

FUSEE: A Fully Memory-Disaggregated Key-Value Store

Jiacheng Shen[1], Pengfei Zuo[2], Xuchuan Luo[3], Tianyi Yang[1],
Yuxin Su[4], Yangfan Zhou[3], and Michael R. Lyu[1]

[1]*The Chinese University of Hong Kong,* [2]*Huawei Cloud,*
[3]*Fudan University,* [4]*Sun Yat-sen University*

ARISE Automated Reliable Intelligent Software Engineering

香港中文大學 The Chinese University of Hong Kong

HUAWEI

FUDAN UNIVERSITY 復旦大學

SUN YAT-SEN UNIVERSITY 中山大學

In-Memory Key-Value Stores

Fundamental building blocks for cloud services:

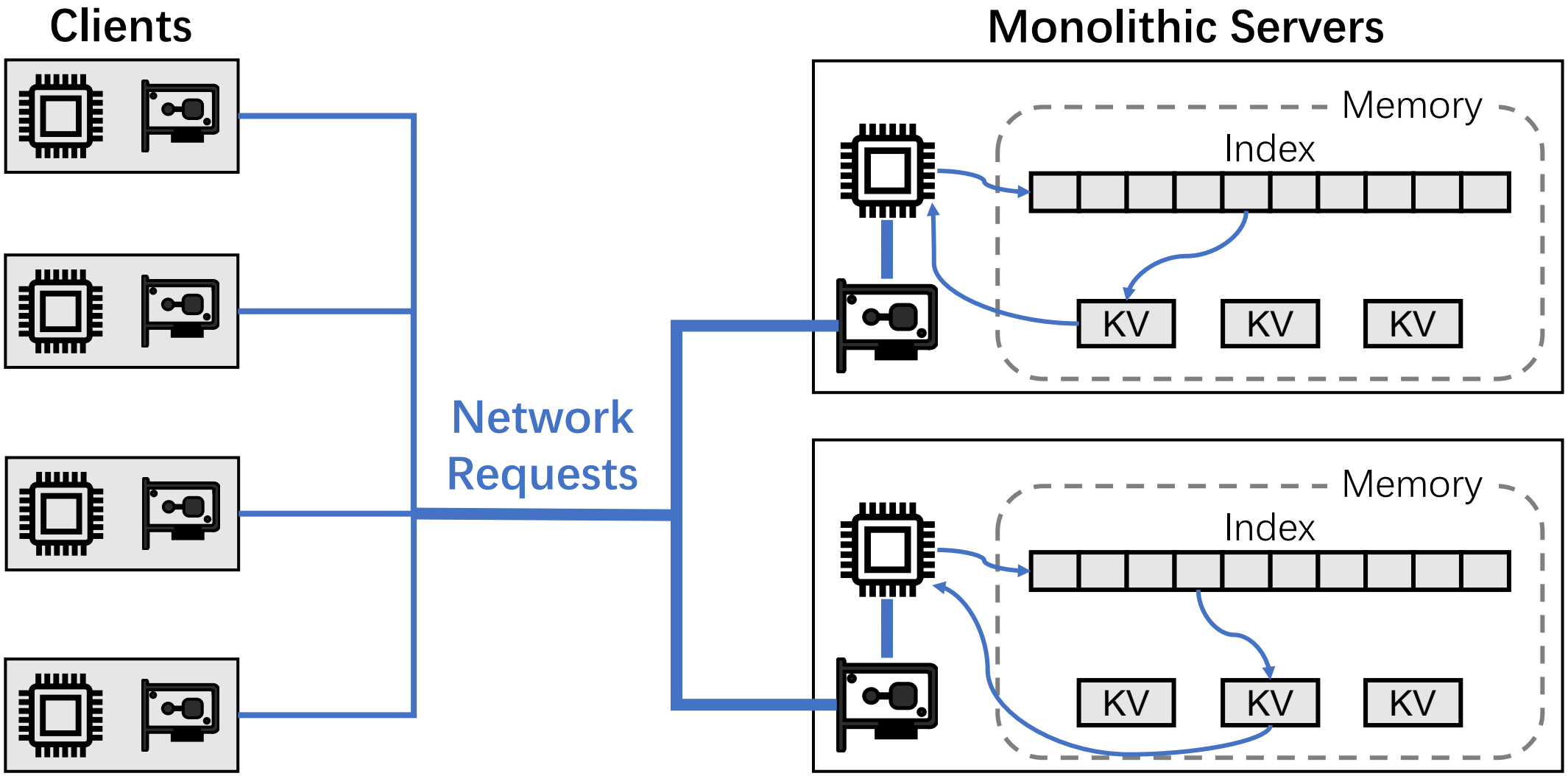
- Databases
- File systems
- Serverless platforms
- …



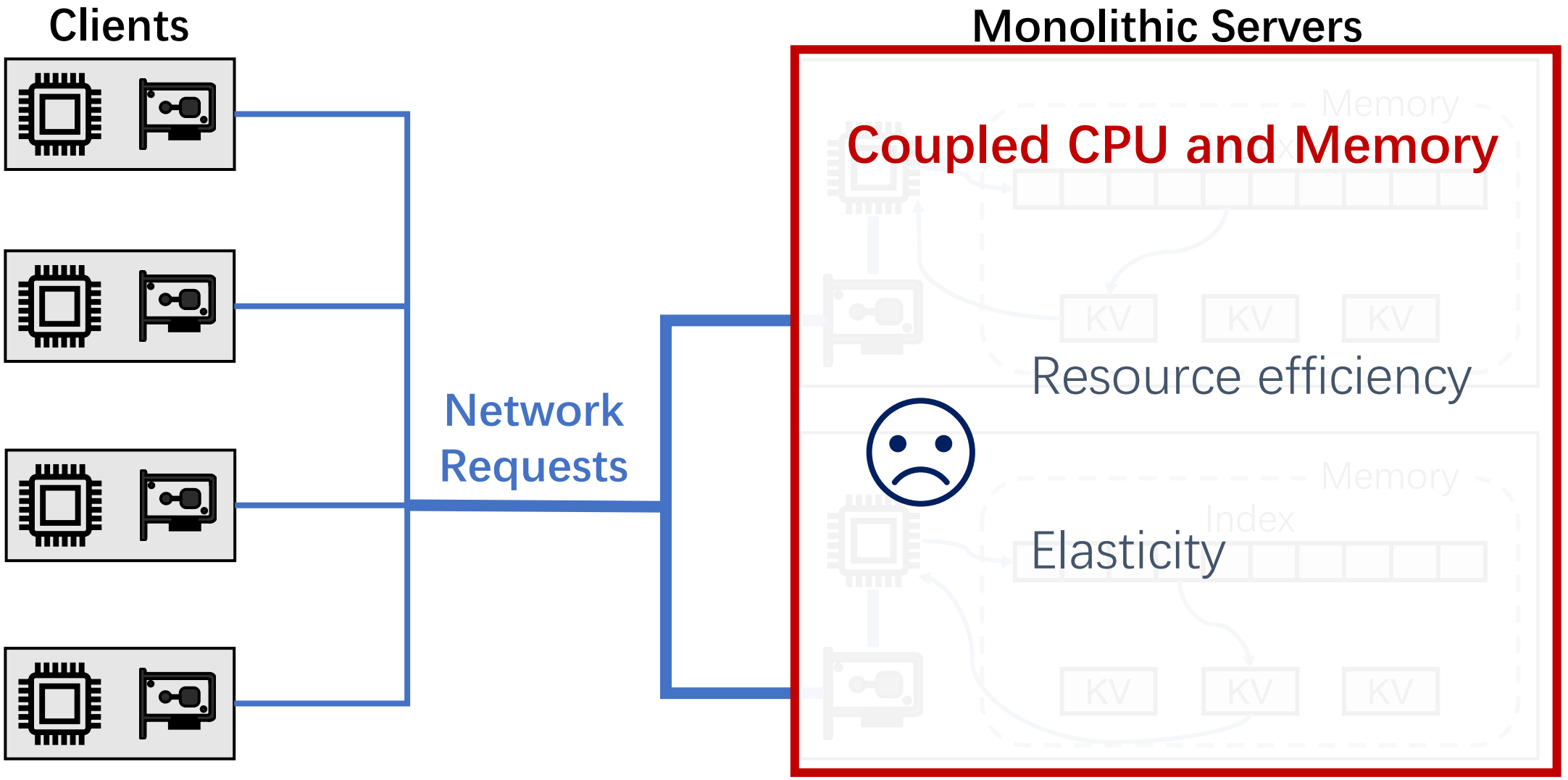




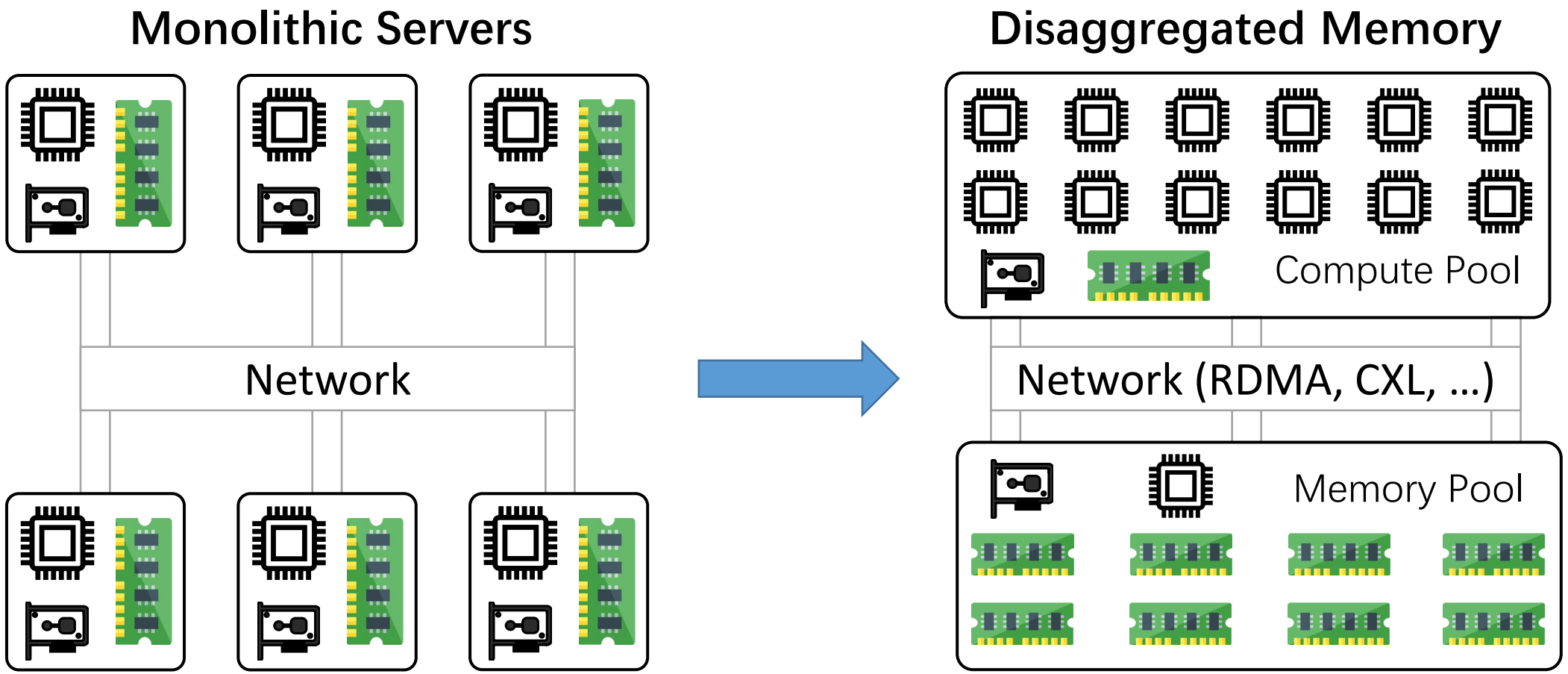
In-Memory Key-Value Stores



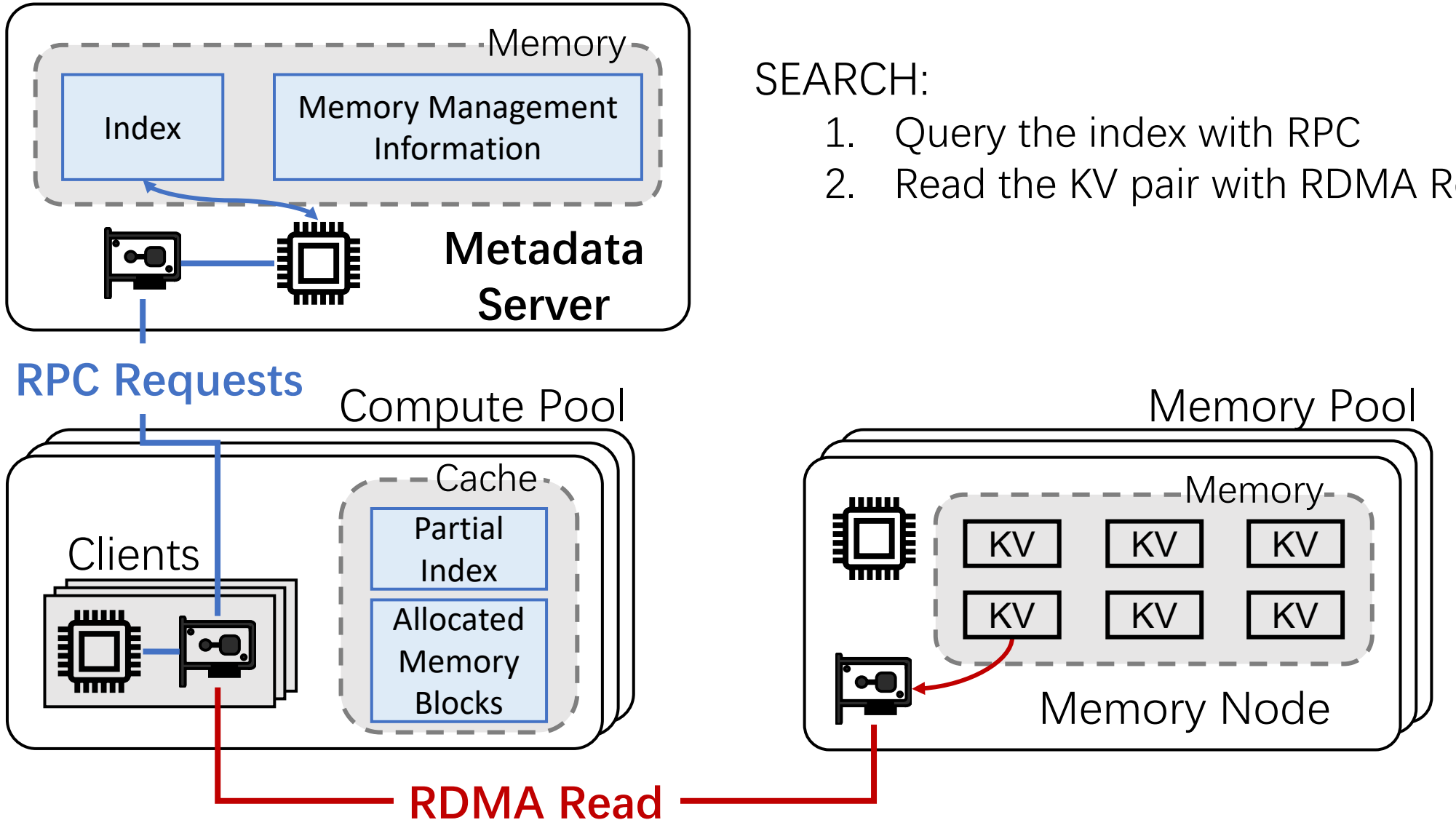
In-Memory Key-Value Stores



Disaggregated Memory



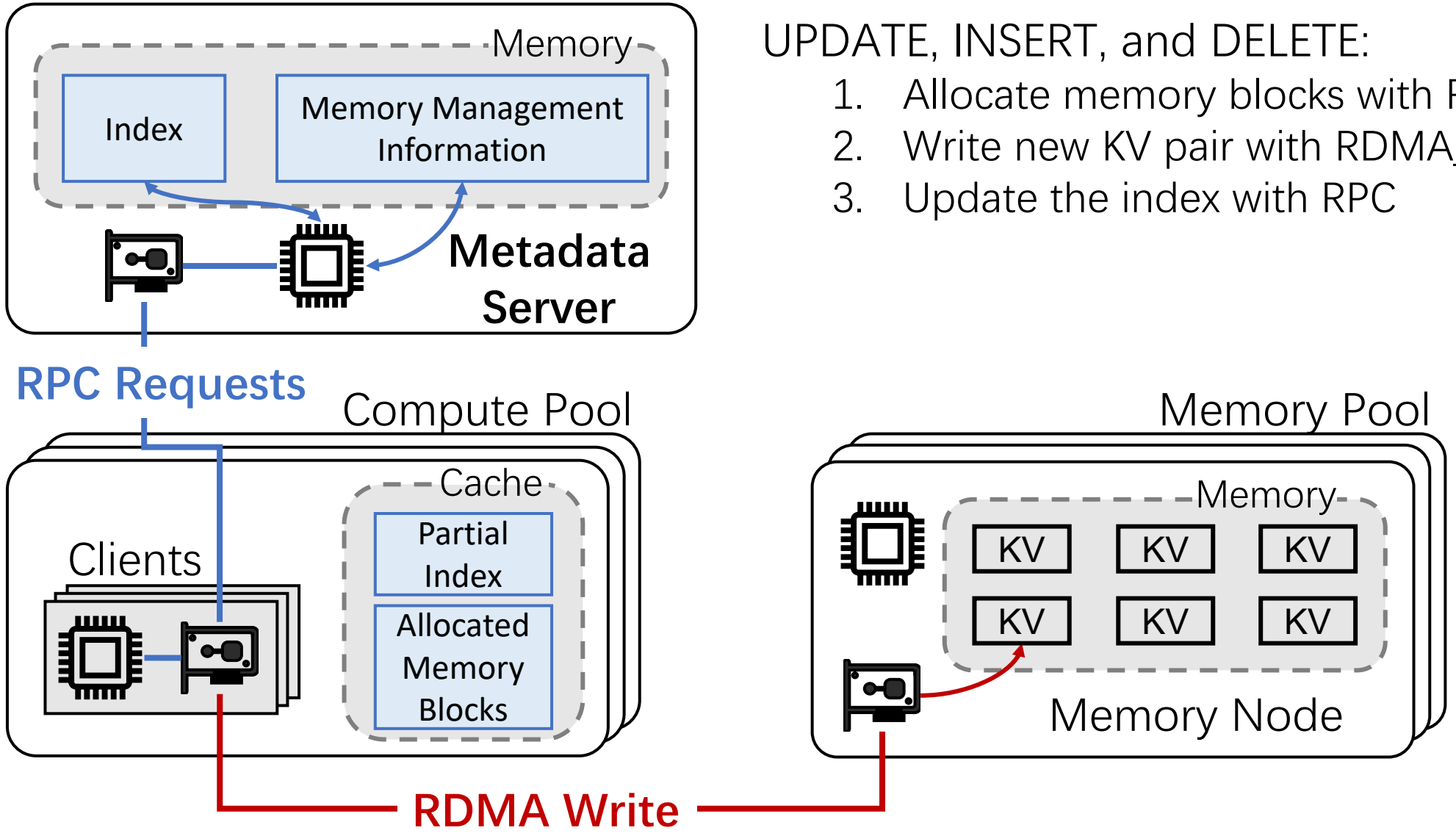
Existing Semi-Disaggregated Key-Value Stores



