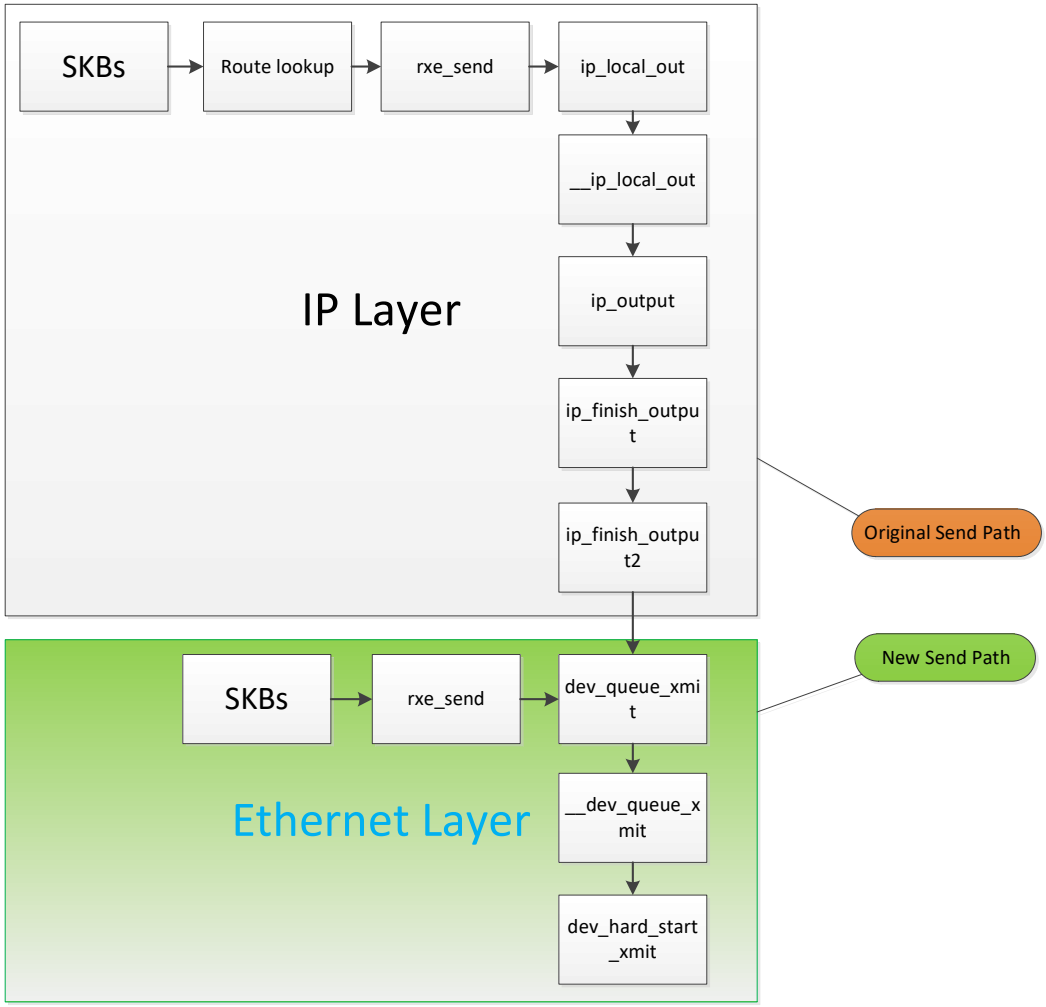
Dmabuf/devmem with RDMA(RXE) via Netkit

Agenda

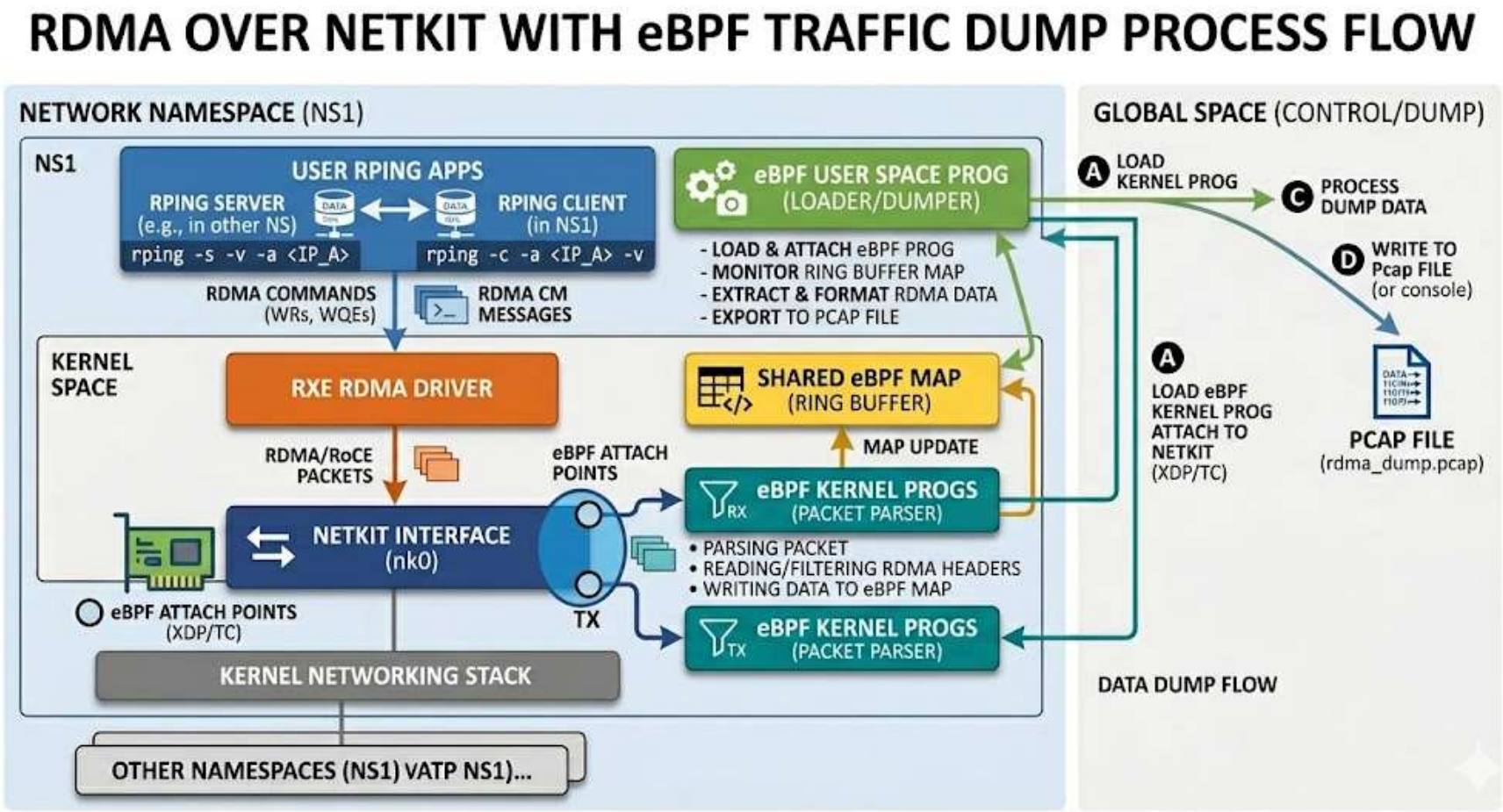
- RXE can work with devmem/dmabuf via netkit
  - RXE can call netkit xmit/recv, bypass ip layer including route_finding, iptalbes, reduce latency
  - Via netkit, ebpf can work well with RXE
  - Via netkit, devmem/dmabuf can work well with RXE
  - Demo for RXE with netkit
- RXE can work with dmabuf/devmem without netkit
  - RXE can queue leasing
  - RXE can work with eBPF directly

