THE LINUX FOUNDATION

OPEN SOURCE SUMMIT INDIA

How Kubernetes Networking Really Works: A Packet's Journey Across Pods and Nodes

S Ashwin & M Viswanath Sai

About Us

- S Ashwin
- Software Engineer - Deutsche Bank
- Open Source Contributor
- Maintainer @Prometheus-Operator
- Linkedin

- M Viswanath Sai (aka. Vishwa)
- Fresh out of college
- Plans to Startup
- Maintainer @ Prometheus Operator
- Linkedin

What Really happens to a Packet?

Pod A
10.244.1.5

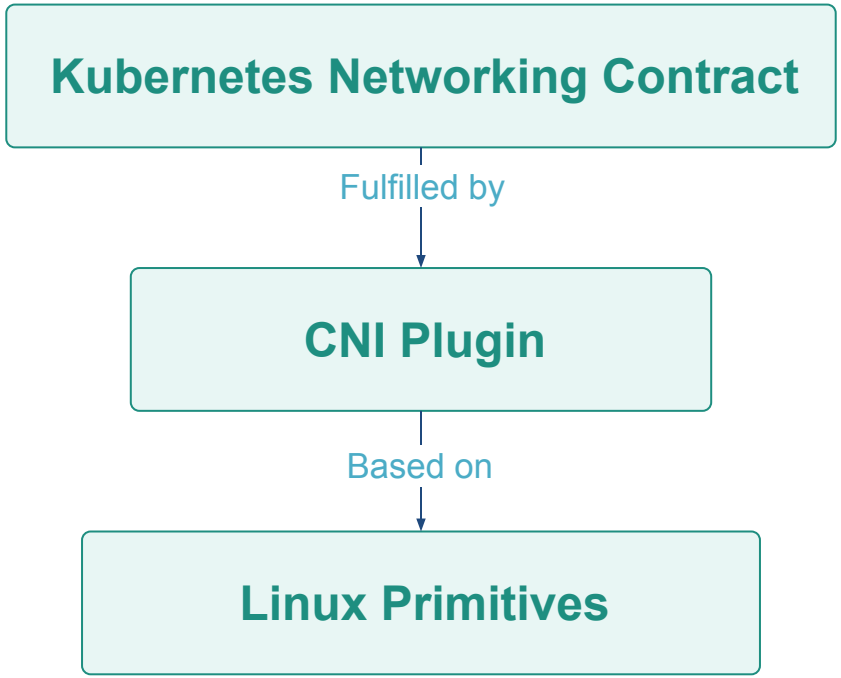
Pod B
10.244.2.8

Kubernetes Networking Contract

Kubernetes expects:

- Unique Pod IPs
- No NAT Between Pods
- Cross-Node Reachability
- Node ↔ Pod Access
- Flat network

