ShiftLock: Mitigate One-sided RDMA Lock Contention via **Handover**

Jian Gao, Qing Wang, Jiwu Shu*

Tsinghua University

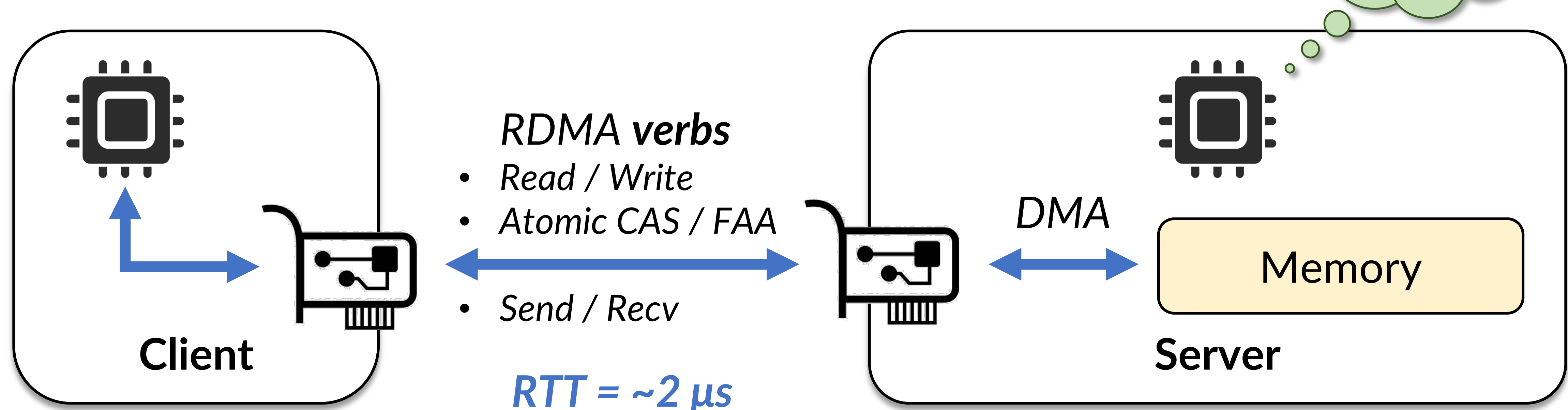
2025/02/26 at FAST'25

Remote Direct Memory Access (RDMA)

- Client directly access memory on server
- No server CPU intervention

Remote Direct Memory Access (RDMA)

- Client directly access memory on server
- No server CPU intervention



RDMA is Very Powerful

(Extended) Atomic verbs can …

- Operate on larger-than-8B values
  - *16-byte atomics with similar performance to original 8-byte atomics*
- Compare-swap (CAS) on arbitrary bits
- Fetch-add (FAA) on separate fields

RDMA is Very Powerful

(Extended) Atomic verbs can …

- Operate on larger-than-8B values
  - *16-byte atomics with similar performance to original 8-byte atomics*
- Compare-swap (CAS) on arbitrary bits
- Fetch-add (FAA) on separate fields

RDMA atomics are more powerful than CPU atomics.

(One-sided) RDMA Locks

Client

Client

Client

Lock Entries

Server

RDMA Locks are Good

Because of …

- Low latency
  - *Acquire/release a lock in an RDMA RTT (2~3 us)*
- High throughput
  - *370M packets/s for NVIDIA ConnectX-7 RNIC*
- Low CPU utilization
  - *Save power, no more CPU bottlenecks*

… But Problems Remain

One-sided = Server-inactive

- Client must retry on failure
  - *Retry traffic all heads to server*
- Retries are expensive
  - *RNIC queues up conflicting requests*

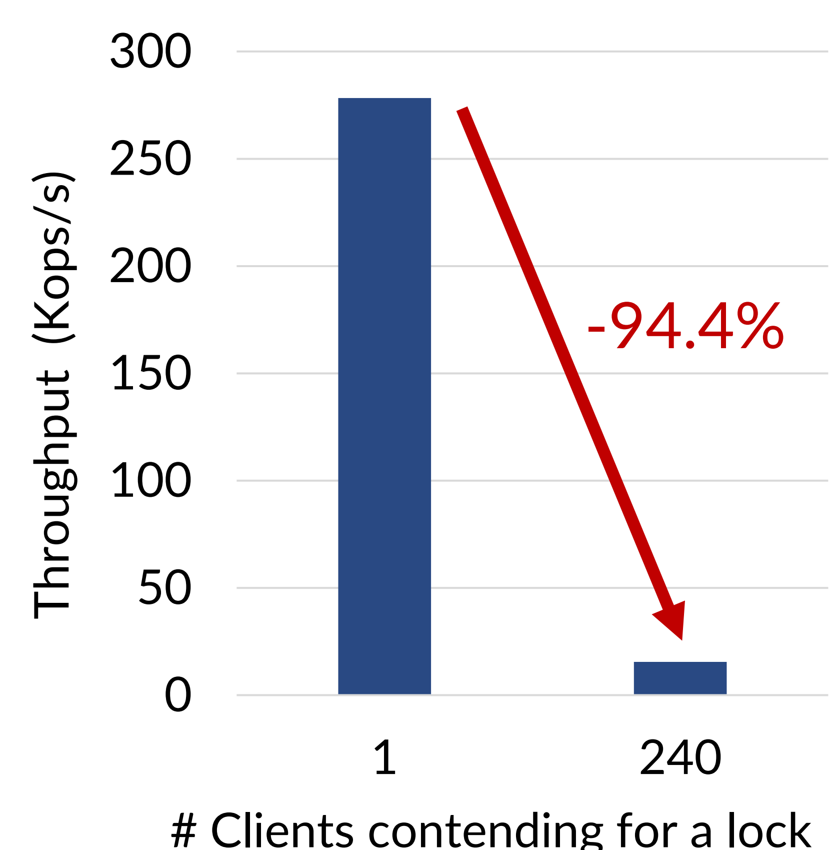
⇒ **Low performance under contention**

… But Problems Remain

One-sided = Server-inactive

- Client must retry on failure
  - *Retry traffic all heads to server*
- Retries are expensive
  - *RNIC queues up conflicting requests*

⇒ Low performance under contention



Problem - Too Many Retries

Client
Client
Client
Client
Server
Client
Client
Client
Client

Successful lock acq/rel
Failed retry

≥ 85%

failed retries
in a typical CAS lock
(240 clients, backoff [1] enabled)

[1] Ren et al. Scaling Up Memory Disaggregated Applications with SMART. ASPLOS (2024).