SEARCH:

1. Query the index with RPC
2. Read the KV pair with RDMA Read

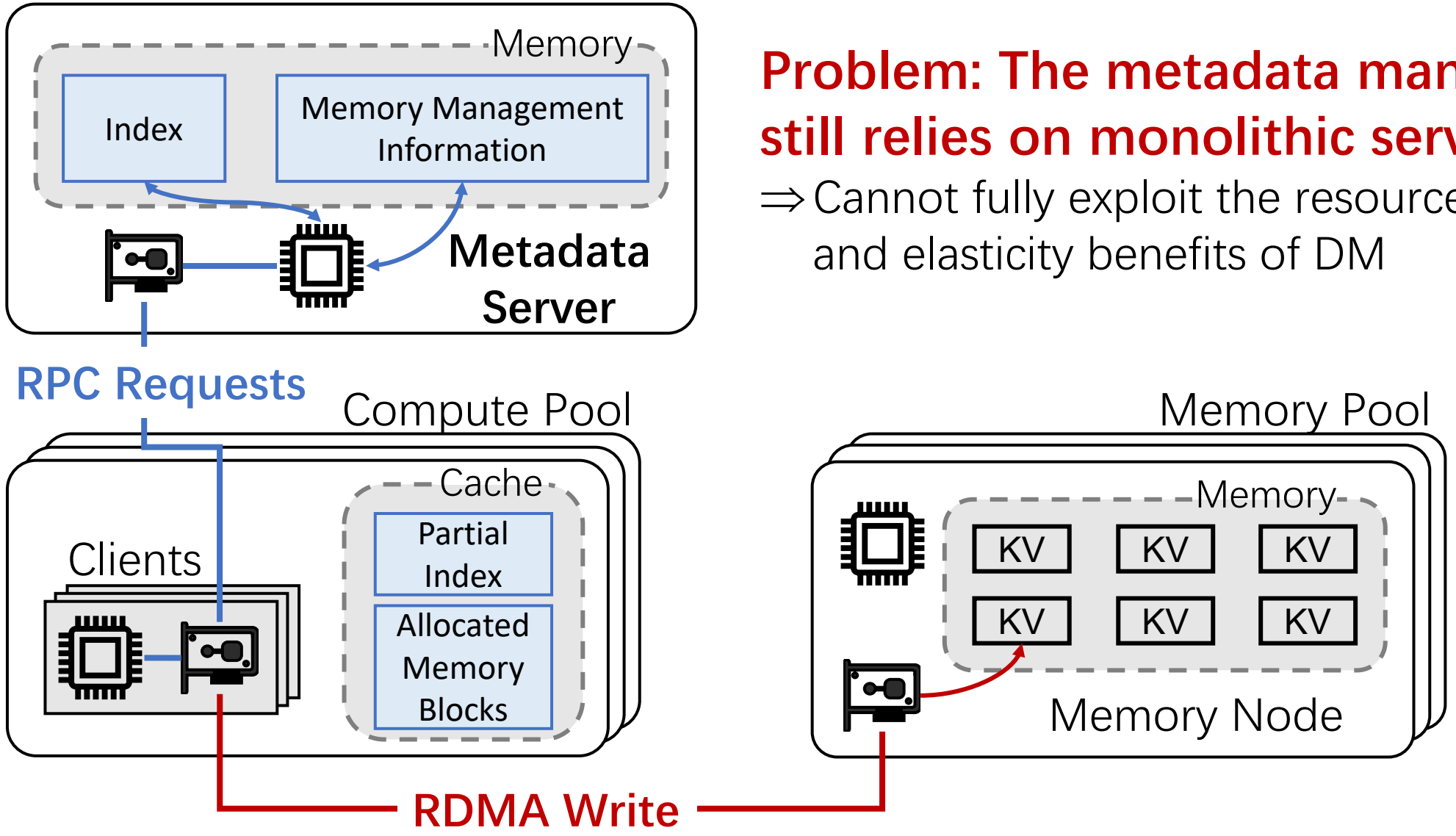
Existing Semi-Disaggregated Key-Value Stores



UPDATE, INSERT, and DELETE:

1. Allocate memory blocks with RPC
2. Write new KV pair with RDMA_WRITE
3. Update the index with RPC

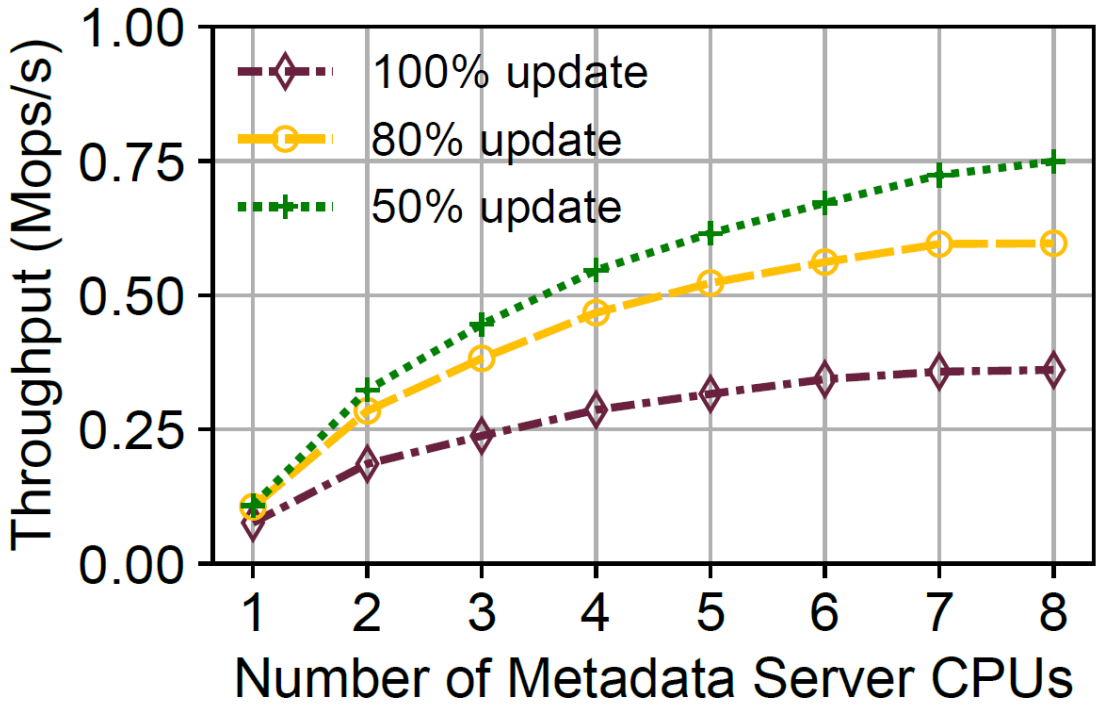
Existing Semi-Disaggregated Key-Value Stores



Problem: The metadata management still relies on monolithic servers!

⇒ Cannot fully exploit the resource efficiency and elasticity benefits of DM

Existing Semi-Disaggregated Key-Value Stores



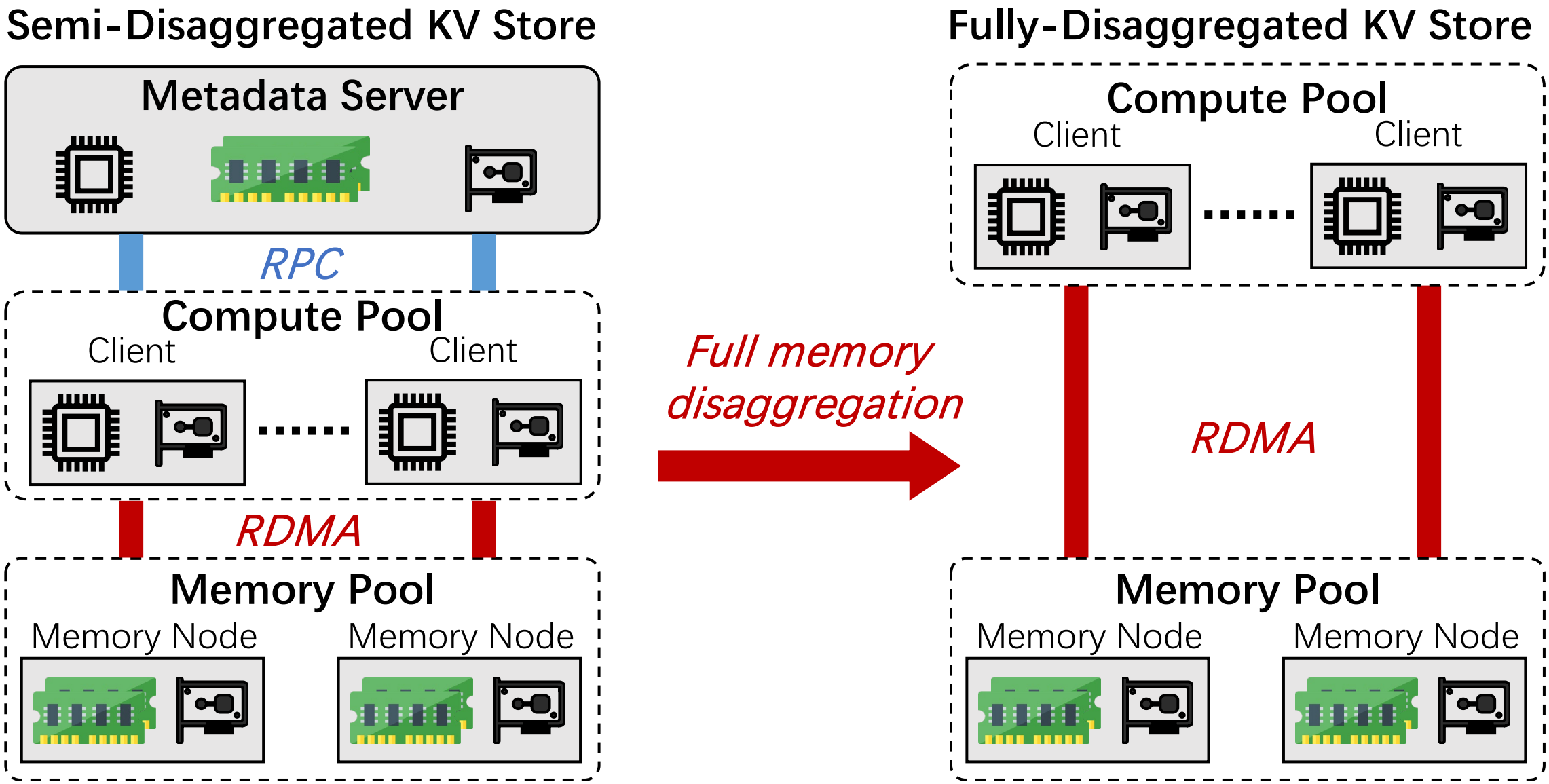
Problem: The metadata management still relies on monolithic servers!

⇒ Cannot fully exploit the resource efficiency and elasticity benefits of monolithic servers

Additional resource required:

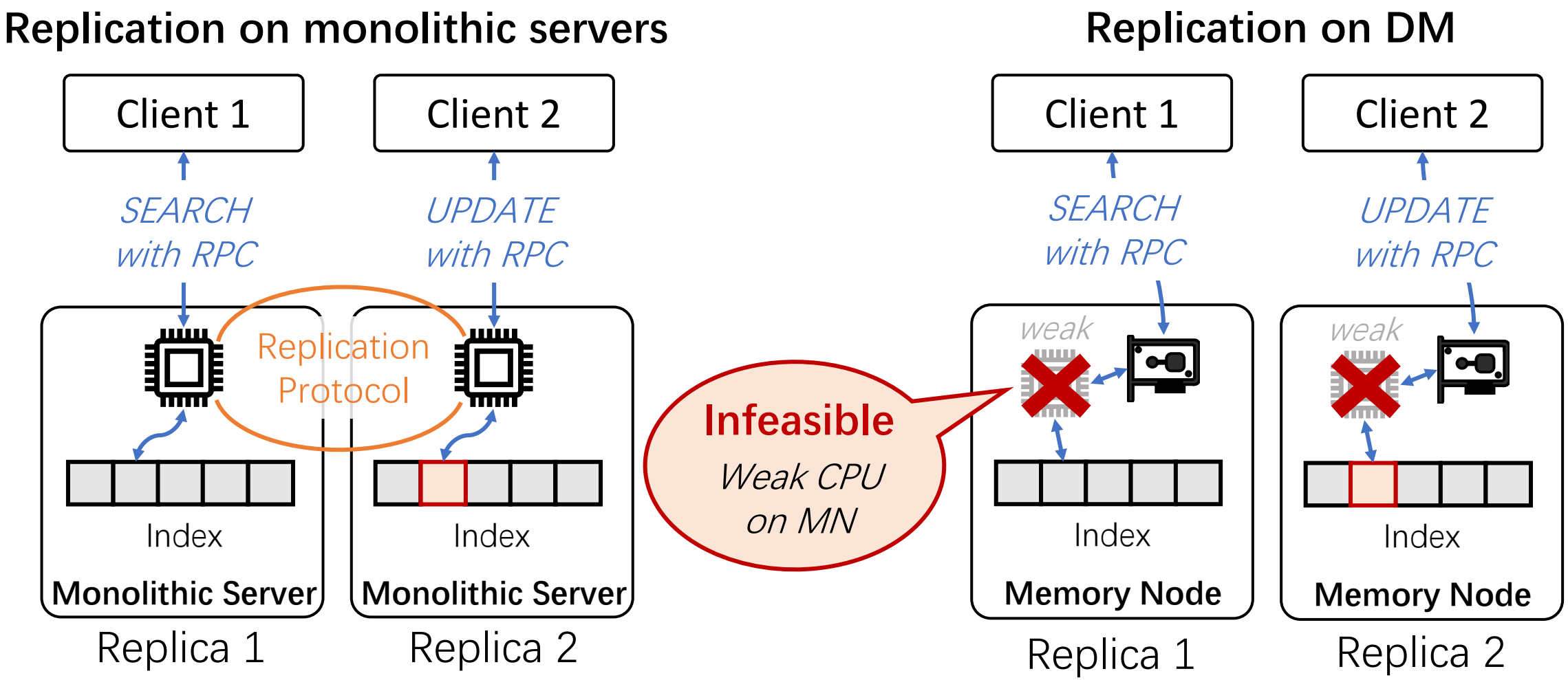
At least *6 CPU* cores and *1 RNIC*

Is it possible to manage metadata directly with clients?



Challenge 1: Client-Centric Index Replication

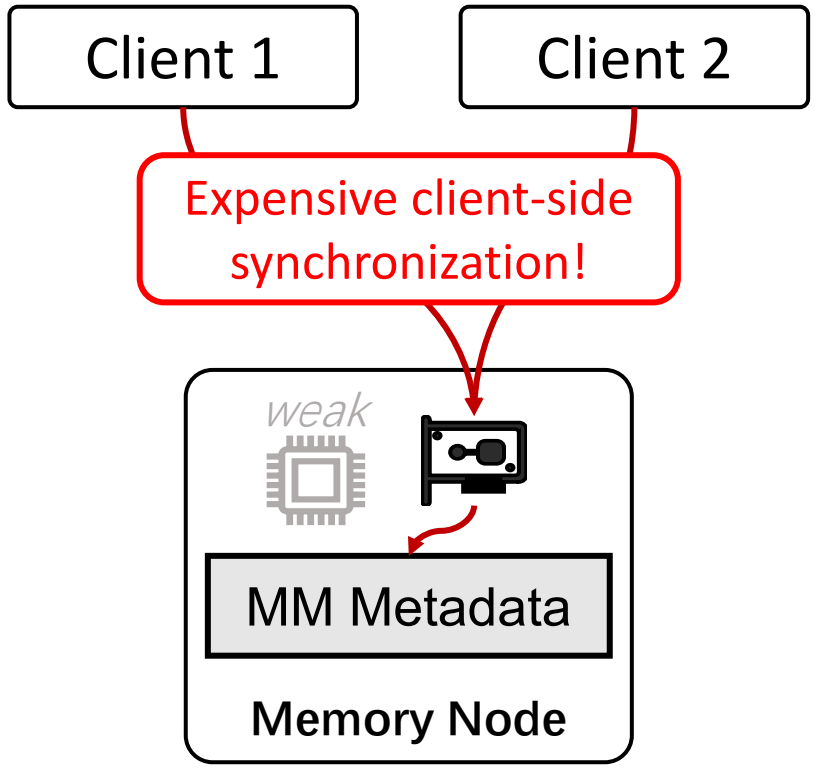
How to achieve strong consistency?



Challenge 2: Remote Memory Allocation

Where to execute the memory management computation?

