CONTAINERDAYS 2026 · HAMBURG

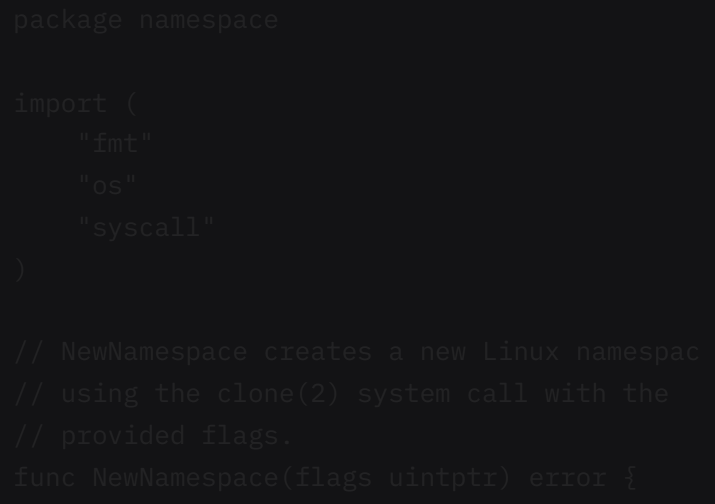
From veth Pairs to Services

The secret life of container packets, followed one packet at a time from a veth pair to a Kubernetes Service.

AS

Ajitem Sahasrabuddhe

Staff DevOps Engineer · Automattic



LIVE terminal

● LIVE frontend

```
$ fetch /api/time → curl -s http://backend
```

13:34:17

THU, 03 SEPT 2026 · CEST

ANSWERED BY	SERVICE	API CALLS	LAST CALL
backend-cf6856757-6t52j	http://backend	2	2ms

This page calls /api/time on the frontend pod once a second.
That handler calls http://backend, a different Service, and
returns what it said. Two real hops, both through kube-proxy.
The rest of the talk is what happens in between.

press ↑ to toggle

▲ terminal

eth0@if13
mtu 1450

We are going to follow
one packet
all the way

01

CHAPTER 01

Namespace

A flat in a housing society
Own doorbell. Own postbox. Own wiring.
The building has the street address.

$ sudo make step1

SRC	DST	AT	
-	-	no path yet	SEND PACKET

NOTHING HERE YET

$ sudo make step1

NETLINK EVENT FEED

fe80::a034:7bff:fe0b:c0b7/128

13:32:11.634 RTM_NEWROUTE
host · if2 · ff00::/8

13:32:11.634 RTM_NEWROUTE
host · if4 · ff00::/8

13:32:11.634 RTM_NEWROUTE
host · if7 · ff00::/8

13:32:12.141 IPTABLES_SNAPSHOT
host · 11 chains

A fresh netns has exactly one interface: loopback, and it is down. No routes, no neighbours, no connectivity. Isolation is the default, not something you switch on.

One interface.
It is down.

ajitem.s // netns.go go · internal/netns

```
  if err := unix.Unshare(unix.CLONE_NEWNET); err != nil {
      _ = os.Remove(path)
      return nil, fmt.Errorf("unshare netns: %w", err)
  }
```

```
  runtime.LockOSThread() // a netns belongs to the OS thread, not the process: pin before unshare
  defer runtime.UnlockOSThread()
```

A netns belongs to the **thread**, not the process.

The Go scheduler moves goroutines between threads whenever it likes.

02

CHAPTER 02

veth

The intercom
Two handsets. One wire.
Whatever goes in one end comes out the other. Always.

$ sudo make step2

SRC	DST	AT	
-	-	no path yet	SEND PACKET

NOTHING HERE YET

$ sudo make step2

NETLINK EVENT FEED

13:32:11.634 RTM_NEWROUTE
host · if2 · ff00::/8

13:32:11.634 RTM_NEWROUTE
host · if4 · ff00::/8

13:32:11.634 RTM_NEWROUTE
host · if7 · ff00::/8

13:32:12.141 IPTABLES_SNAPSHOT
host · 11 chains

The @ifN you see in the pod is the host-side index, and it is on the diagram because the kernel reported it, not because it was typed in. One cable, two ends, two namespaces. This is where runtime.LockOSThread matters: setns binds to a thread, and the Go scheduler will move you.