RXE works with Devmem/dmabuf

- RXE can send skb to netkit xmit/recv via ethernet layer, bypass ip layer including route_finding, iptalbes, reduce latency
- This will reduce about 10% - 20% latency depending on the iptables and route rules, since this will bypass these iptables and route rules.
- If needed, RXE can work with NIC driver directly, bypass ip and ethernet layers. To now, since bypassing ethernet layer will not bring much benefit, thus not bypassing ethernet.
- The changesets are ready in github.



Via netkit, ebpf can work well with RXE - Architecture



Traffic Generation: Utilizes Soft-RoCE (RXE) and netkit to simulate RDMA traffic across network namespaces.

Kernel-Level Hooking: eBPF programs are attached to netkit (nk0) interfaces using XDP/TC for high-performance packet parsing.

Efficient Data Transfer: Extracted RDMA headers are written to a Shared eBPF Map (Ring Buffer) to minimize overhead.

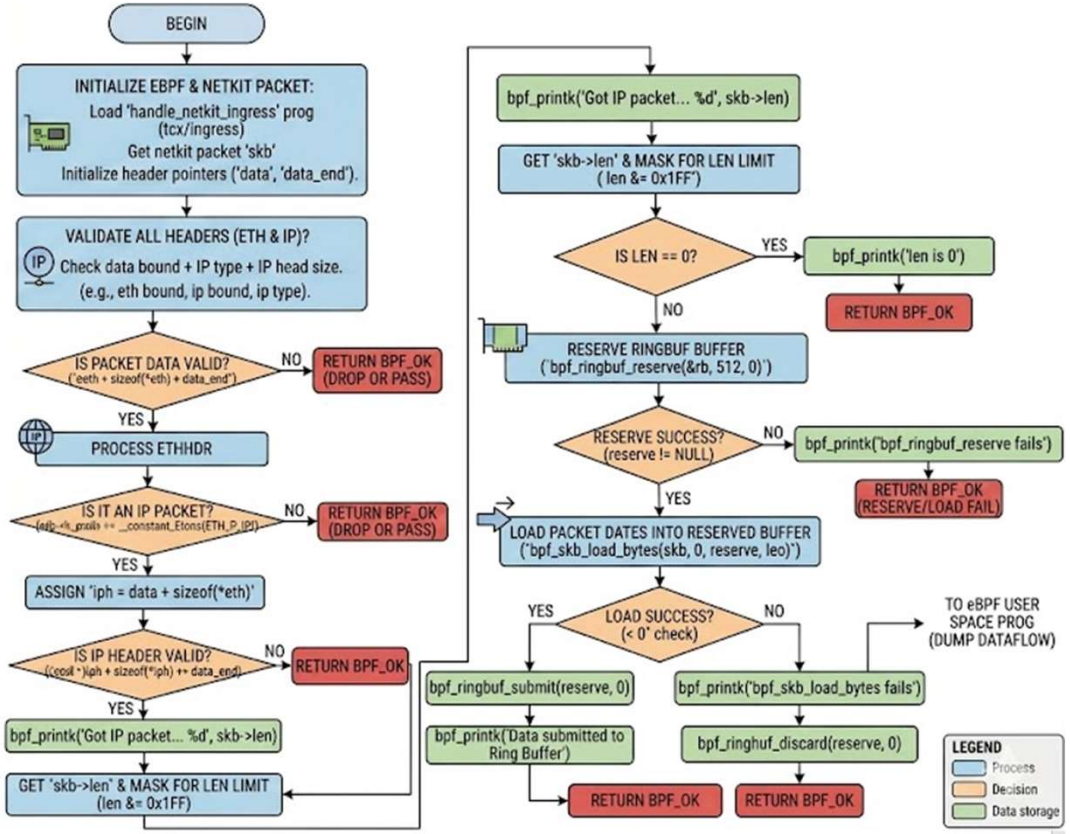
Analysis-Ready Output: A user-space dumper processes the buffer data and exports it into a standard PCAP file for external analysis.