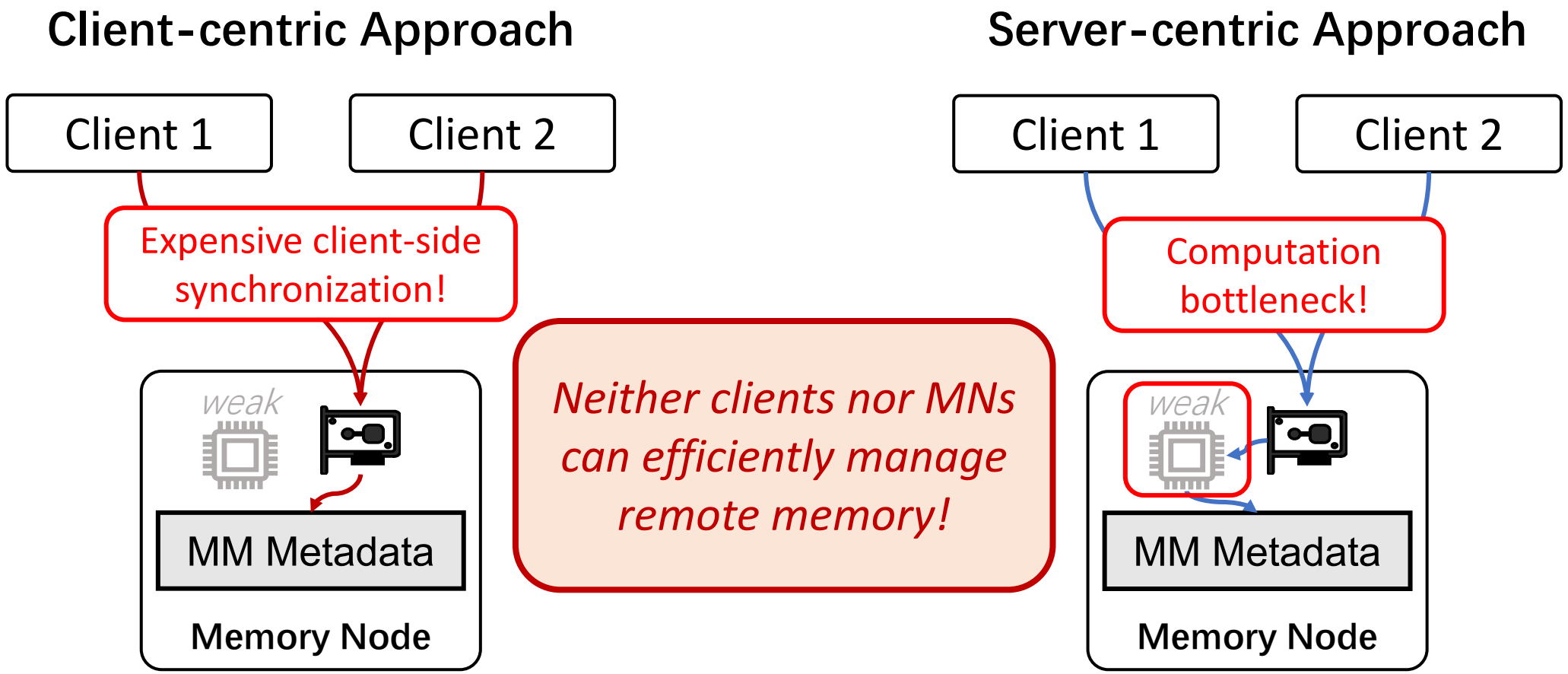
Client-centric Approach



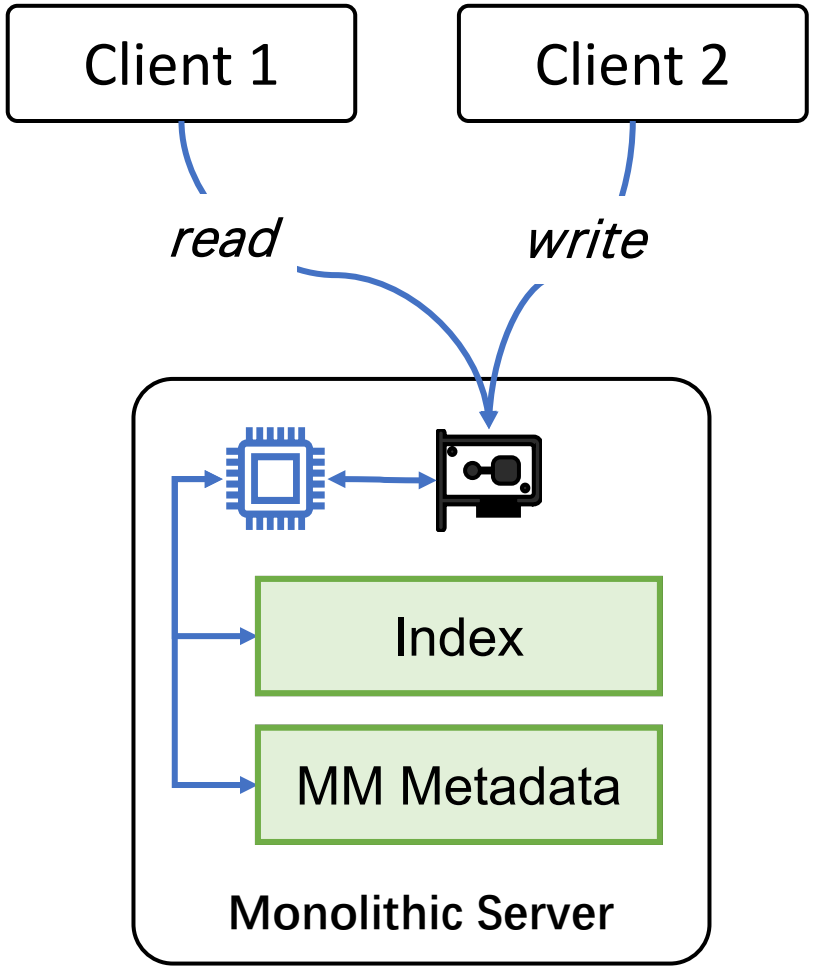
Server-centric Approach

Challenge 2: Remote Memory Allocation

Where to execute the memory management computation?



Challenge 3: Metadata Corruption

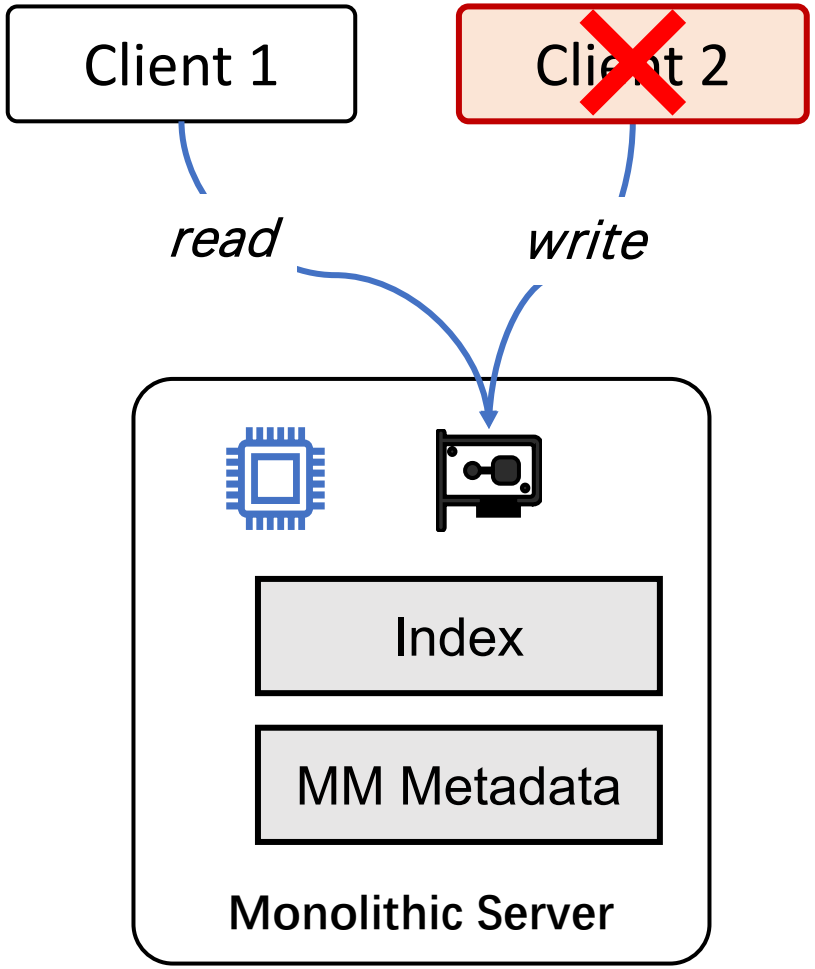


Clients may crash at any time!

Client Failure on monolithic-server-based KV stores:

- **CPUs** of the KV store server exclusively access and modify metadata

Challenge 3: Metadata Corruption

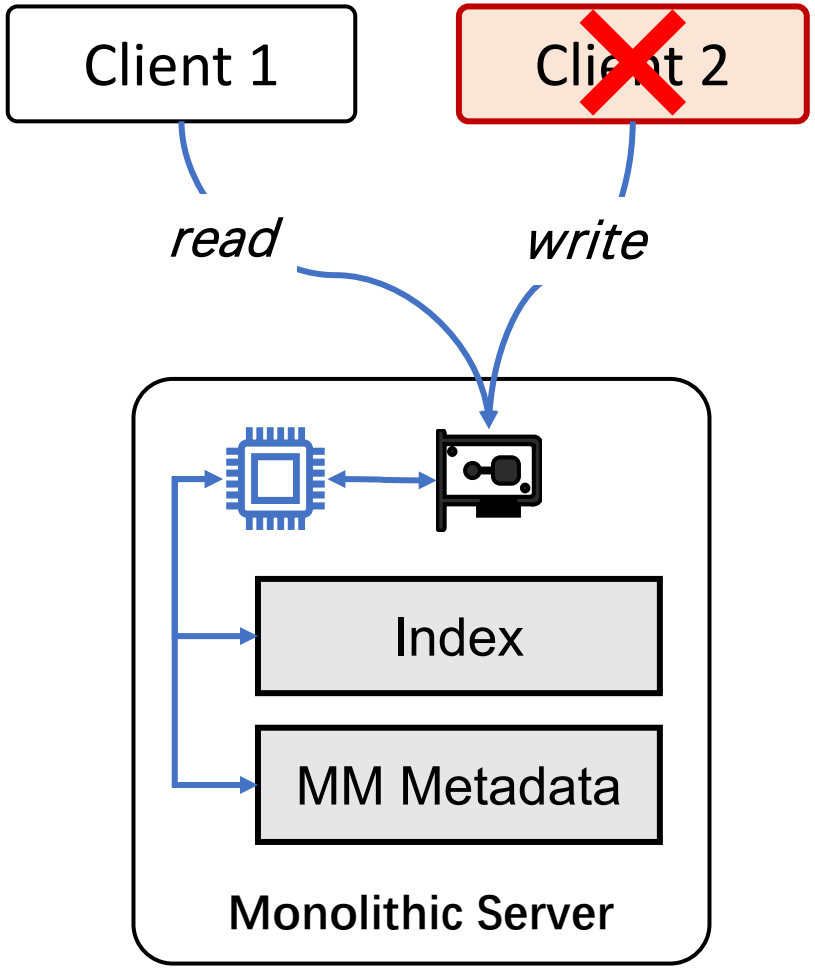


Clients may crash at any time!

Client Failure on monolithic-server-based KV stores:

- **CPUs** of the KV store server exclusively access and modify metadata

Challenge 3: Metadata Corruption

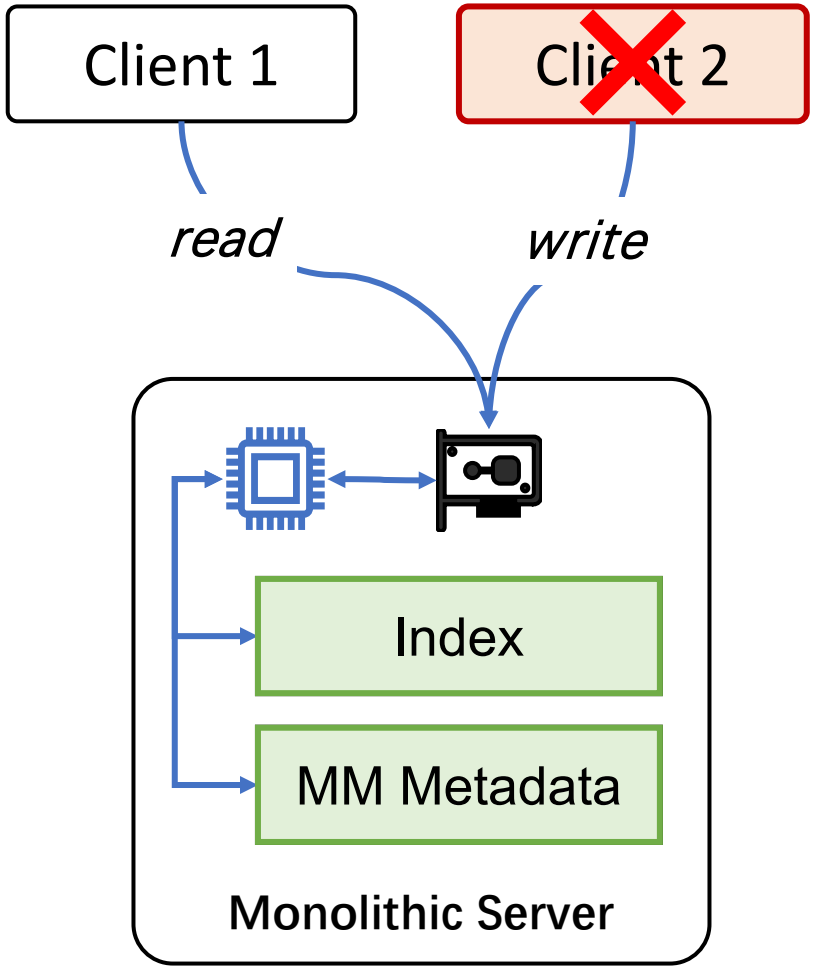


Clients may crash at any time!

Client Failure on monolithic-server-based KV stores:

- **CPUs** of the KV store server exclusively access and modify metadata

Challenge 3: Metadata Corruption

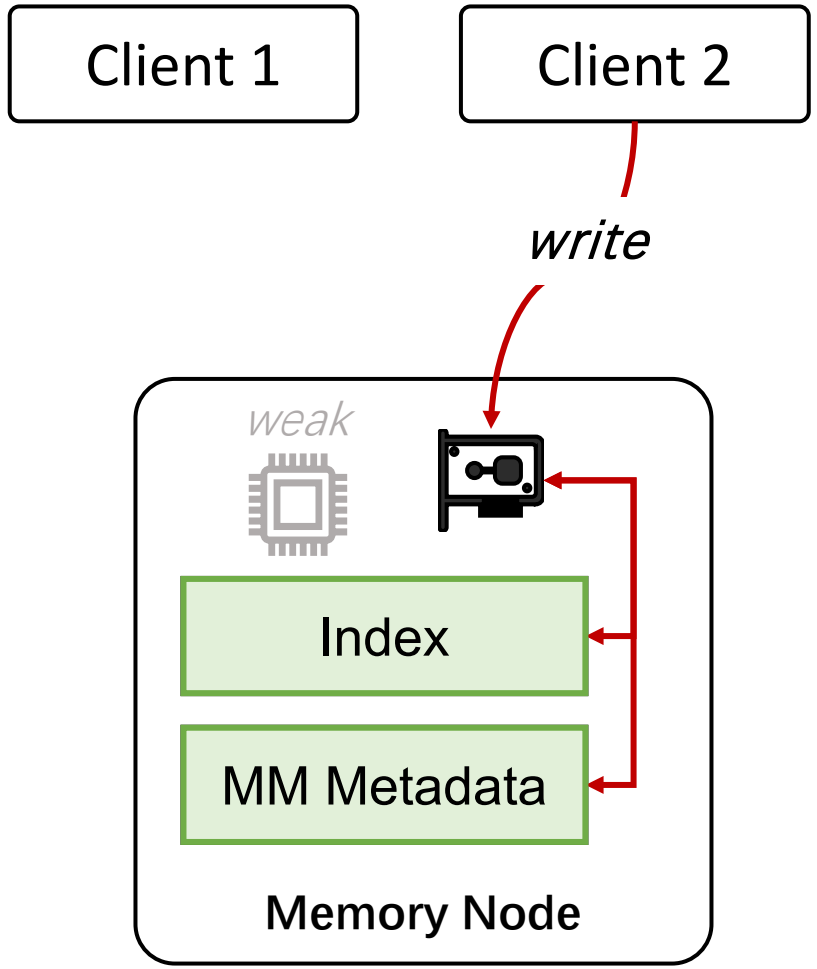


Clients may crash at any time!

Client Failure on monolithic-server-based KV stores:

- **CPUs** of the KV store server exclusively access and modify metadata
- Metadata are not affected by client failures!

Challenge 3: Metadata Corruption

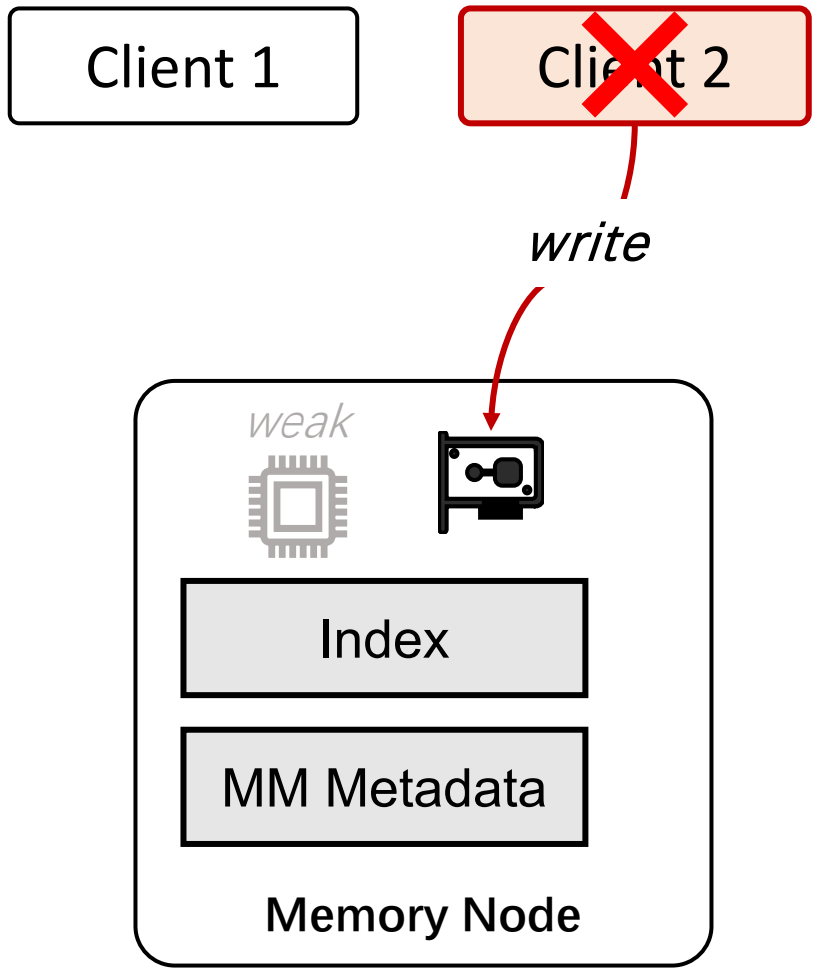


Clients may crash at any time!

