nsdi '25

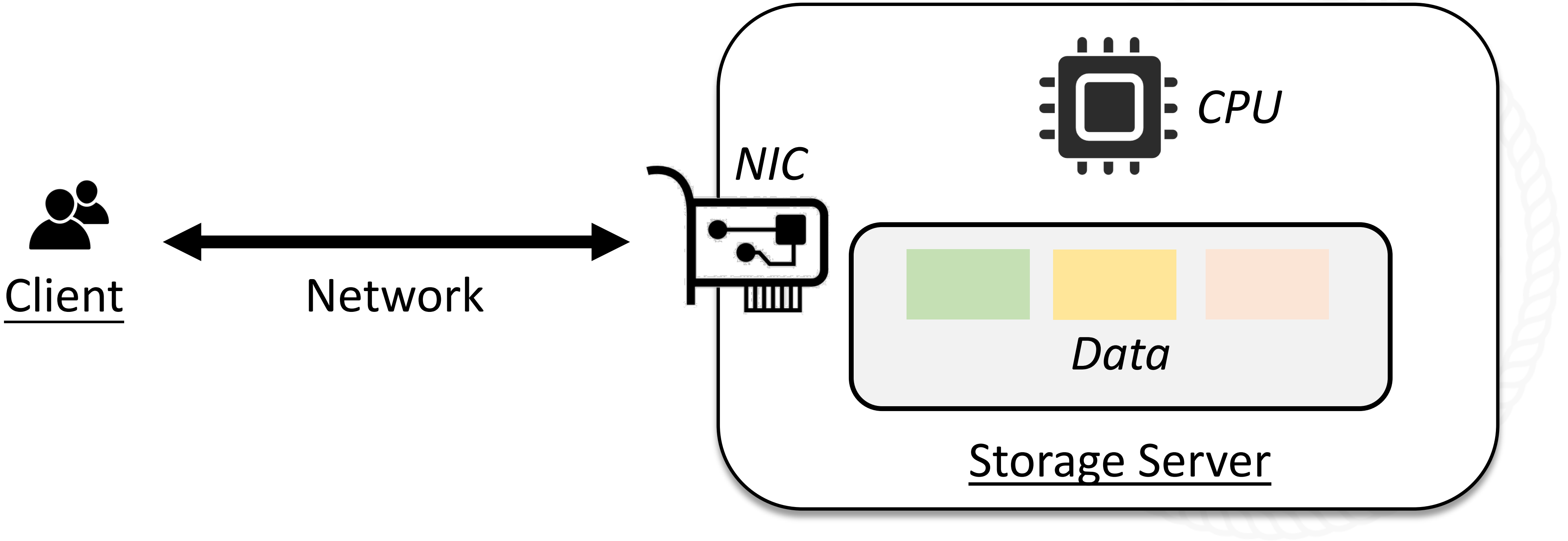
Achieving Wire-Latency Storage Systems by Exploiting Hardware ACKs

Qing Wang, Jiwu Shu, Jing Wang, Yuhao Zhang

Tsinghua University

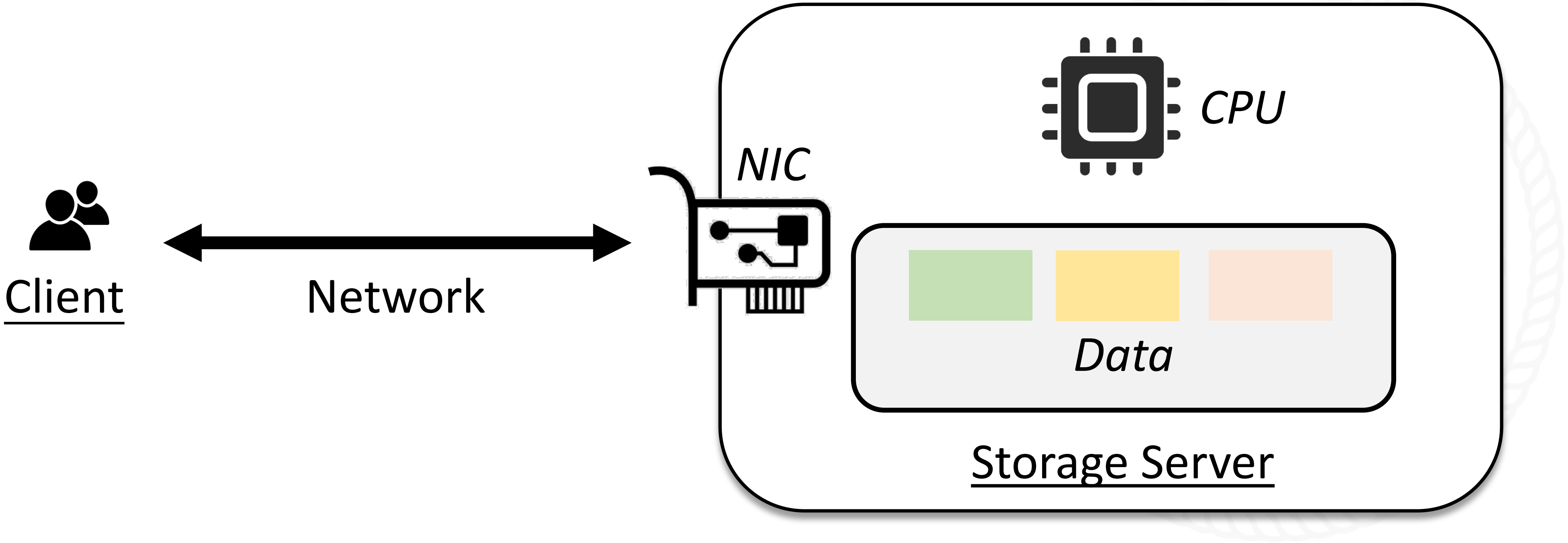
Communication in networked storage systems

- Clients manipulate data in the storage server connected via network
  - Post requests and obtain responses



Communication in networked storage systems

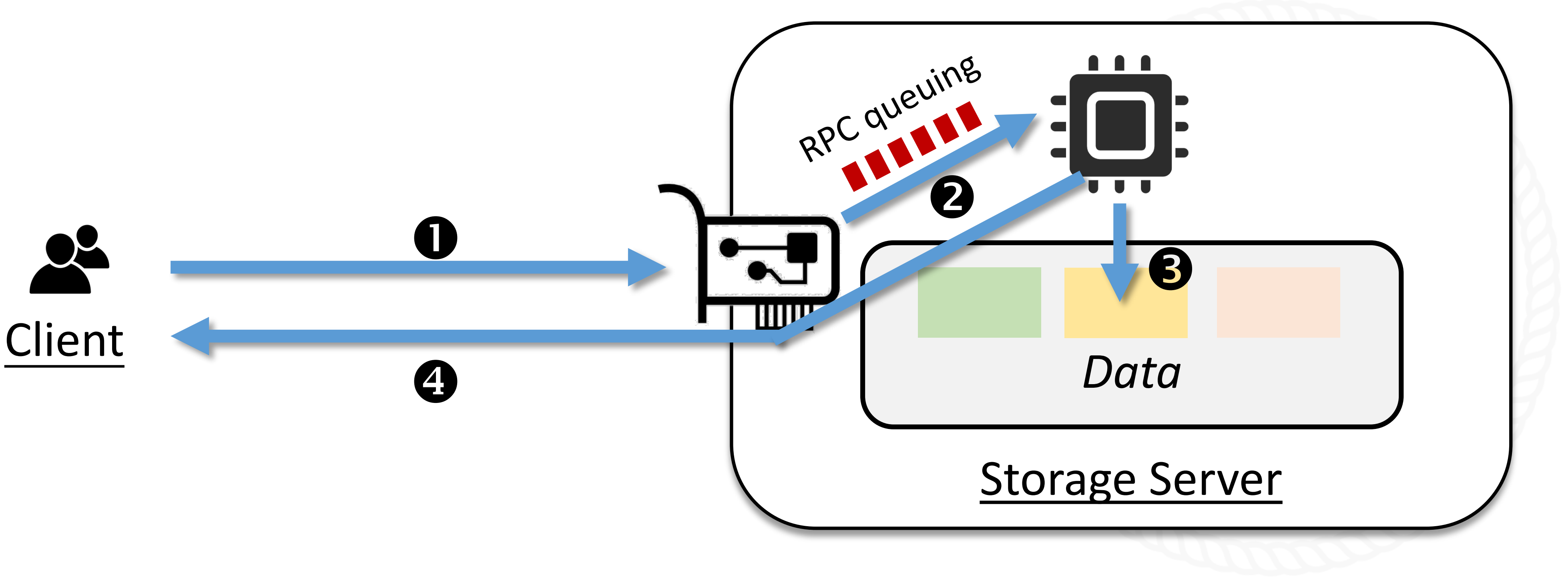
- Clients manipulate data in the storage server connected via network
  - Post requests and obtain responses



We explore how to achieve extremely low latency for storage requests

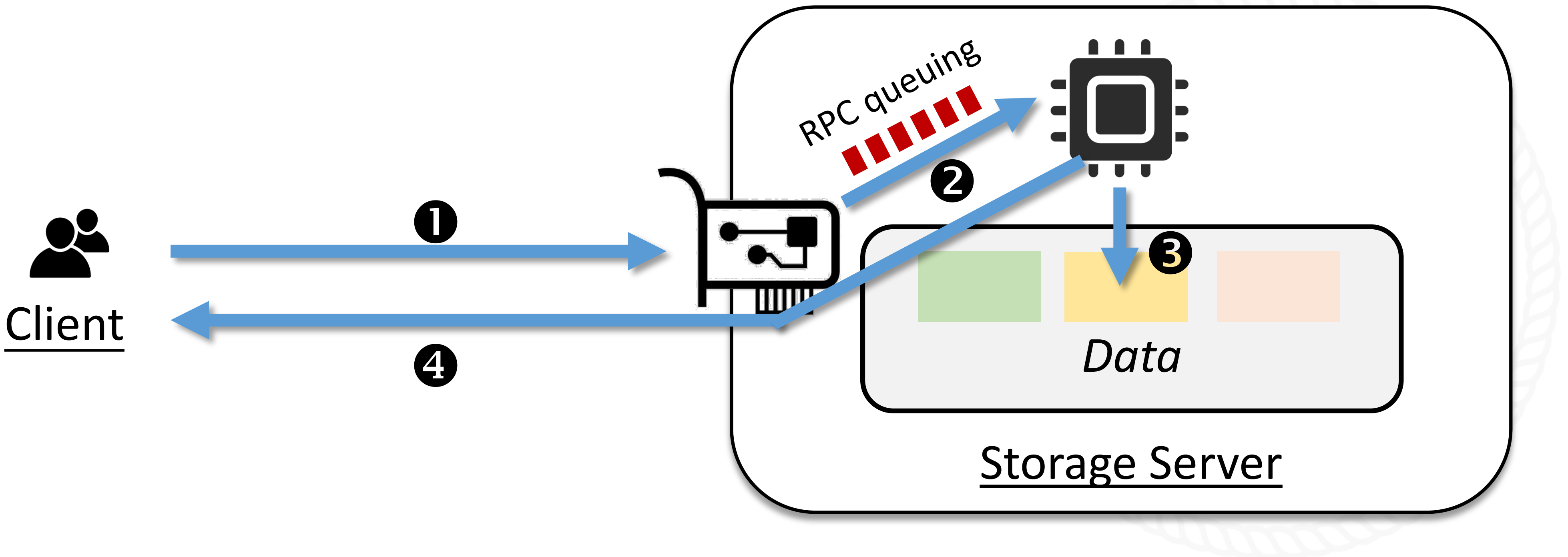
Common paradigm: RPC

- **Client:** sends a request ❶
- **Server:** CPU obtains the request❷, executes it❸, and returns a response❹



Common paradigm: RPC

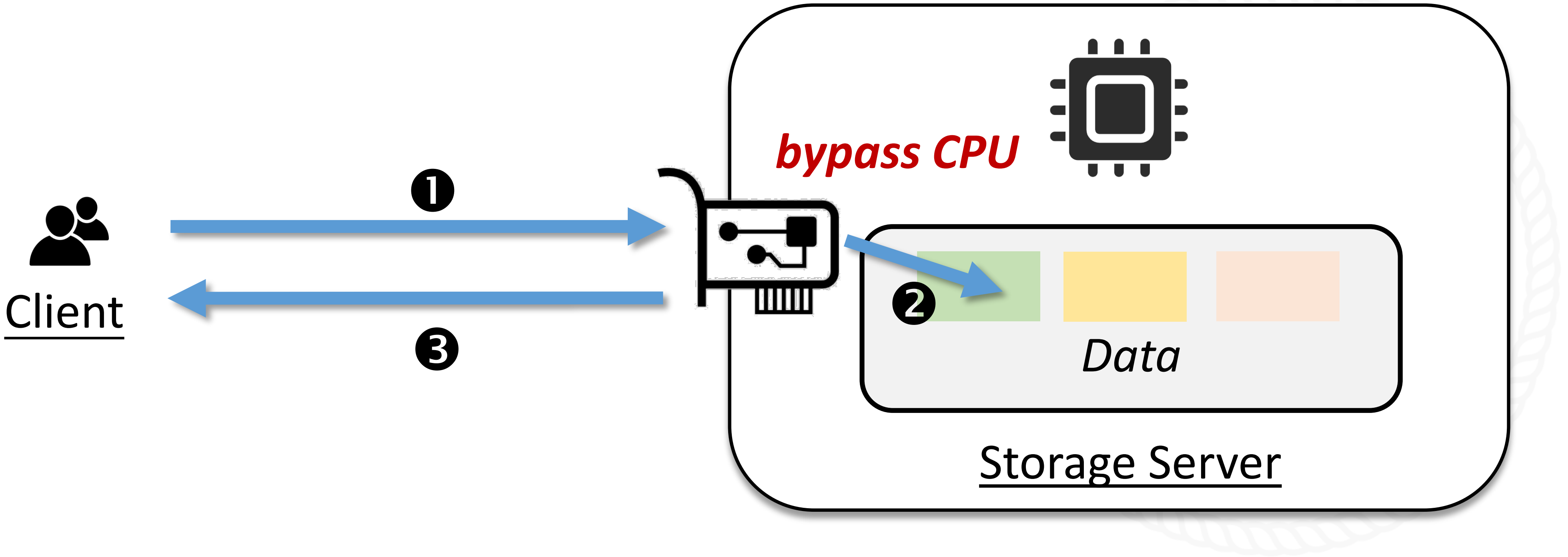
- **Client:** sends a request ❶
- **Server:** CPU obtains the request❷, executes it❸, and returns a response❹



😭 **Server CPU latency in the critical path:** queuing and execution

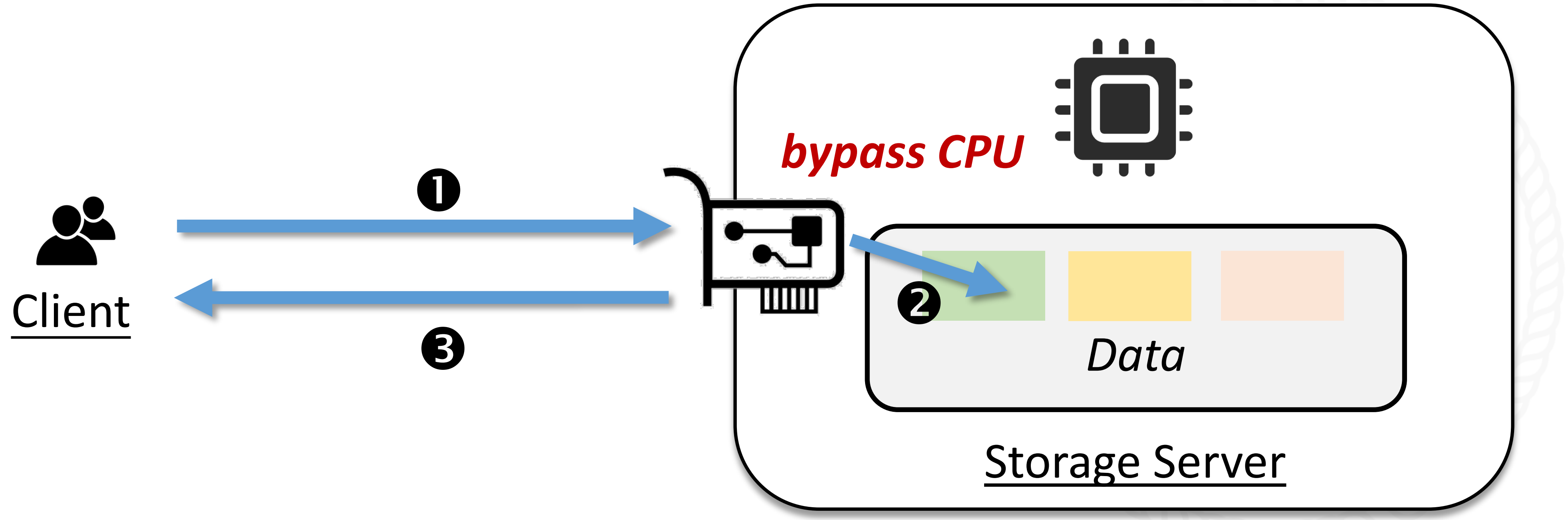
Eliminate server-side CPU latency via one-sided RDMA

- **Client:** sends a request via RDMA write/read ❶
- **Server:** NIC executes write/read ❷, and returns a response ❸



Eliminate server-side CPU latency via one-sided RDMA

- **Client:** sends a request via RDMA write/read ❶
- **Server:** NIC executes write/read ❷, and returns a response ❸

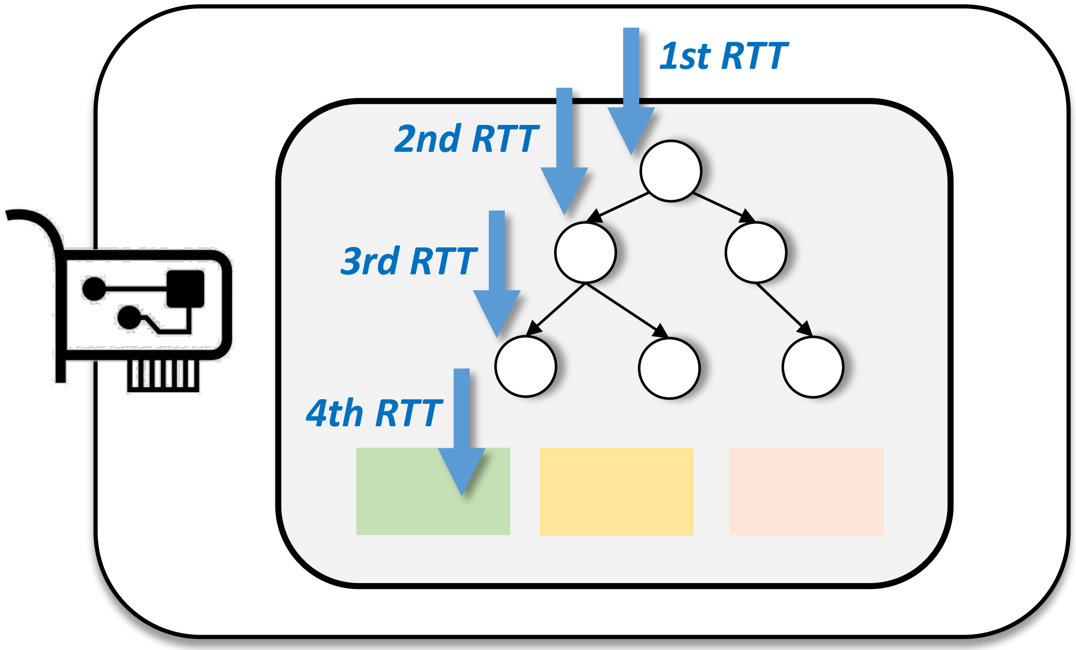


However, one-sided RDMA will induce **multiple RTTs**, offsetting the latency benefits of bypassing remote CPUs

