ptnetmap:
a netmap passthrough for virtual machines

Stefano Garzarella, Giuseppe Lettieri, Luigi Rizzo
stefanogarzarella@gmail.com, g.lettieri@iet.unipi.it, rizzo@iet.unipi.it,

Università di Pisa

AsiaBSDCon 2015, Tokyo
March 15, 2015

Outline

- Introduction
- Background
  (netmap, paravirtualized device, netmap backend)
- ptnetmap
  - architecture
  - implementation
- Performance evaluation

Virtual Machines (VMs)

- widely used to build Cloud services:
  - modular
  - flexible
  - secure

- performance bottlenecks, especially for intensive I/O operations
  - networking services:
    - router
    - firewall
    - middle-boxes

Fast network interfaces (10/40 Gbps)

TCP is ok thanks to GSO/TSO

high packet rates are hard everywhere:

- Physical servers
  - OS-bypass (Intel DPDK, PFRING DNA)
  - network stack bypass (netmap)
- Virtual Machines (VMs)
  - hardware passthrough
  - fast switches (eg. VALE)

Virtual Machines (VMs)

- VMs solutions have some **drawbacks**:
  - **fast switches**
    - hypervisor frontend/backend overhead
  - **hardware passthrough**
    - all communications (even among VMs) use the PCIe bus
    - strictly hardware dependent (VMs migration difficult)

virtual passthrough

our proposal: ptnetmap

- uses netmap API to implement the virtual passthrough of any device supported by the netmap framework
- fully hardware independent
- high-speed communication with:
    - physical NICs (14.88 Mpps - 10Gbps line rate)
    - VALE ports (20 Mpps)
    - netmap-pipes (75-150 Mpps)

virtual passthrough (2)

- most of **netmap strengths**:
  - vendor independence
  - use commodity hardware
  - avoid busy polling

- **flexible memory sharing**:
  - VALE ports
    - to isolate untrusted VMs
  - netmap pipes
    - to create chains of trusted VMs

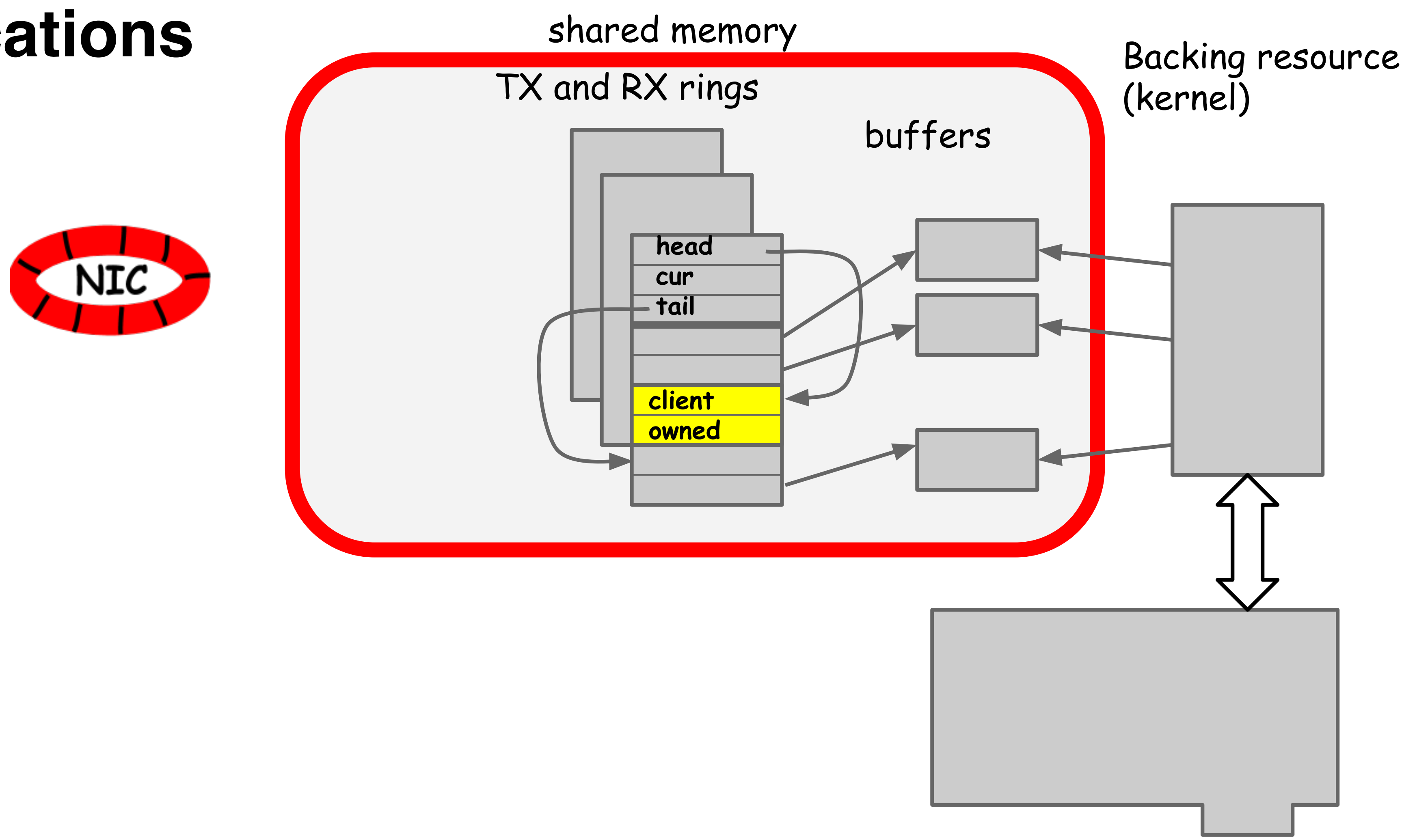
Background

ptnetmap leverages on:

- **netmap framework**
  - L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. *USENIX ATC'12*
  - L. Rizzo and G. Lettieri. VALE, a switched ethernet for virtual machines. *CoNEXT '12*
- **paravirtualized ethernet devices**
  - R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev. '08*
  - L. Rizzo, G. Lettieri, and V. Maffione. Speeding up packet I/O in virtual machines. *ANCS '13*

netmap framework

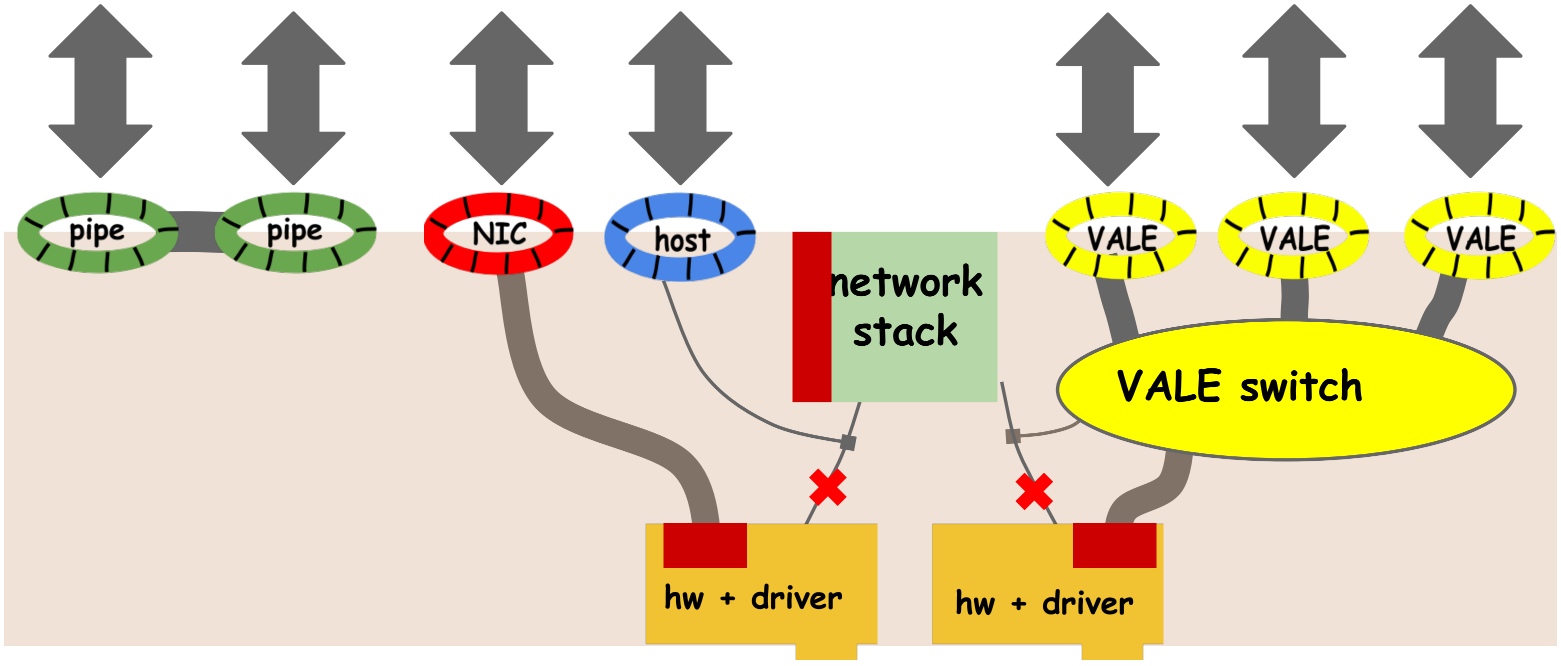
- The netmap framework provides **high speed network ports to clients**
  - userspace applications
  - in-kernel applications



netmap framework

netmap ports can be

- **physical NICs**
- **port of a software switch (VALE switch)**
- **high performance, shared memory channel, called netmap pipe**
- **other types of ports are also present, e.g. for mirroring, logging etc.**



paravirtualized ethernet devices

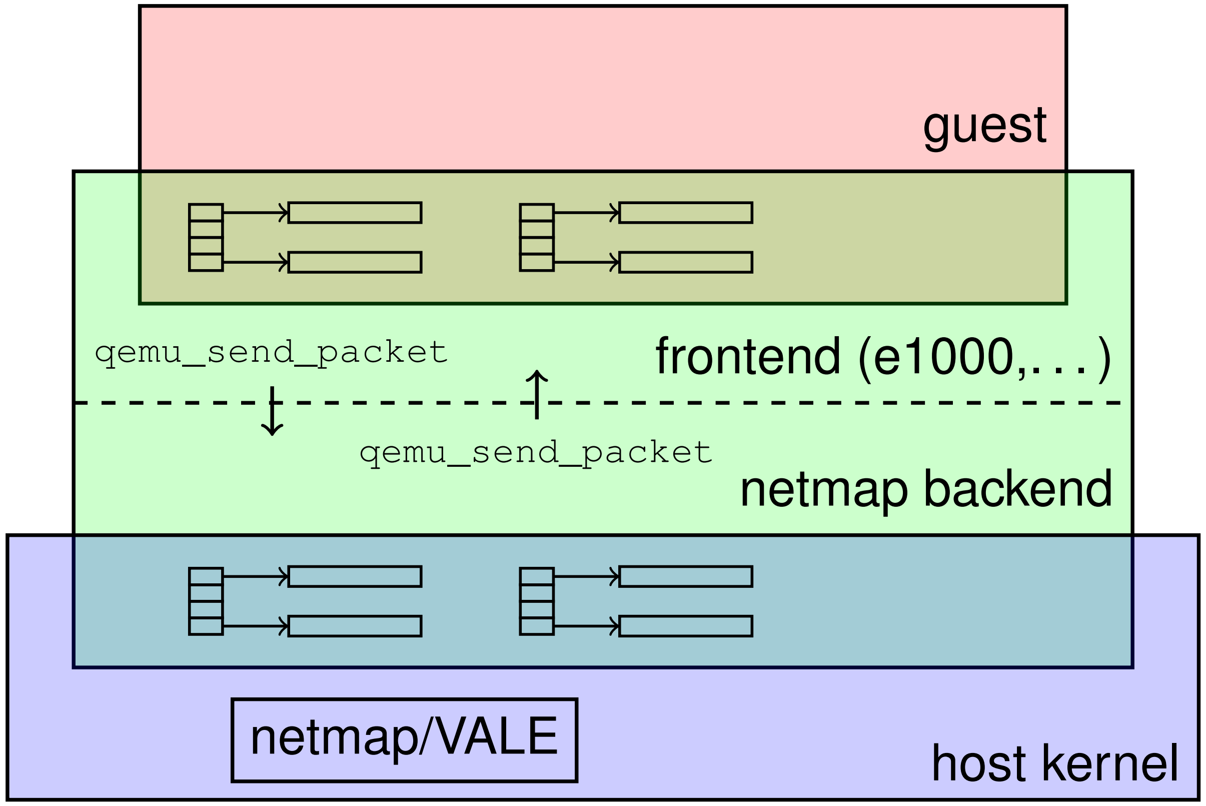
Goal: Minimize consumer/producer notifications

