# Tailwind: Fast and Atomic RDMA-based Replication

Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, Toni Cortes

# In-Memory Key-Value Stores

- General purpose in-memory key-value stores are widely used nowadays

# In-Memory Key-Value Stores

- General purpose in-memory key-value stores are widely used nowadays
- Recent systems can process millions of requests/second/machine: E.g. RAMCloud, FaRM, MICA, …

cassandra

redis

AEROSPIKE

# In-Memory Key-Value Stores

- General purpose in-memory key-value stores are widely used nowadays

- Recent systems can process millions of requests/second/machine: E.g. RAMCloud, FaRM, MICA, …

- Key enablers: eliminating **network** overheads (e.g., kernel bypass) and leveraging **multicore** architectures

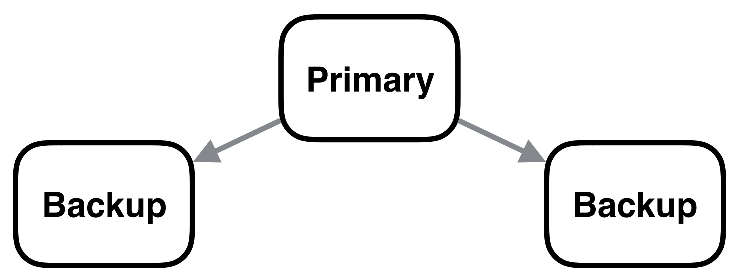
# In-Memory Key-Value Stores

- CPU is becoming a bottleneck in modern kv-stores
  ->Random memory access, key-value GET/PUT processing

# In-Memory Key-Value Stores

- CPU is becoming a bottleneck in modern kv-stores
  ->Random memory access, key-value GET/PUT processing

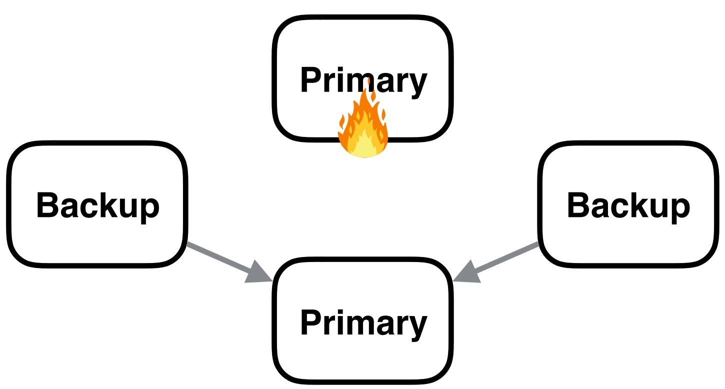
- Persistent in-memory kv-stores have to replicate data to survive failures



# In-Memory Key-Value Stores

- CPU is becoming a bottleneck in modern kv-stores
  ->Random memory access, key-value GET/PUT processing

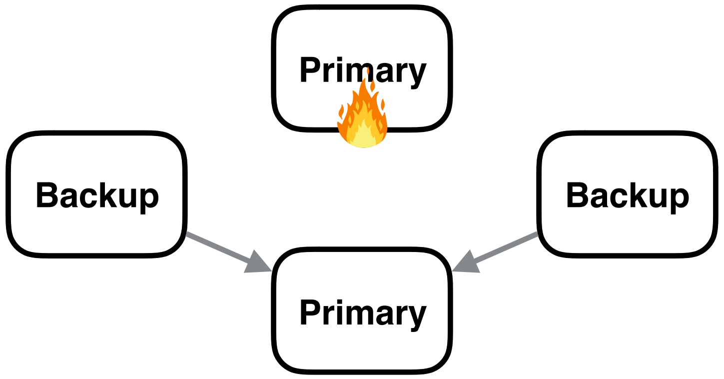
- Persistent in-memory kv-stores have to replicate data to survive failures



# In-Memory Key-Value Stores

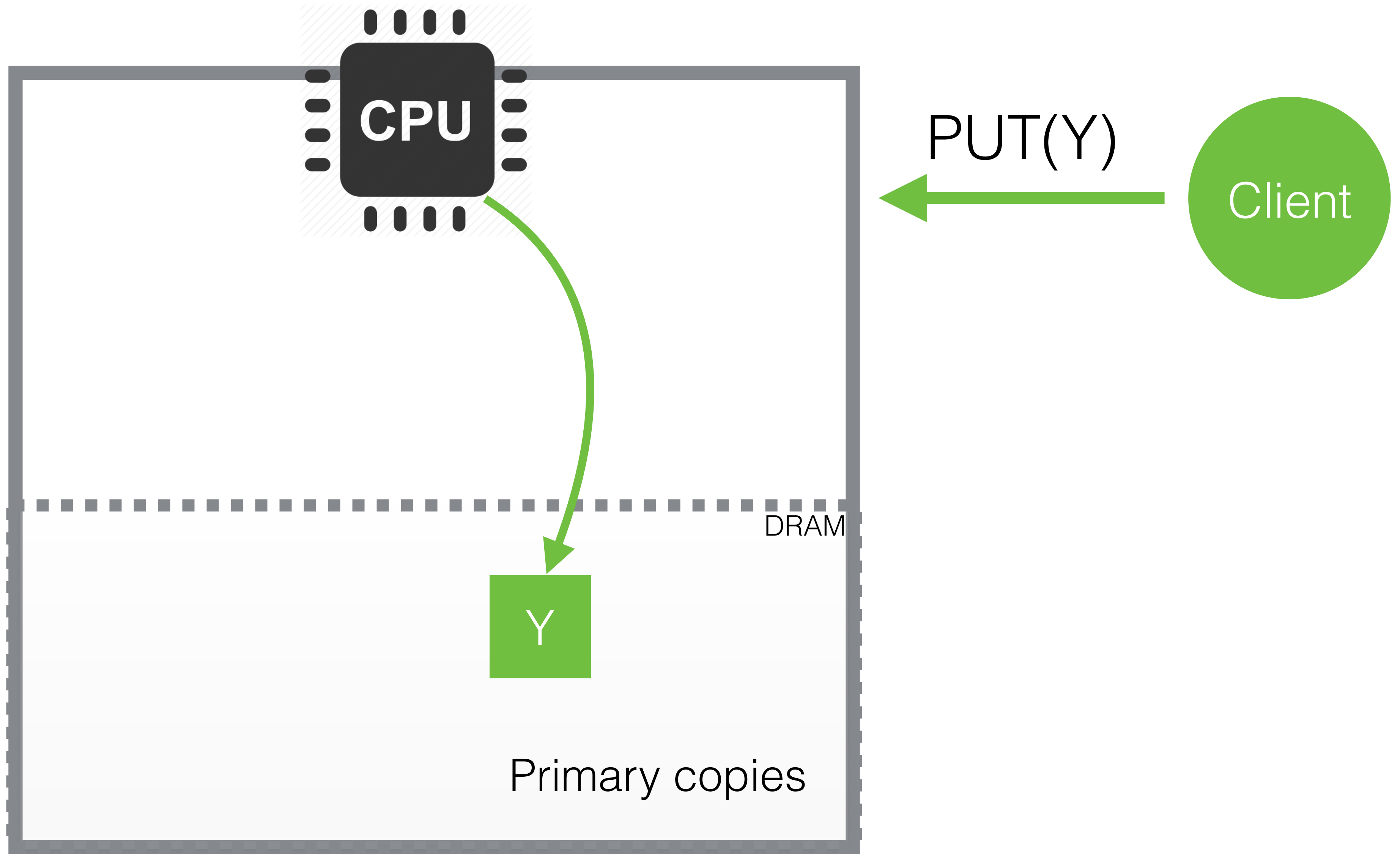
- CPU is becoming a bottleneck in modern kv-stores
  ->Random memory access, key-value GET/PUT processing