One-sided RDMA will induce multiple RTTs

- Limited semantics of one-sided verbs ⇨ ***Multiple RTTs***
- Need ***code refactoring*** to make data be accessed by one-sided RDMA

***Case 1**: unknown address*

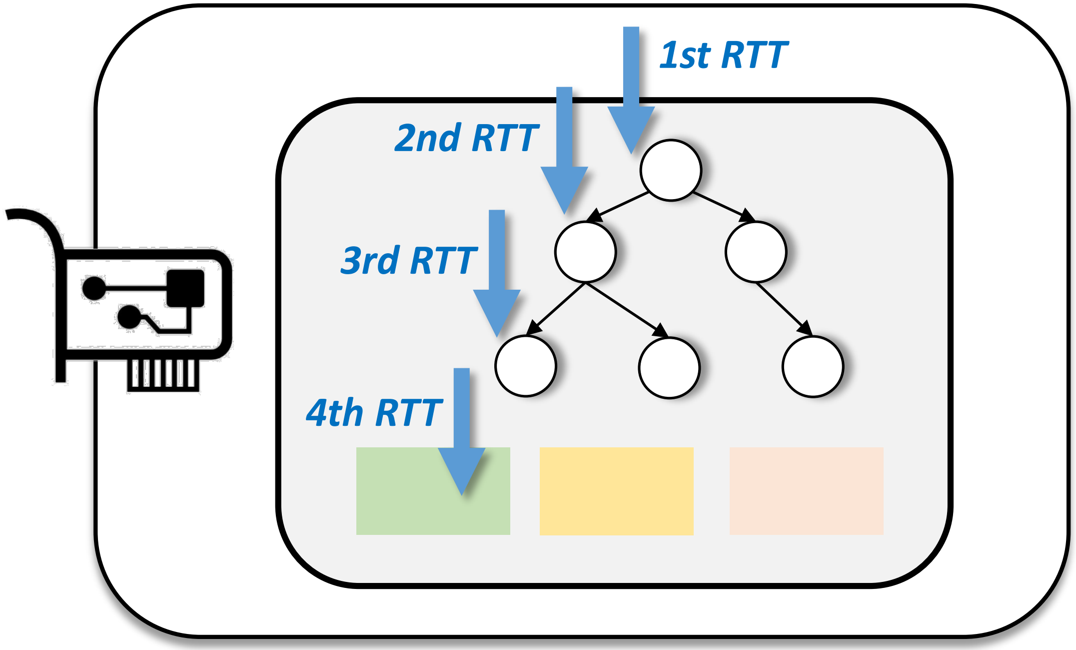


RTTs from index traversal and pointer chasing

One-sided RDMA will induce multiple RTTs

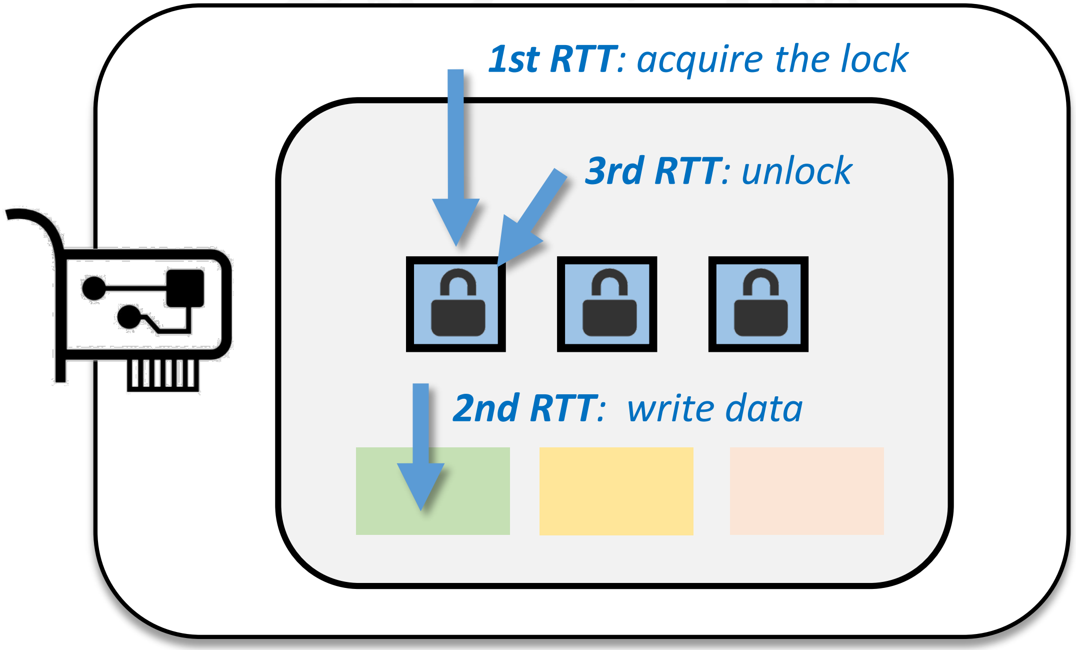
- Limited semantics of one-sided verbs ⇨ ***Multiple RTTs***
- Need ***code refactoring*** to make data be accessed by one-sided RDMA

Case 1: *unknown address*



RTTs from index traversal and pointer chasing

Case 2: *concurrent access*



RTTs from synchronization
(e.g., lock acquire/release/retry)

Our Goal: Wire-latency

Wire-latency: a single round trip **&** server CPU bypass

Like RPC

Like one-sided RDMA

Our Goal: Wire-latency

Wire-latency: a single round trip **&** server CPU bypass

Like RPC

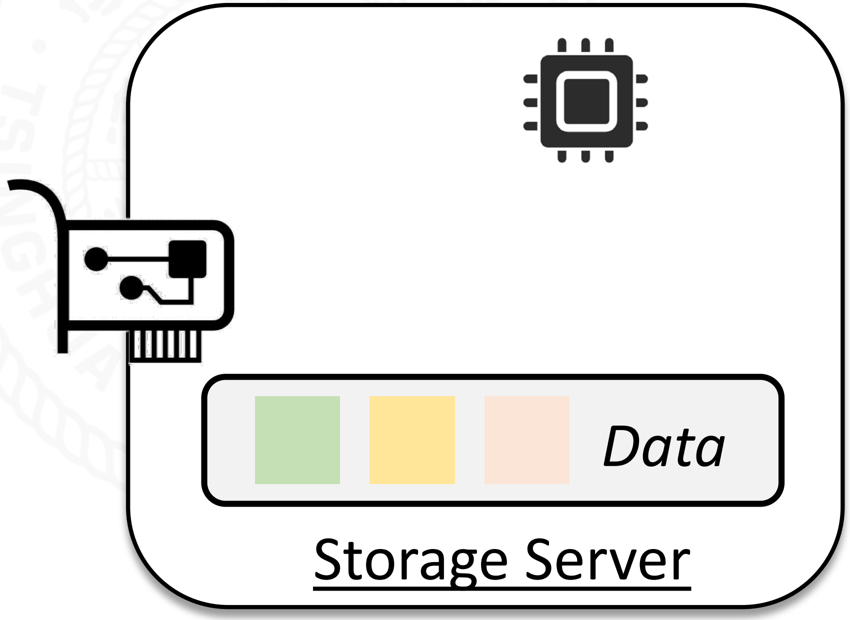
Like one-sided RDMA

Can we achieve wire-latency for general storage requests ?

Basic Idea: Asynchronous execution for nilext requests

Nil-externalizing (or **nilext**) requests [SOSP'19[1], Ganesan et al.]

- Does not externalize its effects or state immediately
- Example: ***set*** in Memcached, ***Put*** in RocksDB
- So we can execute them asynchronously, which makes wire-latency possible



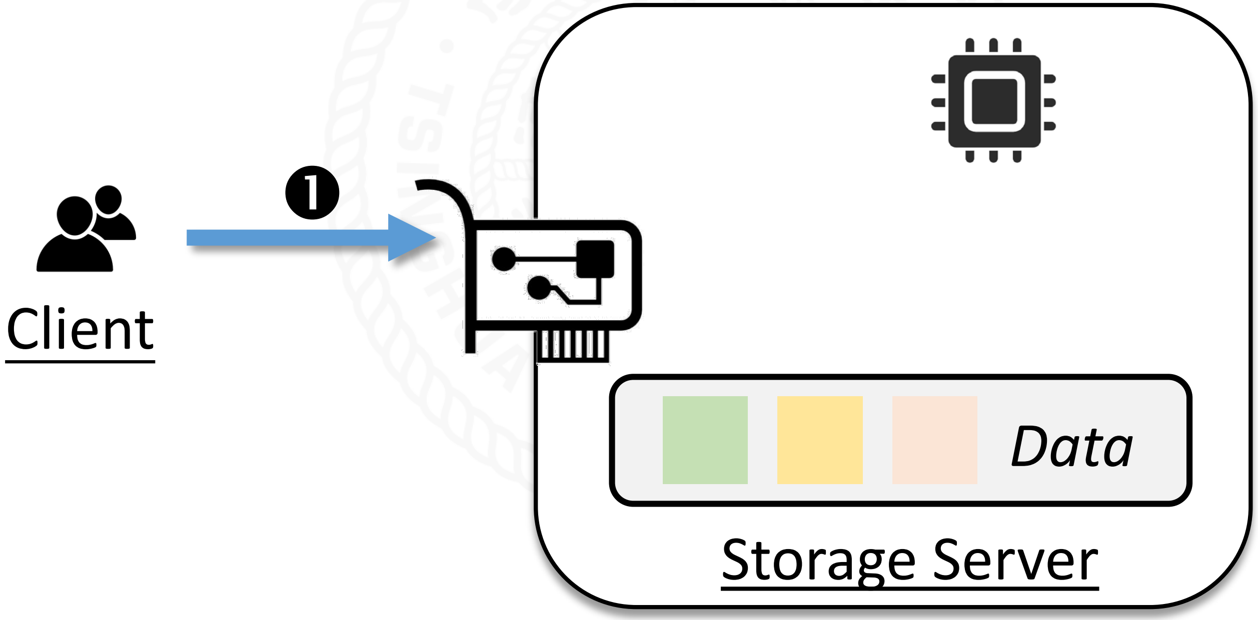
[1] Aishwarya Ganesan, et al. Exploiting Nil-Externality for Fast Replicated Storage, SOSP'21

Basic Idea: Asynchronous execution for nilext requests

Nil-externalizing (or **nilext**) requests [SOSP’19[1], Ganesan et al.]

- ❖ Does not externalize its effects or state immediately
- ❖ Example: ***set*** in Memcached, ***Put*** in RocksDB
- ❖ So we can execute them asynchronously, which makes wire-latency possible

❶ Client issues RDMA_write(set<k, v>)



[1] Aishwarya Ganesan, et al. Exploiting Nil-Externality for Fast Replicated Storage, SOSP’21

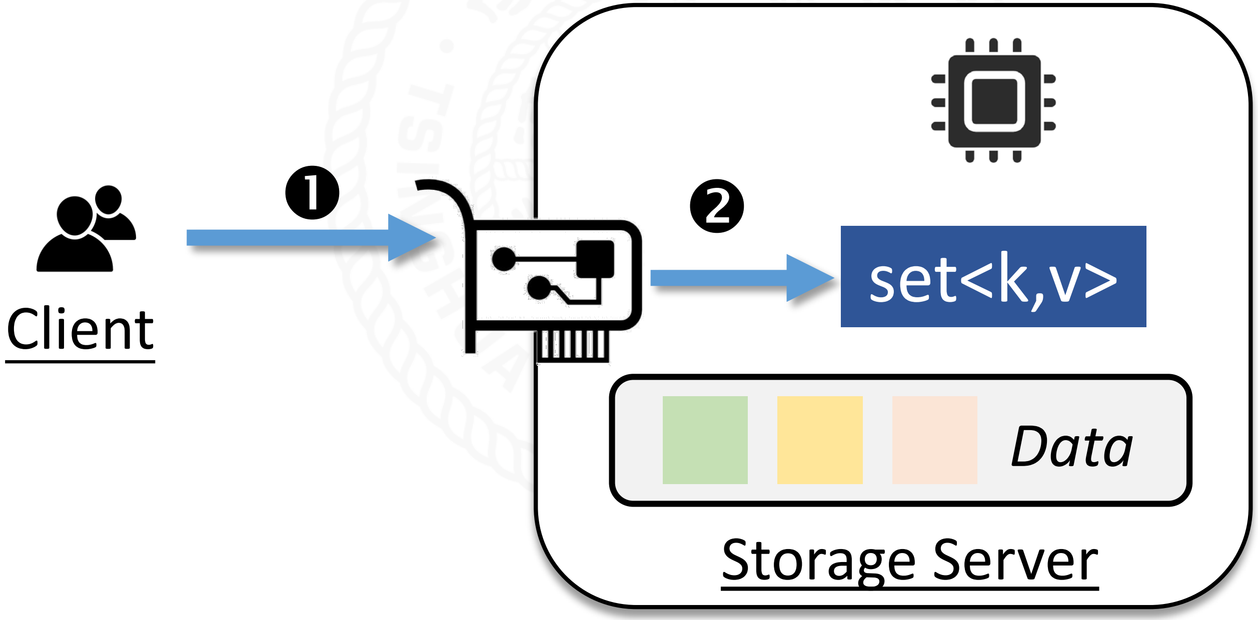
Basic Idea: Asynchronous execution for nilext requests

Nil-externalizing (or **nilext**) requests [SOSP'19[1], Ganesan et al.]

- ❖ Does not externalize its effects or state immediately
- ❖ Example: ***set*** in Memcached, ***Put*** in RocksDB
- ❖ So we can execute them asynchronously, which makes wire-latency possible

❶ Client issues RDMA_write(set<k, v>)

❷ Server NIC writes it to mem



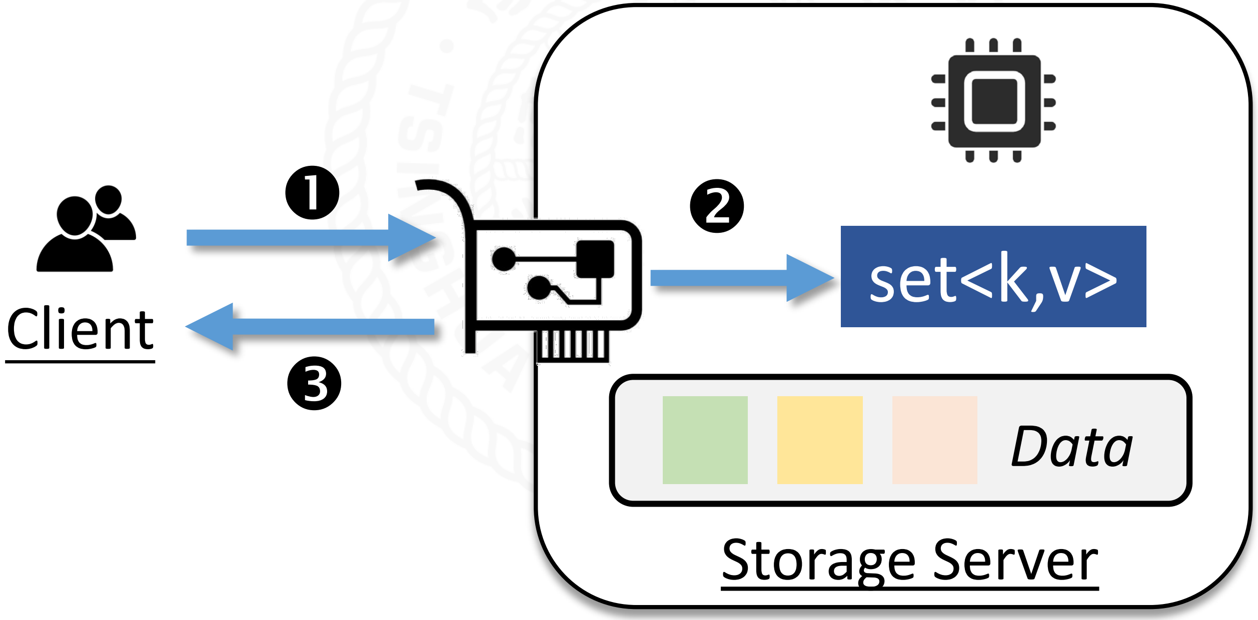
[1] Aishwarya Ganesan, et al. Exploiting Nil-Externality for Fast Replicated Storage, SOSP'21

Basic Idea: Asynchronous execution for nilext requests

Nil-externalizing (or **nilext**) requests [SOSP'19[1], Ganesan et al.]

- ❖ Does not externalize its effects or state immediately
- ❖ Example: ***set*** in Memcached, ***Put*** in RocksDB
- ❖ So we can execute them asynchronously, which makes wire-latency possible

❶ Client issues RDMA_write(set<k, v>)
❷ Server NIC writes it to mem
❸ Server NIC return an ack



[1] Aishwarya Ganesan, et al. Exploiting Nil-Externality for Fast Replicated Storage, SOSP'21

Basic Idea: Asynchronous execution for nilext requests

Nil-externalizing (or **nilext**) requests [SOSP’19[1], Ganesan et al.]

