IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

RDMA: Ultra-fast networking

Xingda Wei, Rong Chen

IPADS, Shanghai Jiao Tong University

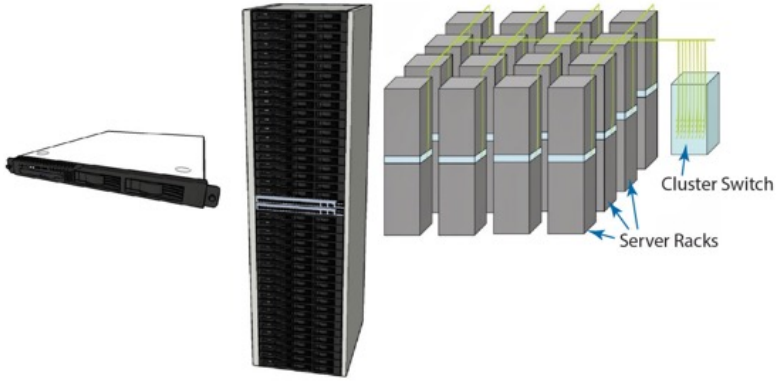
Networking hardware between large-scale website

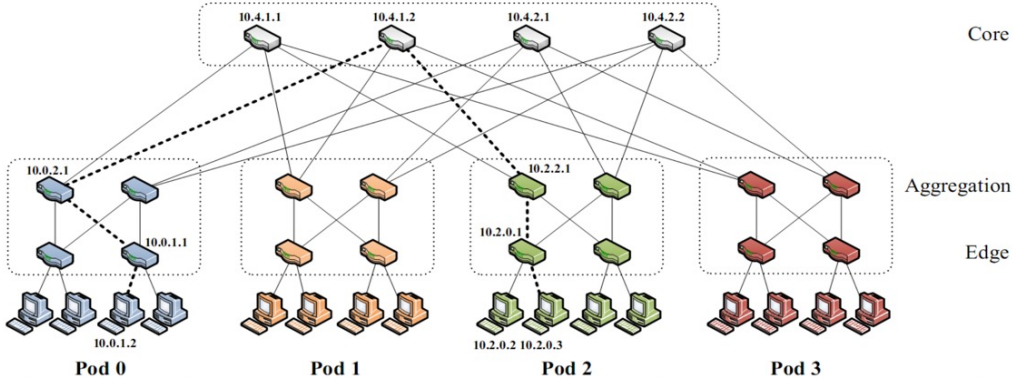
Datacenter: large server and storage farms

- CPUs: > 20,000,000 cores
- Disk capacity: > 1,000,000 GB

Each rack: 40 servers

- Network: 10Gbps – 100Gbps in rack
- 10-100 MW of power





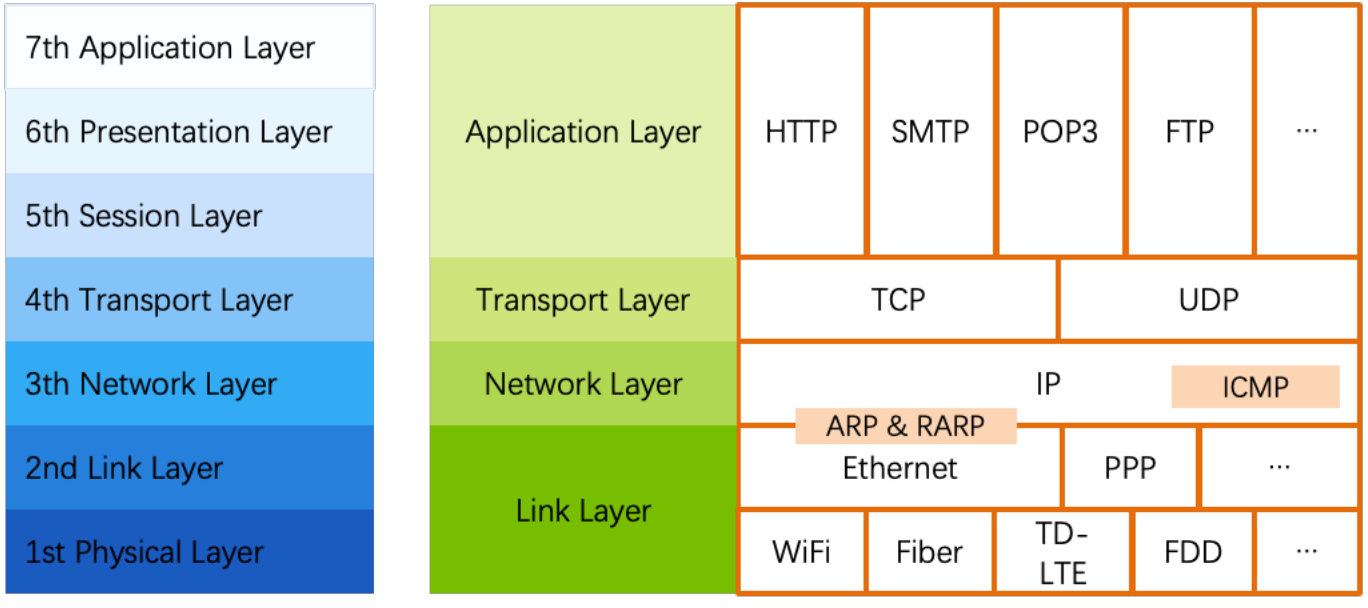
Source: "The Datacenter as a Computer --- An Introduction to the Design of Warehouse-Scale Machines"

Source: https://www.cs.cornell.edu/courses/cs5413/2017sp/lectures/16-datacenter-topologies2.pdf

An overview of traditional networking

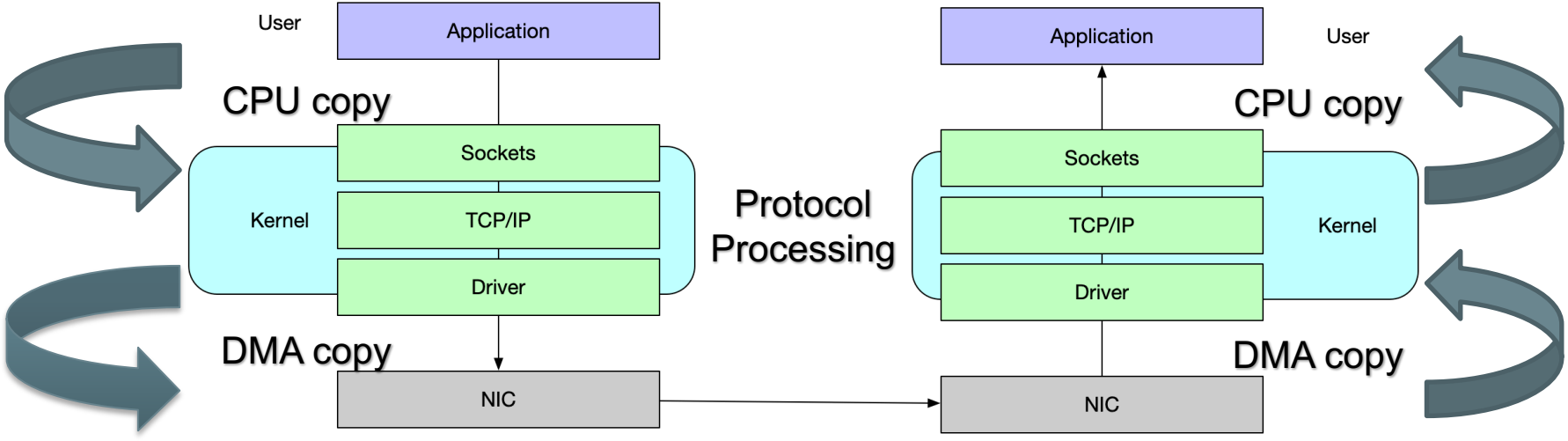
OSI

- The open systems interconnection (OSI) model
- 7-layer architecture



OSI **TCP/IP**

The OS implements the traditional network stack



The OS implements the traditional network stack

Linux provides simple socket interfaces

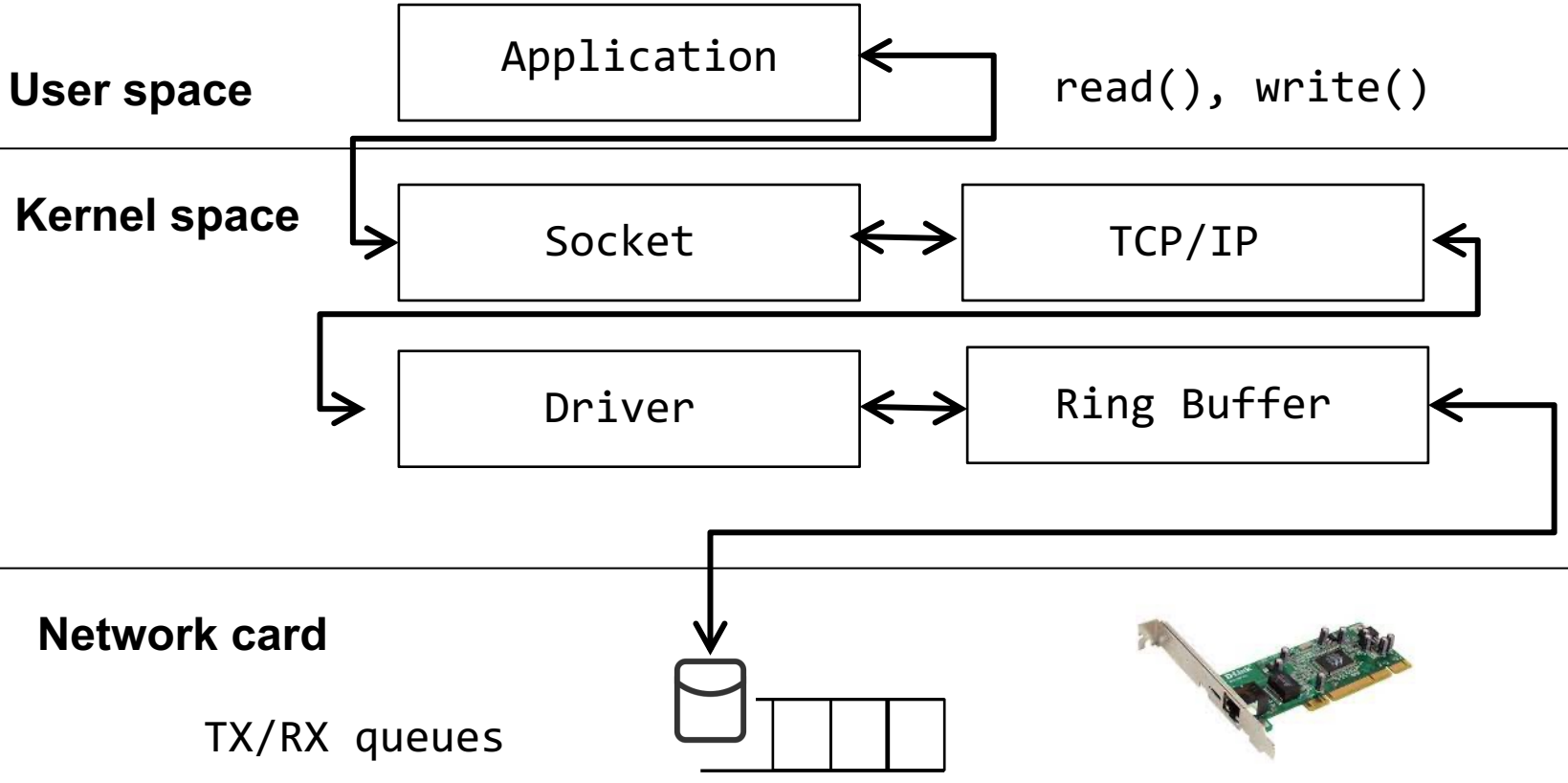
- `socket()`
- `bind()`
- `listen()`
- `connect()`
- `accept()`
- `read()`
- `write()`

Does the software interfaces that simple as it seems?

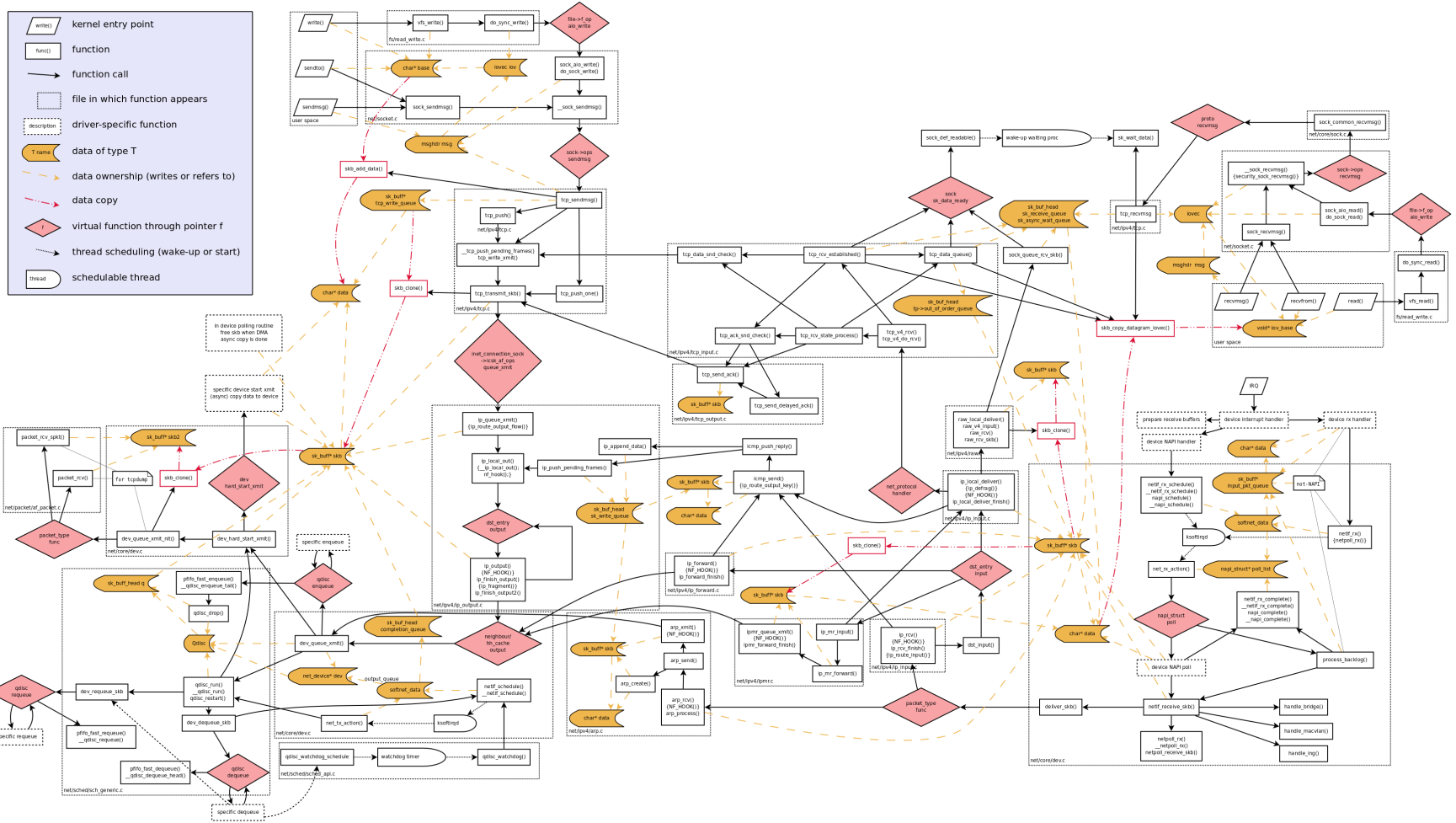
Can be a little complex if want high-performance