Via netkit, ebpf can work well with RXE – eBPF kernel prog

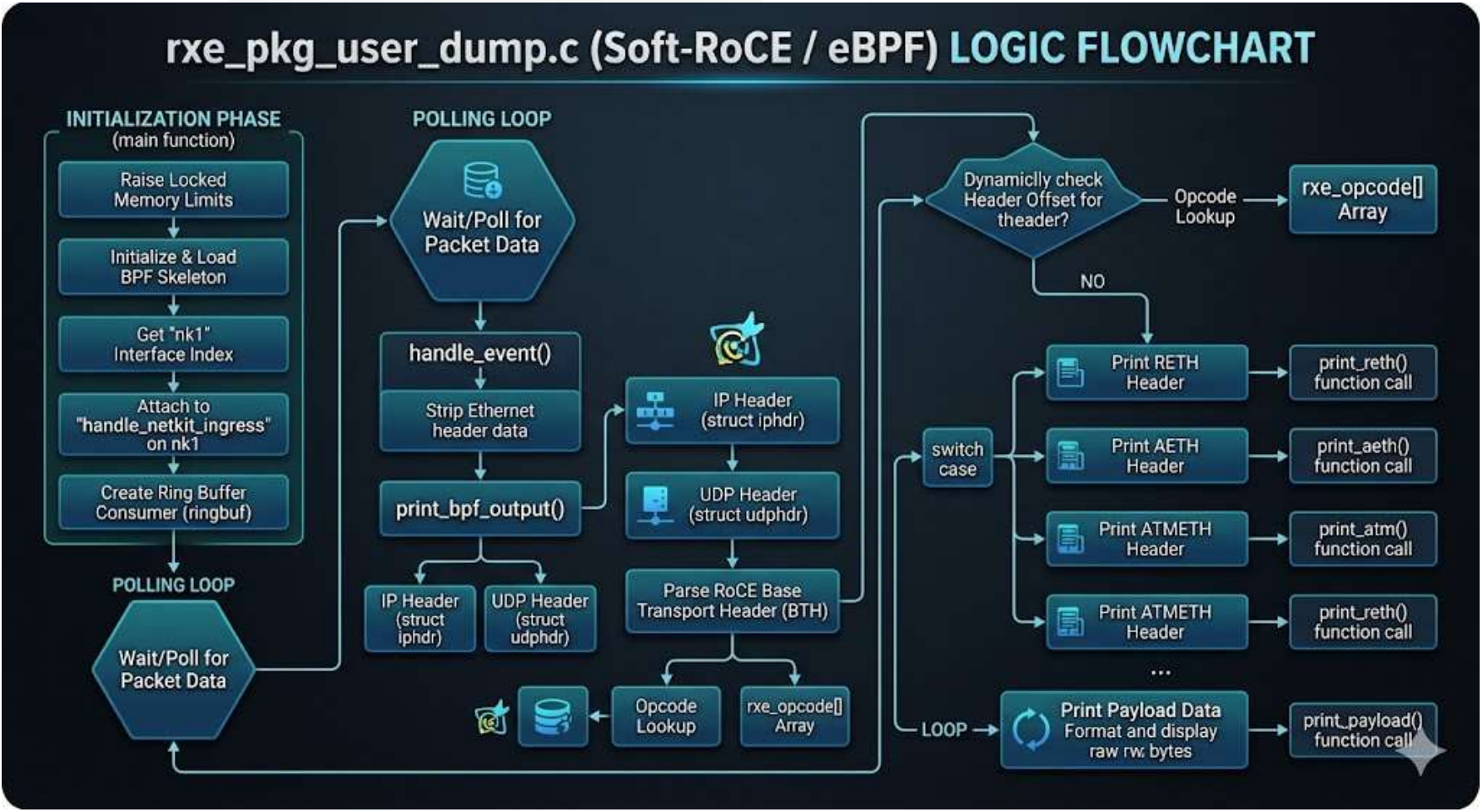
This left flowchart, illustrates the logical architecture of an eBPF-based netkit ingress traffic handler. It translates complex C code into an intuitive progression, starting with the initialization of the skb and the extraction of header pointers, defining the exact entry point for network data processing within the kernel.

The core of the process features a rigorous **protocol validation and filtering layer**. The flowchart details how the program sequentially inspects Ethernet frames and IP headers to ensure packets are complete and match the target protocol, such as ETH_P_IP. Any data failing these boundary or type checks is passed through using BPF_OK, maintaining high-performance kernel operations.

The final stage covers **data collection and reporting**. After calculating the packet length, the program attempts to reserve space in the **eBPF Ring Buffer** and utilizes bpf_skb_load_bytes to copy the raw packet into this buffer. Once submitted, the data flows to a user-space program for dump analysis, completing the full cycle from kernel capture to user-space export.



Via netkit ebpf can work well with RXE – ebpf userspace



1.Initialization: Raises locked memory limits and initializes/loads the BPF skeleton.
Interface Attaching: Retrieves the target nk1 interface index and hooks into handle_netkit_ingress.
Buffer Creation**:** Establishes the user-space ring buffer consumer (ringbuf) to prepare for kernel data.
2. Polling & Event Handling
Efficient Polling: Waits and polls block-by-block for raw network packet data passed up from the kernel.
Header Stripping: Triggers handle_event() to strip the Ethernet header, passing the remaining data to print_bpf_output.
Layer 3/4 Extraction: Extracts basic network metadata (IP and UDP headers) via struct iphdr and struct udphdr.
3. RoCE & Payload BTH Parsing: Parses the RoCE Base Transport Header (BTH) and performs an opcode lookup against the rxe_opcode[] array.
Dynamic Extension Matching: Evaluates header offsets dynamically, routing through a switch case block to handle specialized headers.
Payload Output: dumps the final raw payload bytes via print_payload().