`eth0@if13`

13 is the host-side interface index.

ajitem.s // link.go go · internal/link

```
// NewVethPair creates a veth pair in the caller's current namespace. host is
// the end meant to stay there, peer is the end meant to move elsewhere.
func NewVethPair(host, peer string) (netlink.Link, netlink.Link, error) {
    veth := &netlink.Veth{
        LinkAttrs: netlink.LinkAttrs{Name: host},
        PeerName:  peer,
    }
    if err := netlink.LinkAdd(veth); err != nil {
        return nil, nil, fmt.Errorf("create veth pair %s/%s: %w", host, peer, err)
    }

    hostLink, err := netlink.LinkByName(host)
    if err != nil {
        return nil, nil, fmt.Errorf("look up %s after creating it: %w", host, err)
    }
    peerLink, err := netlink.LinkByName(peer)
    if err != nil {
        return nil, nil, fmt.Errorf("look up %s after creating it: %w", peer, err)
    }
    return hostLink, peerLink, nil
}

// MoveToNamespace moves l into the namespace referenced by nsFD, the way
// `ip link set <l> netns <fd>` does.
func MoveToNamespace(l netlink.Link, nsFD int) error {
    if err := netlink.LinkSetNsFd(l, nsFD); err != nil {
        return fmt.Errorf("move %s to namespace: %w", l.Attrs().Name, err)
    }
```

$ sudo make step3

SRC
192.168.200.2

DST
192.168.200.3

AT
no path yet

SEND PACKET

NOTHING HERE YET

$ sudo make step3

NETLINK EVENT FEED

fe80::a034:7bff:fe0b:c0b7/128

13:32:11.634 RTM_NEWROUTE
host · if2 · ff00::/8

13:32:11.634 RTM_NEWROUTE
host · if4 · ff00::/8

13:32:11.634 RTM_NEWROUTE
host · if7 · ff00::/8

13:32:12.141 IPTABLES_SNAPSHOT
host · 11 chains

Pod-to-pod on the same node never leaves layer 2. The bridge learns MAC addresses into its FDB exactly like the switch under your desk. No routing, no NAT, no iptables.

The bridge learns

bash, cnwbr0 — bridge fdb show dynamic

```
$ bridge fdb show br cnwbr0 dynamic
(nothing)

$ sudo ip netns exec cnw-03a ping -c1 192.168.200.3

$ bridge fdb show br cnwbr0 dynamic
2a:c1:9e:44:01:05 dev cnwbh0 master cnwbr0
6e:83:2f:11:02:06 dev cnwbh1 master cnwbr0
```

> Same MAC learning as the switch under your desk.

03

CHAPTER 03

Leaving the node

masquerade

BEFORE THE RULE

```
$ sudo ip netns exec cnw-04 ping -c1 1.1.1.1
(no reply)
```

AFTER THE RULE

```
$ sudo iptables -t nat -A POSTROUTING \
  -s 192.168.104.0/24 -o eth0 -j MASQUERADE
$ sudo ip netns exec cnw-04 ping -c1 1.1.1.1
64 bytes from 1.1.1.1
```

`conntrack` is the receptionist's notebook
Without it the reply arrives
and nobody knows whose call it was.

04

CHAPTER 04

Across nodes

CNI is a contract, not a technology

CNI contract

```
ADD stdin: netns path, container id, config JSON
    stdout: assigned IPs, routes, DNS

DEL tear it all down
CHECK is it still as I left it
```

> The kubelet runs a binary. That is the whole interface.

Three answers to one question

Criterion	Overlay	Routed	VPC-native	Verdict
How the packet crosses	Wrapped in UDP, unwrapped on arrival	Ordinary routing, no wrapper	No crossing. It was never separate	
Pod IP is	Fake to the network	Real to the fabric	A real VPC address	the point

05

CHAPTER 05

The IP that is not there

```
ajitem@cnw-dev:.../c
ds-2026-hamburg-cnw-code$
```

This address exists
on no interface
in the entire cluster.

It is a rule

iptables -t nat docker exec cnw-control-plane

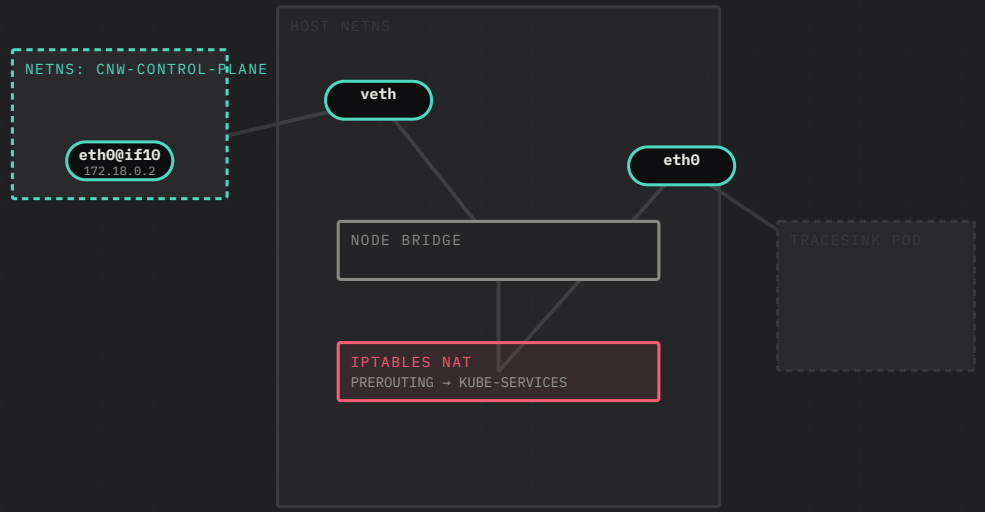
```
ajitem@cnw-dev:.../c
ds-2026-hamburg-cnw-code$
```