Kubernetes Networking is Linux Networking



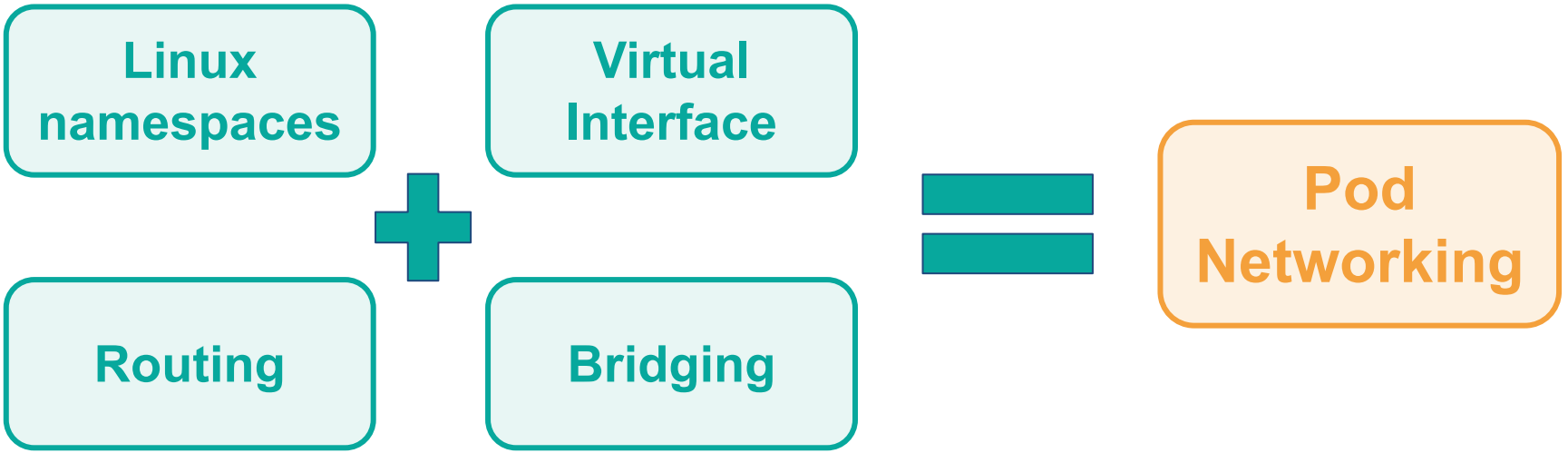
THE LINUX FOUNDATION

OPEN SOURCE SUMMIT INDIA

Building Pod Networking With Linux Primitives

Creating A Pod Network With Linux

Building Blocks



Translating Kubernetes to Linux

Kubernetes	Linux Primitives
Pod's isolated network stack *(each Pod = its own netns)*	**Network Namespace (netns)** *ip netns add*
Container's virtual NIC (eth0) *connecting Pod netns to the node*	**veth pair** *ip link add type veth*
Node's pod network bridge *(e.g. br0 created by the CNI plugin)*	**Linux Bridge (cbr0)** *brctl / ip link add type bridge*
Pod IP allocation by the CNI's IPAM *(e.g. 10.244.x.x range)*	**IP addressing + routes** *ip addr, ip route*
Service *Stable access to dynamic pods*	**Iptables** *Packet processing rules*