Via netkit, ebpf can work well with RXE – RDMA pkg

1. Analyze a rdma package. Get the ip header, udp header, bth header, reth header and rdma payload

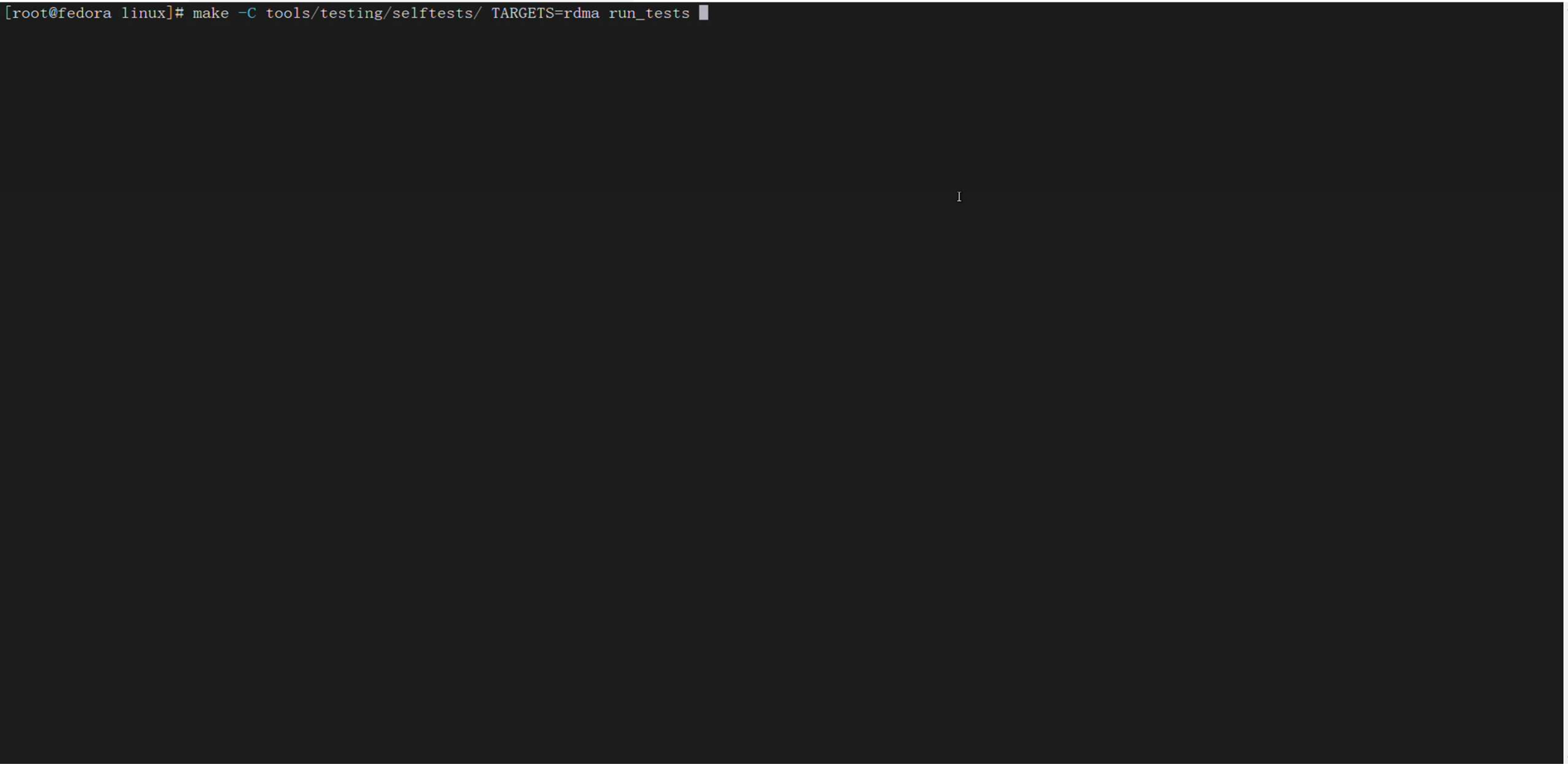
2. rxe_pkg_kernel_dump.c

rxe_pkg_netkit.sh

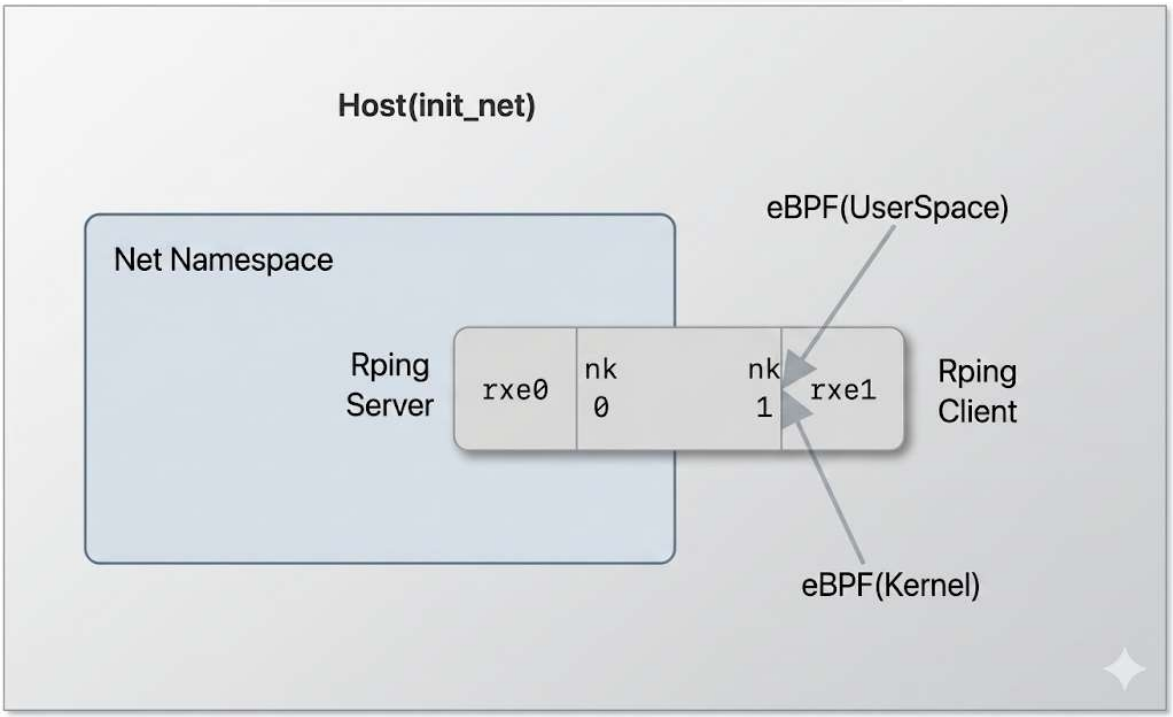
rxe_pkg_user_dump.c

```
# struct iphdr {
#       __be32  saddr; 10.0.0.1
#       __be32  daddr; 10.0.0.2
# };
# udp hdr:
# struct udphdr {
#       __be16  source; 57350
#       __be16  dest;   4791
#       __be16  len;    104
#       __sum16 check;  0
# };
# IB_OPCODE_RC_RDMA_WRITE_ONLY
# struct rxe_bth {
#       __u8            opcode; 0xa
#       __u8            flags;  0x0
#       __be16          pkey;   0xffff
#       __be32          qpn;    0x11
#       __be32          apsn;   0x9c37fd
# };
# RXE_RETH
# struct rxe_reth {
#       __be64          va;
#       __be32          rkey;
#       __be32          len;
# };
# RXE_PAYLOAD
# rdma-ping-1: BCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrs00ffffffe6ffffff83ffffffc4ffffff9a0000000000000000
```