Client Failure on fully memory-disaggregated KV stores:

- **Clients** directly access and modify metadata with **one-sided RDMA verbs**

Challenge 3: Metadata Corruption

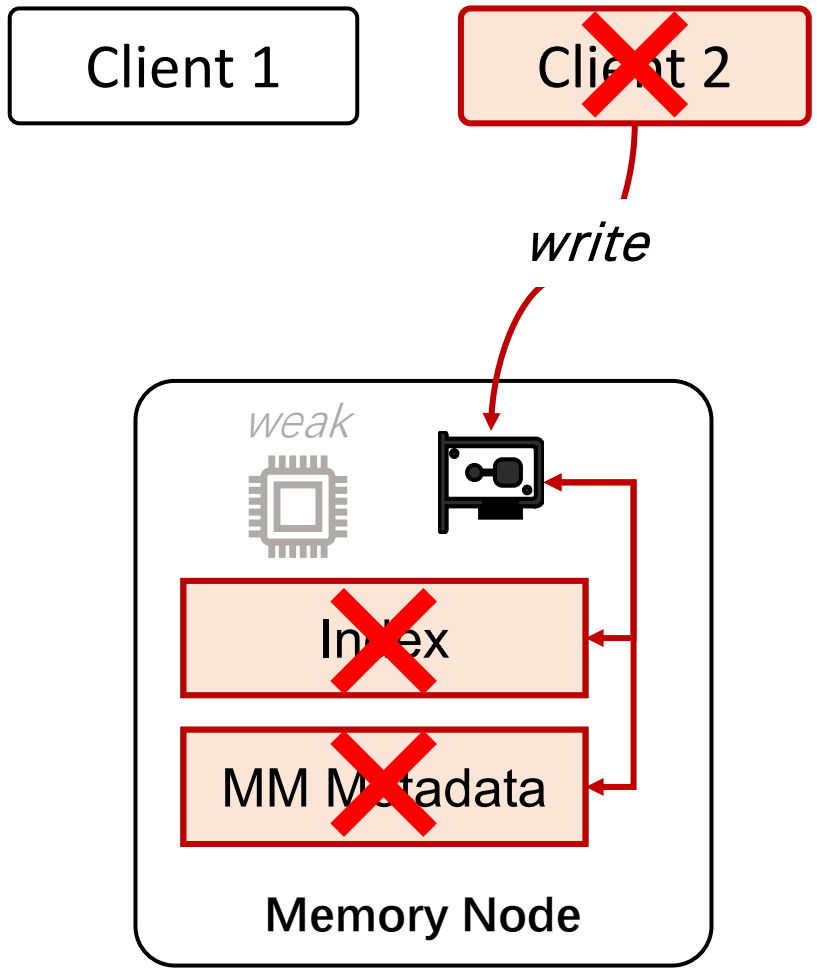


Clients may crash at any time!

Client Failure on fully memory-disaggregated KV stores:

- **Clients** directly access and modify metadata with **one-sided RDMA verbs**

Challenge 3: Metadata Corruption

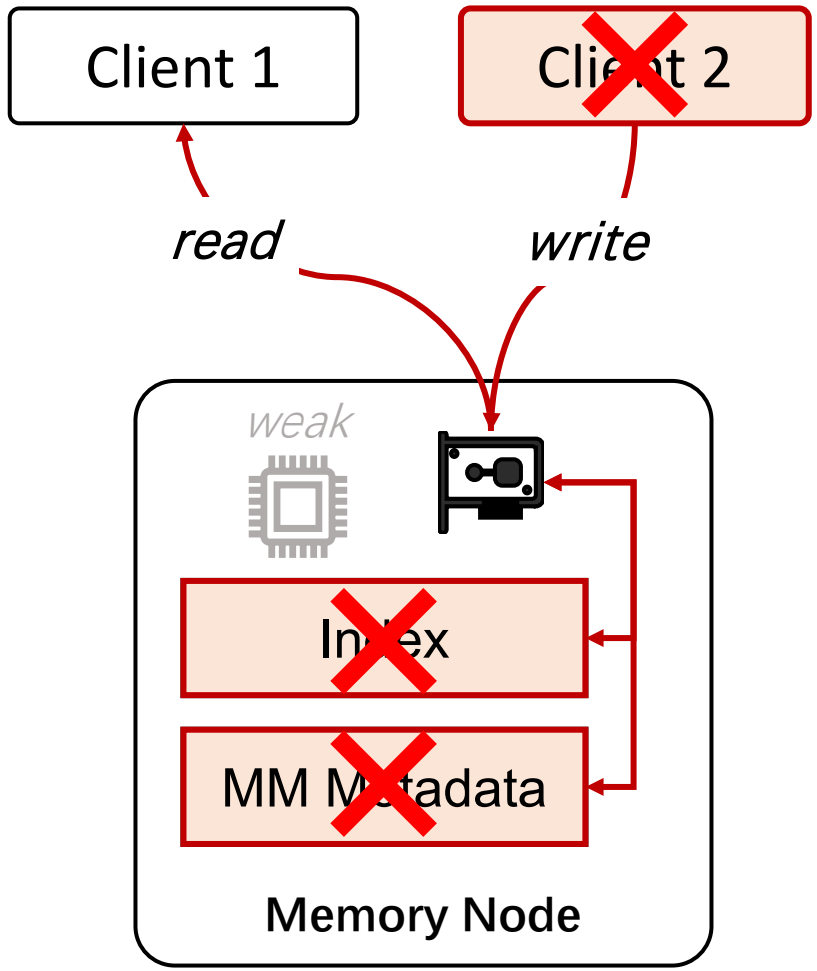


Clients may crash at any time!

Client Failure on fully memory-disaggregated KV stores:

- **Clients** directly access and modify metadata with **one-sided RDMA verbs**
- Metadata will be corrupted by clients' partial updates and accessed by other clients!

Challenge 3: Metadata Corruption



Clients may crash at any time!

Client Failure on fully memory-disaggregated KV stores:

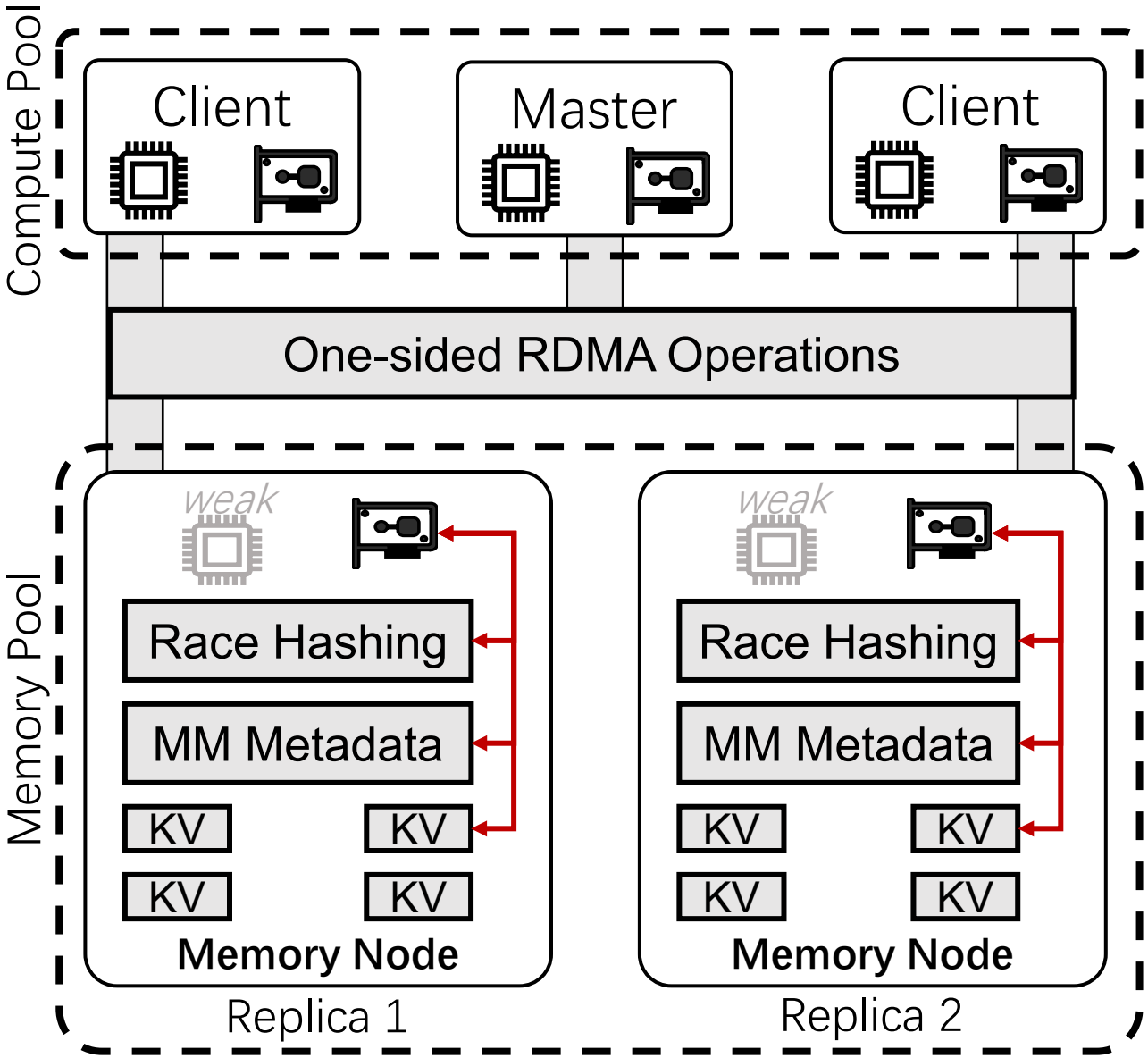
- **Clients** directly access and modify metadata with **one-sided RDMA verbs**
- Metadata will be corrupted by clients' partial updates and accessed by other clients!

Challenges

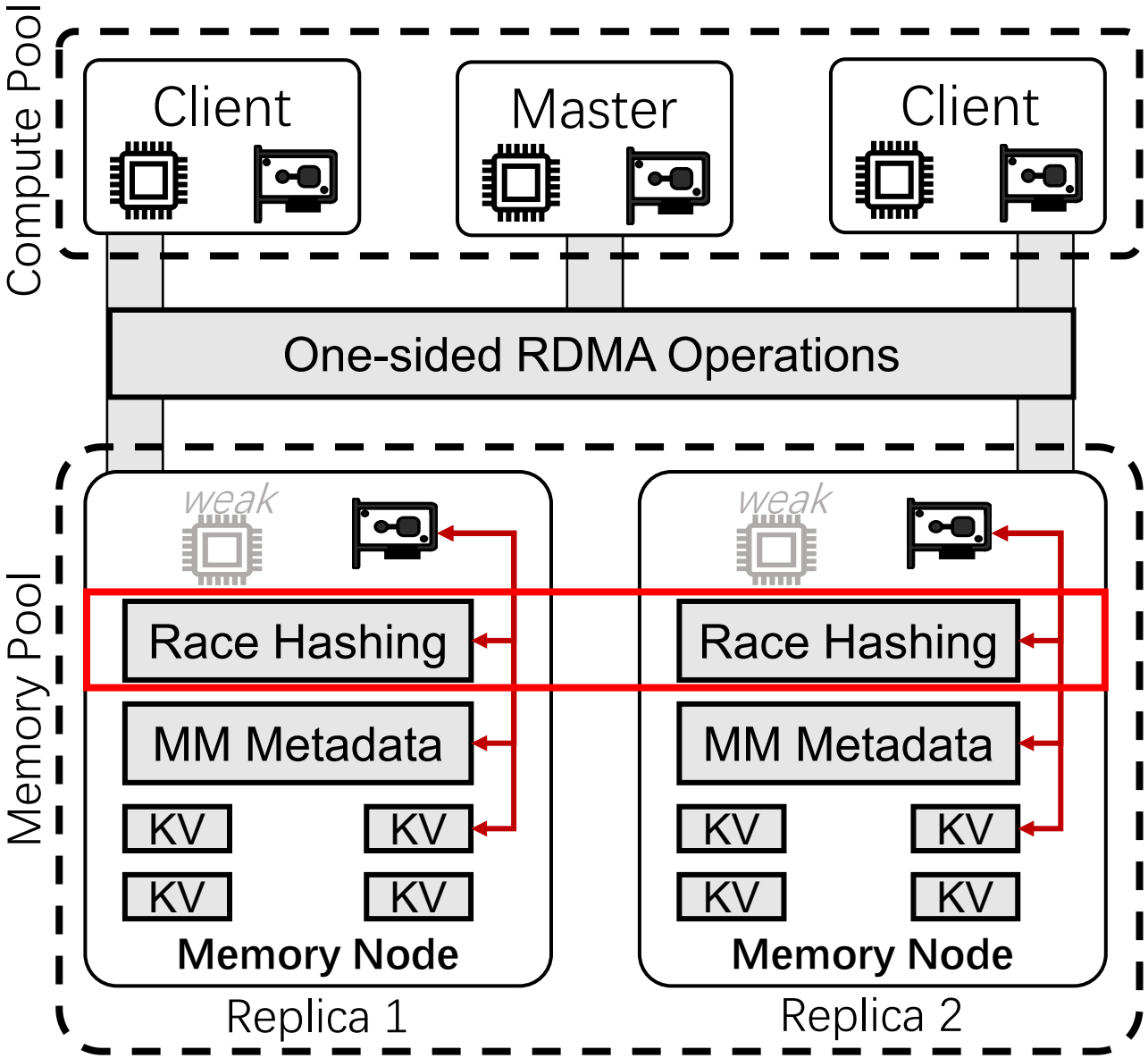
Summary

- Client-centric index replication
  - Weak MN-side compute power cannot execute replication protocols
- Remote memory allocation
  - Weak MN-side compute power cannot efficiently allocate memory spaces
- Metadata corruption
  - Client-failures compromises the correctness of the entire KV store