Let's Ponder for a Minute

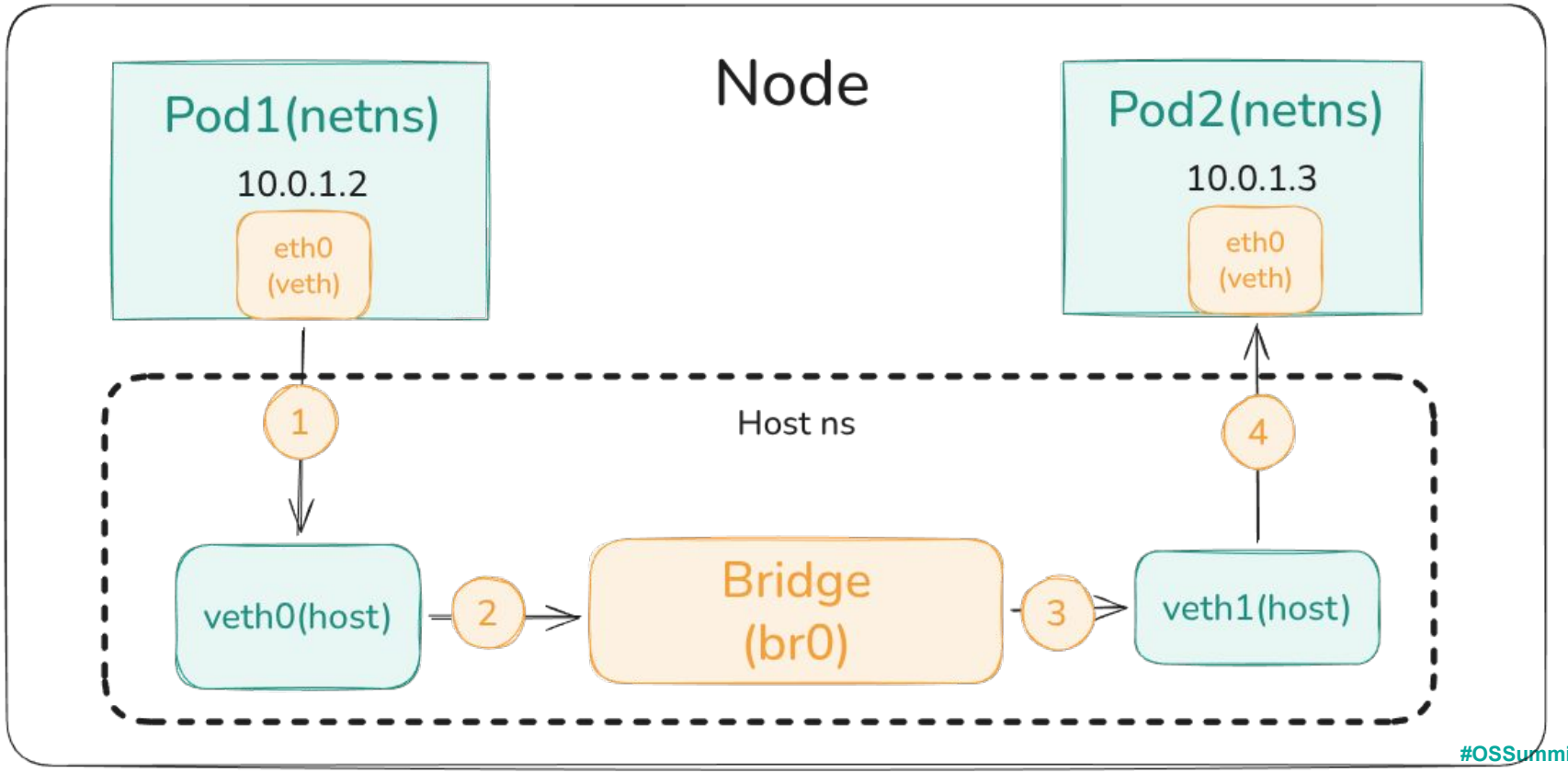
Node owns a CIDR block
10.0.1.0/24

Pods on this node get IPs in that CIDR block
Pod1 -> 10.0.1.2, Pod2 -> 10.0.1.3

You can play around with the subnets
10.0.1.0/24

How to establish Pod1<->Pod2 communication following the Kubernetes contract?

