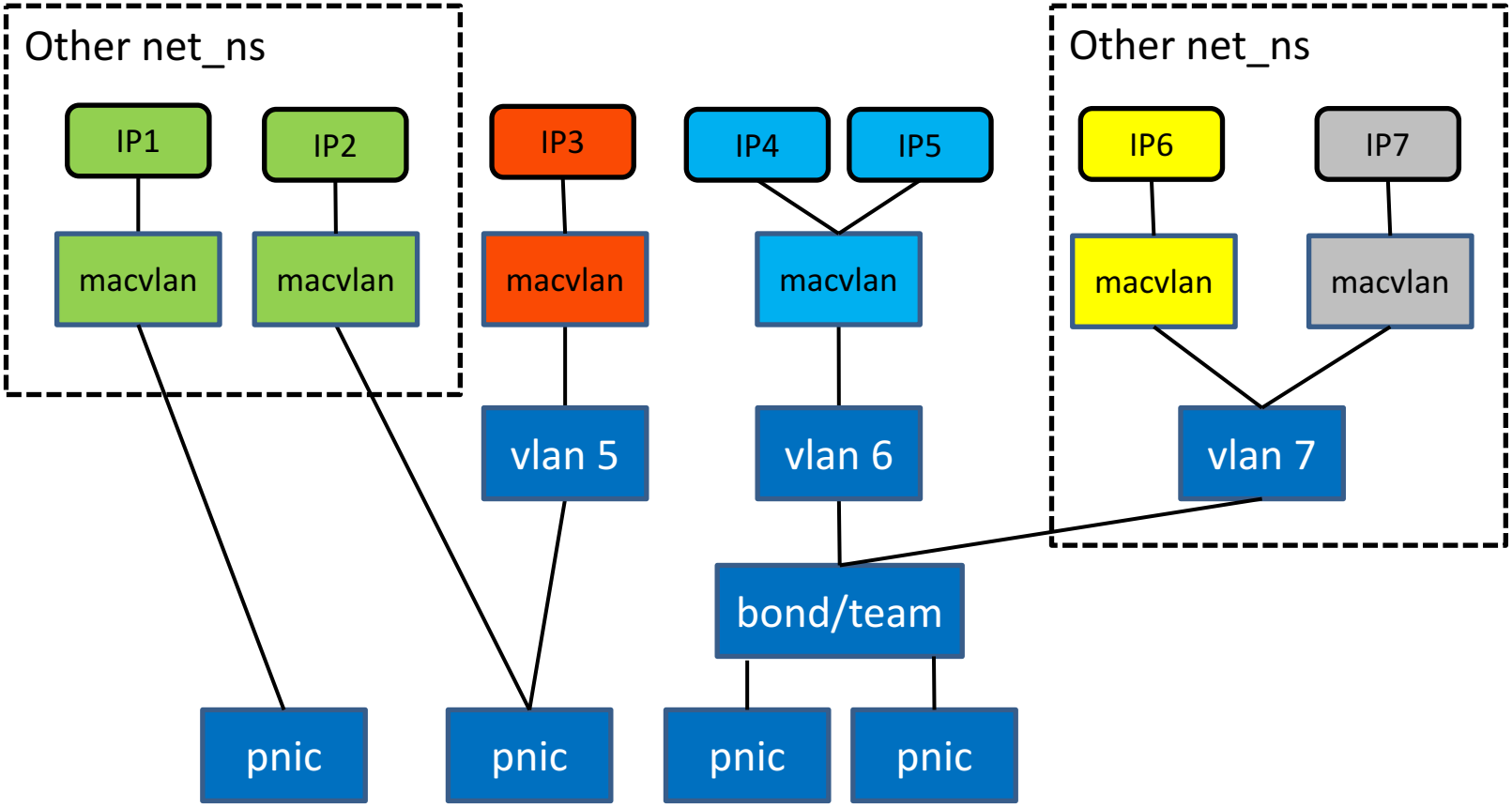
- We need to handle VLAN tagged traffic
- We need network-level HA for some clients without native multi-pathing



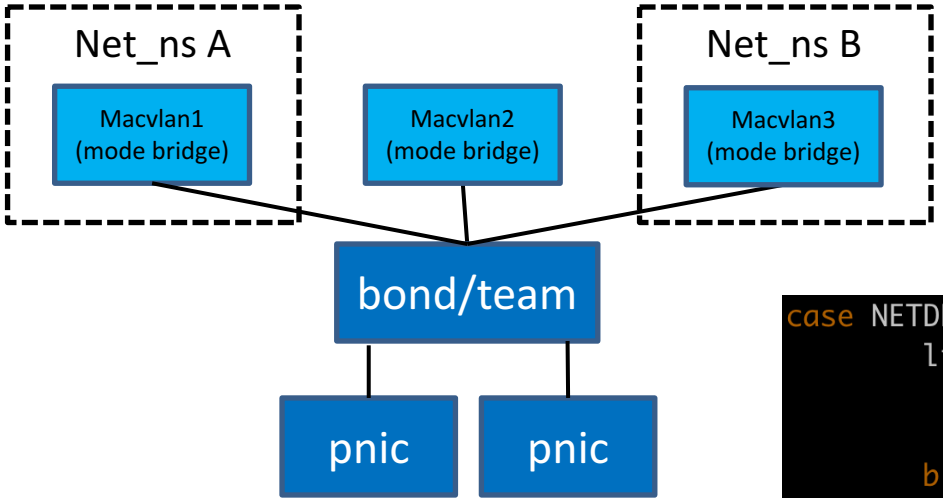
Still not enough

- No way to move a logical network or its part to another namespace
- It’s hard to change MTU independently for different logical networks
  - no net_device, need to set it per route
- More difficult TC, firewall, xfrm and PBR implementation
- No way to group the devices of one logical network
  - e.g., to shutdown them all together
- Etc