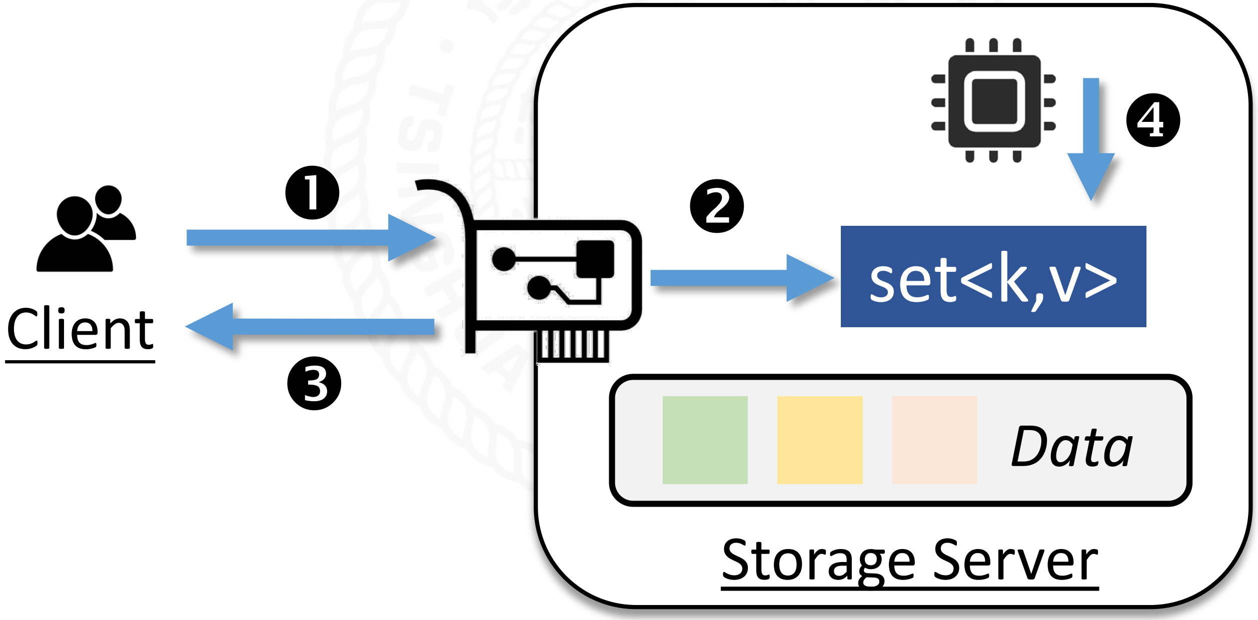
- Does not externalize its effects or state immediately
- Example: ***set*** in Memcached, ***Put*** in RocksDB
- So we can execute them asynchronously, which makes wire-latency possible

❶ Client issues RDMA_write(set<k, v>)

❷ Server NIC writes it to mem

❸ Server NIC return an ack

❹ Server CPU executes it ***async***



[1] Aishwarya Ganesan, et al. Exploiting Nil-Externality for Fast Replicated Storage, SOSP’21

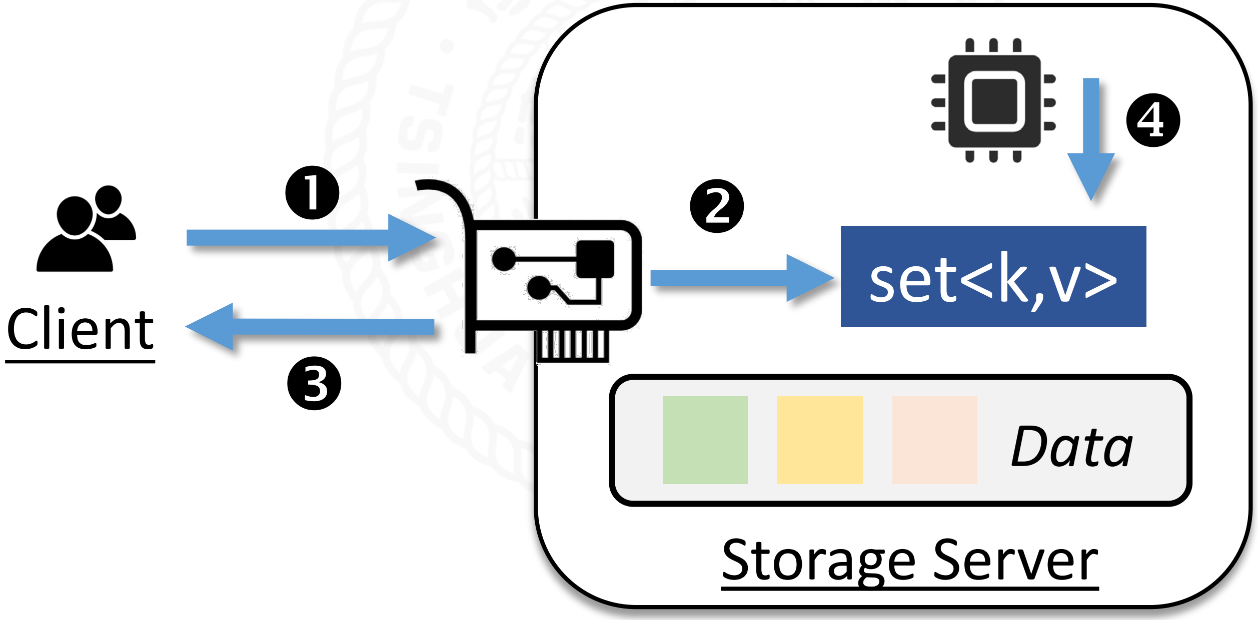
Basic Idea: Asynchronous execution for nilext requests

Nil-externalizing (or **nilext**) requests [SOSP’19[1], Ganesan et al.]

- Does not externalize its effects or state immediately
- Example: ***set*** in Memcached, ***Put*** in RocksDB
- So we can execute them asynchronously, which makes wire-latency possible

❶ Client issues RDMA_write(set<k, v>)
❷ Server NIC writes it to mem
❸ Server NIC return an ack
❹ Server CPU executes it ***async***

Critical path:❶❷❸ ⇨ Wire-latency

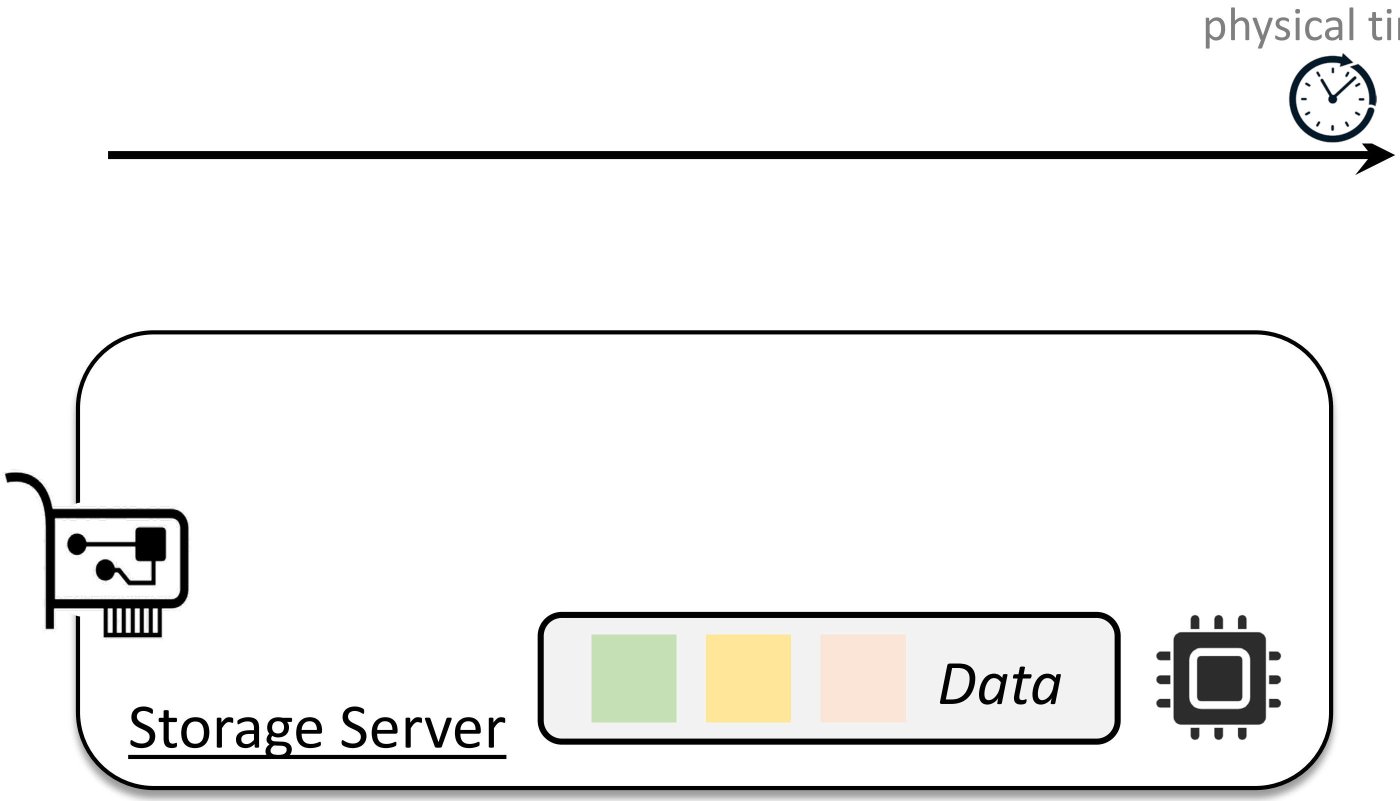


[1] Aishwarya Ganesan, et al. Exploiting Nil-Externality for Fast Replicated Storage, SOSP’21

Challenge: Linearizability (1)

How to maintain linearizability in the presence of multiple clients ?

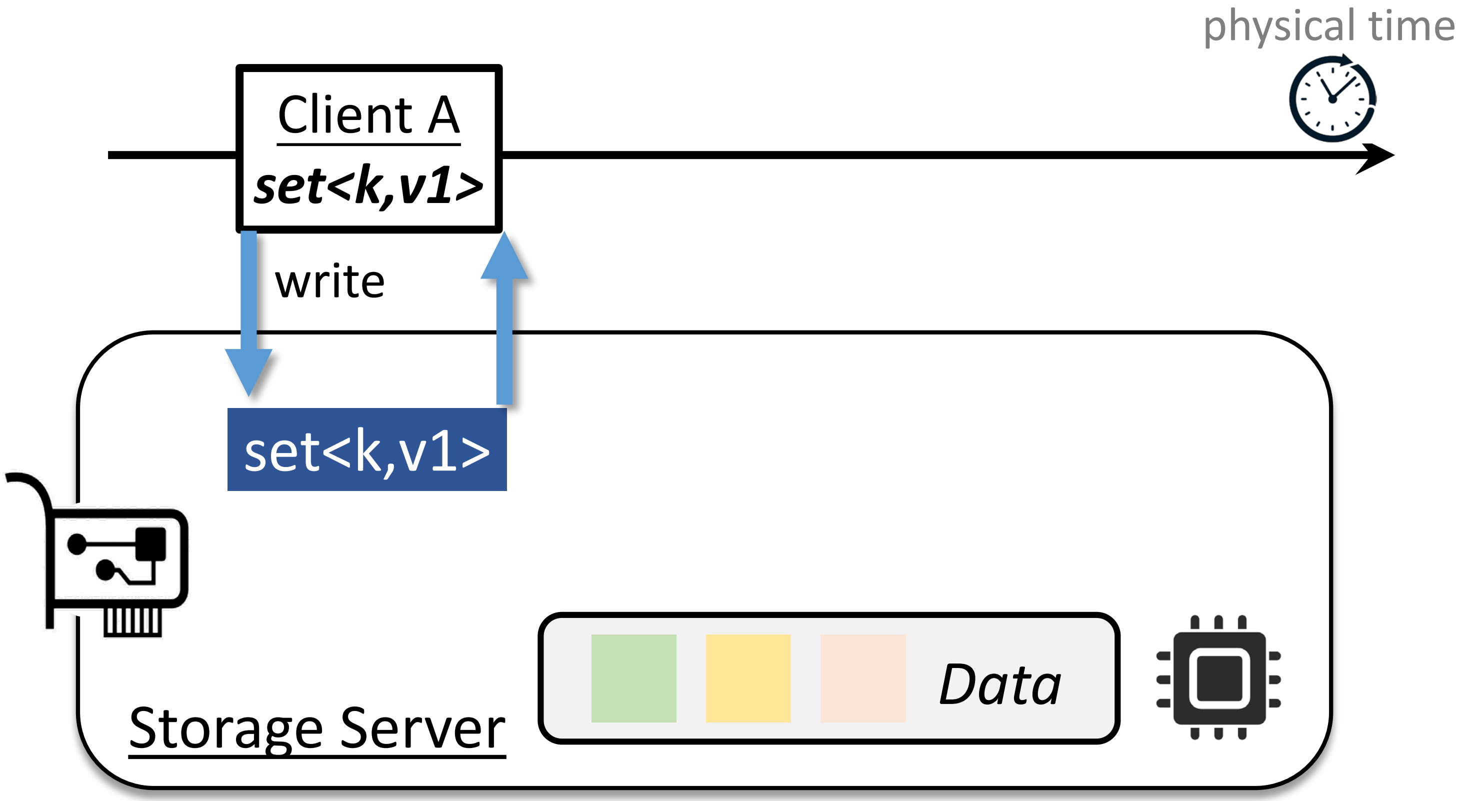
- ❖ The execution of nilext requests is asynchronous
- ❖ The responses of nilext requests are generated by NICs



Challenge: Linearizability (1)

How to maintain linearizability in the presence of multiple clients ?

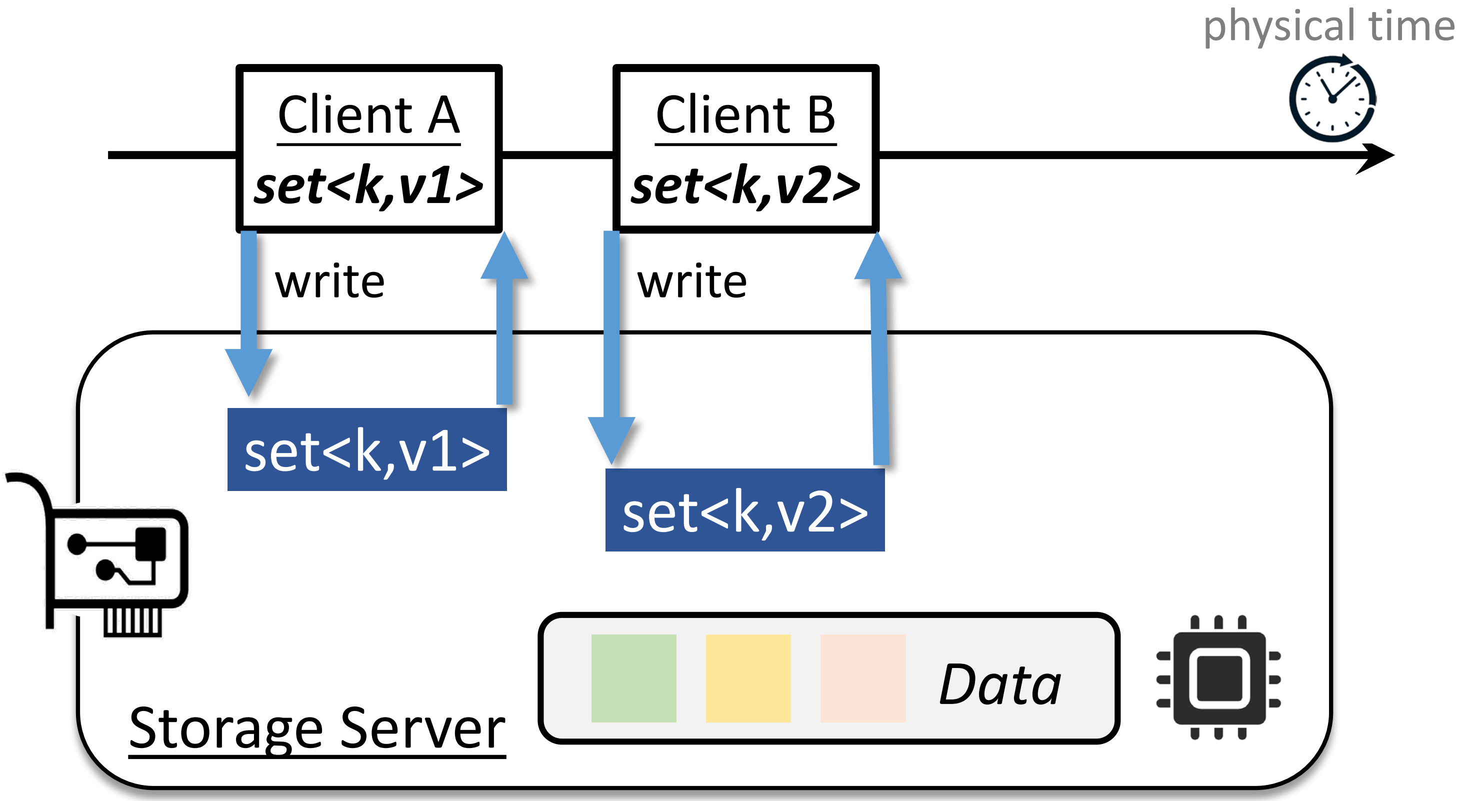
- ❖ The execution of nilext requests is asynchronous
- ❖ The responses of nilext requests are generated by NICs



Challenge: Linearizability (1)

How to maintain linearizability in the presence of multiple clients ?

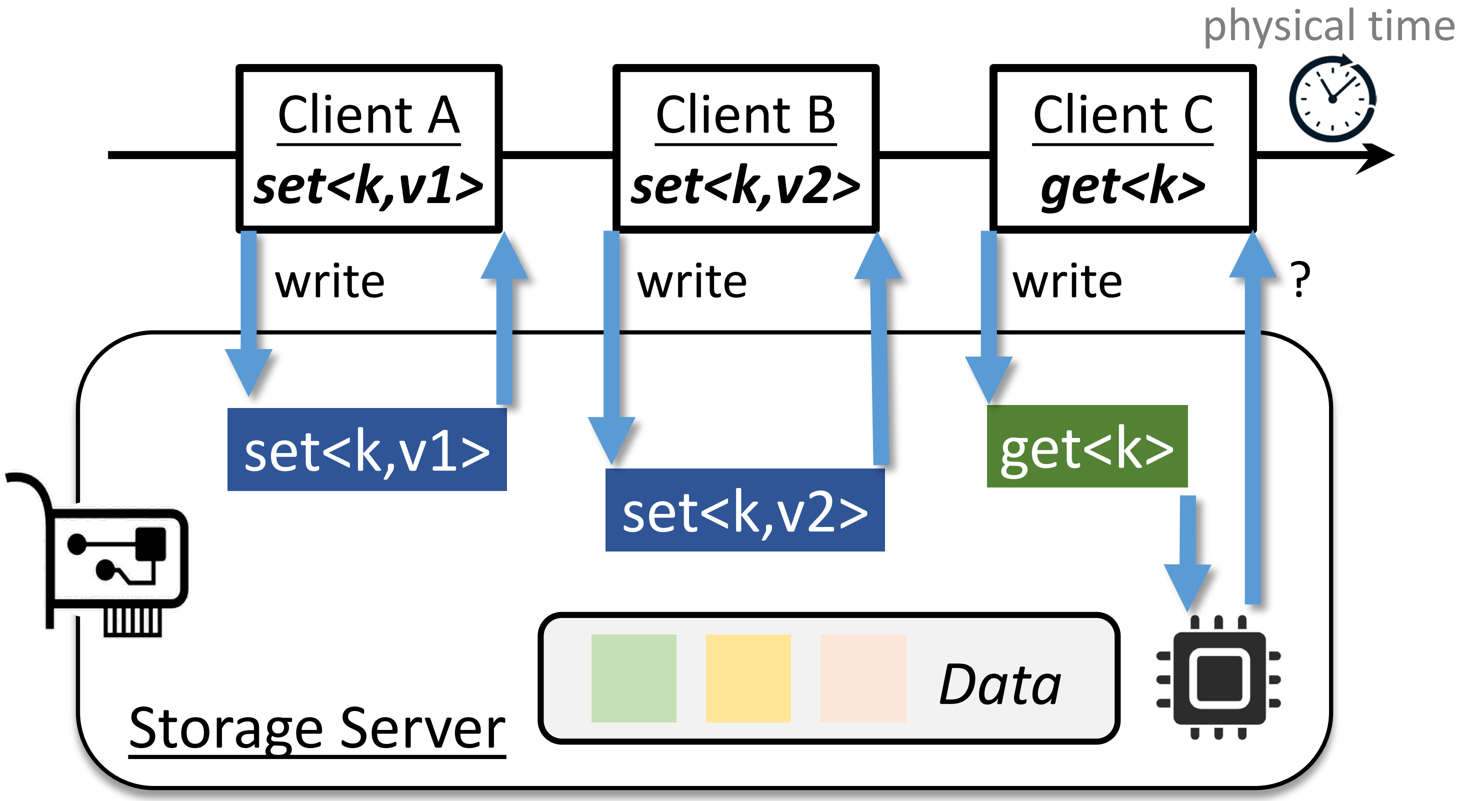
- ❖ The execution of nilext requests is asynchronous
- ❖ The responses of nilext requests are generated by NICs



Challenge: Linearizability (1)

How to maintain linearizability in the presence of multiple clients ?

