

Inter-Kernel Networking for Multikernel Architecture

November 8, 2025

Cong Wang cwang@multikernel.io

Founder & CEO, Multikernel Technologies

Multikernel Architecture

- Multiple independent Linux kernel instances running on the same physical machine
- Each kernel instance has dedicated resources:
 - Dedicated CPU cores
 - Isolated physical memory partitioning
 - Separate PCI/platform devices
 - Dynamic allocation via IPI messaging

Multikernel Shared Memory

- Per-instance memory pools using `gen_pool`
- Physical memory regions managed via resource hierarchy
- Memory isolation between instances
- IPI uses shared ring buffer to transmit messages

Multikernel IPI Messaging

- Lock-free ring buffer with 64 slots
- 256 bytes per message (`MK_MAX_DATA_SIZE`)
- Direct IPI delivery between instances
- Message-oriented architecture with type handlers

Current Message Types

- Dynamic resource management (CPU, memory, devices)
- I/O interrupt forwarding (doorbell)
- System control messages (TBD)

Why Inter-Kernel Communication

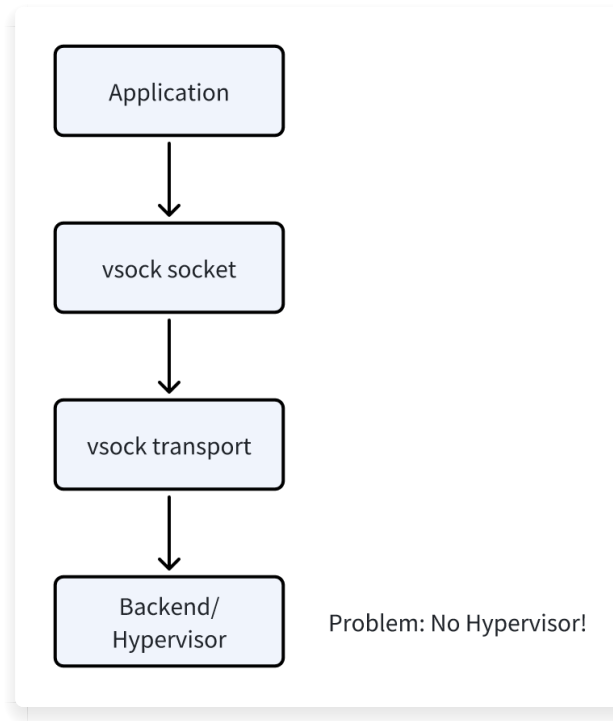
- Dynamic resource management is already built in the kernel
- Kernel auto-healing: heartbeat + resource takeover
- Kernel live-update: requires a sophisticated live-update protocol (compete with LUO)
- General networking

Possible Solutions

We evaluated four main approaches for inter-kernel networking:

- **Option 1:** Vanilla `vsock`
- **Option 2:** Software NIC (`virtio-net` style)
- **Option 3:** Multikernel Socket Interface
- **Option 4:** `virtio` Device Framework

Option 1: Vanilla **vsock**



Key Problem

vsock's assumptions:

- Hypervisor/backend (trusted) <-> Guest OS (untrusted)
- Unidirectional trust model
- Central hypervisor manages communication

multikernel reality:

- All kernels are equal peers
- Bidirectional, symmetric communication
- No central hypervisor needed

Evaluation

Advantages:

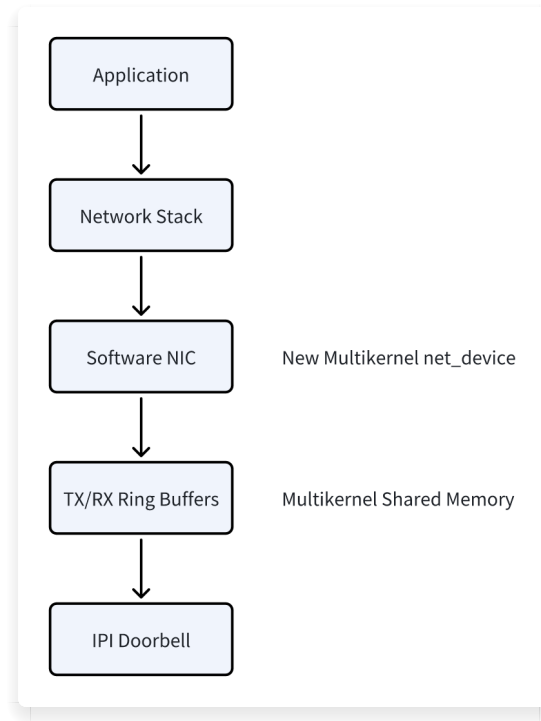
- Well-established `AF_VSOCK` socket API
- Stream and datagram support
- Many existing tools work (`socat`, `nc`, etc.)