- guest/host notification drawback:
    - VM Exit
    - interrupts
- we slightly modified legacy Ethernet device emulation (e1000) and the guest drivers to work like virtio:
    - one thread on each side activated on demand
    - shared memory to exchange status information
    - avoid notifications
        - threads actively poll the shared memory when there is traffic
        - sends notifications, through NIC register (VM exit), only when the other part is sleeping

network path in VMs

- **frontend: emulates the hardware that the guest device driver expects to talk to**
- **backend: transfers data packets to the actual network port on the host**
- **frontend/backend data exchange**
  - **data format conversion**
    - **touch every packet**
    - **data copy**

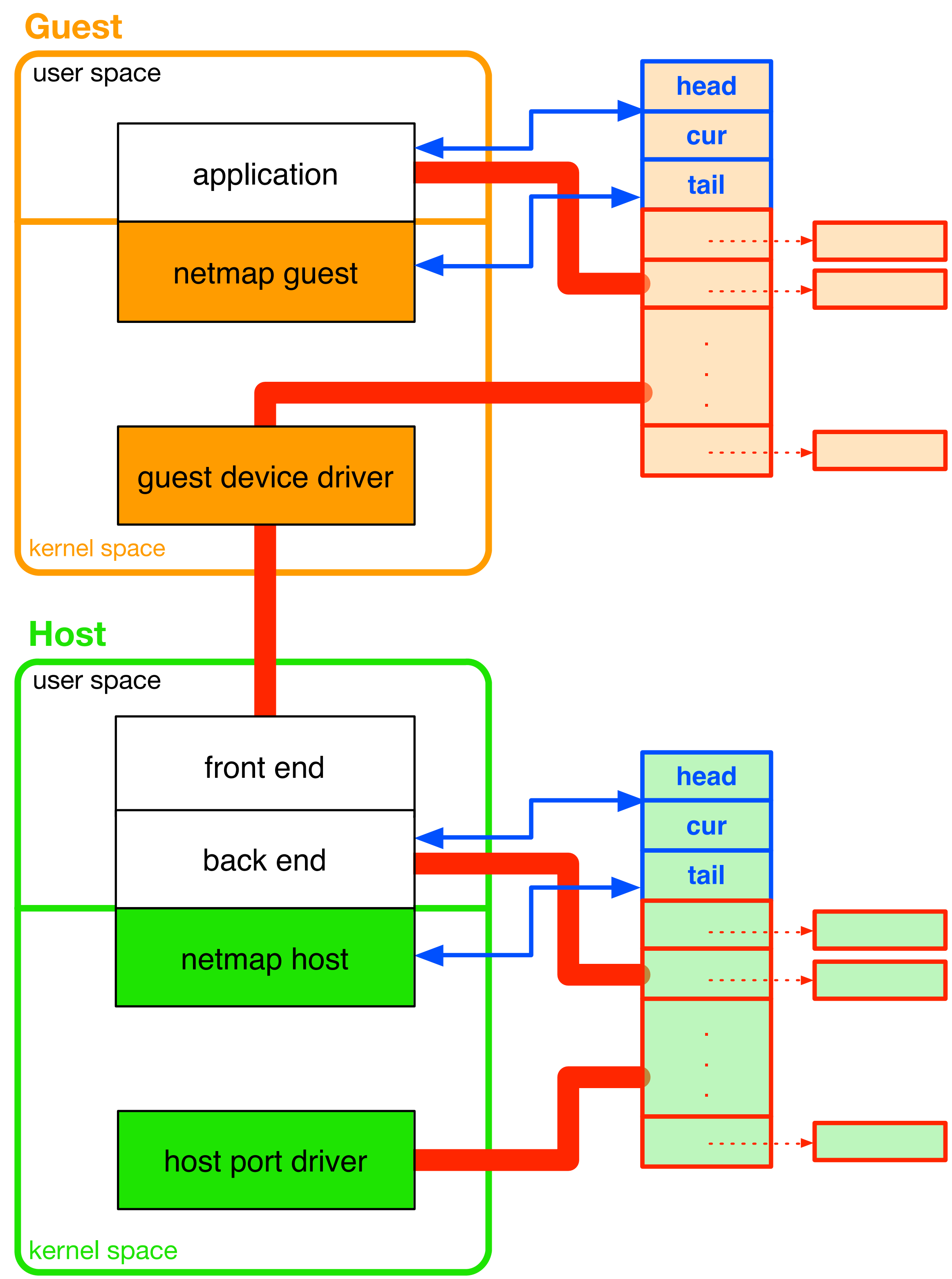
- **netmap backend**
  - **uses a fast netmap port in the hypervisor**
  - **now is available for:**
    - **QEMU**
    - **bhyve**



network path in VMs (2)

even using netmap in the guest we are inefficient:

- **netmap guest is completely unaware of the netmap host** (and vice versa)
  - packets need to be copied along the way:
    - guest - frontend
    - frontend - backend.