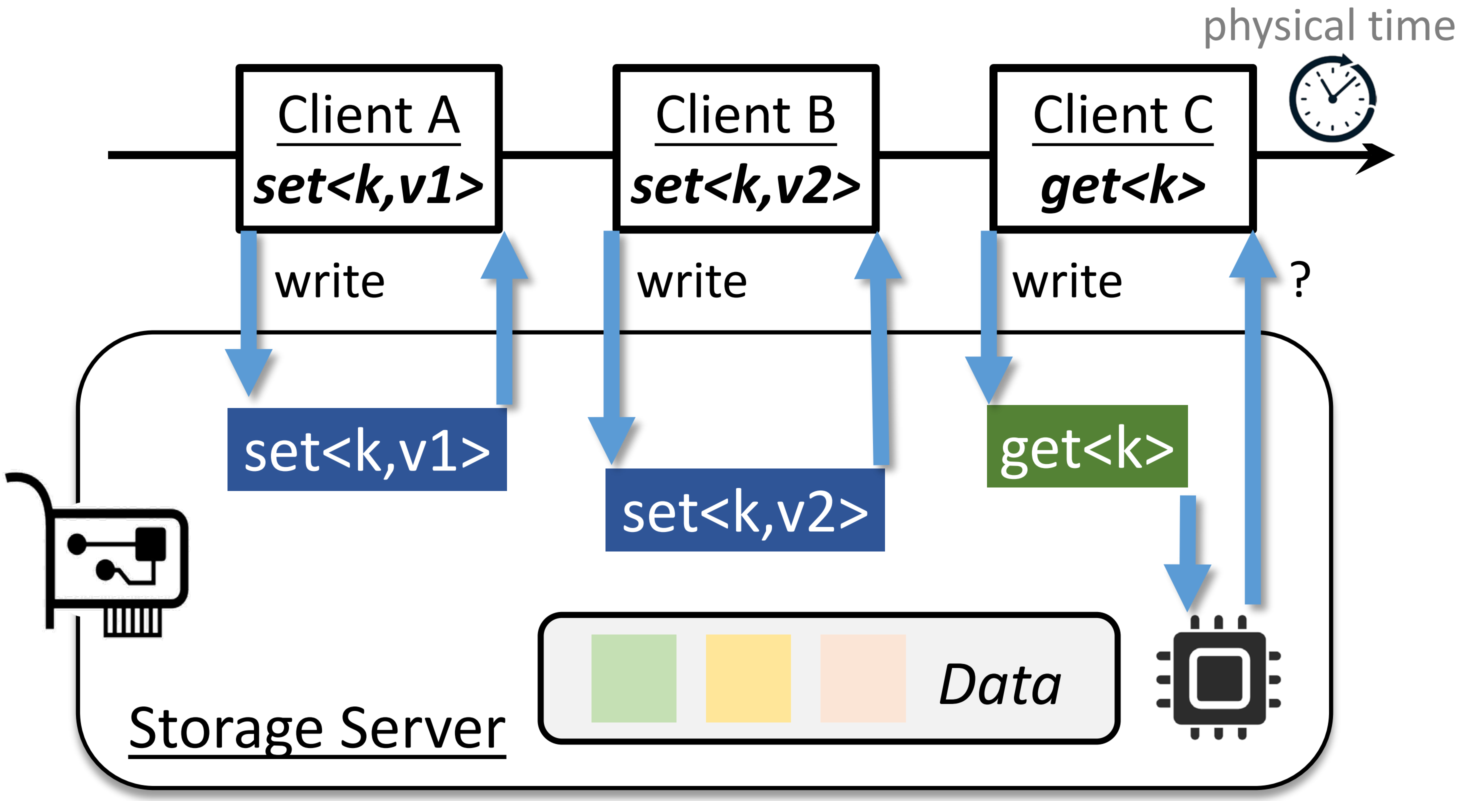
- The execution of nilext requests is asynchronous
- The responses of nilext requests are generated by NICs



Challenge: Linearizability (1)

How to maintain linearizability in the presence of multiple clients ?

- ❖ The execution of nilext requests is asynchronous
- ❖ The responses of nilext requests are generated by NICs



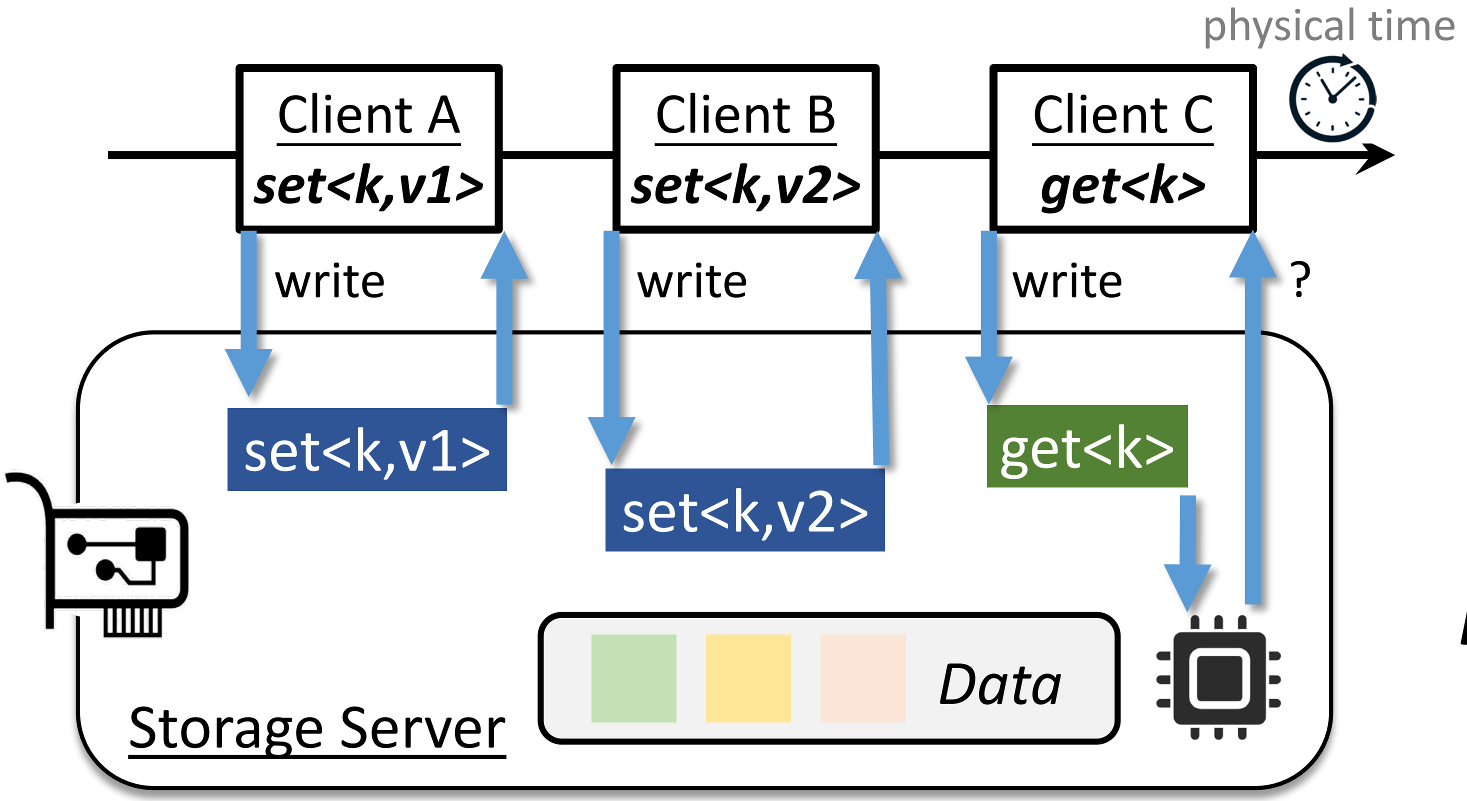
Correct linearizable order:

set<k,v1> | set<k,v2> | get<k>

Challenge: Linearizability (1)

How to maintain linearizability in the presence of multiple clients ?

- The execution of nilext requests is asynchronous
- The responses of nilext requests are generated by NICs



Correct linearizable order:

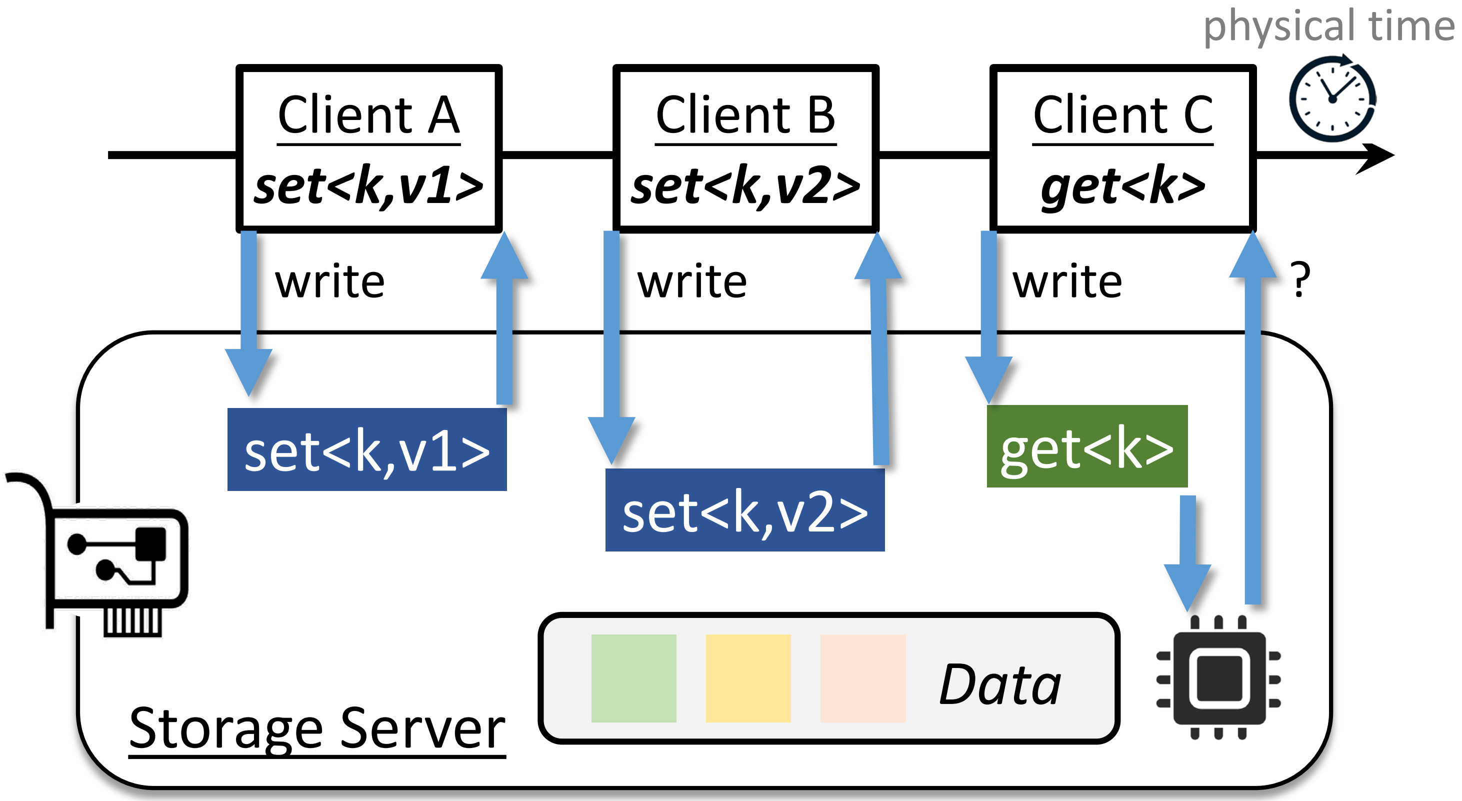
set<k,v1>	set<k,v2>	get<k>

How does the server CPU know this order and execute accordingly?

Challenge: Linearizability (2)

RDMA write lacks the ability of inter-client coordination

- Clients specify the destination address
- Server CPU does not have enough information to obtain the linearizable order



Correct linearizable order:

set<k,v1> set<k,v2> get<k>

Observation: The key of remote CPU bypass is hardware ACKs

What we talk about when we talk about **remote CPU bypass**?

- RDMA write: when polling the completion signal of an RDMA write, the client can guarantee that the data will reliably reach the server memory

Observation: The key of remote CPU bypass is hardware ACKs

What we talk about when we talk about **remote CPU bypass**?

- RDMA write: when polling the completion signal of an RDMA write, the client can guarantee that the data will reliably reach the server memory

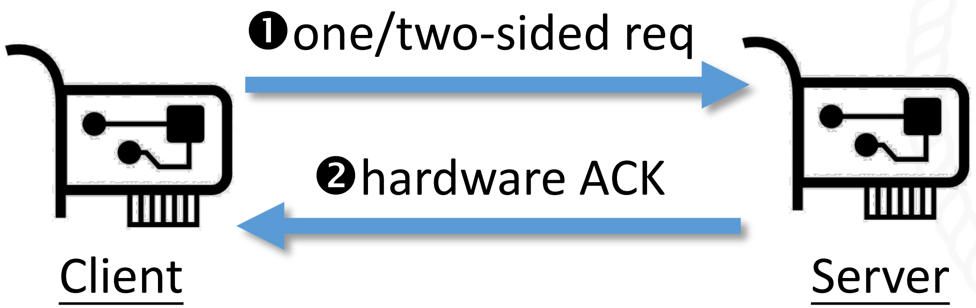
How does the client obtain the guarantee in a remote-CPU-bypass manner?

Observation: The key of remote CPU bypass is hardware ACKs

What we talk about when we talk about **remote CPU bypass**?

- ❖ RDMA write: when polling the completion signal of an RDMA write, the client can guarantee that the data will reliably reach the server memory

How does the client obtain the guarantee in a remote-CPU-bypass manner?

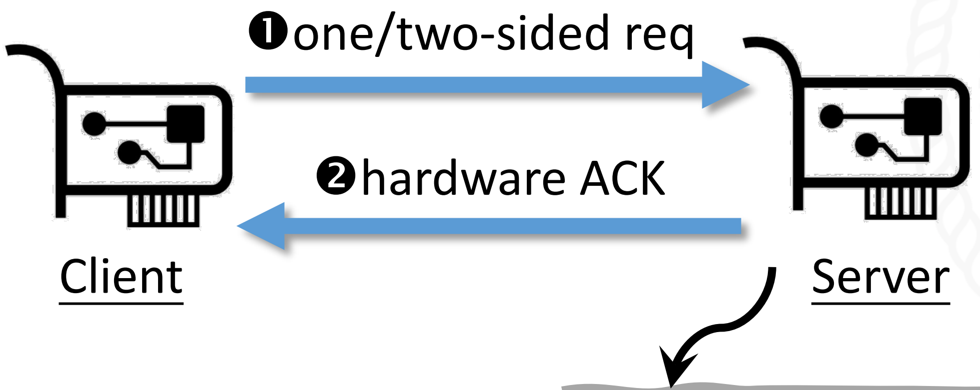


Observation: The key of remote CPU bypass is hardware ACKs

What we talk about when we talk about **remote CPU bypass**?

- RDMA write: when polling the completion signal of an RDMA write, the client can guarantee that the data will reliably reach the server memory

How does the client obtain the guarantee in a remote-CPU-bypass manner?



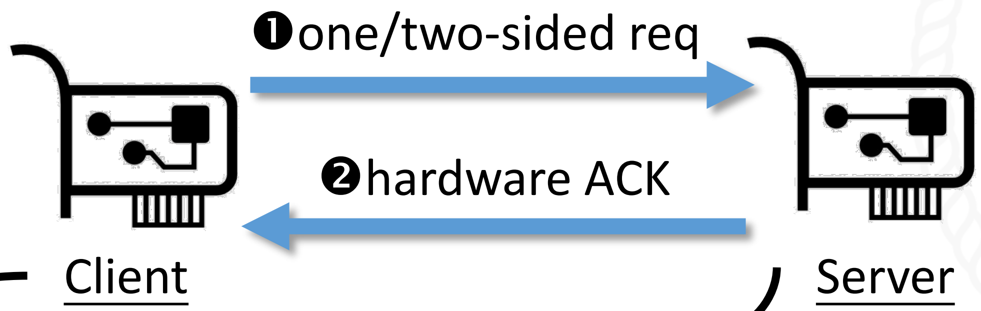
When offloading reliability, server NIC generates ***hardware ACKs*** to confirm safe delivery

Observation: The key of remote CPU bypass is hardware ACKs

What we talk about when we talk about **remote CPU bypass**?

- RDMA write: when polling the completion signal of an RDMA write, the client can guarantee that the data will reliably reach the server memory

How does the client obtain the guarantee in a remote-CPU-bypass manner?