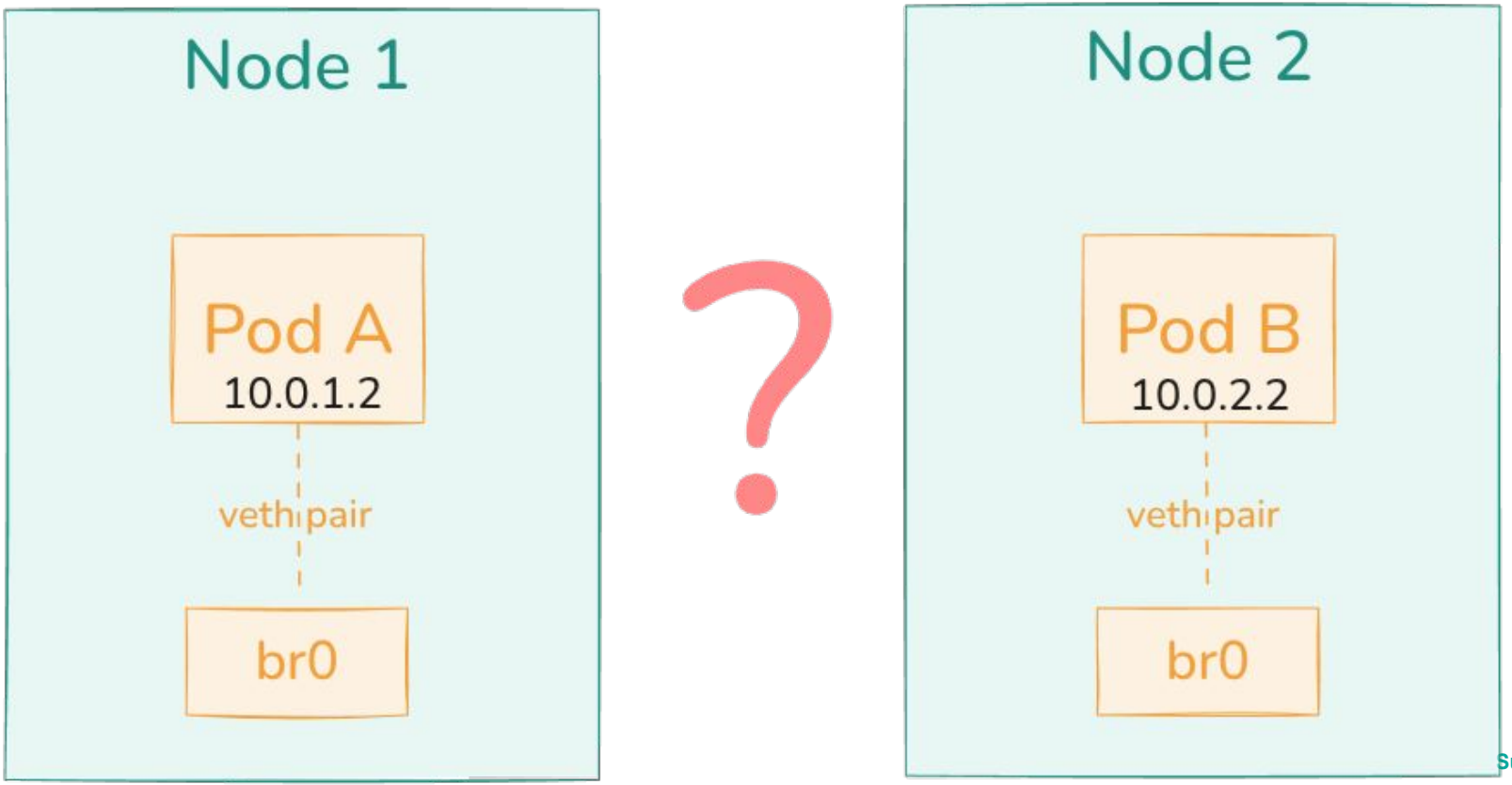
Packet Journey: Same-Node Pod-to-Pod



THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
INDIA

Act 1: Connecting Pods Across Different Nodes

Crossing Node Boundaries



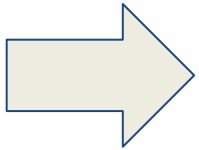
We have all the Ingredients

Network doesn't know pods

Pods <-> Host Node

Node A <-> Node B

NAT not allowed

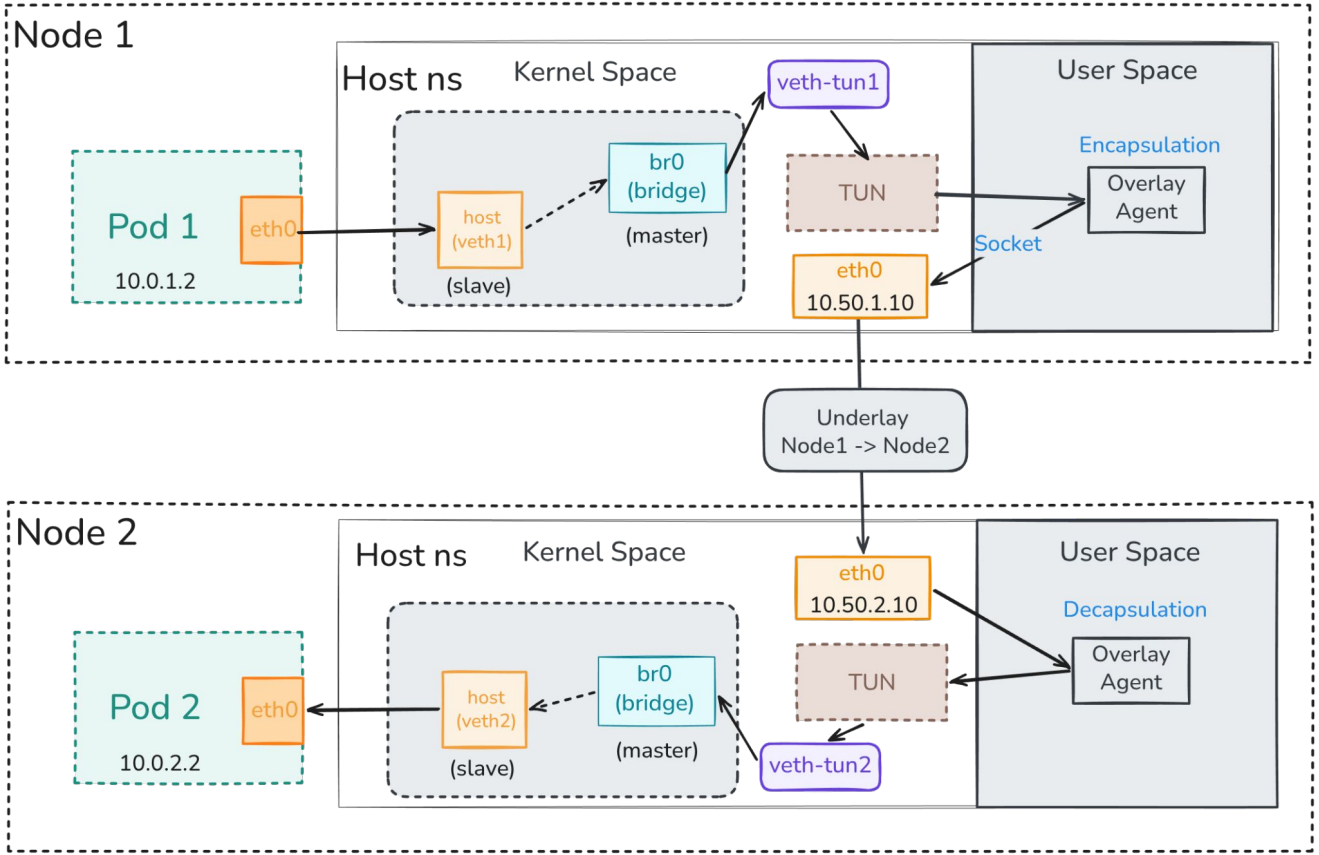


Encapsulation

Node A to Node B

Pod A to Pod B

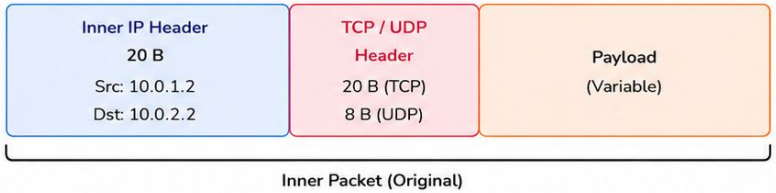
Building an Overlay Network with TUN



TUN Encapsulation

1) Original Packet (Inner Packet)