Solution - Let Clients **Hand over** Locks

Client

Client

Client

Client

Server

Client

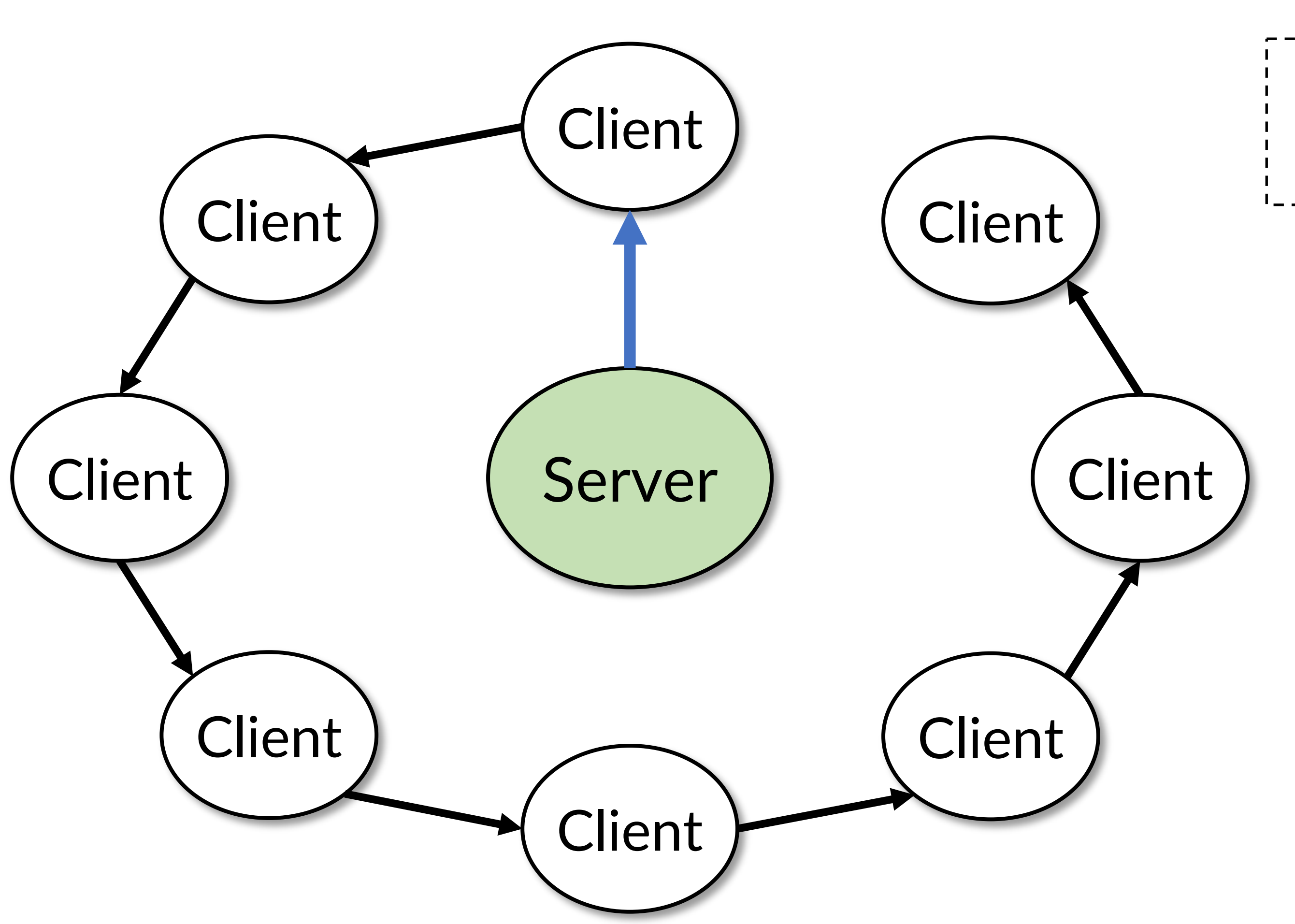
Client

Client

Client

Successful lock acq/rel
Enqueue

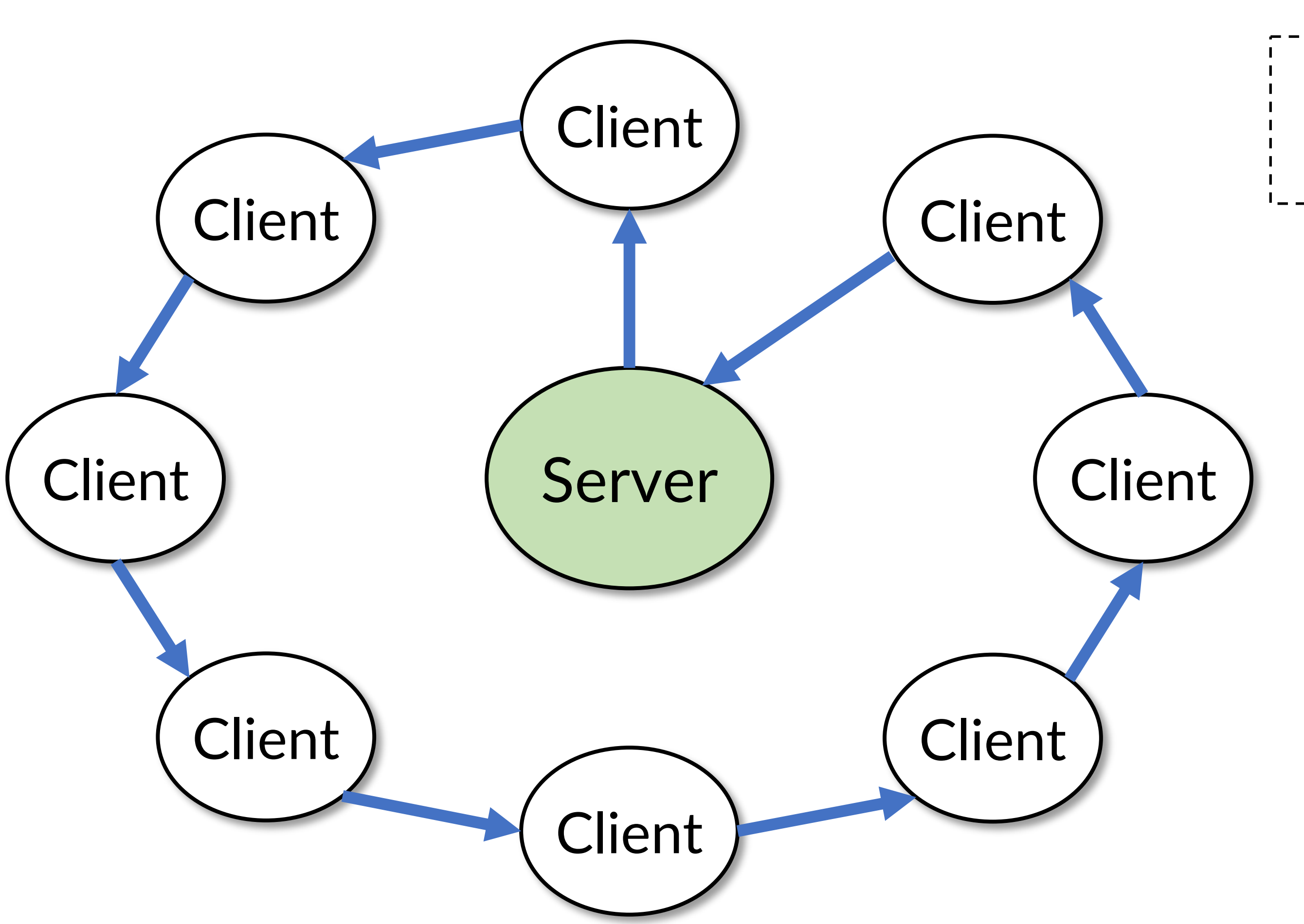
Solution - Let Clients **Hand over** Locks



Successful lock acq/rel
Enqueue

- Clients queue up
  - Similarly to **MCS / CLH locks**
  - Cli-to-cli direct messaging

Solution - Let Clients **Hand over** Locks



- → Successful lock acq/rel
- → Enqueue

- Clients queue up
  - Similarly to **MCS / CLH locks**
  - Cli-to-cli direct messaging
- Lock travels along the queue

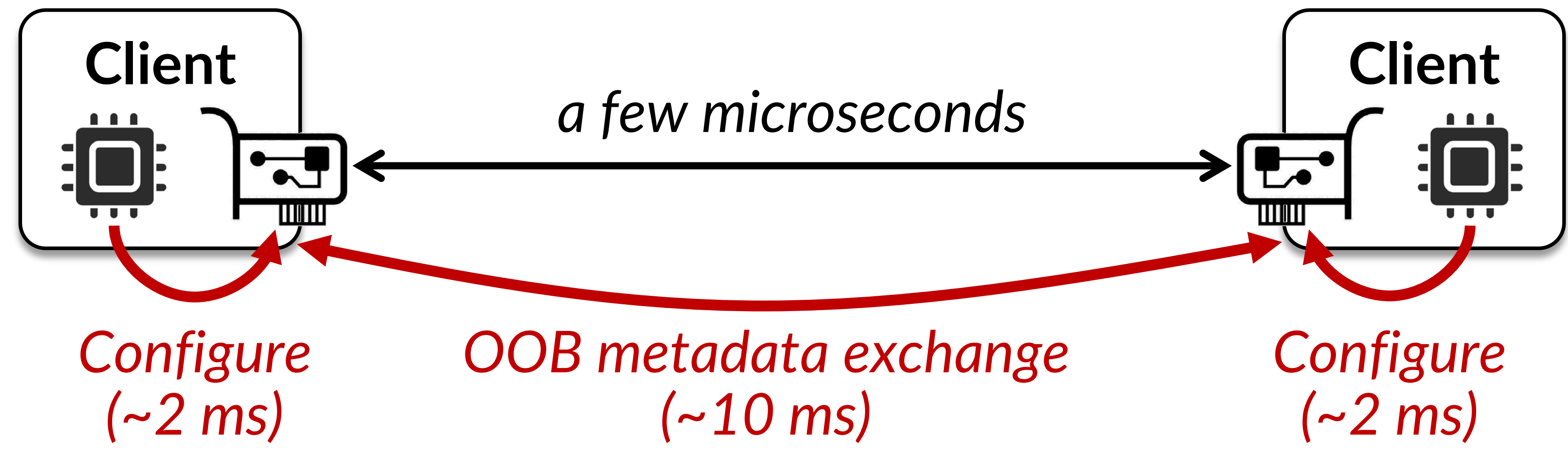
Challenges

MCS lock[1] is 34 yrs old. Why don't many RDMA locks use it?

- Expensive RDMA control path
  - *Difficult to enable cli-to-cli messaging*
- Long network RTT
  - *Single-node rwlock protocols won't work*
- Need of recovery
  - *Clients may fail*

[1] J.M. Mellor-Crummey. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. ACM TOCS (1991).