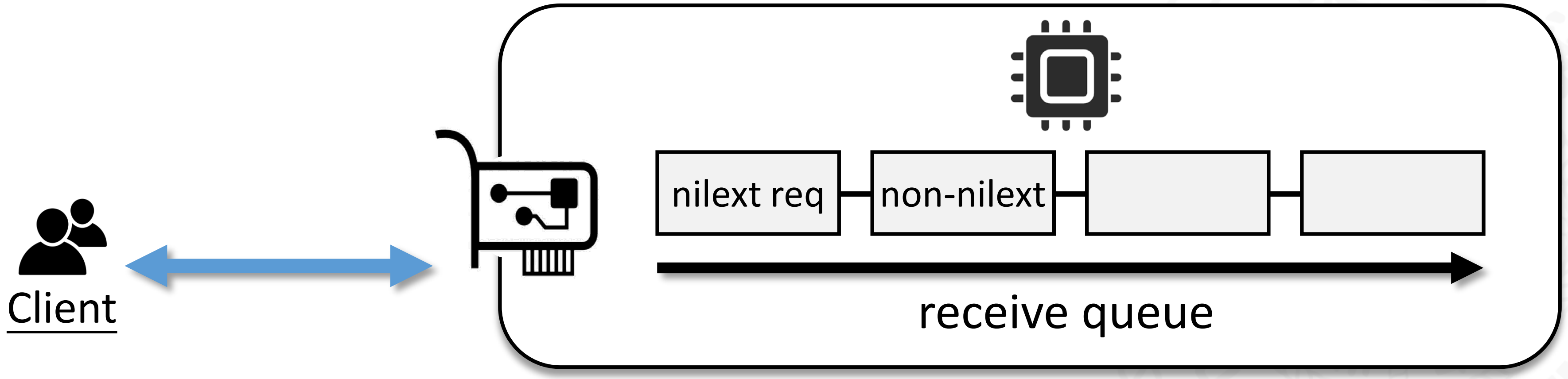
Upon receiving an ACK (via polling CQ), the client can obtain the guarantee

When offloading reliability, server NIC generates ***hardware ACKs*** to confirm safe delivery

Ordered Queue (OQ) Abstraction

Ordered Queue (OQ) Abstraction: enabling wire-latency nilext requests

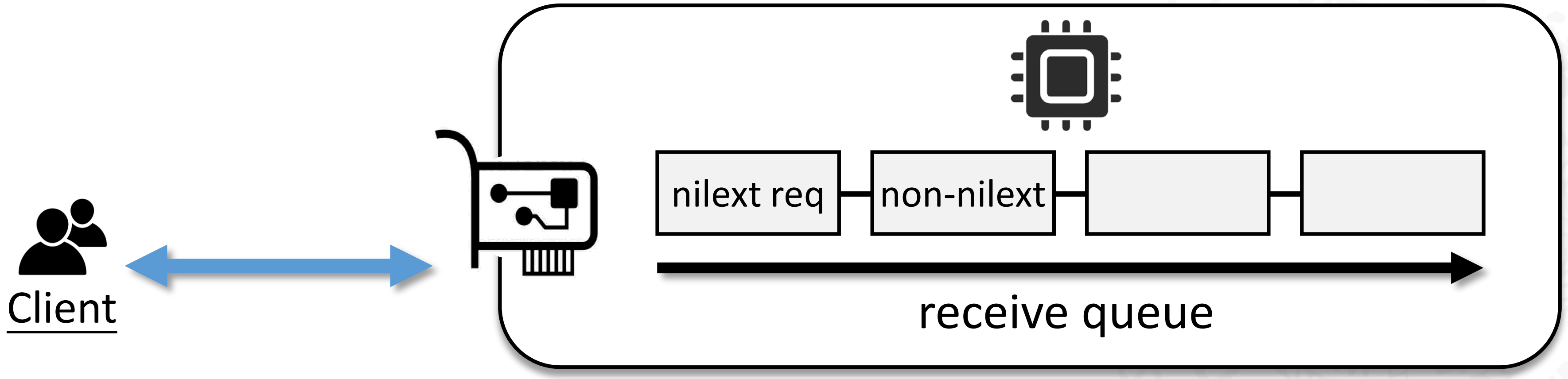
- ❖ Leveraging receive queue (RQ) to produce the linearizable order
- ❖ ***Linearizable order == positions of receive buffers in the RQ***



Ordered Queue (OQ) Abstraction

Ordered Queue (OQ) Abstraction: enabling wire-latency nilext requests

- ❖ Leveraging receive queue (RQ) to produce the linearizable order
- ❖ ***Linearizable order == positions of receive buffers in the RQ***



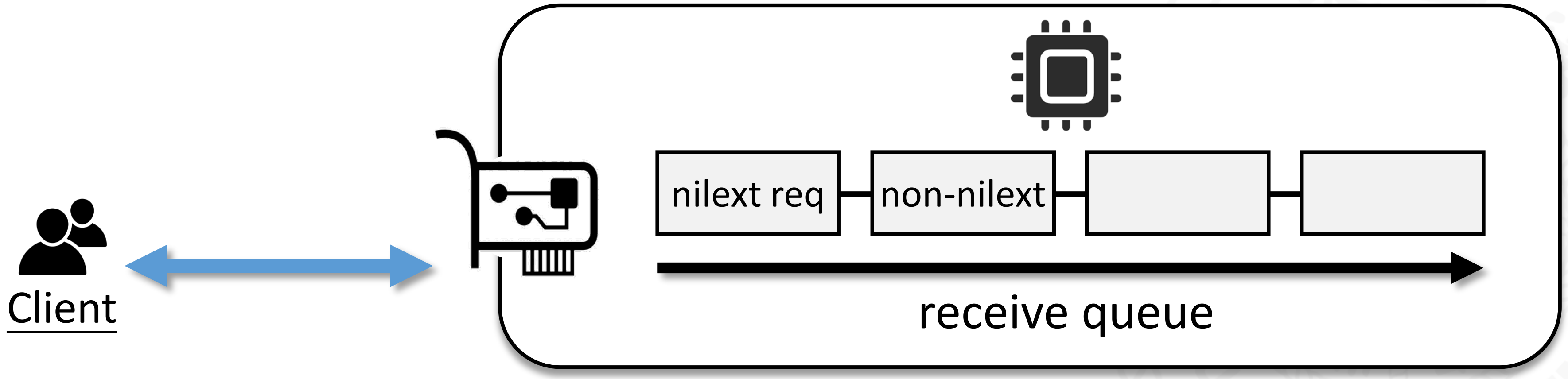
NIC Rule

NIC allocates receive buffers in order and returns hardware ACKs

Ordered Queue (OQ) Abstraction

Ordered Queue (OQ) Abstraction: enabling wire-latency nilext requests

- ❖ Leveraging receive queue (RQ) to produce the linearizable order
- ❖ ***Linearizable order == positions of receive buffers in the RQ***



NIC Rule

NIC allocates receive buffers in order and returns hardware ACKs

CPU Rule

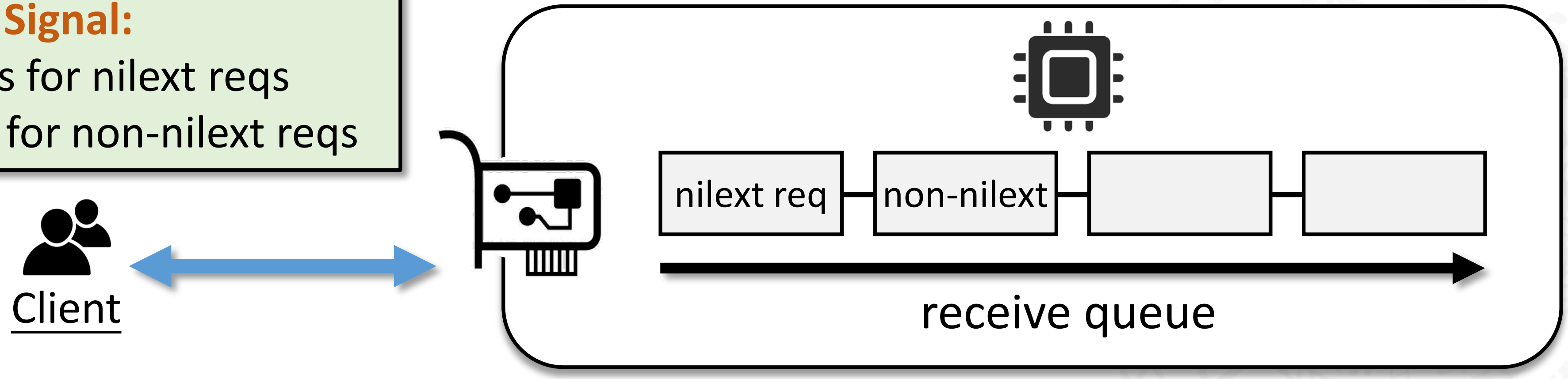
CPU executes requests in order and returns response for non-nilext reqs

Ordered Queue (OQ) Abstraction

Ordered Queue (OQ) Abstraction: enabling wire-latency nilext requests

- Leveraging receive queue (RQ) to produce the linearizable order
- ***Linearizable order == positions of receive buffers in the RQ***

Commit Signal:
HW ACKs for nilext reqs
SW reps for non-nilext reqs



NIC Rule

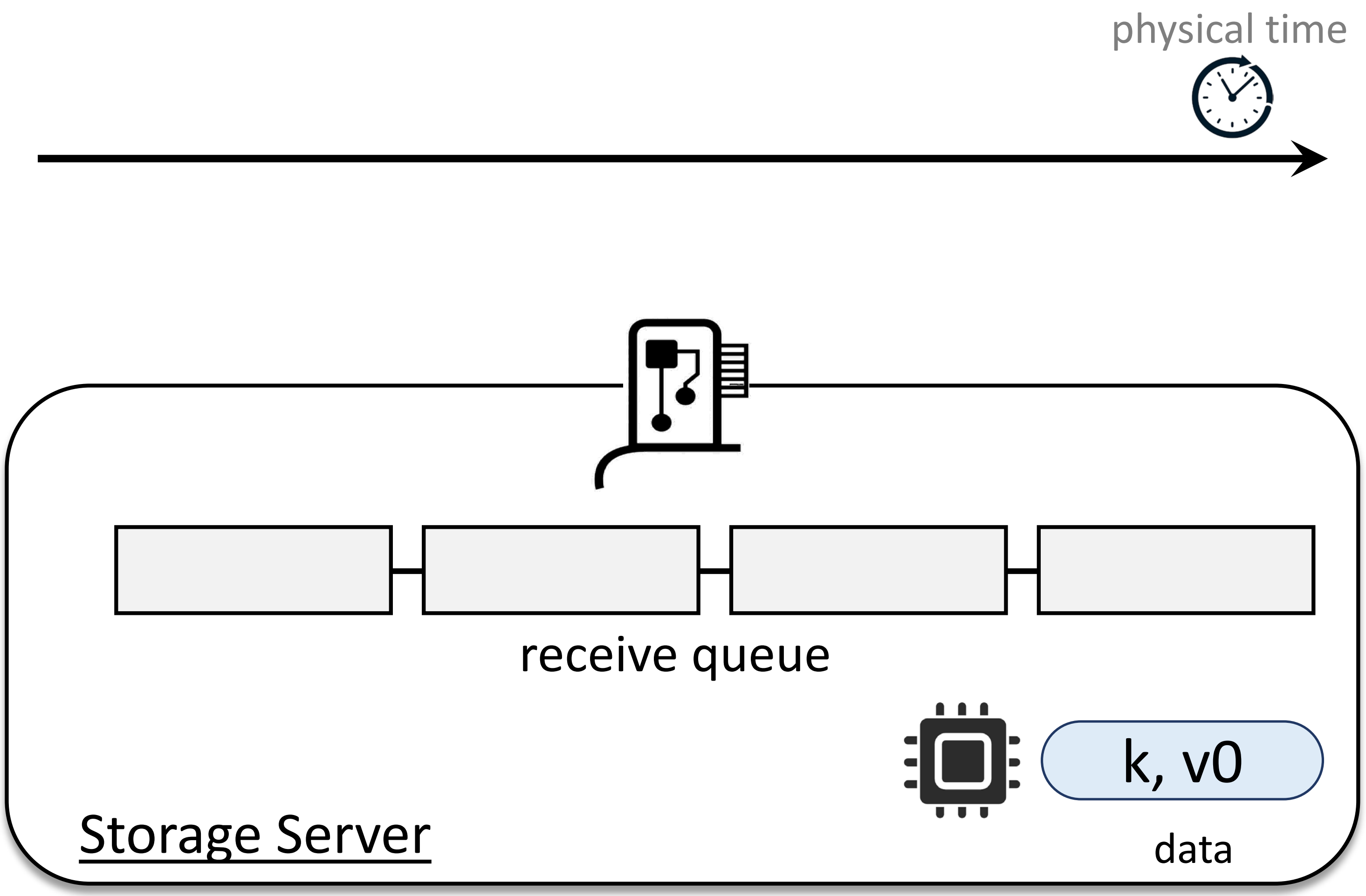
NIC allocates receive buffers in order and returns hardware ACKs

CPU Rule

CPU executes requests in order and returns response for non-nilext reqs

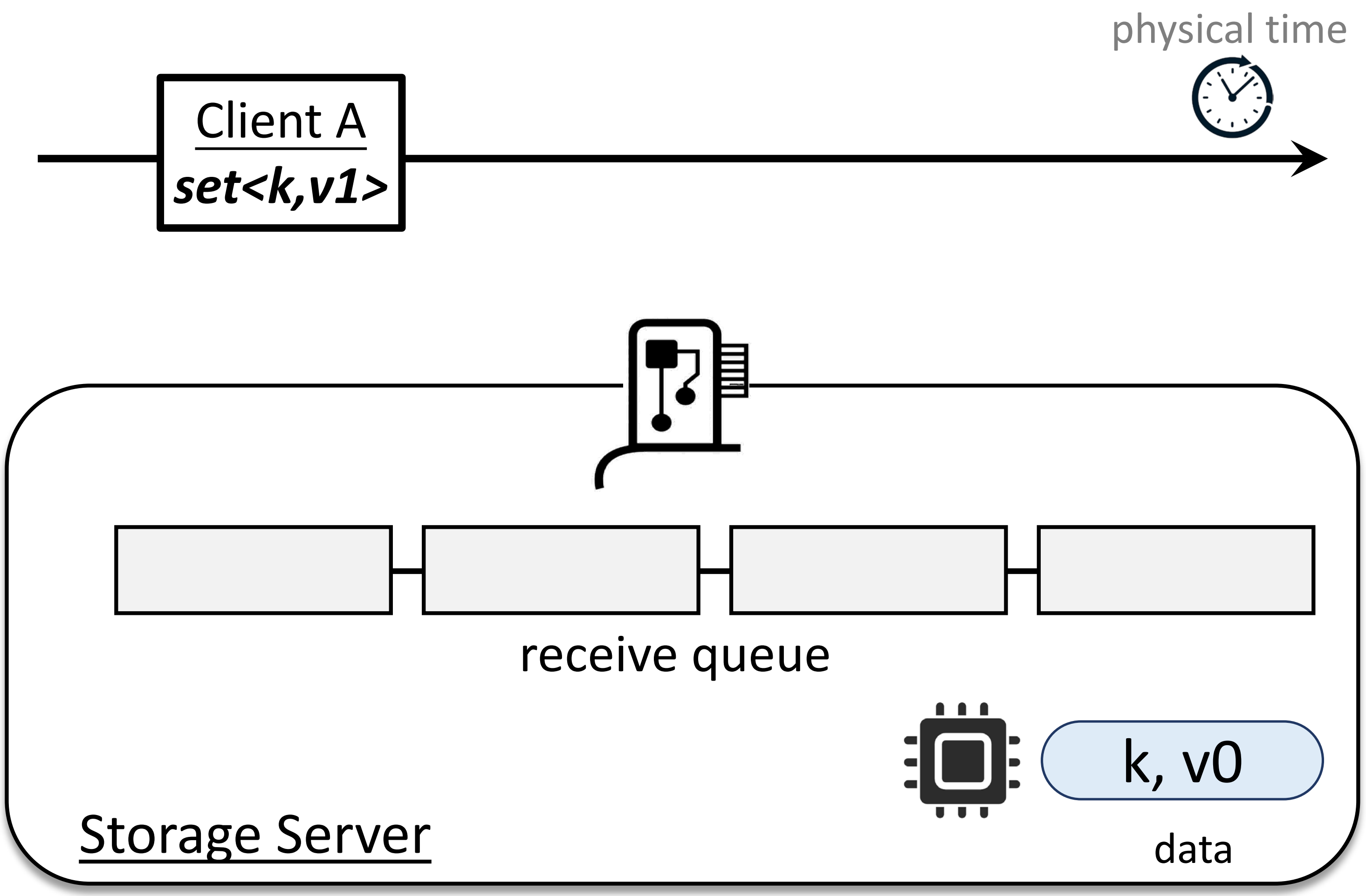
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



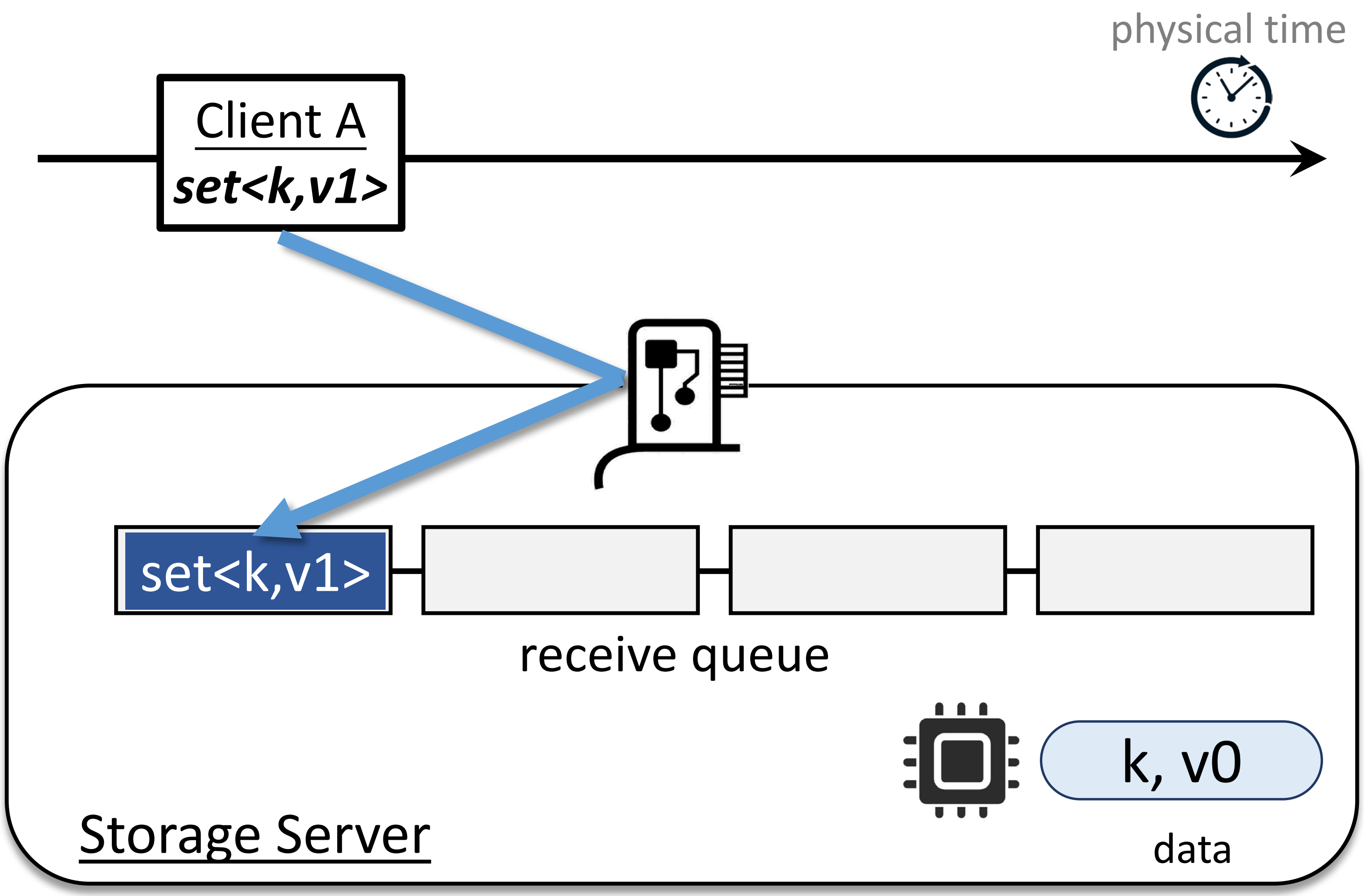
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