press f for cast

> Load balancing is a coin flip in a chain.

```
$ sudo go run ./cmd/tracepkt -ns cnw-control-plane -dst <ip>:9999
```

SRC	DST	AT	
node	**ClusterIP**	ready	SEND PACKET



NETLINK EVENT FEED

13:34:08.535 RTM_NEWNEIGH
cnw-control-plane · if4 · 10.244.0.9 STALE

13:34:14.171 RTM_NEWNEIGH
cnw-control-plane · if4 · 10.244.0.9 PROBE

13:34:14.171 RTM_NEWNEIGH
cnw-control-plane · if4 · 10.244.0.9 REACHABLE

The ClusterIP is not configured on any interface in the cluster. Grep every namespace on every node and you will not find it. It exists only as a DNAT rule, and conntrack is the notebook that lets the reply find its way home.

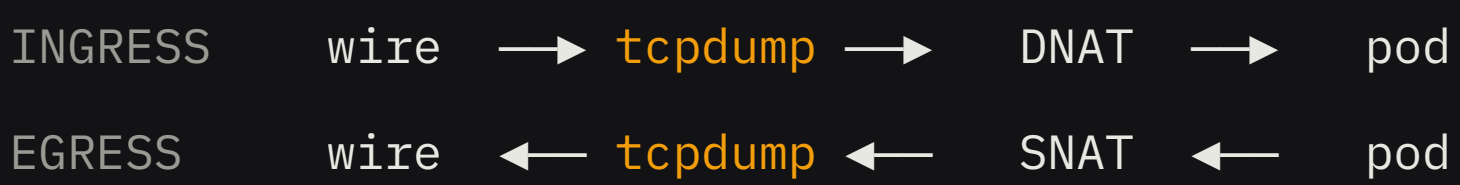
Three captures, one packet

testdata/trace.pcapng

```
$ tshark -r testdata/trace.pcapng -X lua_script:tools/nviz.lua \
  -T fields -e frame.number -e nviz.trace -e ip.src -e ip.dst

1  0xa1c55502710c1f34  192.168.104.2    1.1.1.1   ← inside the pod
2  0xa1c55502710c1f34  192.168.104.2    1.1.1.1   ← on the host veth
3  0xa1c55502710c1f34  192.168.139.107  1.1.1.1   ← on the uplink
```

> Same trace id. Different source.
> The masquerade happened between the veth and the uplink.



> The tap is always on the wire side of the hook.
> Ingress you see it **before** DNAT, egress **after** SNAT.
> Two captures, either side, or you see nothing.

06

CHAPTER 06

Who decided that MTU

```
mtu 1450
```

VXLAN takes 50 bytes.
Small requests work. Large ones hang.

07

CHAPTER 07

The whole path

```
curl
  ↓  pod netns
eth0 (veth)
  ↓
if13 on the host
  ↓
cni0
  ↓  PREROUTING: DNAT   ClusterIP → pod IP
routing decision
  ↓  VXLAN wrap, or a BGP route, or nothing at all
node-2
  ↓
cni0 → veth → backend pod
```

conntrack reverses every step on the way home

`eth0@if13` `mtu 1450`

You know exactly what both of these mean now.

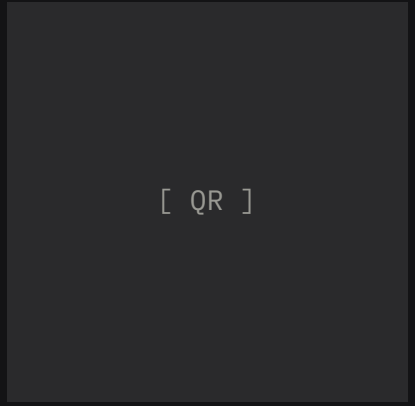
// THAT'S A WRAP

Thank You

The slides, code, and gcr repo are at:

`github.com/asahasrabuddhe/cds-2026-hamburg-cnw-code`

@asahasrabuddhe



scan for slides + repo