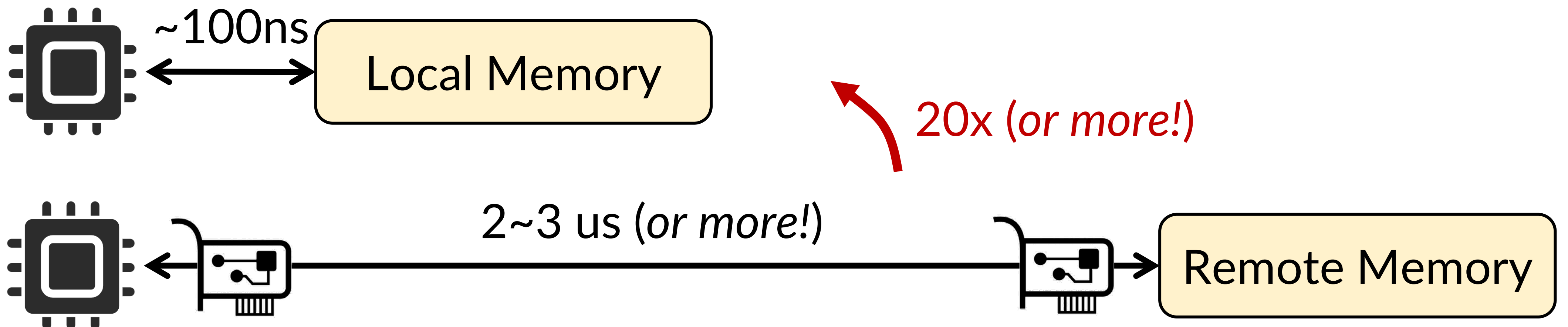
Challenge #1: Expensive Control Path



How to **enable cli-to-cli messaging?**

- All-to-all connection? *Unscalable*
- Connect at runtime? *Slow*

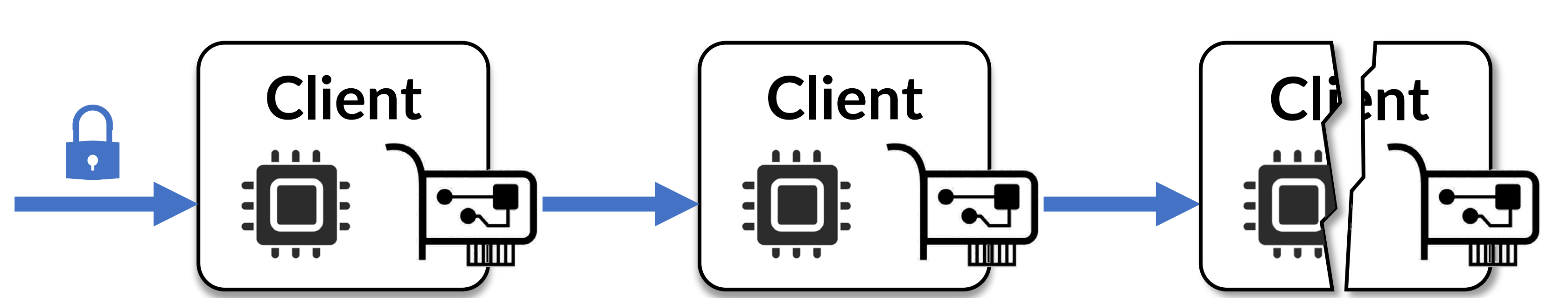
Challenge #2: Long RTT

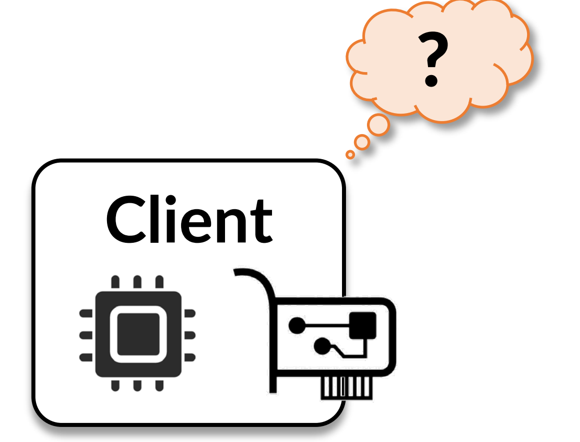


What lock semantics to implement?

- Mutex? *Low concurrency*
- Reader-writer lock? *Extra remote accesses*

Challenge #3: Fault Tolerance





How to **tolerate client failures**?

- How to detect a failure?
- How to correctly recover?

Idea of MCS lock, plus …

- Scalable cli-to-cli communication
- Single-RTT rwlock protocol
- Fault tolerance

* For the sake of clarity, terms used in this presentation may differ from what we used in our paper.

Basis - MCS Mutex

Lock Entry
(a pointer)

Tail
`null`

ok	**next**
`false`	`null`

Client A

ok	**next**
`false`	`null`

Client B

Basis - MCS Mutex

Lock Entry
(a pointer)

Tail
A

ok	**next**
`false`	`null`

Fetch-and-Store
(? → A)

ok	**next**
`false`	`null`

Client A

Client B

tail == `null`, OK

Basis - MCS Mutex

Lock Entry
(a pointer)

Tail
B

ok	**next**
`false`	`null`

Client A

Fetch-and-Store
(? → B)

ok	**next**
`false`	`null`

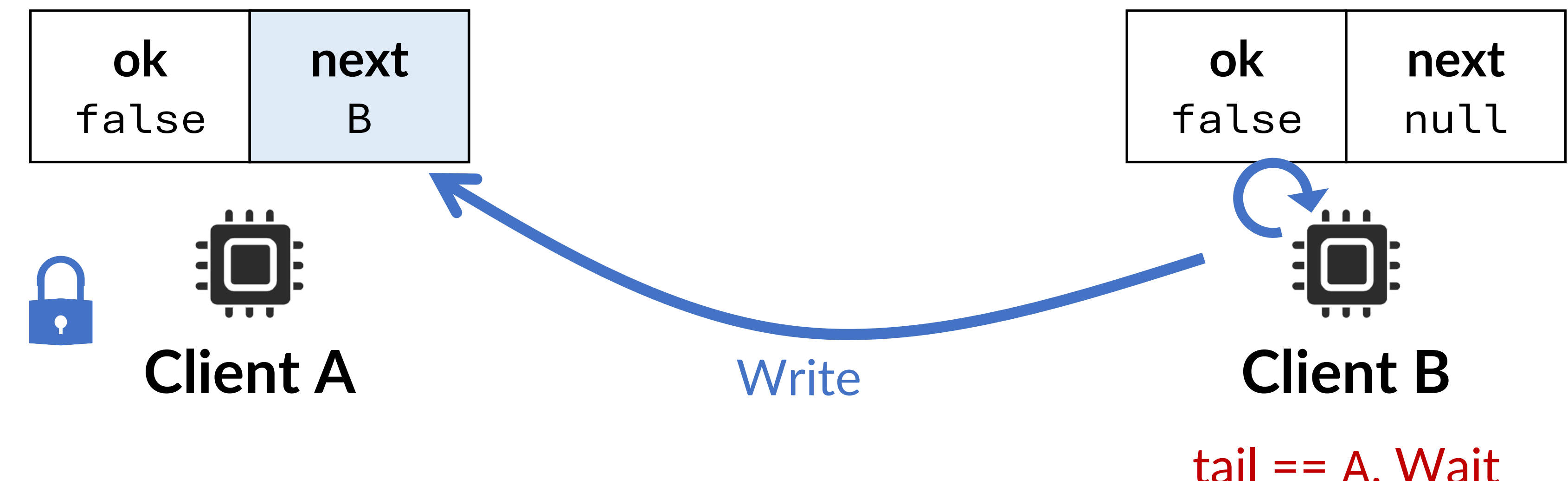
Client B

tail == A, Wait

Basis - MCS Mutex

Lock Entry
(a pointer)