- Persistent in-memory kv-stores have to replicate data to survive failures

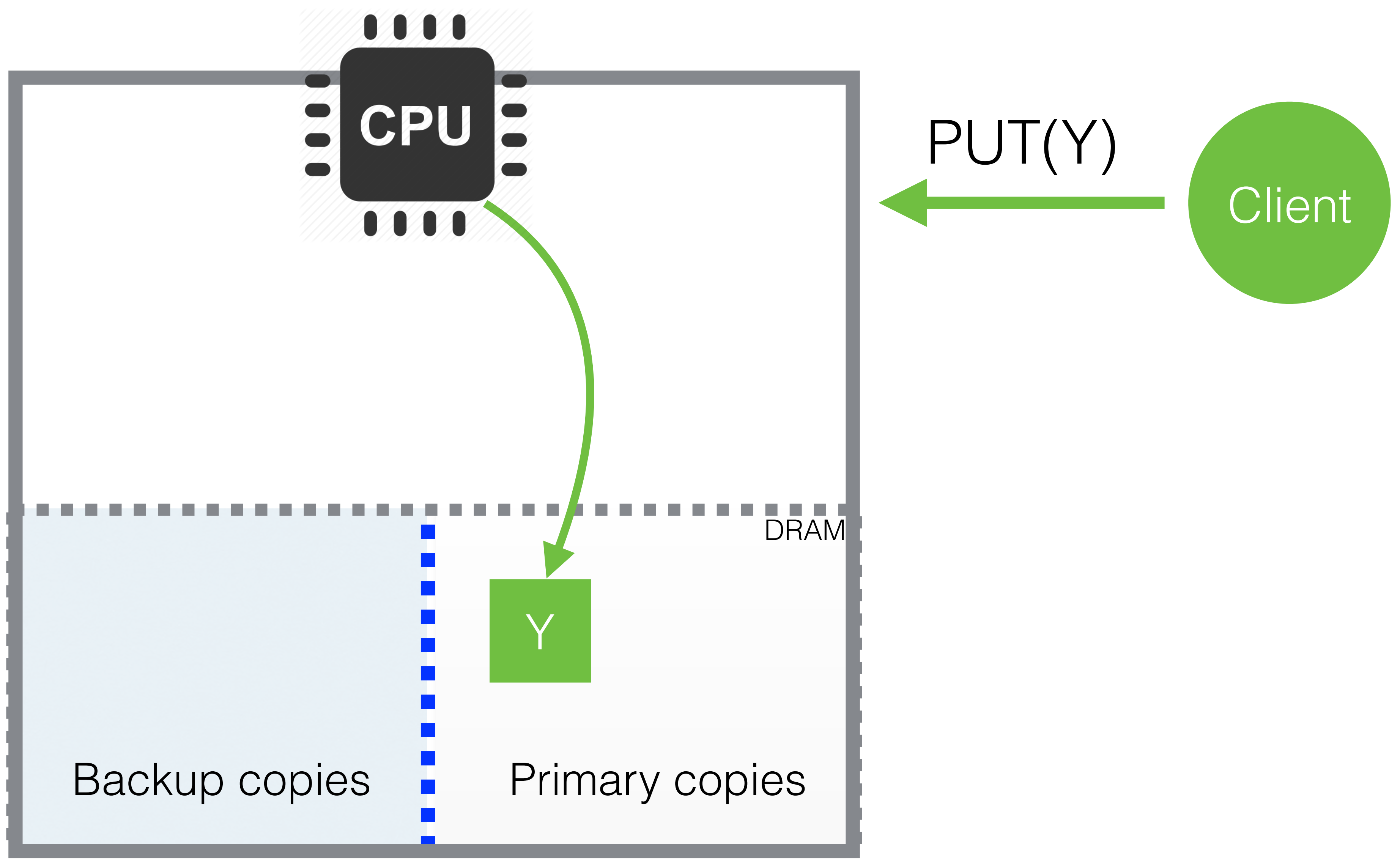



**->Replication contends with normal request processing**

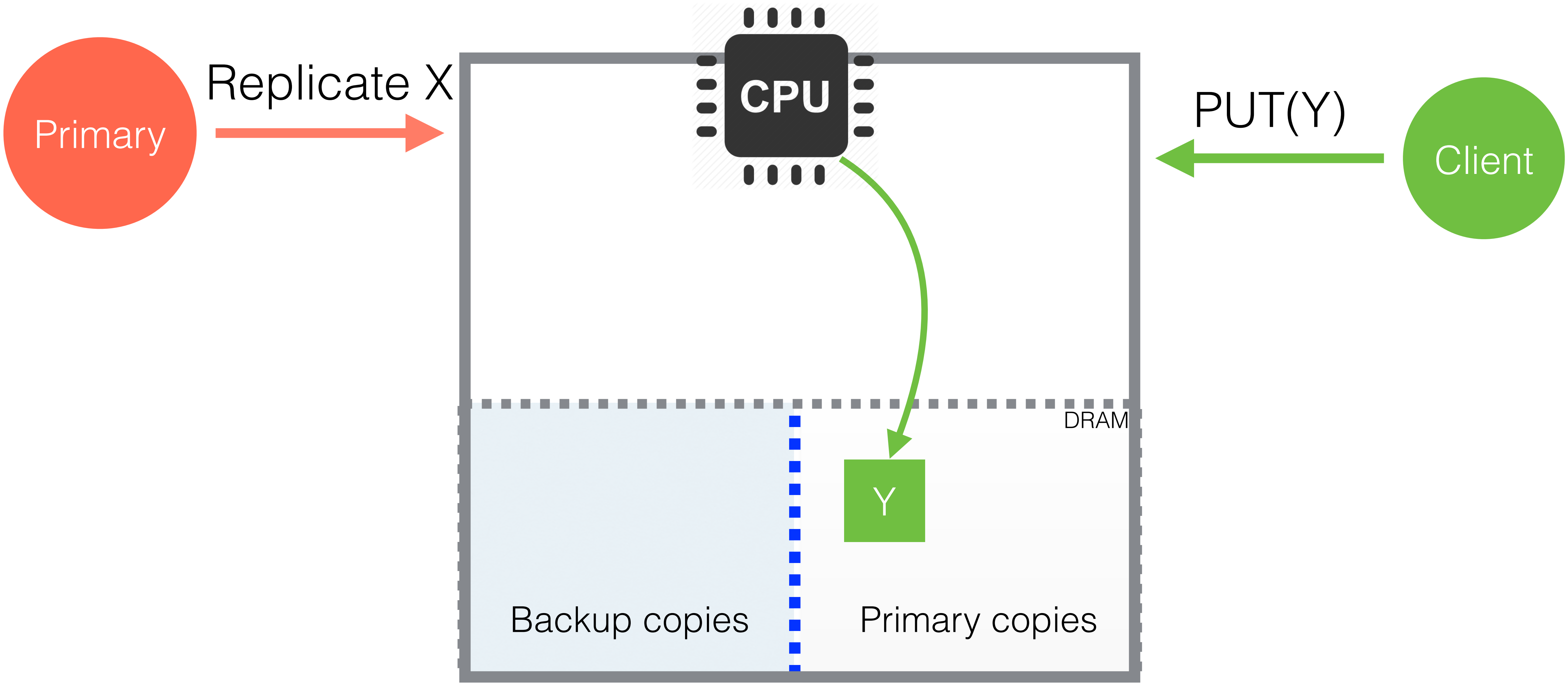
# In-Memory Key-Value Stores



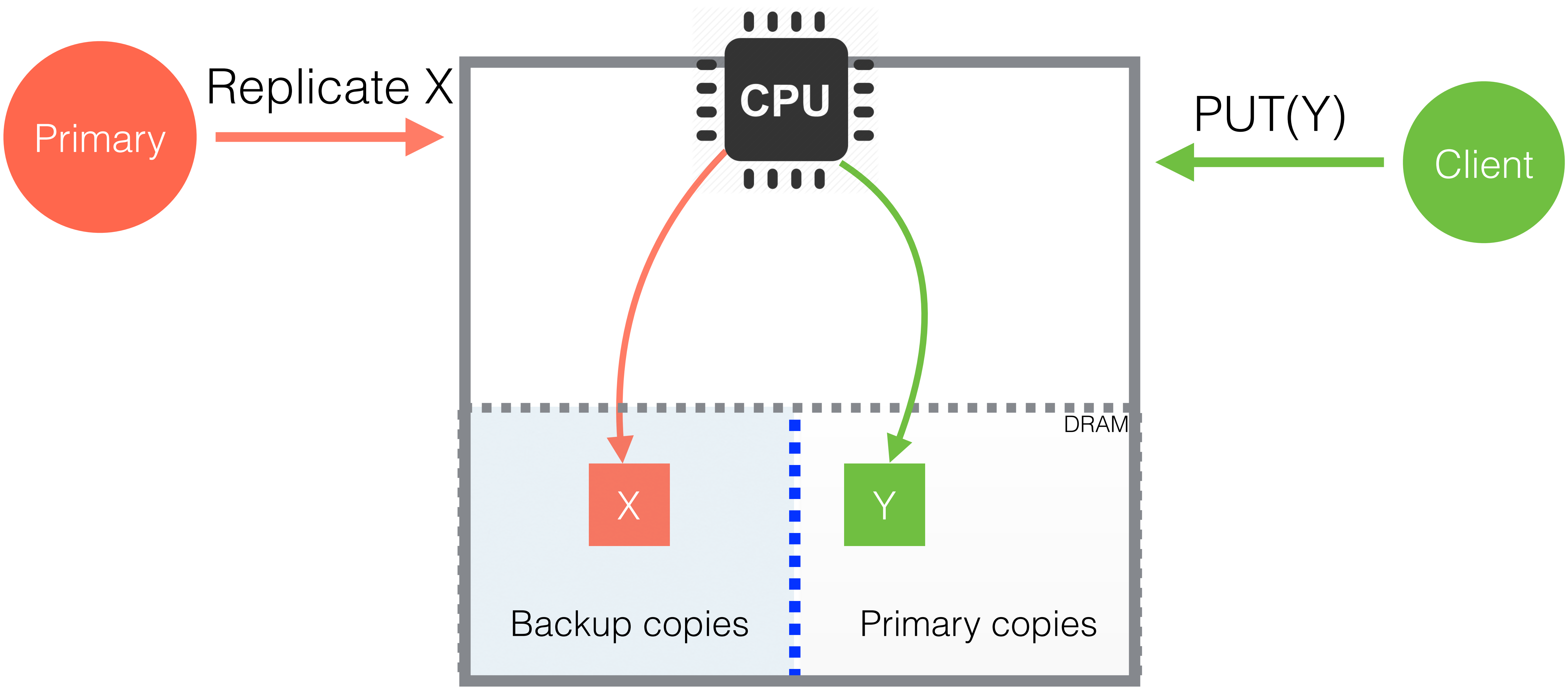
# In-Memory Key-Value Stores



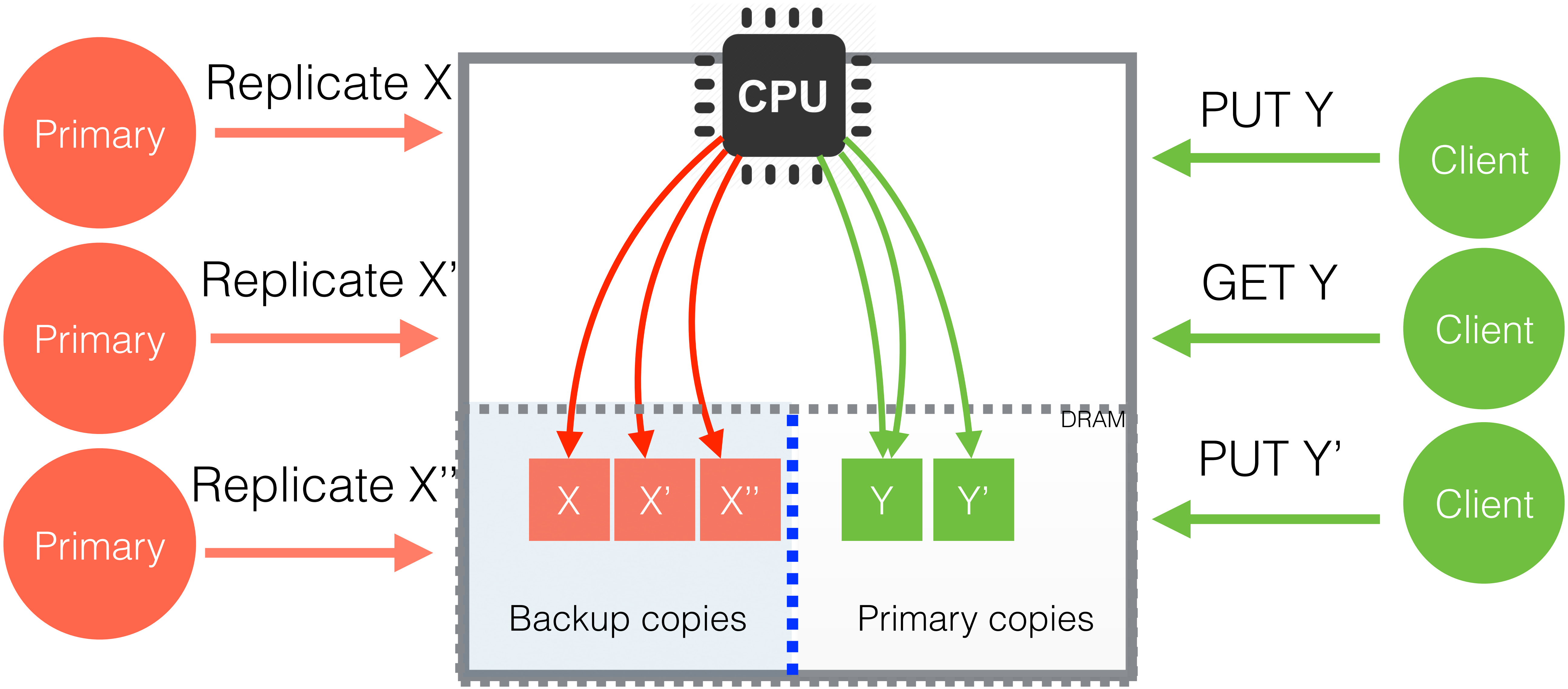
# In-Memory Key-Value Stores



# In-Memory Key-Value Stores



# In-Memory Key-Value Stores

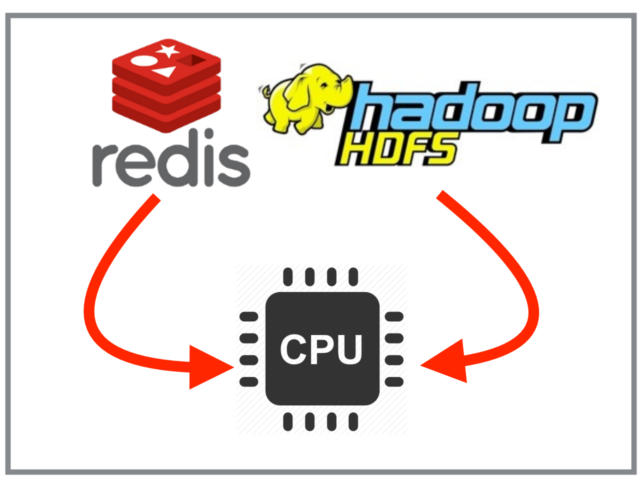


# Replication in In-Memory Key-Value Stores

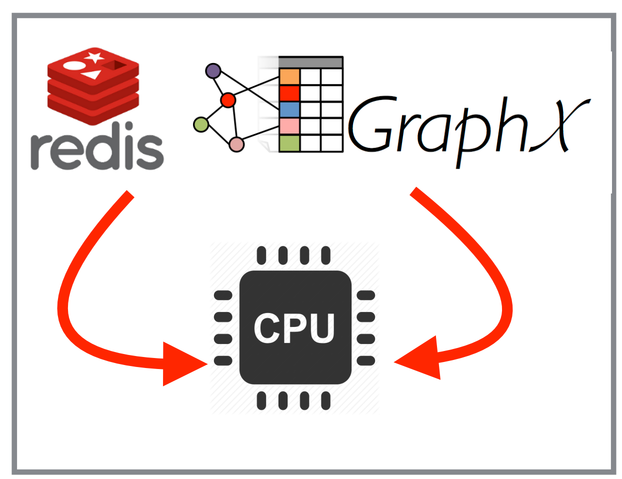
- Replication impedes modern kv-stores

# Replication in In-Memory Key-Value Stores

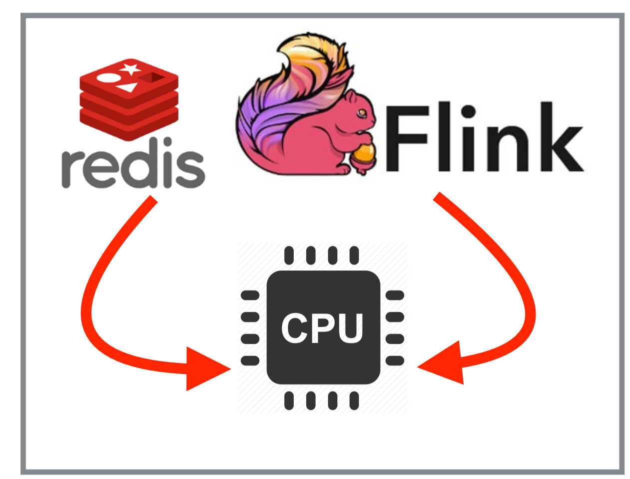
- Replication impedes modern kv-stores **and collocated applications**