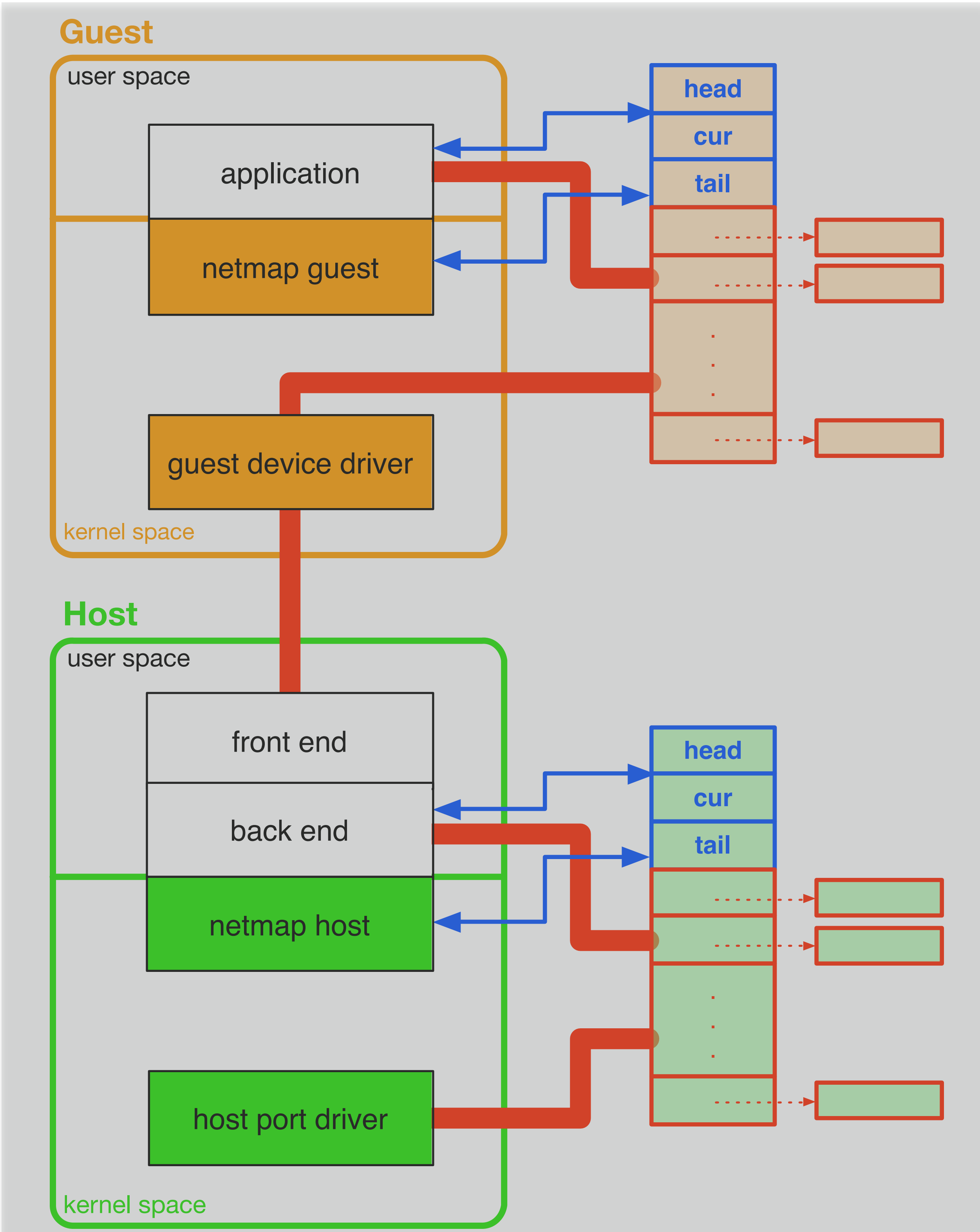


ptnetmap: architecture

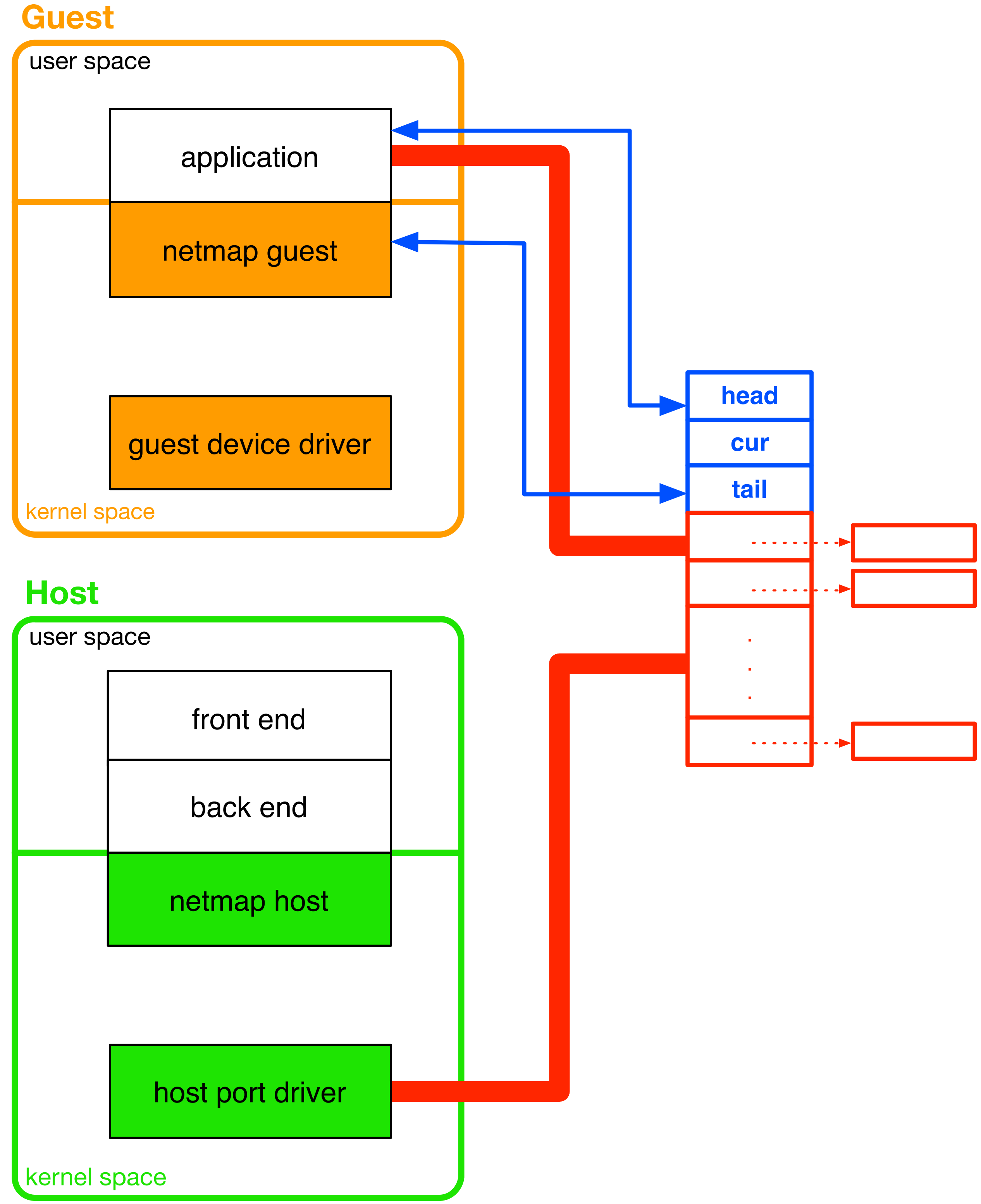
- ptnetmap
  - reduces the overhead induced by the hypervisor frontend and backend
  - provides passthrough mechanism to directly access netmap host devices from a virtual machine
  - data regions are exposed, in a protected way, to the guest VM, such as hardware passthrough

ptnetmap: architecture

without ptnetmap

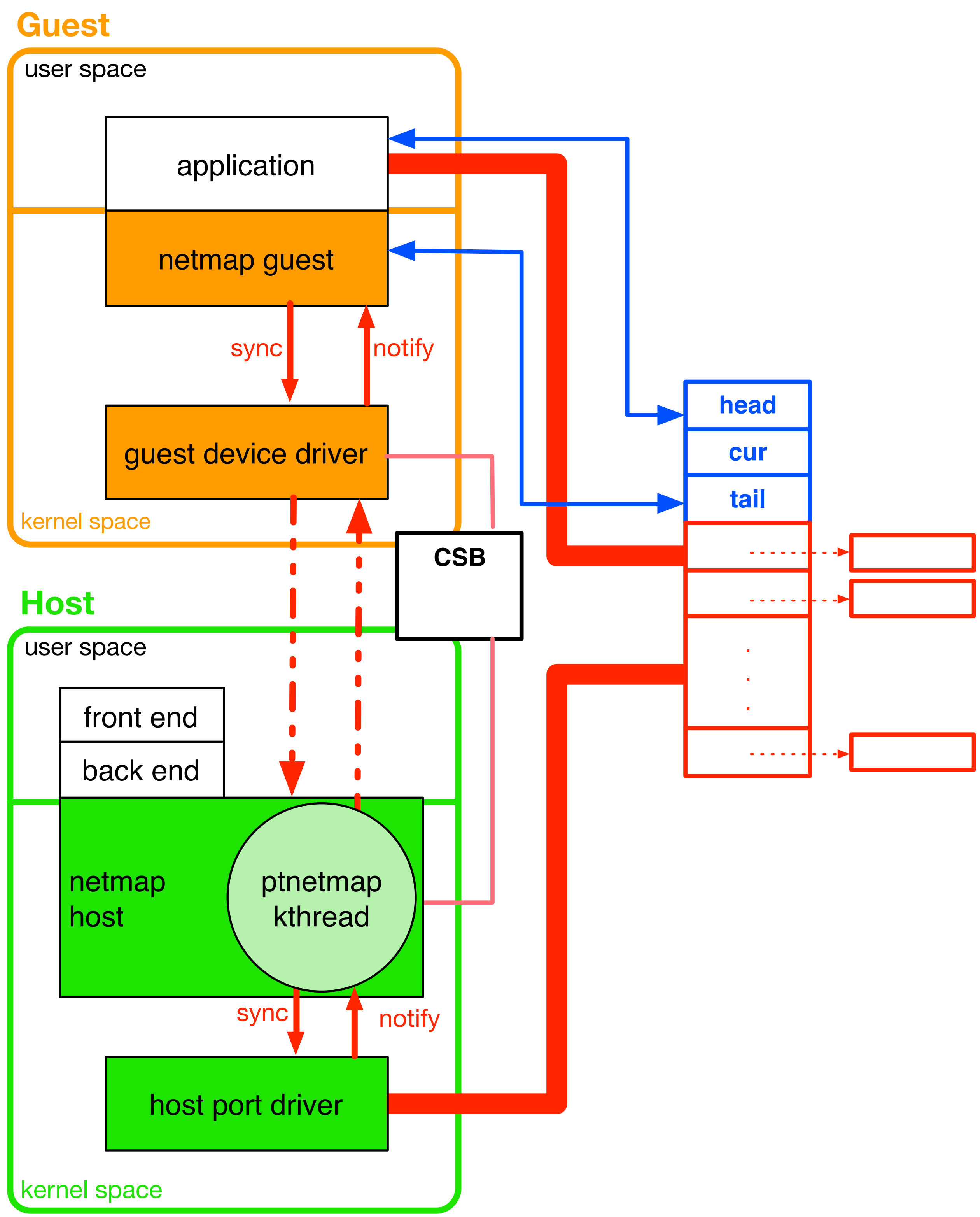


ptnetmap



ptnetmap: architecture

- **CSB (Communication Status Block)**
  - **shared memory page between guest and host to exchange messages and information on their status**
- **netmap system calls issued by the guest application**
  - **synchronized with netmap host using the CSB**
  - **notification (VMExit) if the host kernel thread is sleeping**



ptnetmap: implementation

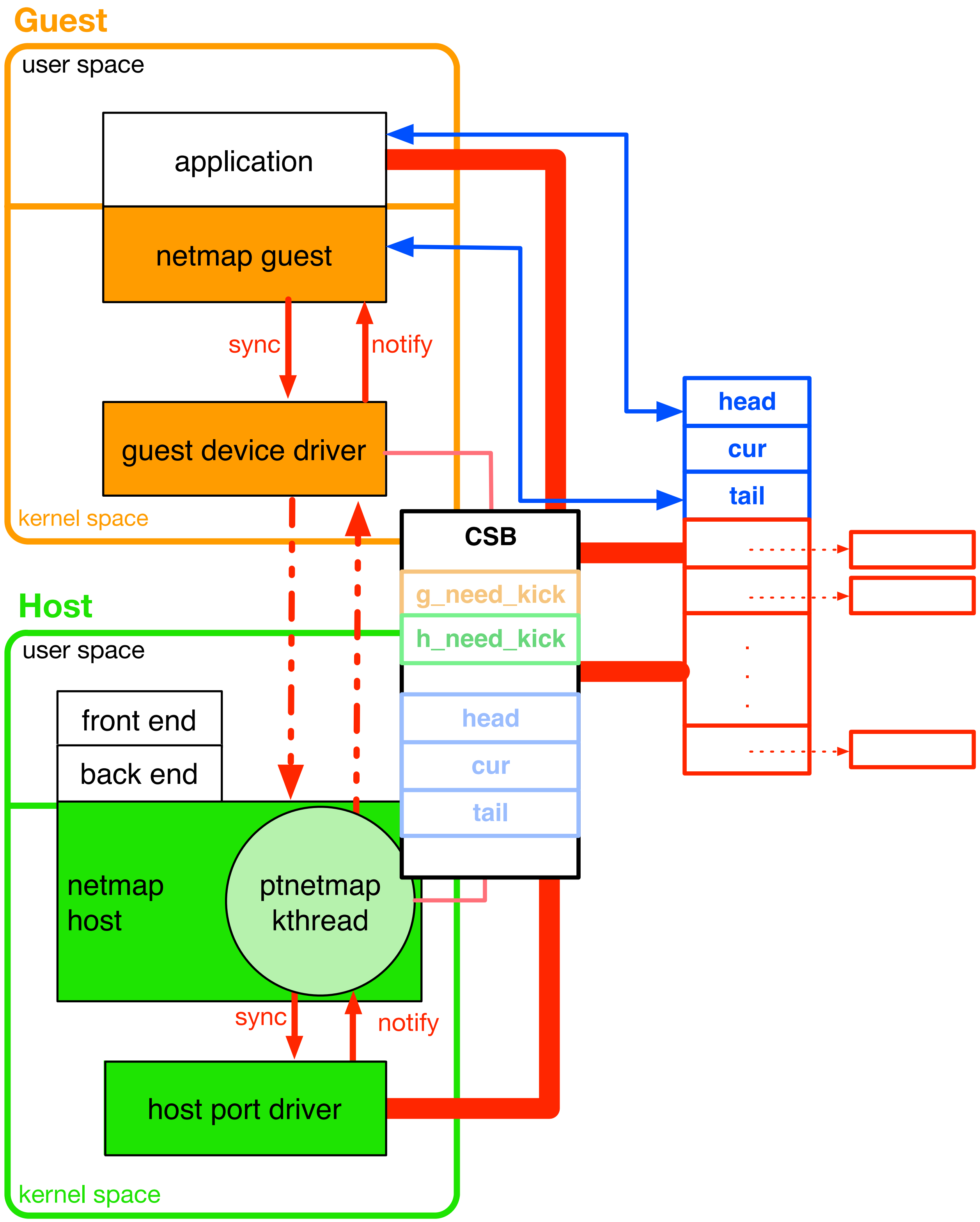
- The implementation of ptnetmap requires small extensions:
  - netmap framework
  - guest driver for the paravirtualized device
  - hypervisor

- Host netmap memory area
  - mmap()ed by the netmap backend in the hypervisor
  - exported by the hypervisor frontend as memory residing on the emulated paravirtual device, described by a PCI BAR
  - mapped by the driver into the guest memory during the device initialization

ptnetmap: implementation

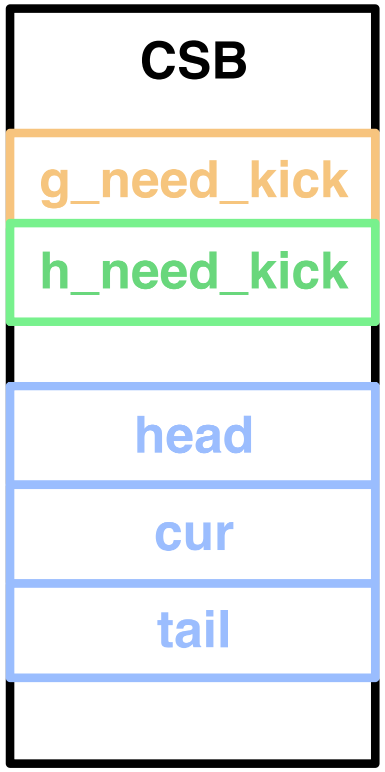
- **notifications guest/host**
  - **trigger action when new packets and/or free slots are available**
  - **avoid busy-wait loops**