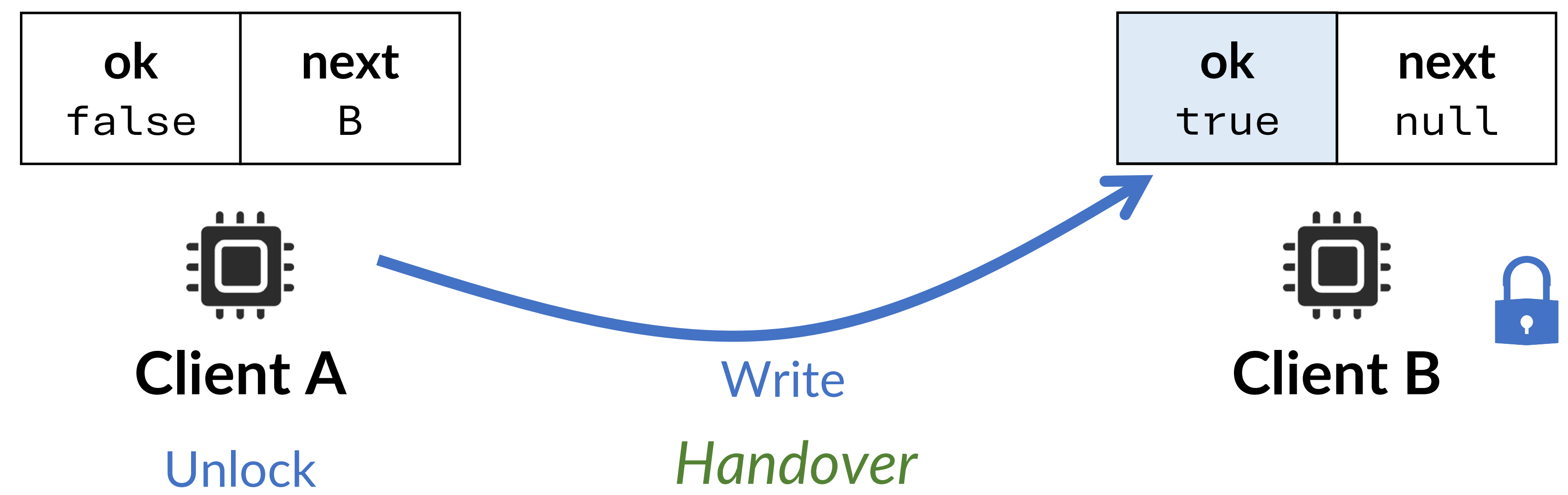
Tail
B



Basis - MCS Mutex

Lock Entry
(a pointer)

Tail
B



ShiftLock Outline

Idea of MCS lock, plus …

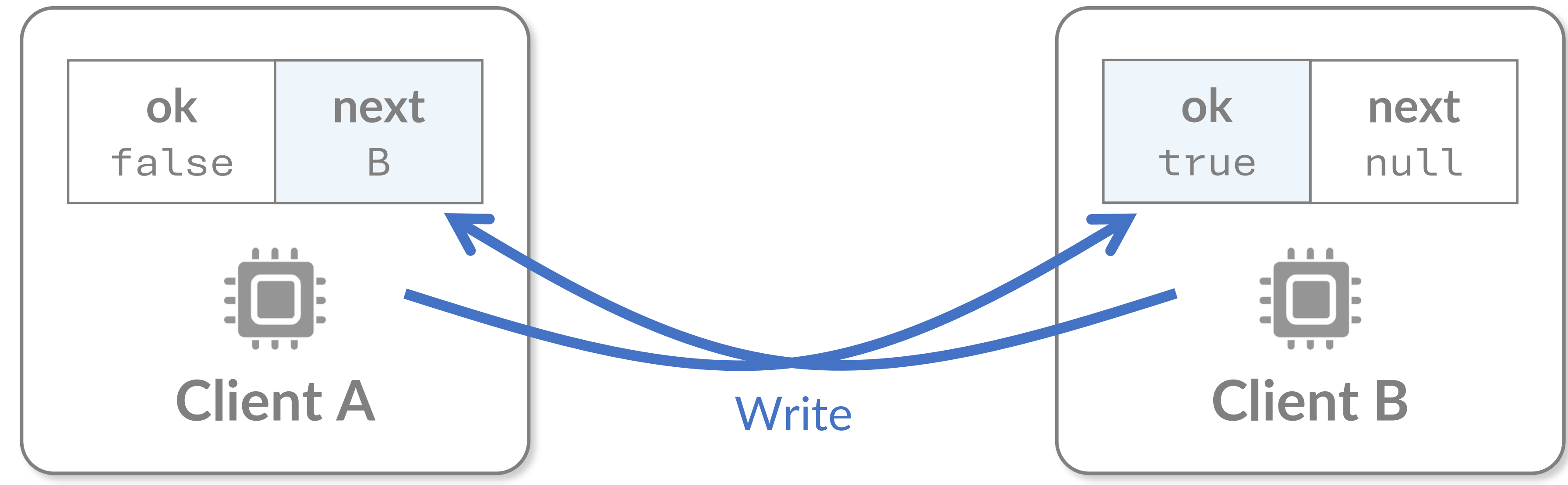
- Scalable cli-to-cli communication
- Single-RTT rwlock protocol
- Fault tolerance

Design #1 - Cli-to-cli Communication

Server

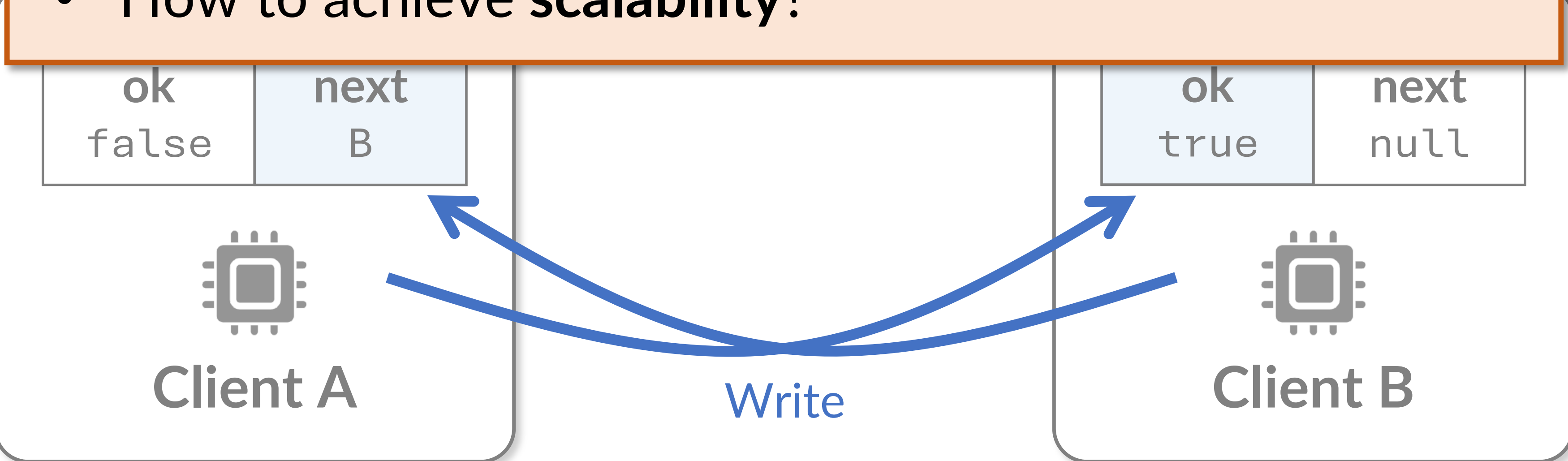
Lock Entry
(a pointer)

Tail
B



Design #1 - Cli-to-cli Communication

Lock Entry
(a pointer)