**Block Storage**




**Graph Processing**




**Stream Processing**

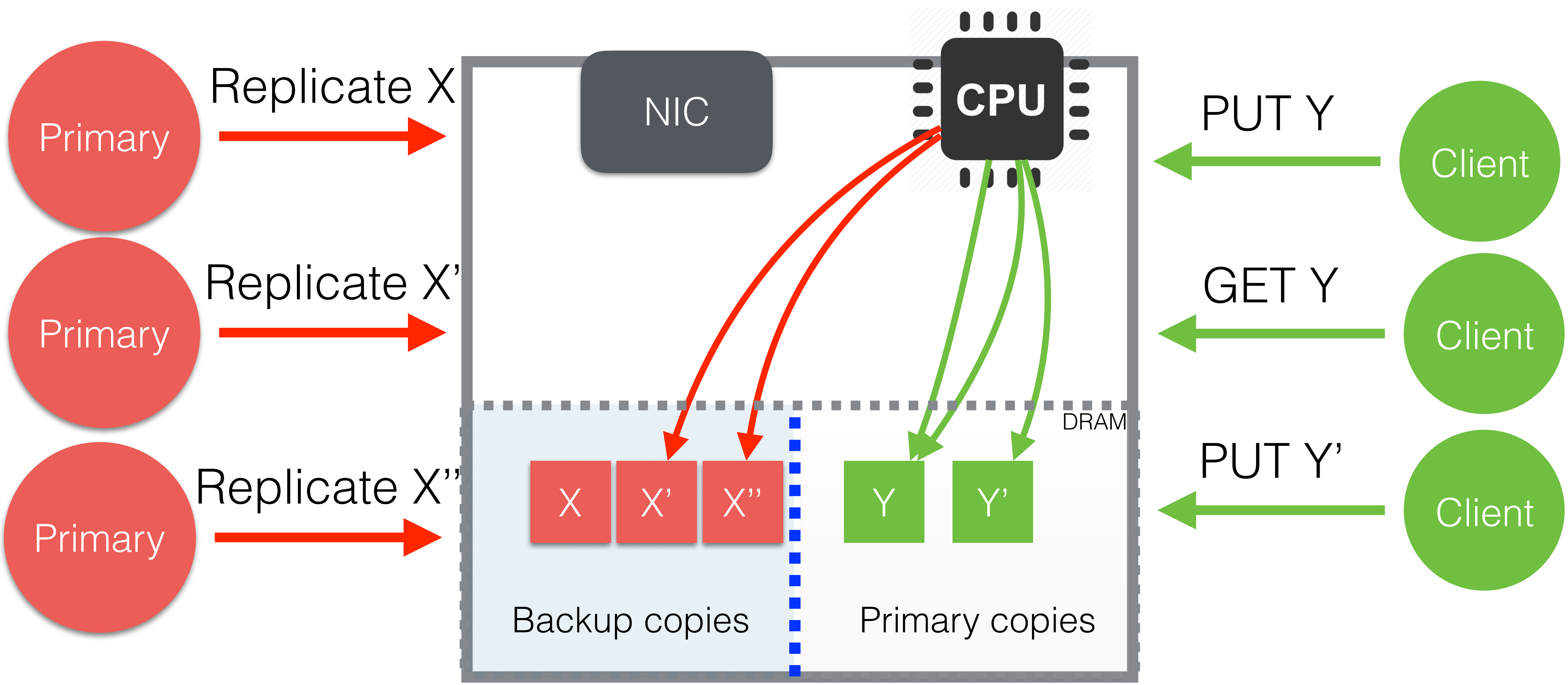
# Replication in In-Memory Key-Value Stores

- How to mitigate replication overhead?

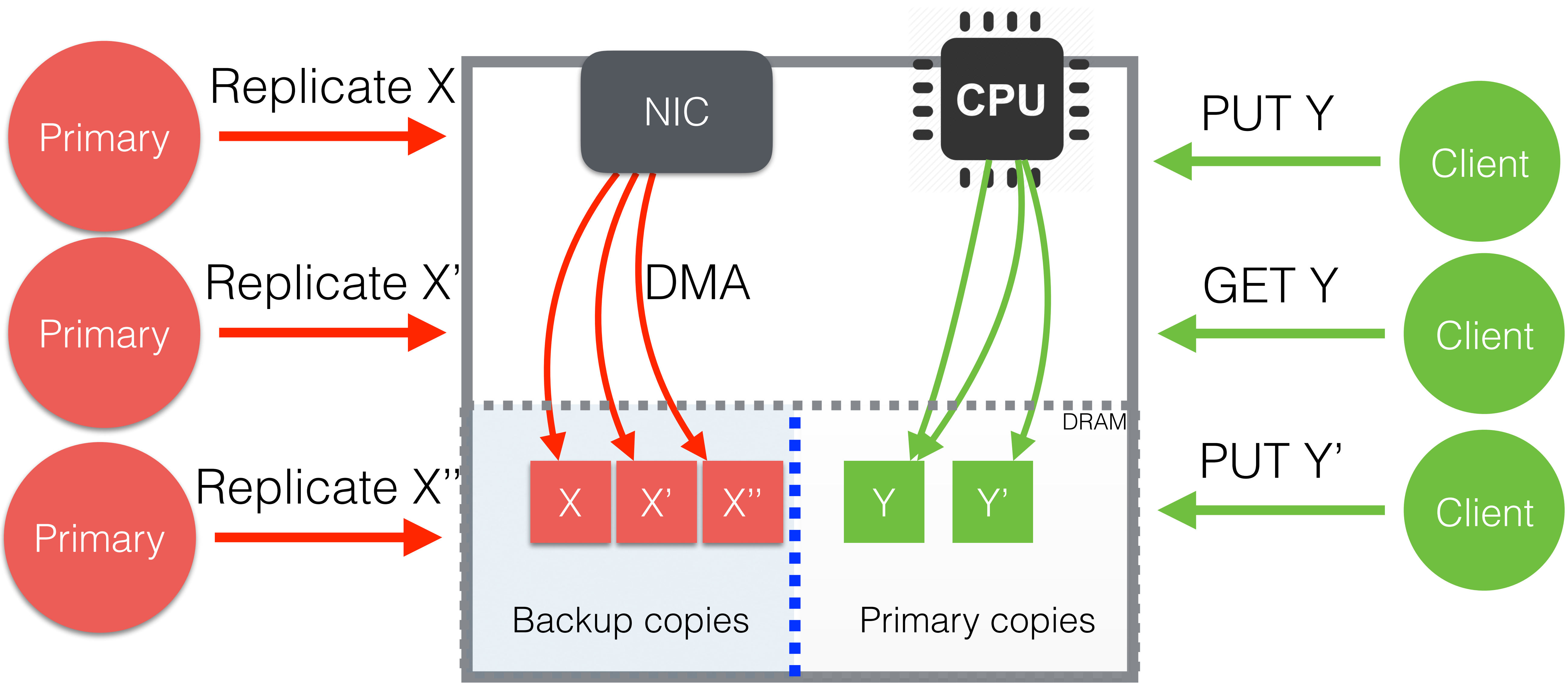
# Replication in In-Memory Key-Value Stores

- How to mitigate replication overhead?
- Techniques like Remote Direct Memory Access (RDMA) seem promising

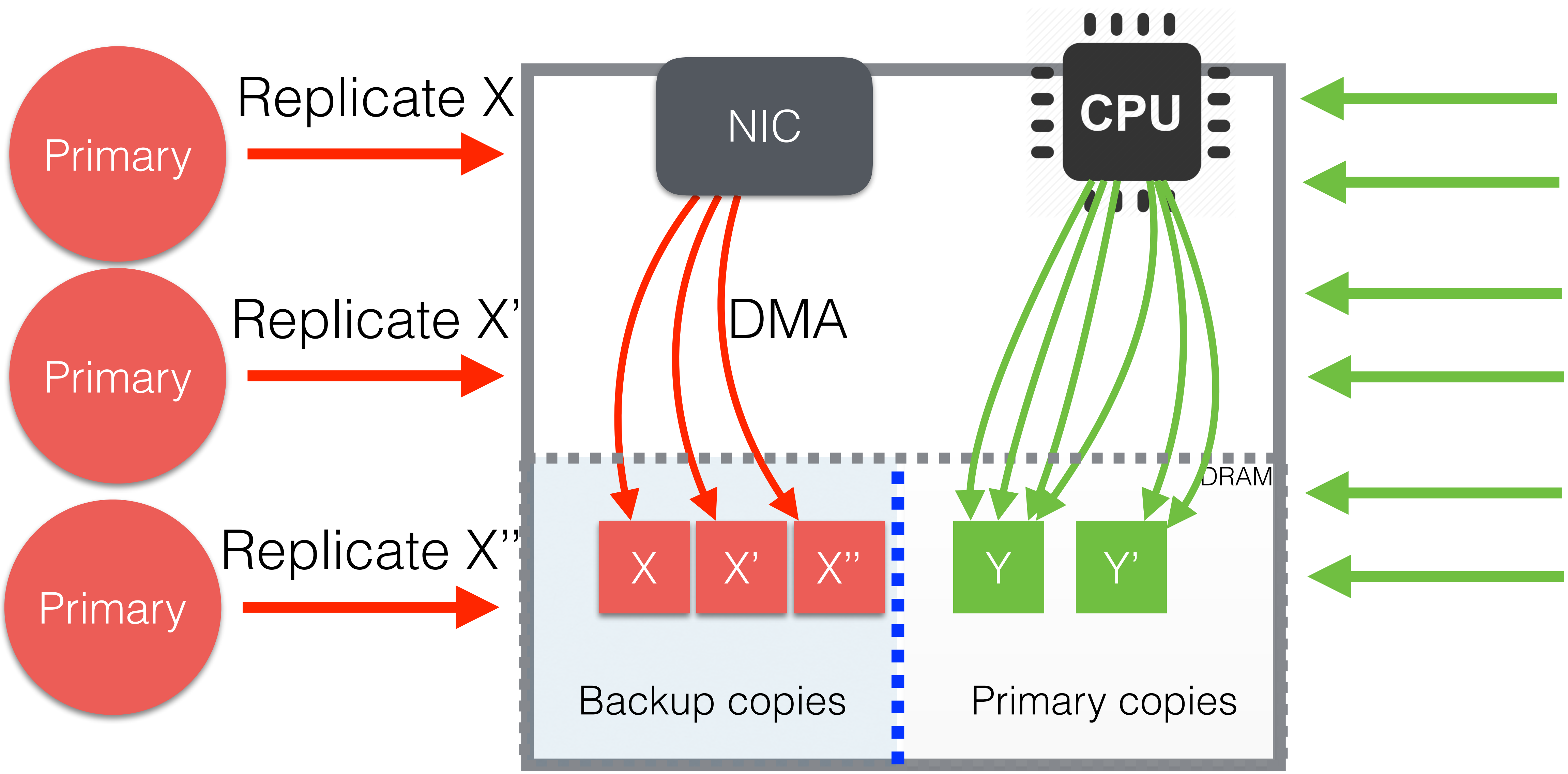
# Replication in In-Memory Key-Value Stores