Okay, just create MACVLANs



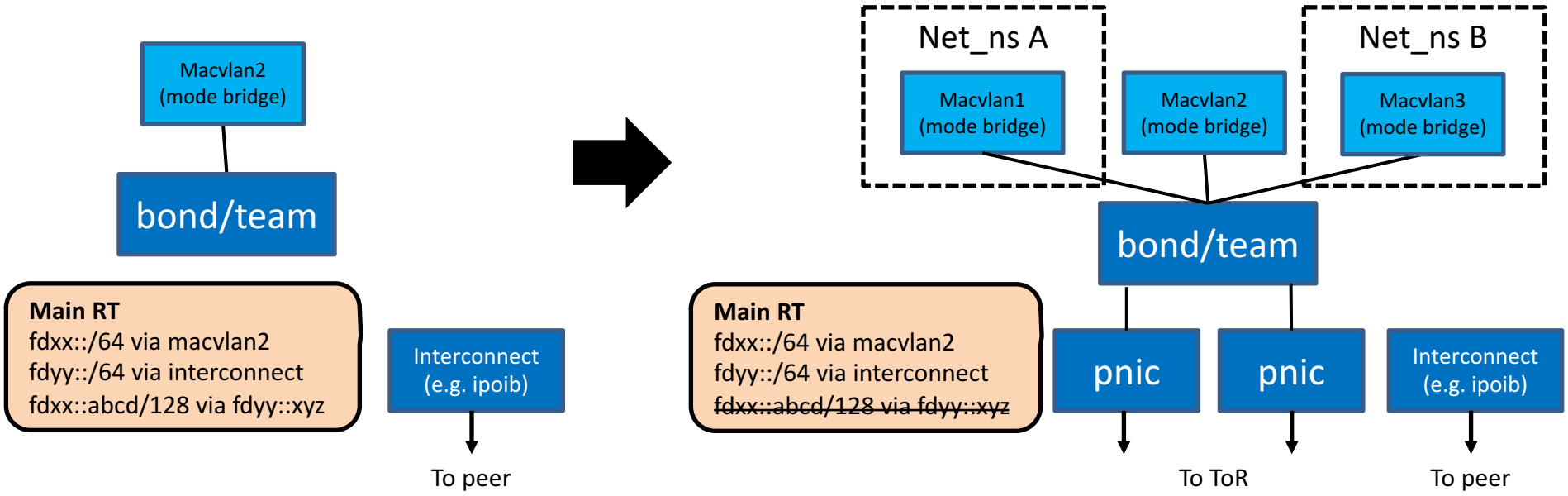
MACVLAN and oper state propagation



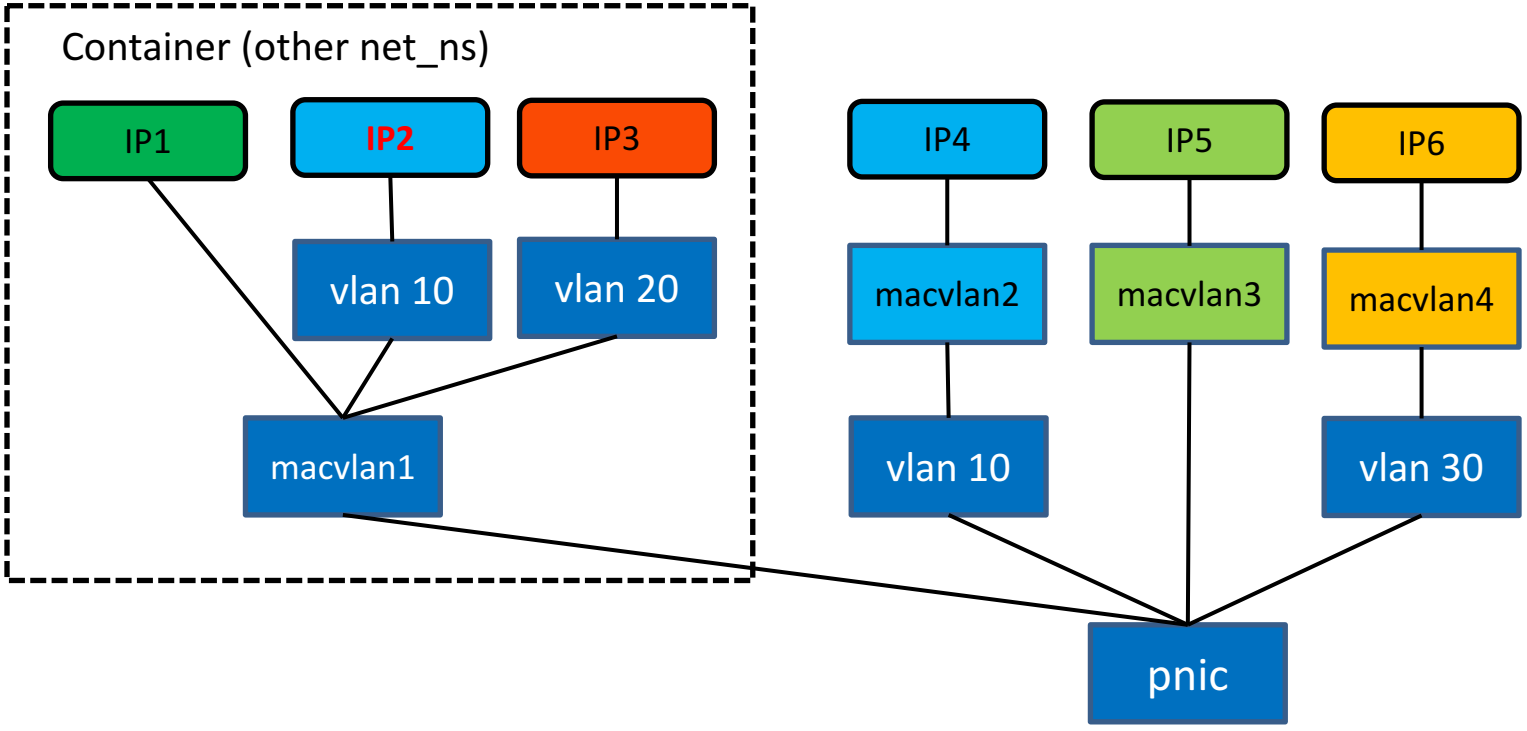

```
case NETDEV_CHANGE:
        list_for_each_entry(vlan, &port->vlans, list)
                netif_stacked_transfer_operstate(vlan->lowerdev,
                                                 vlan->dev);
        break;
```

- In bridge mode, sibling MACVLANs communicate directly (in-memory)
- If you pull cables from underlying ports, oper state will be transferred up through the hierarchy and MACVLANs will report NO_CARRIER
- The *in-memory* communication between MACVLANs will fail
- Can be solved in many ways, we just disabled propagation for MACVLANs in bridge mode (driver patch, unconditional change for now)