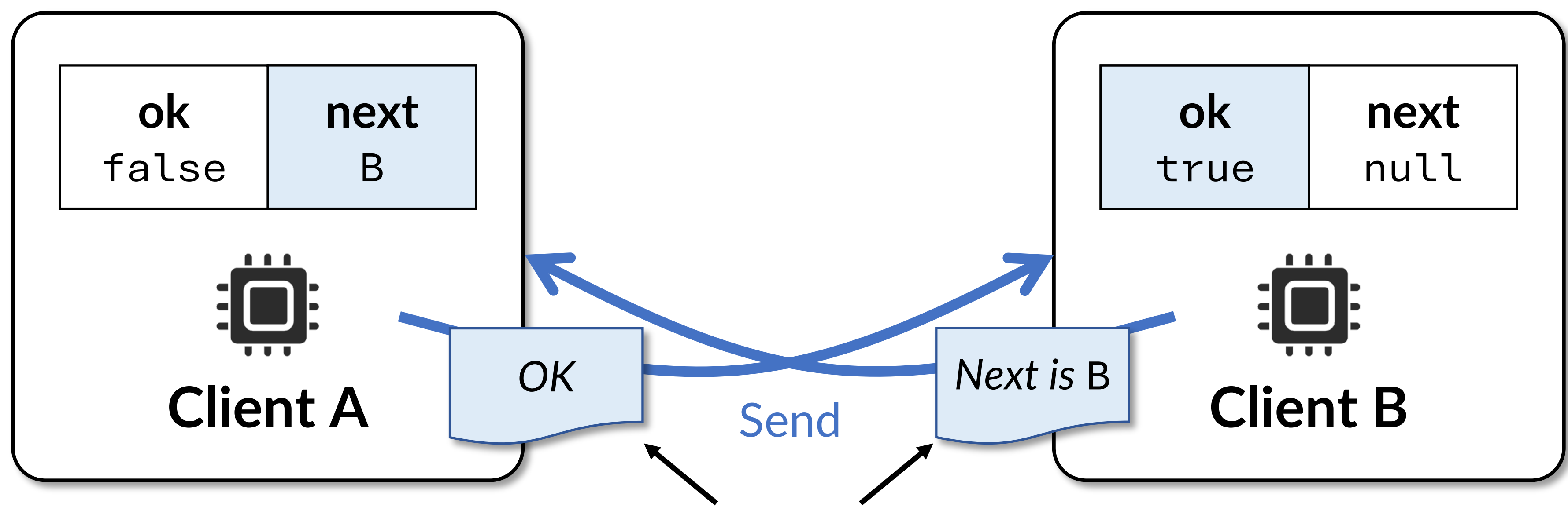


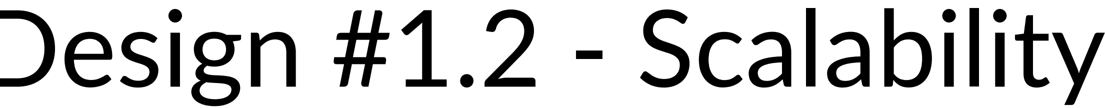
- What **RDMA verbs** should cli-to-cli communication use?
- How to achieve **scalability**?

Design #1.1 - Using Two-sided Verbs

One-sided RDMA Write:	GID 128b	LID 16b	QPN 24b	+	VAddr 64b	RKey 32b
Two-sided RDMA Send:	GID 128b	LID 16b	QPN 24b			

Save 96 bits of metadata





*RDMA **Dynamic Connection***

Target (Receiver)
Initiator (Sender)

DCT
DCI
Client A

DCT
DCI
Client B

DCT
DCI
Client C

DCT
DCI
Client D

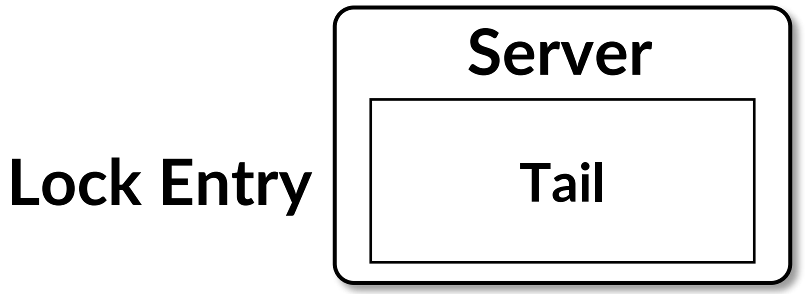
Reconnect on data path
Overhead < 1 μs

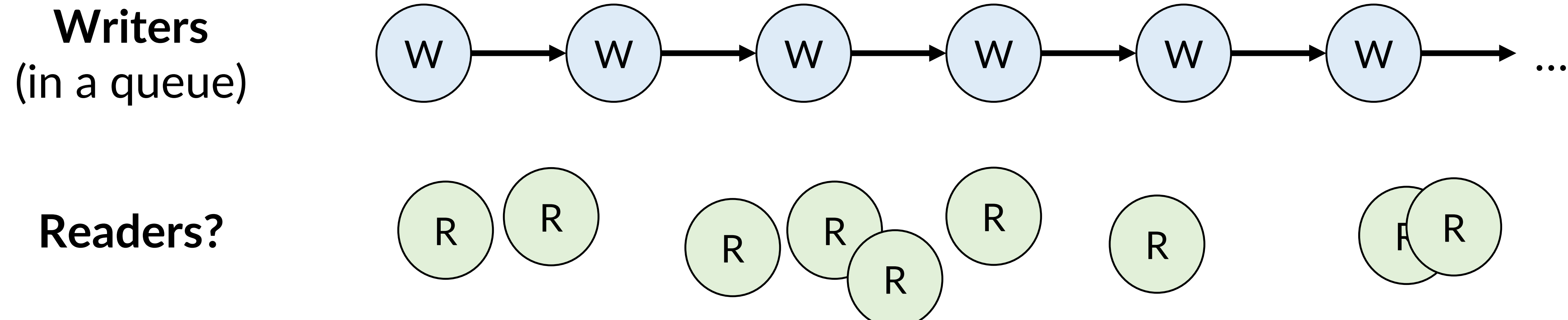
ShiftLock Outline

Idea of MCS lock, plus …

- Scalable cli-to-cli communication
- Single-RTT rwlock protocol
- Fault tolerance

Design #2 - Rwlock Semantics





Design #2 - Rwlock Semantics

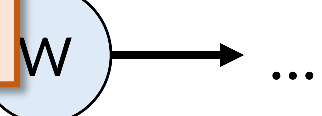
Server

Lock Entry

Tail

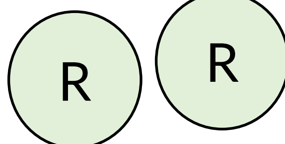
- How to **maintain readers**?
- How to **transfer lock** between readers & writers?

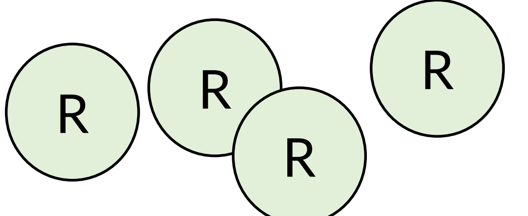
Writers
(in a queue)

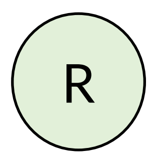


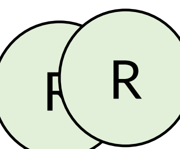
…

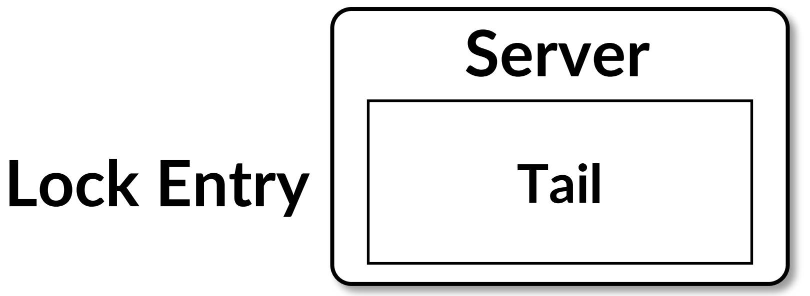
Readers?



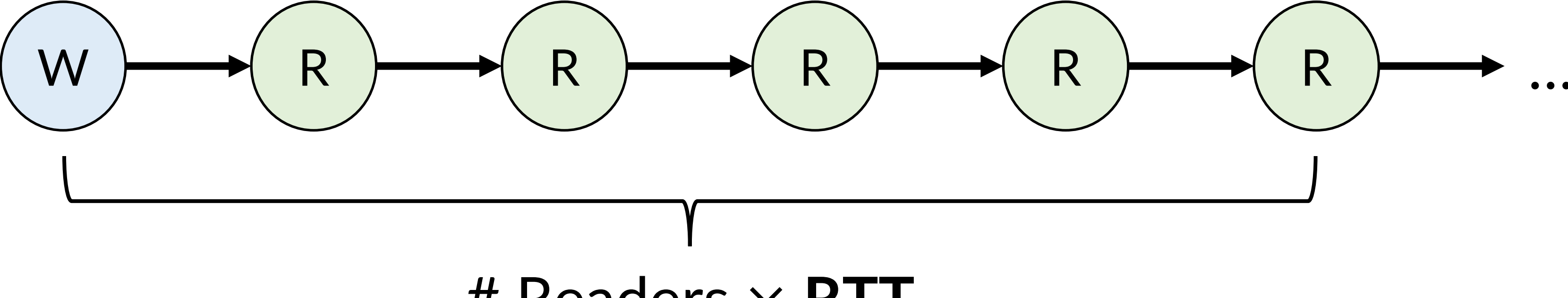








A unified queue?



Latency too high, NG!

Server

Lock Entry

WtrQueueTail	Epoch	RdrCnt	RelCnt
`null`	`0`	`0`	`226`

Count current readers

- *+1 when try to acquire*
- *−1 when release*

Writer acquire: *(except handover)* WtrQueueTail == null && RdrCnt == 0

(no writers) *(no readers)*

Reader acquire: WtrQueueTail == null

(no writers)

Design #2.2 - RLock → WLock

Server

Lock Entry

WtrQueueTail	Epoch	RdrCnt	RelCnt
`null`	`0`	`14`	`226`

When enqueuing, atomically read the entire entry

Count lock releases
- *+1 upon every release*

Readers before have all unlocked, acquire!