FUSEE Overview

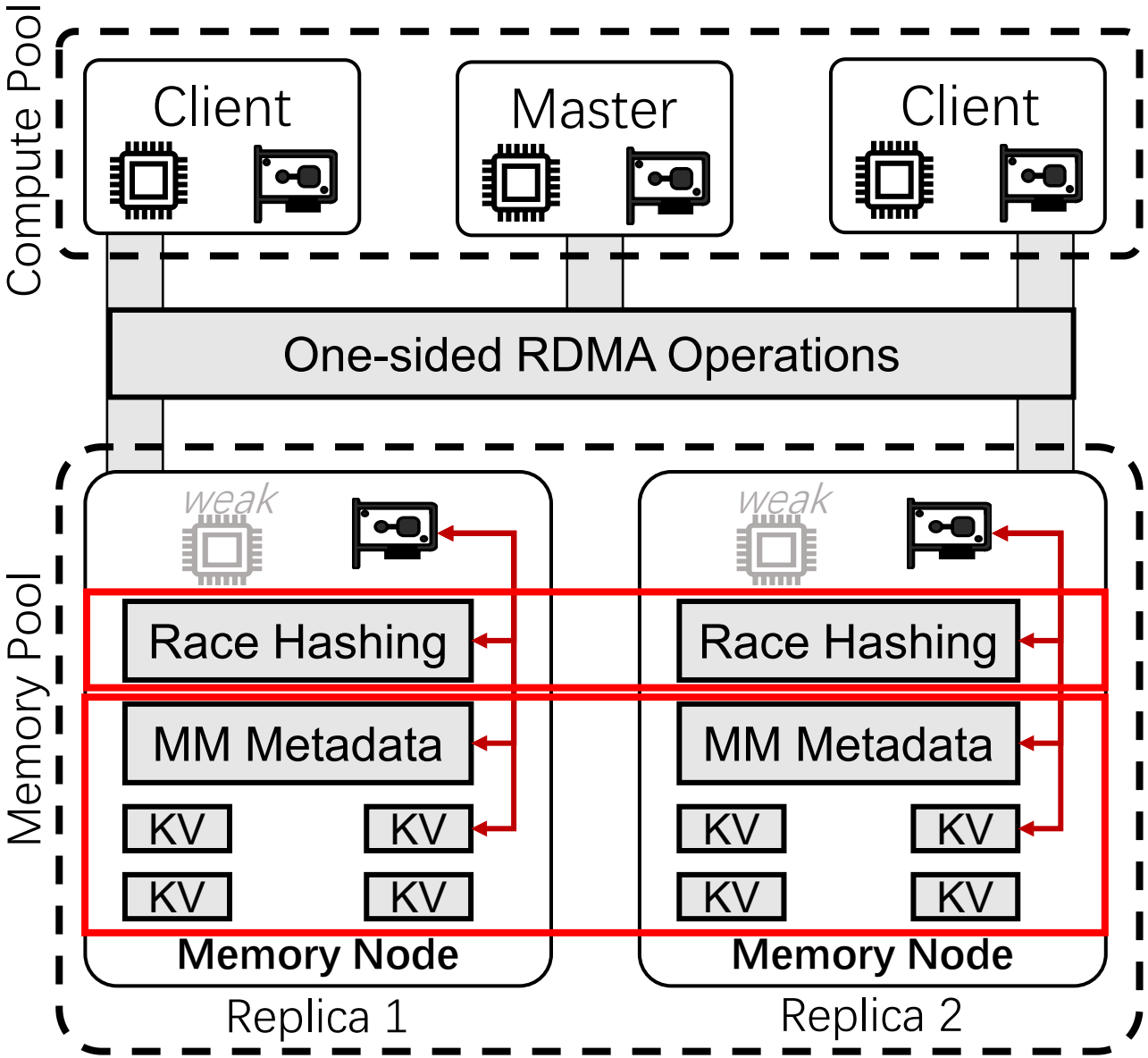


FUSEE Overview



Client-centric index replication:
The SNAPSHOT protocol

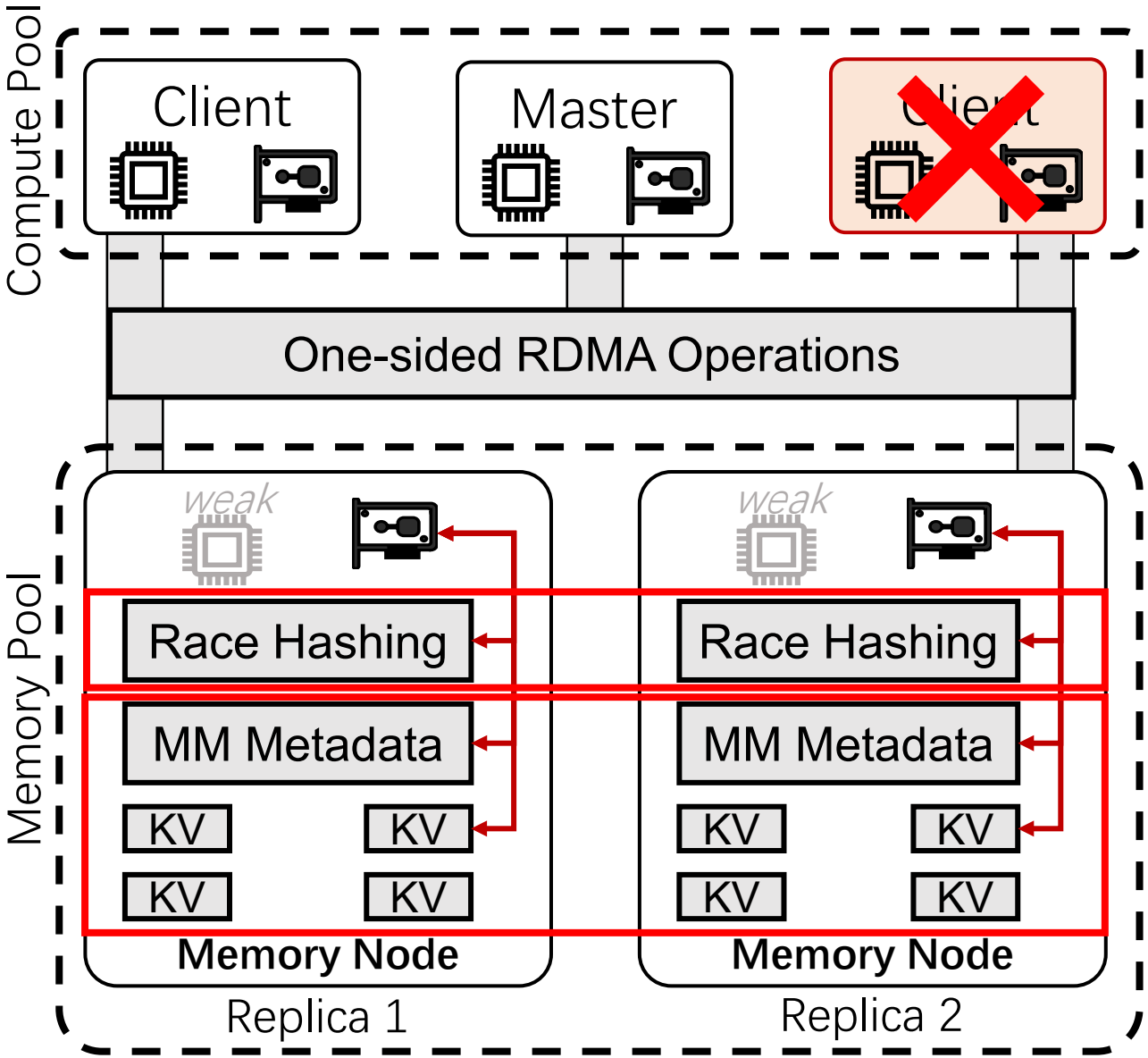
FUSEE Overview



Client-centric index replication:
The SNAPSHOT protocol

Remote memory allocation:
Two-level memory allocation

FUSEE Overview



Metadata corruption:
Embedded operation log

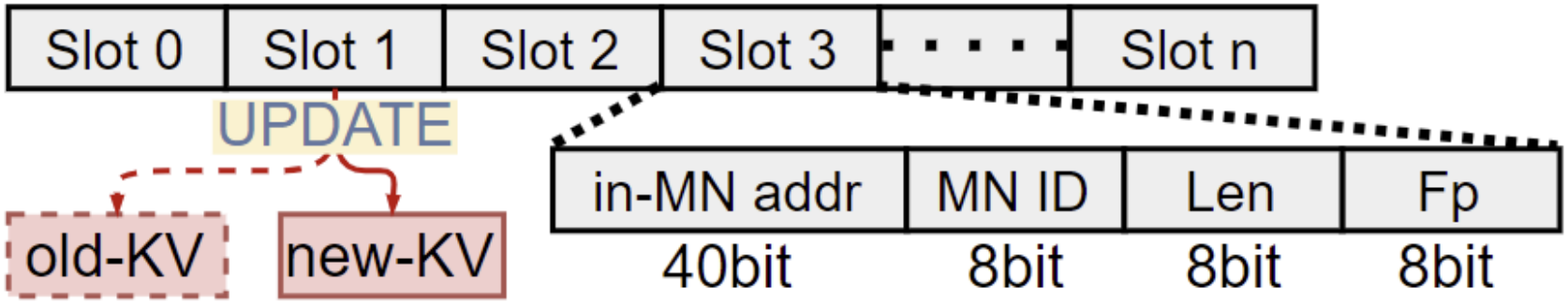
Client-centric index replication:
The SNAPSHOT protocol

Remote memory allocation:
Two-level memory allocation

FUSEE Overview

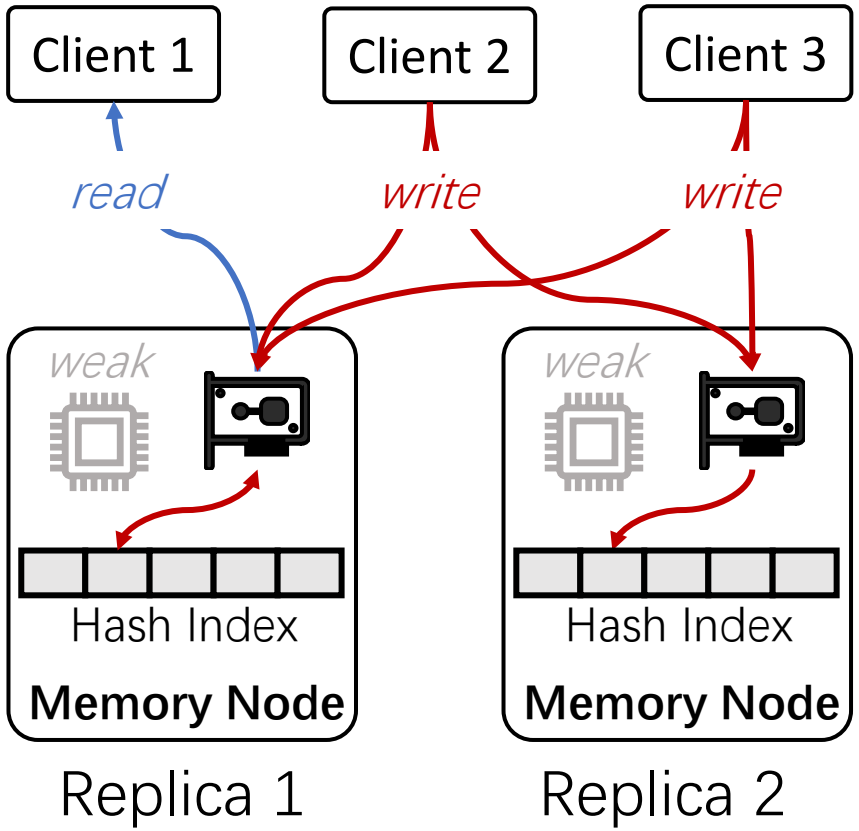
RACE Hashing

- One-sided RDMA-friendly hash index structure [ATC 21]
- Multiple 8-byte slots
- Out-of-place modification scheme



Client-Centric Index Replication

Index Replication on DM



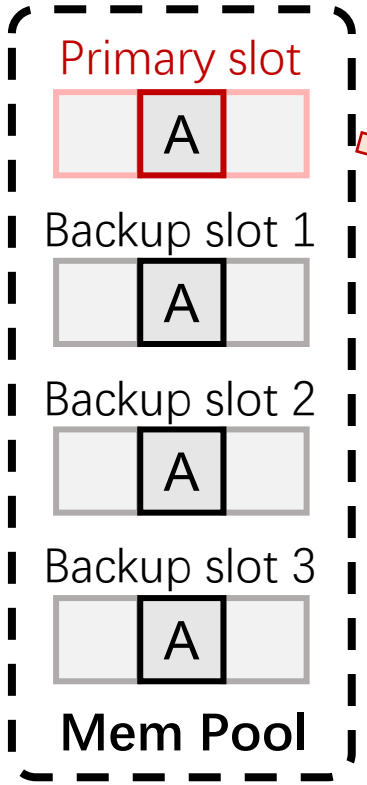
Two key problems:

- How to protect readers from reading incomplete writes?
- How to efficiently resolve write-write conflicts among clients?

Client-Centric Index Replication

How to protect readers from reading incomplete values?

- Separate index replicas into primary and backups
- Resolve write-write conflicts on backup replicas



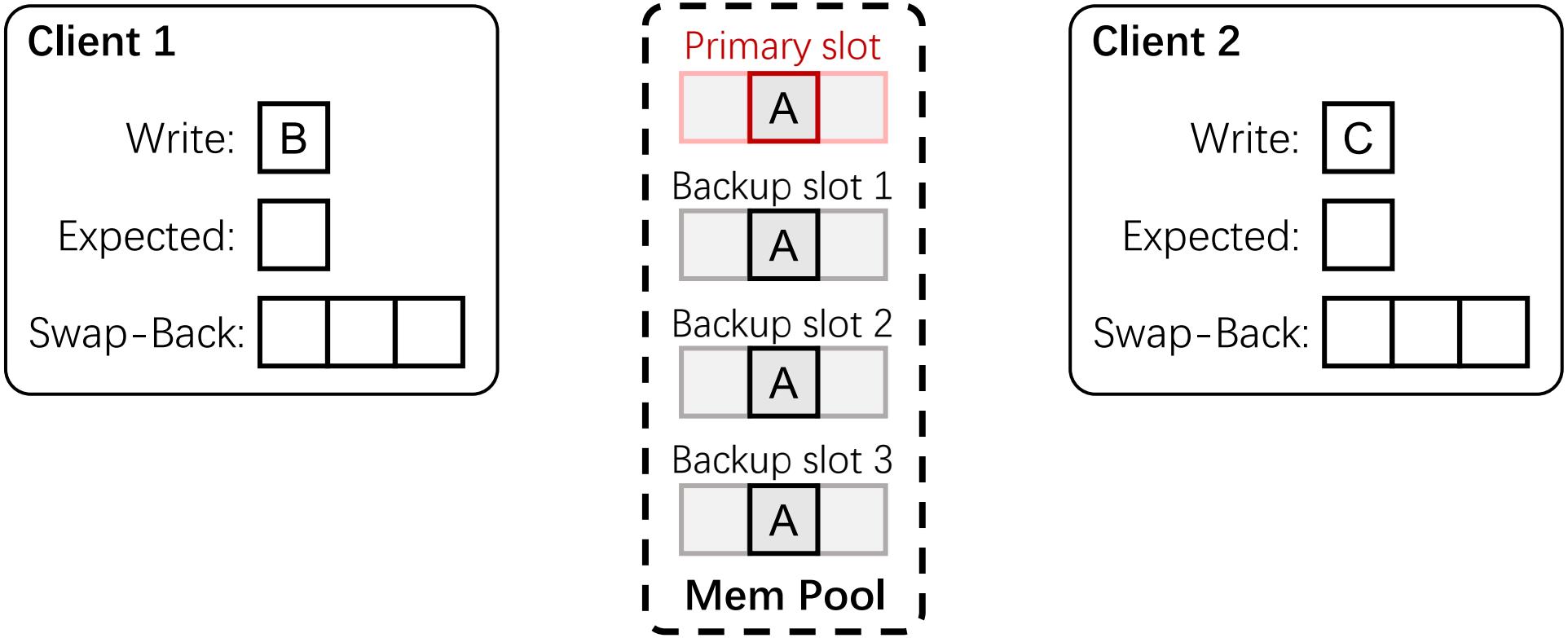
Always contain the correct value!

Directly read with RDMA_READ.

Client-Centric Index Replication

How to efficiently resolve write-write conflicts on clients?

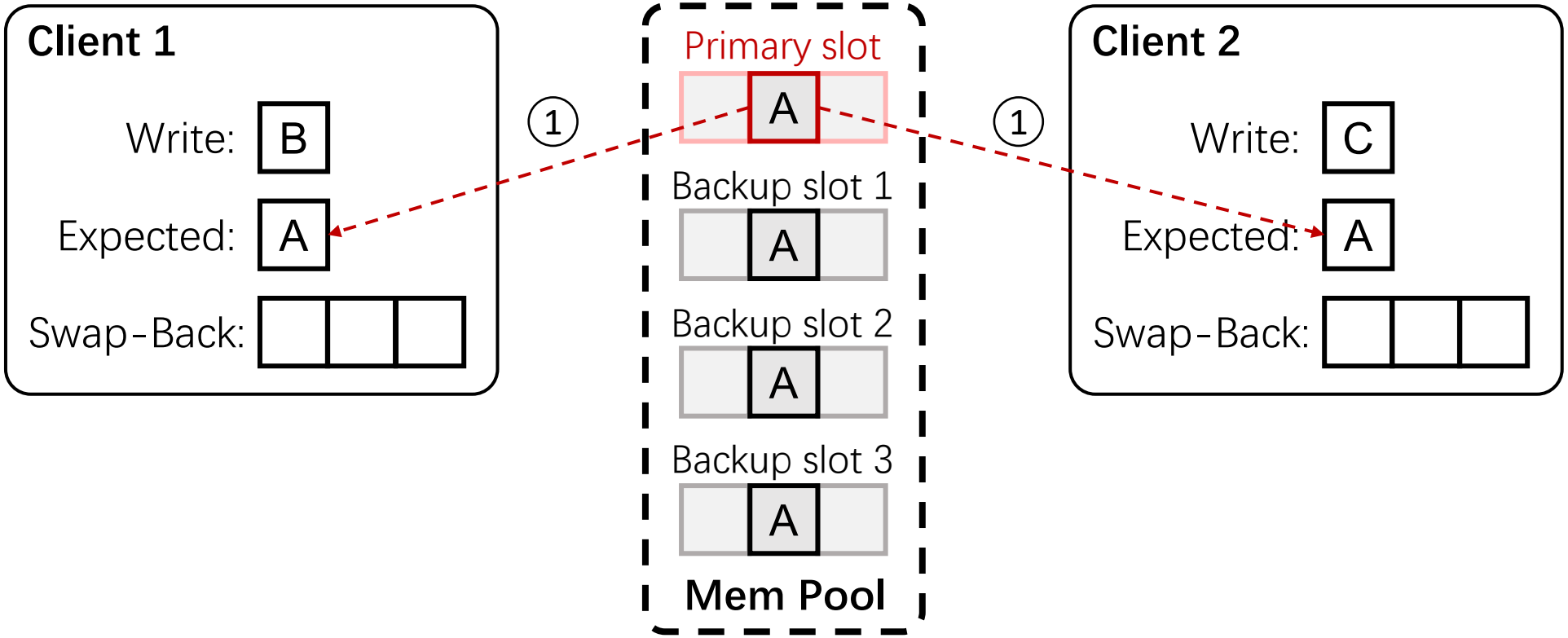
- Out-of-place KV modification ⇨ Conflict clients write different values
- Last-writer-wins conflict resolution ⇨ Simple rules on client sides



Client-Centric Index Replication

How to efficiently resolve write-write conflicts on clients?

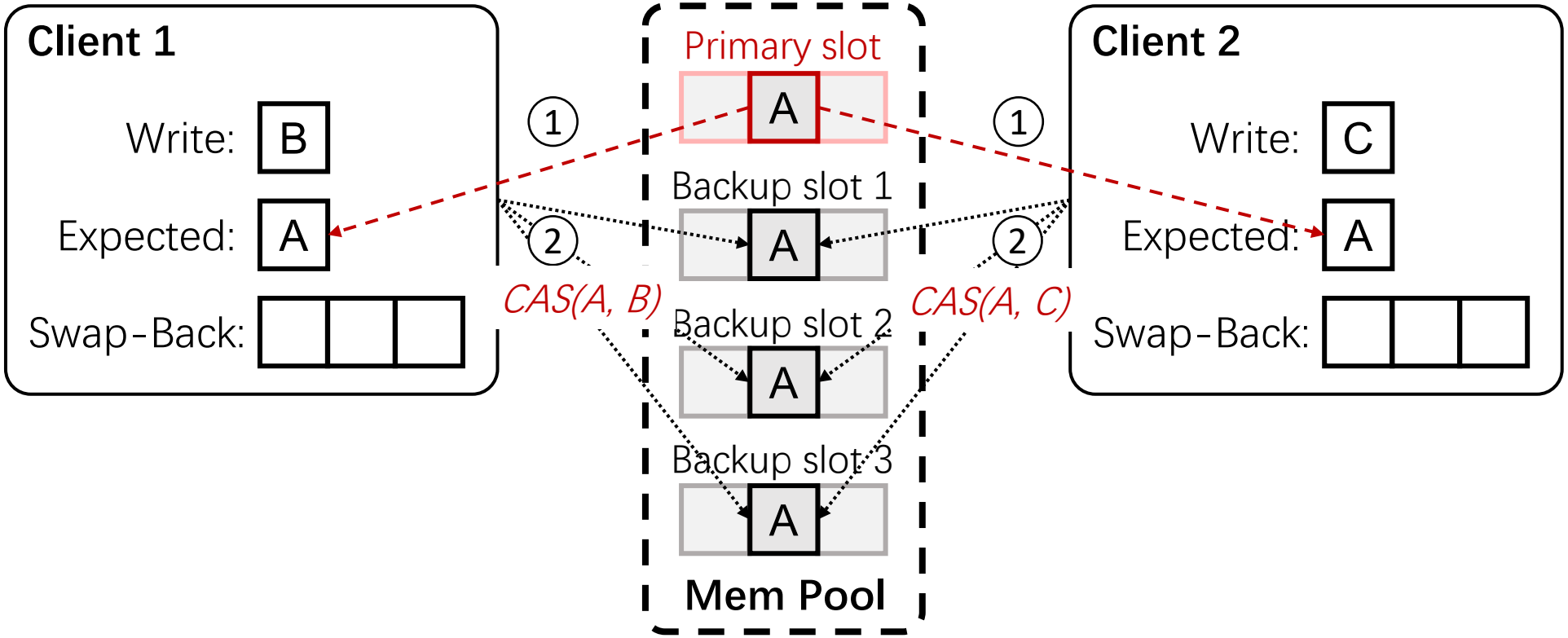
- Out-of-place KV modification ⇨ Conflict clients write different values
- Last-writer-wins conflict resolution ⇨ Simple rules on client sides



Client-Centric Index Replication

How to efficiently resolve write-write conflicts on clients?

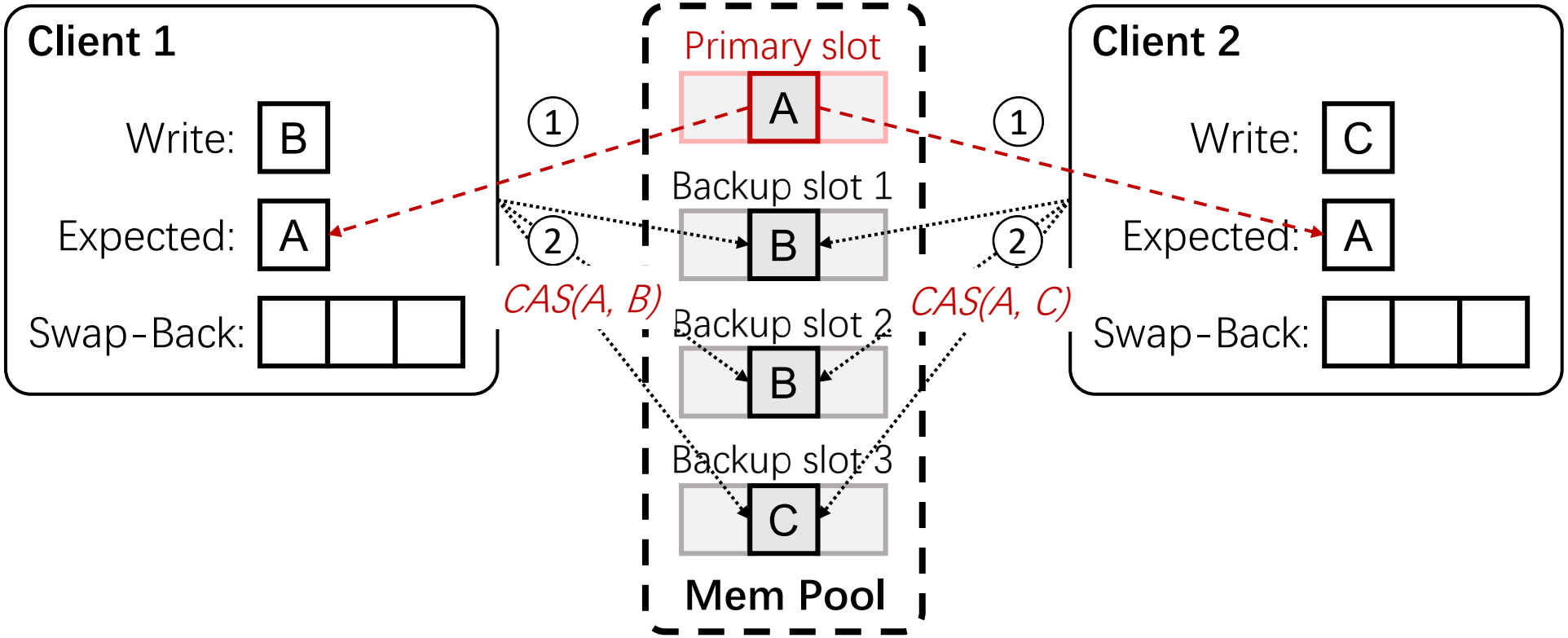
- Out-of-place KV modification ⇨ Conflict clients write different values
- Last-writer-wins conflict resolution ⇨ Simple rules on client sides



Client-Centric Index Replication

How to efficiently resolve write-write conflicts on clients?