Hot-plug trick

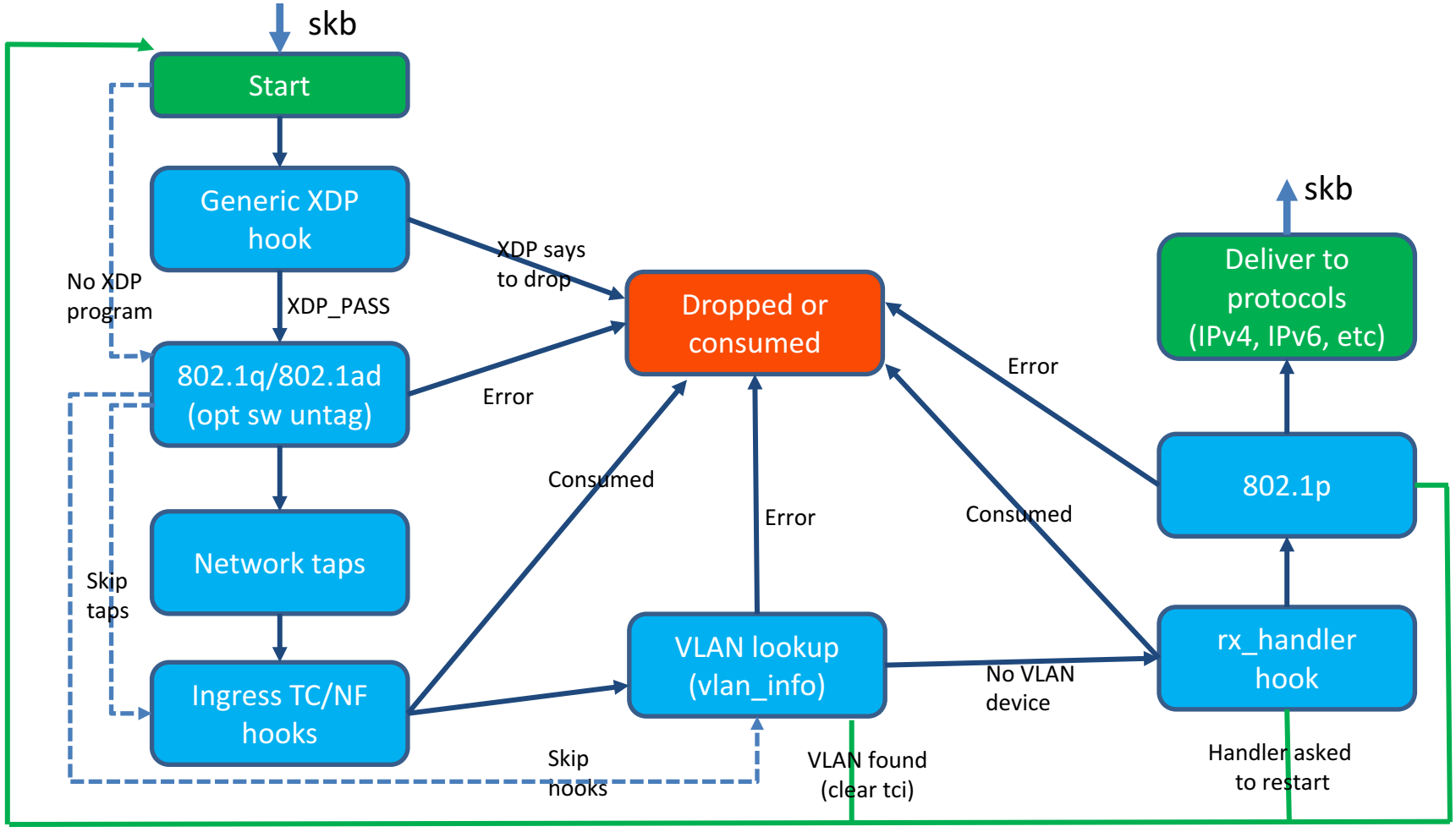


Asymmetric VLAN configuration

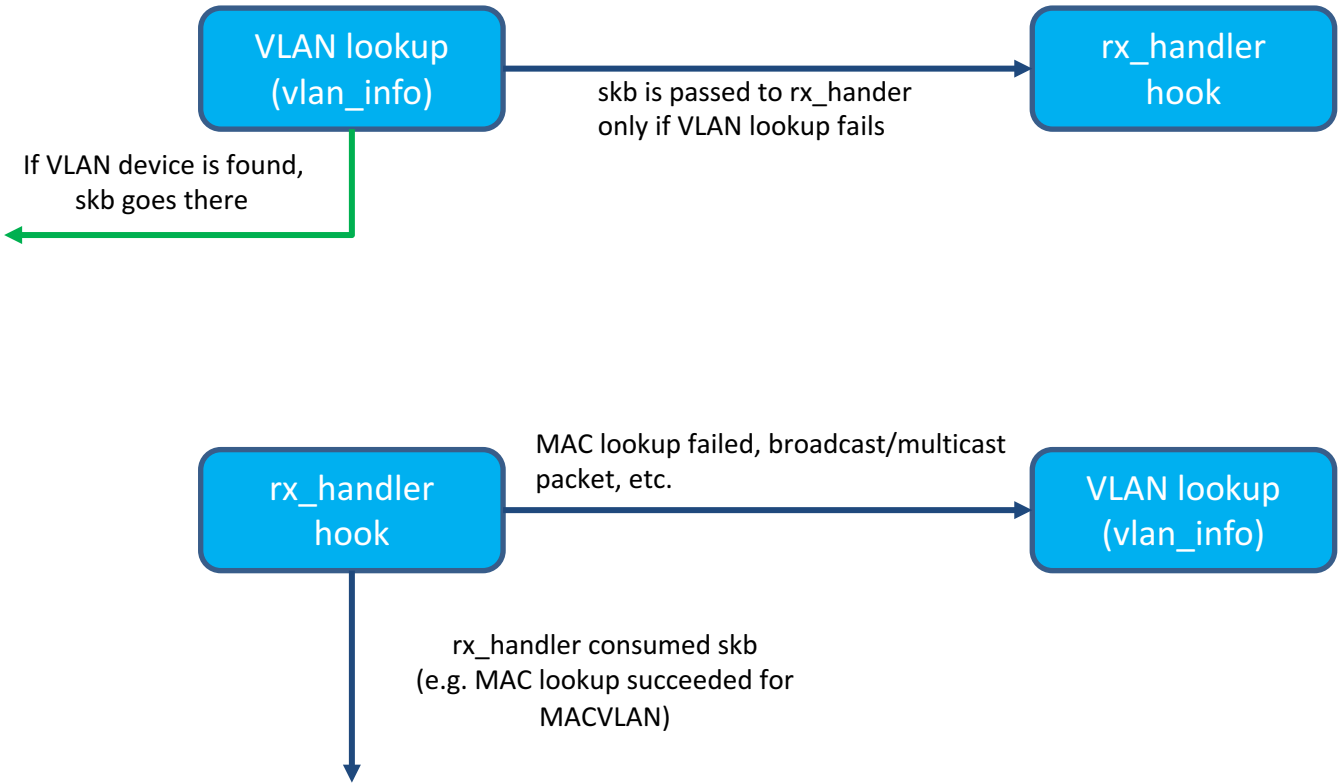


- E.g., different network manager in the system container (other device layout)
- Sharing of the same VLAN ID is not going to work
- Address IP2 will not be reachable from the outside

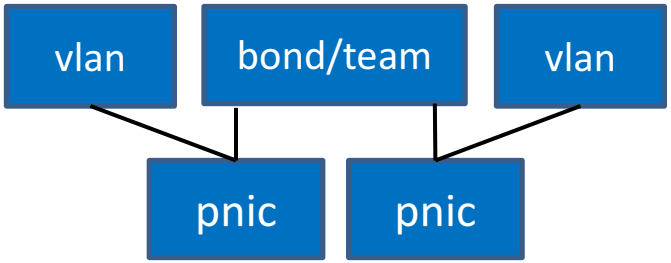
__netif_receive_skb_core()

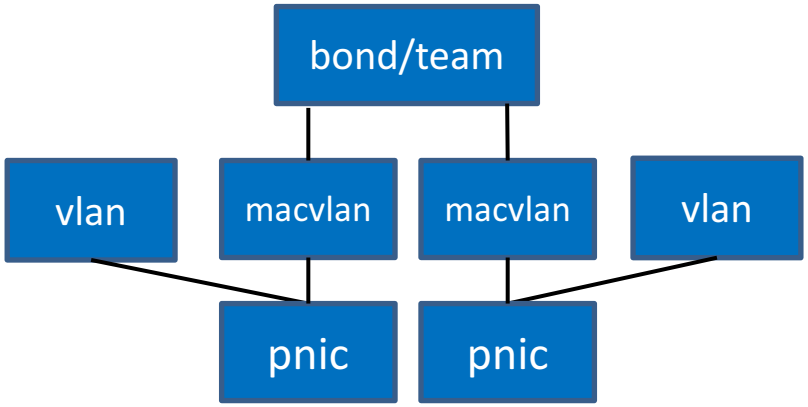


Can we just change the order?



No, some configurations will break





- In some setups dedicated VLANs are used to steal traffic from a/b bonds
- If we change the order, we will break this
- Make order configurable?
  - per lower net_device?
  - per VLAN ID?
  - per skb (e.g., based on TC provided hints)?

MAC addresses proliferation

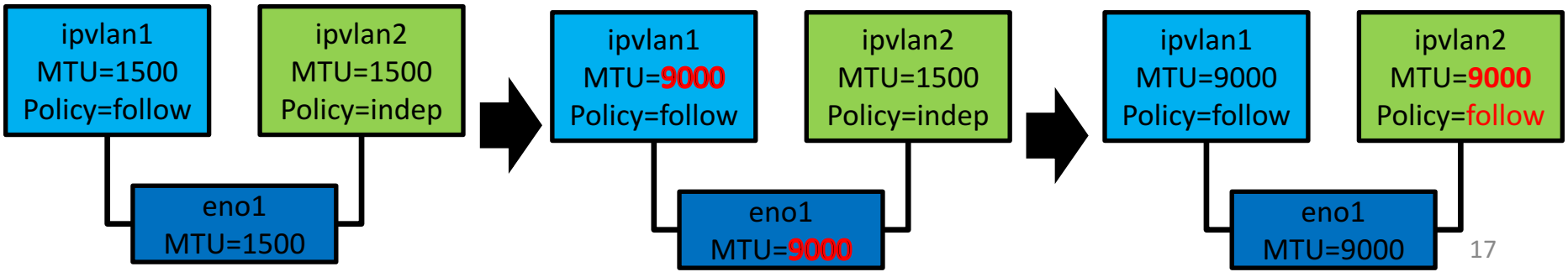
- As we scale the configuration and add networks, we need to add more MACVLANs
- Each MACVLAN has a unique auto-generated unicast MAC address
- UC/MC lists must be propagated down to the NIC (limited space)
- MAC tables on the switches are limited
- Problematic in the virtual deployment of the stack (HCI, SDS)
  - e.g., on VMware ESXi vSwitches don’t do learning
  - all VM MACs are supposed to be assigned by the hypervisor
  - any auto-generated MAC behind vNIC won’t be reachable by default
  - works only with non-standard security settings (promisc mode, forged transmit)
  - causes network model divergence (flat vs. hierarchical)

Is IPVLAN better?

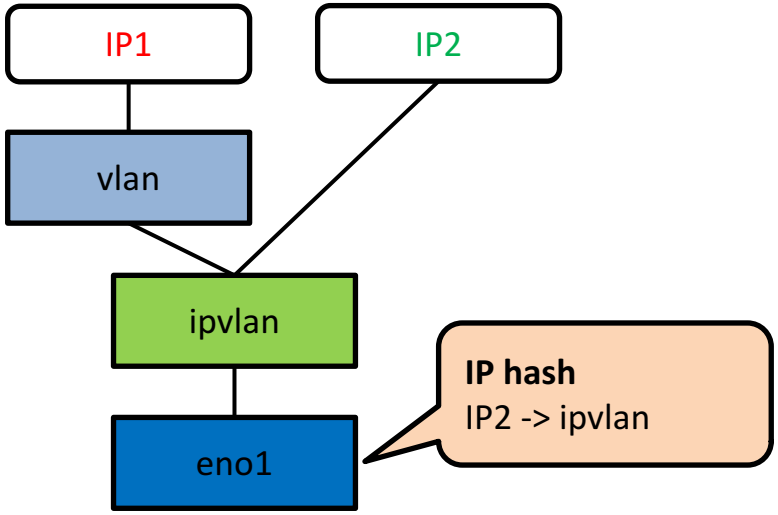
- Shares MAC with the lower net_device (i.e., single MAC behind pNIC/vNIC)
- Supports L2 mode
- But has its own issues…

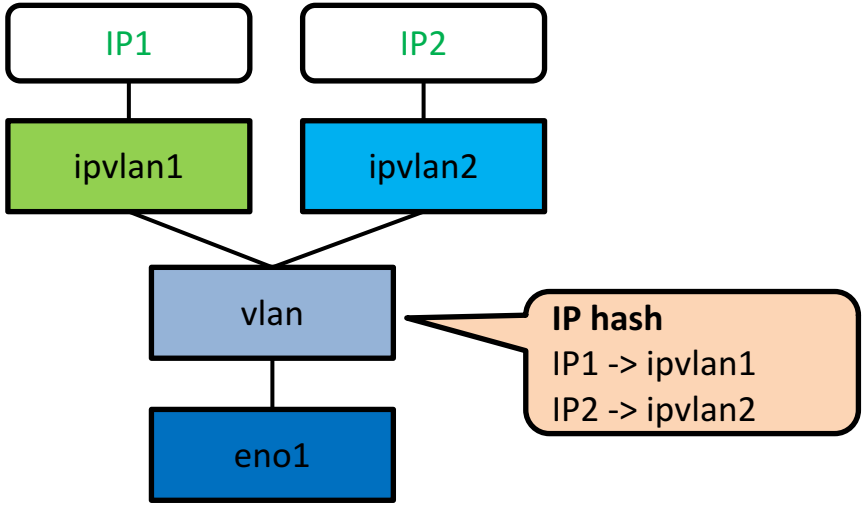
IPVLAN MTU handling

- ipvlan always follows lower net_device's MTU
- There was an attempt to fix that before
  - https://patchwork.ozlabs.org/patch/857994/
  - Backward-compatibility concerns
- We introduced per-ipvlan mtu_policy
  - Defaults to IPVLAN_MTU_POLICY_FOLLOW_PARENT
  - Can be changed to IPVLAN_MTU_POLICY_INDEPENDENT
  - Can be set at creation time and changed later via netlink
  - The patch will be submitted soon (pending iproute2 changes)



VLAN over IPVLAN





- Was allowed before, but never worked (IP1 is unreachable in this example)
- Explicitly disallowed since 3518e40b3cd8e (kernel 4.16)

```
$ sudo ip link add link ipvlan2 name ipvlan2.10 type vlan id 10
Error: 8021q: VLANs not supported on device.
$
```