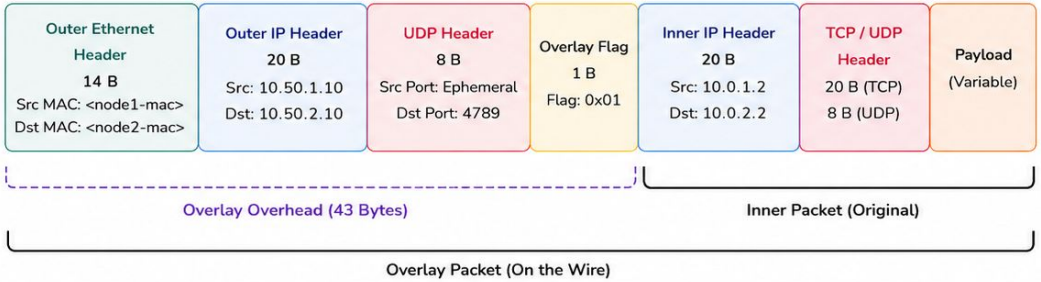
Packet at tun0 (raw IP packet read by the overlay)



TUN Encapsulation
(Added by overlay network)

2) Overlay Packet (On the Wire)

Packet after encapsulation over the underlay



TUN Overhead Breakdown

Outer Ethernet Header	14 B
Outer IPv4 Header	20 B
UDP Header	8 B
Overlay Flag	1 B
Total Overhead	**43 Bytes**

Header Details

- Outer IP Protocol : UDP (17)
- UDP Destination Port : 4789
- Overlay Flag (1 B) : 0x01
  - Meaning: Overlay (TUN) packet
- Inner IP Packet : Raw IP packet from tun0 (no Ethernet header)
- TUN (Not TAP/VXLAN): No inner Ethernet header is carried.

Example Values

Outer (Underlay)
Src IP: 10.50.1.10 (node1 underlay IP)
Dst IP: 10.50.2.10 (node2 underlay IP)
UDP Dst Port: 4789
Overlay Flag: 0x01

Inner (Overlay)
Src IP: 10.0.1.2
Dst IP: 10.0.2.2

Notes
- TUN device carries raw IP packet (no Ethernet header)
- Overlay adds: Outer L2 + Outer IP + UDP + Overlay Flag

What Changed?

Inter-node pod communication

Underlay stays untouched

Works across any IP network

Asymmetric Network Problem

Same Node

Pod1
↓
Bridge
↓
Pod2

L2 Switching

Different Node

Pod1
↓
TUN → Encapsulation → Node2 -> Decapsulation → TUN → Pod2

L3 Routing

"Am I debugging inter or intra node traffic"