Disadvantages:

- Requires significant adaptation of vsock core assumptions
- Designed for unidirectional trust model
- May not optimally leverage our IPI messaging infrastructure

Verdict: Possible but requires rethinking vsock's hypervisor-centric model

Option 2: Software NIC



Implementation Details

- Create virtio-net-like driver in `drivers/net/`
- Shared memory ring buffers for TX/RX operations
- IPI for packet notifications between instances
- Full network stack integration (IP, TCP, UDP layers)

Evaluation

Advantages:

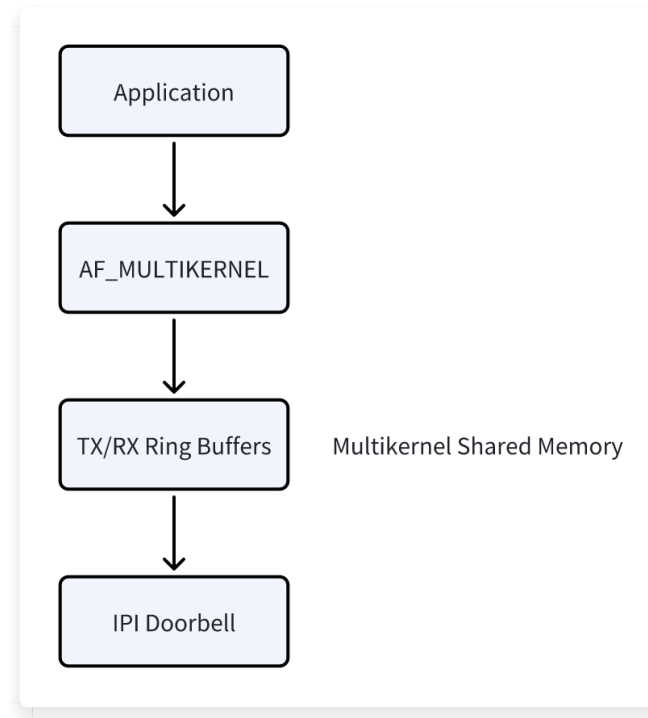
- Proven model (e.g. `virtio-net`, `veth`)
- Transparent network stack integration (IP, TCP, UDP)
- Can assign IP addresses per instance

Disadvantages:

- High implementation complexity (TX/RX rings, SKB management)
- Network stack overhead for local communication
- Memory overhead for packet buffers

Verdict: Too complex for the problem at hand

Option 3: Multikernel Socket Interface



Design Layers

Bottom Layer: Enhanced data transport

- IPI for control messages and small data
- Shared memory circular buffers for bulk data
- Credit-based flow control (like vsock)

Middle Layer: Stream/Datagram abstraction

Top Layer: Socket interface (`AF_MULTIKERNEL`)

Evaluation

Advantages:

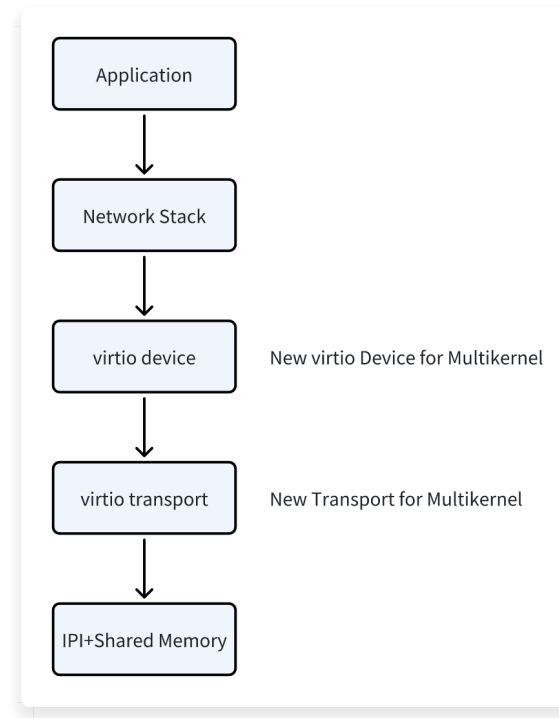
- Leverages existing IPI + shared memory
- Has minimum layers
- Low latency for small messages, high bandwidth for bulk transfer

Disadvantages:

- Requires userspace to use new socket family (`AF_MULTIKERNEL`)
- Literally reinventing `vsock`

Verdict: Not userspace application friendly

Option 4: **virtio** Device Framework



Key Benefits

- Standardized `virtqueue` ring buffers
- Device discovery through `virtio` bus
- Feature negotiation
- Standardized `virtio` specification

Architectural Mismatch

virtio assumes device/driver split:

- Device side: Hypervisor/backend (trusted)
- Driver side: Guest OS (untrusted)
- Unidirectional trust model

multikernel is peer-to-peer:

- All kernels are equal peers
- Bidirectional, symmetric communication
- No device/driver distinction

Unnecessary Complexity

With `virtio`, we need 4 abstraction layers:

1. Implement `virtio` transport
2. Setup `virtqueues`: rings, descriptors
3. Implement device: like `virtio-vsock` or `virtio-net`
4. Protocol layer

Evaluation

Advantages:

- Proven ring buffer implementation
- Well-established virtualization infrastructure

Disadvantages:

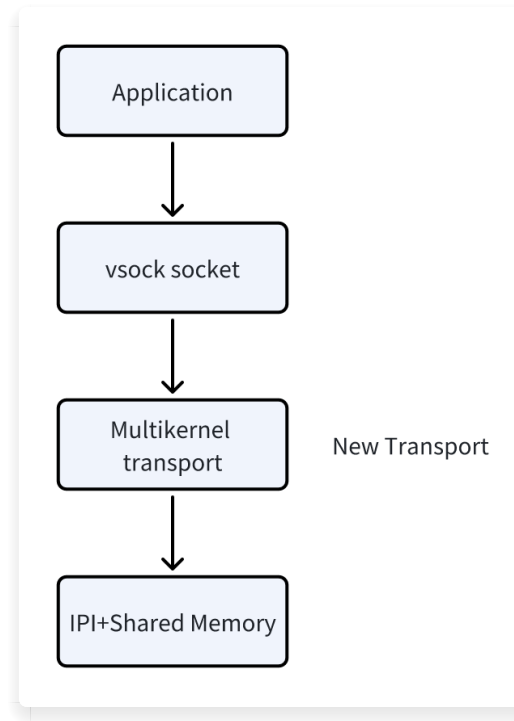
- Architectural mismatch (device/driver vs peer-to-peer)
- Unnecessary abstraction layers
- High implementation complexity

Verdict: Complex abstraction for this problem

Our Design: Direct `vsock` Transport

- Key Insight: **vsock is transport-agnostic**
- `vsock` infrastructure supports multiple transport types
- For multikernel: Use `VSOCK_TRANSPORT_F_LOCAL` and treat all instances as local peers!

Architecture



Implementation Overview

```
/* Transport operations structure */
static struct vsock_transport mk_transport = {
    /* Lifecycle */
    .init          = mk_vsock_init,
    .destruct       = mk_vsock_destruct,
    .release        = mk_vsock_release,
    /* Connection management */
    .connect        = mk_vsock_connect,
    .cancel_pkt     = mk_vsock_cancel_pkt,
    /* Stream operations - map to shared memory */
    .stream_dequeue = mk_vsock_stream_dequeue,
    .stream_enqueue = mk_vsock_stream_enqueue,
    /* ... more operations ... */
};
```

Address Mapping

Direct CID Mapping

vsock CID	→ multikernel instance ID
VMADDR_CID_HOST (2)	→ root instance (ID 0)
CID 3, 4, 5...	→ spawn instances (ID 1, 2, 3...)

Code Example

```
struct sockaddr_vm sa = {  
    .svm_family = AF_VSOCK,  
    .svm_cid = 3,           // CID 3 = instance 1  
    .svm_port = 9000,  
};  
connect(sock, (struct sockaddr*)&sa, sizeof(sa));
```

Transport Design

- Register as local transport: Peer-to-Peer
- **Stream operations**: Map to shared memory
- **Datagram operations**: Direct IPI messaging
- **Addressing**: Trivial CID to instance ID mapping
- **Flow control**: Credit-based inherited from `vsock`

Data Path Implementation

Small Messages (< 256 bytes): Direct IPI

- Send data directly via existing IPI messaging infrastructure
- Single function call with minimal overhead

Large Messages: Shared Memory + IPI Doorbell

- Copy data to shared memory ring buffer
- Send lightweight doorbell via IPI
- Receiver reads data from shared memory
- Combines high bandwidth with low-latency signaling

Shared Memory Structure

```
/* Extend existing mk_shared_data */
struct mk_shared_data {
    struct mk_ipi_ring ring;          /* Existing: control messages */

    /* Add: network data rings */
    struct mk_net_ring tx_ring;      /* TX data ring */
    struct mk_net_ring rx_ring;      /* RX data ring */

    /* Flow control */
    atomic_t tx_credits;
    atomic_t rx_credits;
};

struct mk_net_ring {
    atomic_t head;
    atomic_t tail;
    u32 size;                        /* Power of 2, e.g., 1024 */
    u32 buffer_size;                 /* Size of each buffer slot */
    u8 buffers[];                    /* Circular buffer data */
};
```

Comparison Matrix

Feature	vsock Transport	Software NIC	Socket Interface	virtio
Implementation Complexity	Low	High	Medium	Very High
Userspace Compatibility	Standard AF_VSOCK	Standard sockets	New API	Standard
Leverages Existing Infra	Yes	Partial	Yes	No
Abstraction Layers	2	3	2	4
Peer-to-Peer Model	Yes	Yes	Yes	No

Performance Evaluation

- **Small messages:** Direct IPI path (faster than `virtio` descriptor chains)
- **Large messages:** Shared memory (faster than network stack processing)
- **No descriptor overhead** (vs `virtio`'s descriptor management)
- **No network stack overhead** for local communication (vs Software NIC)
- **Efficient notification:** IPI only when needed (vs polling)

Conclusion

Evaluated Solutions:

1. Vanilla `vsock` - requires hypervisor adaptation
2. Software NIC - too complex, unnecessary overhead
3. Socket Interface - breaks userspace compatibility
4. `virtio` framework - architectural mismatch

Our Choice: Direct `vsock` Transport

Thank You!

Questions & Discussion

Cong Wang

Founder & CEO, Multikernel Technologies

www.multikernel.io