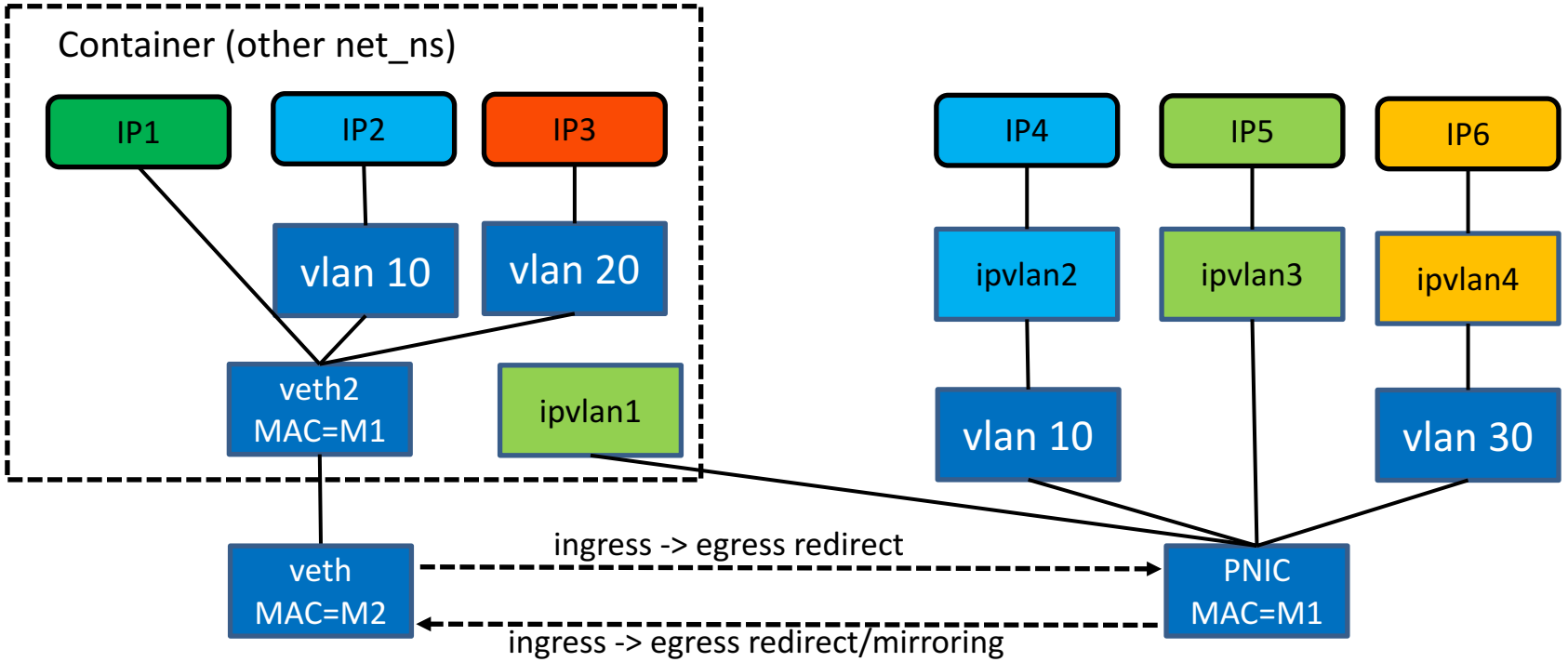
VLAN over IPVLAN, details

```
$ sudo ip link add link enp0s10 name ipvlan2 type ipvlan mode l2
$ sudo ip link set up dev ipvlan1
$ sudo ip link set up dev ipvlan2
$ sudo ip link add link ipvlan2 name ipvlan2.10 type vlan id 10
$ sudo ip link set up dev ipvlan2.10
$ sudo ip a add 192.168.5.5/24 dev ipvlan1
$ sudo ip a add 192.168.10.5/24 dev ipvlan2.10
$ ip link show | grep ^1[13]:
11: ipvlan1@enp0s10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN
13: ipvlan2.10@ipvlan2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP m
$
```

```
$ ptrace probe:ipvlan_*
     0.016 ip/9566 probe:ipvlan_addr4_event:(ffffffffc07dc8e4) ifindex=11
     0.028 ip/9566 probe:ipvlan_add_addr:(ffffffffc07dc7e0) ifindex=11
 18393.844 ip/9568 probe:ipvlan_addr4_event:(ffffffffc07dc8e4) ifindex=13
```

- Only address on the IPVLAN gets added to the hash, so lookup of the 2nd address will fail in the rx_handler and skb will be dropped
- There is a single hash in IPVLAN master device, no support of per-VLAN hashes
- Does it make sense to add it? Probably no…

Alternative approach



- Use IPVLAN for untagged traffic
- Add veth pair for tagged/untagged traffic, change MAC in a container (hack)
- Setup redirection and mirroring in the default namespace

Alternative approach, options

- TC ingress hooks
- TC filters + mirred action
  - e.g., via a slightly modified basic filter and ipsets
  - Redirect vs. mirror depending on the filter
- TC cls_bpf in the direct-action mode
  - Classification in BPF program using user-space provided maps
  - Redirect/mirroring via bpf_redirect/bpf_clone_redirect helpers
- Issues
  - Control plane dependencies (not transparent)
    - Need to update filters/maps
    - Need to sync VIDs
  - Hard to scale beyond one device

Zeroconf

Zeroconf issues

- IPv4/IPv6 LL + mDNS + DNS-SD
  - SSDP is another option
- Works fine on servers in DC networks
- Surprisingly, many issues on client machines
  - Multi-homing hosts
  - Firewalls blocking mDNS
  - IPv6 LL in browsers
  - Bugs in mDNS libraries
  - …
- Need a simple plan B…

IPv4LL address claiming

```
$ sudo avahi-autoipd disc0
Found user 'avahi-autoipd' (UID 170) and group 'avahi-autoipd' (GID 170).
Successfully called chroot().
Successfully dropped root privileges.
Starting with address 169.254.11.131
Callout BIND, address 169.254.11.131 on interface disc0
Successfully claimed IP address 169.254.11.131

$ sudo tcpdump -nn -i disc0
04:35:42.969311 ARP, Request who-has 169.254.11.131 tell 0.0.0.0, length 28
04:35:44.321825 ARP, Request who-has 169.254.11.131 tell 0.0.0.0, length 28
04:35:45.936737 ARP, Request who-has 169.254.11.131 tell 0.0.0.0, length 28
04:35:47.938216 ARP, Request who-has 169.254.11.131 tell 169.254.11.131, length 28
04:35:49.939246 ARP, Request who-has 169.254.11.131 tell 169.254.11.131, length 28
```

Announcement phase

Probing phase

Zeroconf workaround via BPF

```
__section("egress")
int tc_egress(struct __sk_buff *skb)
{
        void *data_end = (void *)(long)skb->data_end;
        void *data = (void *)(long)skb->data;
        uint32_t arp_offset = sizeof(struct eth_hdr) + sizeof(struct arp_hdr);

        if(!is_arp_valid_arp_for_ipv4(data, data_end)) return TC_ACT_OK;
        if(!is_arp_probe_for_upper_range(data + arp_offset)) return TC_ACT_OK;
        transform_to_arp_reply(skb, data, data, arp_offset);
        return bpf_redirect(skb->ifindex, BPF_F_INGRESS);
}
```

```
sudo sh -c "rm -fv /var/lib/avahi-autoipd/*"
clang -O2 -Wall -Werror -target bpf -c avahi.c -o avahi.o || exit 1
sudo tc qdisc add dev $DEV clsact || exit 1
sudo tc filter add dev $DEV egress bpf da obj avahi.o sec egress || exit 1
sudo avahi-autoipd $DEV
```

Zeroconf workaround via BPF

```
$ sudo avahi-autoipd disc0
Found user 'avahi-autoipd' (UID 170) and group 'avahi-autoipd' (GID 170).
Successfully called chroot().
Successfully dropped root privileges.
Starting with address 169.254.10.102
Received conflicting normal ARP packet.
Trying address 169.254.96.118
Received conflicting normal ARP packet.
Trying address 169.254.143.216
Callout BIND, address 169.254.143.216 on interface disc0
Successfully claimed IP address 169.254.143.216

$ sudo tcpdump -nn -i disc0
04:41:11.287018 ARP, Reply 169.254.10.102 is-at 02:03:04:05:06:07, length 28
04:41:12.203103 ARP, Reply 169.254.96.118 is-at 02:03:04:05:06:07, length 28
04:41:13.130931 ARP, Request who-has 169.254.143.216 tell 0.0.0.0, length 28
04:41:14.253264 ARP, Request who-has 169.254.143.216 tell 0.0.0.0, length 28
04:41:15.817976 ARP, Request who-has 169.254.143.216 tell 0.0.0.0, length 28
04:41:17.824230 ARP, Request who-has 169.254.143.216 tell 169.254.143.216, length 28
04:41:19.826067 ARP, Request who-has 169.254.143.216 tell 169.254.143.216, length 28
```

No ARP requests for matching packets
(TC egress hook is before taps)