Via netkit, devmem/dmabuf can work well with RXE - Demo



Architecture of RXE with netkit and summary



Network Isolation Boundary

Host Environment (init_net):
The primary global host space where the Rping Client and the rxe1 Soft-RoCE instance reside.

Isolated Net Namespace: A separate containerized network environment enclosing the Rping Server and the rxe0 Soft-RoCE instance.

Inter-Namespace Connectivity (netkit)

A netkit virtual ethernet pair (nk0 and nk1) acts as the network bridge, crossing the boundary between the isolated namespace and the host space.

Soft-RoCE (rxe) Mapping: The RoCE emulation layer maps rxe0 to nk0 inside the namespace, and rxe1 to nk1 on the host, allowing native RDMA traffic (rping) to run over standard ethernet links.

Dual-Layer eBPF Observability Hook

eBPF (Kernel): Injected directly into the kernel network pathway at the host interface (nk1) to intercept and parse inbound/outbound packets with zero-copy efficiency.

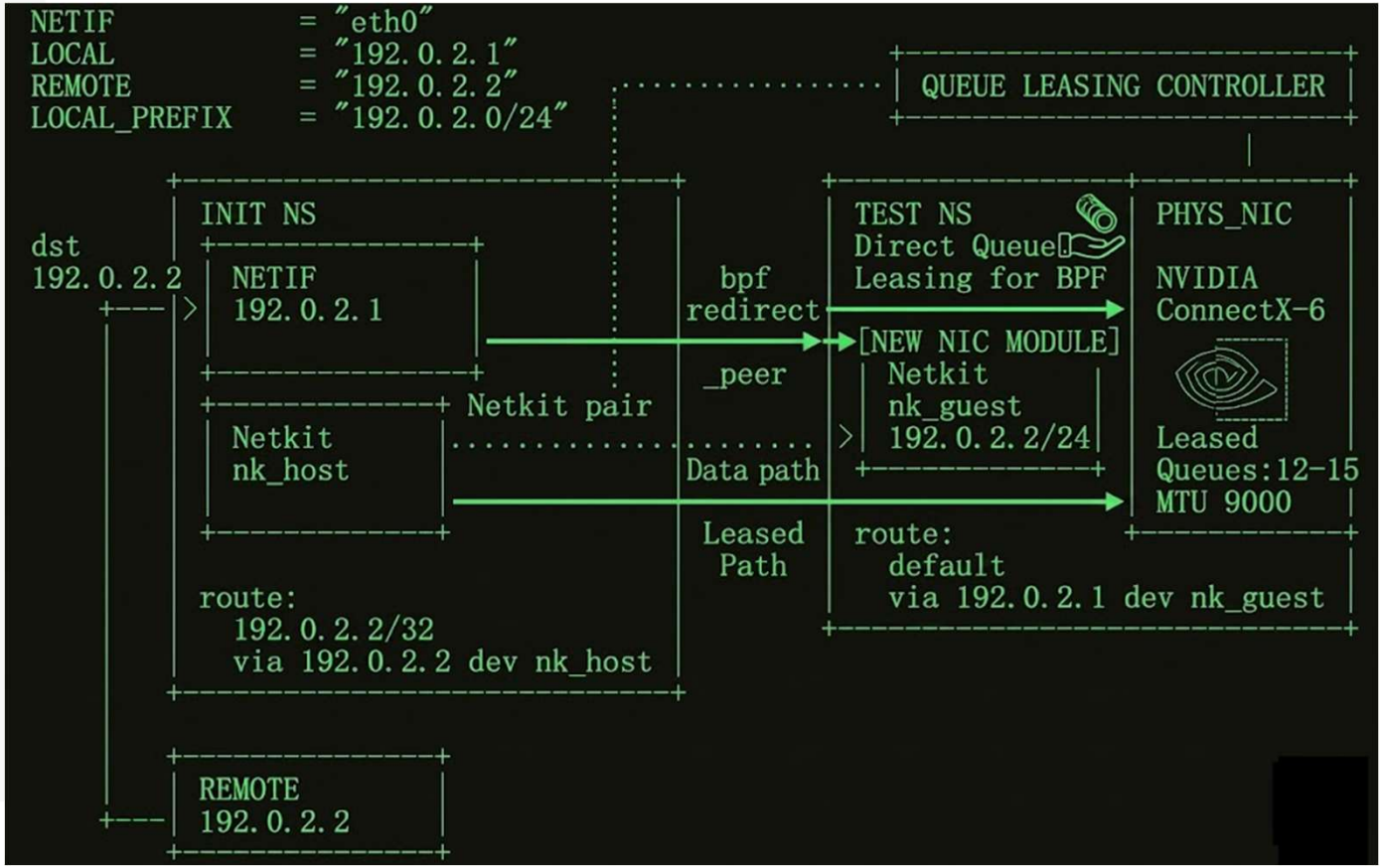
eBPF (UserSpace): Represents the control and collection plane executing in the host user space, polling the extracted metadata from the kernel for tracing, dumping, or analysis.

RXE works with Devmem via netkit

1. The architecture consists of four main building blocks:**INIT NS** (Host/Root Namespace): Contains the root network interface **NETIF** (192.0.2.1) and the host side of the Netkit pair (nk_host).TEST NS (Target Container/Test Namespace): Represents the isolated container environment running a [NEW NIC MODULE] with the guest side of the Netkit pair (nk_guest, 192.0.2.2/24).**QUEUE LEASING** CONTROLLER: A control-plane component that dynamically manages, isolates, and allocates specific hardware queues from the physical NIC to target namespaces.**PHYS_NIC** (Physical Network Interface): Represented here by an NVIDIA ConnectX-6 NIC, which supports hardware partitioning.