- `epoll()`, etc

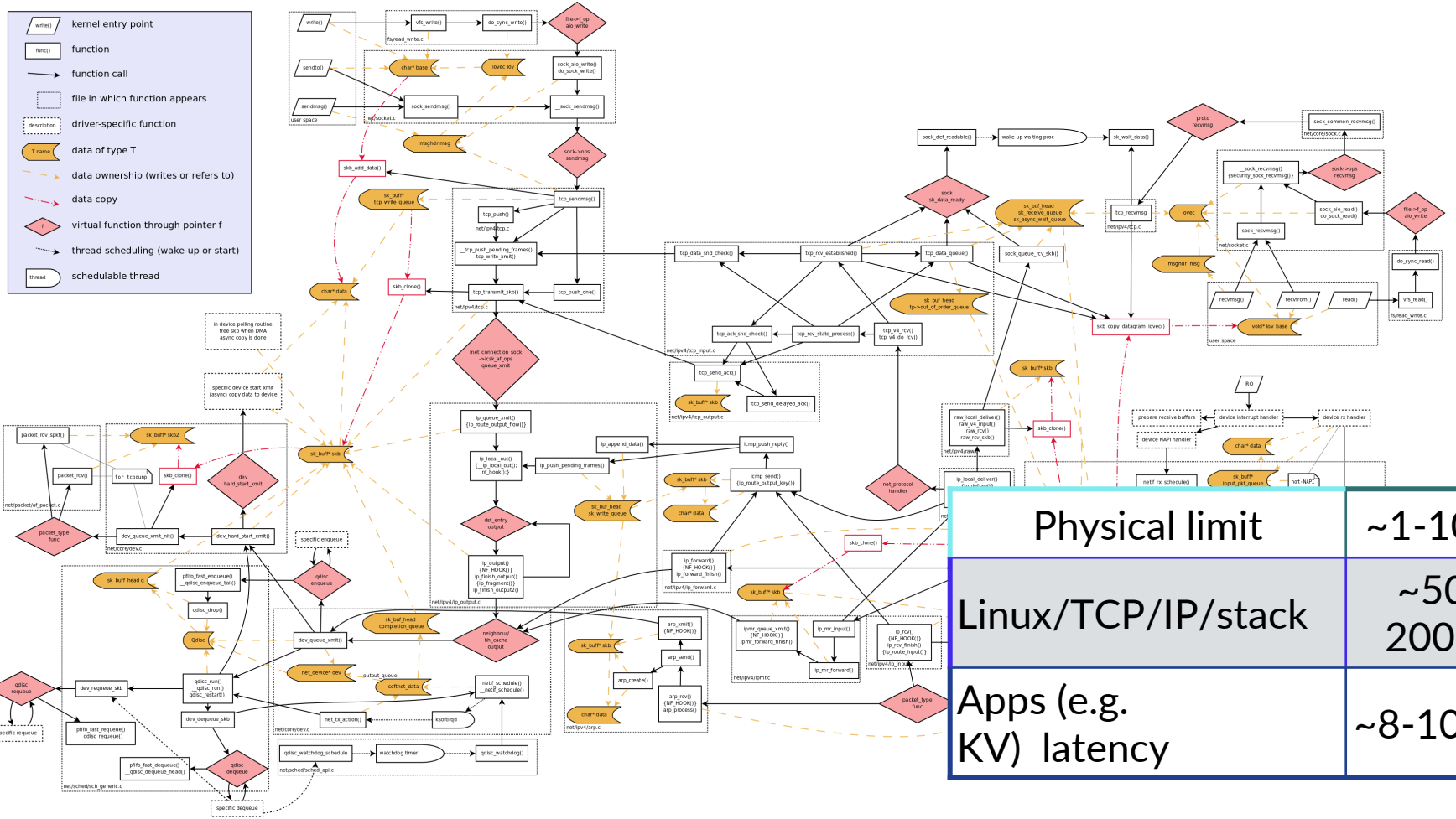
Packet processing in Linux



Linux networking software stack



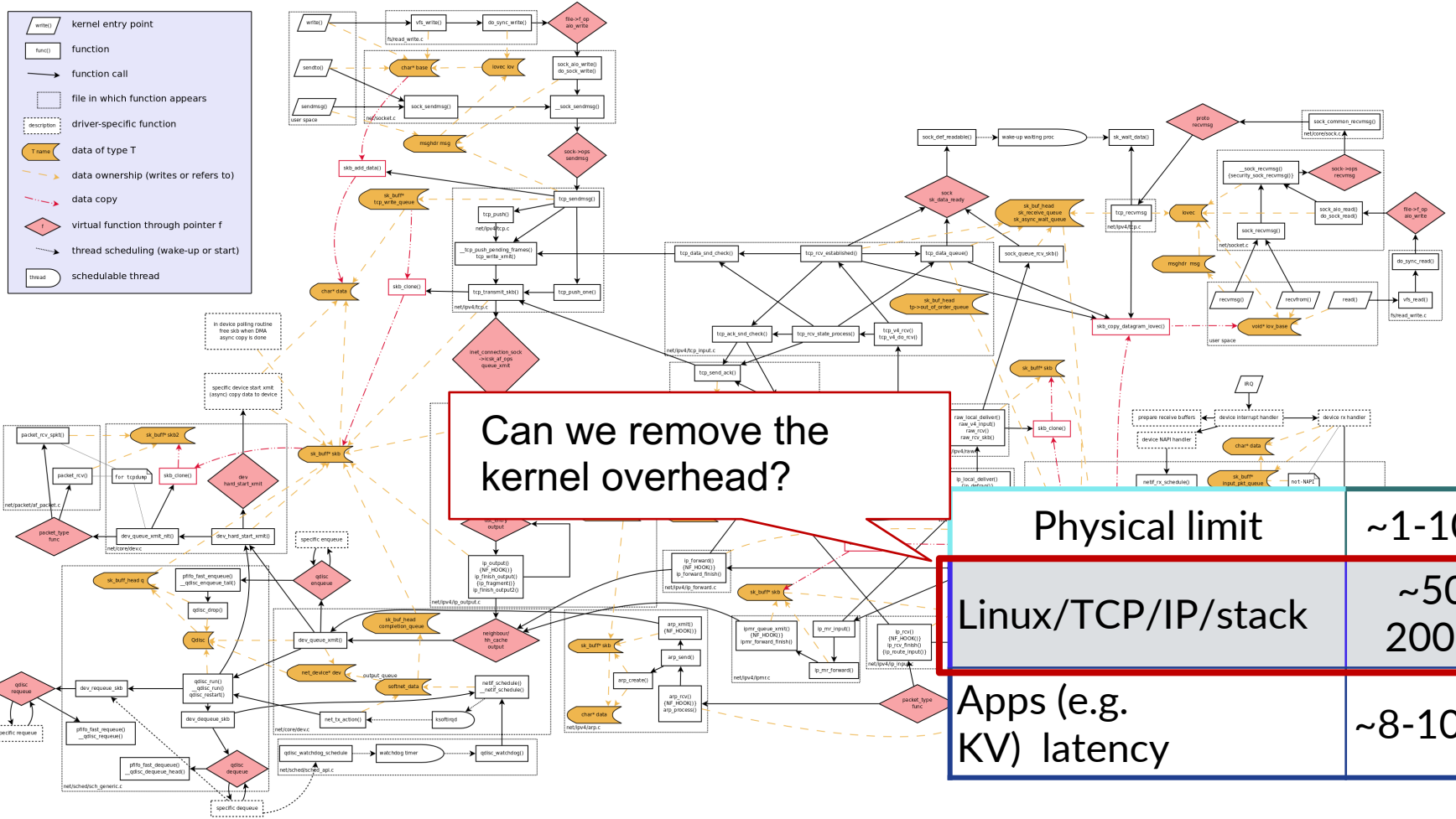
What are the costs of such a complex networking stack?



Physical limit	~1-10us
Linux/TCP/IP/stack	~50-200us
Apps (e.g. KV) latency	~8-100us

Maybe work for 10Gbps network
But what about 100Gbps?

What are the costs of such a complex networking stack?



Kernel-bypassing network

DPDK

– Set of **user-space** driver library to use the network card

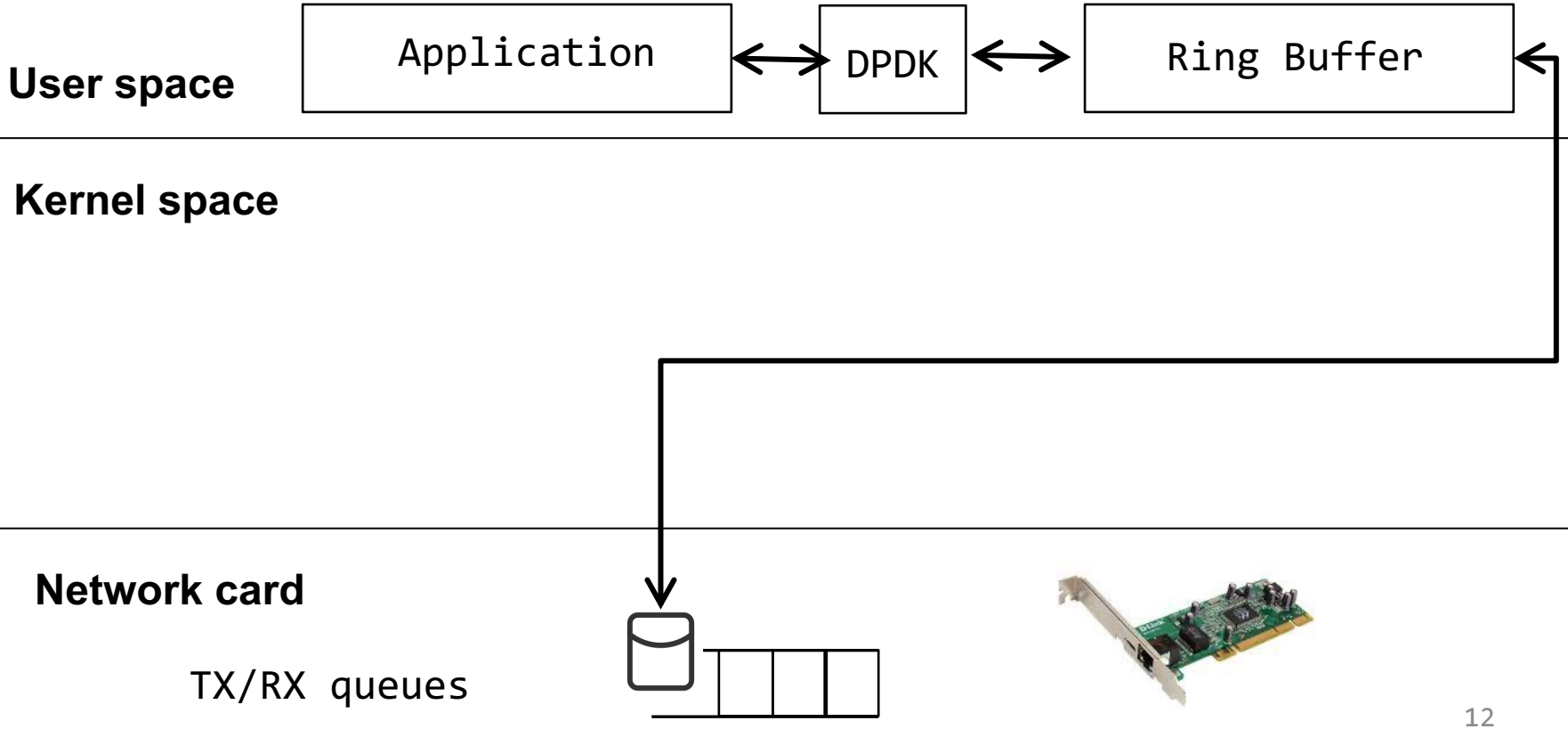
DPDK directly exposes network queues to the user-space

– Forward network packets to/from NIC from/to user applications
    - Without any kernel networking stack!

How to achieve so?

– OS will map the NIC's TX (transmit queue) and RX (receive queue) to the user-land

Packet processing with DPDK



Pros & Cons of DPDK

Pros

1. Bypasses the kernel
2. Reduced `memcpy` overhead
3. User can customize the networking, e.g., not using TCP/IP

    E.g., co-design key-value store w/ DPDK (MICA@SIGCOMM'14)

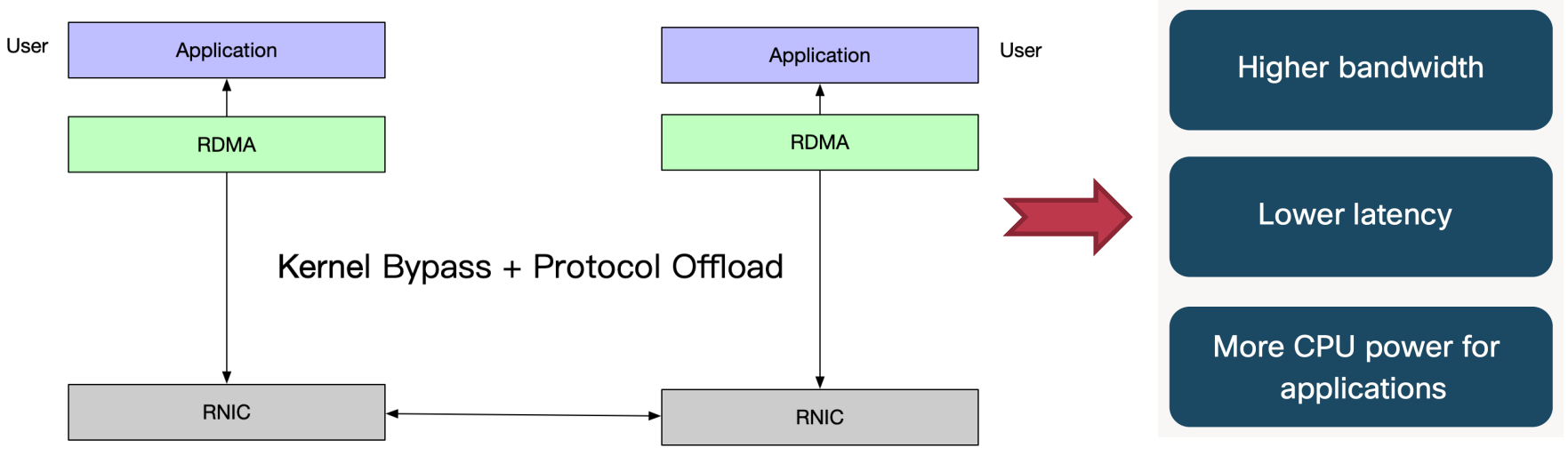
Cons

- Not compatible to traditional networking
- Still need a transport layer at the user applications, e.g., mTCP
    - Which has non-trivial overheads

Can we put the networking stack into the hardware?

RDMA: offload the complete network stack to the NIC

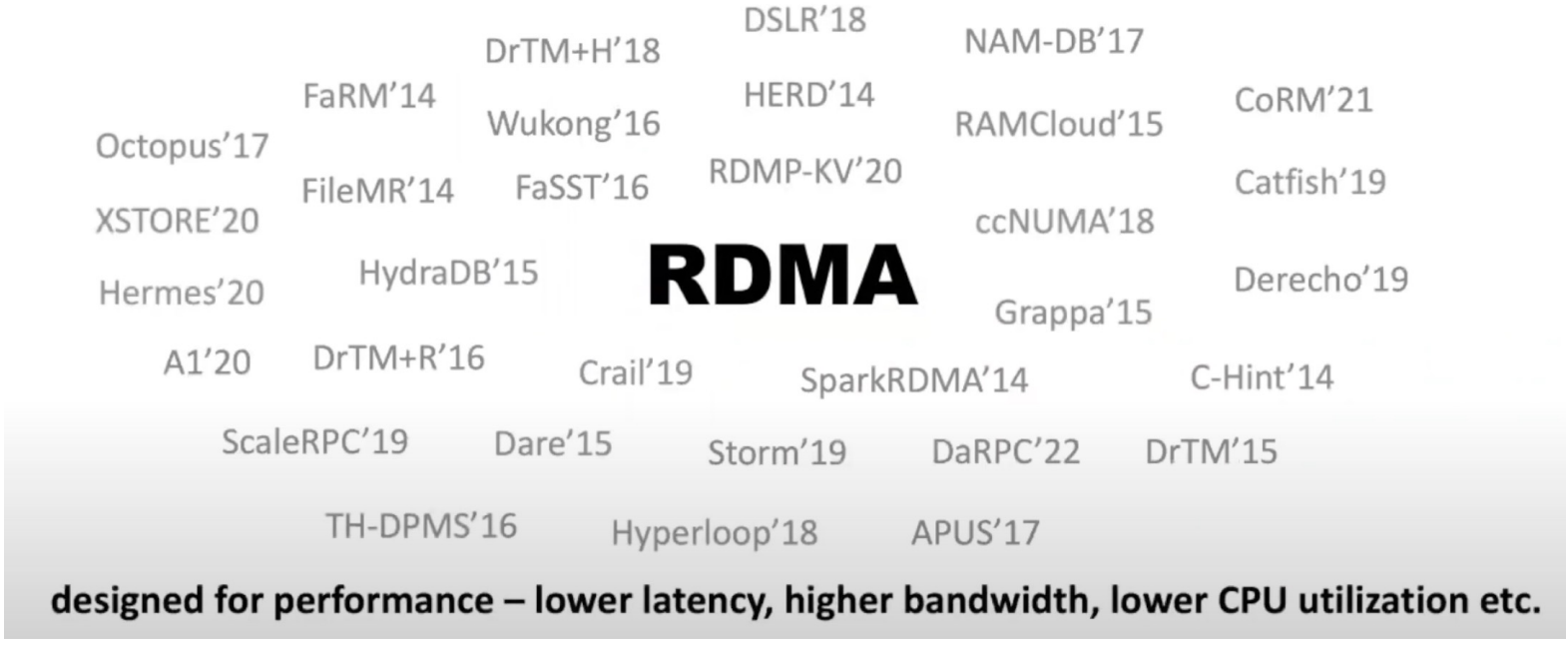
RDMA = Remote **D**irect **M**emory **A**ccess



Direct memory-to-memory data communication over network.

RDMA: offload the complete network stack to the NIC

RDMA is a Trending Topic in Cloud Systems.



RDMA is the default networking for AI

2.2.1 Training Hardware & Carbon Footprint

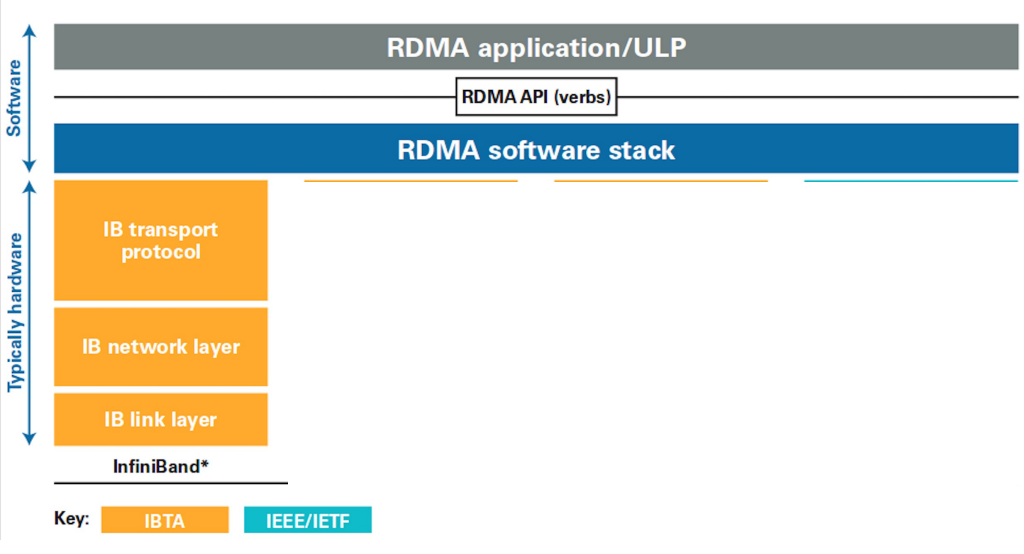
Training Hardware. We pretrained our models on Meta's Research Super Cluster (RSC) (Lee and Sengupta, 2022) as well as internal production clusters. Both clusters use NVIDIA A100s. There are two key differences between the two clusters, with the first being the type of interconnect available: RSC uses NVIDIA Quantum InfiniBand while our production cluster is equipped with a RoCE (RDMA over converged Ethernet) solution based on commodity ethernet Switches. Both of these solutions interconnect 200 Gbps end-points. The second difference is the per-GPU power consumption cap — RSC uses 400W while our production cluster uses 350W. With this two-cluster setup, we were able to compare the suitability of these different types of interconnect for large scale training. RoCE (which is a more affordable, commercial interconnect network)

Source: Llama 2: Open Foundation and Fine-Tuned Chat Models

RDMA implementations: IB

Native infiniband (IB)

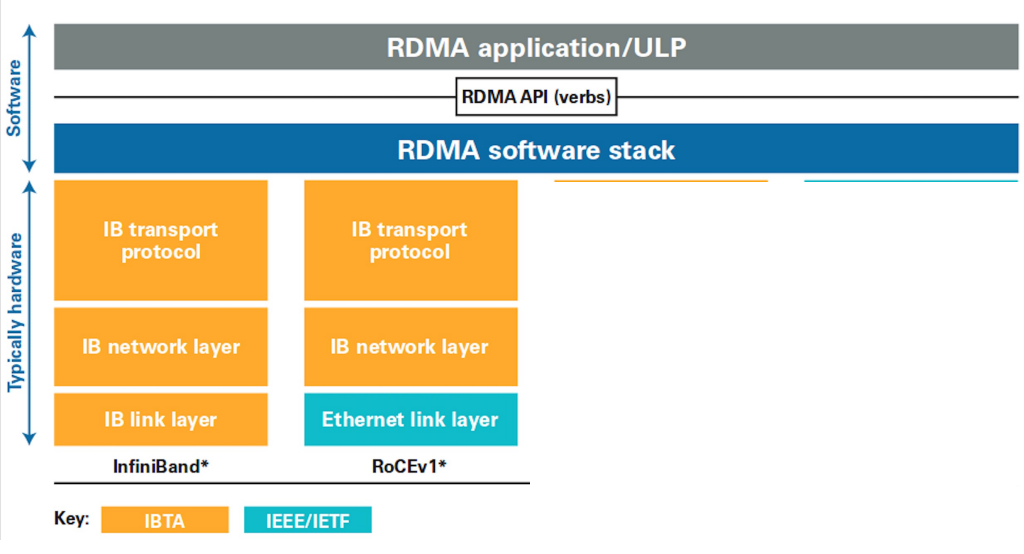
- Complete new protocol stack
- Best Performance
- Need specialized IB NIC and IB Switch (High Cost)



RDMA implementations: RoCE-v1

RDMA over Converged Ethernet Version 1

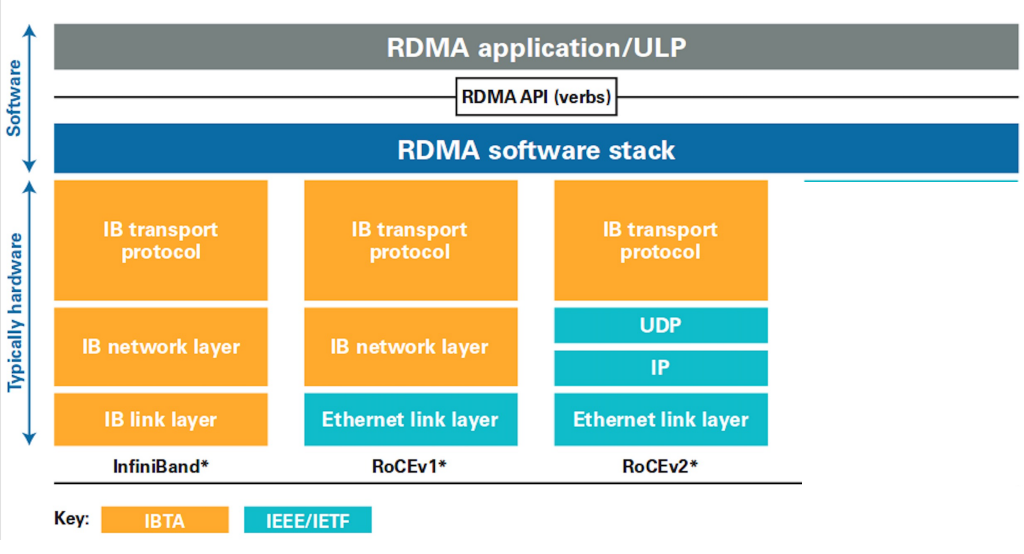
- Based on Ethernet in link layer.
  - Compatible with the current network infrastructure.