Writer acquire:
(*when **reader** holds the lock*)

RdrCnt	RelCnt
`14`	`226`

Wait →

RdrCnt	RelCnt
`?`	**`240`**

+

Design #2.3 - WLock → RLock

Lock Entry

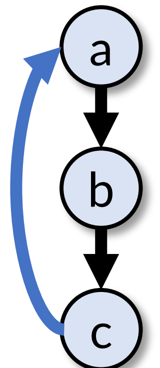
Server

WtrQueueTail	Epoch	RdrCnt	RelCnt
non-null	0	9	280

Give lock to readers

Lock lifetime contains many epochs. An epoch contains 3 steps:

Give lock to readers
= Start a new epoch



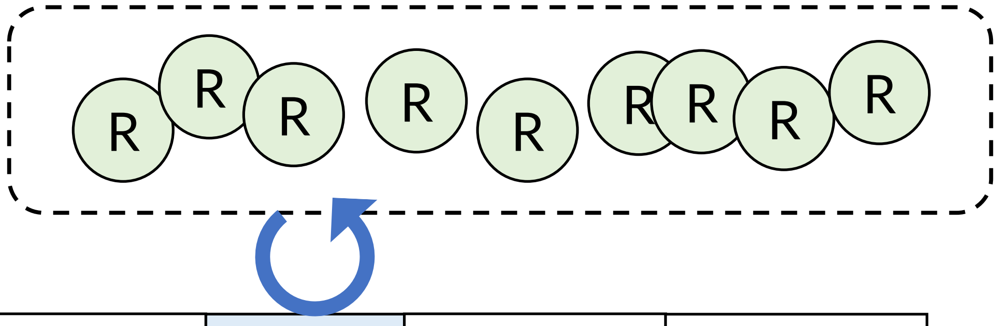
Readers acquire/release

A writer appears and takes it

Writers handover

Design #2.3 - WLock → RLock

Reader acquire:
(*when **writer** holds the lock*)



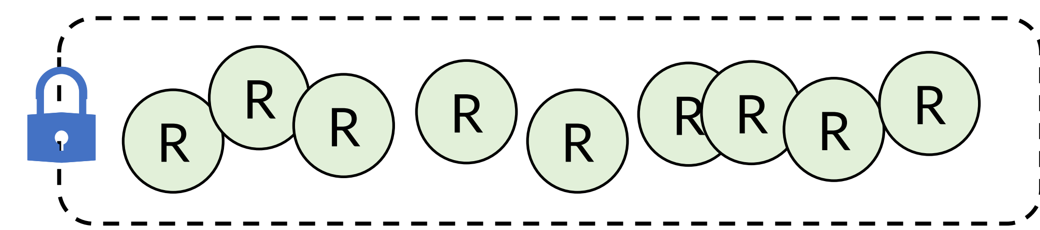
WtrQueueTail	Epoch	RdrCnt	RelCnt
non-null	0	9	280

Writer A
next = B

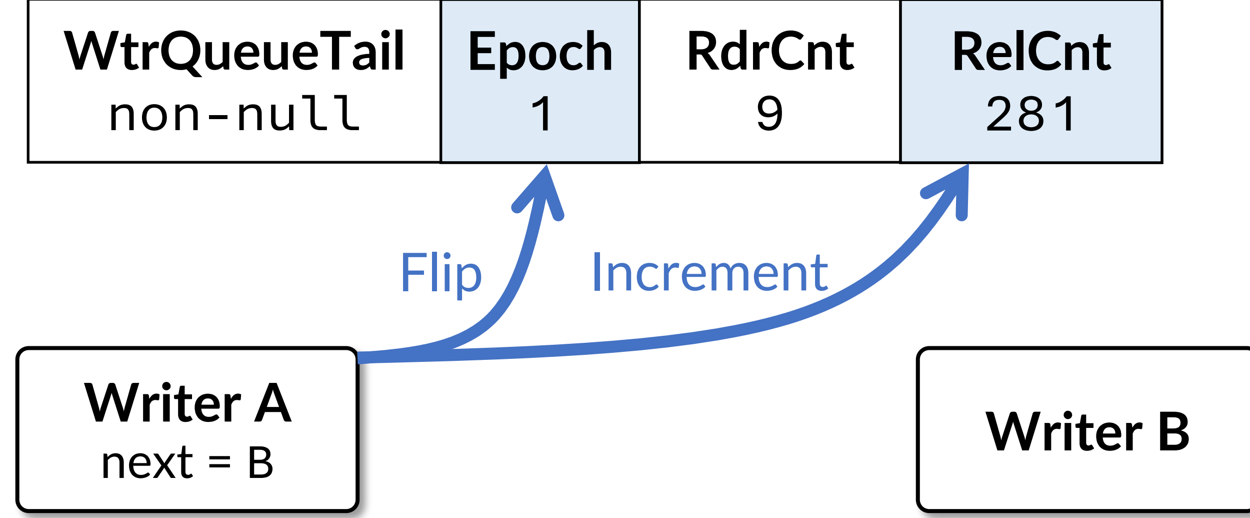
Writer B

Design #2.3 - WLock → RLock

Reader acquire:
(*when **writer** holds the lock*)



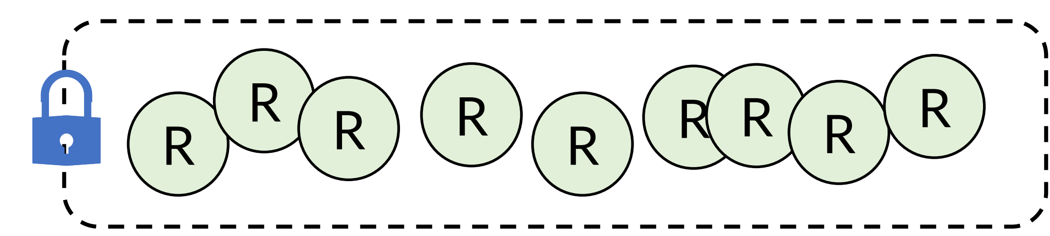
Epoch changed, acquire



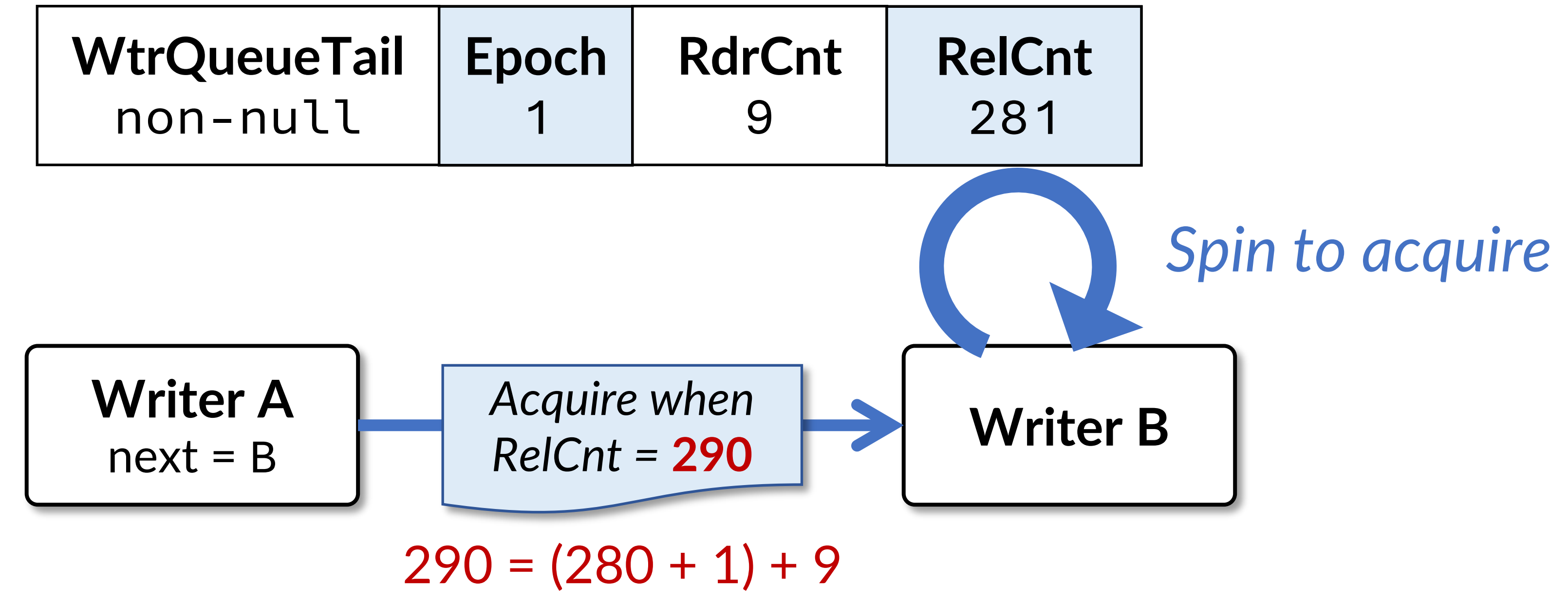
RdrCnt = 9, RelCnt = 280

Design #2.3 - WLock → RLock

Reader acquire:
(*when **writer** holds the lock*)