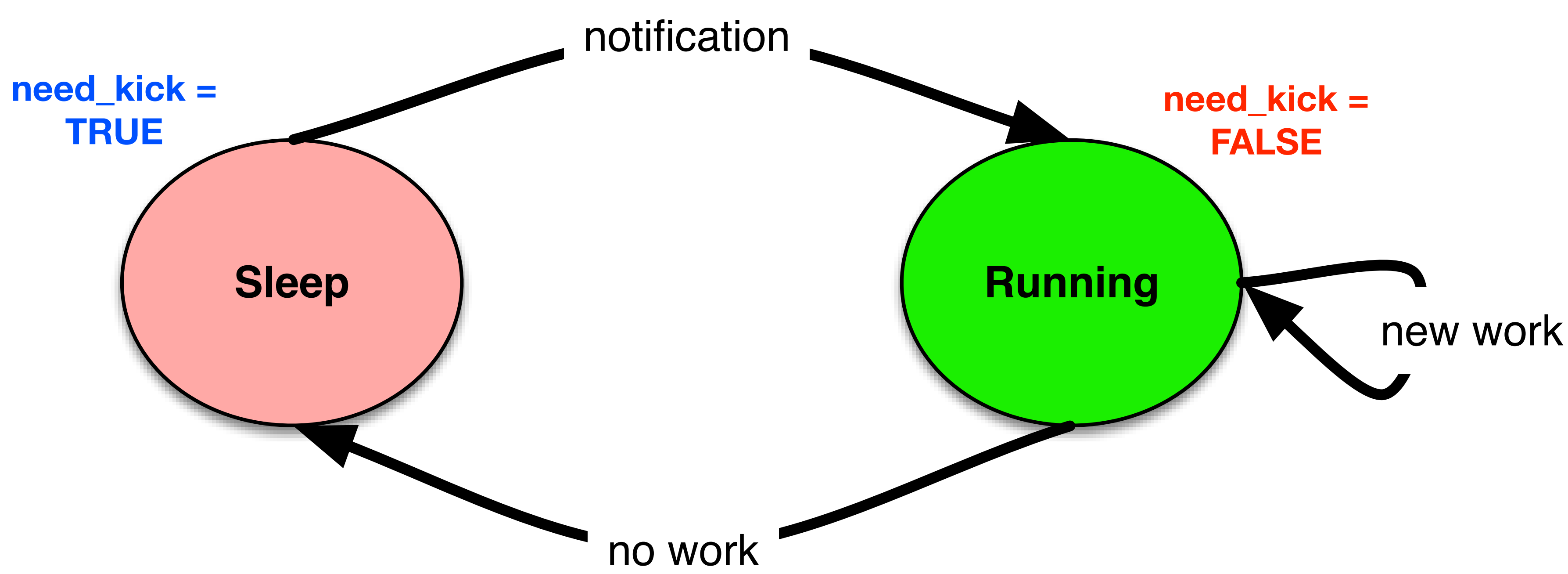
- **guest in-kernel ring states**
  - **updated asynchronously with the corresponding host in-kernel states, using the CSB**



ptnetmap: implementation

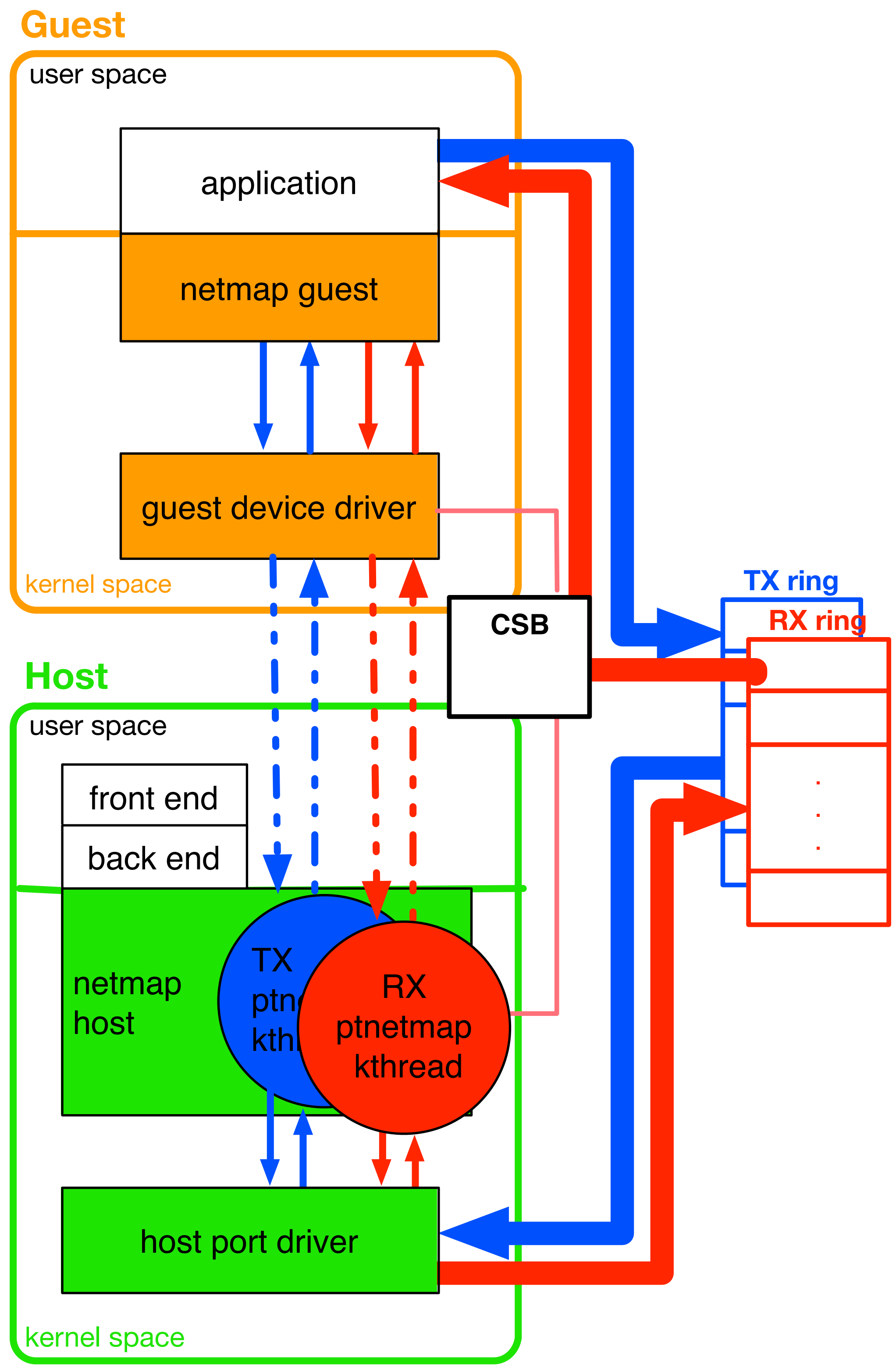
- The data-path adopts a virtio-like interaction between guest and host:
  - shared memory page (CSB) contains:
    - copy of the ring pointers
    - flags to disable notifications
  - threads sleep when there isn't work to do
    - avoid busy-wait loops





ptnetmap: implementation

- two **kernel threads** in the host netmap adapter:
  - RX ring
  - TX ring



Performance: metrics

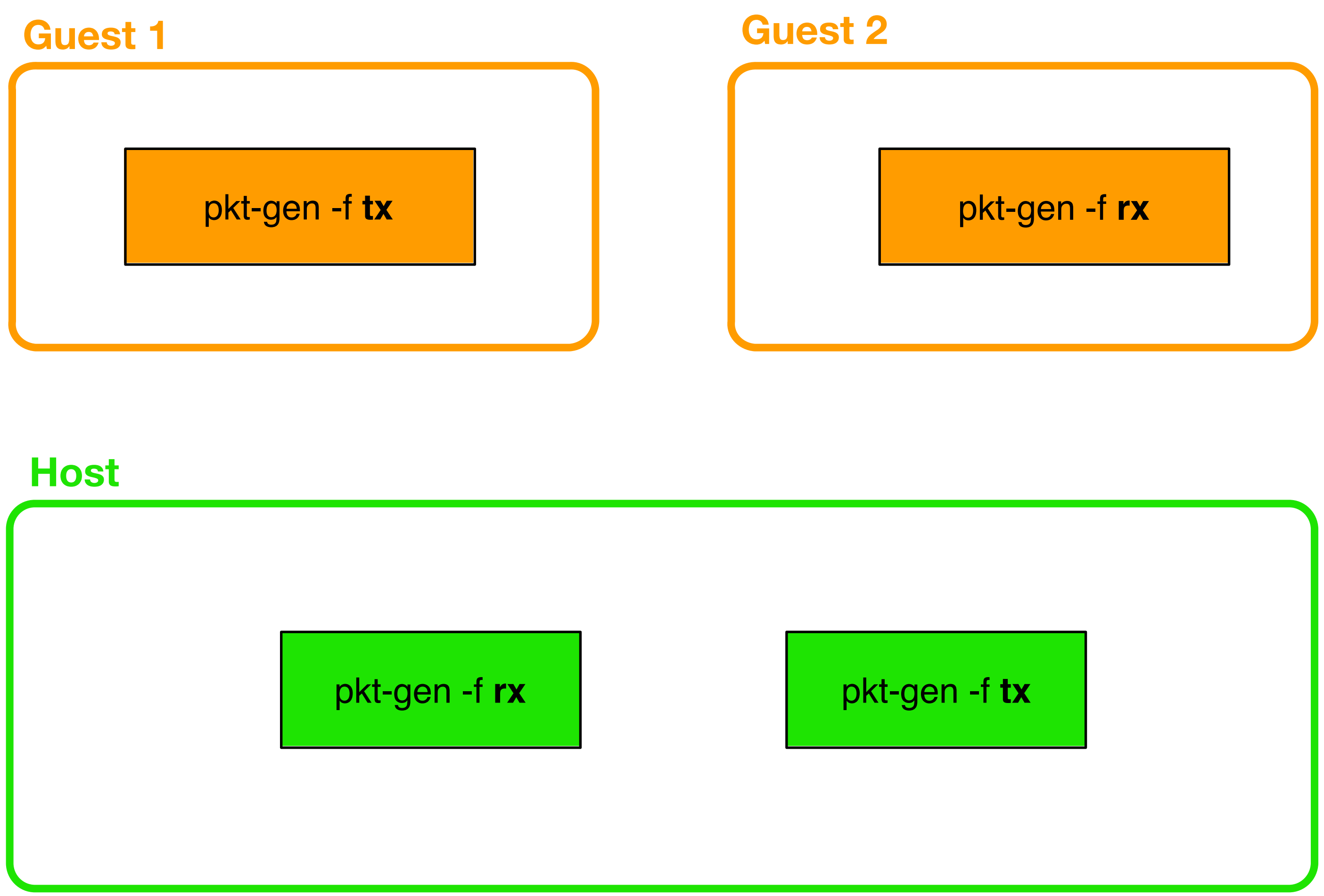
- **Throughput**
  - `pkt-gen`: general purpose sender/receiver with configurable:
    - packet size
    - rate
    - batch size
    - number of threads.
  - `poll()` to do I/O
    - blocks when there are no slots in the queue to send or receive.