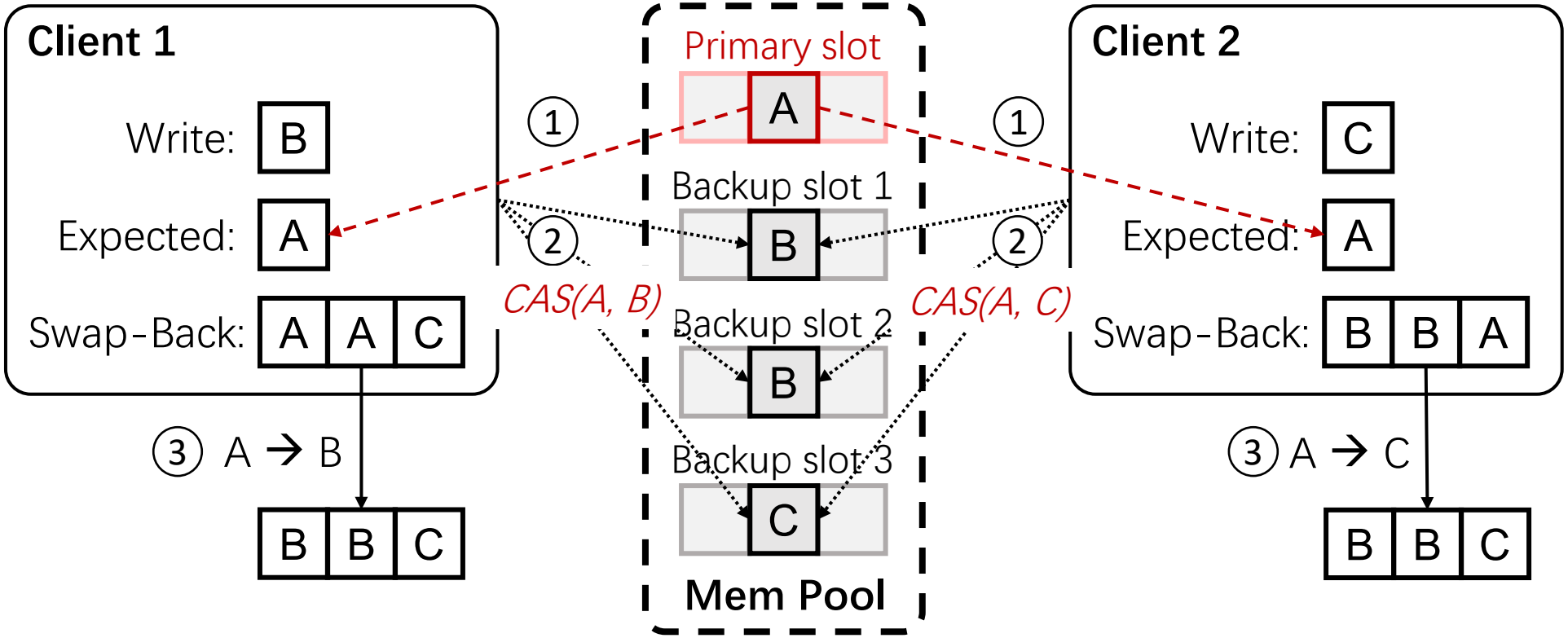
- Out-of-place KV modification ⇨ Conflict clients write different values
- Last-writer-wins conflict resolution ⇨ Simple rules on client sides



Client-Centric Index Replication

A client is a last writer if:

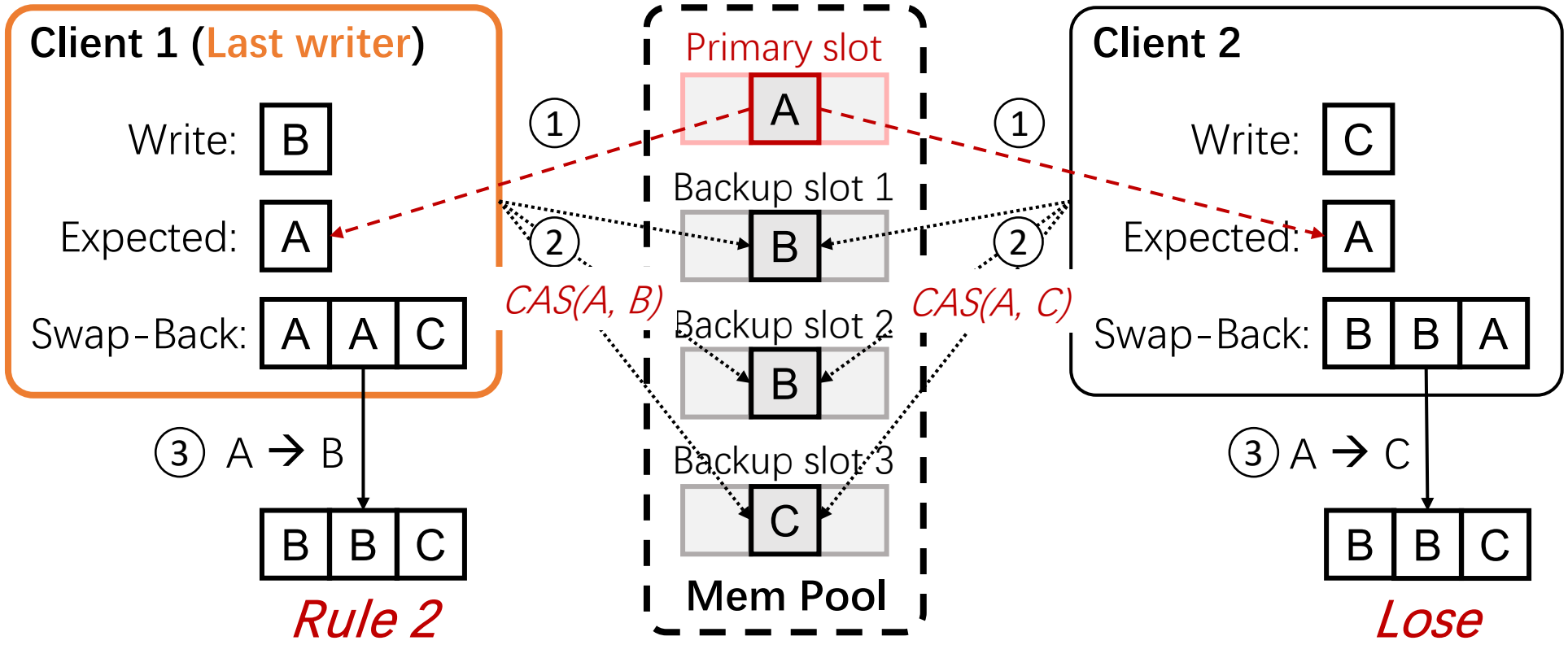
- *Successfully modified all backup slots*
- *Or, modified a majority of backup slots*
- *Or, writes the smallest value when there is no majority*



Client-Centric Index Replication

A client is a last writer if:

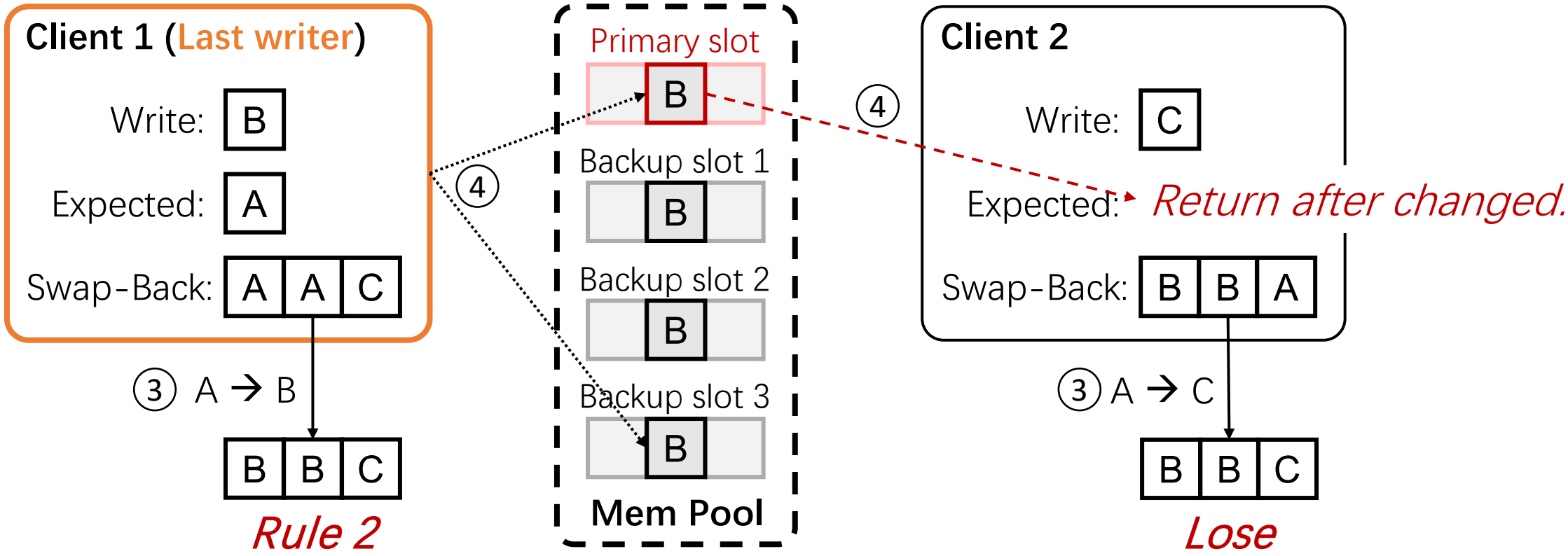
- *Successfully modified all backup slots*
- *Or, modified a majority of backup slots*
- *Or, writes the smallest value when there is no majority*



Client-Centric Index Replication

A client is a last writer if:

- *Successfully modified all backup slots*
- *Or, modified a majority of backup slots*
- *Or, writes the smallest value when there is no majority*



Client-Centric Index Replication

A client is a last writer if:

- *Successfully modified all backup slots*
- *Or, modified a majority of backup slots*
- *Or, writes the smallest value when there is no majority*

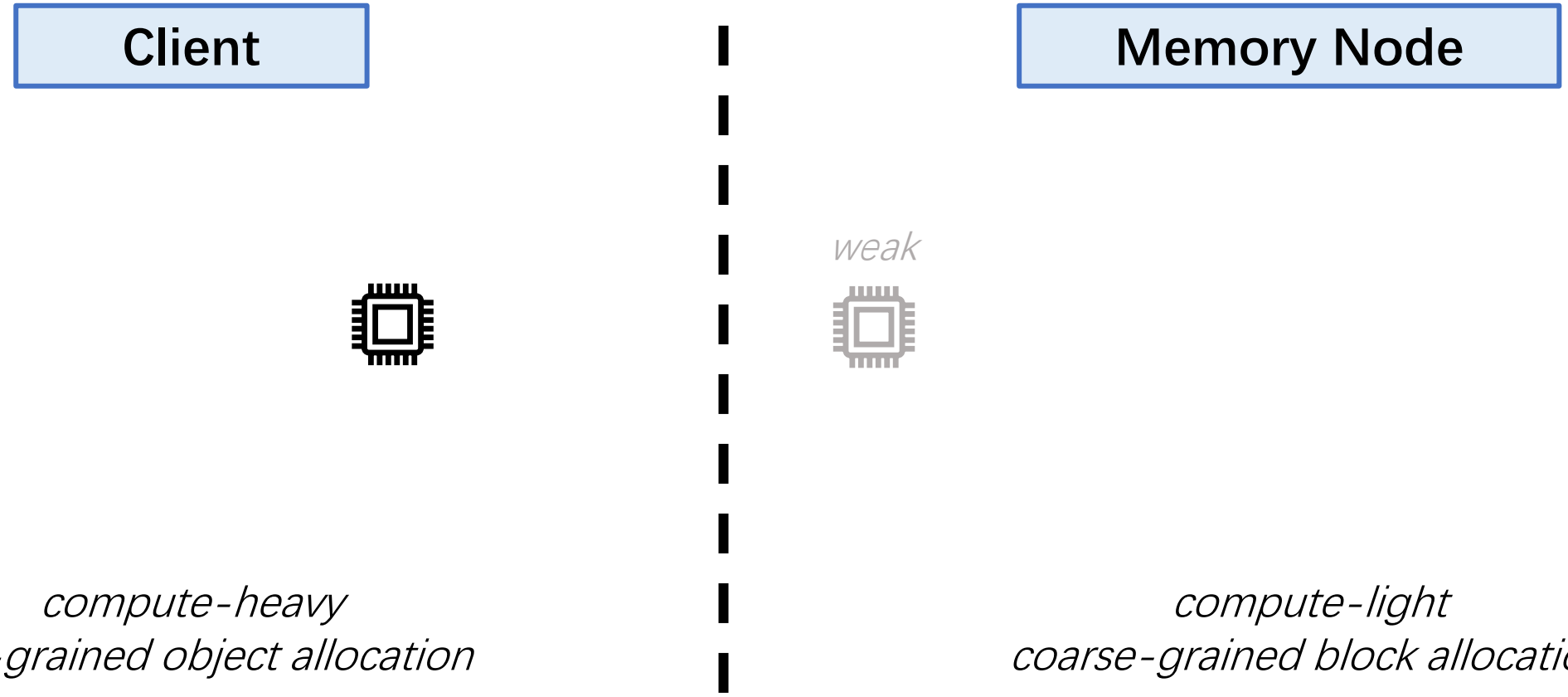
Bounded latency for index read and write:

- *1 RTT for read operations*
- *3-5 RTTs for write operations and 3 RTTs for most cases*

Two-Level Memory Management

Challenge: Where to execute memory management logic?

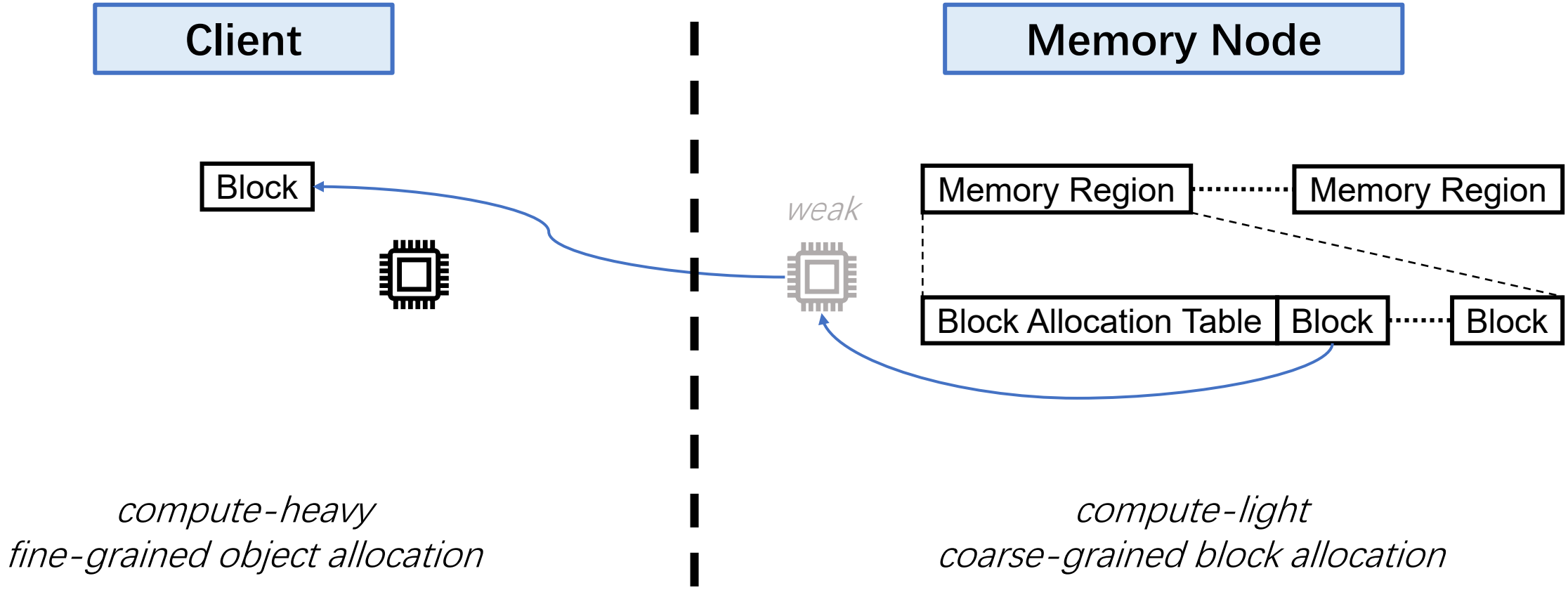
- ***Key idea:*** Separate into compute-light and compute-heavy tasks



Two-Level Memory Management

Challenge: Where to execute memory management logic?