Epoch changed, acquire



Design #2.3 - WLock → RLock

Reader a
(*when* **writer** *h*

h changed,
acquire

Readers acquire when Epoch *changes*

Epoch is not an "mode" that "must be correct"
Guarantees ***single-RTT acquisition*** *w/o contention*

WtrQueueTail | **Epoch** | **RdrCnt** | **RelCnt**

acquire

Writers ***proactively*** transfer the lock

Criteria: consecutive handovers ≥ threshold

next = B

RelCnt = **290**

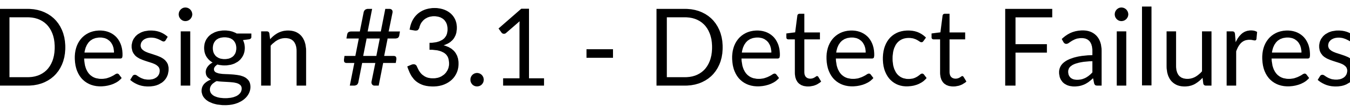
Writer B

290 = (280 + 1) + 9

ShiftLock Outline

Idea of MCS lock, plus …

- Scalable cli-to-cli communication
- Single-RTT rwlock protocol
- Fault tolerance



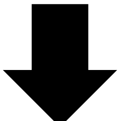
Design #3.1 - Detect Failures

Server

Lock Entry	**WtrQueueTail**	**Clk**	**RdrCnt**	**RelCnt**

Observation:

Client failure = eventually, no one will ***release*** the lock
= *RelCnt will stop changing*



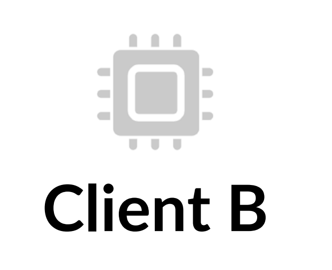
Detect a failure if ***RelCnt keeps unchanged*** for a certain period
(i.e., leasing)

Design #3.2 - Two-sided Recovery

Server

Lock Entry

WtrQueueTail	Clk	RdrCnt	RelCnt	Era 3



Client B

Preempted by OS

Design #3.2 - Two-sided Recovery

Server

	WtrQueueTail	Clk	RdrCnt	RelCnt	Era 3
Lock Entry					

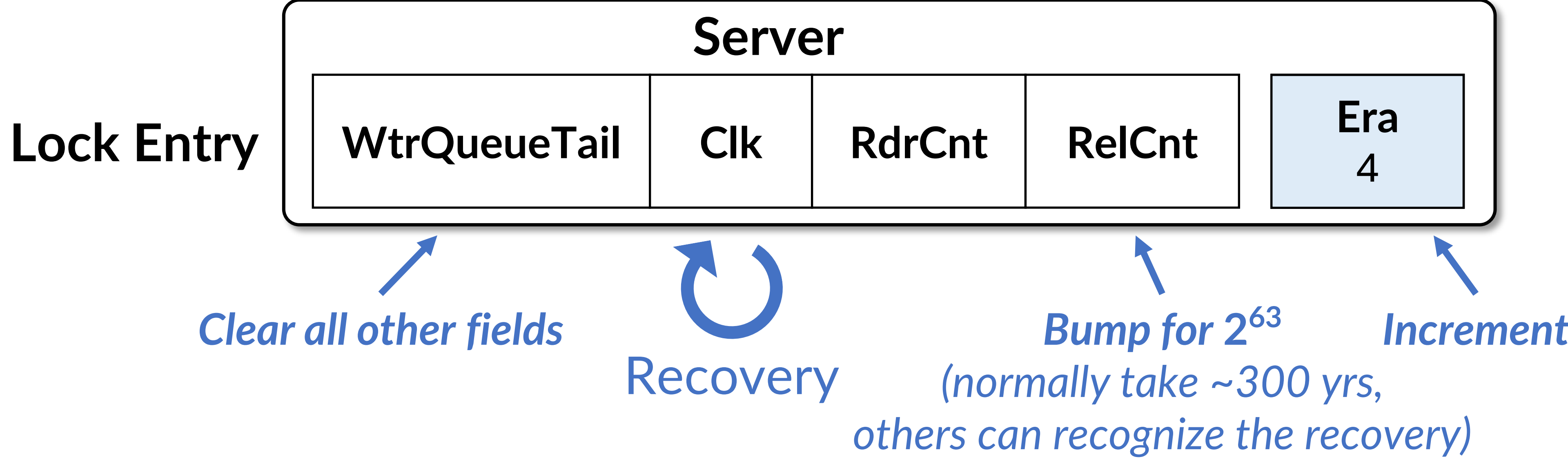
Ask for recovery
in Era 3

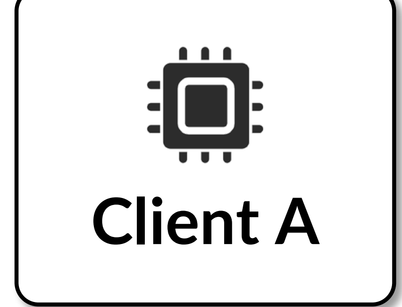
Client A

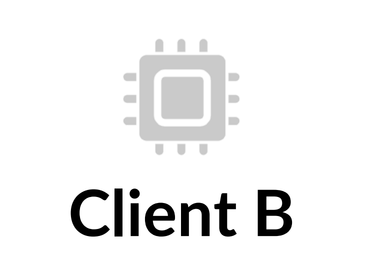
Client B

Preempted by OS

Design #3.2 - Two-sided Recovery



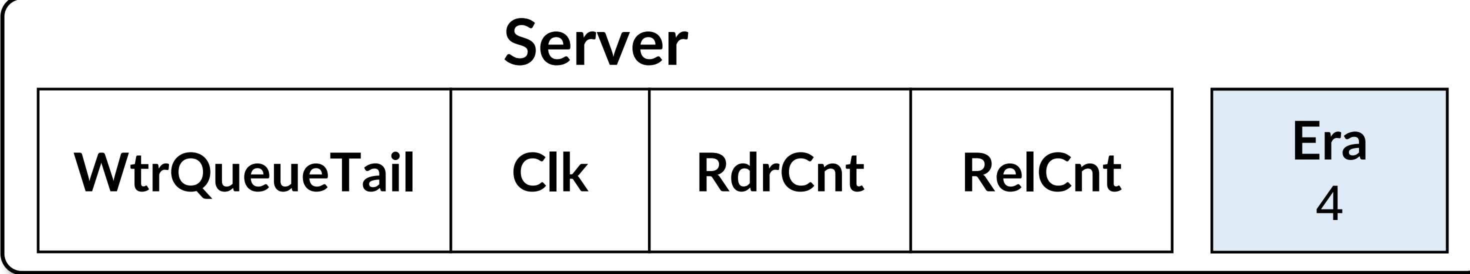




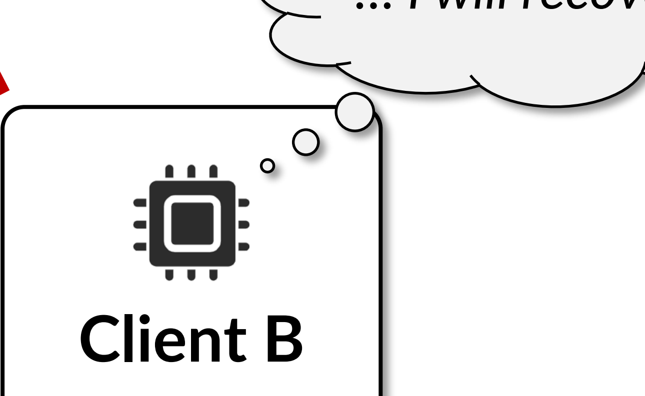
Preempted by OS

Design #3.2 - Two-sided Recovery

Lock Entry



Ask for recovery in era 3
(will definitely fail)



Design Summary

- Scalable cli-to-cli communication
  - *Two-sided RDMA on Dynamic Connection*
- Single-RTT rwlock protocol
  - *Minimum latency w/o contention*
- Fault tolerance
  - *Correct two-sided recovery*

See our paper for more details! ☺

Evaluation

- Implemented in **Rust, ~7.2k LoC**
- Compared with:
  - DSLR [SIGMOD'18], RMA-RW [HPDC'16], CAS (+Backoff), RPC, …