Performance: metrics (2)

- Latency
  - `pkt-gen` ping/pong
    - sender
      - transmits the packet
      - waits for a response (`poll()`)
    - receiver
      - waits for a packet (`poll()`)
      - immediately bounces it back.
    - packets carry a timestamp generated by the sender
      - measure the Round Trip Time.

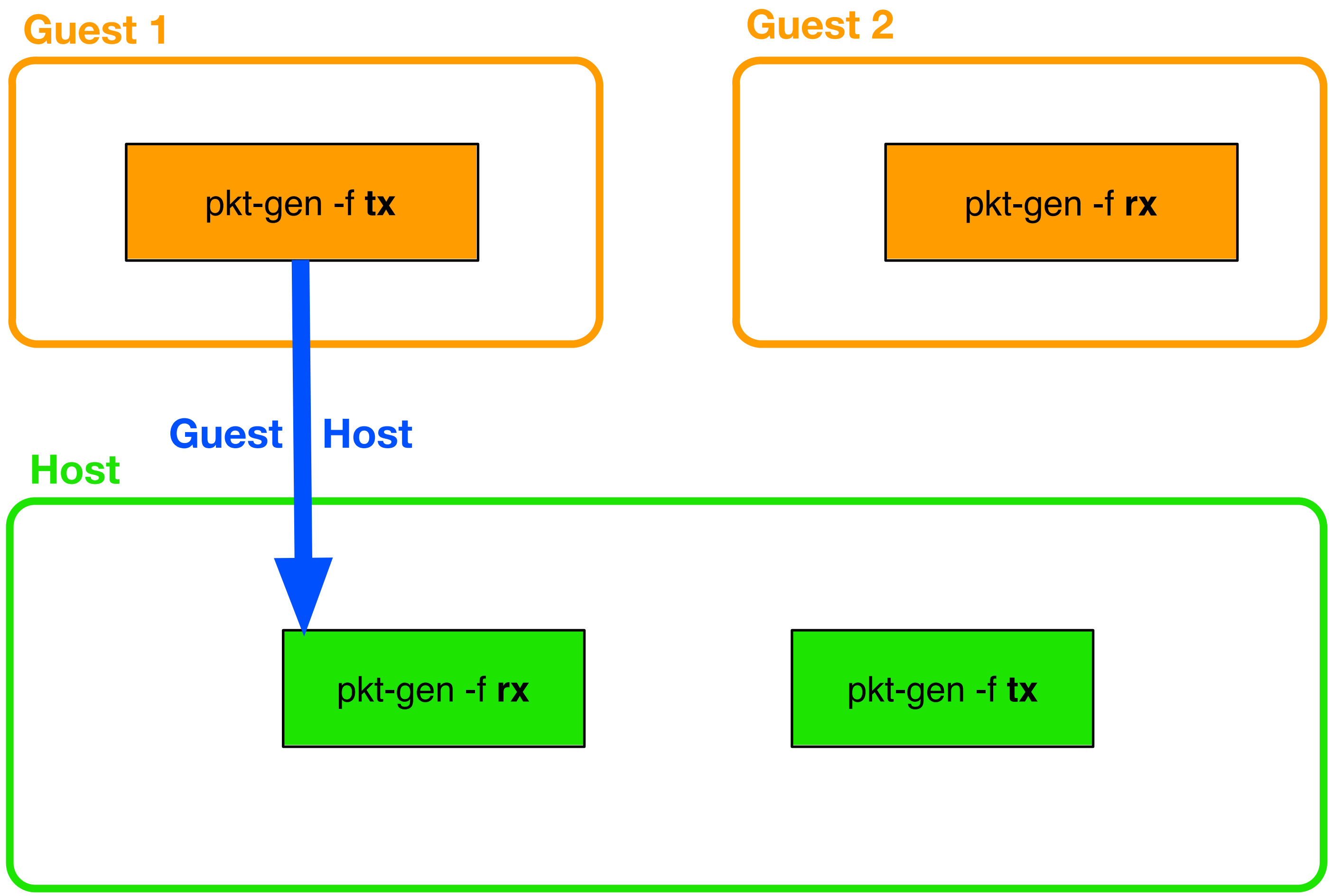
Performance

- **Configuration**



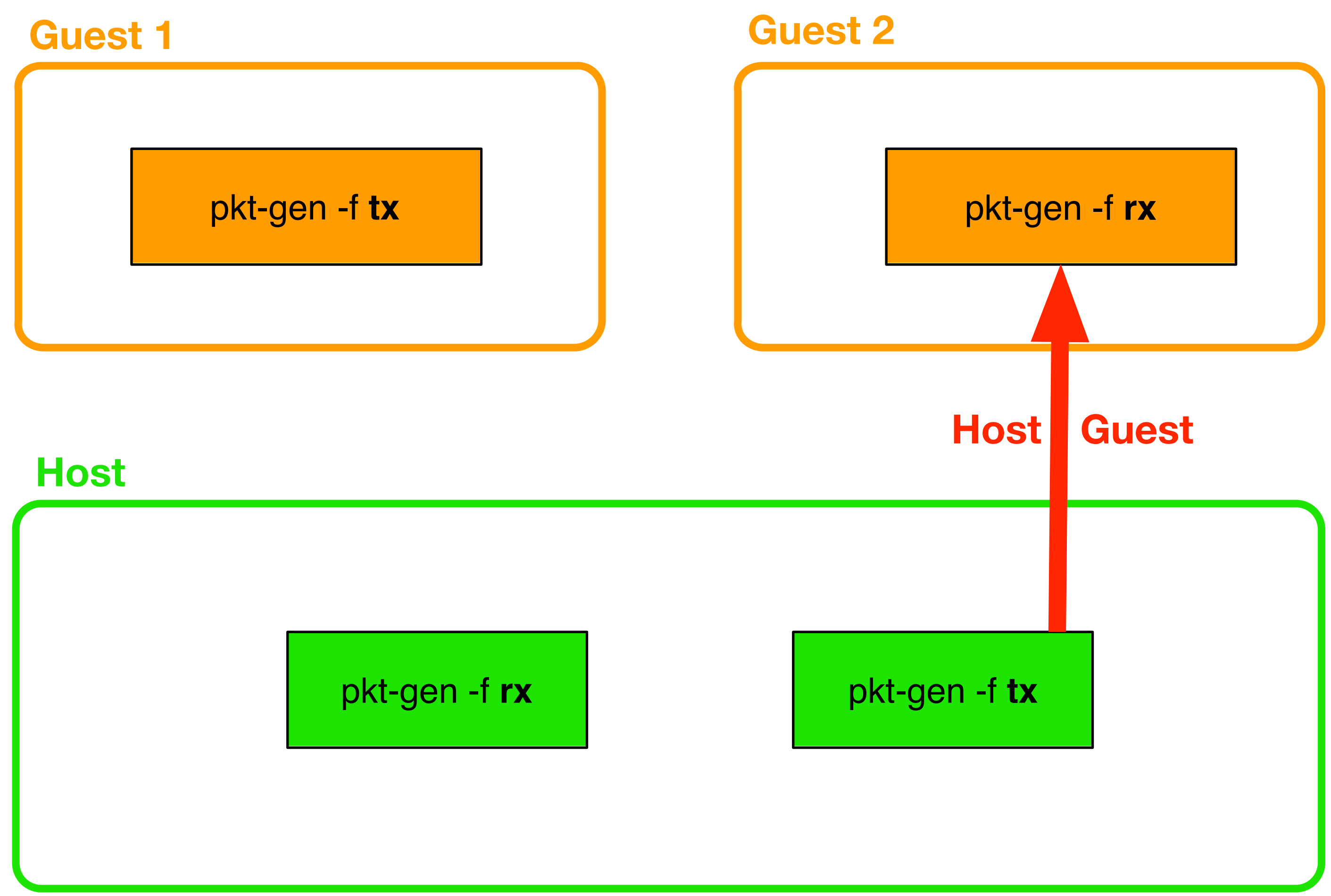
Performance

- **Configuration**
  - **Guest - Host**



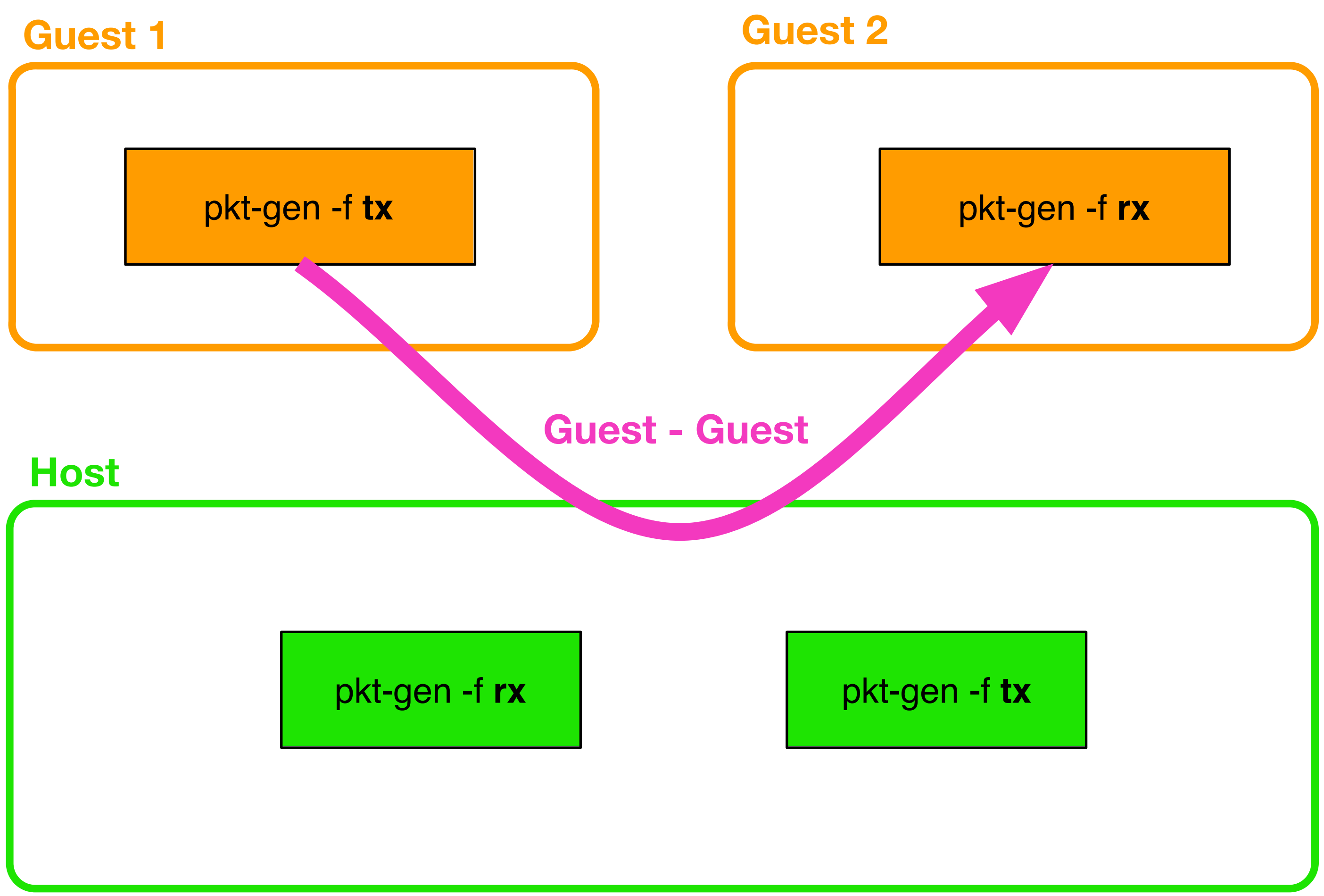
Performance

- **Configuration**
  - **Guest - Host**
  - **Host - Guest**