# RDMA-Based Replication



# RDMA-Based Replication



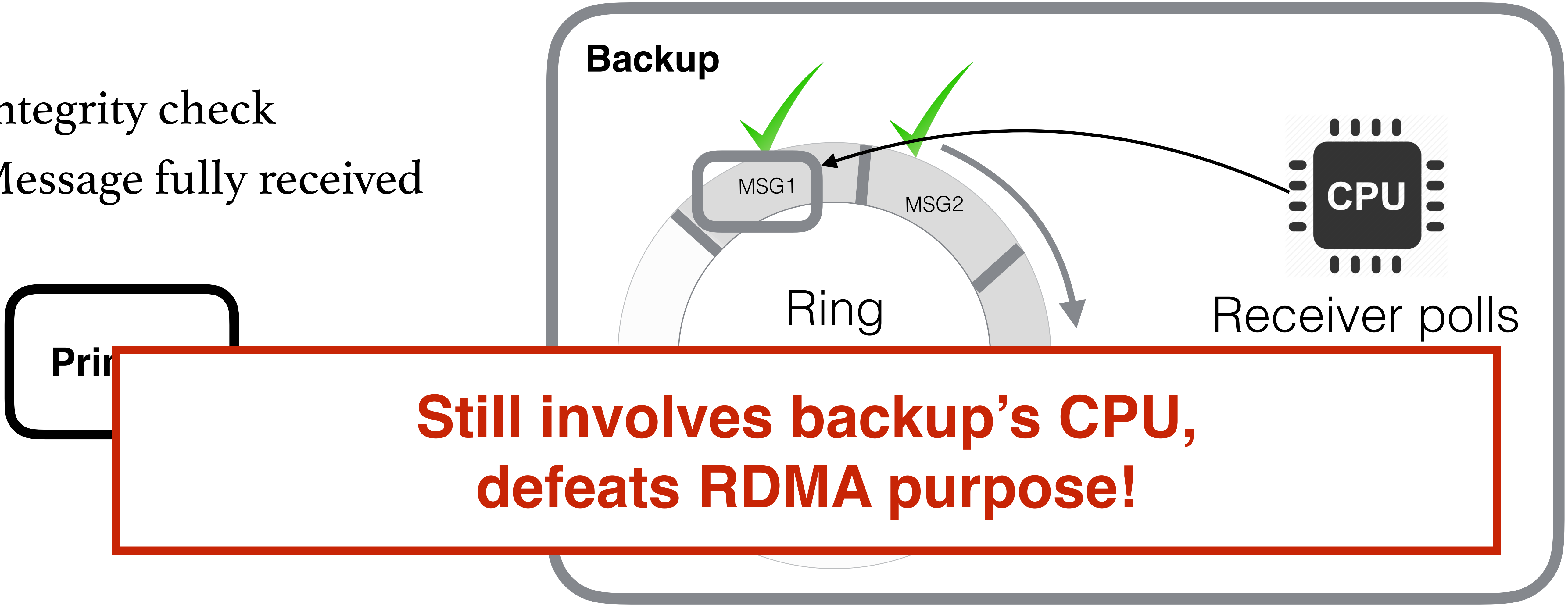
# Existing Protocols

- Many systems use one-sided RDMA for replication

  FaRM NSDI'14, DARE HPDC'15, HydraDB SC'16

Integrity check

Message fully received




**Still involves backup's CPU, defeats RDMA purpose!**

# Outline

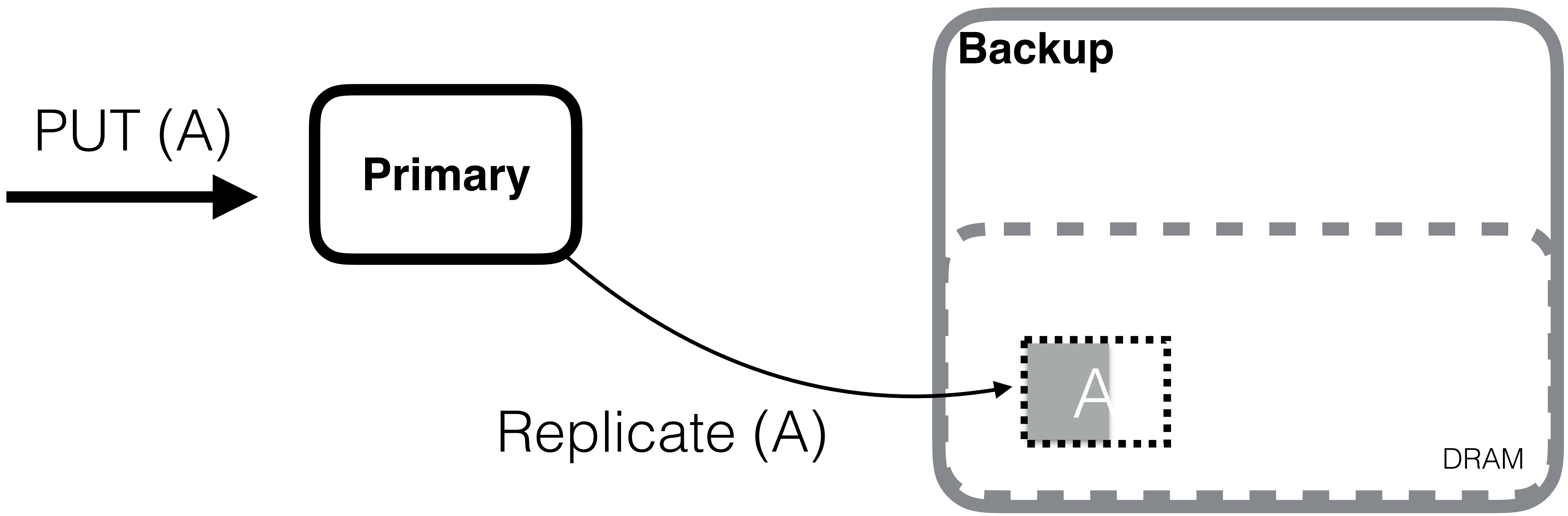
- Context
- Existing RDMA-based replication protocols
- Tailwind's design
- Evaluation
- Conclusion

# RDMA-based Replication

- Why not just perform RDMA and leave target idle?

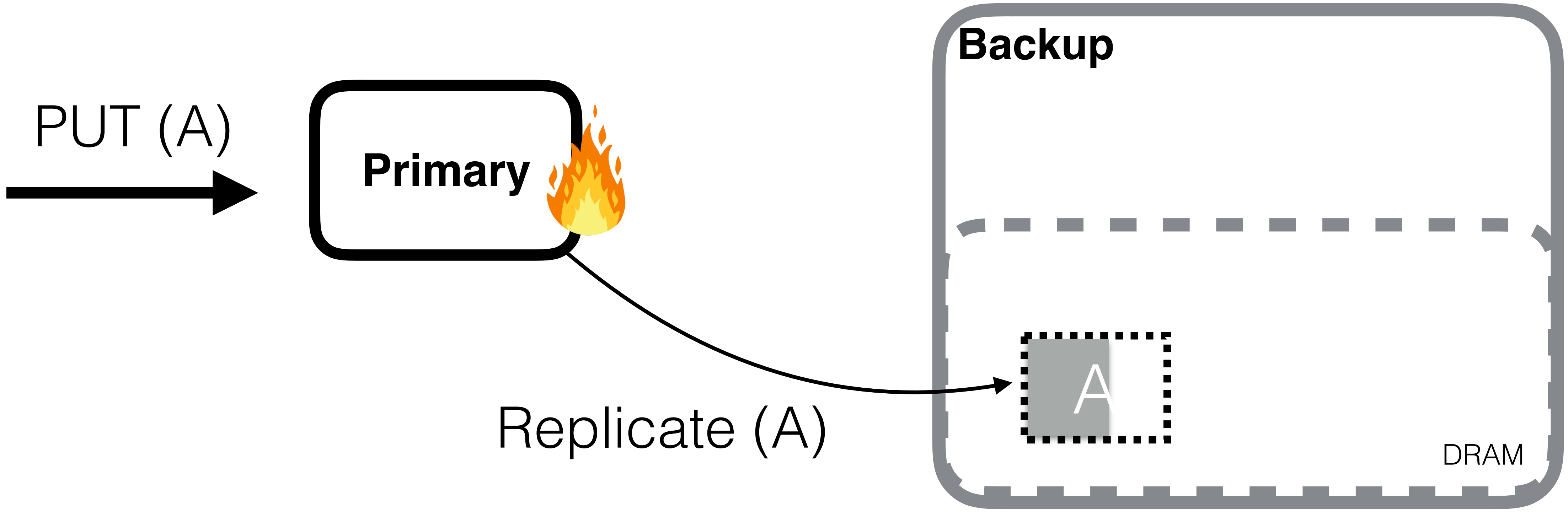
# RDMA-based Replication

- Why not just perform RDMA and leave target idle?



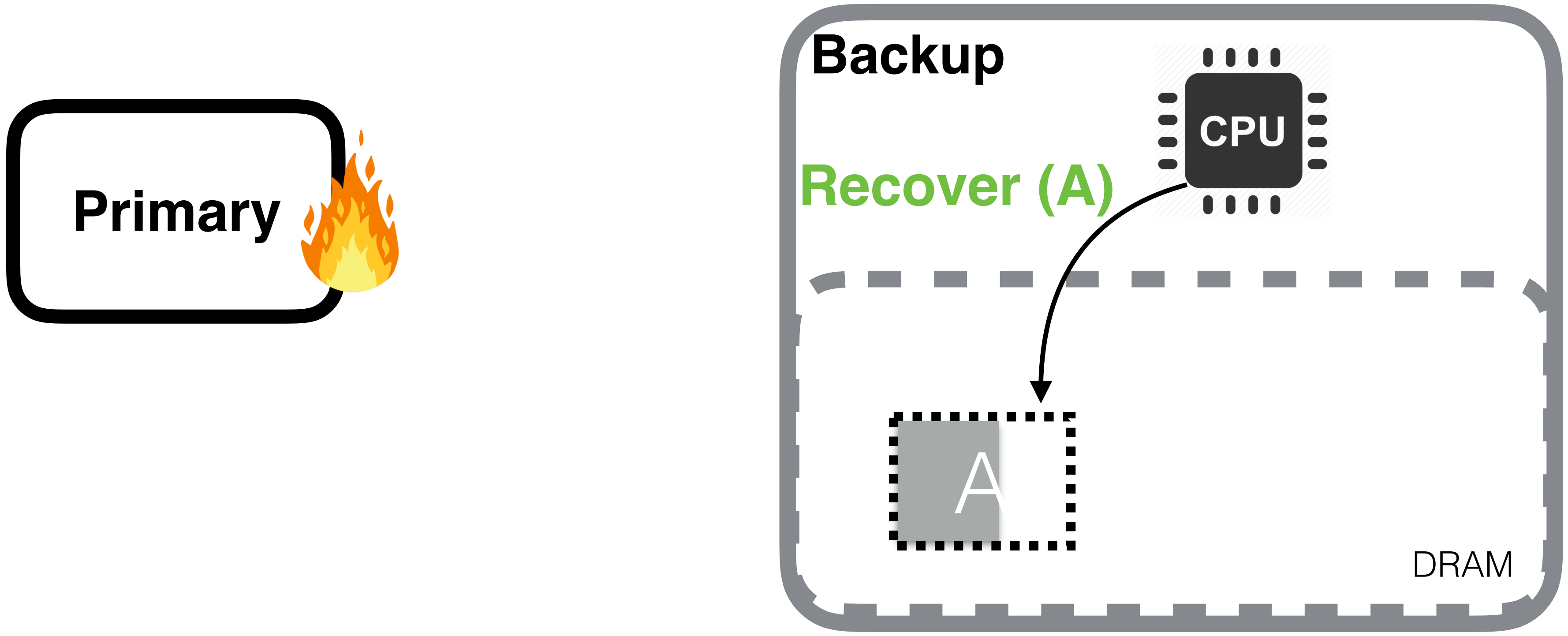
# RDMA-based Replication

- Why not just perform RDMA and leave target idle?



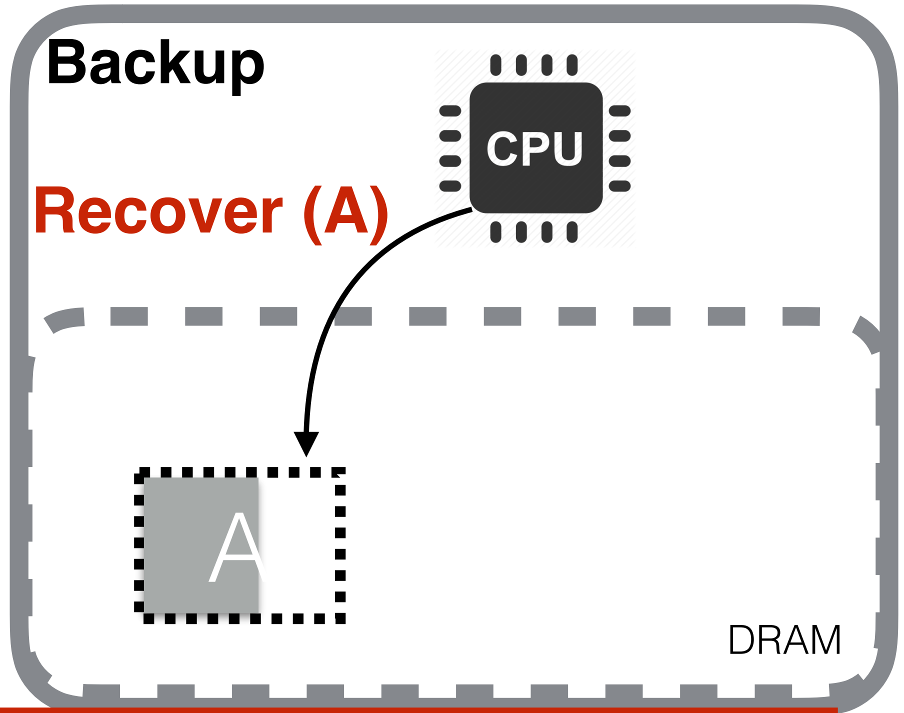
# RDMA-based Replication

- Why not just perform RDMA and leave target idle?