- 1 server + 5 clients
- 2x Intel® Xeon® E5-2650v4 CPUs
  - 2 × 24 = 48 cores, 48 × 5 = 240 client threads
- Mellanox ConnectX-5 100Gbps RNIC + QM8790 switch

Overall Performance

CAS DSLR RPC RMA-RW ShiftLock

Throughput (Mops/s)

10
8
6
4
2
0
10^3 10^4 10^5 10^6 10^7
Number of Locks

1.38x

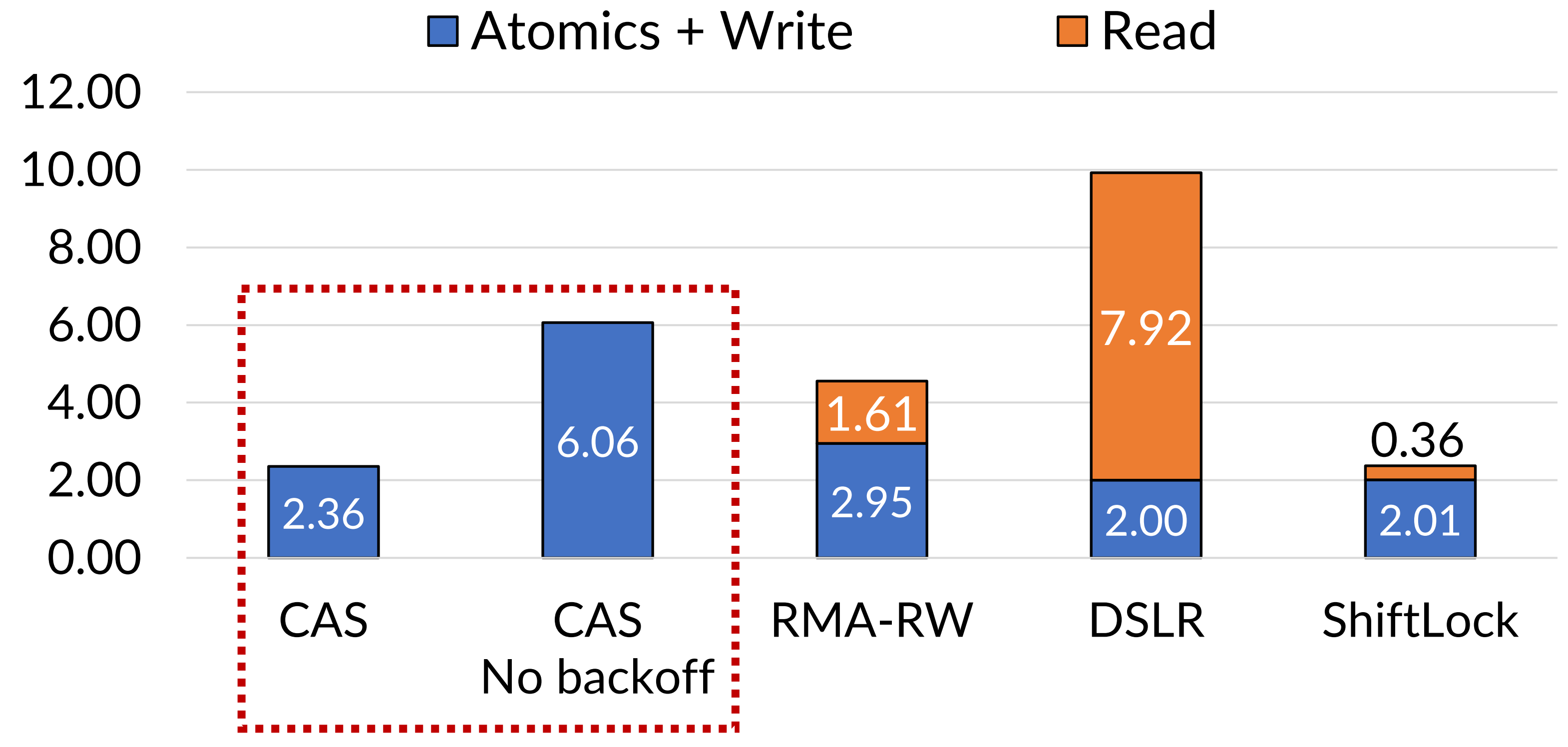
P99 Latency (µs)

2000
1600
1200
800
400
10^3 10^4 10^5 10^6 10^7
Number of Locks

+53%

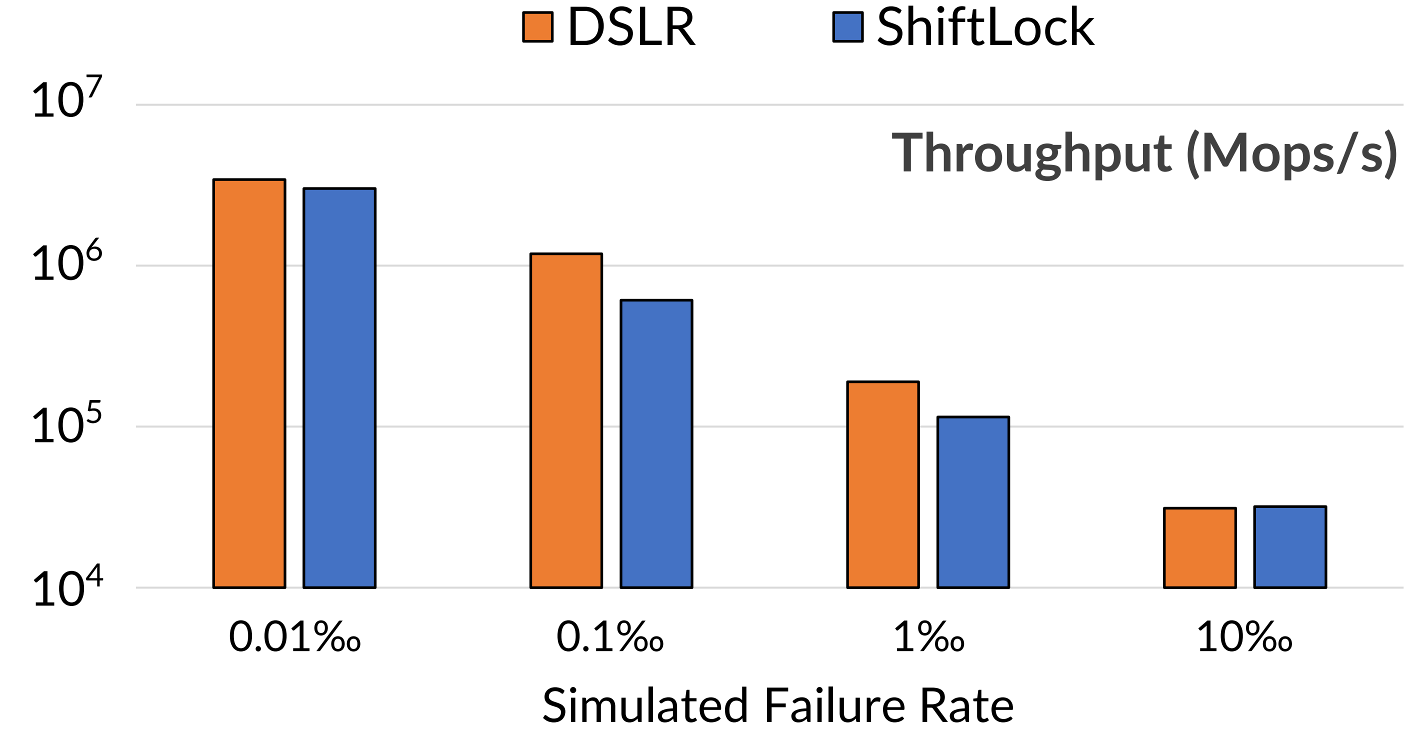
- ShiftLock **improves throughput & latency** for **rwlock** at different scales
- Compared to mutex, **writers wait longer** *(due to actively giving locks to readers)*

One-sided Verbs Received by the Server



- ShiftLock **effectively reduces retries** *(server RNIC has the lowest load)*
- Backoff has impressive effects, but **cannot solve the retry problem entirely**

Appendix: Recovery Performance



ShiftLock has **lower *(but similar)* recovery performance** than existing RDMA locks

Thank you! Questions are welcome.

- **ShiftLock**: Mitigate One-sided RDMA Lock Contention via Handover
- **MCS rwlock**, with …
  - Scalability *(**two-sided** cli-to-cli messaging with **Dynamic Connection**)*
  - Low latency *(guaranteed **single-RTT** w/o contention)*
  - Fault tolerance *(**correct** recovery)*
- **Fewer retries**, higher throughput

- Artifact at https://github.com/thustorage/shiftlock
- Contact: gaoj20@mails.tsinghua.edu.cn