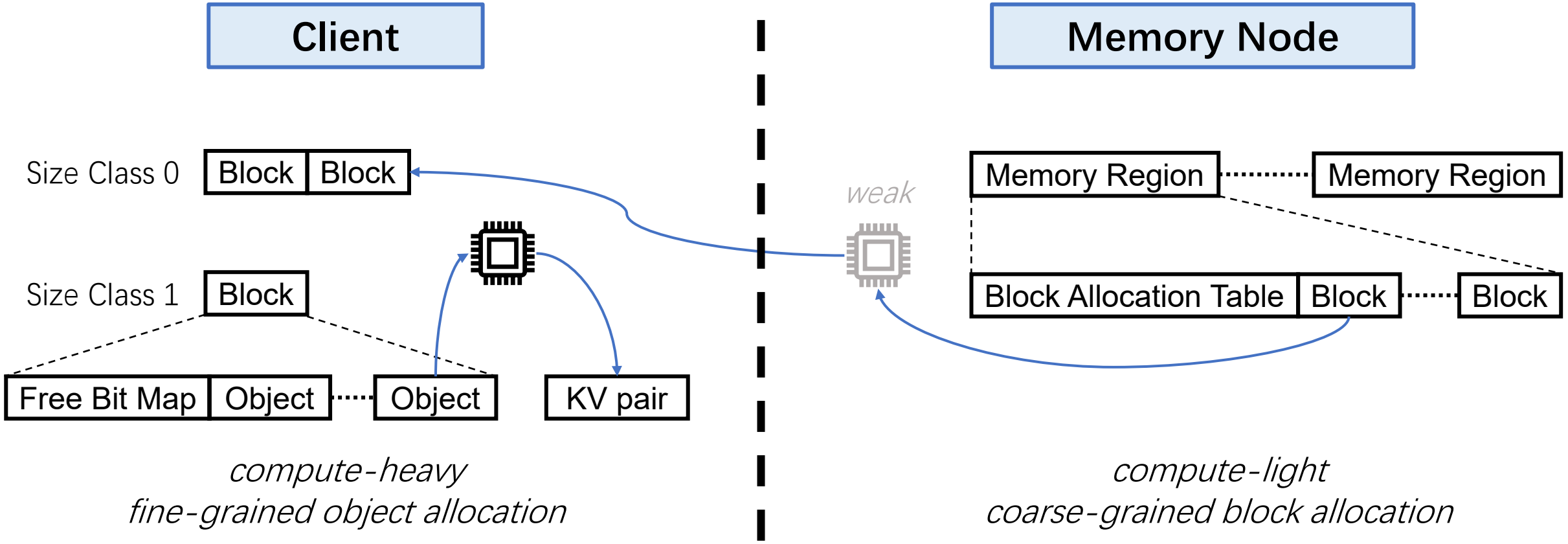
- ***Key idea:*** Separate into compute-light and compute-heavy tasks



Two-Level Memory Management

Challenge: Where to execute memory management logic?

- ***Key idea:*** Separate into compute-light and compute-heavy tasks



Embedded Operation Log

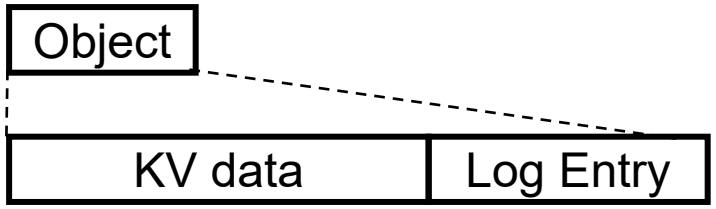
Challenge: An additional RTT to create a log entry before operation starts

- ***Key idea:*** Embed operation logs into KV pairs to reduce the additional RTT

Embedded Operation Log

Challenge: An additional RTT to create a log entry before operation starts

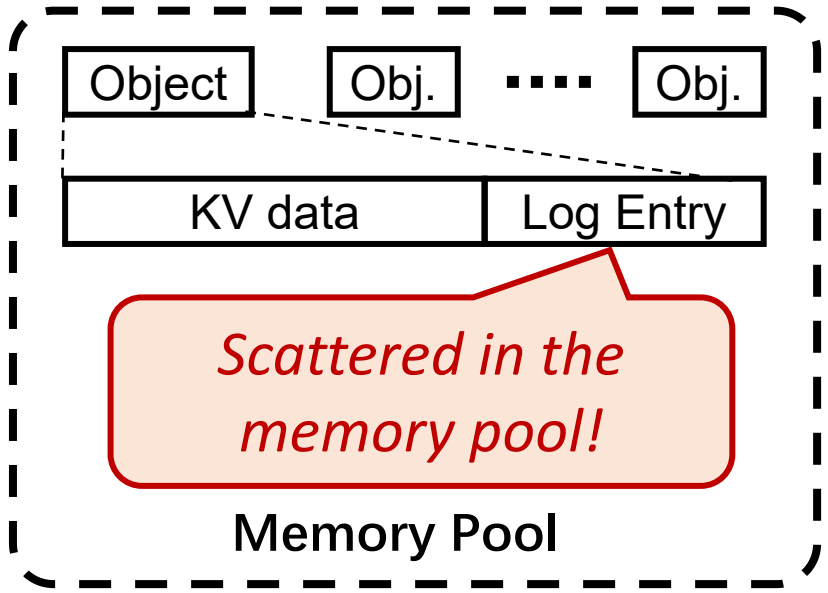
- ***Key idea:*** Embed operation logs into KV pairs to reduce the additional RTT



Embedded Operation Log

Challenge: An additional RTT to create a log entry before operation starts

- ***Key idea:*** Embed operation logs into KV pairs to reduce the additional RTT

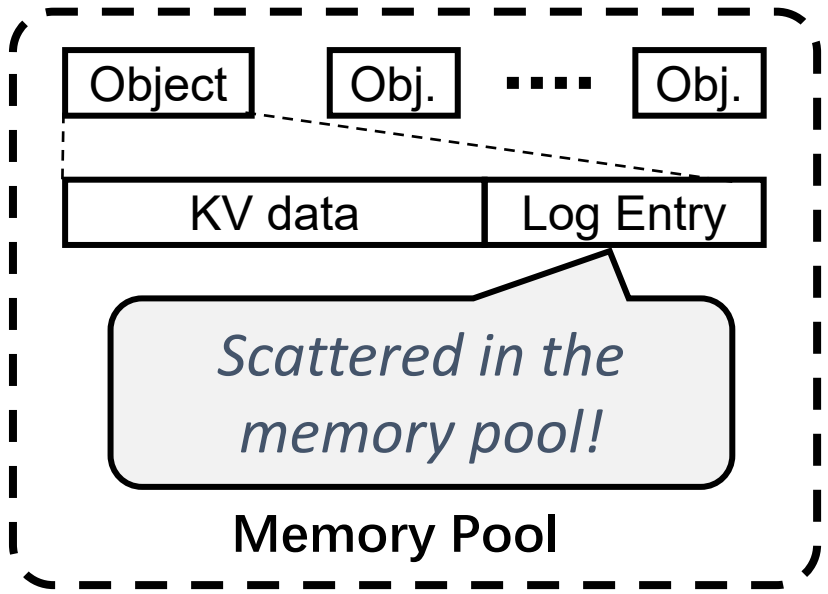


How to construct operation sequence?

Embedded Operation Log

Challenge: An additional RTT to create a log entry before operation starts

- ***Key idea:*** Embed operation logs into KV pairs to reduce the additional RTT



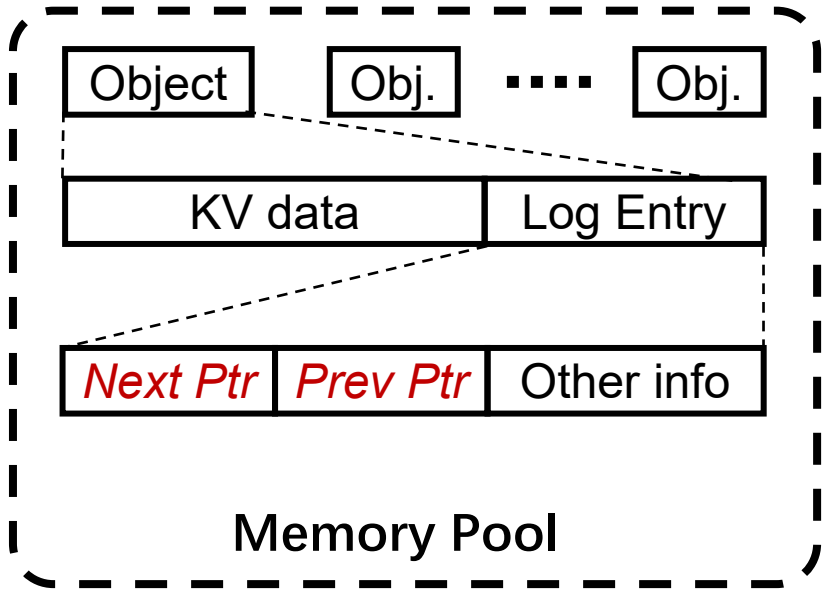
Use memory allocation order!

How to construct operation sequence?

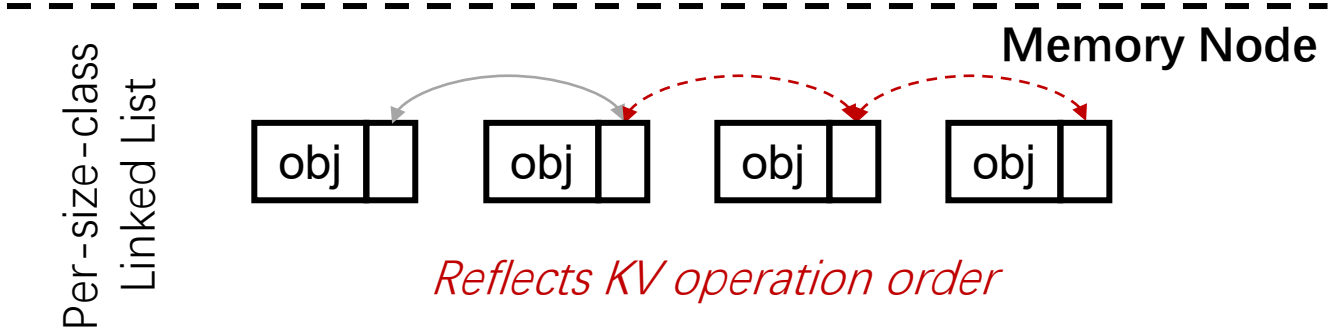
Embedded Operation Log

Challenge: An additional RTT to create a log entry before operation starts

- ***Key idea:*** Embed operation logs into KV pairs to reduce the additional RTT



Use memory allocation order!

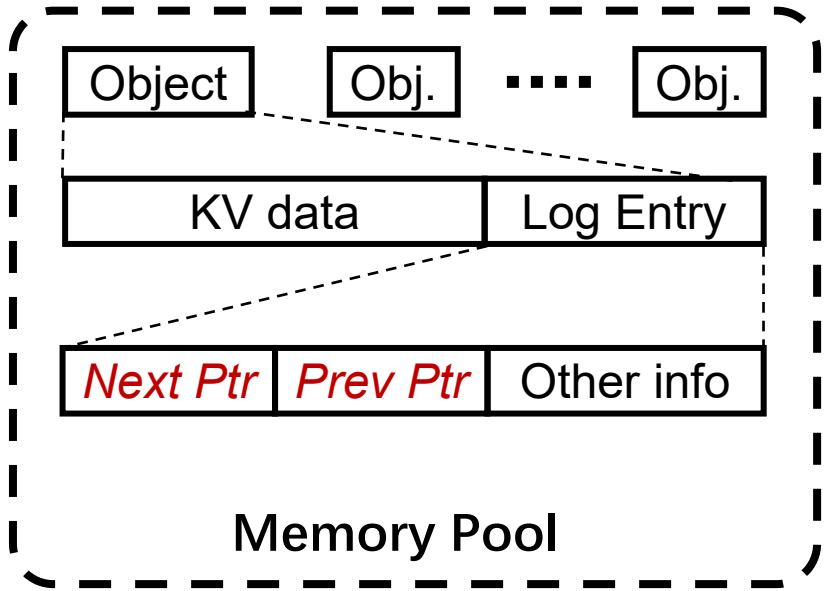


How to construct operation sequence?

Embedded Operation Log

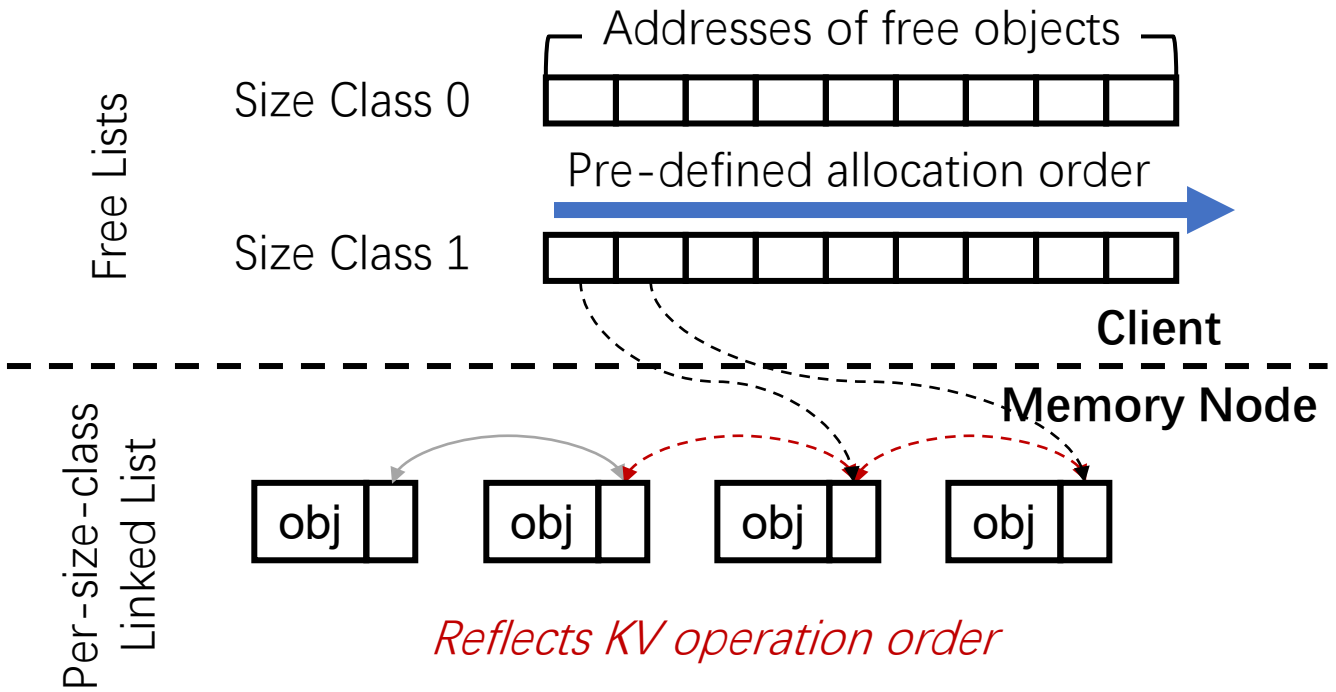
Challenge: An additional RTT to create a log entry before operation starts

- ***Key idea:*** Embed operation logs into KV pairs to reduce the additional RTT



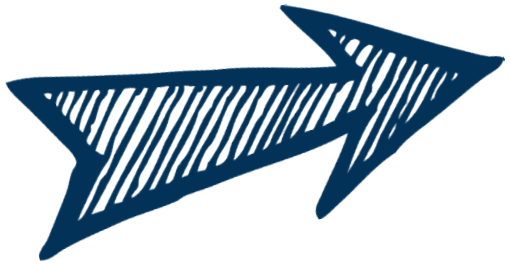
How to construct operation sequence?

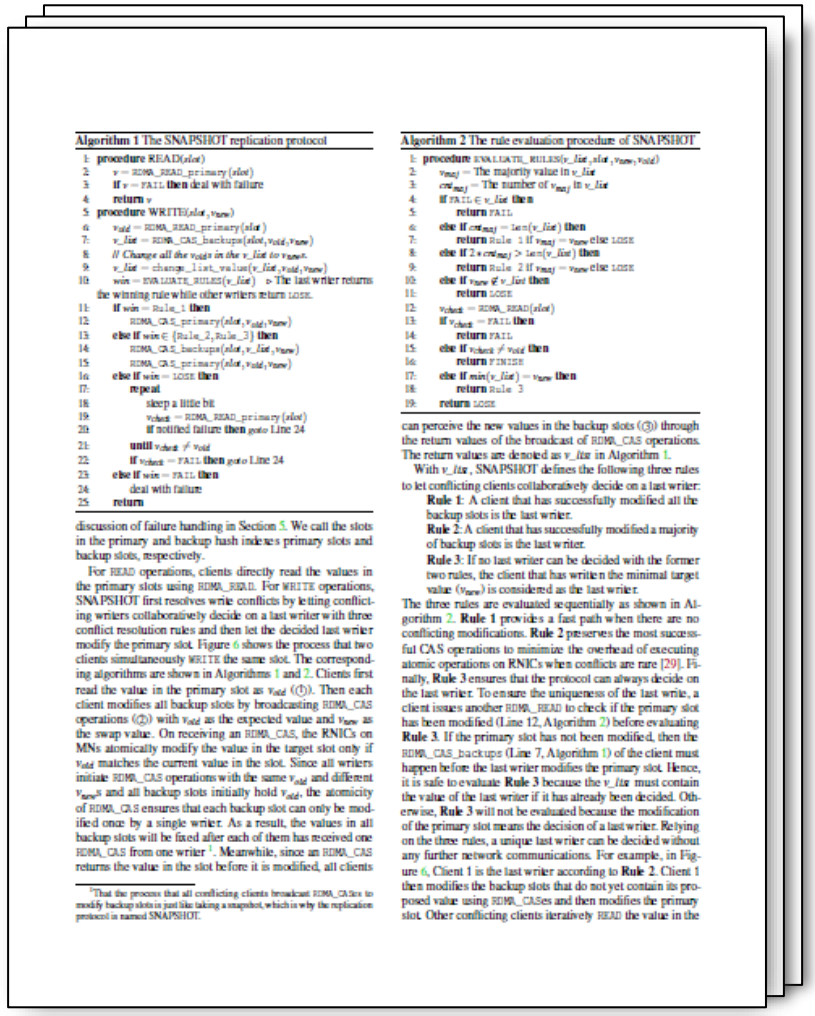
Use memory allocation order!



More Details

- Correctness of SNAPSHOT
- Adaptive index cache
- Recovery
- …





Algorithm 1 The SNAPSHOT replication protocol

```
1:  procedure READ(slot)
2:      v ← RDMA_READ_primary(slot)
3:      if v = FAIL then deal with failure
4:      return v
5:  procedure WRITE(slot, v_new)
6:      v_old ← RDMA_READ_primary(slot)
7:      v_list ← RDMA_CAS_backups(slot, v_old, v_new)
8:      // Change all the v_olds in the v_list to v_new.
9:      v_list ← change_list_value(v_list, v_old, v_new)
10:     win ← EVALUATE_RULES(v_list)   ▷ The last writer returns
    the winning rule while other writers return LOSE.
11:     if win = Rule_1 then
12:         RDMA_CAS_primary(slot, v_old, v_new)
13:     else if win ∈ {Rule_2, Rule_3} then
14:         RDMA_CAS_backups(slot, v_list, v_new)
15:         RDMA_CAS_primary(slot, v_old, v_new)
16:     else if win = LOSE then
17:         repeat
18:             sleep a little bit
19:             v_check ← RDMA_READ_primary(slot)
20:             if notified failure then goto Line 24
21:         until v_check ≠ v_old
22:         if v_check = FAIL then goto Line 24
23:     else if win = FAIL then
24:         deal with failure
25:     return
```

discussion of failure handling in Section 5. We call the slots in the primary and backup hash indexes primary slots and backup slots, respectively.