# RDMA-based Replication

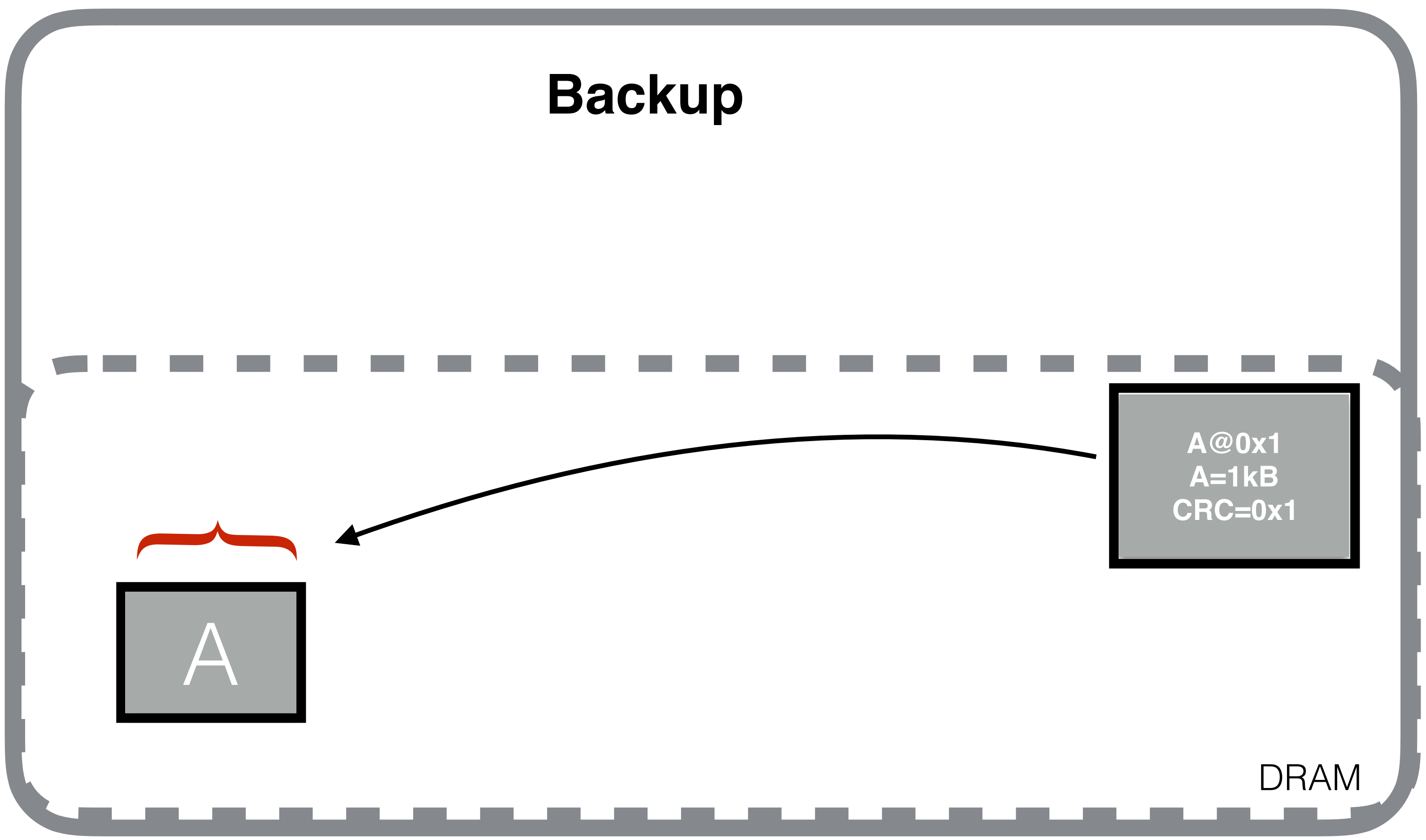
- Why not just perform RDMA and leave target idle?