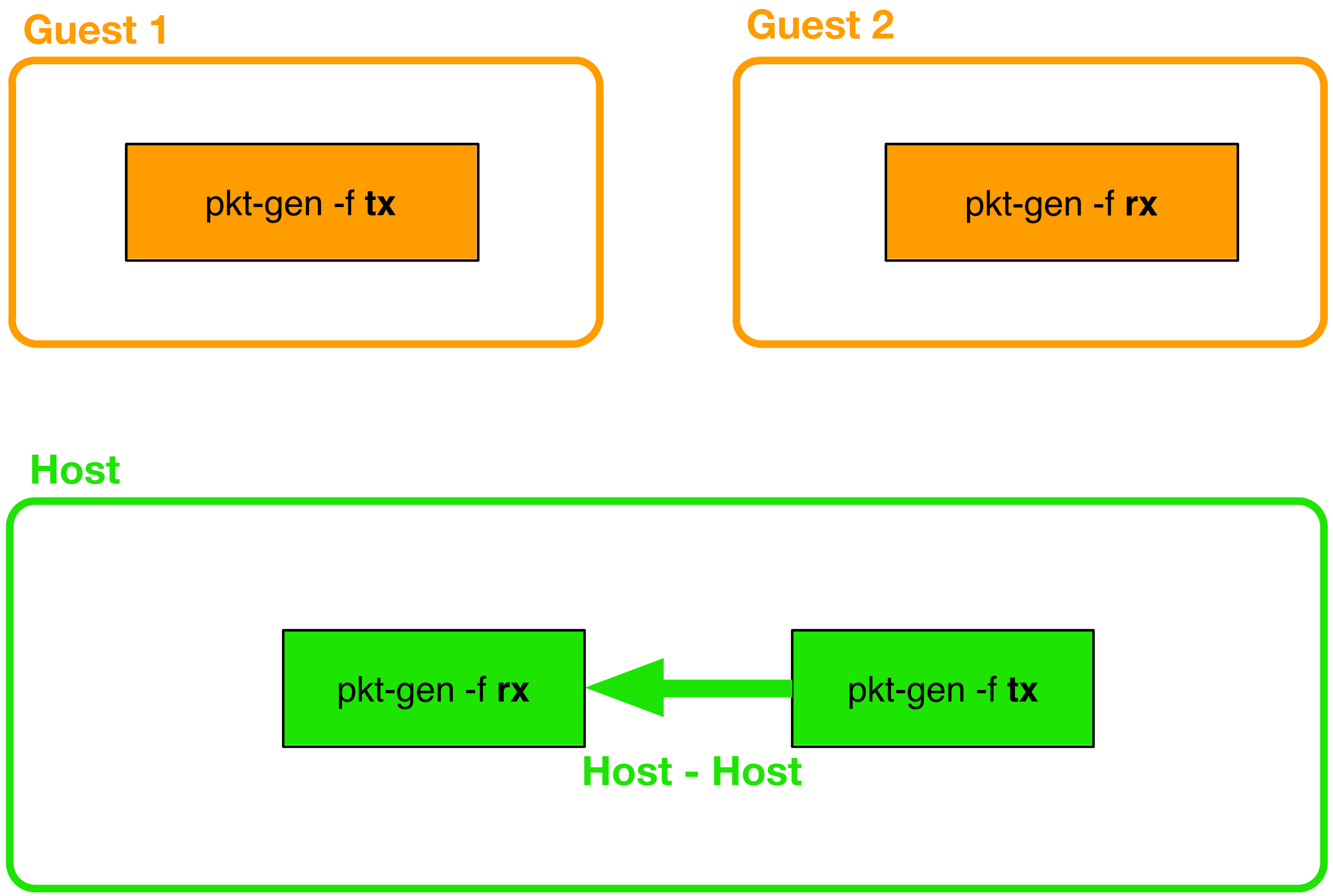
Performance

- **Configuration**
  - **Guest - Host**
  - **Host - Guest**
  - **Guest - Guest**



Performance

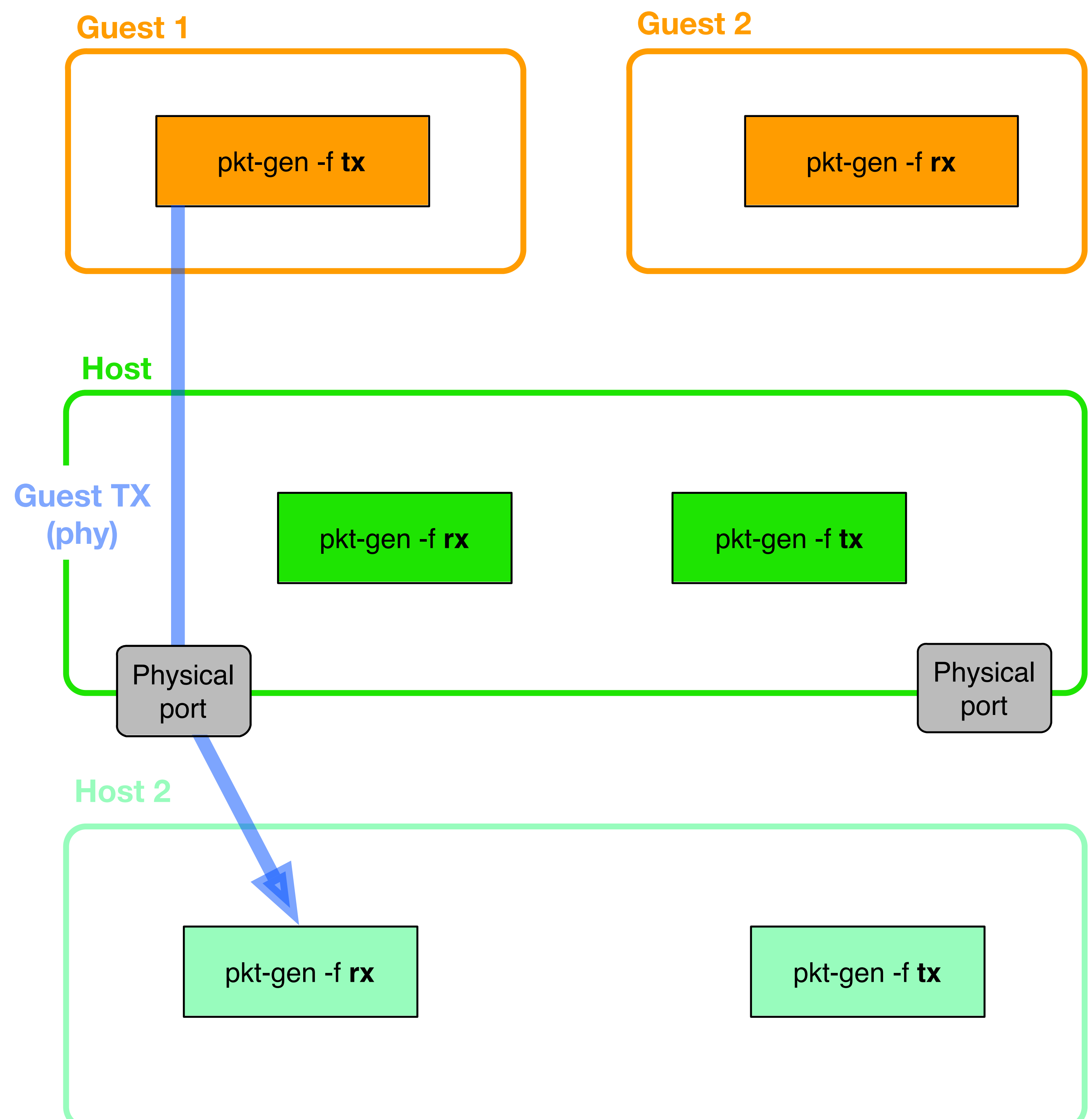
- **Configuration**
  - **Guest - Host**
  - **Host - Guest**
  - **Guest - Guest**
  - **Host - Host**



Performance

- **Configuration**
  - **Guest - Host**
  - **Host - Guest**
  - **Guest - Guest**
  - **Host - Host**

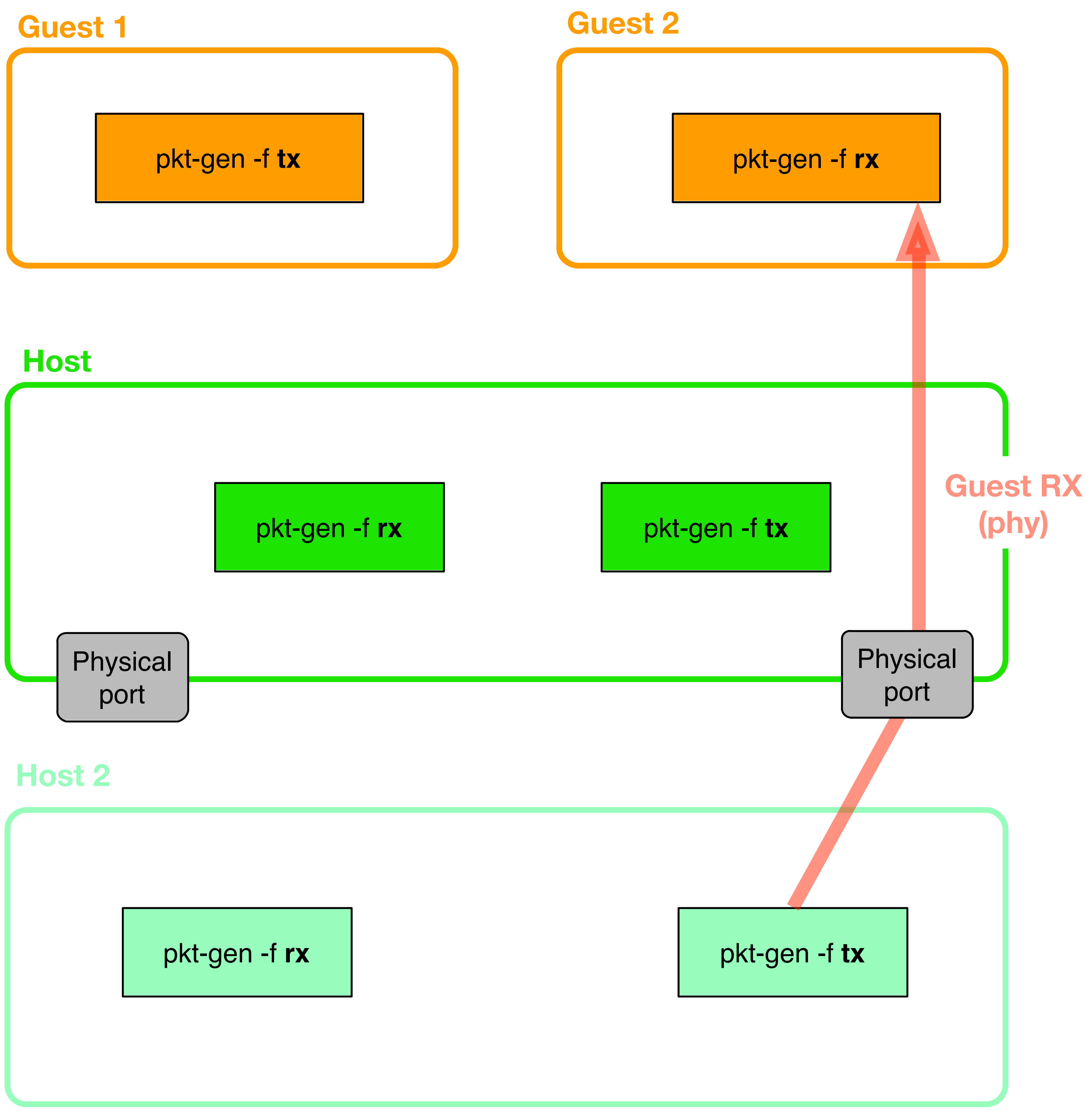
  - **Guest TX over physical port**



Performance

- **Configuration**
  - **Guest - Host**
  - **Host - Guest**
  - **Guest - Guest**
  - **Host - Host**