For READ operations, clients directly read the values in the primary slots using RDMA_READ. For WRITE operations, SNAPSHOT first resolves write conflicts by letting conflicting writers collaboratively decide on a last writer with three conflict resolution rules and then let the decided last writer modify the primary slot. Figure 6 shows the process that two clients simultaneously WRITE the same slot. The corresponding algorithms are shown in Algorithms 1 and 2. Clients first read the value in the primary slot as v_{old} (①). Then each client modifies all backup slots by broadcasting RDMA_CAS operations (②) with v_{old} as the expected value and v_{new} as the swap value. On receiving an RDMA_CAS, the RNICs on MNs atomically modify the value in the target slot only if v_{old} matches the current value in the slot. Since all writers initiate RDMA_CAS operations with the same v_{old} and different v_{new}s and all backup slots initially hold v_{old}, the atomicity of RDMA_CAS ensures that each backup slot can only be modified once by a single writer. As a result, the values in all backup slots will be fixed after each of them has received one RDMA_CAS from one writer [1]. Meanwhile, since an RDMA_CAS returns the value in the slot before it is modified, all clients

[1]That the process that all conflicting clients broadcast RDMA_CASes to modify backup slots is just like taking a snapshot, which is why the replication protocol is named SNAPSHOT.

Algorithm 2 The rule evaluation procedure of SNAPSHOT

```
1:  procedure EVALUATE_RULES(v_list, slot, v_new, v_old)
2:      v_maj ← The majority value in v_list
3:      cnt_maj ← The number of v_maj in v_list
4:      if FAIL ∈ v_list then
5:          return FAIL
6:      else if cnt_maj = len(v_list) then
7:          return Rule 1 if v_maj = v_new else LOSE
8:      else if 2 * cnt_maj > len(v_list) then
9:          return Rule 2 if v_maj = v_new else LOSE
10:     else if v_new ∉ v_list then
11:         return LOSE
12:     v_check ← RDMA_READ(slot)
13:     if v_check = FAIL then
14:         return FAIL
15:     else if v_check ≠ v_old then
16:         return FINISH
17:     else if min(v_list) = v_new then
18:         return Rule 3
19:     return LOSE
```

can perceive the new values in the backup slots (③) through the return values of the broadcast of RDMA_CAS operations. The return values are denoted as v_list in Algorithm 1.

With v_list, SNAPSHOT defines the following three rules to let conflicting clients collaboratively decide on a last writer:

Rule 1: A client that has successfully modified all the backup slots is the last writer.

Rule 2: A client that has successfully modified a majority of backup slots is the last writer.

Rule 3: If no last writer can be decided with the former two rules, the client that has written the minimal target value (v_{new}) is considered as the last writer.

The three rules are evaluated sequentially as shown in Algorithm 2. **Rule 1** provides a fast path when there are no conflicting modifications. **Rule 2** preserves the most successful CAS operations to minimize the overhead of executing atomic operations on RNICs when conflicts are rare [29]. Finally, **Rule 3** ensures that the protocol can always decide on the last writer. To ensure the uniqueness of the last write, a client issues another RDMA_READ to check if the primary slot has been modified (Line 12, Algorithm 2) before evaluating **Rule 3**. If the primary slot has not been modified, then the RDMA_CAS_backups (Line 7, Algorithm 1) of the client must happen before the last writer modifies the primary slot. Hence, it is safe to evaluate **Rule 3** because the v_list must contain the value of the last writer if it has already been decided. Otherwise, **Rule 3** will not be evaluated because the modification of the primary slot means the decision of a last writer. Relying on the three rules, a unique last writer can be decided without any further network communications. For example, in Figure 6, Client 1 is the last writer according to **Rule 2**. Client 1 then modifies the backup slots that do not yet contain its proposed value using RDMA_CASes and then modifies the primary slot. Other conflicting clients iteratively READ the value in the

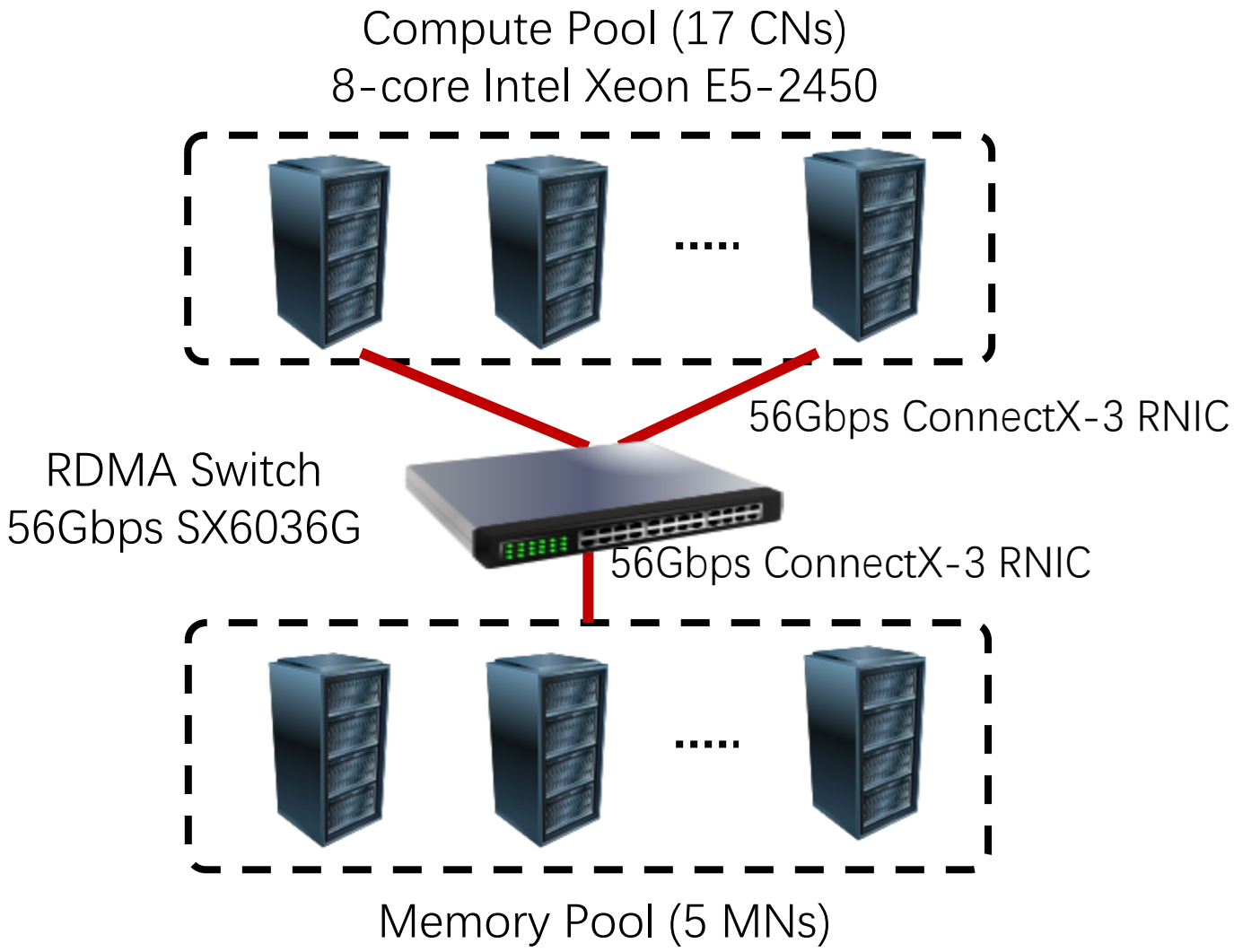
Evaluation

Benchmarks:

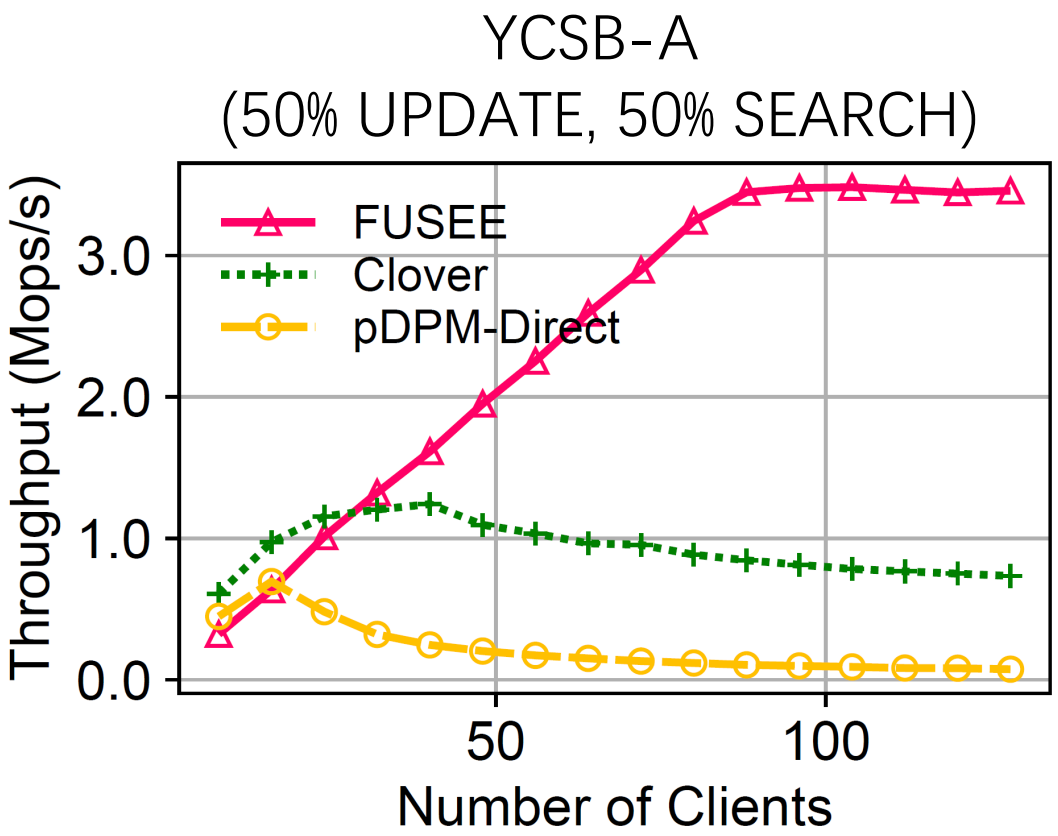
- YCSB hybrid benchmarks
- Microbenchmark

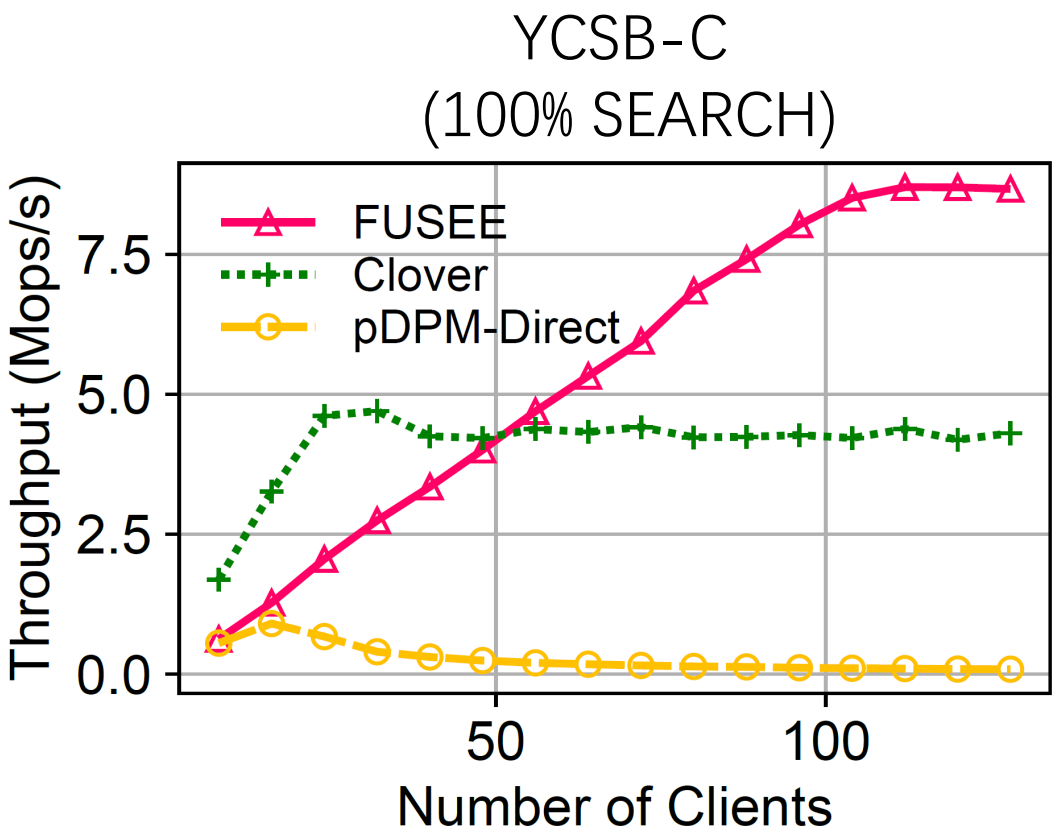
Comparisons:

- Clover [ATC 20]
- pDPM-Direct [ATC 20]
- FUSEE-NC (no cache)
- FUSEE-CR (chain replication)



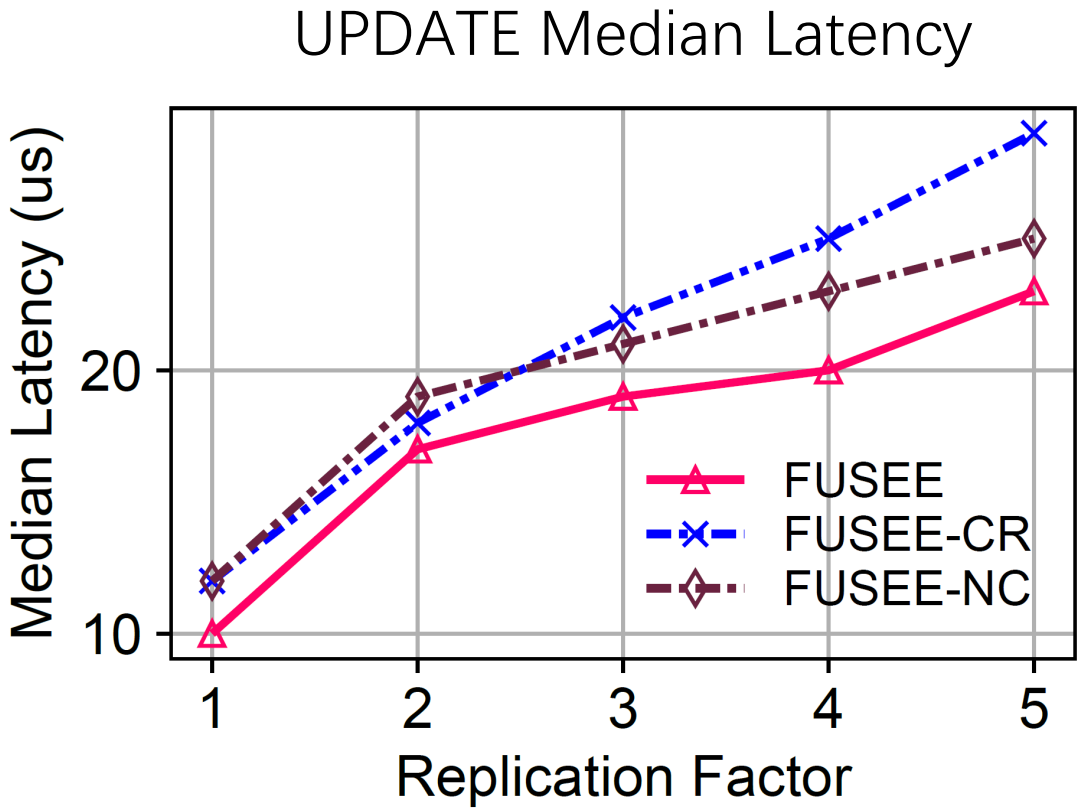
YCSB Hybrid Benchmarks

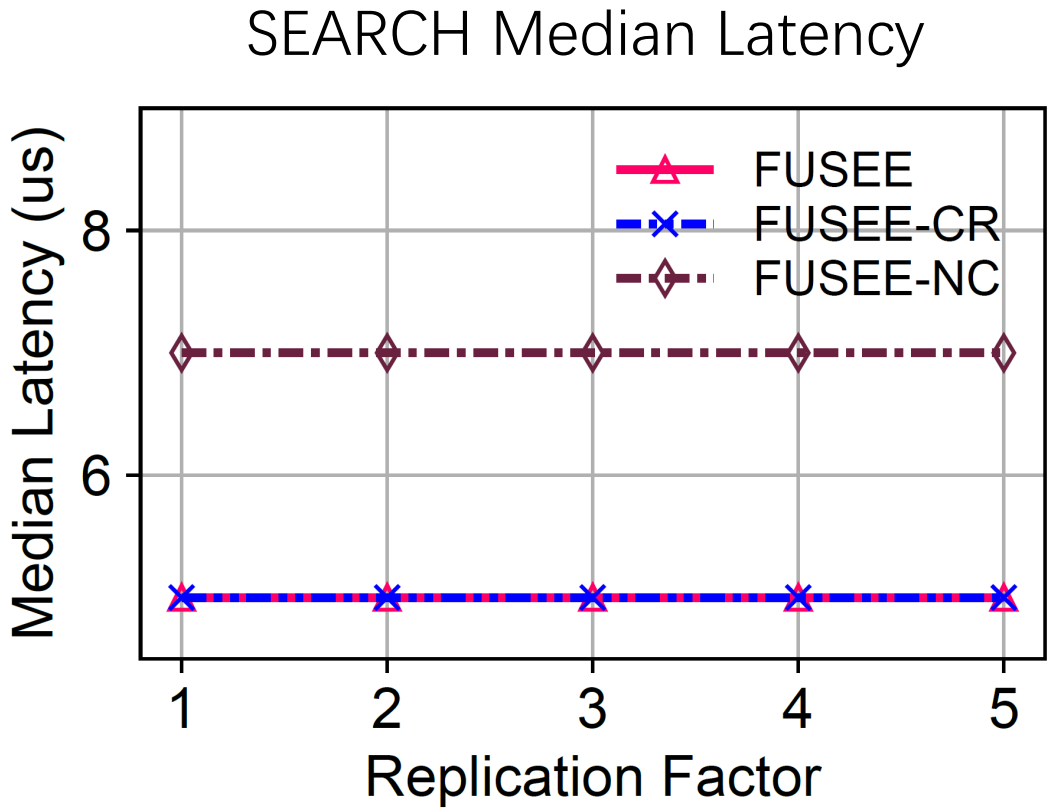




Reaches up to 4.9x and 117x higher throughput than Clover and pDPM-Direct

Replication Latency





SNAPSHOT replication has bounded number of RTTs under various replication factors

Conclusion

- Existing memory-disaggregated KV stores still cannot fully exploit the resource efficiency benefits of disaggregated memory
  - Semi-disaggregated architecture still relies on monolithic servers

- We propose FUSEE, a fully memory-disaggregated KV store
  - Client-centric index replication
  - Two-level memory management
  - Embedded operation log scheme

- FUSEE outperforms the state-of-the-art memory-disaggregated KV store by up to 4.9x in YCSB hybrid workloads with less resource consumption

Thank you! Q&A