**The backup cannot "guess" the length of data, it needs some metadata**

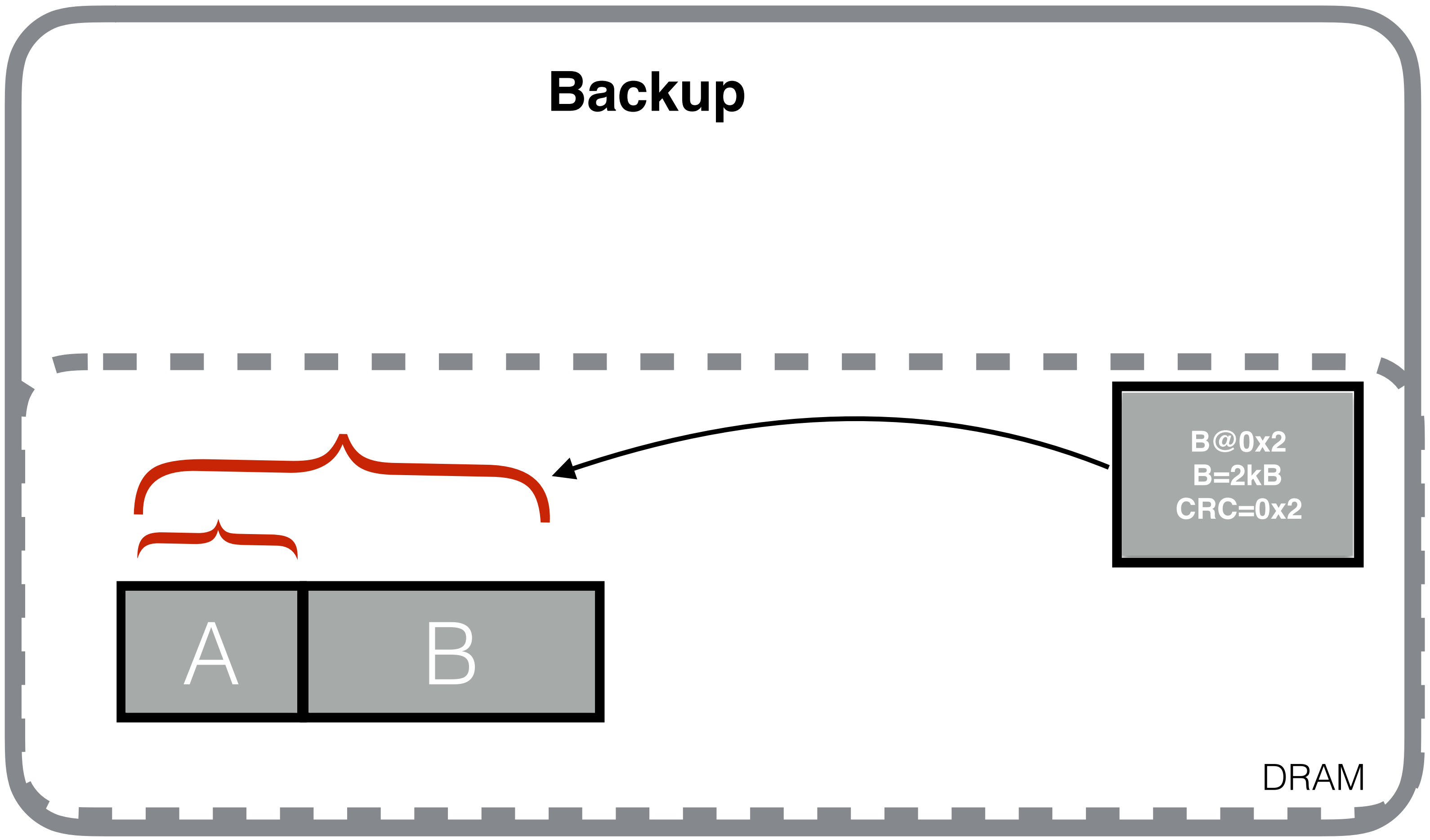
# RDMA-based Replication Metadata

- A backup needs to insure the integrity of log-backup data + length of data



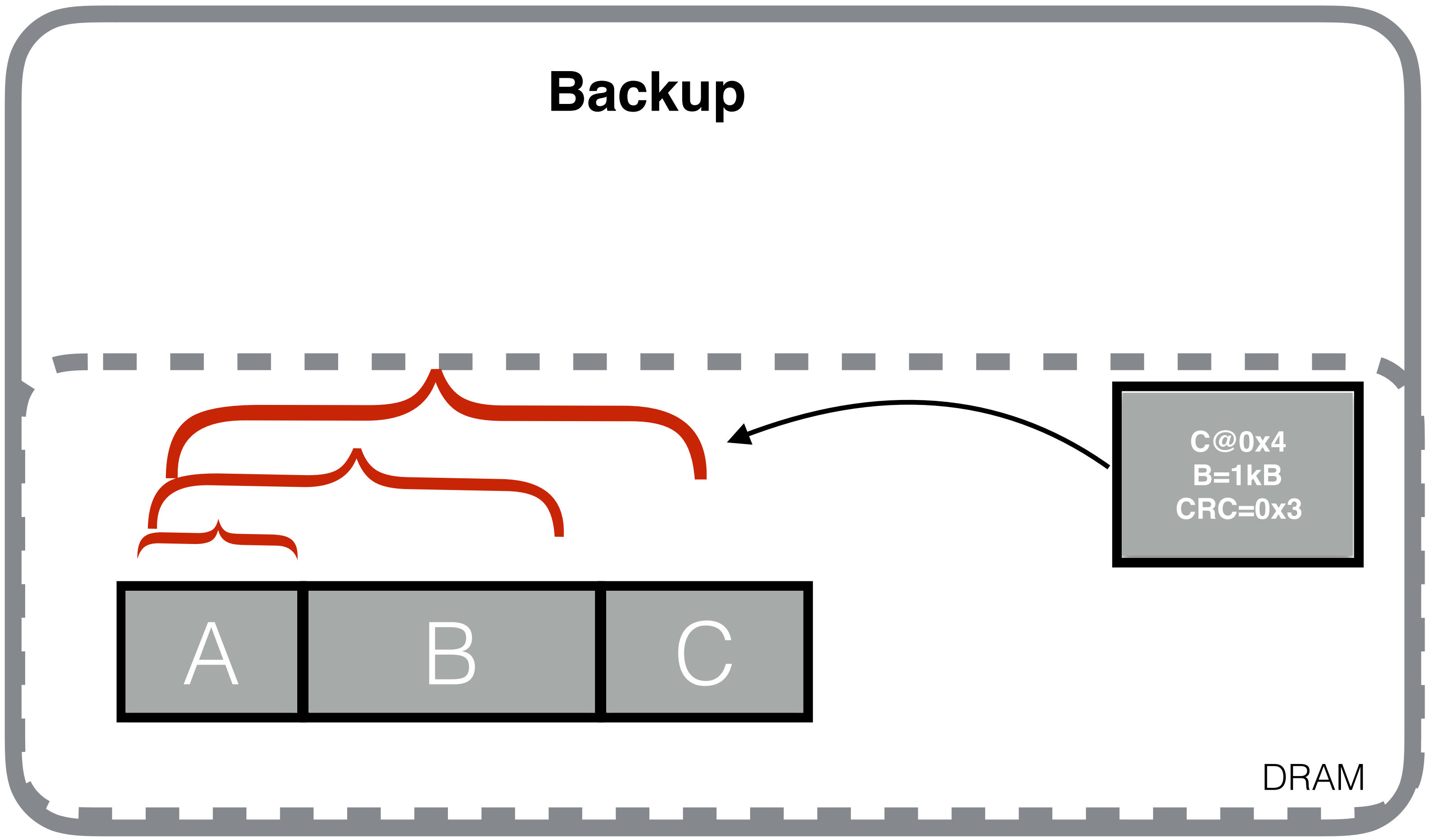
# RDMA-based Replication Metadata

- The last checksum can cover all “previous” log entries



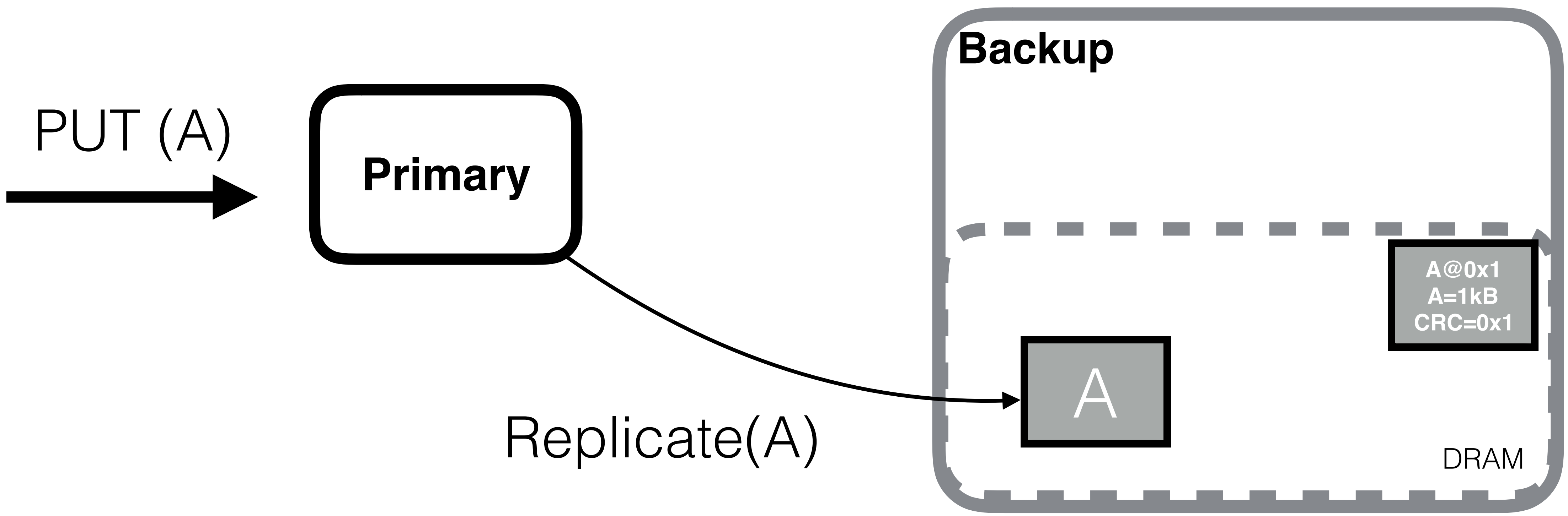
# RDMA-based Replication Metadata

- The last checksum can cover all "previous" log entries



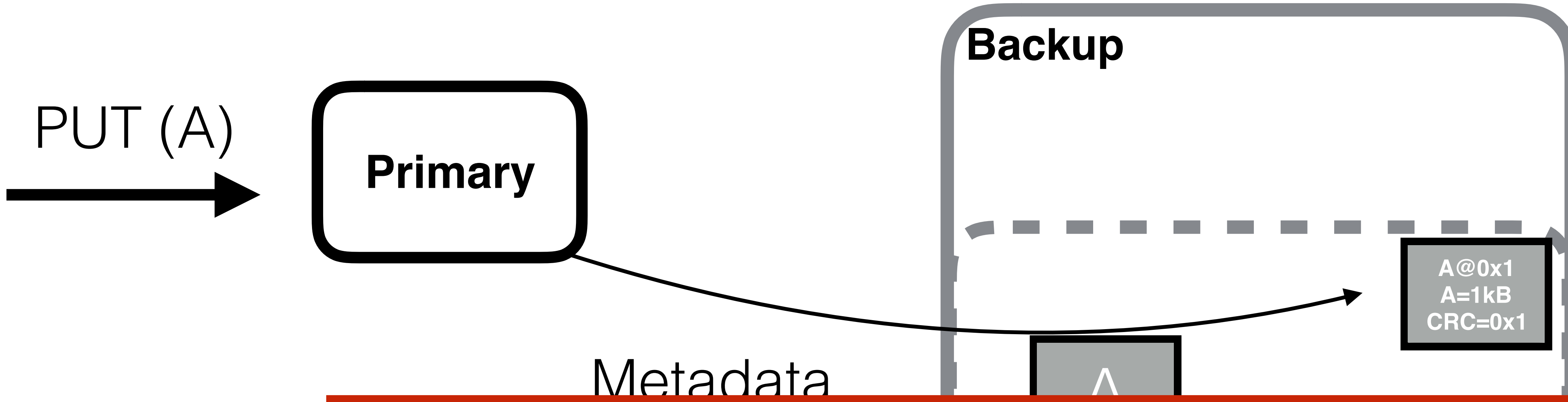
# RDMA-based Replication
# Second Try

- The primary writes metadata on a remote buffer



# RDMA-based Replication
# Second Try

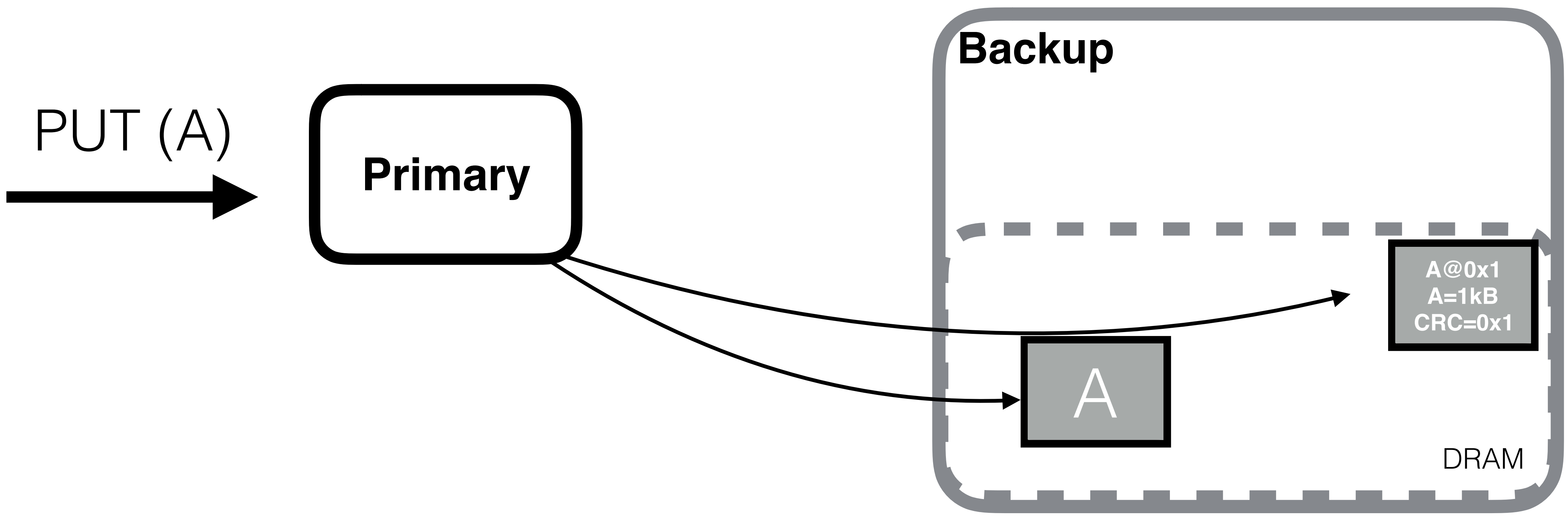
- The primary writes metadata on a remote buffer




**Current RDMA implementations can only target a single, contiguous memory region**

# RDMA-based Replication
# Second Try

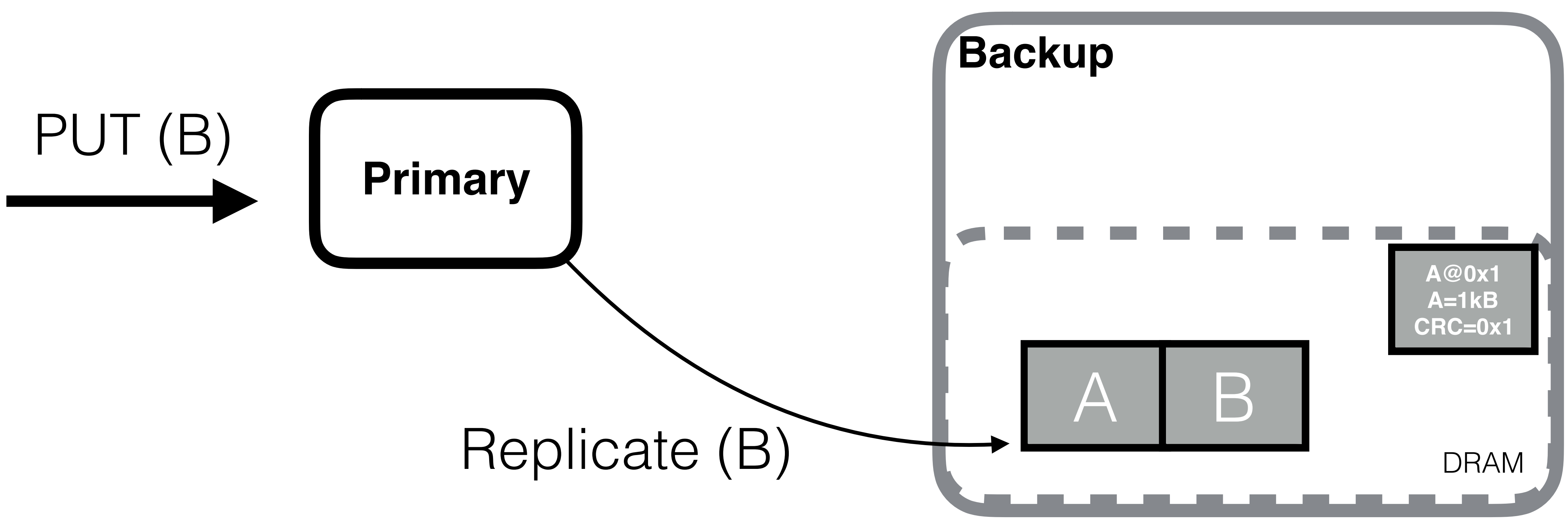
- The primary writes metadata on a remote buffer