Consequences of L2 Switching

Pods can’t trust each other → ARP Spoofing

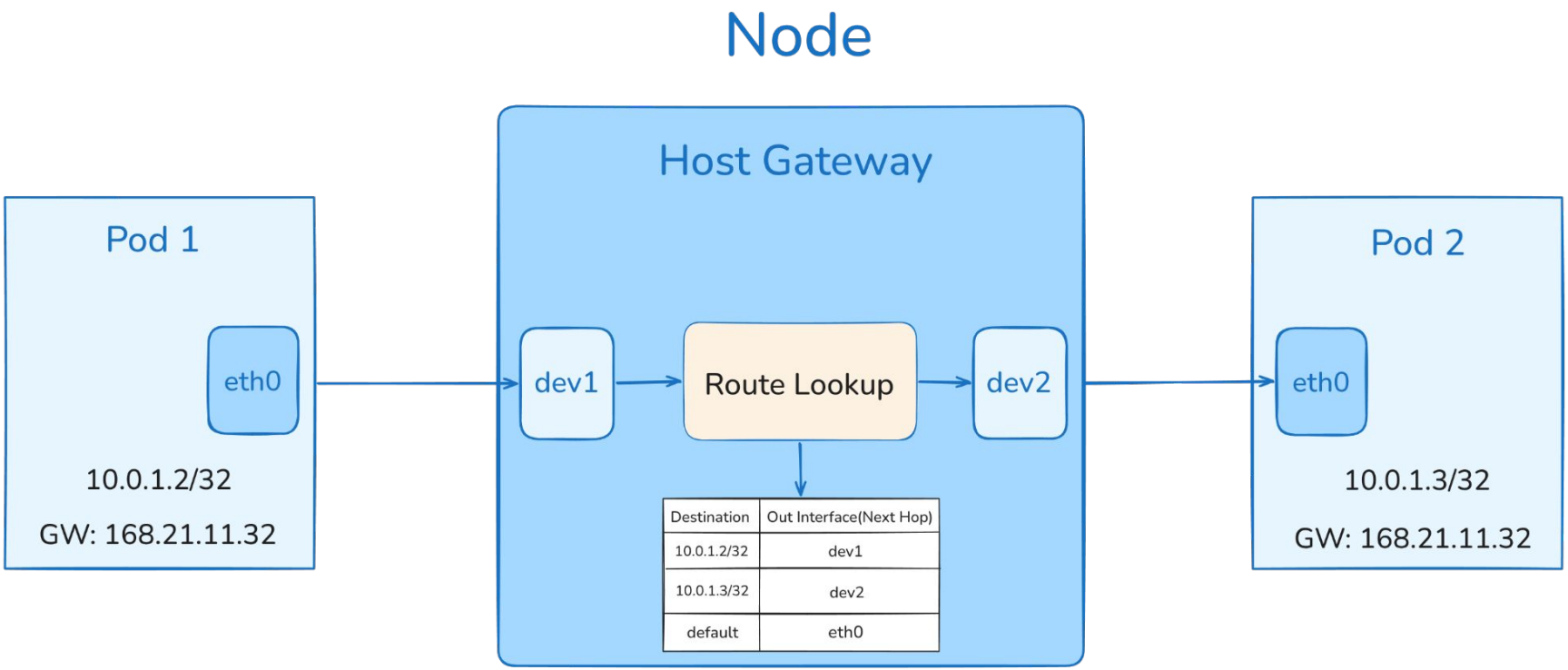
Pods are ephemeral →ARP Flooding

What if we remove L2 altogether?

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
INDIA

Act 2: Building a Pure Layer 3 Network

Packet Journey inside Node - L3 Routing



Conceptual Shift

Act 1	Act 2
Pods consider each other neighbours	Pods talk only to their host
L2 switching	L3 Routing
Bridge handles forwarding	Host Node handles routing
ARPs for peers	ARP only for gateway (host)

The ARP trick

The Problem	The "Trick"
Need Gateway MAC - ARP	Lie to the Pod
No Gain in Assigning Veth IPs	Host Proxy ARPs

ARP broadcasts avoided, IPs saved, and Packet routed!!

What Did We Solve? What Still Remains?

Solved	Still a Problem….
L2/L3 Asymmetry (Host-driven Routing)	Userspace Encapsulation
Side Stepped ARP Broadcasts	Kernel ↔ Userspace transitions
Simpler networking model	CPU Load, Engineering effort, bandwidth….

The Hidden Overhead

Kernel
↓
Userspace
↓
Kernel

Every packet crosses the kernel/userspace boundary

Every packet is copied in the userspace and written back again into the UDP socket