2. Software Acceleration via eBPF (bpf redirect _peer) When a packet destined for 192.0.2.2 hits INIT NS, an eBPF program intercepts it. Instead of passing it up the host's network stack, it executes a bpf redirect directly to the peer interface (_peer) in TEST NS.

3. Hardware Acceleration via Queue Leasing (Leased Path) The Queue Leasing Controller slices the physical NVIDIA NIC and allocates a dedicated subset of hardware queues (Queues: 12-15) directly to TEST NS via a Leased Path. It is configured with a Jumbo Frame MTU of 9000.



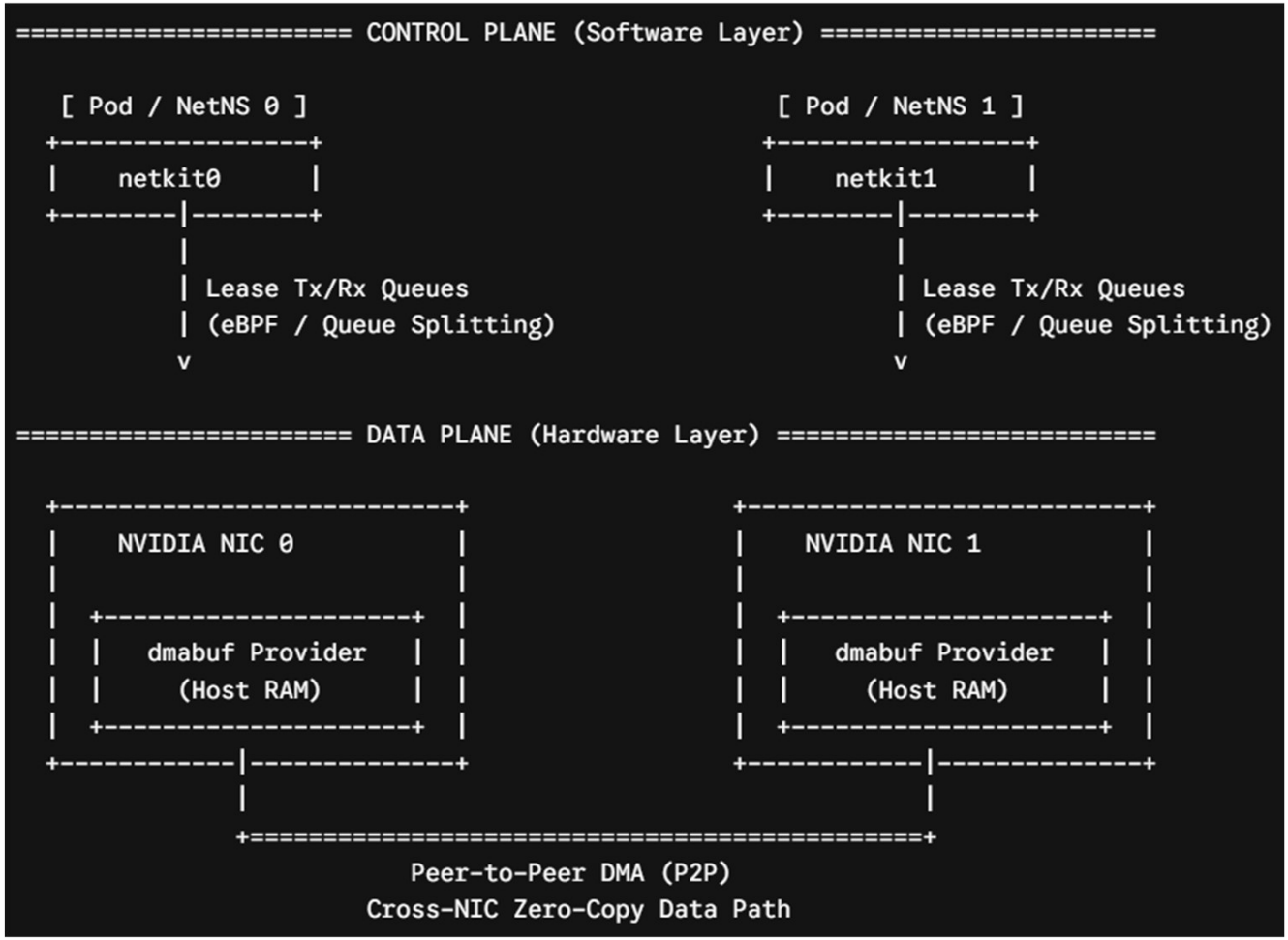
Netkit and Phy NIC Architecture

The Core Challenge:Traditional container networking (e.g., standard veth pairs) suffers from high CPU overhead and latency due to multiple memory copies when moving data across different physical NICs.

Software Innovation (Control Plane):netkit combined with eBPF / Queue Splitting bypasses the core network stack by leasing hardware Tx/Rx queues directly to individual Pods.