2x Messages, RPCs would be more efficient

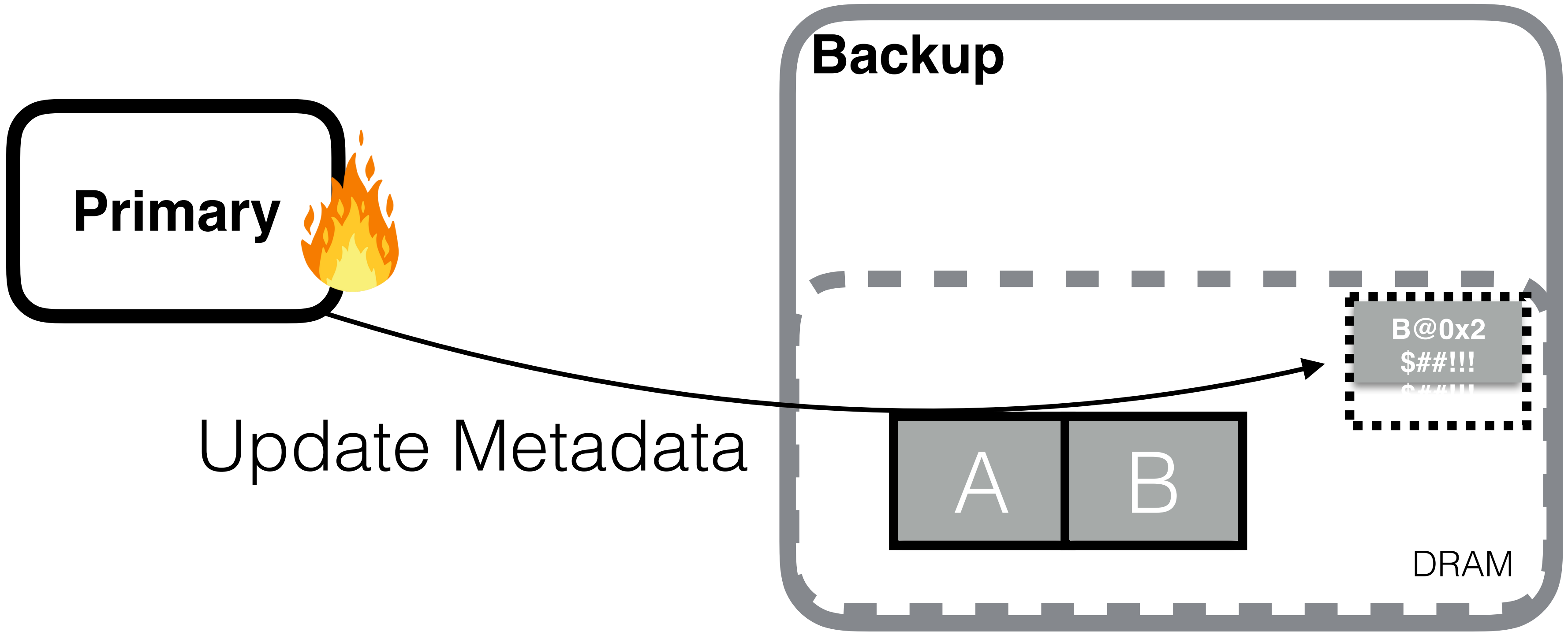
# RDMA-based Replication
# Second Try

- The primary writes metadata on a remote buffer



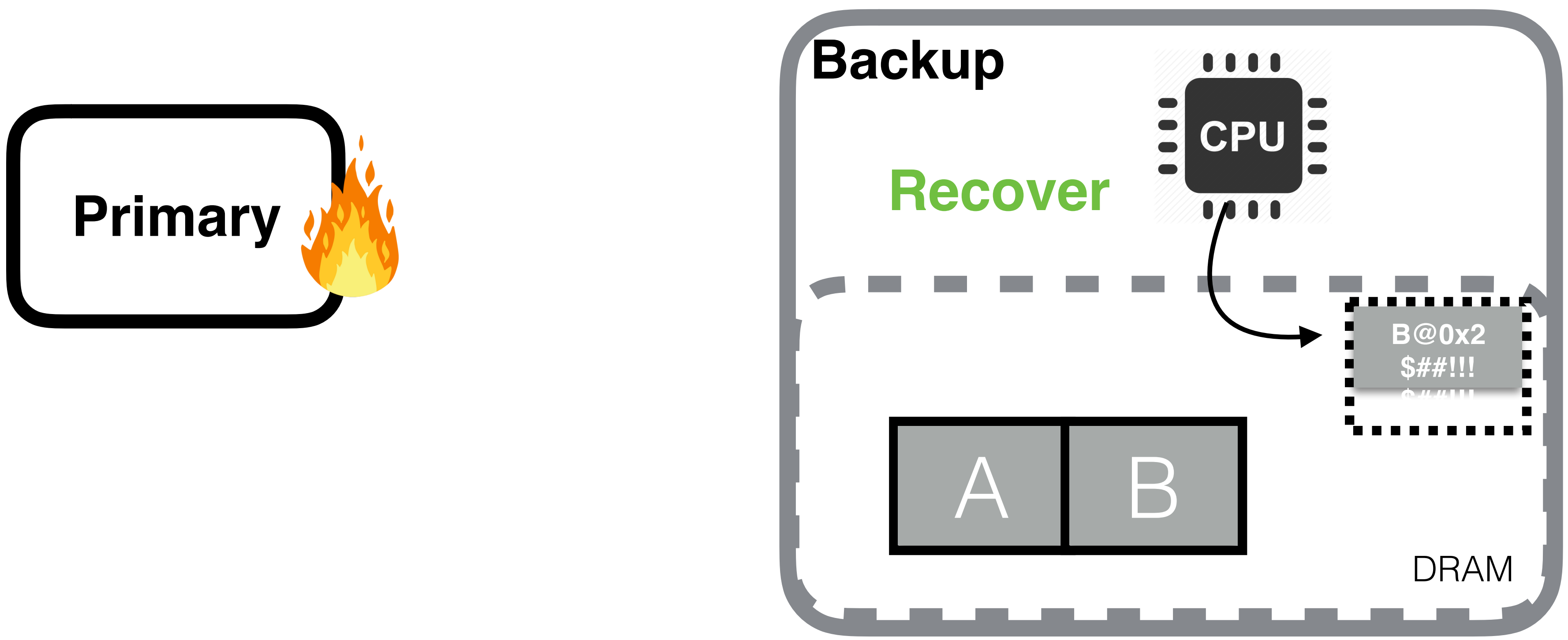
# RDMA-based Replication
# Second Try

- The primary writes metadata on a remote buffer



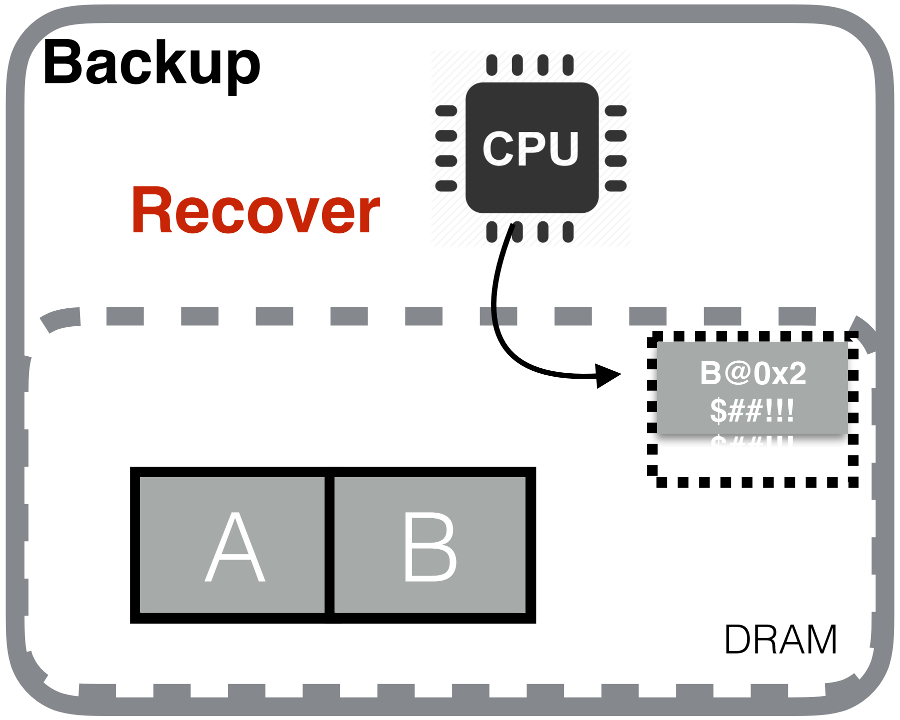
# RDMA-based Replication
# Second Try

- The primary fails in the midst of updating metadata!



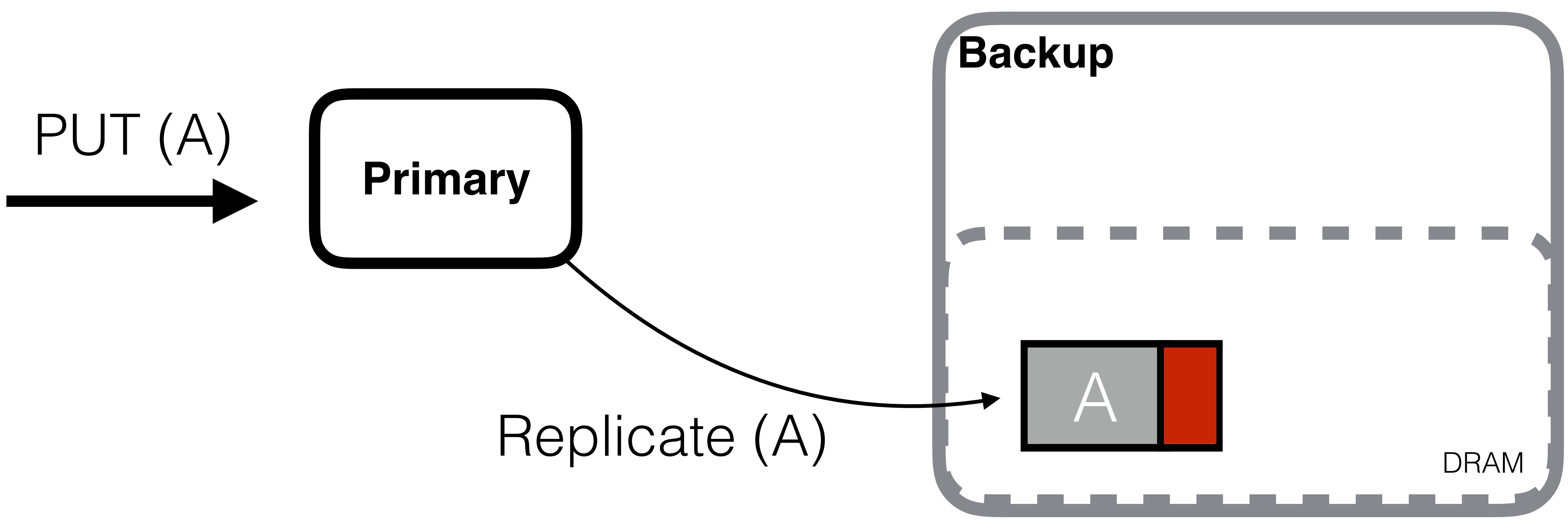
# RDMA-based Replication
# Second Try

- Backup data unusable!