RDMA implementations: RoCE-v2

RDMA over Converged Ethernet Version 1

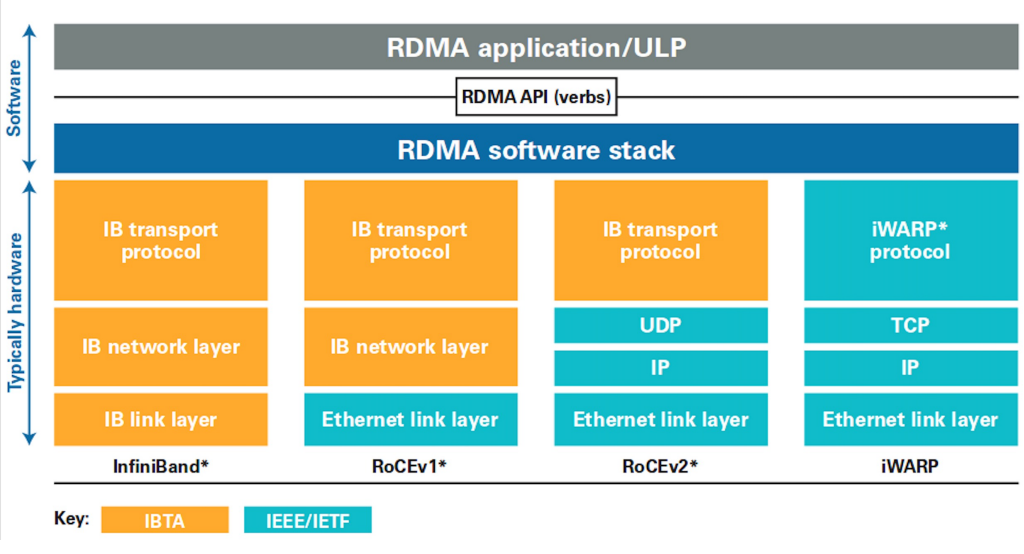
- Carried by IP and UDP in network layer.
- More flexible & scalable than RoCE v1
  - Most cloud datacenters choose RoCEv2 for large-scale deployments.



RDMA implementations: iWrap

An attempt to patch up on existing LAN/WAL network

- Layered on existing TCP/IP protocols
- In contrast, RoCE is a purpose-built RDMA transport protocol for Ethernet



Programming w/ RDMA (verbs)

It is quite different from socket

High-level concepts of RDMA

Queue Pairs (Send Queue, Complete Queue and Receive Queue)

- SQ, CQ & RQ
- A group of queues form an RDMA connection
    - Analog to connections in TCP/IP

User posts networking requests to the SQ

- Two-sided: message datagram (like `send/recv`)
- One-sided: remote read/write (will be talked about later)

Get the completion events of the send requests from the CQ

Receive two-sided messages from the RQ

Queue Pairs have different models

Reliable connected (RC)

- Like TCP/IP socket, provides reliable communications, supports all RDMA requests

Unreliable datagram (UD)

- Like UDP socket
- No reliability guarantees, limited RDMA requests type supported

Others

- Less commonly used

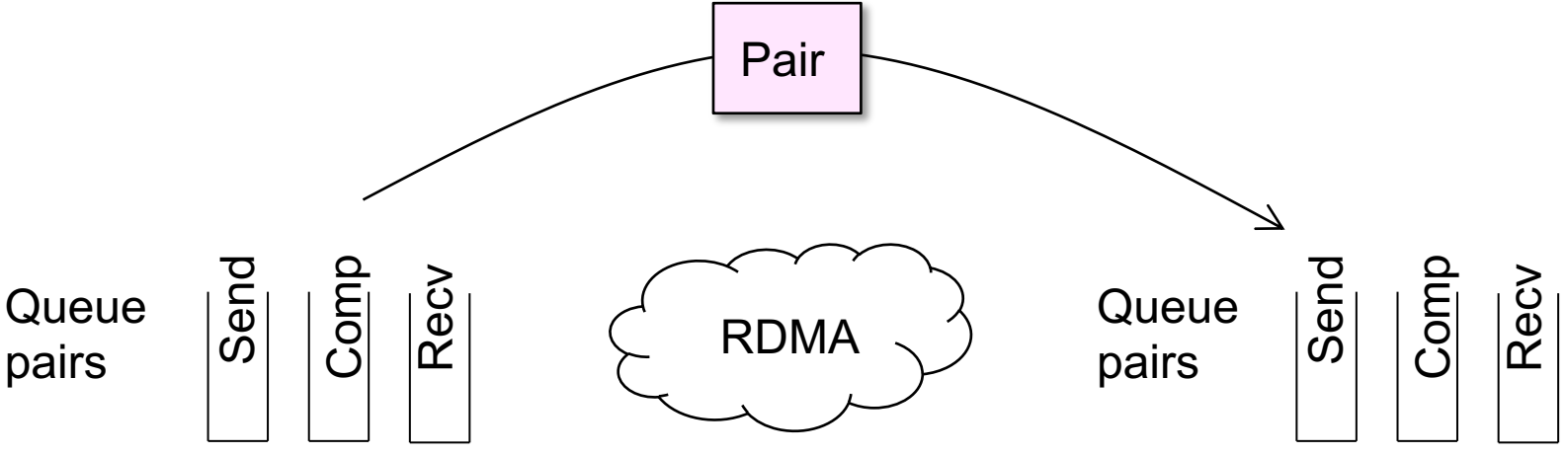
We focus on RC queue pair in this lecture

Reliable connection Queue Pairs (QP)s

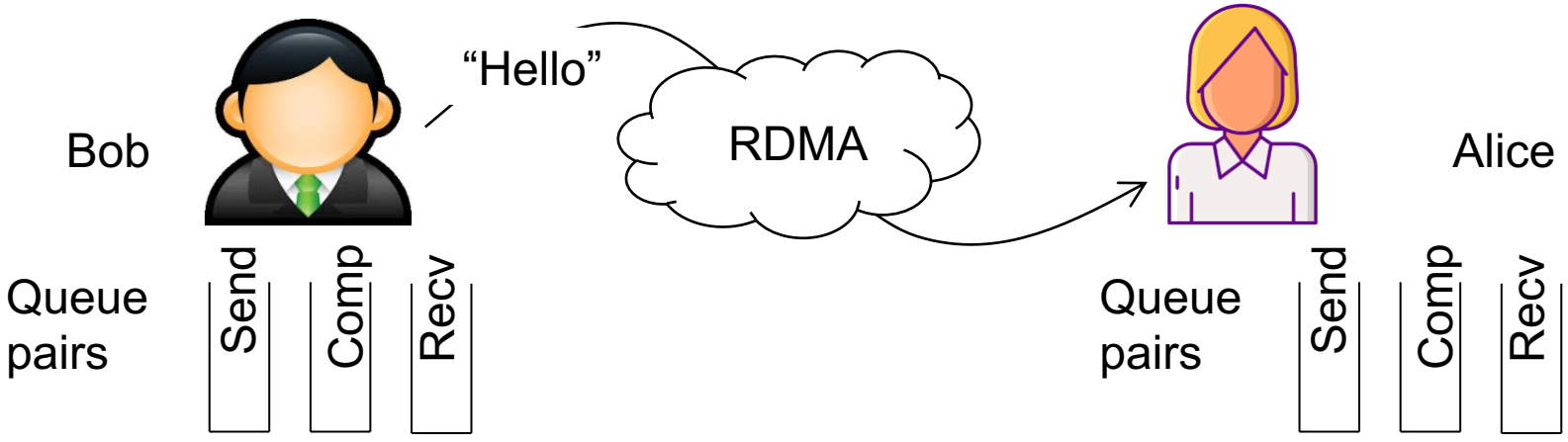
Two QP must first establish a connection via handshake so as to proceed

- Similar to handshakes in TCP/IP
- We will talk about the connection establishment later

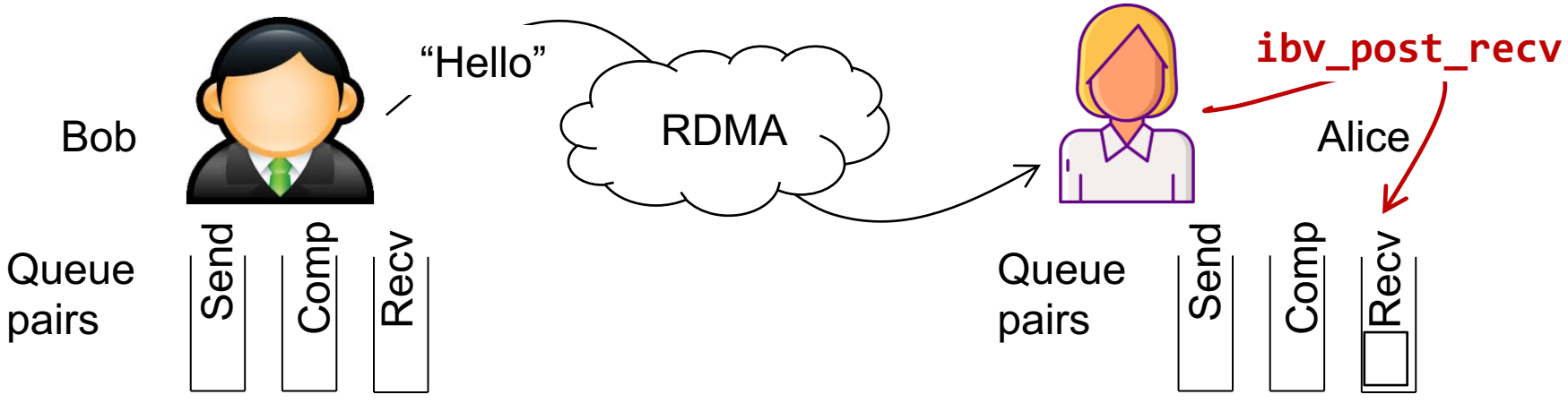
An reliable connection QP must be only connected to one other



Example: send message with RDMA



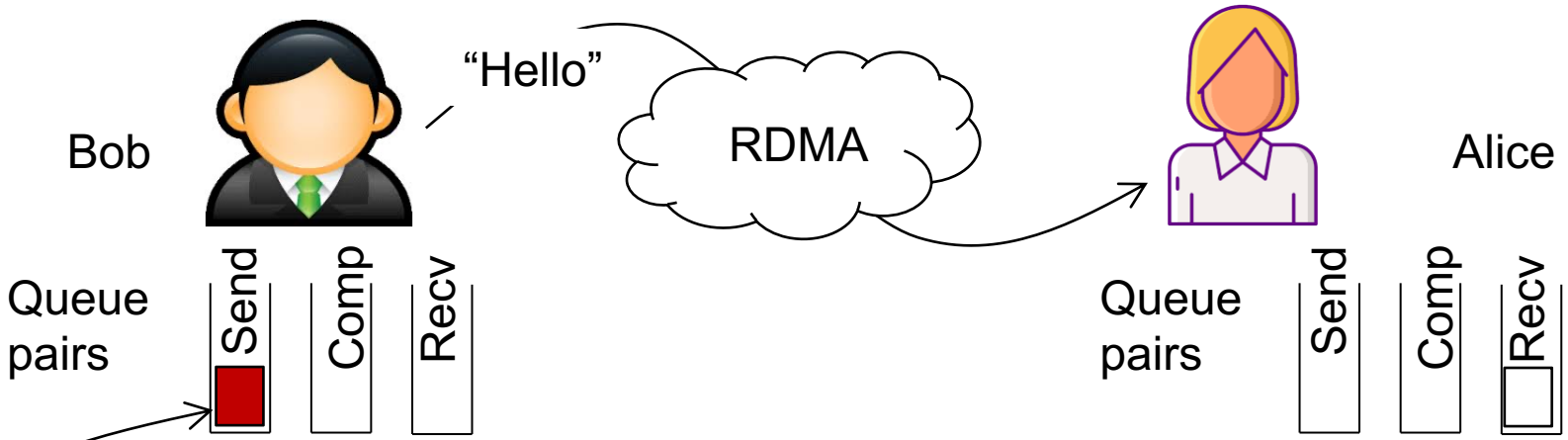
Register a message buffer to receive incoming message



Alice

```
struct ibv_sge list = { .addr = msg_buf, .length=5 };
struct ibv_recv_wr wr = {
    .sg_list = &list,
    ..., // other fields
};
ibv_post_recv(alice_qp, &wr, ...); // post to the receive queue
```

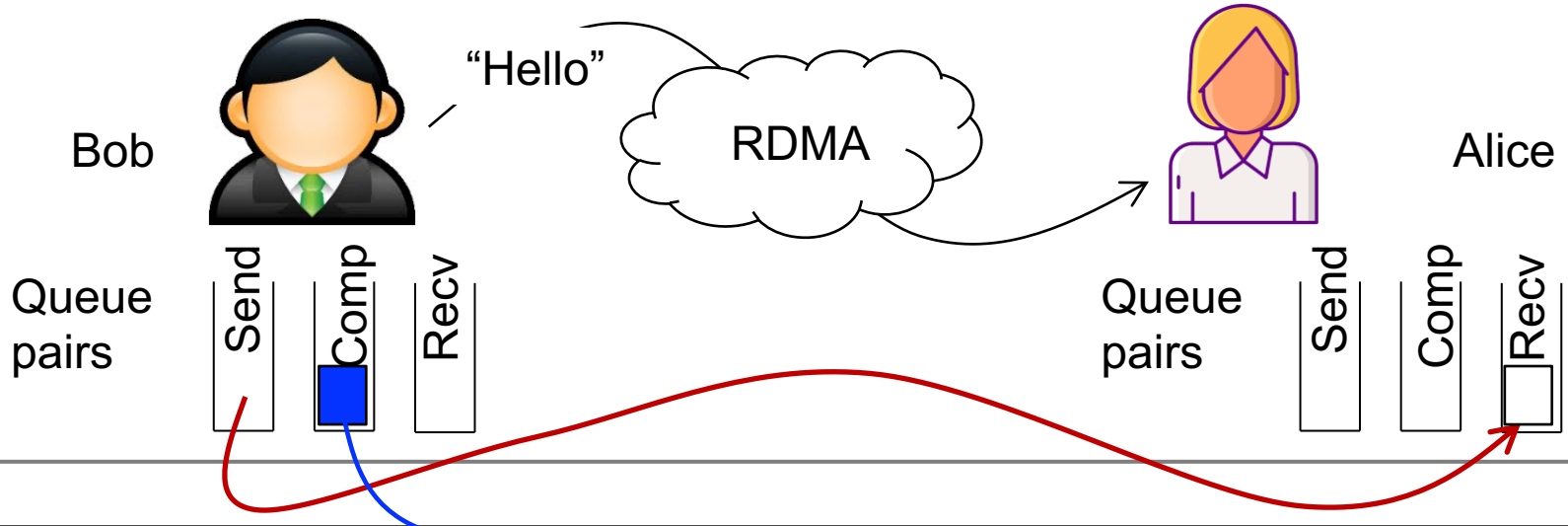
Send the message with post_send



bob

```
struct ibv_sge list = { .addr = (uintptr_t)"Hello", .length=5 };
struct ibv_send_wr wr = {
    .sg_list = &list,
    .op_code = IBV_WR_SEND // send a two-sided message
    ..., // other fields
};
ibv_post_send(bob_qp, &wr, ...); // send queue
```

Poll the completion of the send via polling

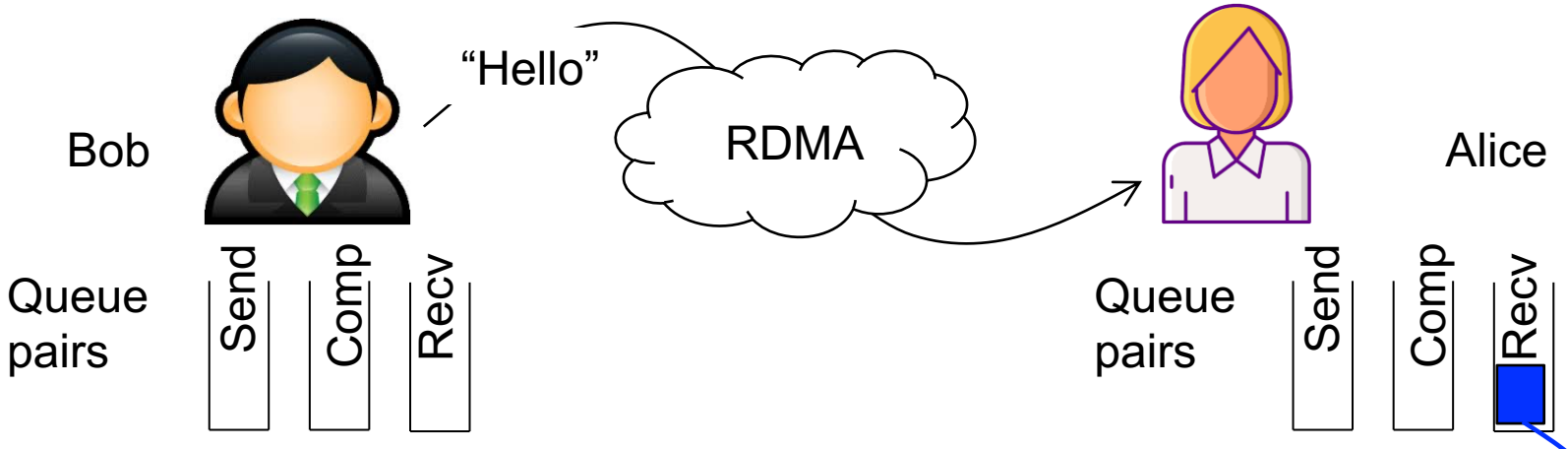


bob

```
do {
    num_completions = ibv_poll_cq(bob_com_qp, ...);
} while (num_completions == 0);

// The request has been successfully sent to the remote
// or an error happens
```