Operational Complexity

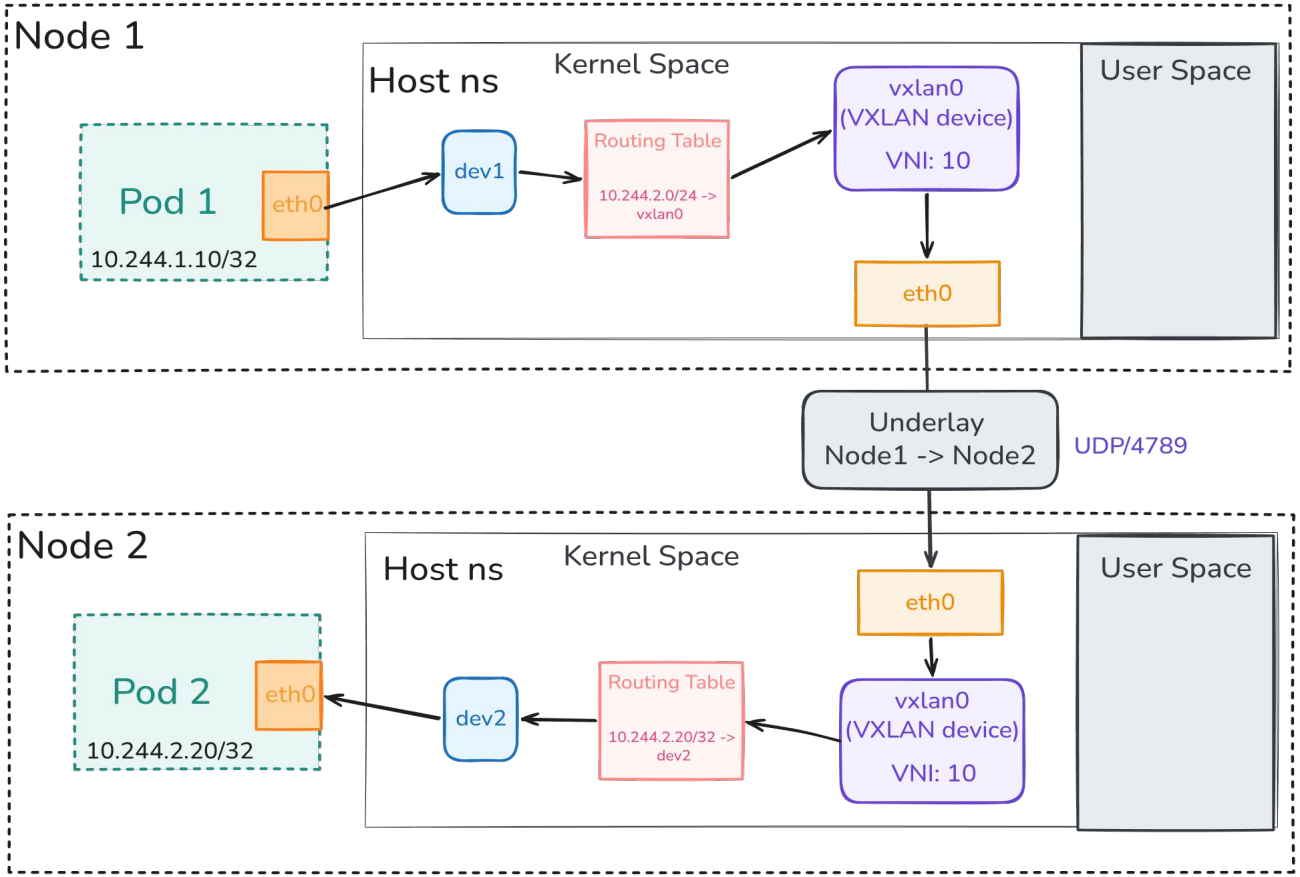
More CPU load, latency, potentially unsafe

Can we handle encapsulation in Kernel Space?

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
INDIA

Act 3: Enter VXLAN: Moving Encapsulation to Kernel Space

Packet Journey with VXLAN Implementation



VXLAN Encapsulation

1) Original Pod Packet (Inner Packet)

Packet inside the overlay (pod to pod)

Inner Ethernet Header 14 B	Inner IP Header 20 B Src: 10.244.1.10 Dst: 10.244.1.20	TCP / UDP Header 20 B (TCP) 8 B (UDP)	Payload (Variable)

Inner Packet (Original)

VXLAN Encapsulation
(Added by overlay network)

2) VXLAN Packet (On the Wire)

Packet after encapsulation over the underlay

Outer Ethernet Header 14 B	Outer IP Header 20 B Src: 192.168.1.1 Dst: 192.168.1.2	UDP Header 8 B Src Port: Ephemeral Dst Port: 4789	VXLAN Header 8 B VNI: 100 Flags: 0x08	Inner Ethernet Header 14 B	Inner IP Header 20 B Src: 10.244.1.10 Dst: 10.244.1.20	TCP / UDP Header 20 B (TCP) 8 B (UDP)	Payload (Variable)

VXLAN Overhead (50 Bytes)

Inner Packet (Original)

VXLAN Packet (On the Wire)

VXLAN Overhead Breakdown

Outer Ethernet Header	14 B
Outer IP Header	20 B
UDP Header	8 B
VXLAN Header	8 B
Total Overhead	**50 Bytes**

Header Details

- Outer IP Protocol : UDP (17)
- UDP Destination Port : 4789
- VXLAN Header (8 B):
  - Flags (1 B) : 0x08 (I flag set)
  - Reserved (3 B) : 0x000000
  - VNI (3 B) : 24-bit (0 - 16M) = 100
  - Reserved (1 B) : 0x00

L2 over L3

Standard encap protocol

Does exactly what TUN overlay did

Here's what Happens

The Issue

Pretends Nodes are L2 connected

Next Hop MAC needed → ARP across the network??

The Solution Flow

Route inter node traffic through VXLAN

Compute VTEP MAC deterministically → Side step ARP

Map VTEP MAC → Node Underlay IP

What Changed?

Act 2	Act 3
Userspace Agent	Kernel VXLAN Device (vxlan0)
Manual UDP Encapsulation	Standard VXLAN Encapsulation
Kernel -> User -> Kernel	Kernel only datapath

VXLAN doesn't change the networking model. It changes where encapsulation happens.

Is Encapsulation the Correct Answer?

Act 1
How do we connect pods with the least friction?

Bridged pods + Manual Overlay

Act 2
How can we improve the architecture of pod connection?

Routed pods + Manual Overlay

Act 3
How can we Reduce overhead?