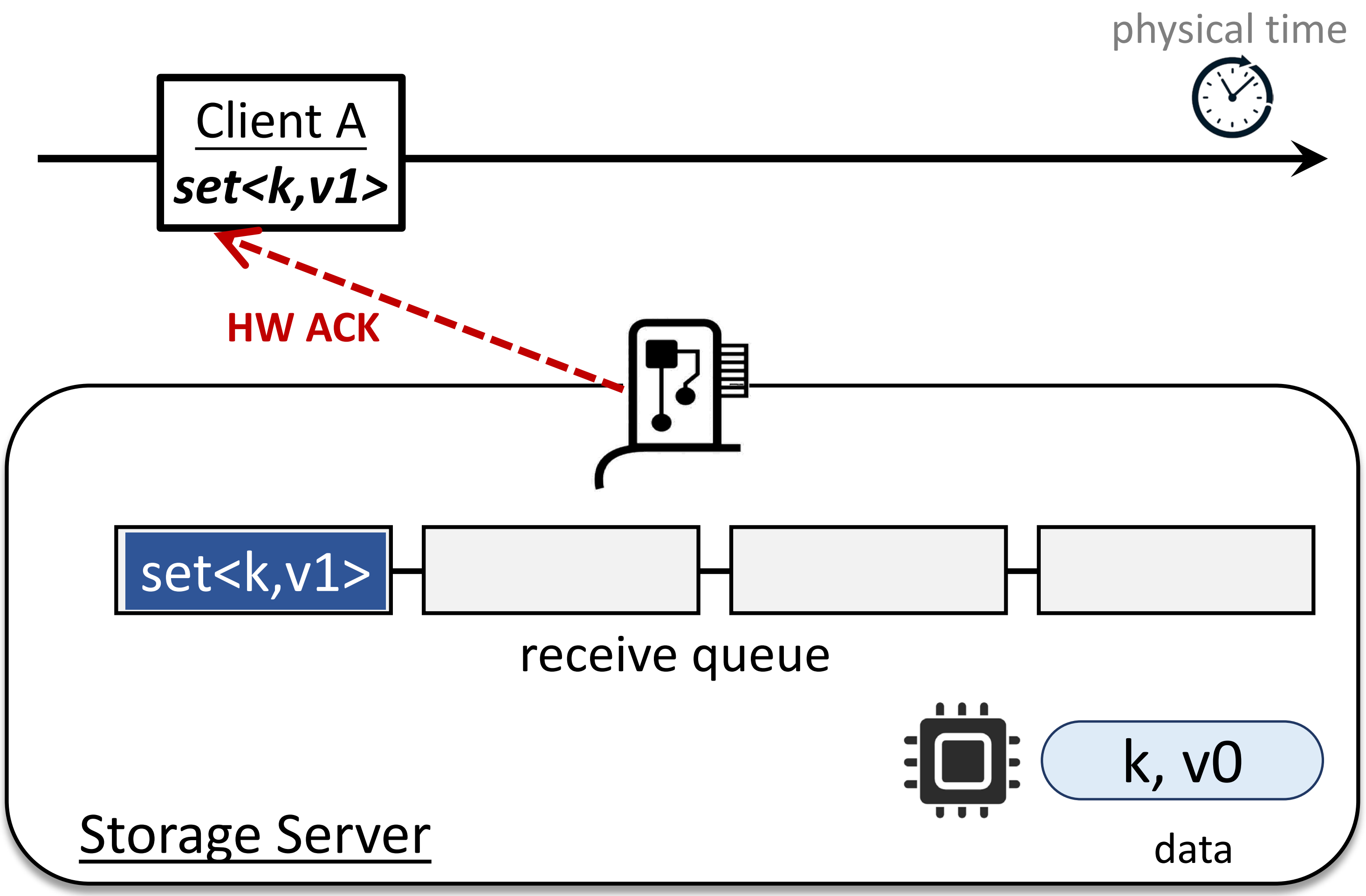
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



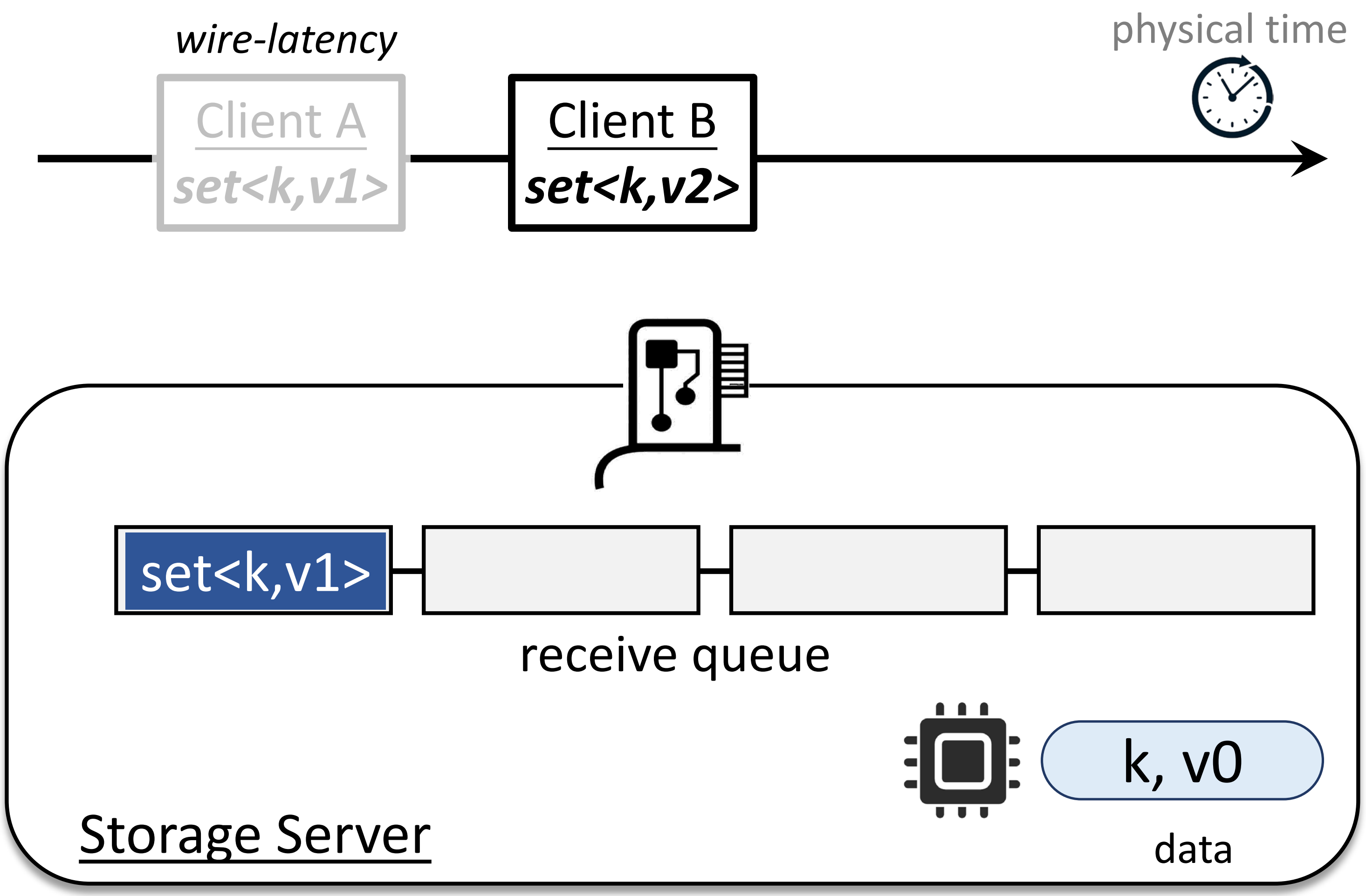
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



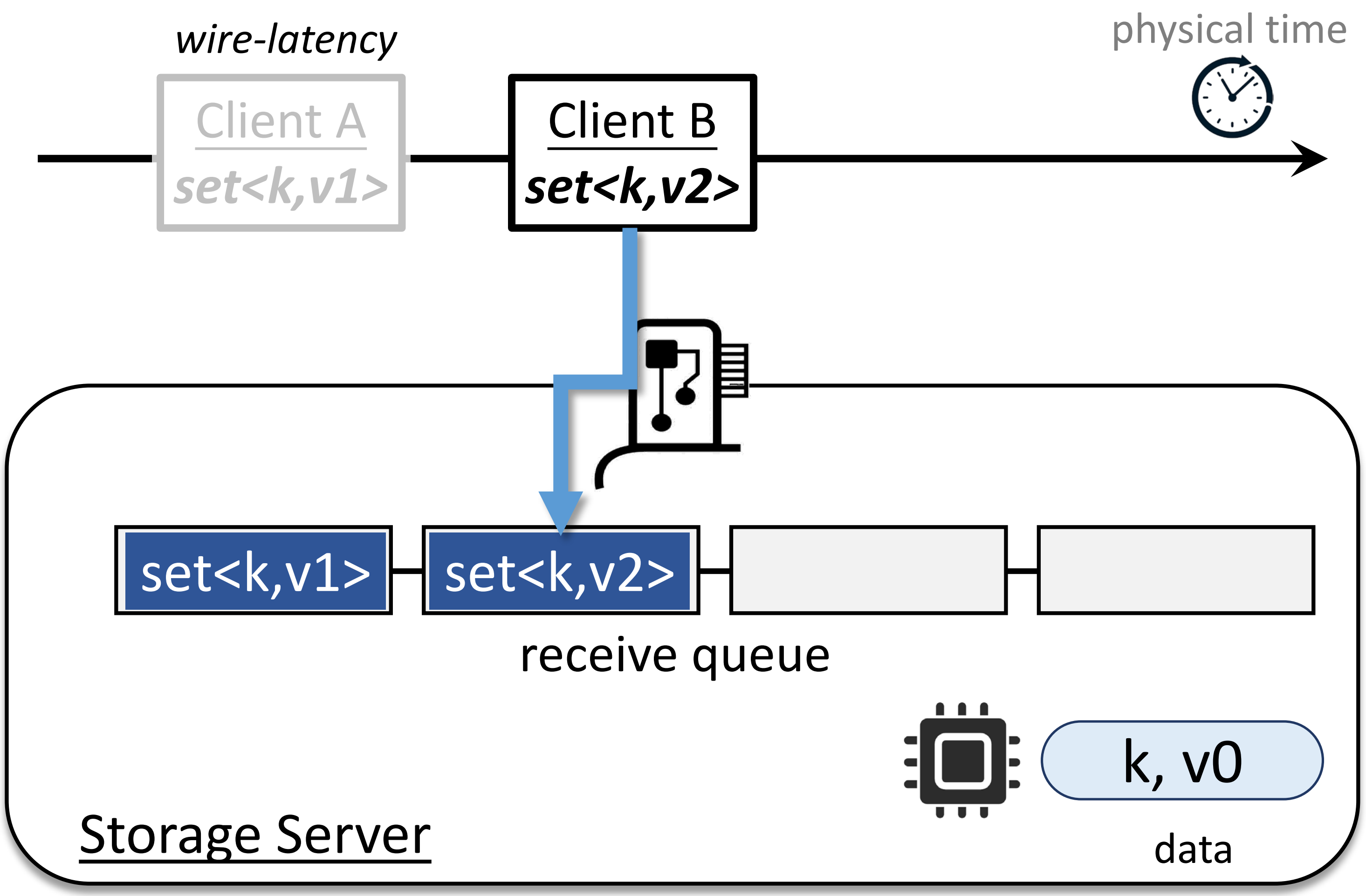
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



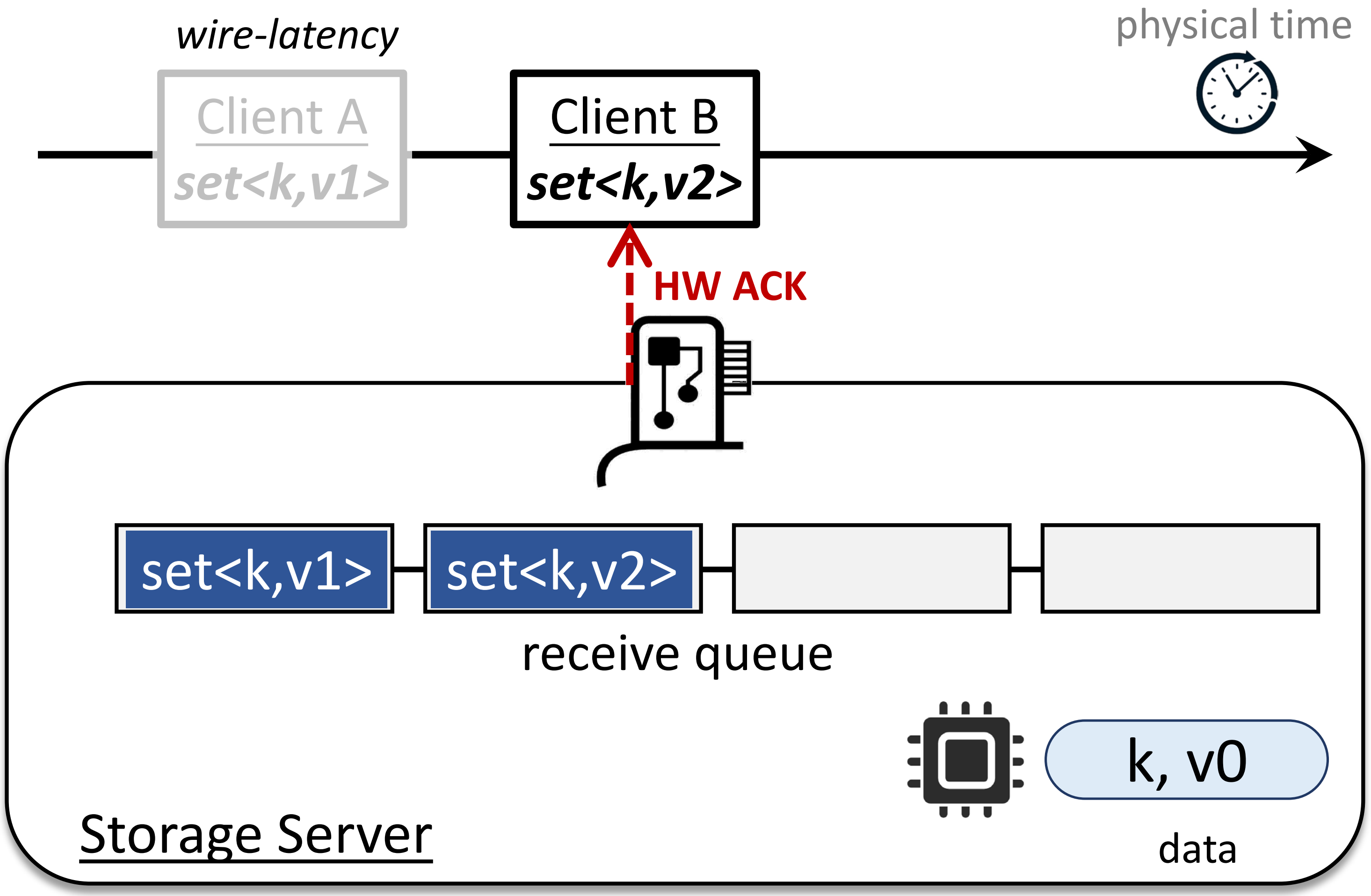
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



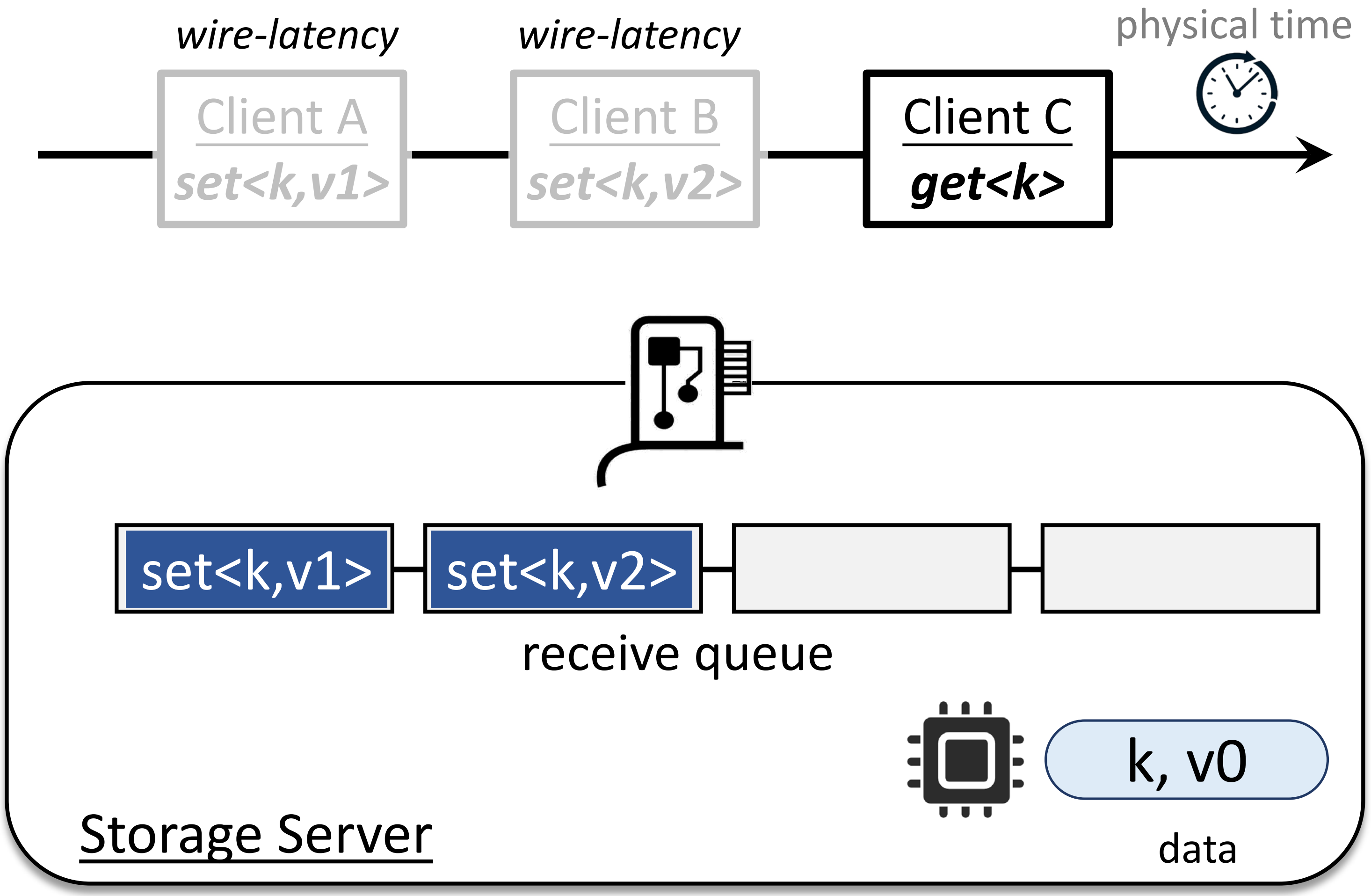
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