Poll the received message from Alice



bob

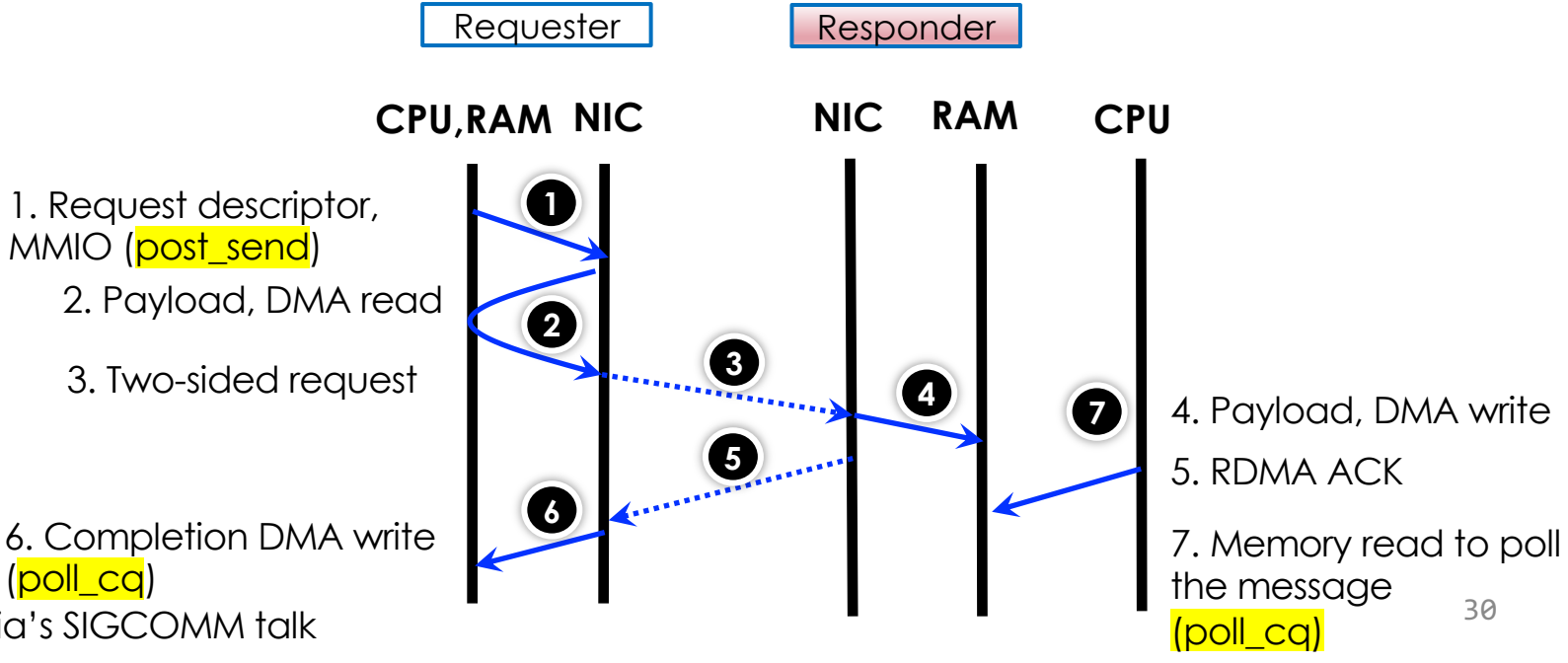
```
do {
    num_completions = ibv_poll_cq(alice_receive_qp, ...);
} while (num_completions == 0);

// The message must be stored in msg_buf if no error
Print(msg_buf) // Hello
```

Lifecycles of the sent message

Secrets behind RDMA: MMIO & DMA

- MMIO: Using LOAD and STORE
- DMA: Device directly read/write the memory (provided by PCIe)



Credit: Anuj Kalia's SIGCOMM talk

RDMA vs. DPDK

Common

- Both are kernel bypassing

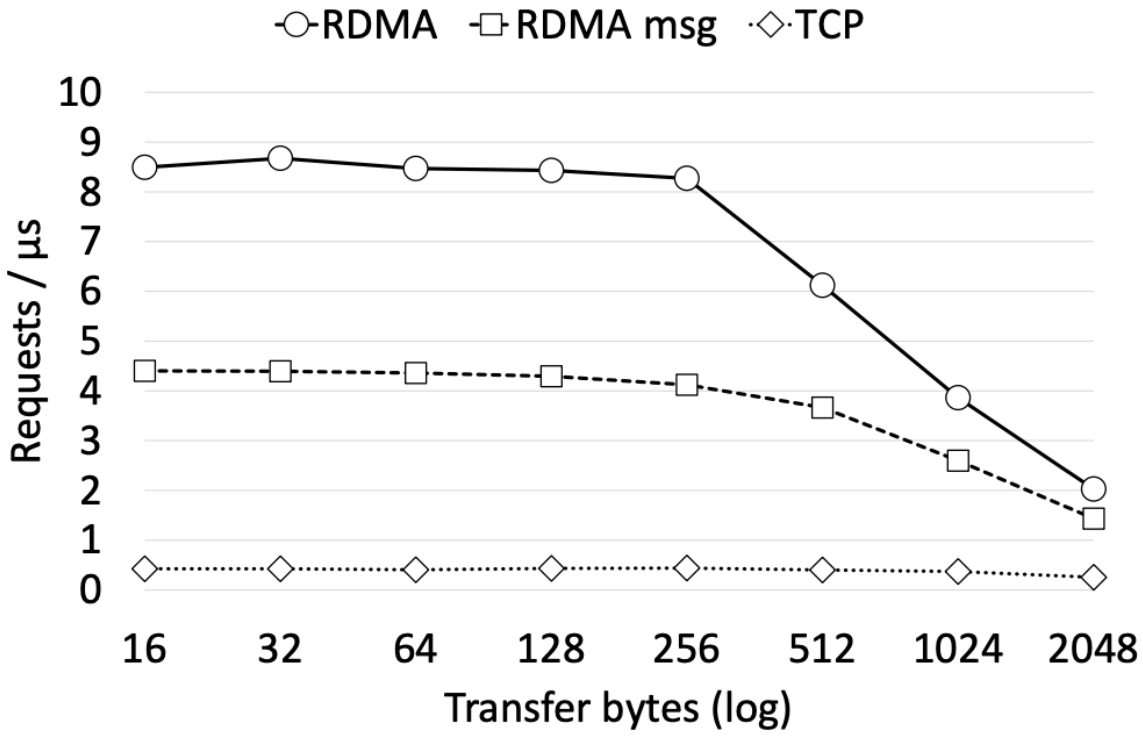
Advantage of RDMA

- Fully supported network in the hardware
- Fully zero copy

Advantage of DPDK

- More flexible, i.e., can use a customized network protocol
    - RDMA's black-boxed network protocol may result in poor performance if not using properly! (we will see)

Performance of RDMA (message)



Source: Fast Remote Memory@NSDI'14

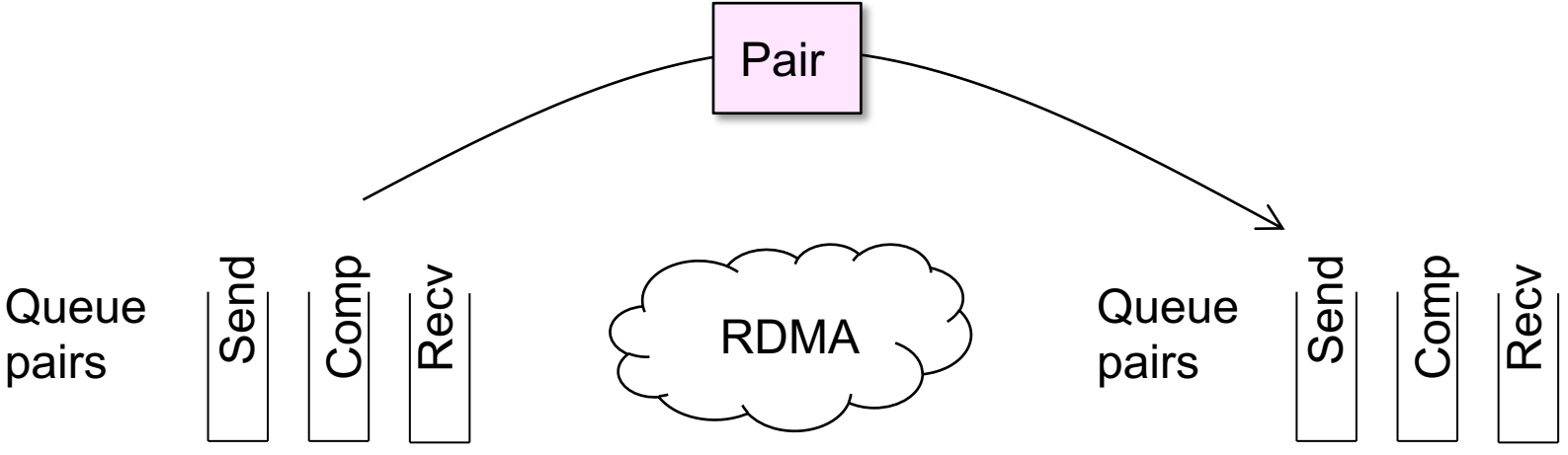
RDMA queue pair bring up

(Reliable connected) Queue pairs (QP) are connected before can be used

- Simplify reliable guarantees, etc.
- We call this bring up QP in this lecture

RDMA needs a handshake protocol to bring up two QPs

- Similar to TCP/IP handshake, but is not specified by the protocol



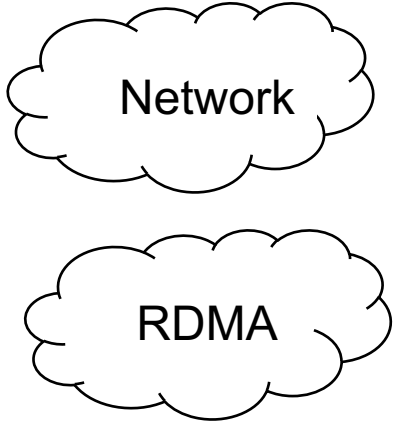
RDMA queue pair bring up

Needs to exchange a piece of connection data by the applications

– Not specified in the RDMA specification!!!

Users typically implements their own exchange method

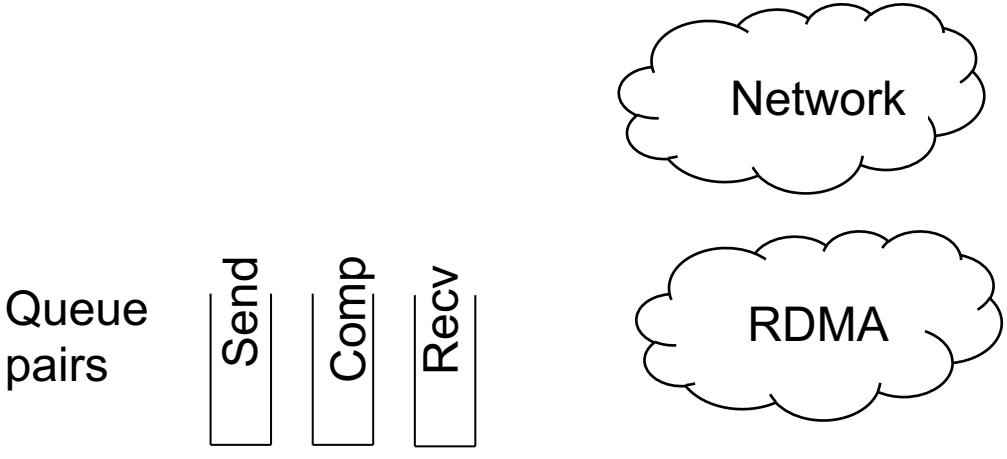
– Typically, through another network: TCP/UDP, etc.



Queue pairs

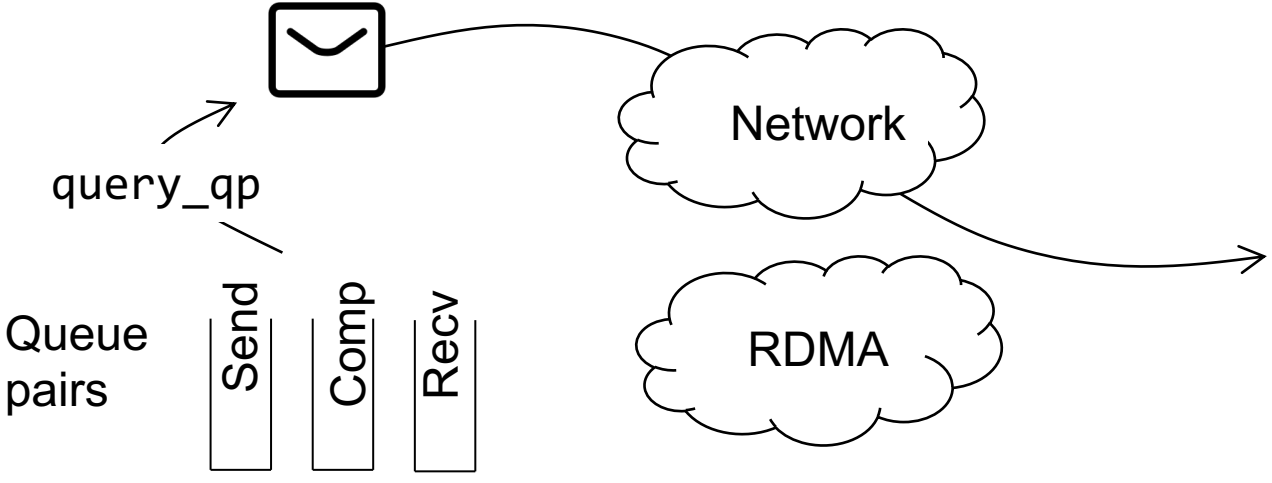
RDMA queue pair bring up

1. **Requester creates its queue pairs**



RDMA queue pair bring up

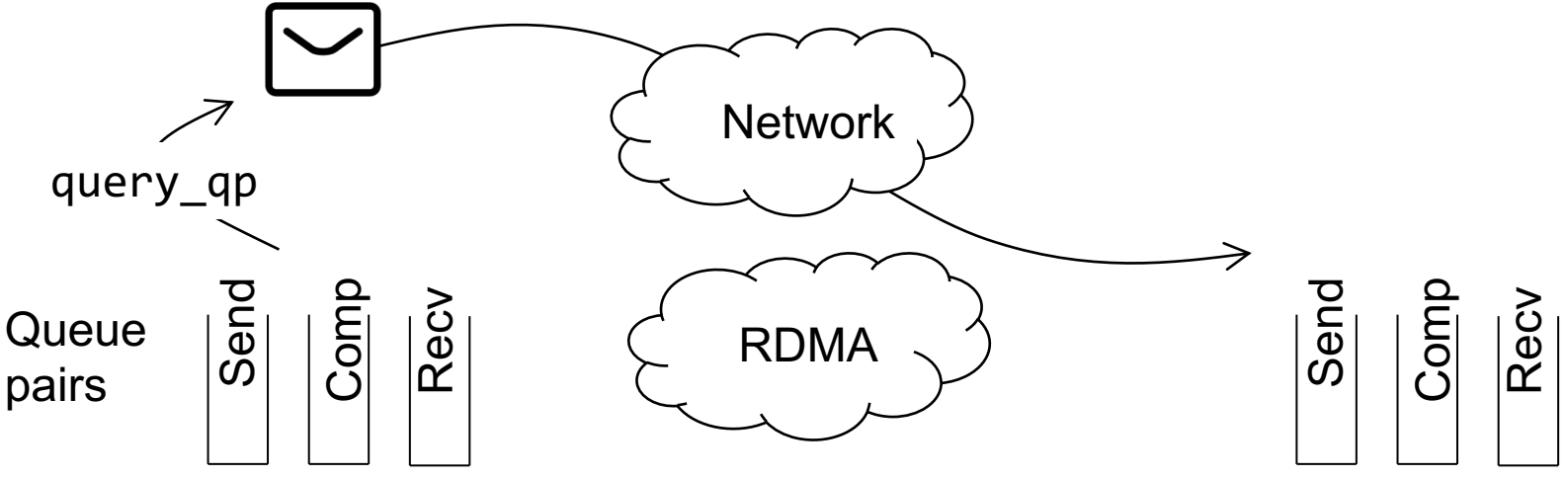
1. Requester creates its queue pairs
2. Extract the queue pair information (via query_qp), send to the responder



RDMA queue pair bring up

1. **Requester creates its queue pairs**
2. **Extract the queue pair information (via query_qp), send to the responder**
3. **(Optional) Responder creates its queue pairs**

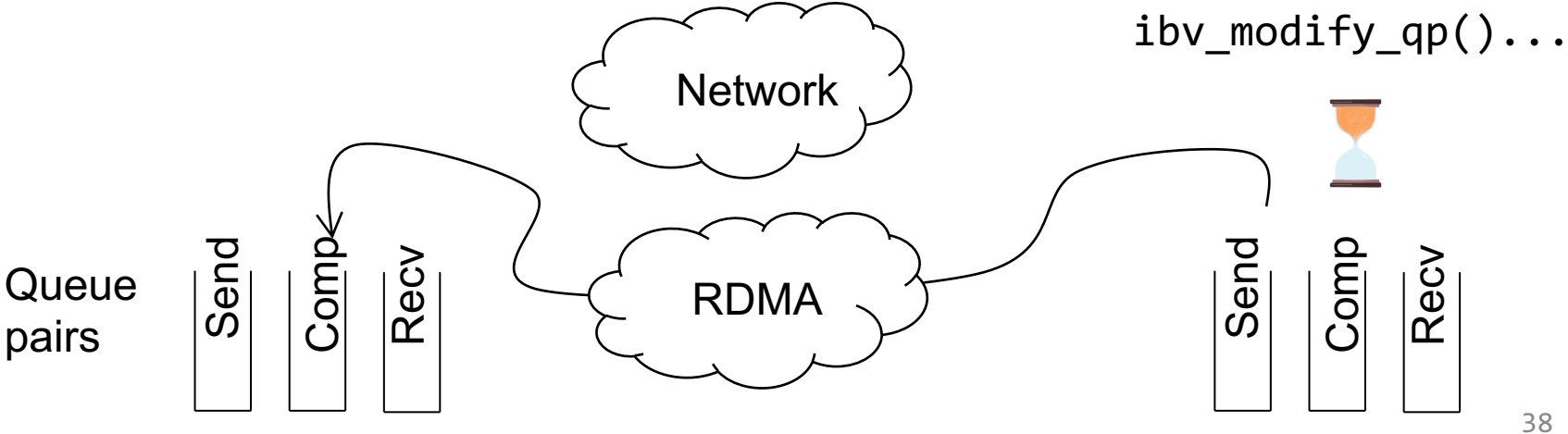
   If it already has a created queue pair, can be ignored



RDMA queue pair bring up

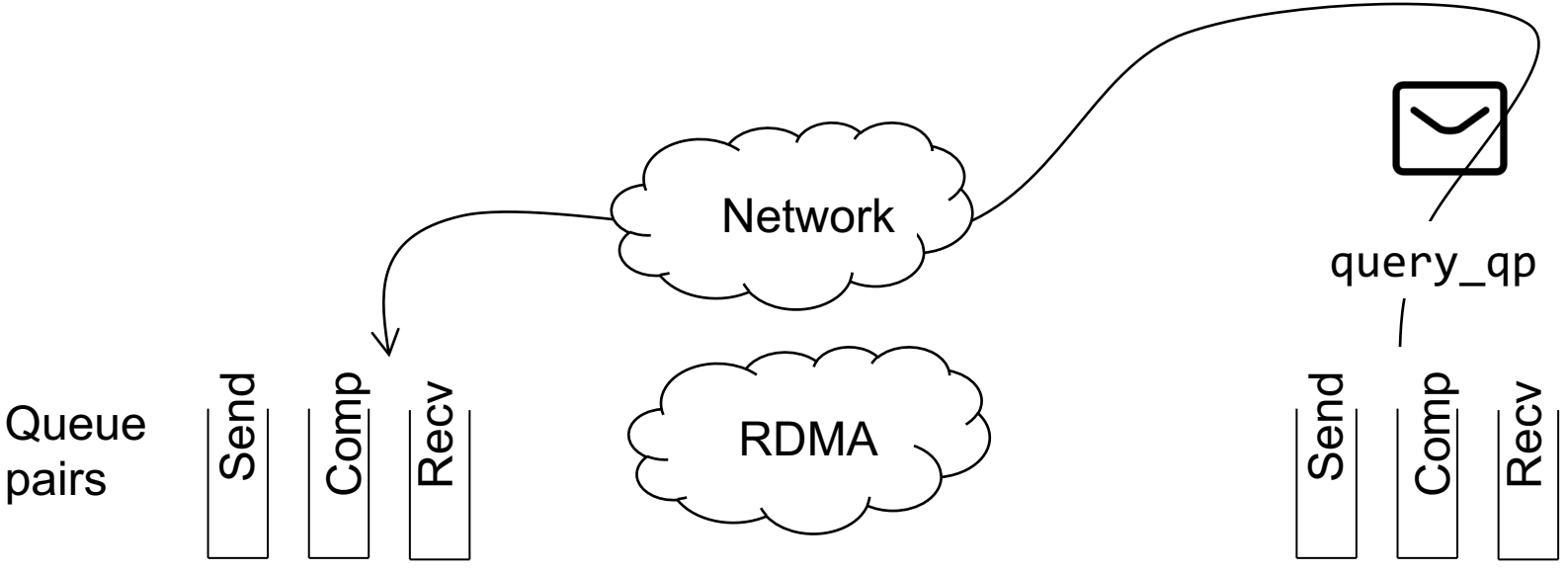
Note that the hardware will do all the connections in the modify QP