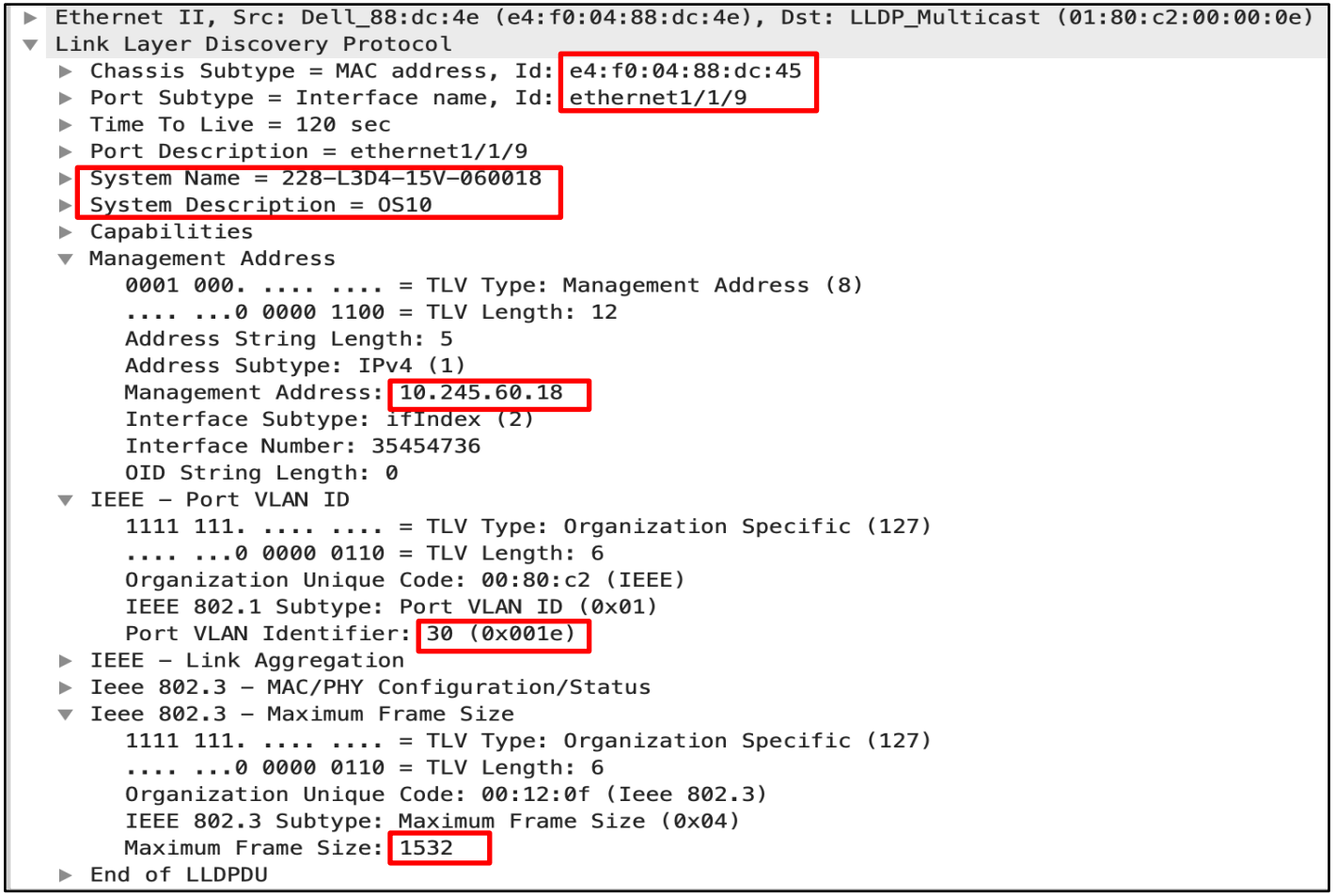
Note, too narrow criteria will
cause rate limiting

When BPF fails…

Link-layer discovery

- L2 protocols (LLDP aka IEEE 802.1ab, Cisco CDP, etc.)
- Provide a lot of useful info about your neighbor device (switch, router, host)
  - Name/ID of the peer device and port
  - Management IP
  - Native and allowed VLANs
  - MTU and LAG info
  - And more
- Our use-cases
  - Automatic configuration, network validation and visualization
  - Network troubleshooting
- Linux support
  - No in-kernel support
  - lldpd daemon + cli (e.g., https://github.com/vincentbernat/lldpd)

LLDP example



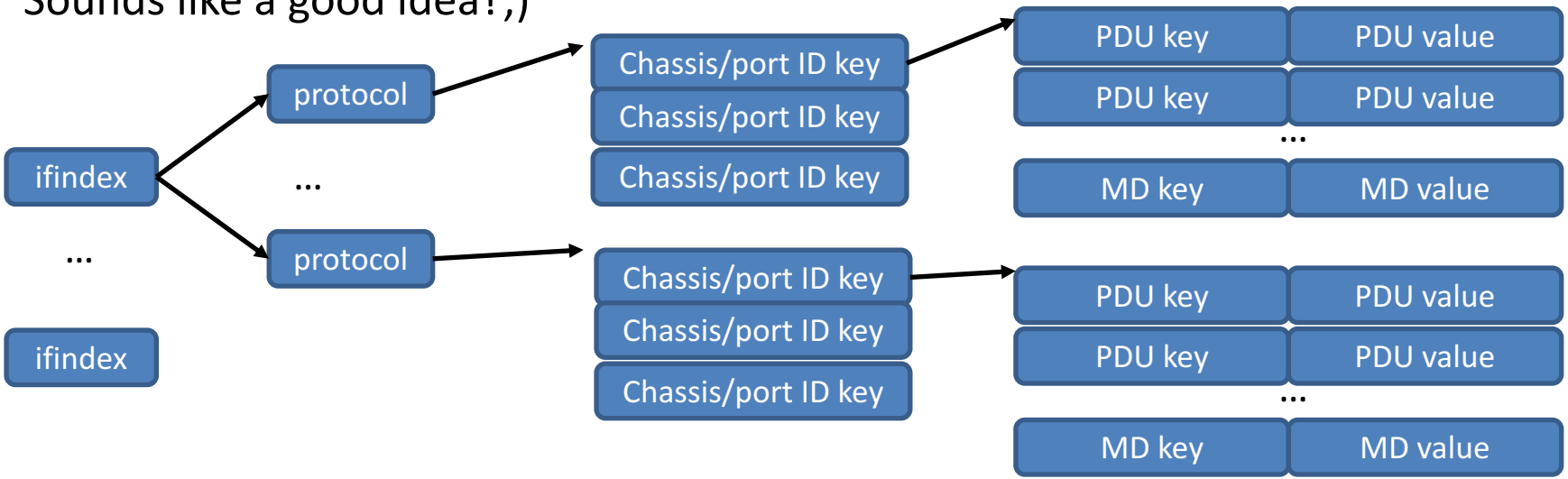
Some not trivial issues

- Fake PDUs is a real problem
  - Especially in the multi-vendor environment
  - Switches pass what they are not supposed to
  - Usually issues with CDP, but seen LLDP as well
- Dual-protocol devices
  - Often need information from both, can’t just pick LLDP
- Historical data is important
  - Need to store N PDUs for a valid chassis/port ID pair per protocol