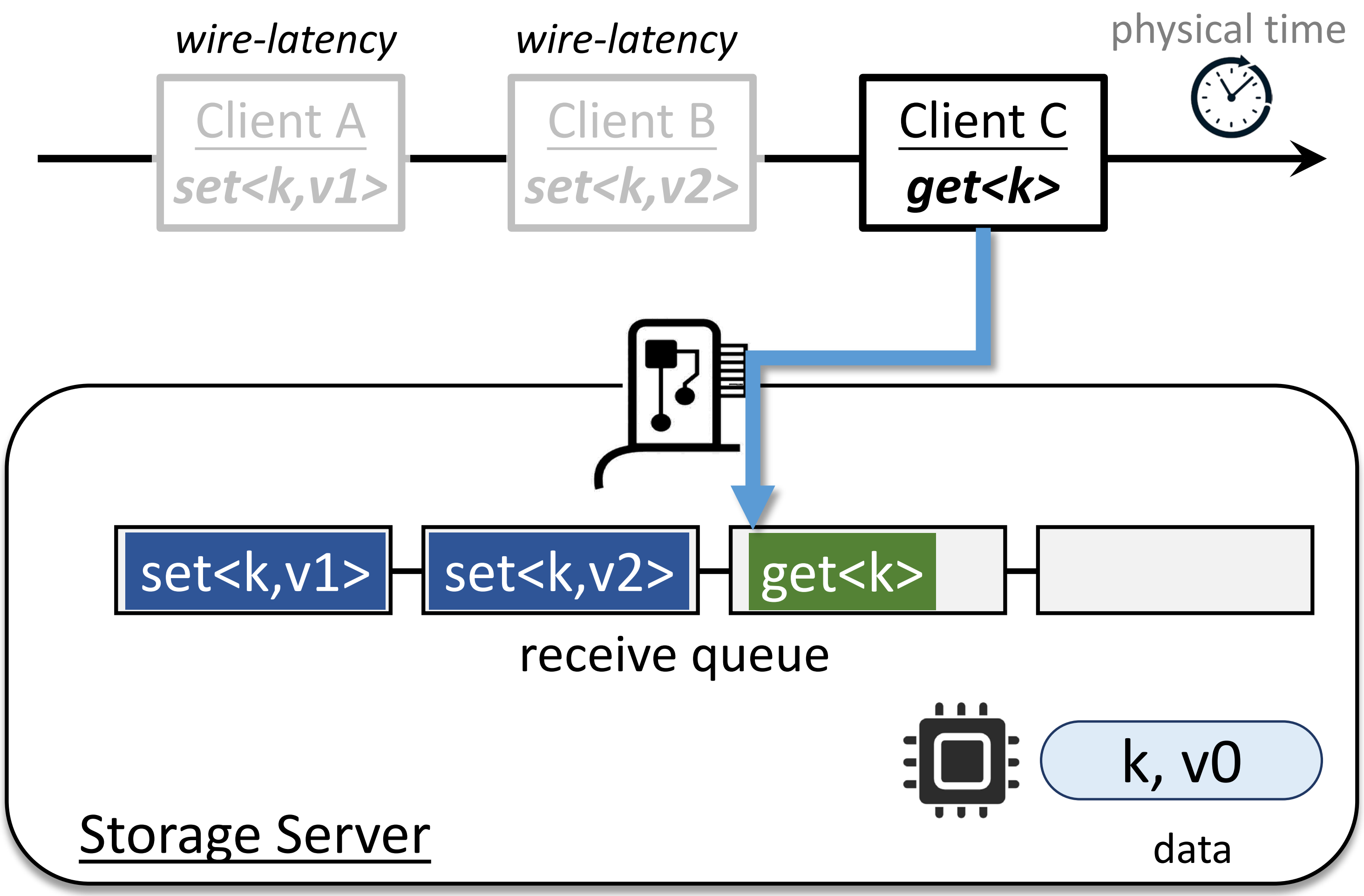
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



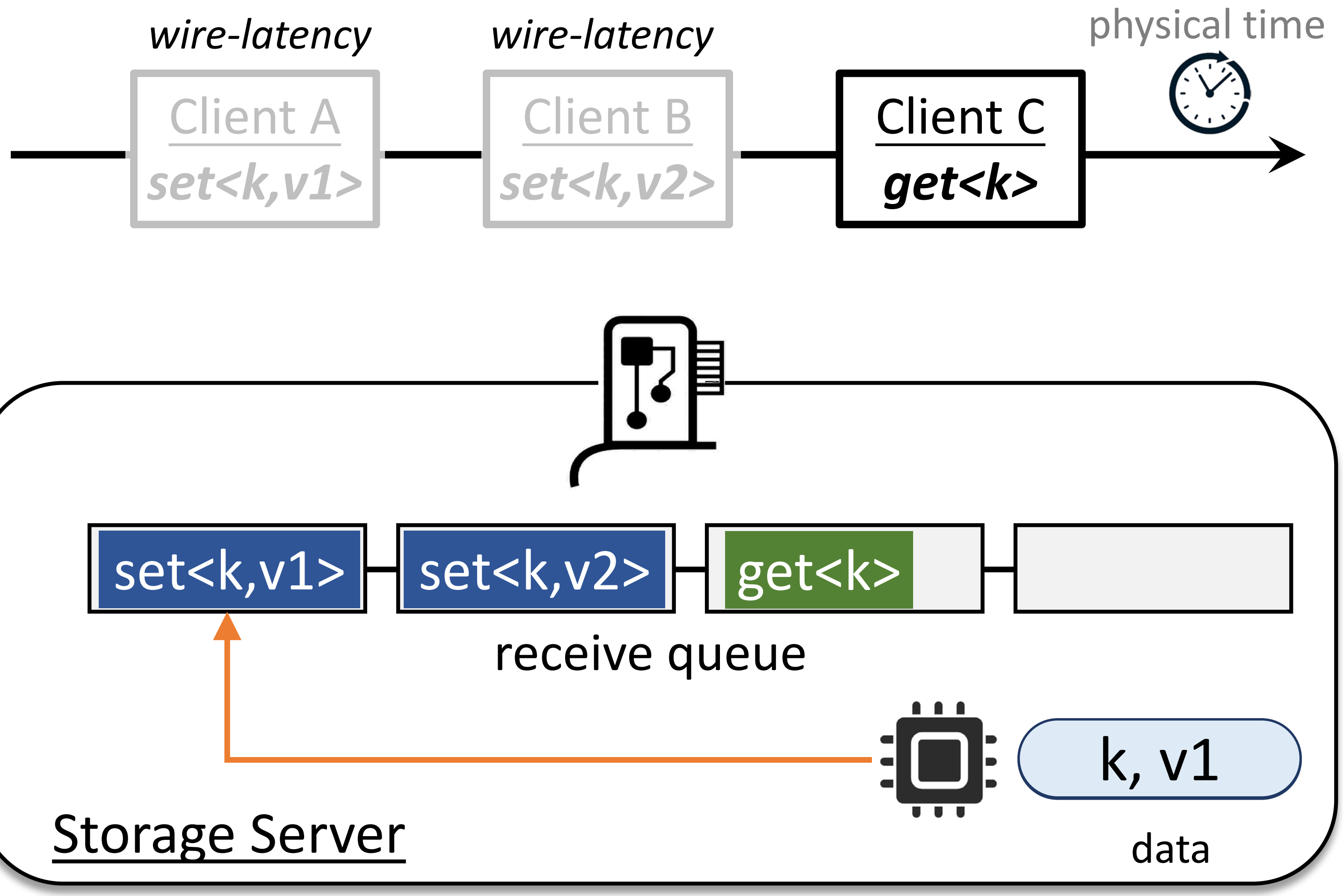
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



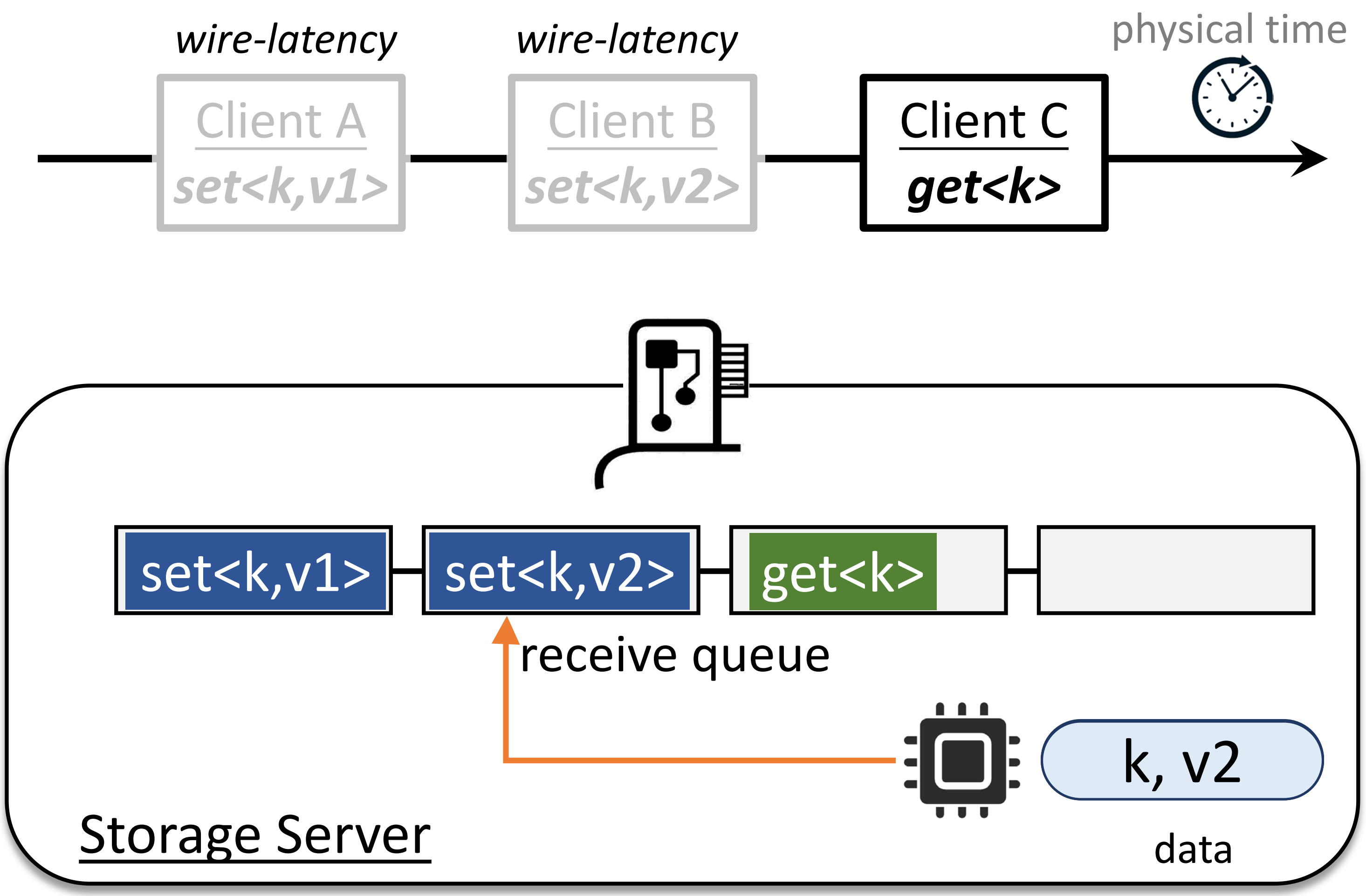
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



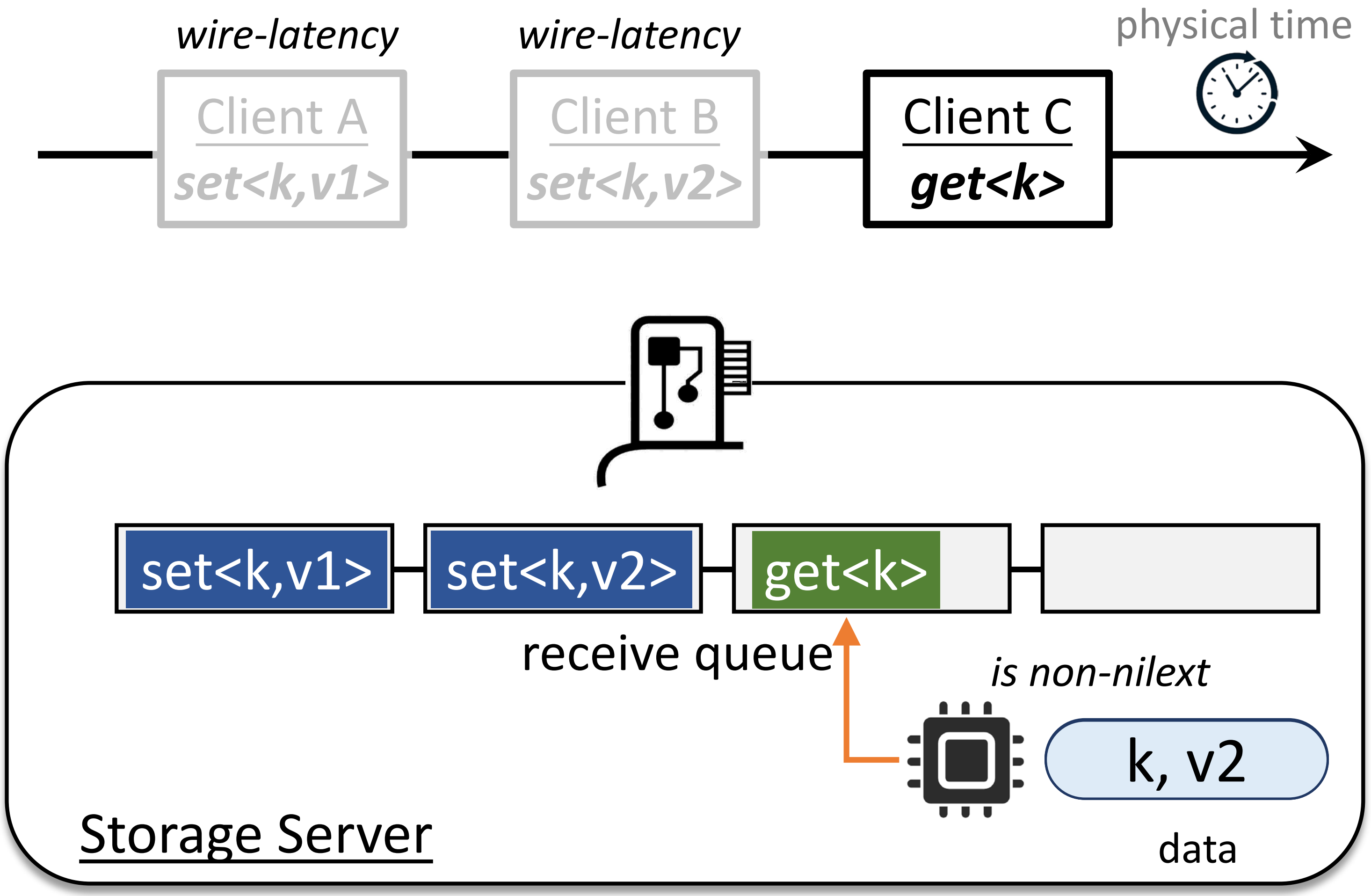
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



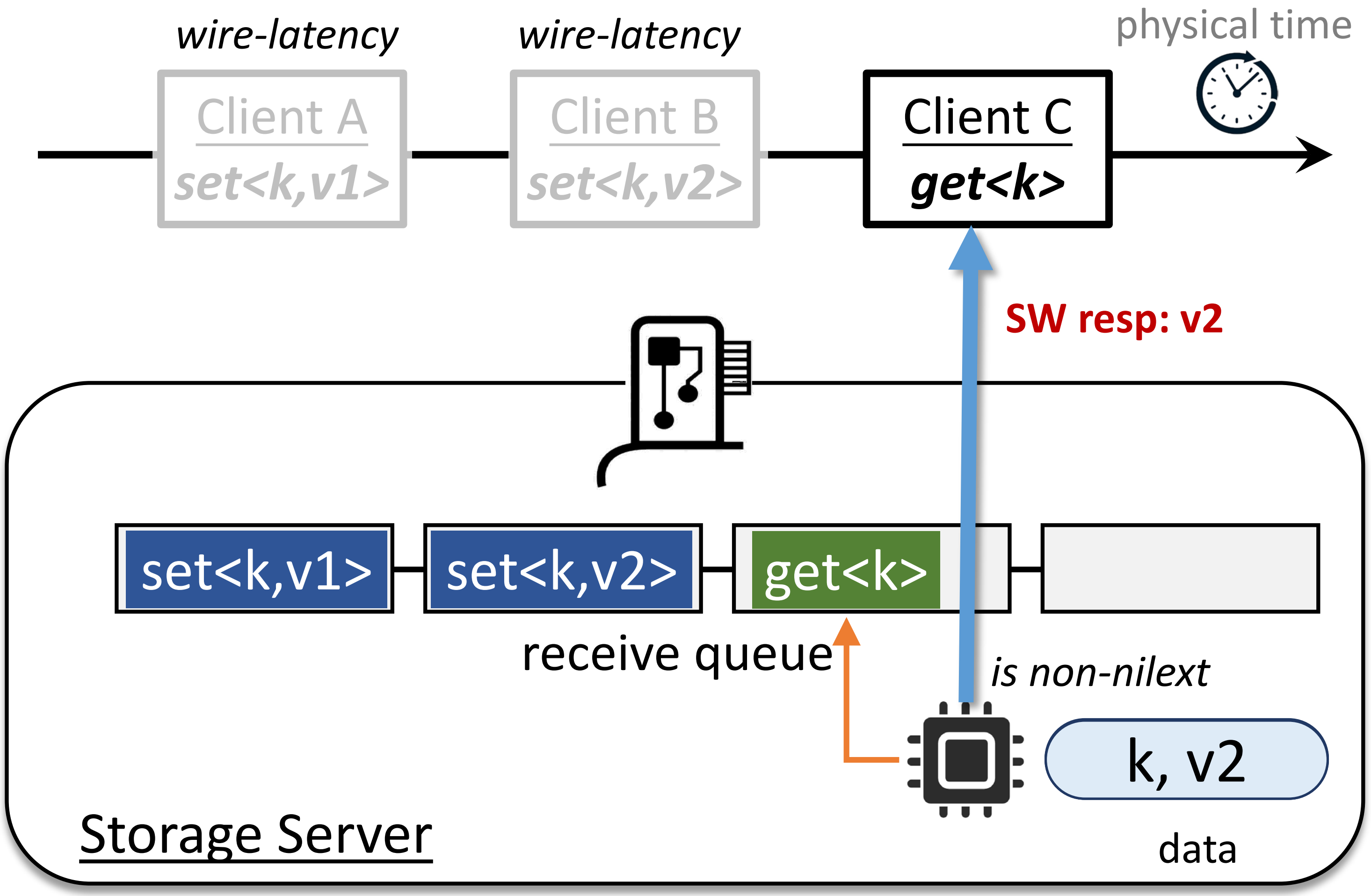
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



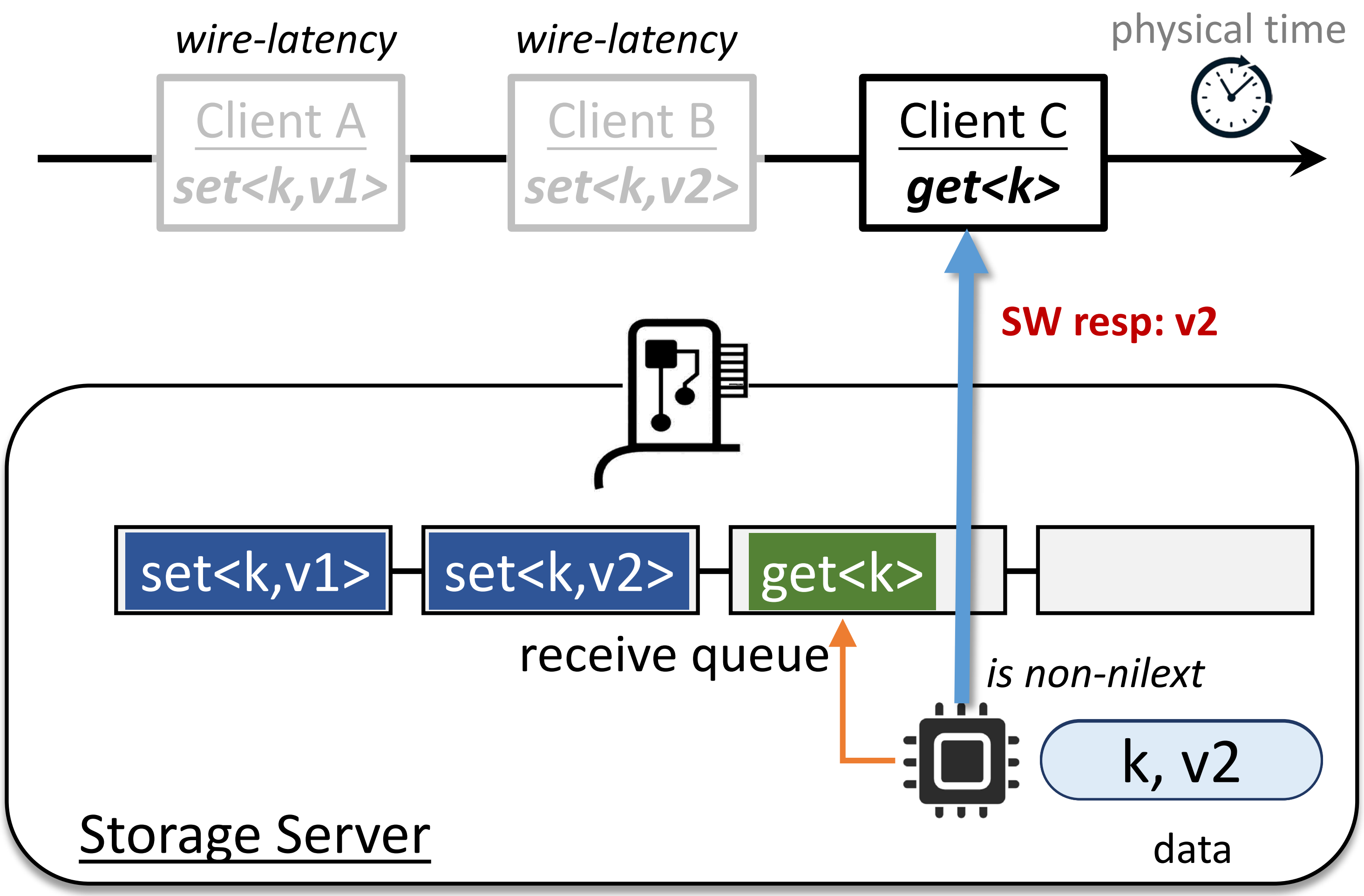
Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