1. Requester creates its queue pairs
2. Extract the queue pair information, send to the responder
3. (Optional) Responder creates its queue pairs
4. Responder modify its queue pair to a connected state



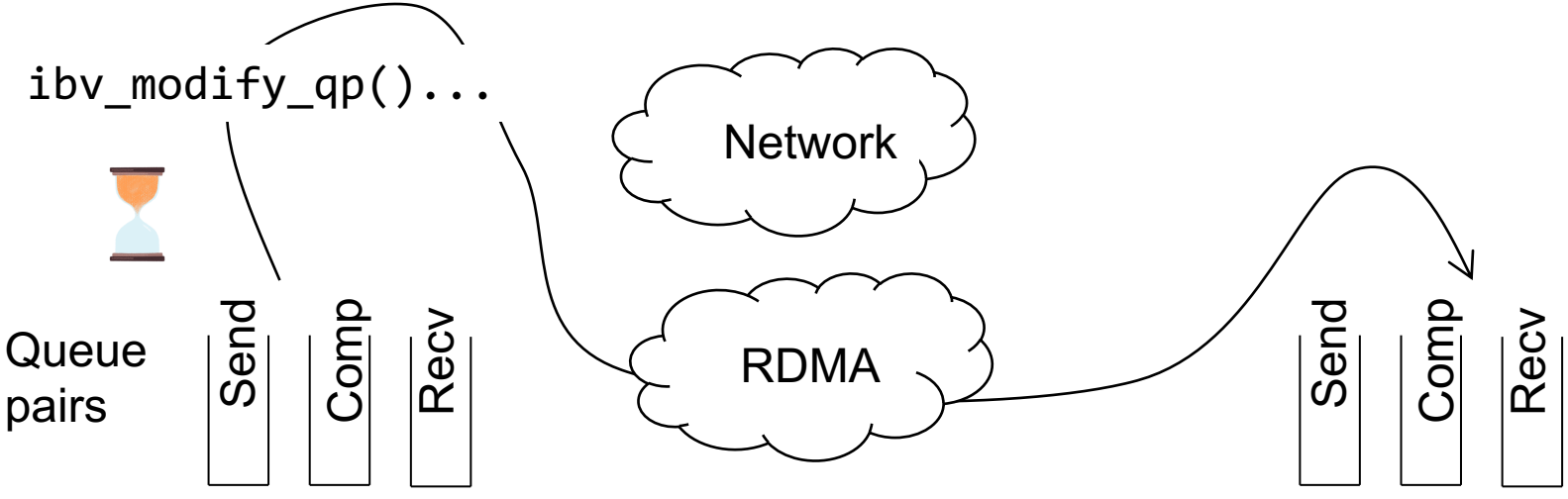
RDMA queue pair bring up

5. **Extract the queue pair information, send to the requester**



RDMA queue pair bring up

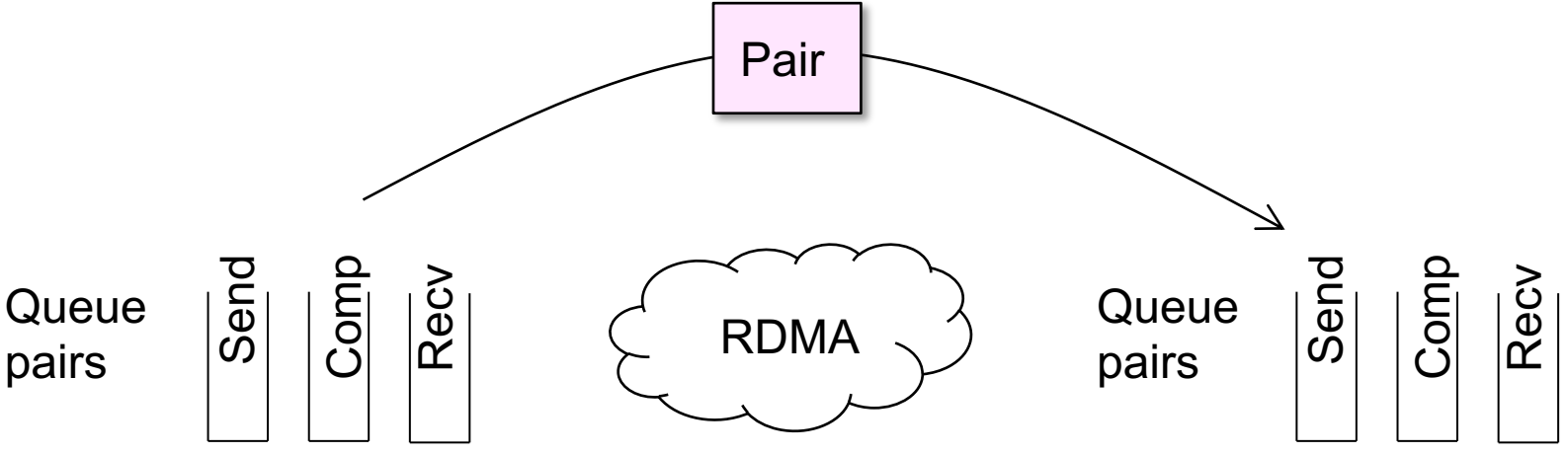
5. Extract the queue pair information, send to the requester
6. Requester modify its queue pair to a connected state with the received queue pair information



RDMA queue pair bring up

5. **Extract the queue pair information, send to the requester**
6. **Requester modify its queue pair to a connected state with the received queue pair information**
7. **Done!**

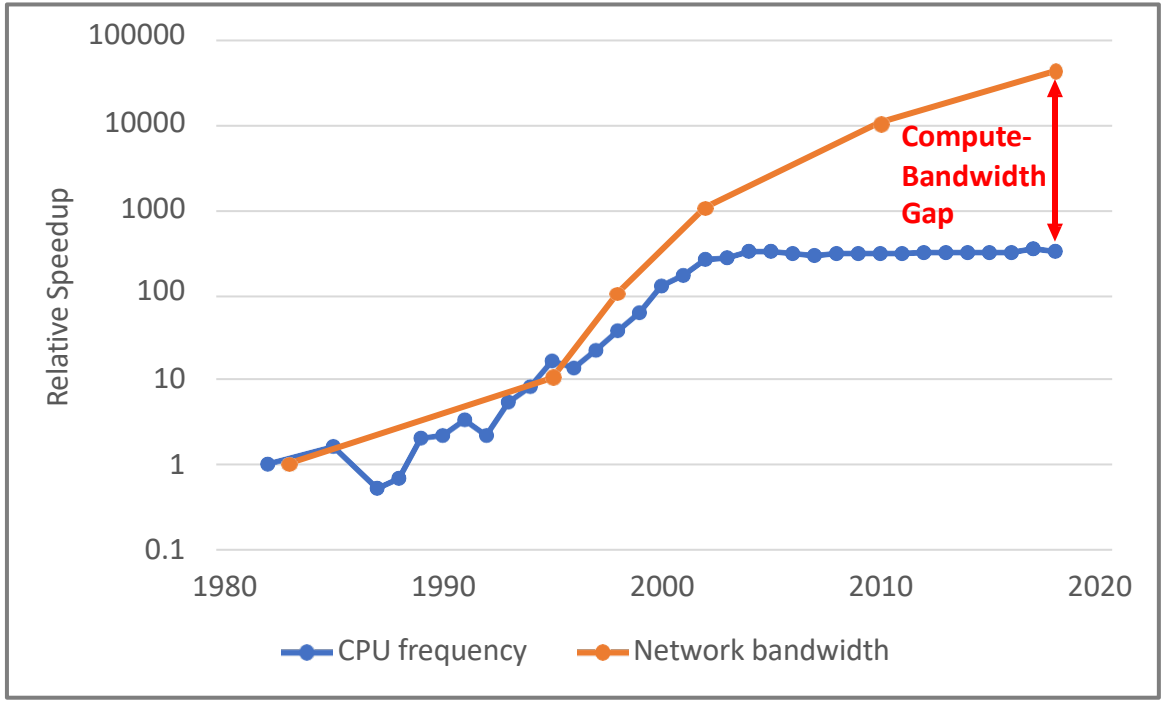
   Both queue pairs can send/receive RDMA requests from the others



Is DPDK & RDMA (messaging) sufficient?

CPU is becoming the problem

Increasing Compute-Bandwidth Gap



CPU is the problem

CPU cannot become faster

- Power wall (e.g., dennard scaling), The decline of Moore's Law

NIC is continuously becoming faster

- Lower latency (down to 600ns)
- Higher bandwidth (up to **400Gbps**)

Maybe we do not need such high bandwidth?

Most of the time the workload is idle

- **Latency** is more important than the bandwidth when the workload is idle

CPU achieves relatively low latency if idle w/ kernel-bypassing network

- No request **queuing** effect
- e.g., Two-sided RDMA achieves 2.2us RTT
- NIC RTT: ~1.5us

A hidden assumption of low-latency two-sided RDMA is using polling

Example: Bob uses polling to receive messages from the receive queue

- The while loop can easily burn the CPU!

bob

```
do {
   num_completions = ibv_poll_cq(alice_receive_qp, ...);
} while (num_completions == 0);

// The message must be stored in msg_buf if no error
Print(msg_buf) // Hello
```

Can we use interrupt to reduce the energy?

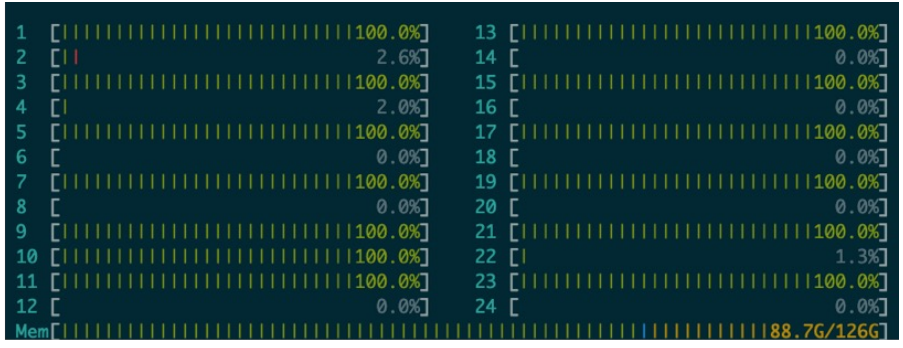
Interrupt:

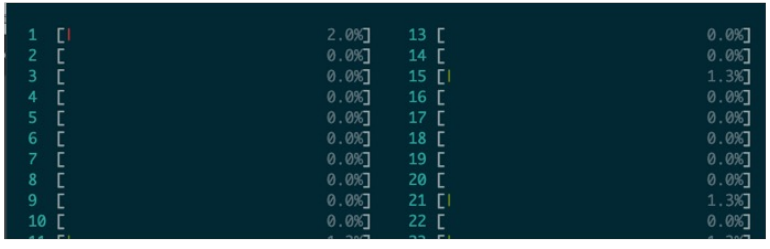
- Yield to the kernel, will be scheduled back once the message is received
- **Energy efficient**: CPU has nearly zero costs if no interrupt comes

Polling

vs.

Interrupts





Can we use interrupt to reduce the energy?

Interrupt:

- Yield to the kernel, will be scheduled back once the message is received

Problem:

- **1000X** latency increases! (2us vs. 2ms)
  - Due to the kernel and scheduling effects of the CPU

Dilemma between energy & performance!

Does kernel-bypassing networking sufficient?

Not sufficient!

Efficiency = performance + low power consumption

+

We also need CPU bypassing!

High-level concepts of RDMA

Queue Pairs (Send Queue, Complete Queue and Receive Queue)

- SQ, CQ & RQ
- A group of queues form an RDMA connection
    - Analog to connections in TCP/IP

User posts networking requests to the SQ

- Two-sided: message datagram (like `send/recv`)
- **One-sided**: remote read/write (in a CPU bypassing way!)

Get the completion events from the CQ

Receive two-sided messages from the RQ

One-sided RDMA

Observation:

- Remote read/write operations is common in applications
  - We can let NIC do all these operations!

One-sided RDMA

Observation:

- Remote read/write operations is common in applications
  - We can let NIC do all these operations!

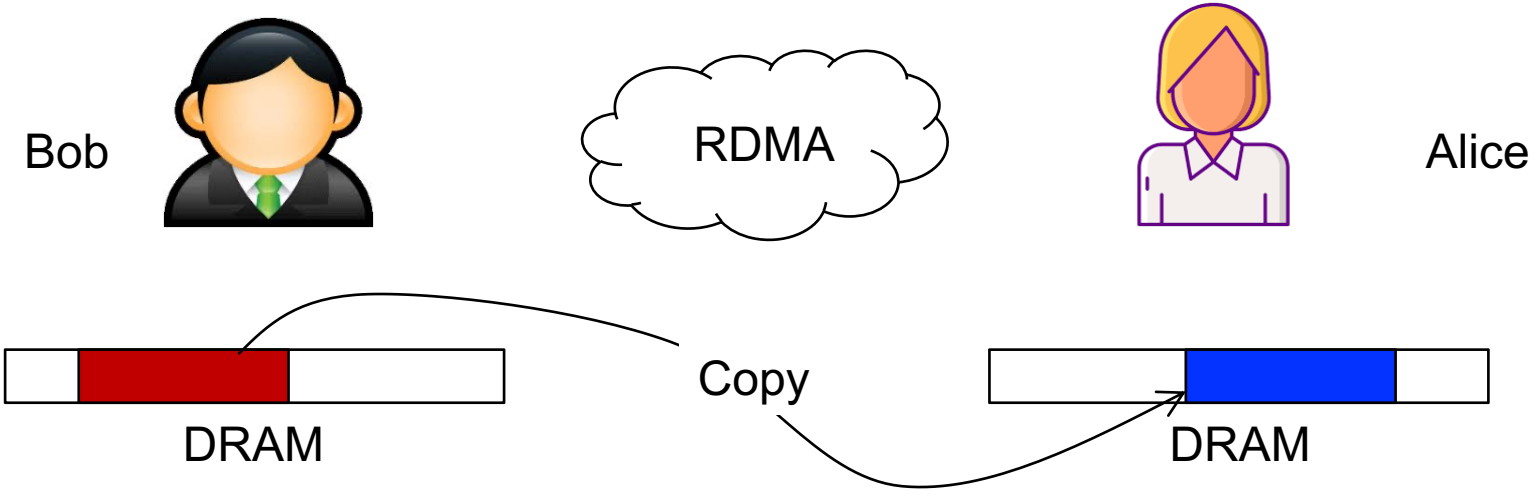
One-sided RDMA

- **READ**
  - The NIC can directly read the server memory bypassing server CPU
- **WRITE**
  - The NIC can directly write the server memory bypassing server CPU
- **ATOMICS** (fetch & add, compare & swap)
  - The NIC can directly execute atomic operations bypassing server CPU

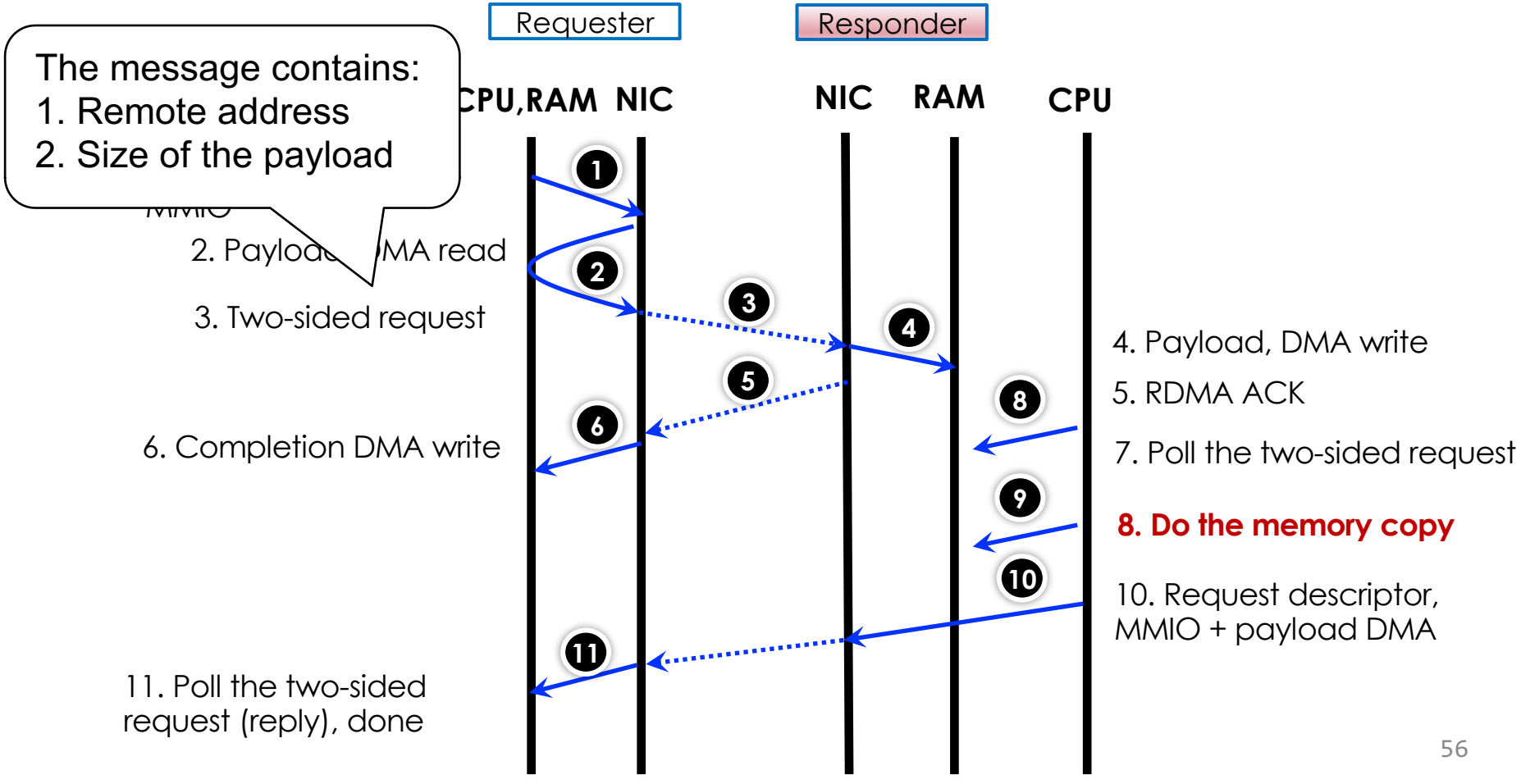
Example: remote memory copy

Suppose we want to implement a remote procedure call (RPC) with RDMA

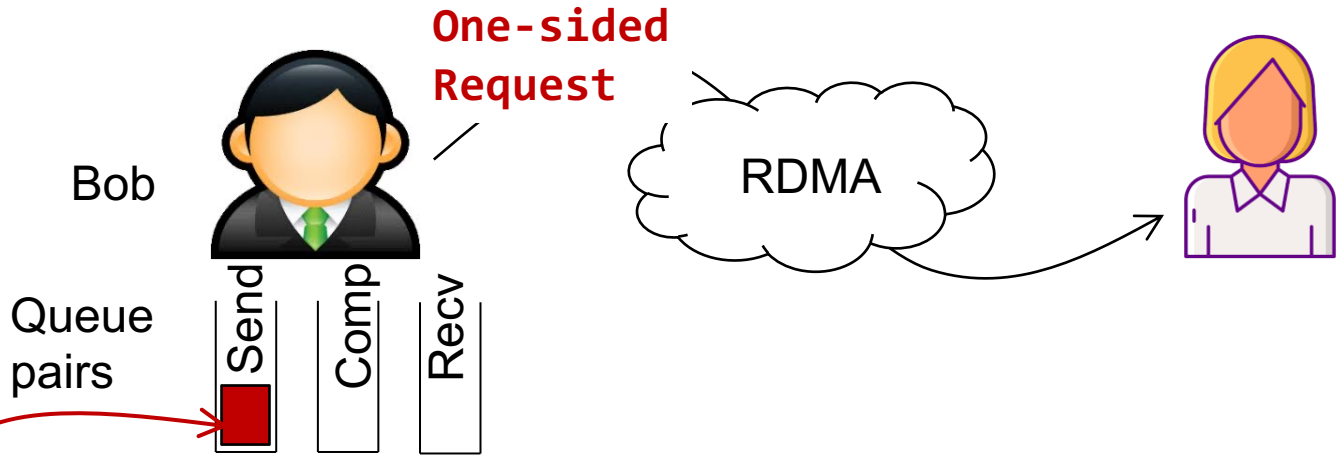
- The RPC will simply do a memory copy (`memcpy`)
- For simplicity, we assume only the requester copies to the responder