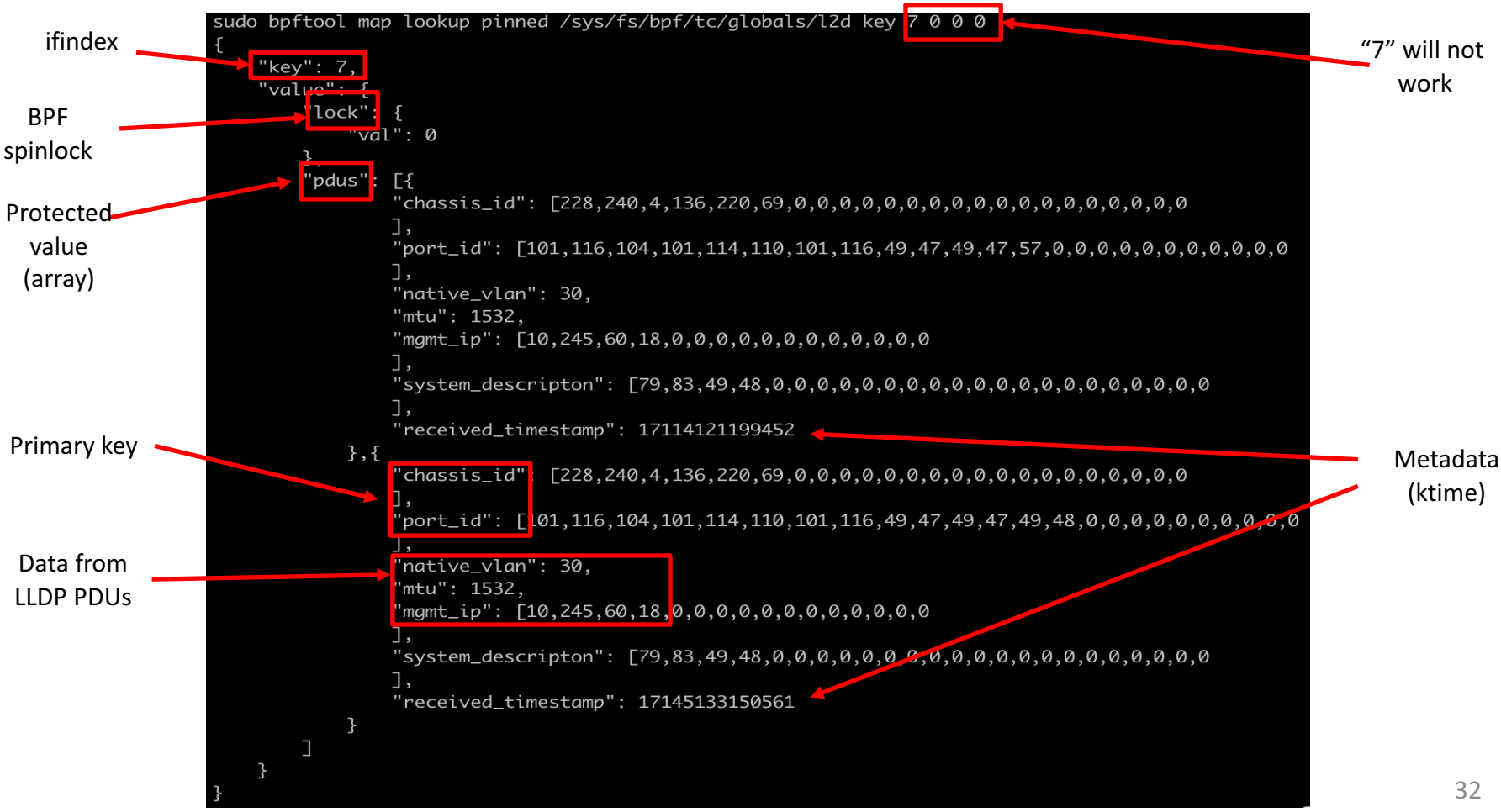
- User-space daemon is required
  - Okay in most cases
  - Problematic in some corner cases (DC after crashes, failed upgrades, etc.)

Can I do it in BPF?

- We have TC ingress hooks to attach BPF programs to
- We can parse LLDP/CDP packets and filter them in BPF
  - Filtering means dropping fake PDUs (optionally saving)
- We can store them in BPF maps as they are received
  - We can store the last N unique chassis/port ID PDUs
- We can read the content of the maps from the user-space!
- Sounds like a good idea?;)



Yes, you can…but think twice!

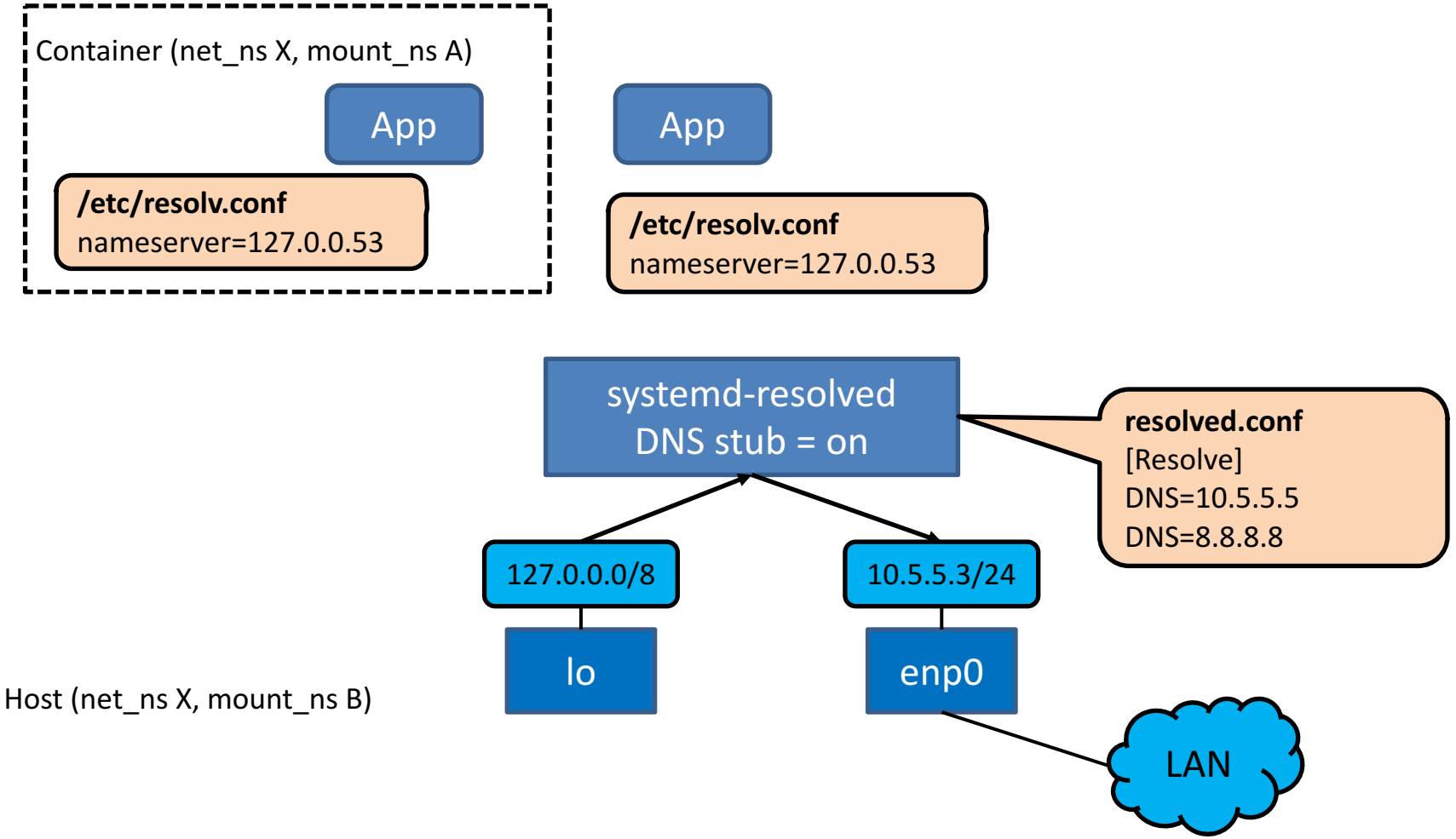


Issues

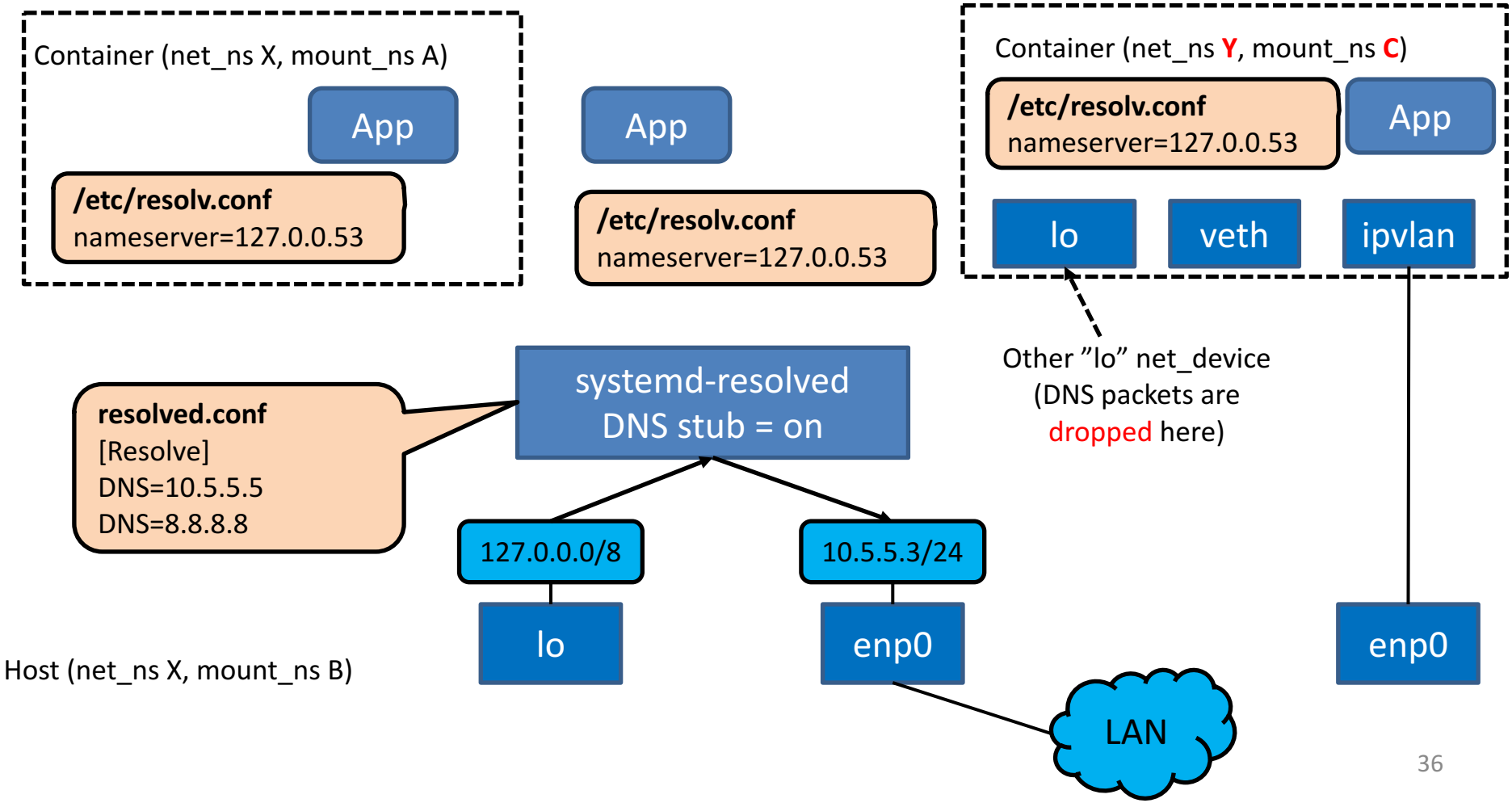
- Parsing of TLV based protocols is very painful!
  - Can be done, but with very ugly hacks
- Maps
  - Map-of-maps, but not map-of-maps-of-maps
  - Can add one level with BPFFS and custom loader
- Concurrency
  - Very restricting BPF spinlocks (see linux/bpf.h)
  - Can only protect map *value* in 3 kinds of maps, no protection of maps
  - Can't build your own primitives on top of spinlocks
  - Per-CPU maps can be used, but need user-space processing
  - Hack for slow protocols like LLDP?
    - XDP redirect to one CPU core and process there with preemption off
    - Not supported for generic XDP today
- Some pain points seen by others:
  - https://mbertrone.github.io › documents › 18-eBPF-experience