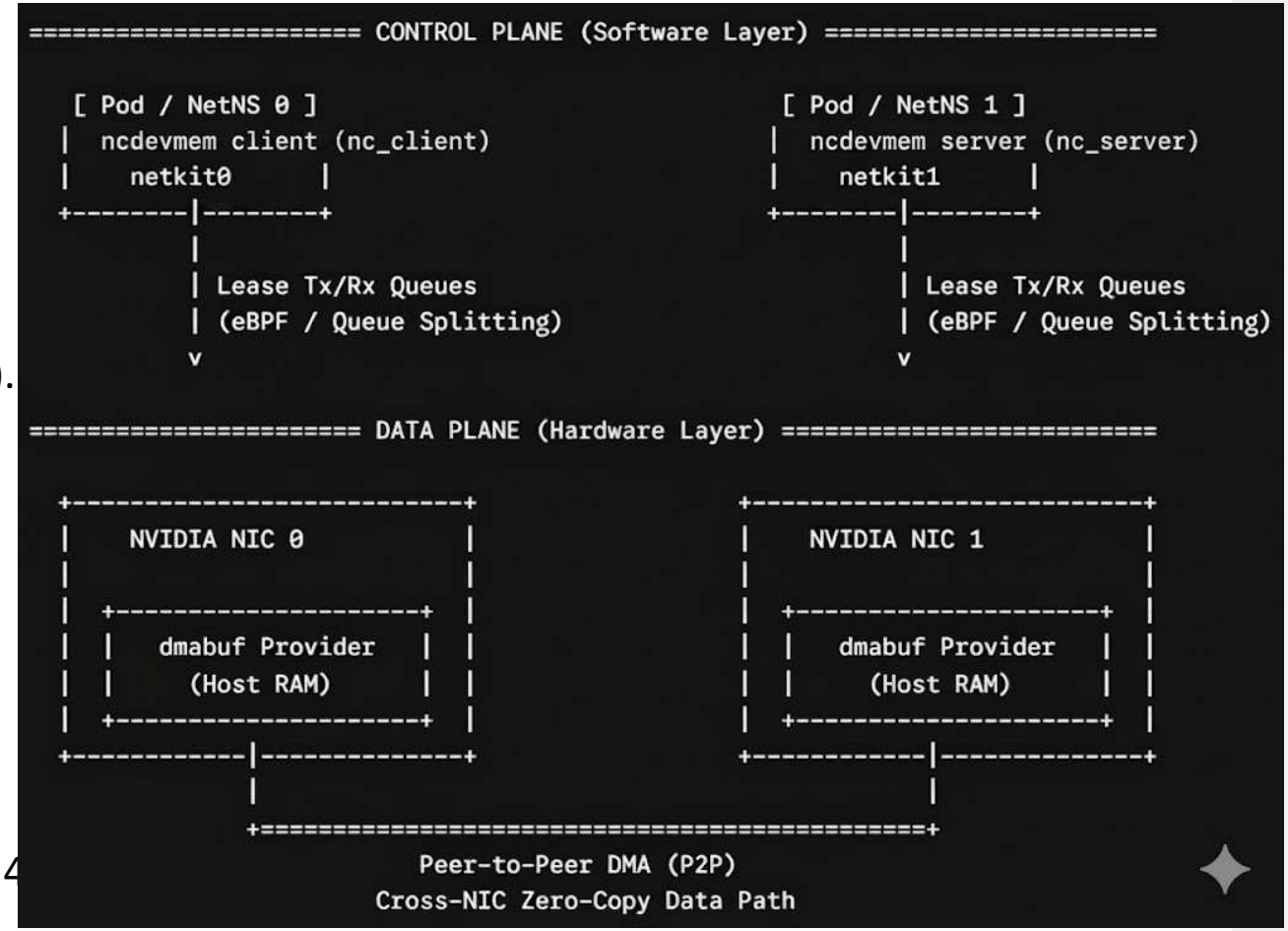
Hardware Innovation (Data Plane):Unified memory management via dmabuf Provider pairs with Peer-to-Peer DMA (P2P) to enable direct NIC-to-NIC hardware-level data transfers.

Key Performance Outcomes:Absolute Zero-Copy: Zero CPU memory-copy cycles during cross-NIC boundary data transfers.Hardware-Enforced Isolation: Dedicated queue leasing guarantees deterministic throughput per Pod.



Ncdevmem work over netkit

Intro: Device Memory TCP (devmem TCP) optimizes high-volume transfers by bypassing host memory copies:Header Split: NIC splits incoming packets; headers go to host RAM for TCP processing.
Payload Direct-to-Device: Data payloads land directly in a bound dmabuf (e.g., GPU memory).
Zero-Copy: CPU never touches the payload, significantly reducing PCIe and memory bandwidth stress.
How to use it:
NIC Setup
Enable header split and flow steering on the interface.
ethtool -G eth1 tcp-data-split on
ethtool -K eth1 ntuple on
Server: ncdevmem -s 192.168.1.100 -p 5201 -q 4
d dmabuf_id_0 -f 2048
Client: ncdevmem -c 192.168.1.100 -p 5201 -f /path/to/large_test_file.bin



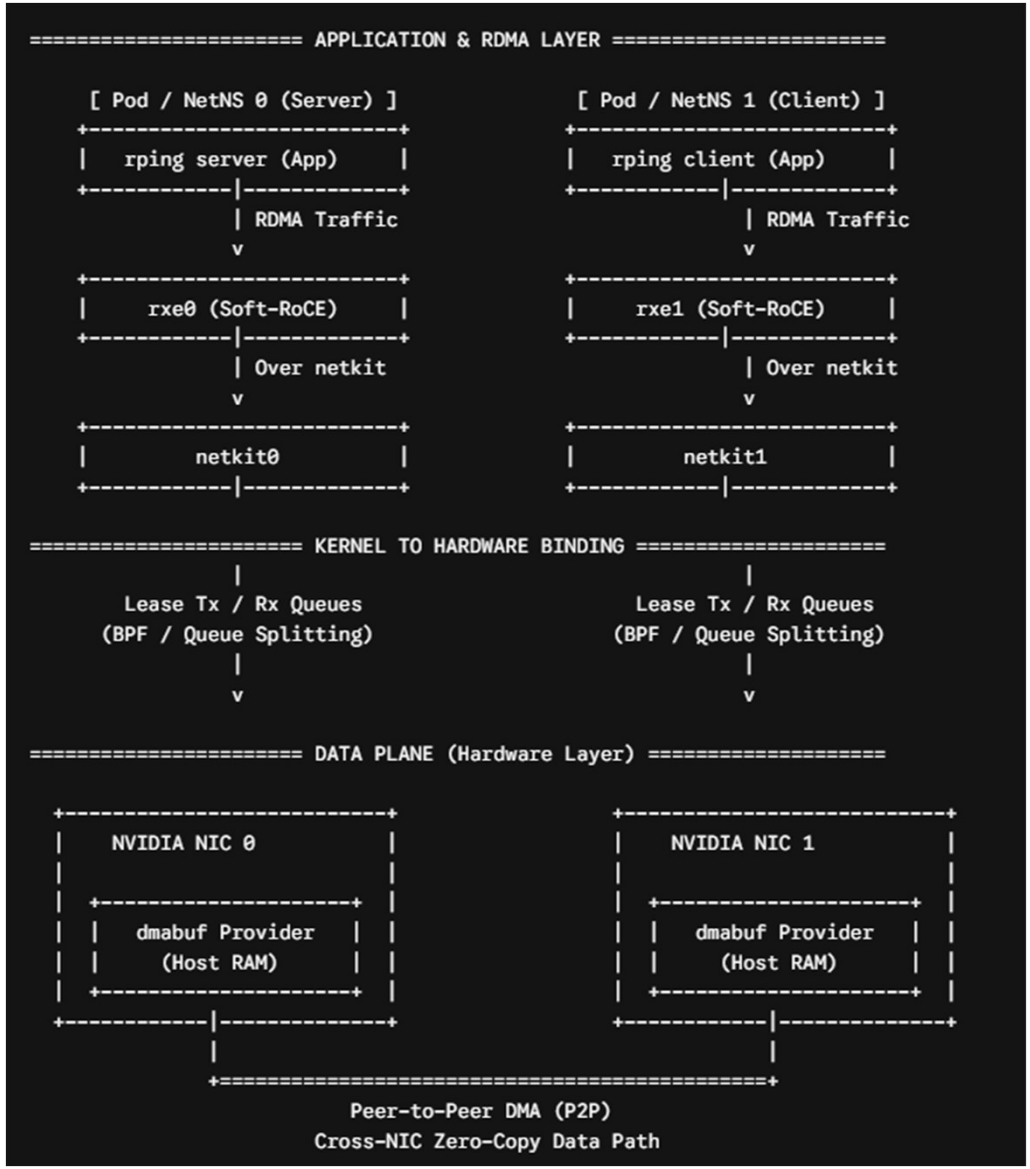
mmsg pack: 2048 total received bytes: 10737418240 (10.00 GB) devmem bytes: 10737418240 (100.00% zero-copy) regular bytes: 0