Existing approach: two-sided RDMA based RPC



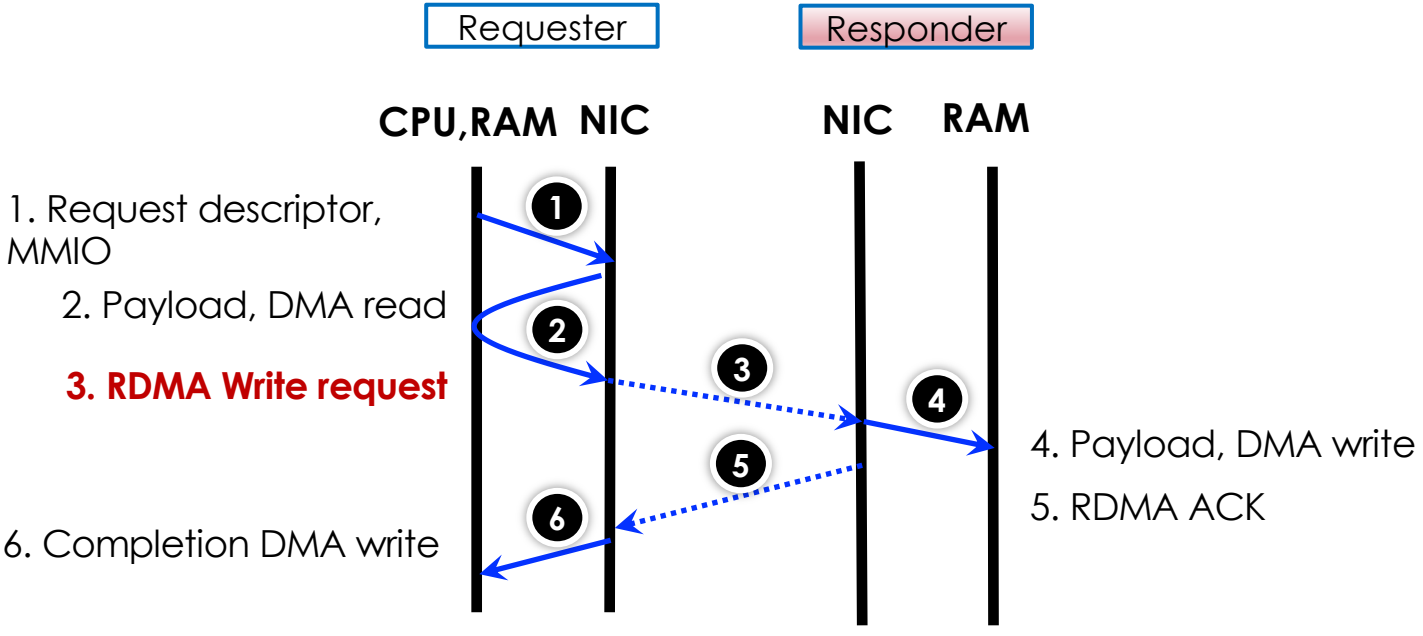
Remote memory copy with one-sided RDMA WRITE



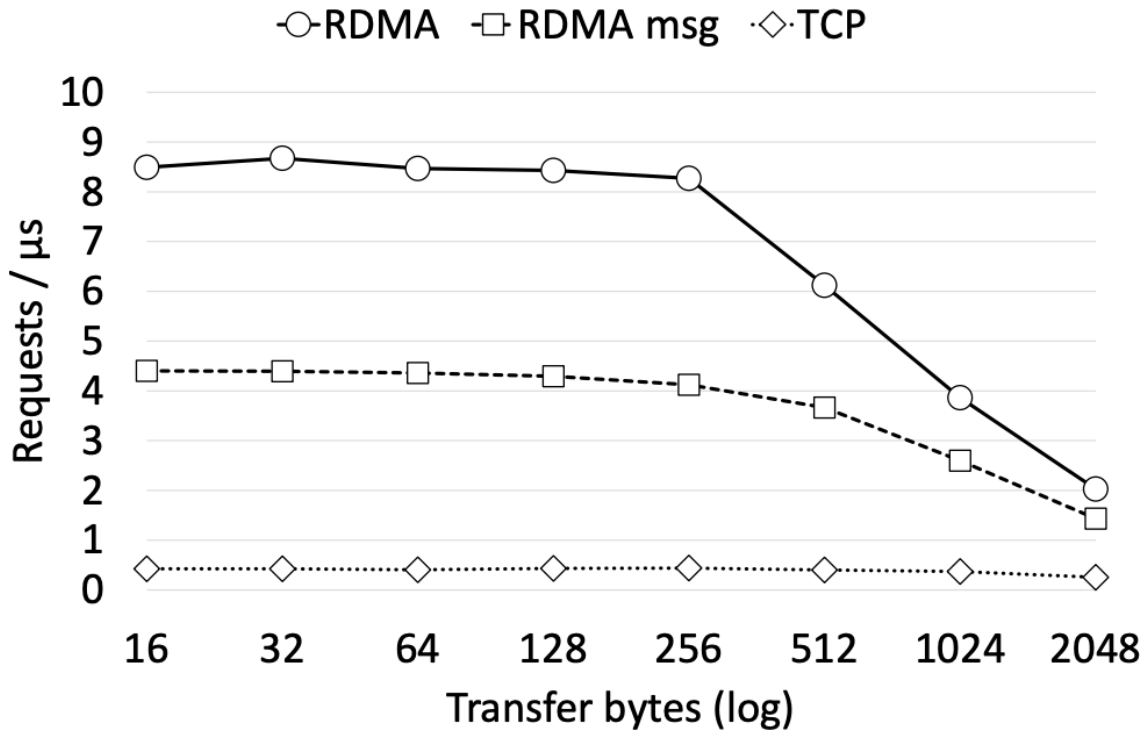

```
struct ibv_sge list = { .addr = src, length=size };
struct ibv_send_wr wr = {
    .sg_list = &list,
    .op_code = IBV_WR_RDMA_WRITE, // to the the one-sided WRITE
    .wr.wr.rdma.remote_addr = dst,
    ..., // other fields
};
ibv_post_send(bob_qp, &wr, ...); // send queue
```

bob

Remote memory copy with one-sided RDMA WRITE



One-sided RDMA WRITE vs. Two-sided RDMA



Question: how can RDMA access the remote memory?

RDMA can directly access the virtual address of the remote end

- How can the NIC find the DMA address?
- How can the remote end do the authentication?
    - E.g., some virtual address cannot be exposed to the others

```
struct ibv_sge list = { .addr = src, length=size };
struct ibv_send_wr wr = {
   .sg_list = &list,
   .op_code = IBV_WR_RDMA_WRITE, // to the the one-sided WRITE
   .wr.wr.rdma.remote_addr = dst,
   ..., // other fields
};
ibv_post_send(bob_qp, &wr, ...); // send queue
```

Memory registration (MR)

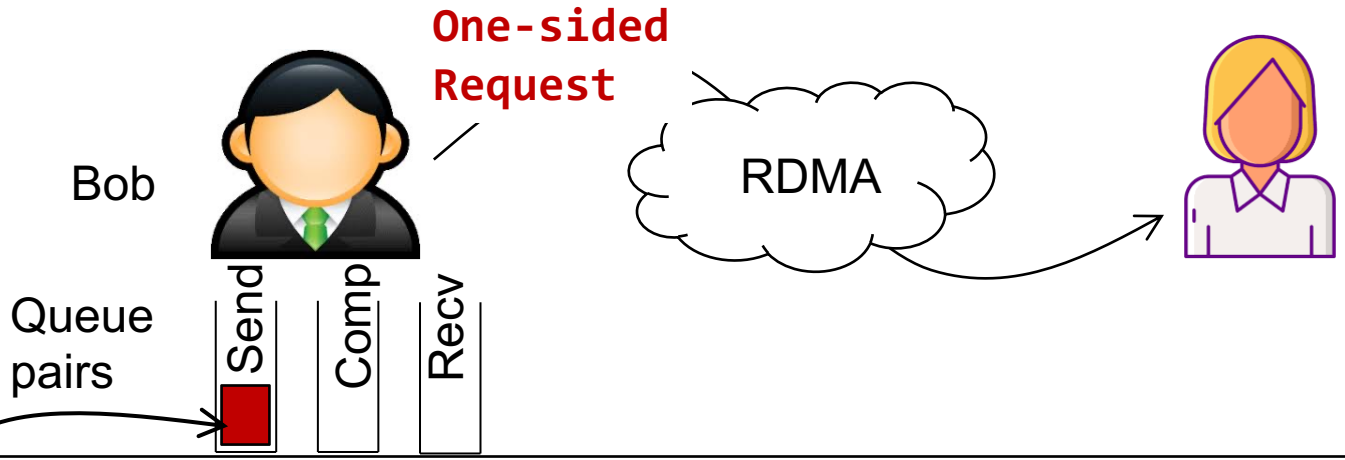
RDMA uses memory registrations to expose the virtual memory

If Alice wants to expose the memory, she must first register it to the RNIC

- Via `ibv_reg_mr()`
- Access flags: read/write,etc.
- Return: rkey

NIC will create an internal mapping

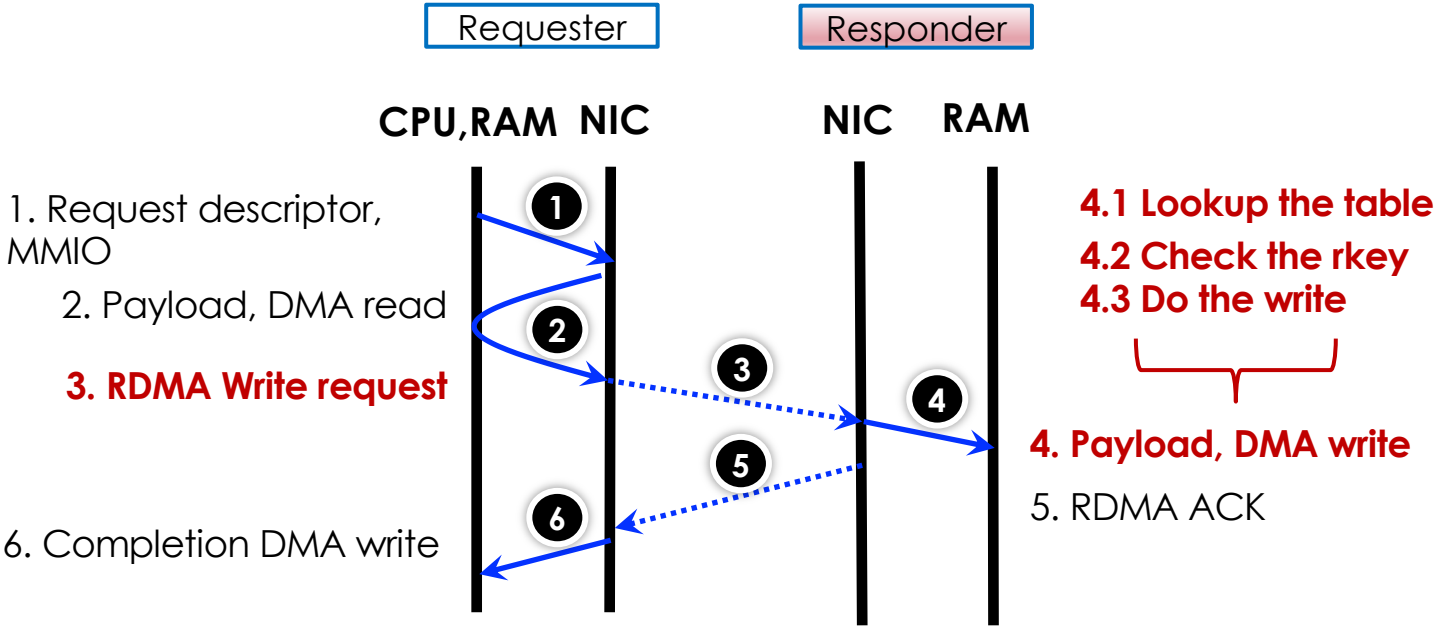
One-sided RDMA with Memory Registration



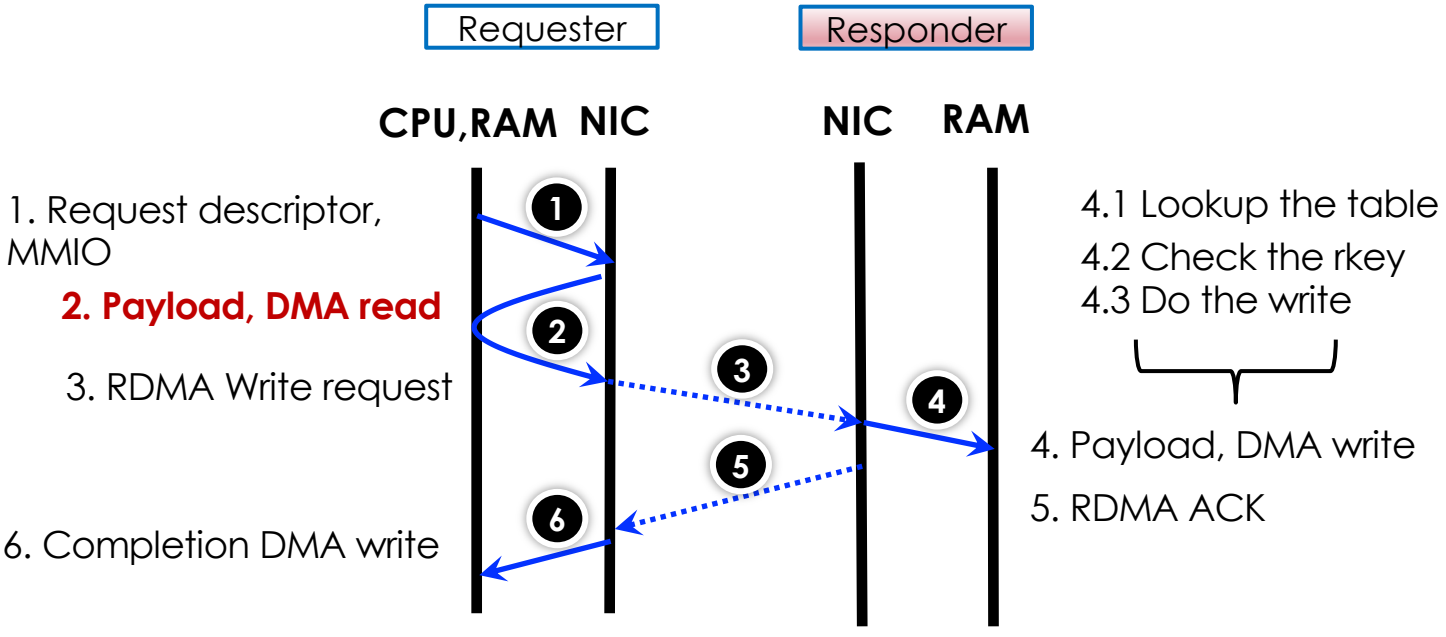
bob

```
struct ibv_sge list = { .addr = src, length=size };
struct ibv_send_wr wr = {
    .sg_list = &list,
    .op_code = IBV_WR_RDMA_WRITE, // to the the one-sided WRITE
    .wr.wr.rdma.remote_addr = dst,
    .wr.wr.rdma.rkey = ...,
    ..., // other fields
};
ibv_post_send(bob_qp, &wr, ...); // send queue
```

One-sided RDMA with Memory Registration



One-sided RDMA with Memory Registration



Question: what about the local DMA?

Local RDMA access also needs MR

The same API `ibv_reg_mr()`

- Will return two keys: `lkey & rkey`
  - Lkey is used for local accesses
  - Rkey is used for remote accesses

bob

```
struct ibv_sge list = { .addr = src, length=size, lkey = lkey };
struct ibv_send_wr wr = {
    .sg_list = &list,
    .op_code = IBV_WR_RDMA_WRITE, // to the the one-sided WRITE
    .wr.wr.rdma.remote_addr = dst,
    .wr.wr.rdma.rkey = ...,
    ..., // other fields
};
ibv_post_send(bob_qp, &wr, ...); // send queue
```

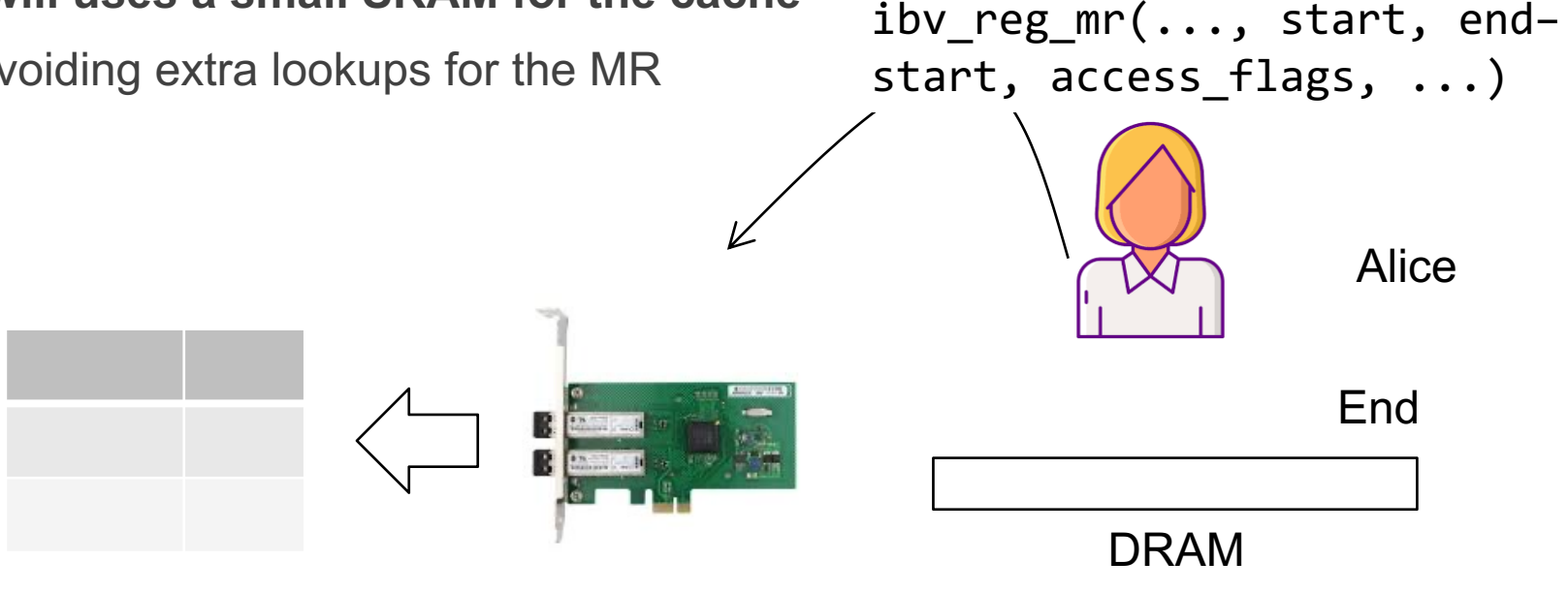
Question: where is the mapping table stored?