systemd-resolved

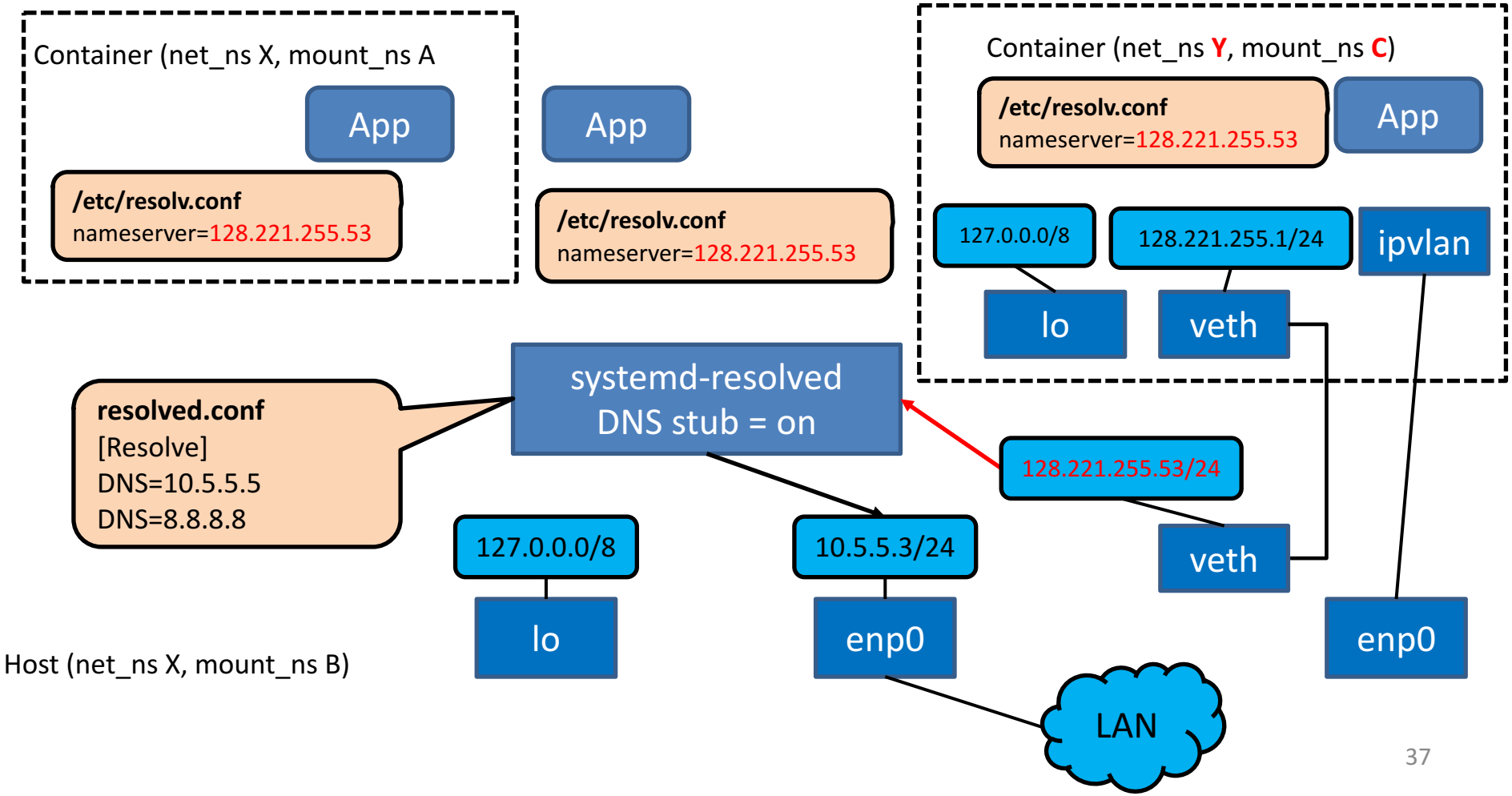
Systemd-resolved DNS stub



Systemd-resolved DNS stub, 2nd net_ns



Will this work?



No…

- 127.0.0.53:53 is not configurable
- The IP address is hard-coded in systemd-resolved

```
8 /* 127.0.0.53 in native endian */
9 #define INADDR_DNS_STUB ((in_addr_t) 0x7f000035U)
```

```
396 static int manager_dns_stub_udp_fd(Manager *m) {
397         union sockaddr_union sa = {
398                 .in.sin_family = AF_INET,
399                 .in.sin_port = htobe16(53),
400                 .in.sin_addr.s_addr = htobe32(INADDR_DNS_STUB),
401         };
```

The first attempt to solve this - BPF

```c
#define MAKE_IP(a,b,c,d) (a<<24 | b<<16 | c<<8 | d)
#define SYSTEMD_RESOLVED_IP MAKE_IP(127,0,0,53)
#define SYSTEMD_RESOLVED_PORT (53)
#define NEW_IP_TO_BIND MAKE_IP(128,221,255,53)

__section("cgroup/bind4")
int bind_v4(struct bpf_sock_addr *ctx)
{
        // resolved supports only TCP and UDP, skip other stuff
        if (ctx->type != SOCK_DGRAM && ctx->type != SOCK_STREAM) return 1;
        // resolved always binds to 127.0.0.53:53
        if (ntohl(ctx->user_ip4) != SYSTEMD_RESOLVED_IP) return 1;
        if (htonl(ctx->user_port) != SYSTEMD_RESOLVED_PORT << 16) return 1;
        // change the IP to bind to (you can read it the map)
        ctx->user_ip4 = htonl(NEW_IP_TO_BIND);
        return 1;
}
```

```
clang -O2 -Wall -Werror -target bpf -c resolved.c -o resolved.o || exit 1
sudo bpftool prog load ./resolved.o $BPF_FS_PATH type cgroup/bind4 || exit 1
sudo bpftool cgroup attach $CGROUP_FS_PATH bind4 pinned $BPF_FS_PATH || exit 1
```

No luck

```
519         r = socket_bind_to_ifindex(fd, LOOPBACK_IFINDEX);
520         if (r < 0)
521                 return r;
```

- Before bind(), it binds to the “lo” device via SO_BINDTOIFINDEX
  - SO_BINDTODEVICE in the previous version
- Even though the BPF program changed the source IP, socket still has sk_bound_dev_if == 1
- UDP packets are not delivered

Bind to different device from BPF

- BPF_PROG_TYPE_CGROUP_SOCK
  - BPF_CGROUP_INET_SOCK_CREATE – too early to change at creation time
  - BPF_CGROUP_INET4_POST_BIND – can't access bound_dev_if here
- BPF_PROG_TYPE_CGROUP_SOCK_ADDR
  - BPF_CGROUP_INET4_BIND - changed the bound device here
  - Extended the ctx (is that right way? or better do sock lookup?)
- In kernel 5.3, there is setsockopt/getsockopt BPF hooks
  - Didn't try them yet

Still no luck

```
288         if (in_addr_is_localhost(p->family, &p->sender) <= 0 ||
289             in_addr_is_localhost(p->family, &p->destination) <= 0) {
290                 log_error("Got packet on unexpected IP range, refusing.");
291                 dns_stub_send_failure(m, s, p, DNS_RCODE_SERVFAIL, false);
292                 goto fail;
293         }
```

- After changing the bound device for the socket, systemd-resolved started seeing the UDP packets!
- But it immediately dropped them all because both source and destination IPs were not 127.x.y.z
- Good example of defensive programming, but bad for us…