Example of Ordered Queue

Memcached, where ***set*** is nilext and ***get*** is non-nilext



Correct linearizable order:

set<k,v1>	set<k,v2>	get<k>

⇕

Positions of buffers in RQ

Implementation of OQ (1):

We build ***Juneberry***, a communication framework that implements OQ

- Interface: Like RPC, but clients can mark a request as nilext and use the hardware ACK as its commit signal (i.e., wire-latency)
- Using commodity RDMA NICs (RNIC): two-sided send/recv on RC mode

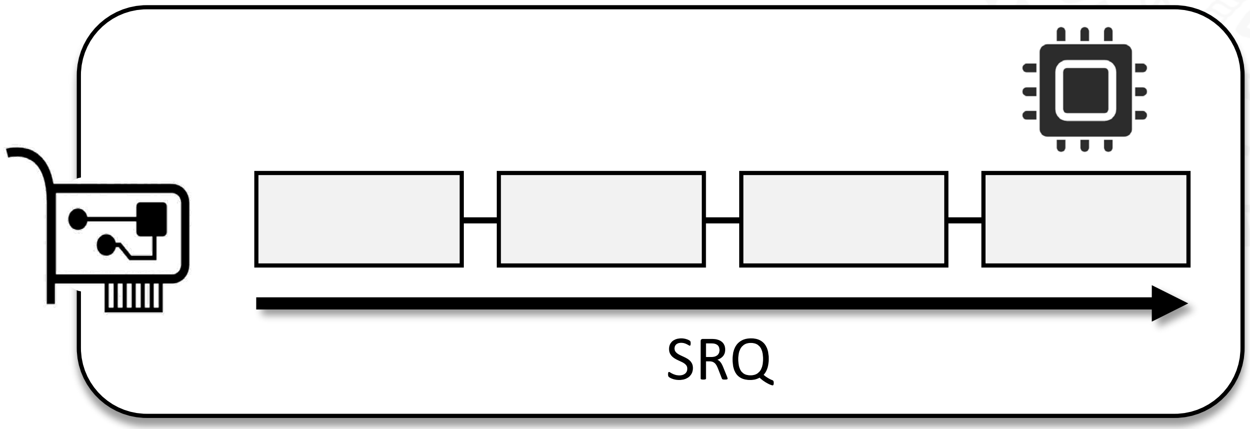
Implementation of OQ (1):

We build ***Juneberry***, a communication framework that implements OQ

- Interface: Like RPC, but clients can mark a request as nilext and use the hardware ACK as its commit signal (i.e., wire-latency)
- Using commodity RDMA NICs (RNIC): two-sided send/recv on RC mode

Juneberry is centered on ***shared receive queue (SRQ)***

- Different clients send requests to the same server-side SRQ



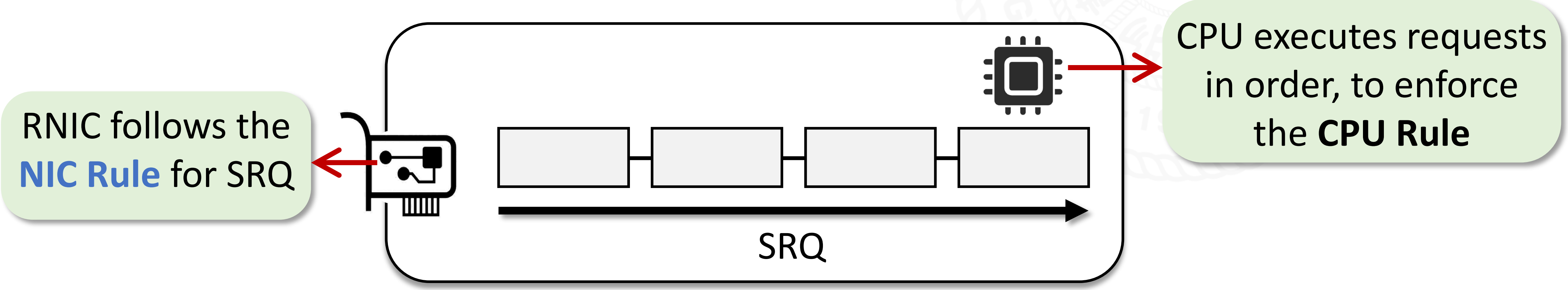
Implementation of OQ (1):

We build ***Juneberry***, a communication framework that implements OQ

- ❖ Interface: Like RPC, but clients can mark a request as nilext and use the hardware ACK as its commit signal (i.e., wire-latency)
- ❖ Using commodity RDMA NICs (RNIC): two-sided send/recv on RC mode

Juneberry is centered on ***shared receive queue (SRQ)***

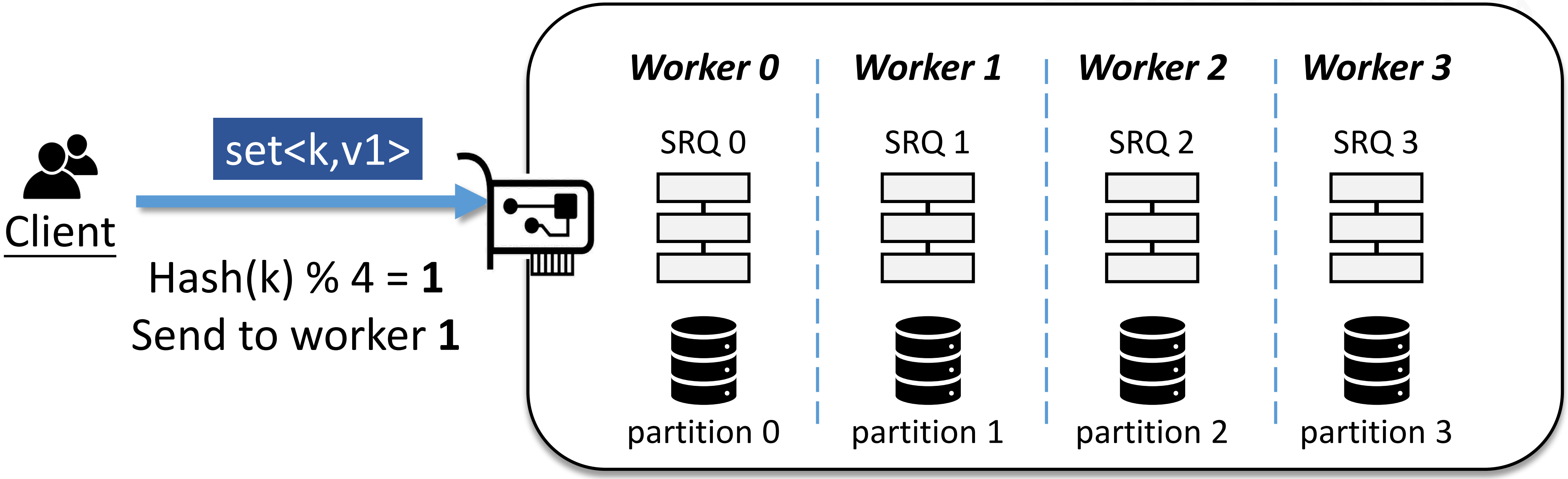
- ❖ Different clients send requests to the same server-side SRQ



Implementation of OQ (2):

Scaling to multiple CPU cores by data partitioning

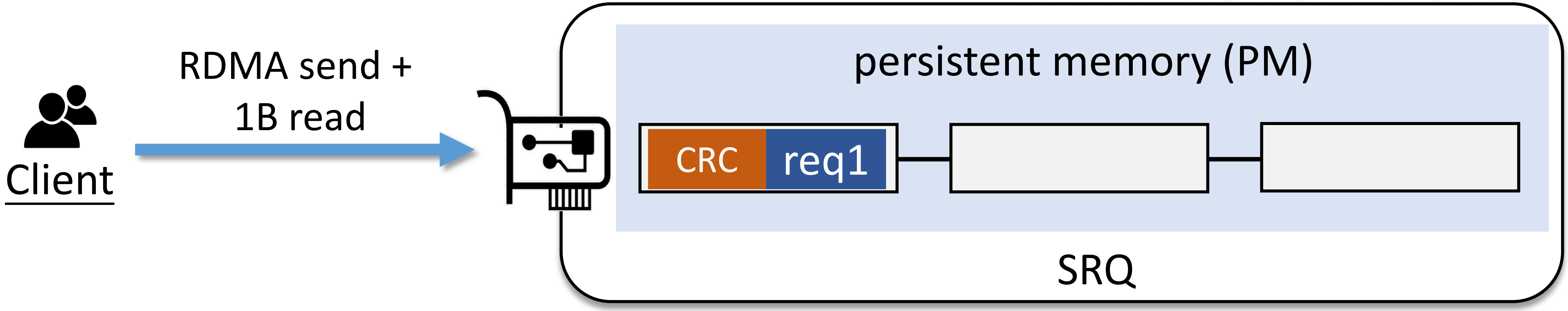
- ❖ ***Server:*** each worker thread creates an SRQ and managing an exclusive set of dataset
- ❖ ***Client:*** send requests to the correct SRQ according to partition scheme
- ❖ Limitation: do not support cross-partition operations



Implementation of OQ (3):

How to support storage systems that contain persistent states ?

- ***Step 1: putting SRQs in persistent memory (PM)***
  - Requests in SRQs can survive power outages
- ***Step 2: making the hardware ACKs have the persistent guarantee***
  - 1-byte RDMA read followed by every RDMA send, to drain requests to PM
  - Thus, upon receiving a hardware ACK, the associated nilext request is persisted in PM
- ***Step 3: embedding a checksum in every request***
  - During recovery, we can identify valid request and execute them



More Details: Check our paper

Performance Optimizations

- ❖ Leveraging multi-packet (MP) features to improve the performance of SRQs
- ❖ Using client-side delegation to reduce to number of QP connections

Recovery Procedure

- ❖ Handling non-idempotent requests
- ❖ ……

Experimental Setup

Hardware Platform

- One server machine: Two 18-core Intel Xeon Gold 6240M CPU, 192GB DRAM
  - 1.5TB Optane PM
  - Using one socket to avoid PM NUMA effect
- Eight client machines: Two 12-core Intel Xeon E5-2650 CPUs and 128GB DRAM
- 100Gbps Mellanox ConnectX-5 RNIC

Target comparisons

- ***eRPC*** [NSDI'19]: a state-of-the-art RPC framework, which uses unreliable datagrams
- ***DeferredExec***: the same codebase as Juneberry. Software solution: *when processing a nilext request, the CPU first returns a software ACK to commit and then executes it*

In-Memory Caching: Memcached

- Using Memcached (version: 1.6.19) as the back-end storage system
- Client threads generate requests at a given rate with Poisson arrivals
- Running four traces from twitter workloads (https://github.com/twitter/cache-trace)

	eRPC	DeferredExec	Juneberry
Cluster-12 (80% set)	24.3	10.3	3.4
Cluster-19 (25% set)	12.8	9.2	5.8
Cluster-27 (15% set)	9.2	6.8	5.9
Cluster-31 (94% set)	34.7	19.7	3.2

Median Latency (us) at Peak Throughput

In-Memory Caching: Memcached

- Using Memcached (version: 1.6.19) as the back-end storage system
- Client threads generate requests at a given rate with Poisson arrivals
- Running four traces from twitter workloads (https://github.com/twitter/cache-trace)

	eRPC	DeferredExec	Juneberry
Cluster-12 (80% set)	24.3	10.3	3.4
Cluster-19 (25% set)	12.8	9.2	5.8
Cluster-27 (15% set)	9.2	6.8	5.9
Cluster-31 (94% set)	34.7	19.7	3.2

Median Latency (us) at Peak Throughput

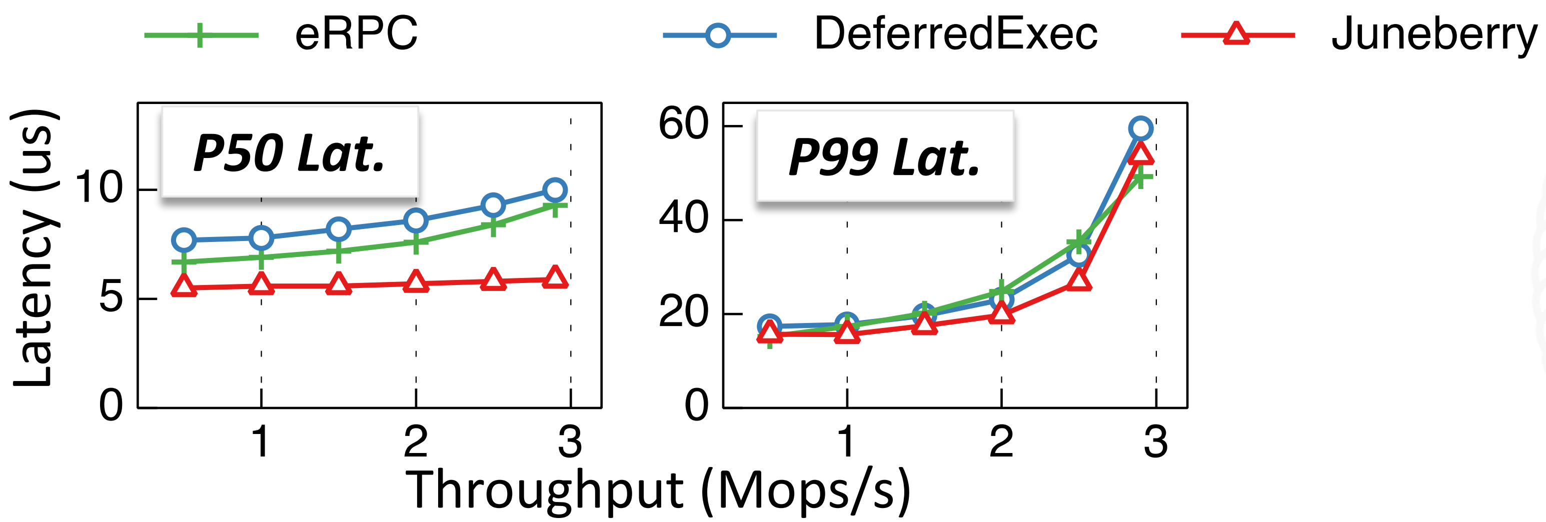
Juneberry reduces the P50 latency by up to 90.8% and 83.8% against eRPC and DeferredExec,
This is because: Juneberry eliminates any software delay in the critical path of set requests.

Persistent KV Store: PMemKV

- ❖ Using PMemKV (https://github.com/pmem/pmemkv) as the back-end storage system
- ❖ Client threads generate requests at a given rate with Poisson arrivals
- ❖ We set the object size to 91-byte (the average object size in Meta’s largest KV system ZippyDB)
- ❖ Workloads follow a Zipf 0.99 access distribution

Persistent KV Store: PMemKV

- Using PMemKV (https://github.com/pmem/pmemkv) as the back-end storage system
- Client threads generate requests at a given rate with Poisson arrivals
- We set the object size to 91-byte (the average object size in Meta's largest KV system ZippyDB)
- Workloads follow a Zipf 0.99 access distribution

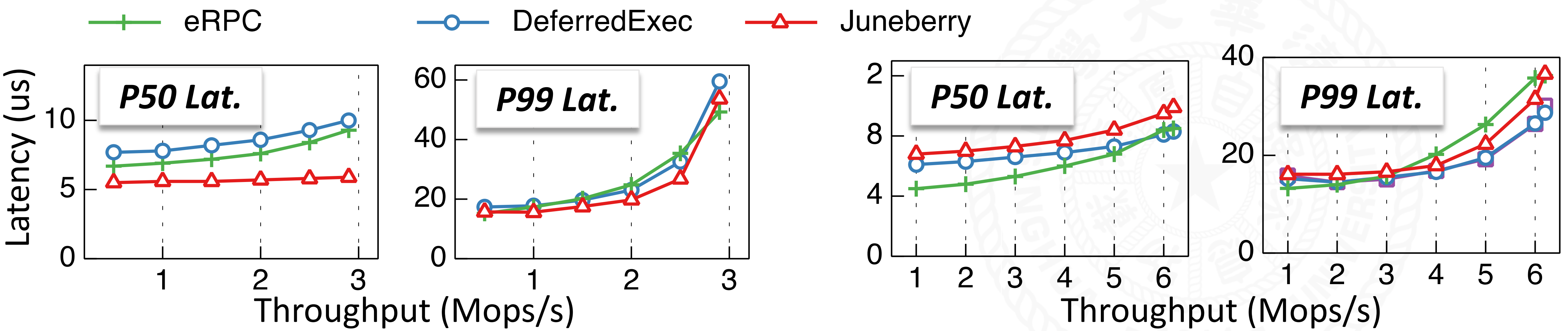


Write-intensive workload (50% put)

Juneberry reduces median latency by 40.83% / 36.71% against DeferredExec/eRPC under write-intensive workloads

Persistent KV Store: PMemKV

- Using PMemKV (https://github.com/pmem/pmemkv) as the back-end storage system
- Client threads generate requests at a given rate with Poisson arrivals
- We set the object size to 91-byte (the average object size in Meta's largest KV system ZippyDB)
- Workloads follow a Zipf 0.99 access distribution



Write-intensive workload (50% put)

Read-intensive workload (5% put)

Juneberry reduces median latency by 40.83% / 36.71% against DeferredExec/eRPC under write-intensive workloads

Juneberry has higher latency under read-intensive workloads due to overhead of PM access

Conclusion

Our Goal:

- Achieving wire-latency (1RTT & remote CPU bypass) for storage requests

Our Idea:

- Repurposing hardware ACKs of NICs as the commit signals of nilext requests

Our Design and Evaluation:

- Ordered Queue (OQ), an abstraction that ensures the linearizability
- Juneberry, a communication framework that implements OQ
- Juneberry can significantly lower request latency under write-intensive workloads

Takeaway:

- Making storage software have visibility into network-level knowledge can bring performance benefits

Achieving Wire-Latency Storage Systems by Exploiting Hardware ACKs

Thanks & QA

Contact Information: wq1997@tsinghua.edu.cn & [wq1997.com](wq1997.com)