Stored in DRAM

– Problem: NIC will use one DMA to fetch for the check (inefficient!)

NIC will uses a small SRAM for the cache

– Avoiding extra lookups for the MR



Performance summary of RDMA

We use one-sided RDMA WRITE as an example

① MMIO: 200-800ns (on Intel x864 machines)

② + (4) DMA: (about) 1us

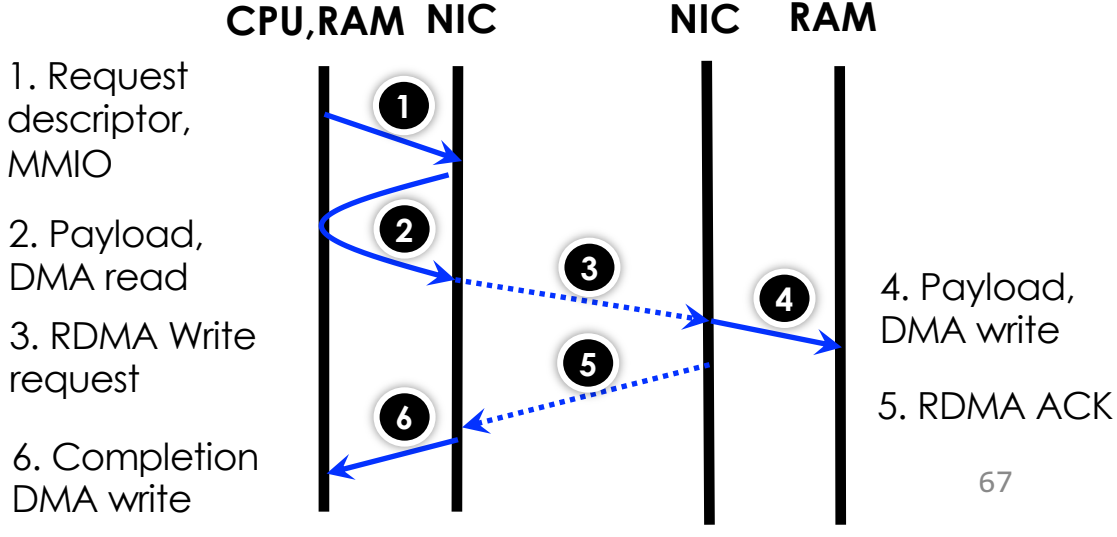
③ + (5): network, depends on the distance & speed of light

<< 1us in rack-scale clusters

⑥ Memory access

100-400ns

Total RTT: 2-5us



Short recap of RDMA

RDMA: a new networking protocol & API

– Extremely fast: 2-5us in a local rack-scale cluster

Kernel, Network protocol bypassing

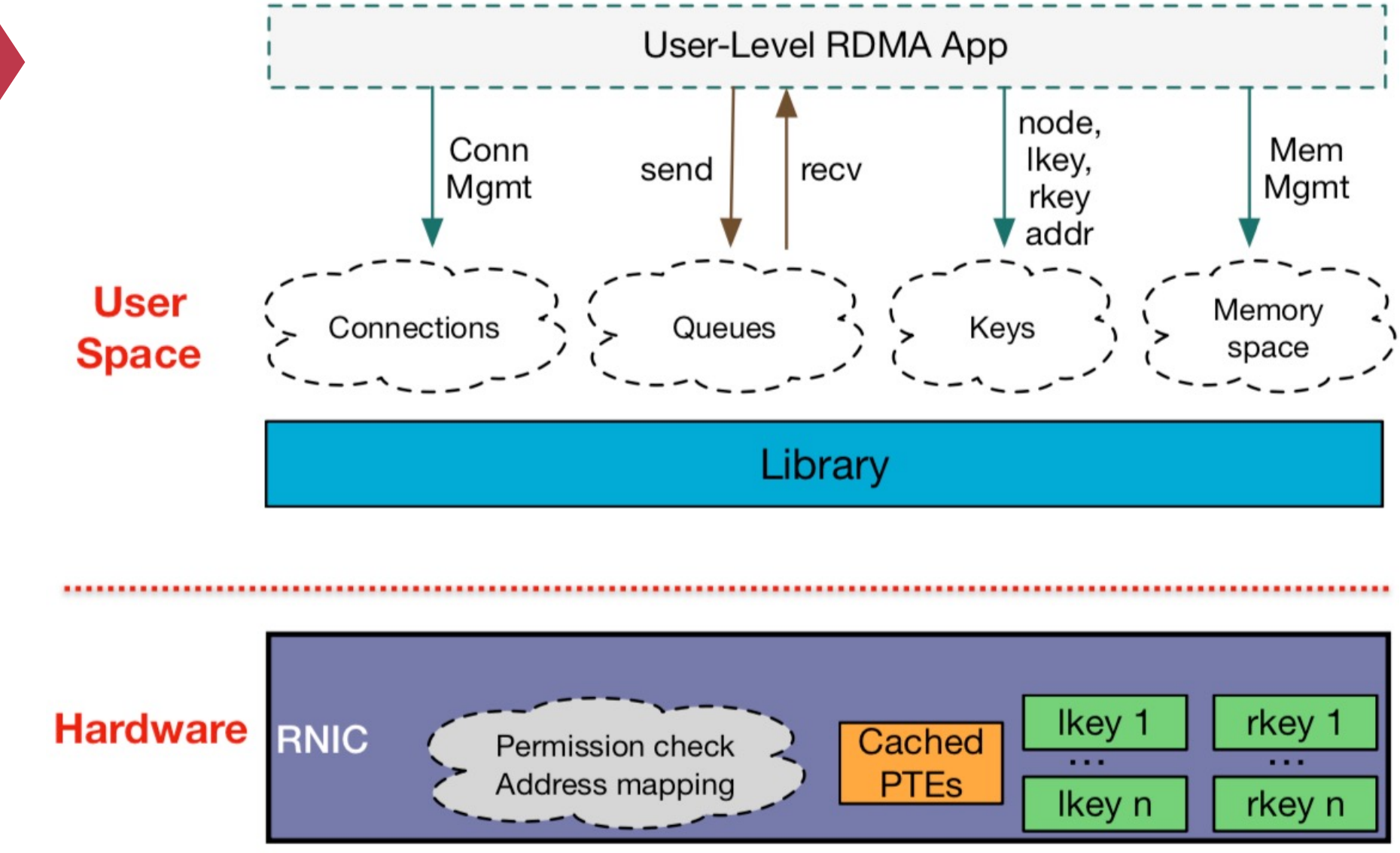
– All RDMA operations

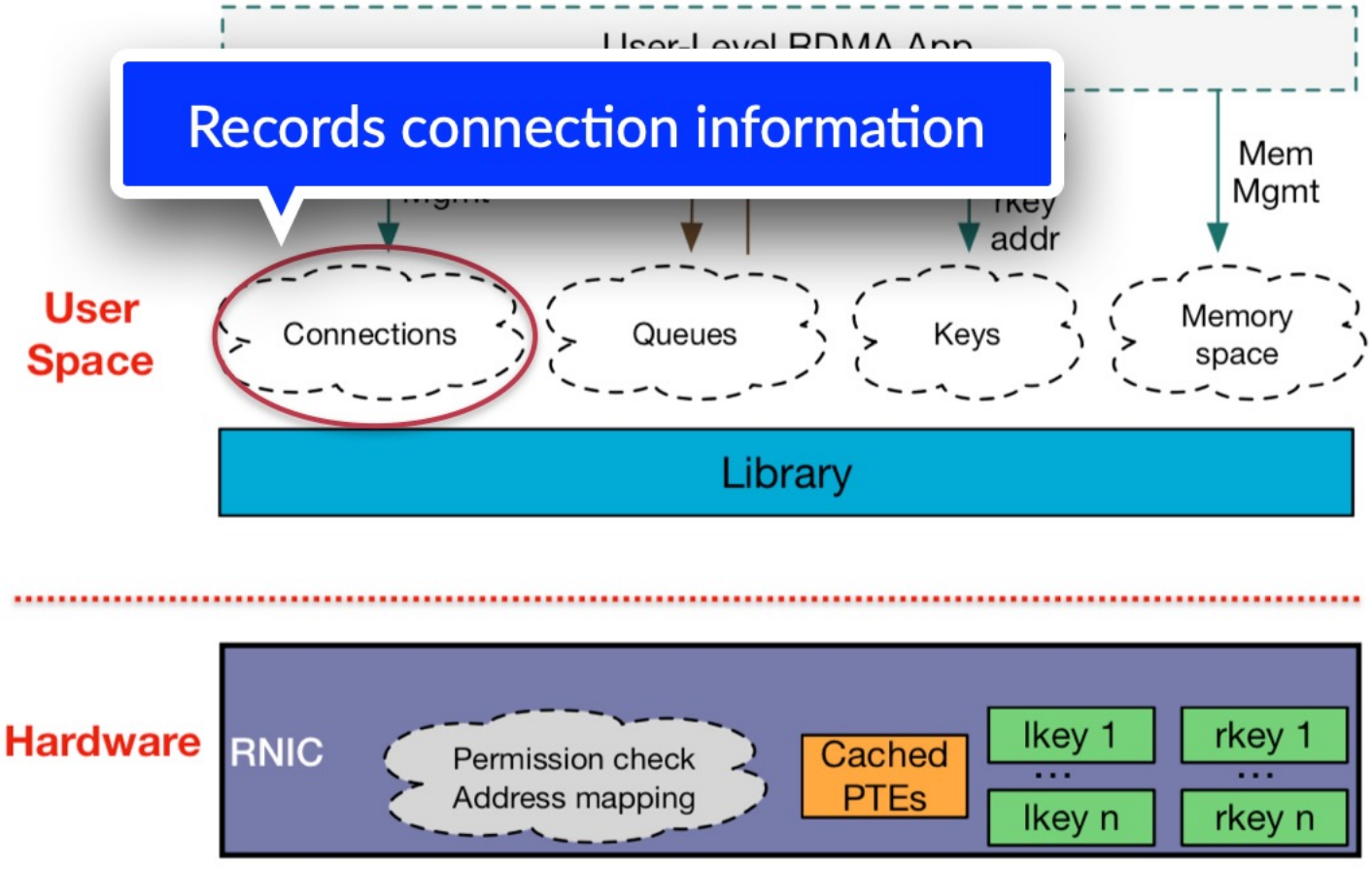
CPU-bypassing

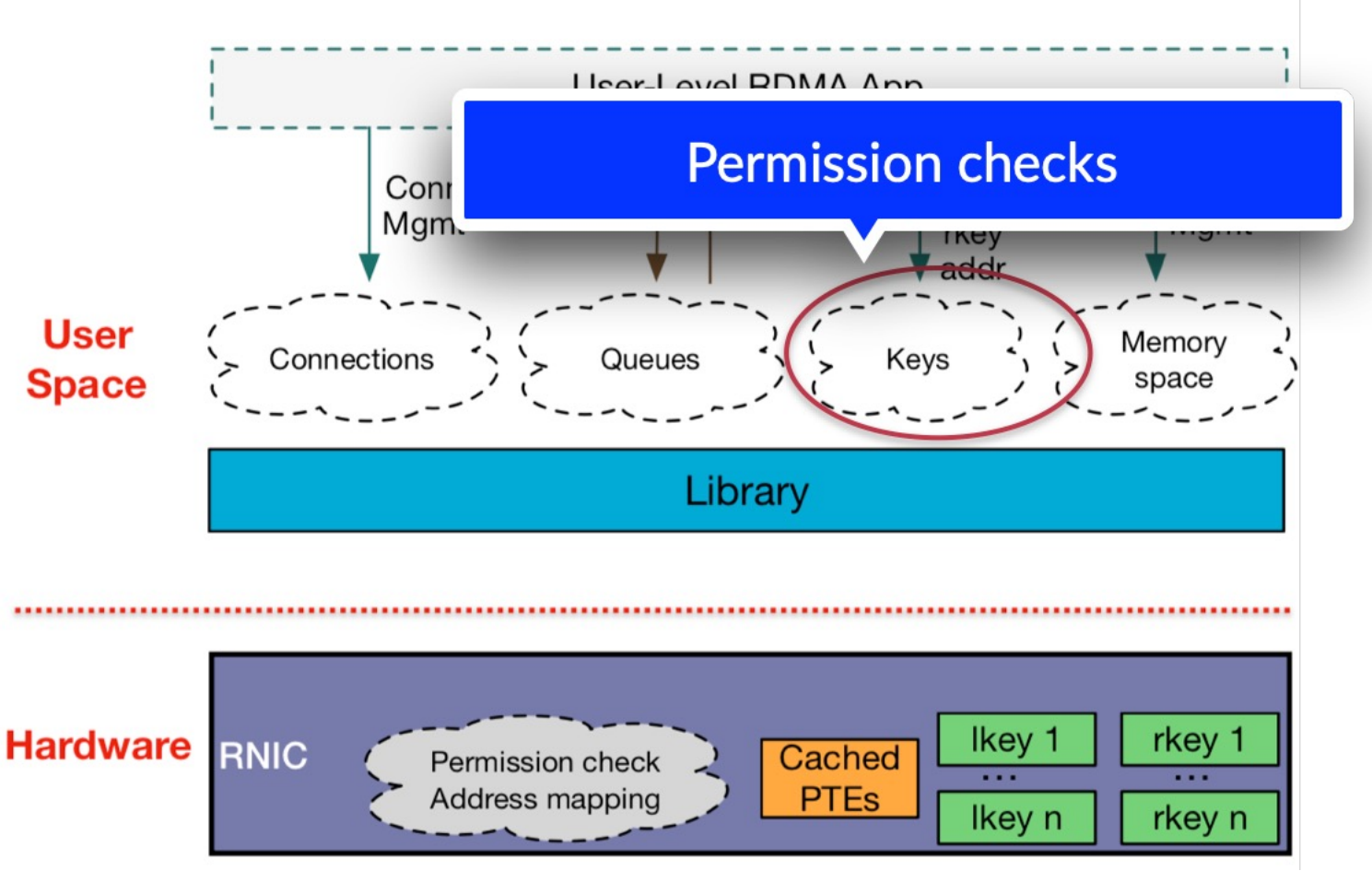
– One-sided RDMA operations

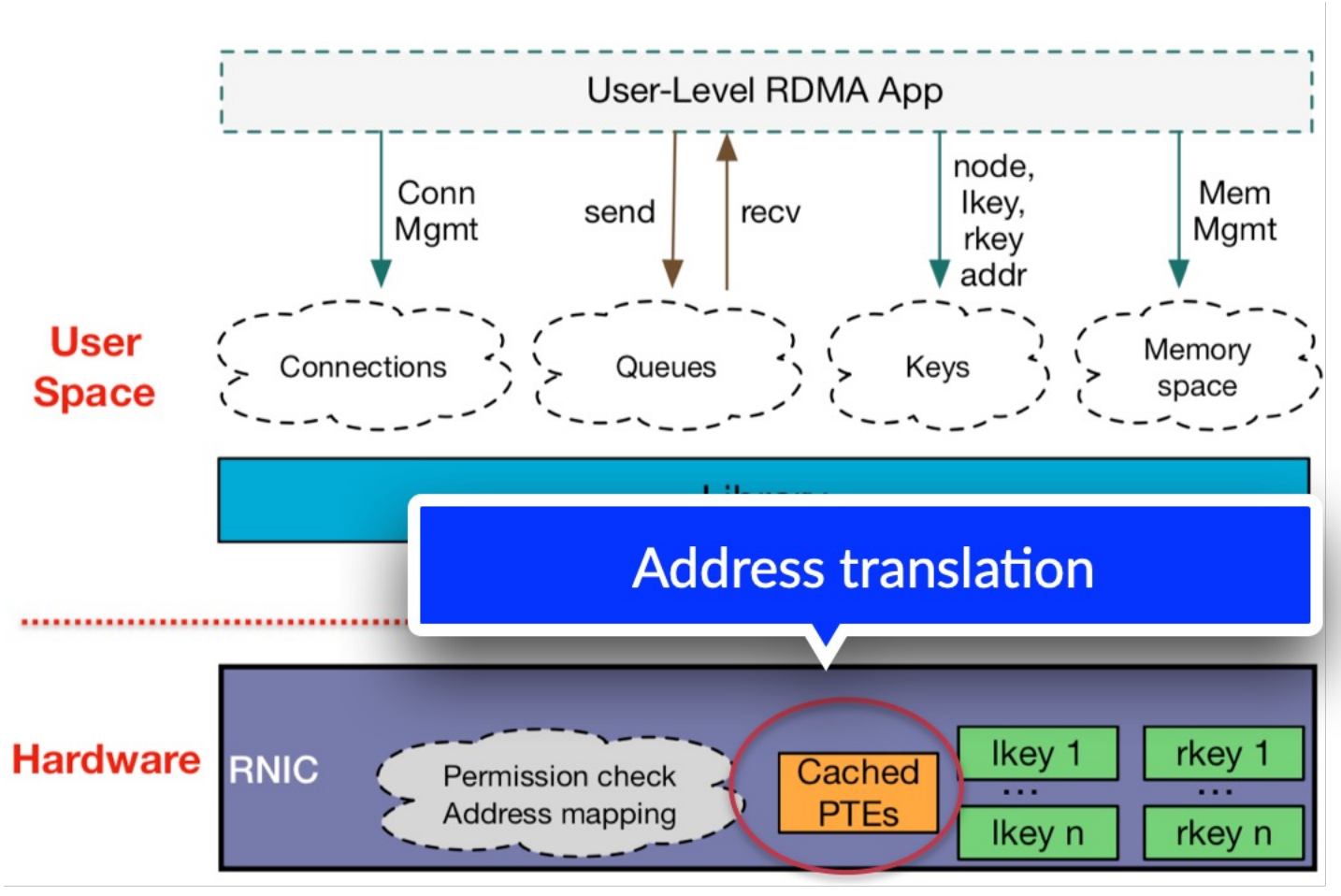
  • READ,WRITE and ATOMICS

Understanding the implementation details of RDMA matters


User-Level RDMA App
Conn Mgmt
send
recv
node, lkey, rkey addr
Mem Mgmt
User Space
Connections
Queues
Keys
Memory space
Library
Hardware
RNIC
Permission check
Address mapping
Cached PTEs
lkey 1
lkey n
rkey 1
rkey n


Records connection information
Mem Mgmt
addr
User Space
Connections
Queues
Keys
Memory space
Library
Hardware
RNIC
Permission check
Address mapping
Cached PTEs
lkey 1
lkey n
rkey 1
rkey n


Permission checks
User Space
Connections
Queues
Keys
Memory space
Library
Hardware
RNIC
Permission check
Address mapping
Cached PTEs
lkey 1
lkey n
rkey 1
rkey n


User-Level RDMA App
Conn Mgmt
send
recv
node, lkey, rkey addr
Mem Mgmt
User Space
Connections
Queues
Keys
Memory space
Address translation
Hardware
RNIC
Permission check
Address mapping
Cached PTEs
lkey 1
lkey n
rkey 1
rkey n

RDMA: out memory design

Connection, translation, permissions, etc. information are stored on the host memory

- NIC only caches them in its SRAM
- However, NIC has limited SRAM!
    - Even high-end NIC today has 2MB buffer (ConnectX-5)

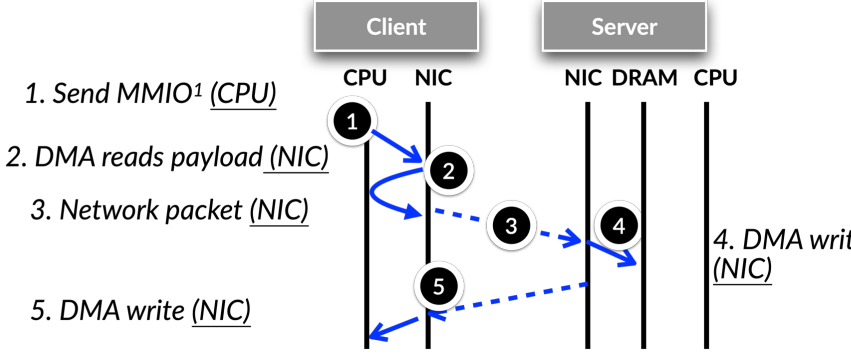
RDMA: out memory design

Connection, translation, permissions, etc. information are stored on the host memory

- NIC only caches them in its SRAM
- However, NIC has limited SRAM!
    - Even high-end NIC today has 2MB buffer (ConnectX-5)

What if the NIC cache misses?