Rping work over Rxe+netkit+ devmem

1. Add rxe0 on netkit0, rxe1 on netkit1;
2. Run rping server on rxe0, rping client on rxe;
3. Rping should work well.

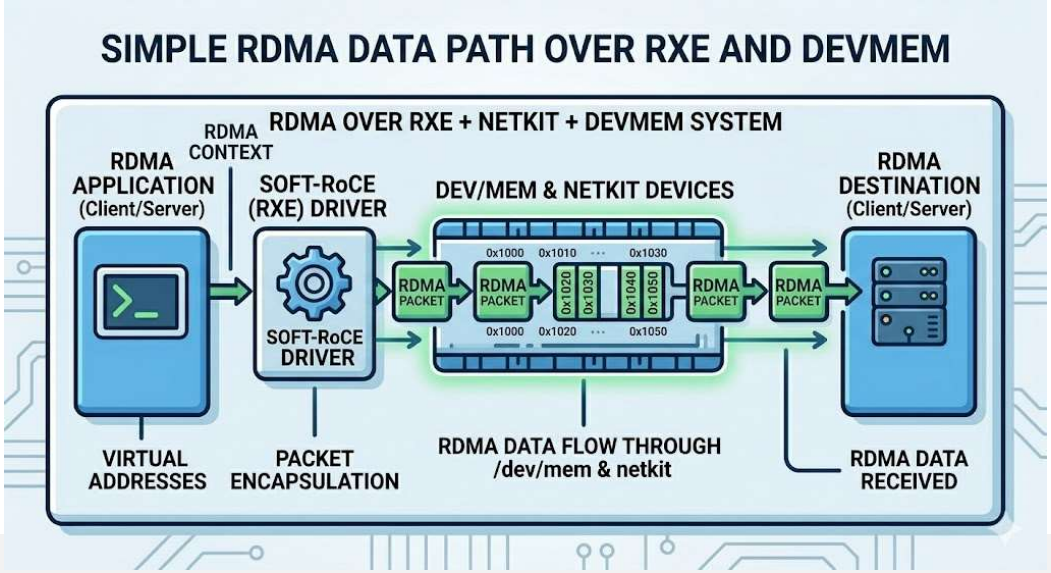
Result:
Server: rping -s -a 192.168.1.100 –v
Client: rping -c -a 192.168.1.100 -v -C 3

ping data: rdma-ping-0: ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqr
ping data: rdma-ping-1: BCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrs
ping data: rdma-ping-2: CDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrst



Summary: RDMA work on devmem via netkit

1. Setup a devmem test environment, via netkit, ncdevmem can work well.
2. Add rxe on netkit, rping can work well.
3. The RDMA application can work well with devmem via netkit.
4. This provides a solution for RDMA application based on devmem device.



Future work

1. High-Performance RDMA Packet Dumping via netkit & eBPFThe Challenge:
   Existing RXE packet dumping methods hit engineering or performance bottlenecks.
   AF_XDP + Underlying XDP Handler: Feasible (as discussed at LSF/MM/BPF 2023) but highly inconvenient. It relies on modifying the underlying physical NIC driver source code.
   kprobes (rxe_xmit / rxe_recv): Functional for debugging, but the context-switching overhead causes severe performance degradation under high loads.
   The Proposal: Leverage netkit + tcxCan we integrate netkit's native architecture to hook eBPF tcx programs directly into the RXE path?This would enable high-performance, non-intrusive RDMA packet dumping without touching the physical NIC driver or suffering kprobe overhead.

2. Bypassing the Network Stack: RXE + devmem via Queue Leasing
   Current State: RXE operates at the IP layer, utilizing the standard network stack for packet transmission and reception.
   The Next-Gen Architecture: Moving RXE directly to the NIC Driver Layer
   Objective: Eliminate IP/Ethernet layer overhead by allowing RXE to interface directly with the physical NIC's xmit and recv functions.
   The netkit + devmem Integration: netkit can already leverage devmem by leasing TX/RX queues from the physical NIC.
   The Core Question: Can RXE adopt this exact mechanism—leasing hardware RX/TX queues directly from the underlying NIC—to achieve zero-copy, hardware-accelerated devmem data paths while bypassing the kernel network stack?

Thanks for your attention!