Routed pods + **Standard Kernel Overlay**

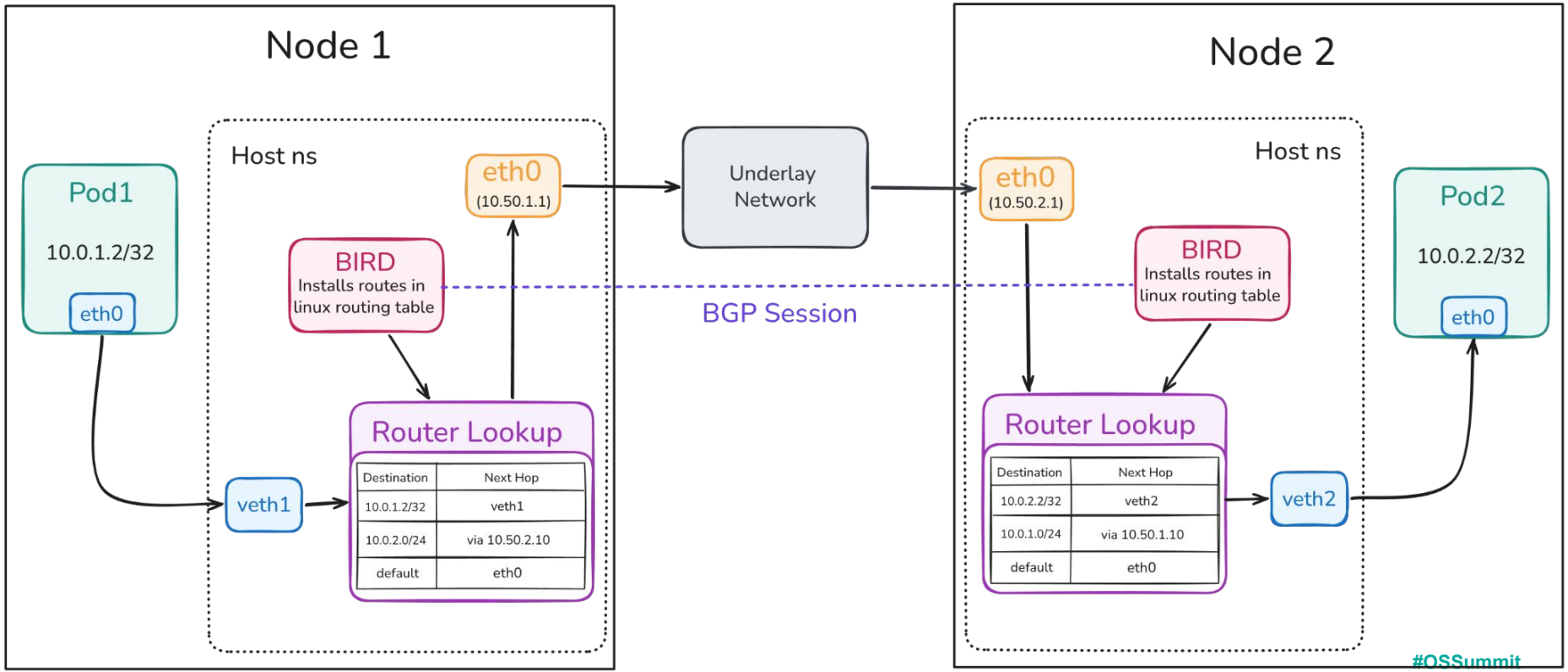
Do we need encapsulation at all?

What if the network could learn pod routes directly?

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
INDIA

Act 4: Beyond Overlays: Teaching the Network about Pod Routes

Packet Journey with Native Routing(BIRD)



What Changed?

Act 3 (VXLAN)	Act 4 (BGP)
Hide Pod routes	Advertises Pod routes
Encapsulation	Native Routing
Dataplane state propagated with custom daemon	Dataplane state propagated with standard protocol
Underlay only sees nodes	Underlay learns Pod CIDRs

Tradeoffs with BGP

Operational Coupling

More Control Plane State

Pod routes consume capacity

There Is No “Best” Network

Model	What we gain
TUN Overlay	Finegrain Control on Overlays
Pure L3 Routing	Symmetric networking model
VXLAN	Kernel-native overlays
BGP Routing	Pod Aware Routing

Summary

- Preserve Pod Identity

- Shrink ARP's Job

- Symmetry makes for a cleaner networking model

- Userspace UDP -> Kernel VXLAN, same model different promises

- Forwarding State Is Paramount: Every model relies on this at some level, via different means

- Underlay Contract Changes

Every solution removed a bottleneck and introduced a new tradeoff.

Welcome to networking

THE LINUX FOUNDATION

OPEN SOURCE SUMMIT INDIA

Thank You