- Use PCIe read to fetch them



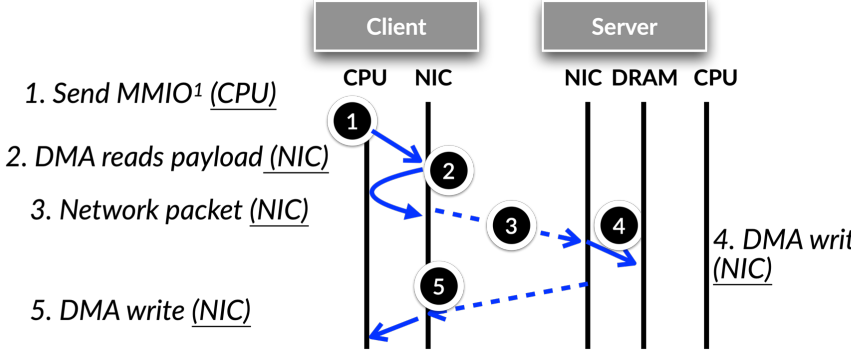
RDMA: out memory design

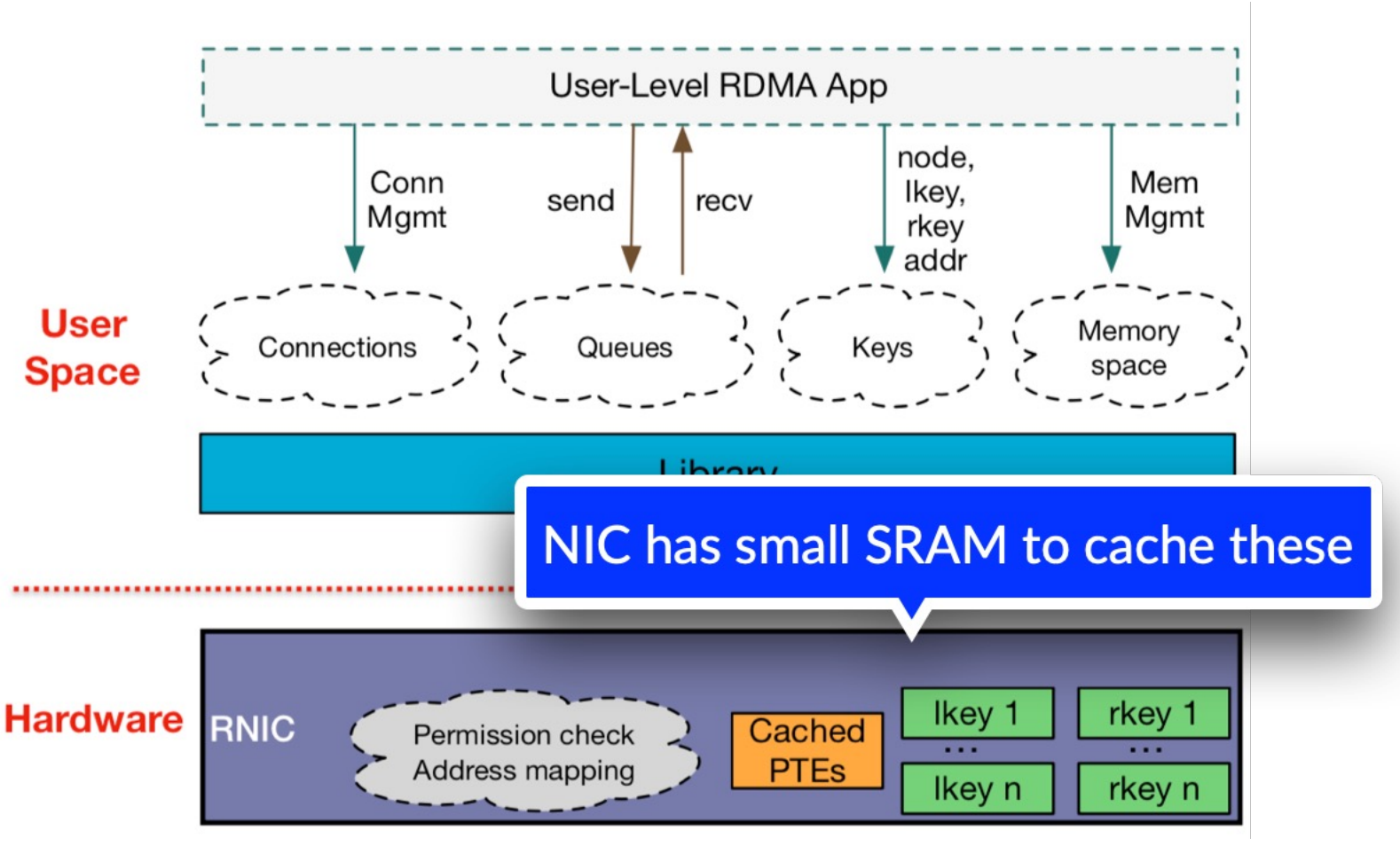
Connection, translation, permissions, etc. information are stored on the host memory

- NIC only caches them in its SRAM
- However, NIC has limited SRAM!
  - Even high-end NIC today has 2MB buffer (ConnectX-5)

What if the NIC cache misses?

- Use PCIe read to fetch them
- Compete DMA w/ normal requests!




User-Level RDMA App
Conn Mgmt
send
recv
node, lkey, rkey addr
Mem Mgmt
User Space
Connections
Queues
Keys
Memory space
Library
NIC has small SRAM to cache these
Hardware
RNIC
Permission check
Address mapping
Cached PTEs
lkey 1
lkey n
rkey 1
rkey n

RDMA: out memory design makes cache miss costly

Driver stores these at the host memory

- NIC caches a portion of such information

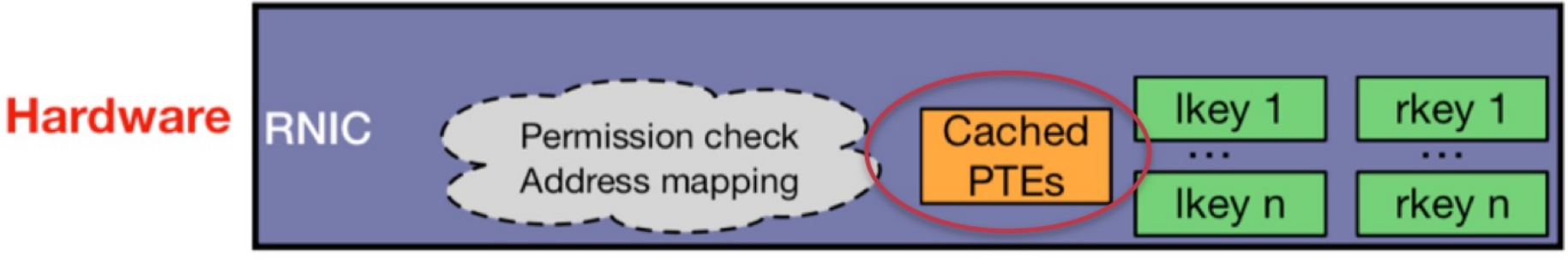
Fetch miss items using PCIe read

- ~ 1us additional latency, fetch from DRAM using PCIe read
- For comparison, RDMA one-sided read takes ~2us

Some can be mitigated, some can not

FaRM@NSDI'14 uses huge page (2MB page instead of 4KB page)

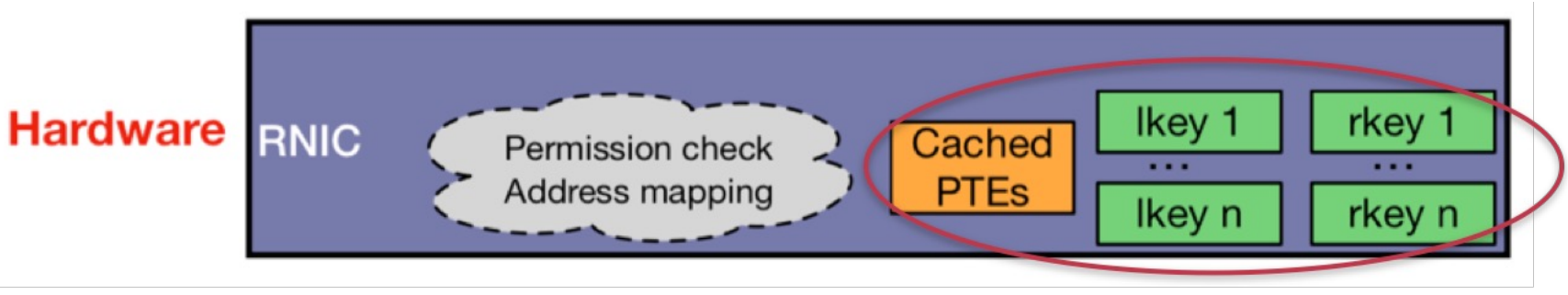
- Reduce #PTE entries
- E.g., 1GB memory only requires 2048 PTE entries, while 4KB needs 250,000 entries



Some can be mitigated, some can not

LITE@SOSP'17 takes to another extreme; it directly physical memory, and uses kernel for security check

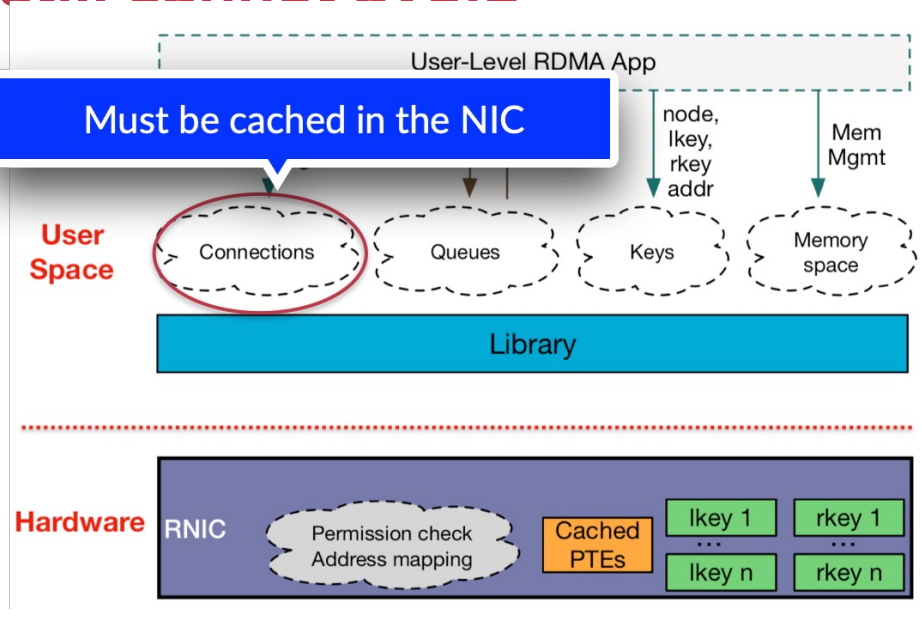
– No PTE and keys need to be cached at the NIC!



But connection information still cannot avoid

NIC must cache the connection information to accelerate the process

- What if there is a cache miss?



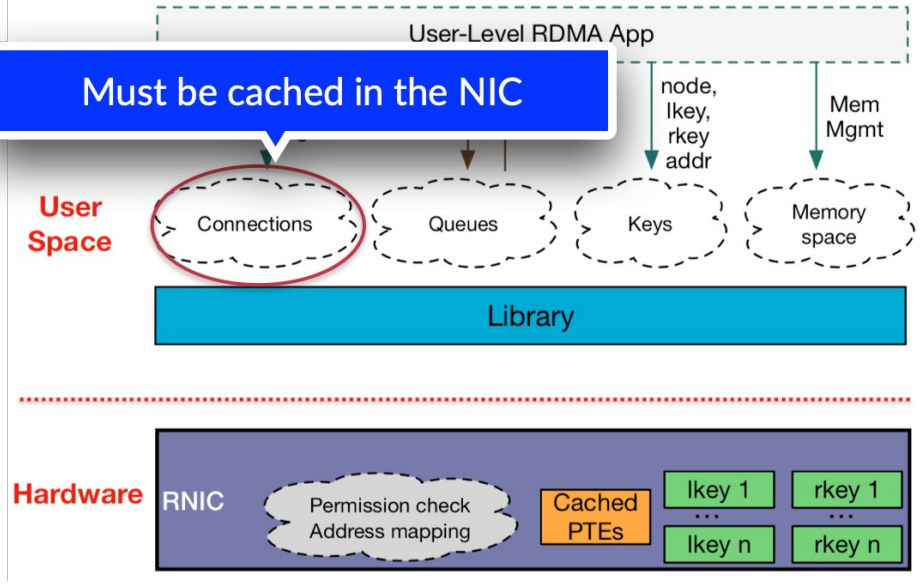
But connection information still cannot avoid

NIC must cache the connection information to accelerate the process

- What if there is a cache miss?

Performance drop caused by cache misses

- Extra PCIe READ for each request
  - 1 -> 2



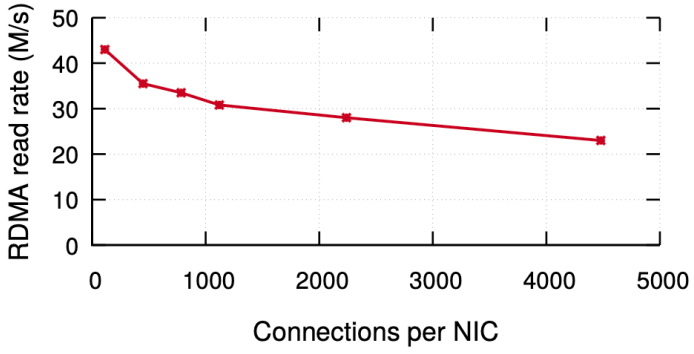


Figure 1: Connection scalability of ConnectX-5 NICs

[1] credits: eRPC@NSDI'19

This leads to another extreme

Case study: FaSST@OSDI16

RDMA provides an unreliable transport: **unreliable datagram (UD)**

- Similar to UDP (consider traditional RDMA connection as TCP)

UD is scalable:

- No connection information is kept at the NIC

Drawbacks of UD:

- No reliability guarantee
- Only supports two-sided RDMA

Two-sided RDMA is sufficient for RPC: a key building block in distributed systems!

Case study: FaSST@OSDI16

RDMA provides an unreliable transport: **unreliable datagram (UD)**

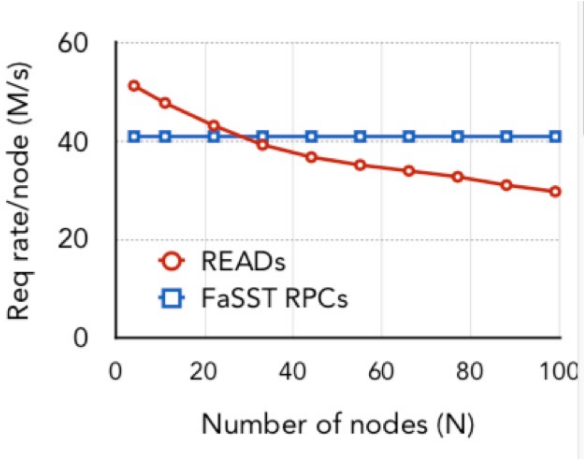
- Similar to UDP (consider traditional RDMA connection as TCP)

UD is scalable:

- No connection information is kept at the NIC

Drawbacks of UD:

- No reliability guarantee
- Only supports two-sided RDMA



Two-sided RDMA is sufficient for RPC: a key building block in distributed systems!

Does no reliability a significant problem?

RDMA provides an unreliable transport: **unreliable datagram (UD)**

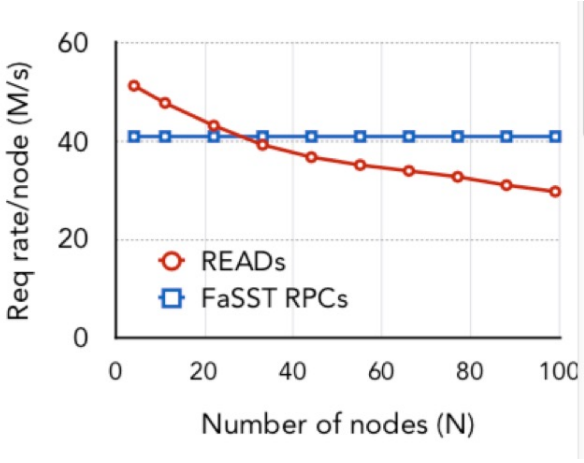
- Similar to UDP (consider traditional RDMA connection as TCP)

UD is scalable:

- No connection information is kept at the NIC

Drawbacks of UD:

- No reliability guarantee
- Only supports two-sided RDMA



Not that significant: RDMA rarely drop packets

#1. RDMA requires the link-layer to guarantee reliability

- e.g., RoCE uses packet flow control to avoid switch drop packets

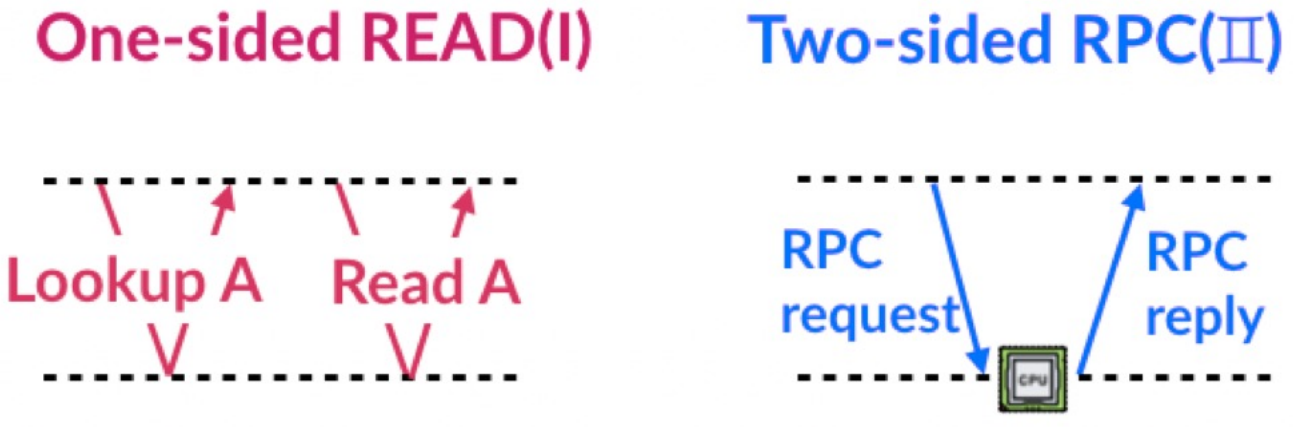
#2. The semantic of RPC simplifies the packet drop checking

- The remote end must send a reply
- If timeout, treat the packet as loss

Other benefits of using RPC

Reduced network roundtrips

- Compared to one-sided RDMA
- Example: implement a key-value store using FaRM's approach



Takeaway of RPC vs. one-sided RDMA

Not a hard conclusion!

Pros of one-sided

- Energy efficient
- Better (theoretical performance)
- CPU bypassing

Cons of one-sided

- Poor offloading semantic
- Scalability issues

Pros of two-sided

- Easy programming
- Scalable performance

Cons of two-sided

- Low performance when scalability is not an issue
- CPU overhead