The second attempt - iptables

```
# Stop treating my packets as martians
sudo sysctl -w net.ipv4.conf.veth.route_localnet=1
# Do DNAT 128.221.255.53:53 -> 127.0.0.53:53
# Needed to reach systemd-resolved from other net_ns
# via the internal veth
sudo iptables -t nat -I PREROUTING -i veth -p udp -d 128.221.255.53 --dport 53 \
              -j DNAT --to-destination 127.0.0.53:53
# Do SNAT 128.221.255.1 -> 127.0.0.1
# Needed to pass systemd-resolved 127.x.y.z filter
# for source and destination IPs
sudo iptables -t nat -I INPUT -i veth -p udp -d 127.0.0.53 -s 128.221.255.1 --dport 53 \
              -j SNAT --to-source 127.0.0.1
```

- The above will **not** work if systemd-resolved binds the socket to the device
- Final thing - clear sk_bound_dev_if from BPF program attached to cgroup/bind4
  - Should be possible to skip setting it vs. clearing it in 5.3

Success

```
$ sudo ip netns exec ns1 dig @128.221.255.53 +short google.com
172.217.12.206
$
```

Wait…success?

```
$ sudo dnsmasq -k --interface=veth  --bind-interfaces


$ sudo ip netns exec ns1 dig @128.221.255.53 +short ya.ru
87.250.250.242
$ dig @128.221.255.53 +short ya.ru
87.250.250.242
$
```

P.S. Don’t blame systemd; it works as designed.

To bind or not to bind?

Interface selection

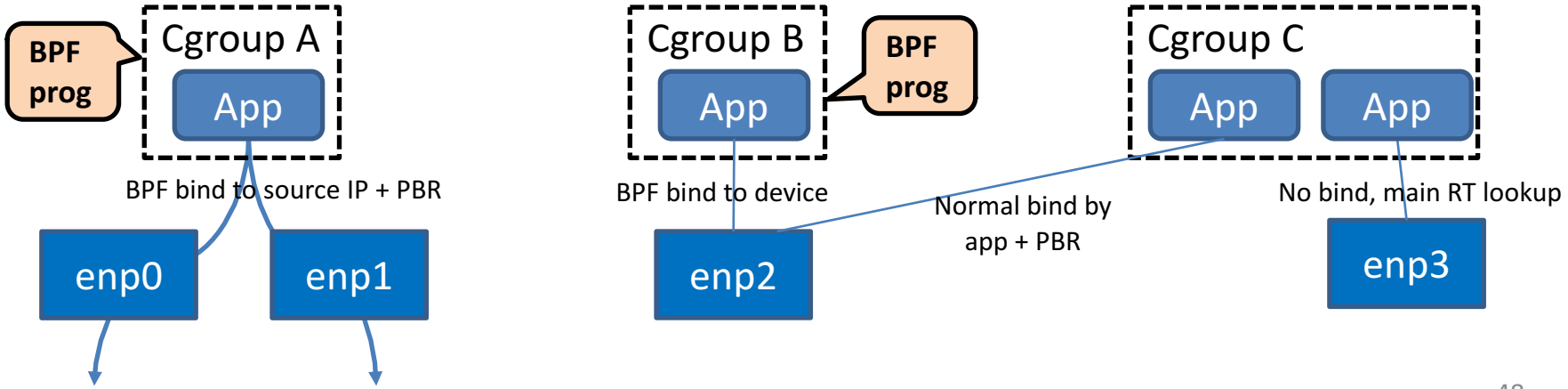
```
$ sudo ip a add 10.10.10.17/24 dev enp0s8
$ sudo ip a add 10.10.10.16/24 dev enp0s10
$ ip r get 10.10.10.10
10.10.10.10 dev enp0s8 src 10.10.10.17 uid 1000
    cache
$ sudo ip a del 10.10.10.17/24 dev enp0s8
$ sudo ip a add 10.10.10.17/24 dev enp0s8
$ ip r get 10.10.10.10
10.10.10.10 dev enp0s10 src 10.10.10.16 uid 1000
    cache
$ ip r get 10.10.10.10 from 10.10.10.16
10.10.10.10 from 10.10.10.16 dev enp0s10 uid 1000
    cache
$ ip r get 10.10.10.10 from 10.10.10.17
10.10.10.10 from 10.10.10.17 dev enp0s10 uid 1000
    cache
$
```

```
$ sudo ip a add 10.10.10.17/24 dev enp0s8
$ sudo ip a add 10.10.10.16/24 dev enp0s10
$ sudo ip r del 10.10.10.0/24 dev enp0s10 table main
$ sudo ip r add 10.10.10.0/24 dev enp0s10 table myrt
$ sudo ip rule add from 10.10.10.16 lookup myrt
$ ip r get 10.10.10.10
10.10.10.10 dev enp0s8 src 10.10.10.17 uid 1000
    cache
$ ip r get 10.10.10.10 from 10.10.10.17
10.10.10.10 from 10.10.10.17 dev enp0s8 uid 1000
    cache
$ ip r get 10.10.10.10 from 10.10.10.16
10.10.10.10 from 10.10.10.16 dev enp0s10 table myrt uid 1000
    cache
$
```

- PBR usually works fine
- but for some apps, it’s problematic to do the explicit bind

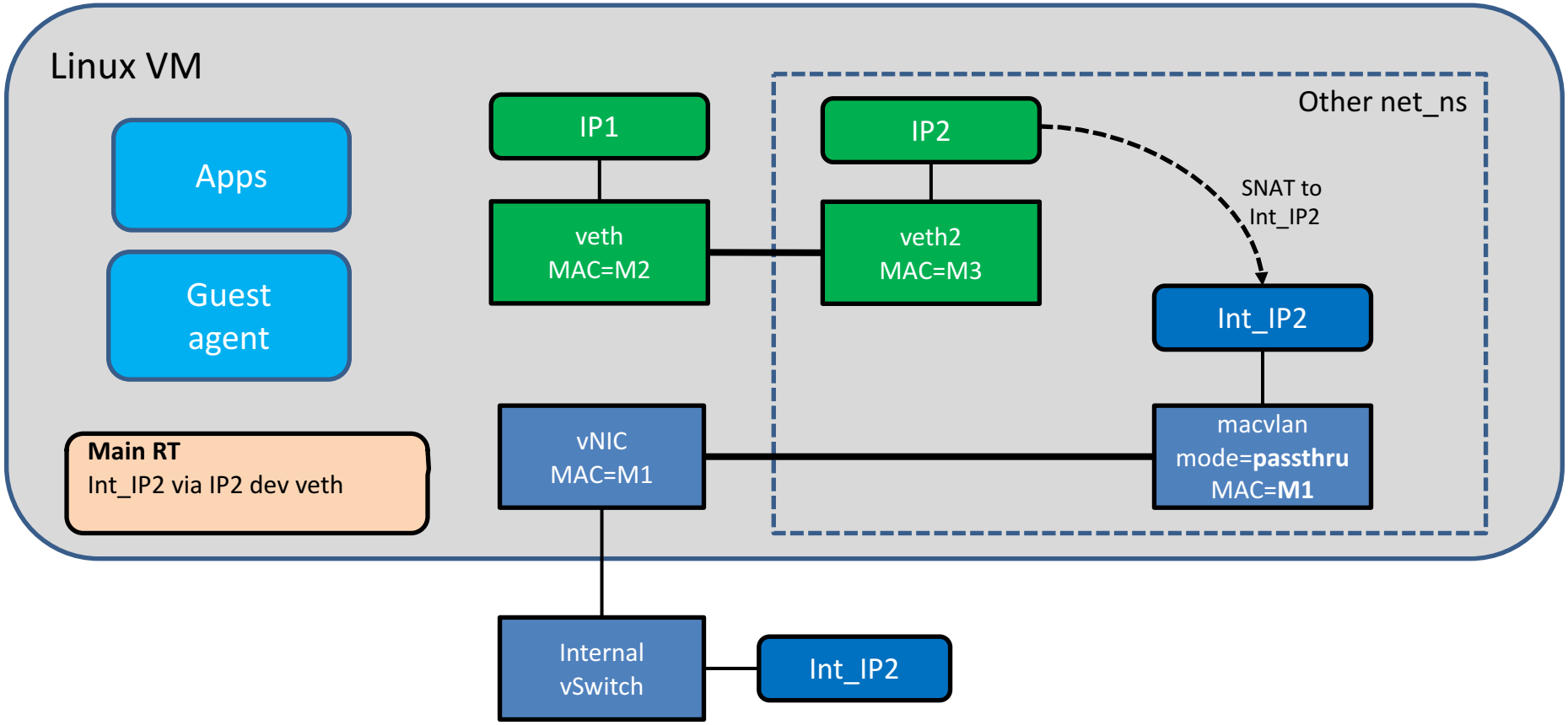
bpf_bind()

- We can attach a BPF program to cgroup and hook into connect/sendmsg path
- Context provides information about where an app is connecting to
- We can pick source address (but not port) and bind to it
  - Most apps are not aware and need no modification
  - Selection may be flexible
- Can also bind to the device
  - bound_dev_if is available from BPF_CGROUP_INET_SOCK_CREATE



Yet another net_ns use-case

Simple interface/IP hiding trick



Thanks!

Q&A