  - **Guest TX over physical port**
  - **Guest RX over physical port**



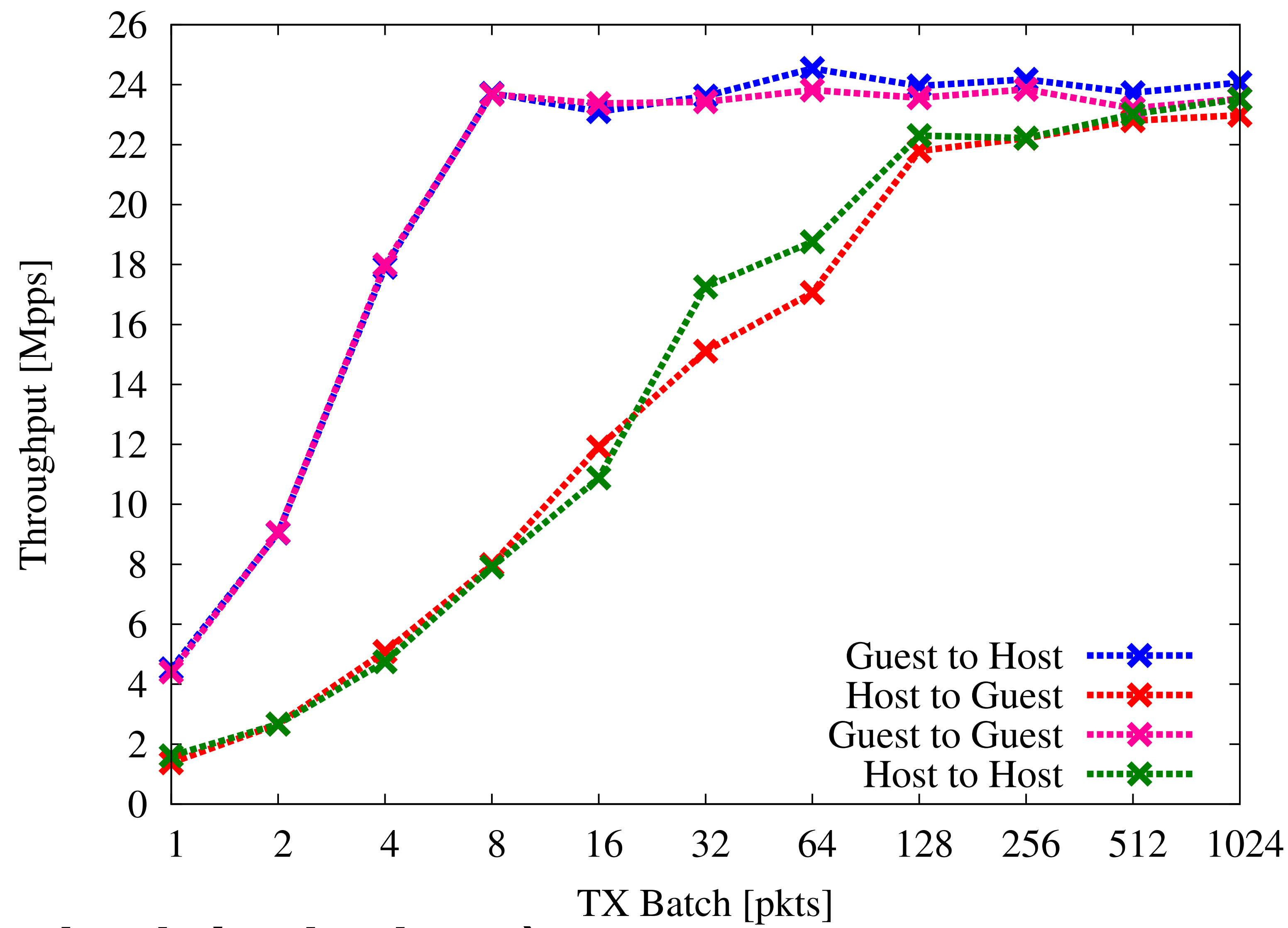
Performance

- **Experimental setup**
  - **CPU: Intel Core i7-3770K at 3.50 GHz**
    - **4 cores / 8 threads**
  - **RAM: 8 GB DDR3 at 1.33 GHz**
  - **NIC: 10Gbps - Intel 82599ES dual-port**
  - **GuestOS: FreeBSD 11.0-CURRENT or Linux 3.12**
  - **HostOS: Linux 3.17 + netmap module + ptnetmap support**
  - **Hypervisor: QEMU-KVM + netmap-backend + ptnetmap**

Throughput

VALE Switch:

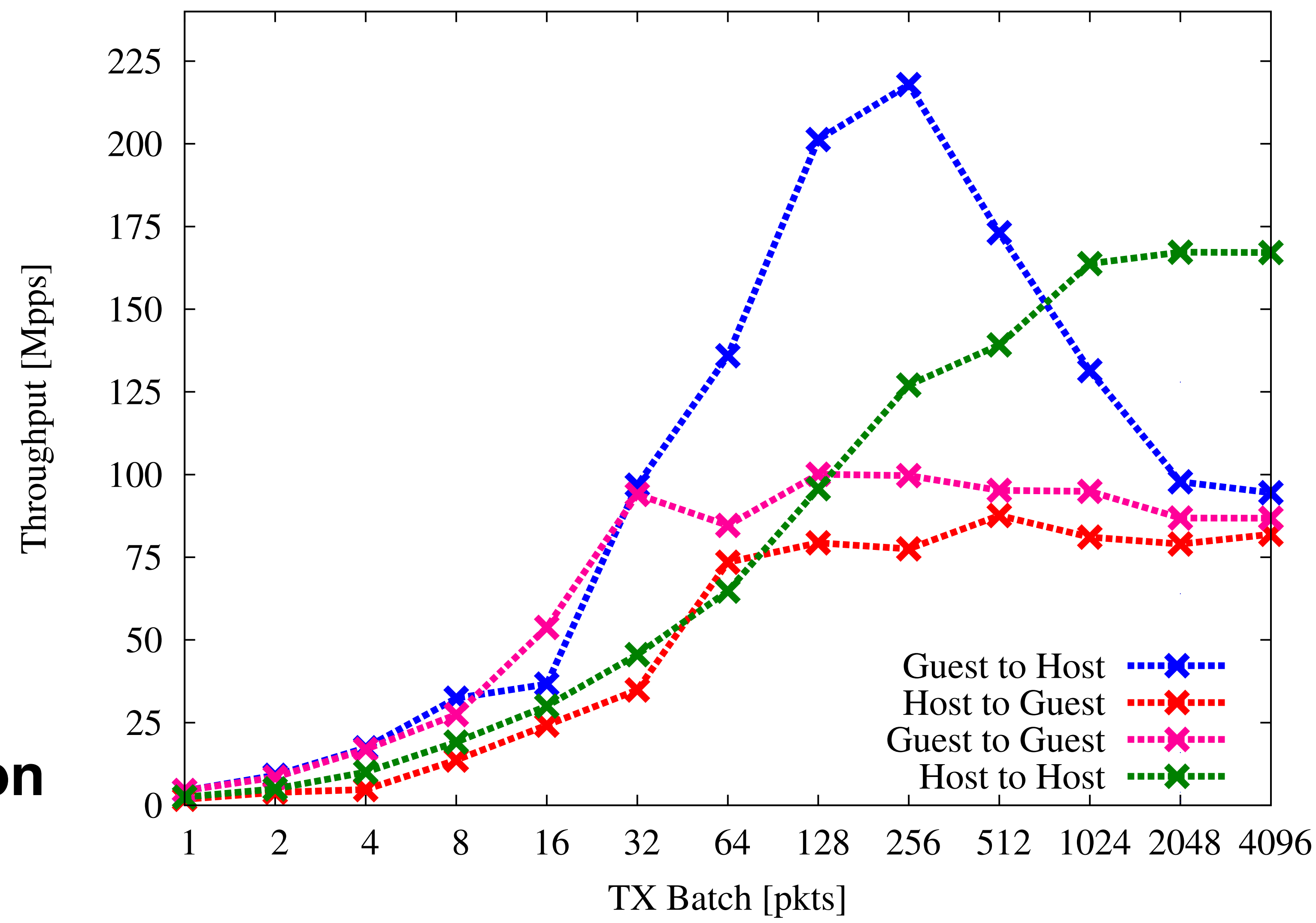
- **ports isolation (data copy)**
- **bottleneck is the thread (on the sender side) that executes the data copy**
- **Host - Host**
  - **reference curve (sender and receiver are both in the host)**
- **Guest - Host**
  - **the sending thread in the guest only has to send notification, while the copy is done by the kernel thread in the host.**



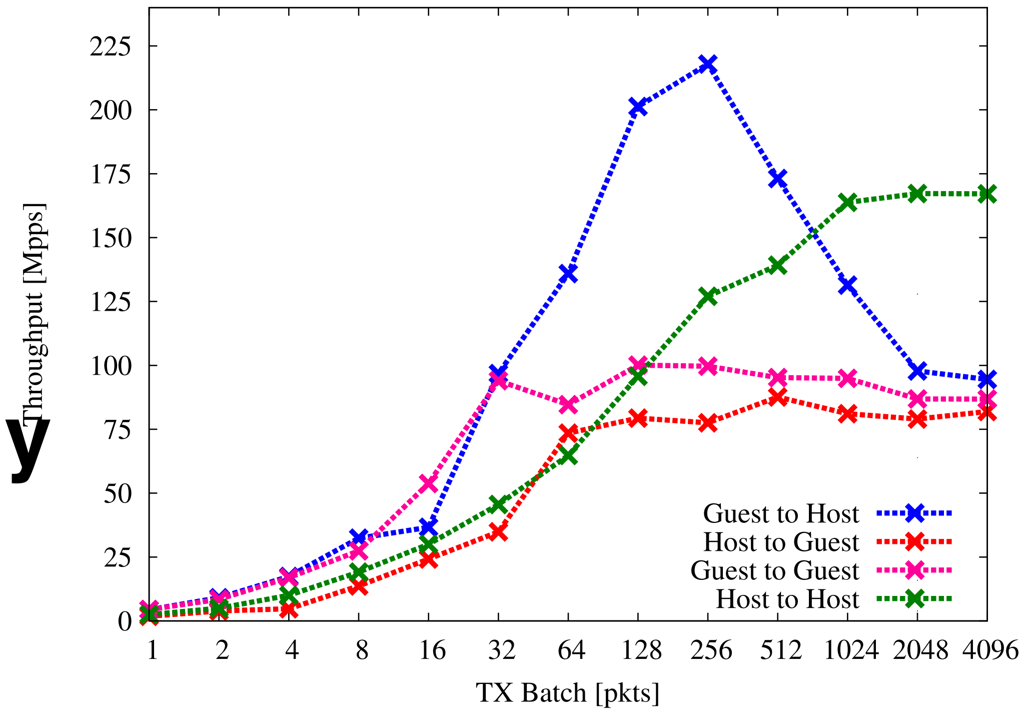
Throughput

netmap pipes:

- **shared memory (zero-copy)**
- **Guest - Host**
  - **similar to VALE ports**
  - **large drop of performance with higher batch sizes is caused by the phenomenon of *Short queue regime***



Short queue regime

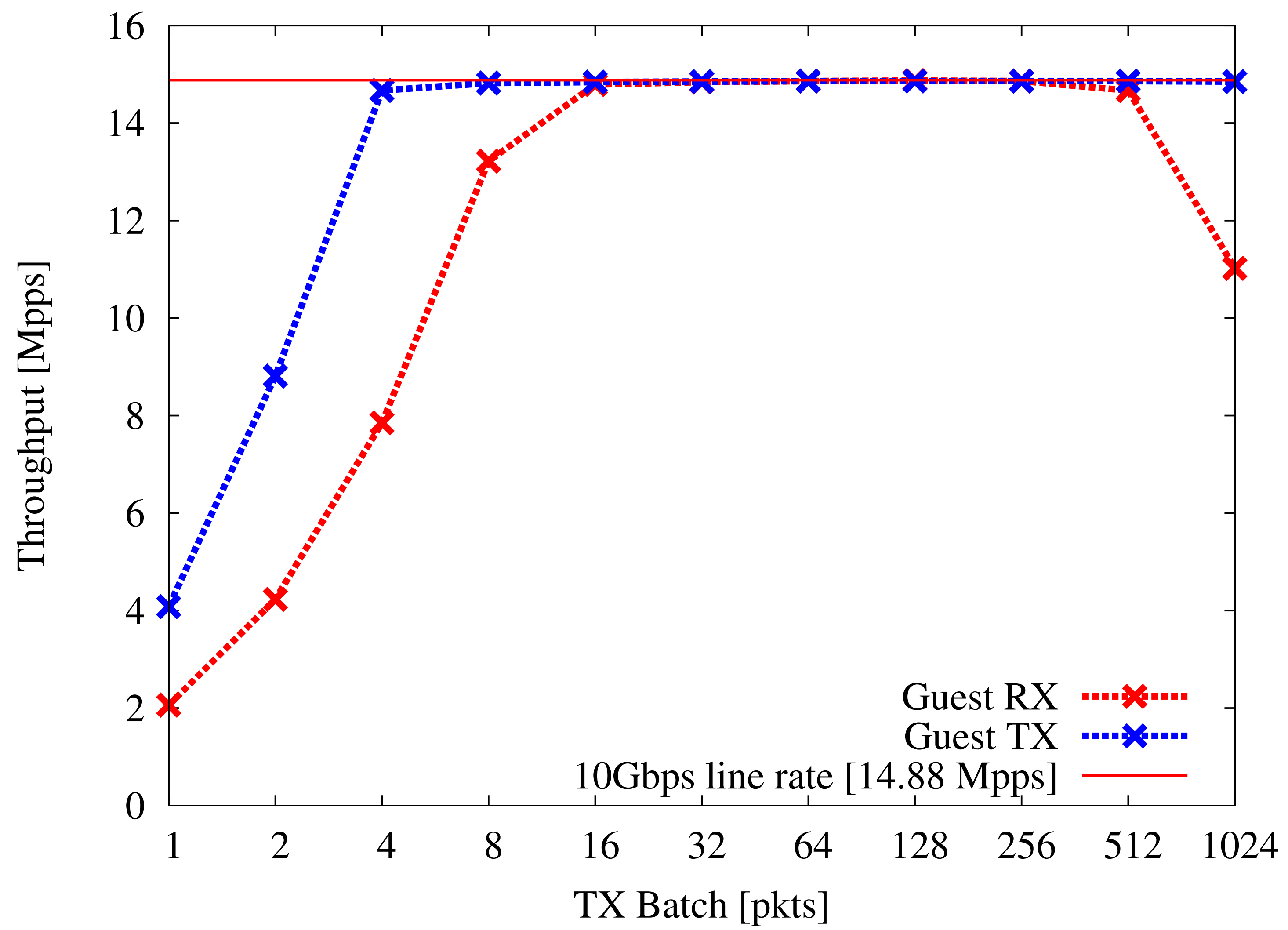


- **when the batch size is small**
  - **bottleneck is the sender and the throughput grows proportionally**
- **above a threshold**
  - **the free space in the queue becomes small and the sender will periodically block**
  - **the blocking in turn causes additional load also on the receiver**
    - **in addition to read the packets, must also wake up the transmitter thread.**

Throughput

physical ports

- **10Gbps NIC**
- **the guest able to reach full line rate even with modest batch sizes**
- **This result matches the performance of other passthrough solutions and netmap on bare metal**



Latency

We measure the latency in two cases:

- **blocking case**
  - **all components waiting an event:**
    - `pkt-gen` **uses** `poll()`
    - **the kernel threads and the guest are sleeping, waiting for a notification or an interrupt**
  - **Host - Host: 5.2 µs**
  - **Guest - Guest: 25 µs**
    - **latency dominated by the cost of two VM exits and two interrupts.**

Latency

- active case
  - we simulate the active state:
    - `pkt-gen` is modified to use non-blocking `ioctl()` to send and receive
    - the kernel threads are forced in an active state, thus avoiding all interrupts, VM exits, and process wakeups
  - Host - Host: 1.2 μs
  - Guest - Guest: 2.1 μs

  - These results have been obtained using VALE switch and are marginally higher than pipes, because has an extra copy and additional locking in the path.

Conclusions and Future Works

- **VMs with ptnetmap support:**
  - **can saturate a 10Gbps link at 14.88 Mpps**
  - **talk at over 20 Mpps to untrusted VMs**
  - **over 75 Mpps to trusted VMs**
- **ptnetmap is implemented as an extension of the netmap framework and it will be publicly available.**
  - **http://info.iet.unipi.it/~luigi/netmap/**
- **We now support FreeBSD and Linux guests and KVM host.**
- **We want to implement the same functionality in bhyve.**

Thank you!

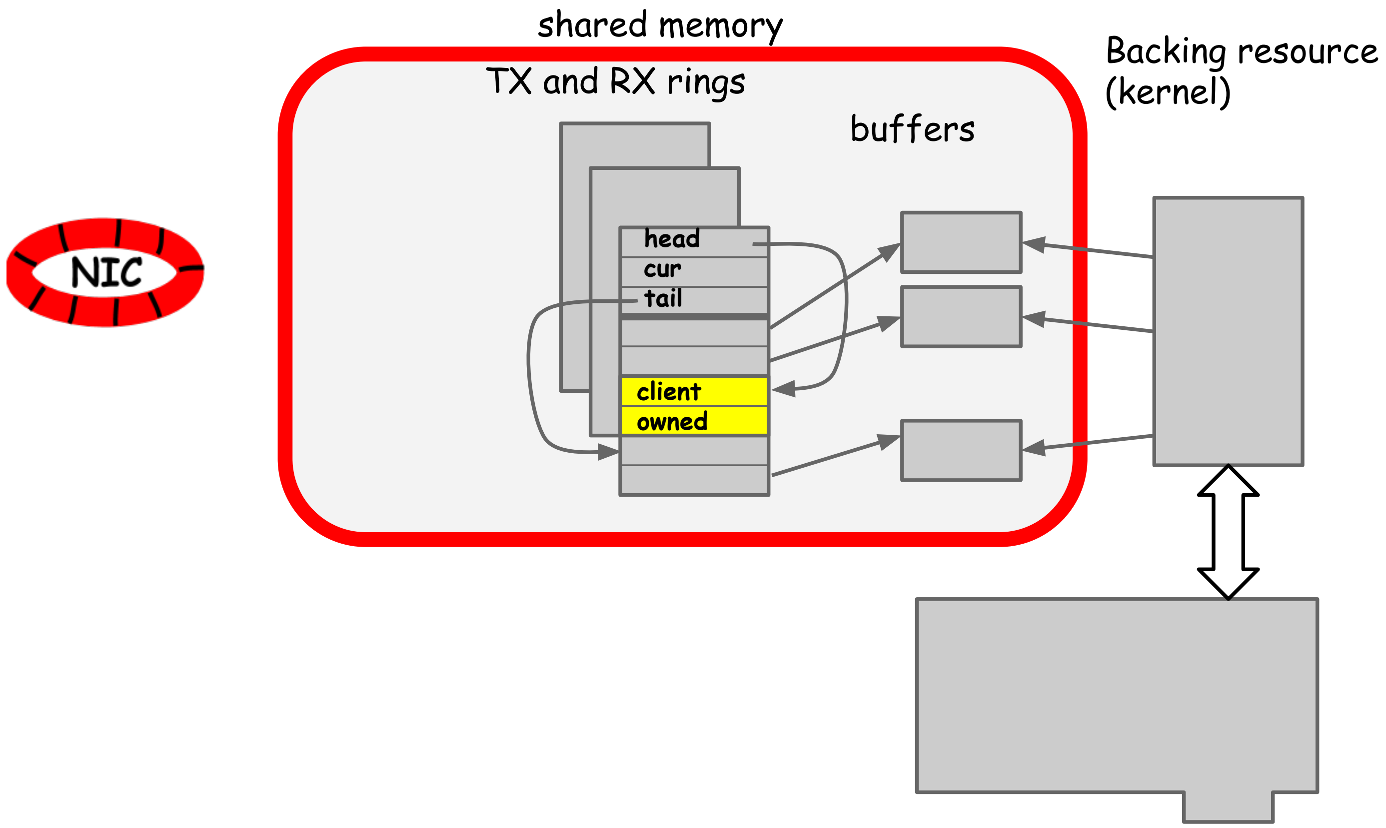
Useful links:

- **http://info.iet.unipi.it/~luigi/netmap/**
- **https://code.google.com/p/netmap/**

Luigi Rizzo rizzo@iet.unipi.it
Giuseppe Lettieri g.lettieri@iet.unipi.it
Stefano Garzarella stefanogarzarella@gmail.com

netmap framework

- **The netmap framework provides high speed network ports to clients**
  - **userspace applications**
  - **in-kernel applications**



- Logically, each port is a set of transmit and receive queues and associated packet buffers, which clients `mmap()` to support zero copy I/O.
- Data transfers occur through the queues using `ioctl()` for non-blocking I/O, and `select()`, `poll()`, `kqueue()`, `epoll()` for blocking I/O.
- Each queue (and its buffers) is logically divided in two regions: one owned by the client, the other owned by the kernel. The boundary between the two regions is marked by two pointers, head and tail.

paravirtualized ethernet devices

- **We slightly modify legacy Ethernet device emulation (e1000) and the corresponding guest drivers to make them work like virtio:**
  - **Real HW and emulated TX**
    - **NIC register (TDT) writes used for both:**
      - **updating status (available packets to send)**
      - **notification (status has changed)**
  - **Paravirtualized TX emulation**
    - **Separate the two functions:**
      - **status only updated in shared memory (CSB - Communication Status Block)**
      - **NIC register (TDT) only used for notification**