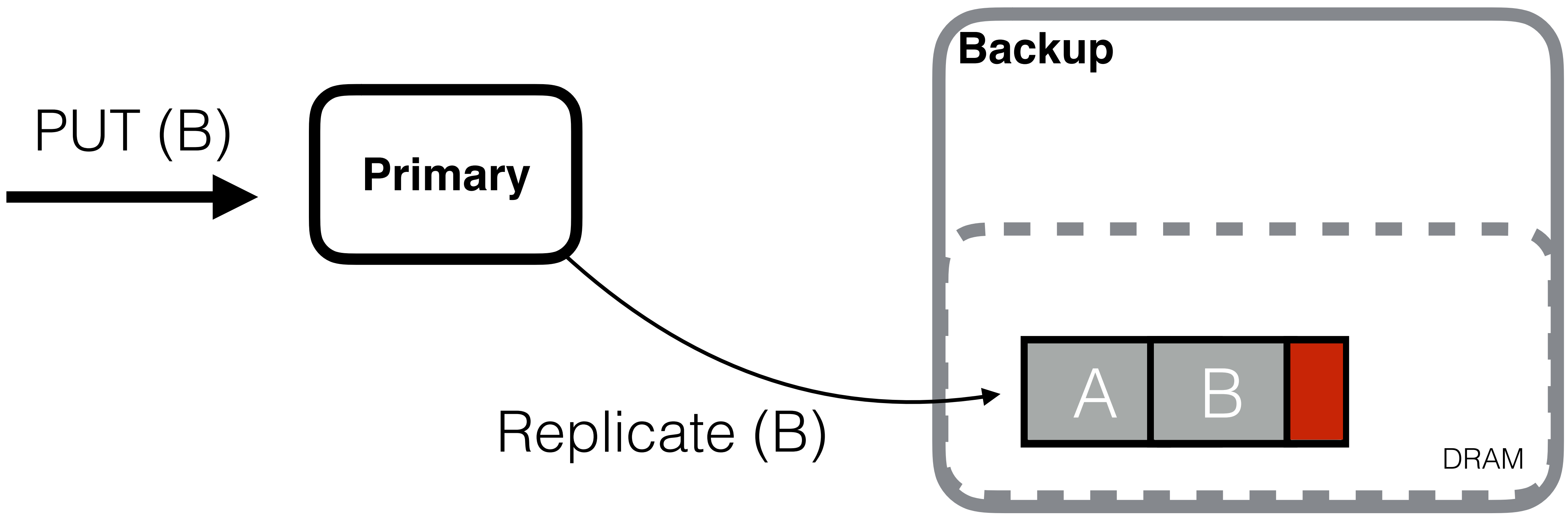
# RDMA-based Replication
# Third Try

- The primary replicates A and corresponding metadata with a single RDMA



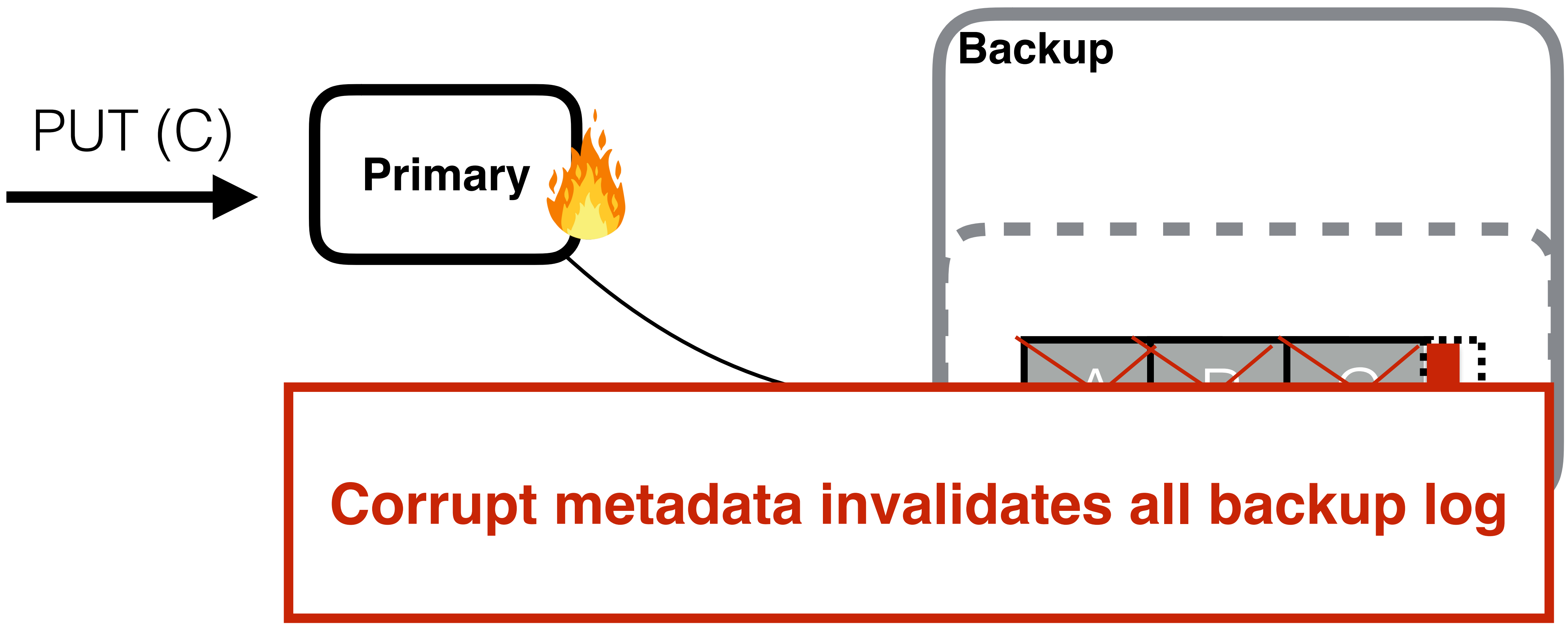
# RDMA-based Replication
# Third Try

- The primary replicates B and corresponding metadata, right after A



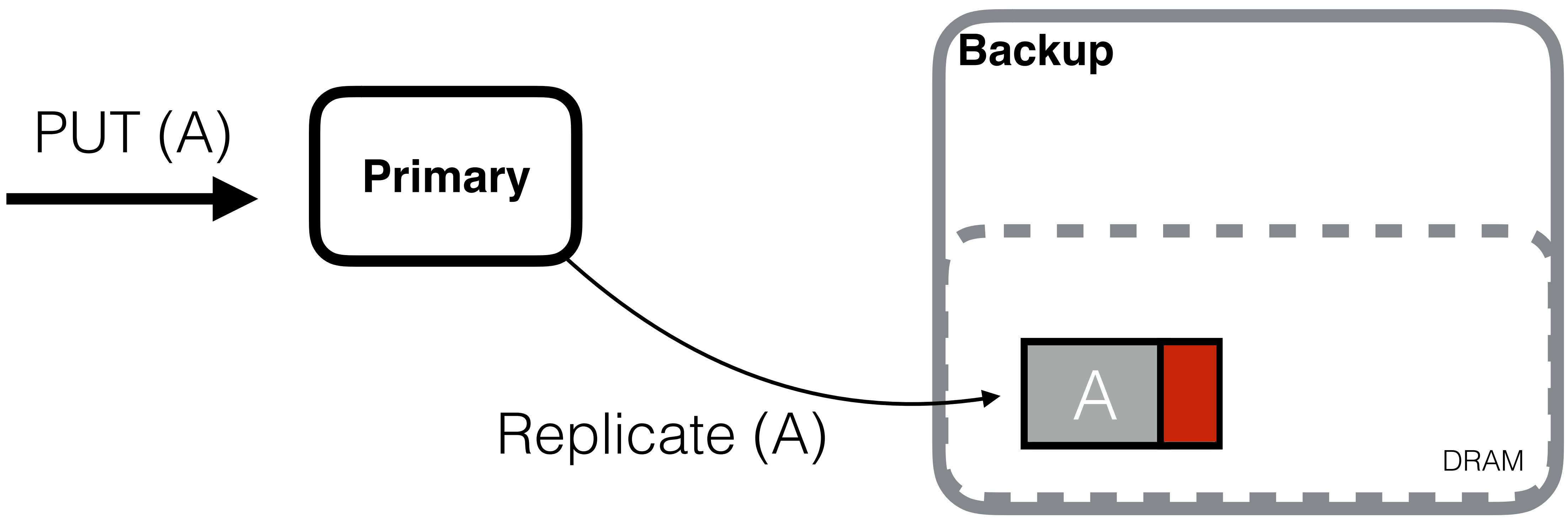
# RDMA-based Replication
# Third Try

- The primary fully replicates C, but partially the metadata



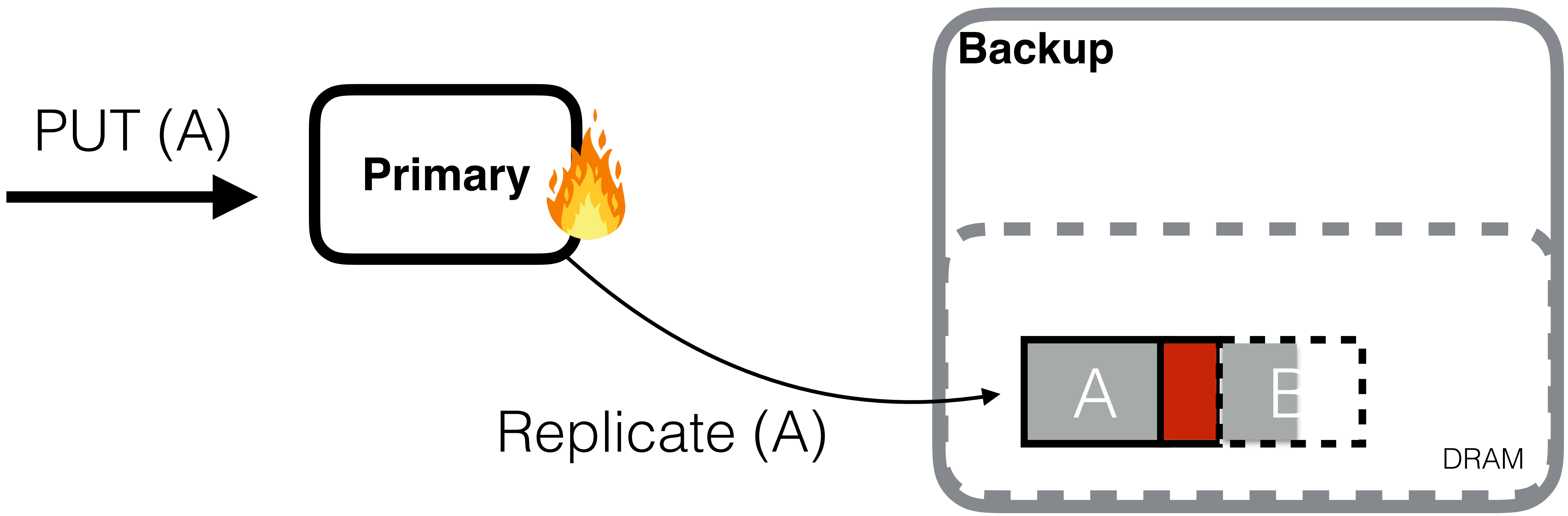
# RDMA-based Replication
# Fourth Try

- The primary replicates A and corresponding metadata



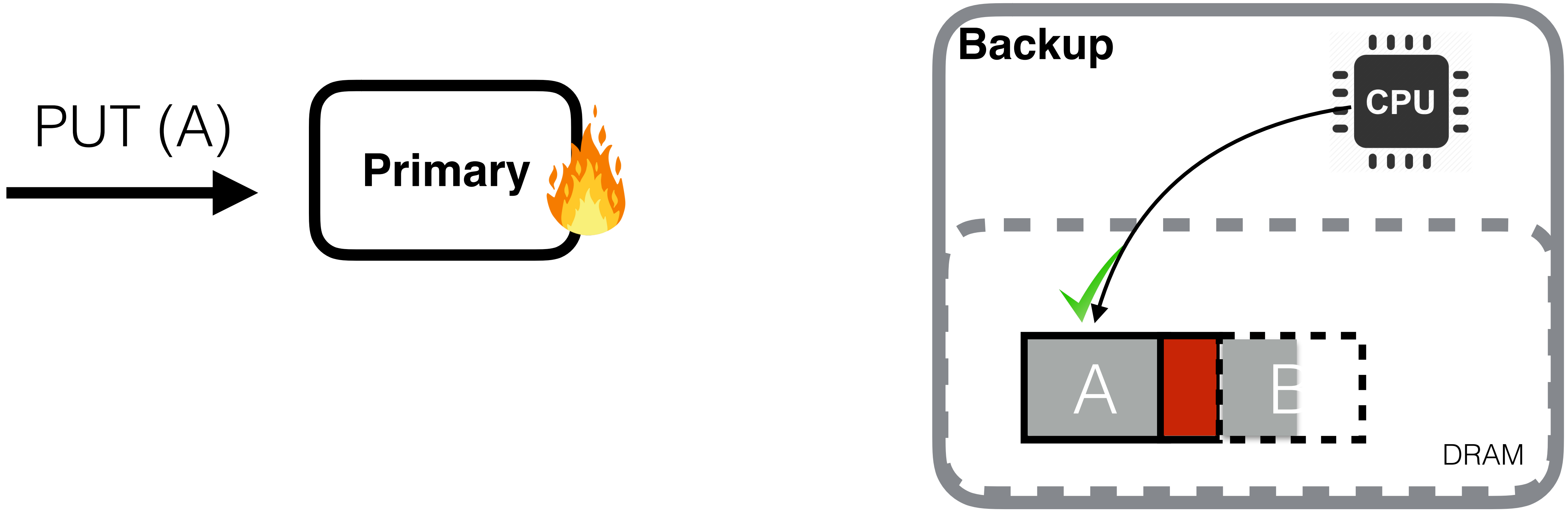
# RDMA-based Replication
# Fourth Try

- The primary partially replicates B, then fails



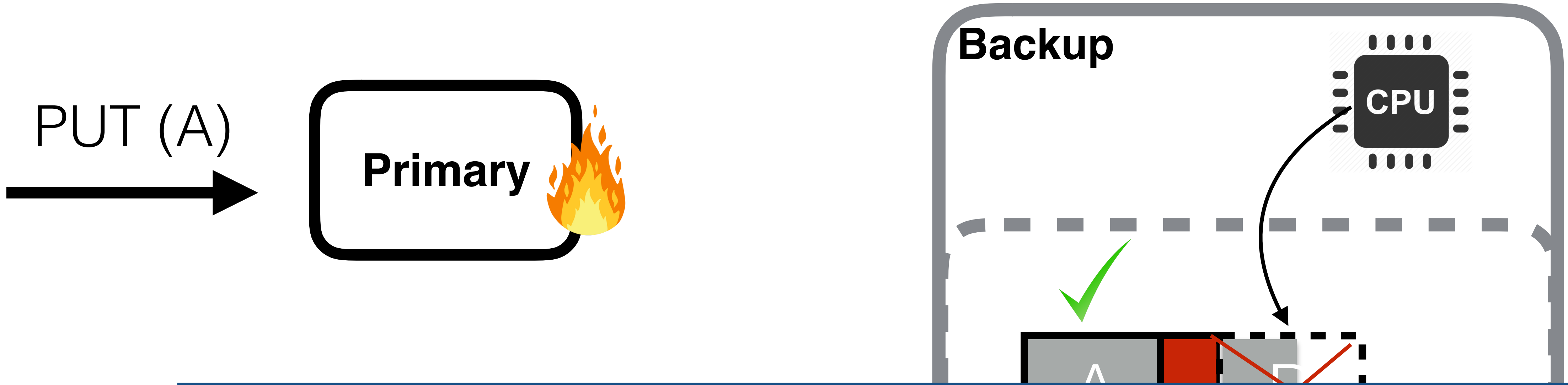
# RDMA-based Replication
# Third Try

- The backup checks if objects were fully received



# RDMA-based Replication
# Third Try

- The backup checks if objects were fully received



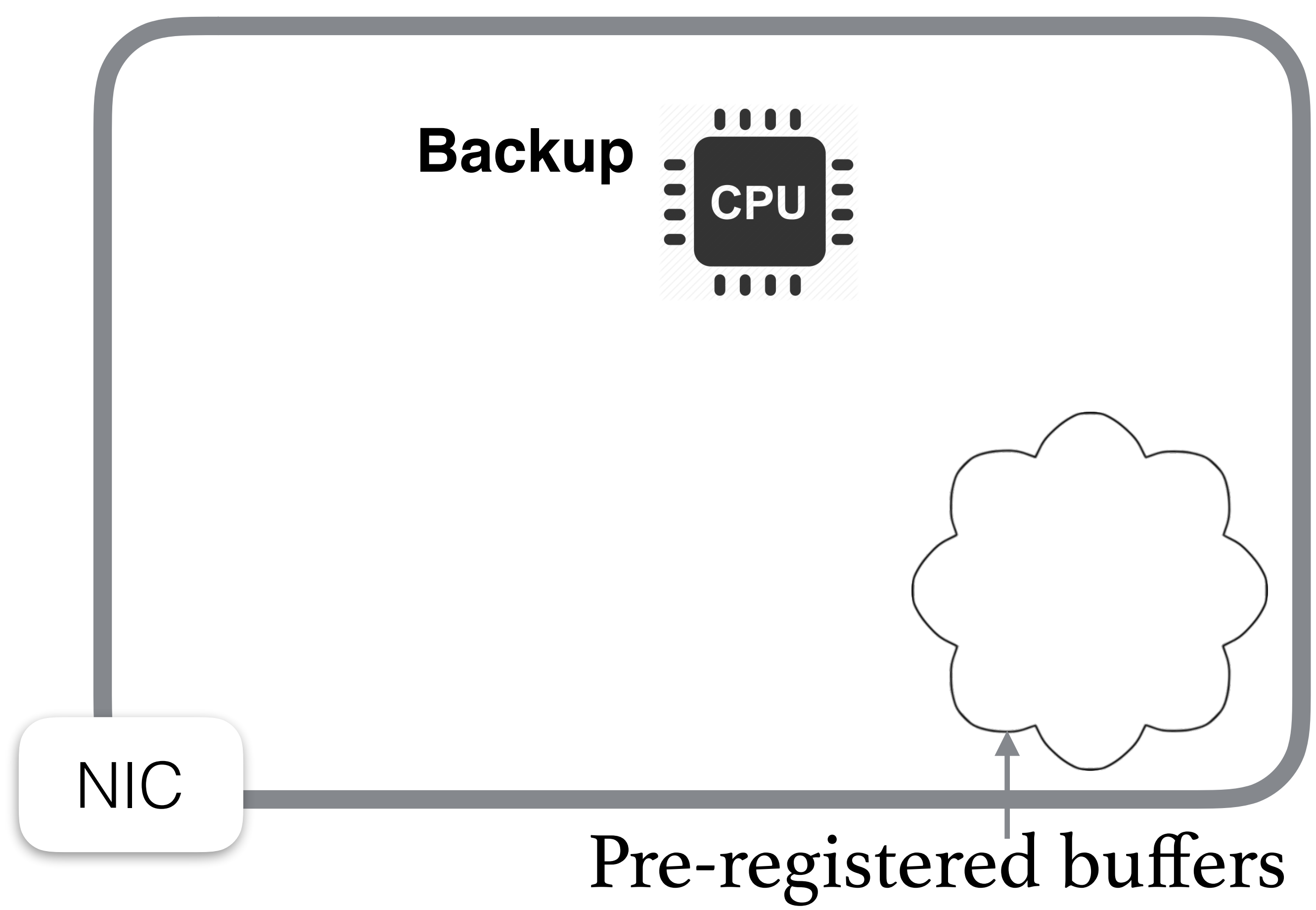

**Only fully-received and correct objects are recovered!**

# Tailwind

- Keep the same client-facing interface (RPCs)
- Strongly-consistent primary-backup systems
- Appends only a 4-byte **CRC32 checksum** after each record
- Relies on **Reliable-Connected** queue pairs: messages are delivered at most once, in order, and without corruption
- Stop failures

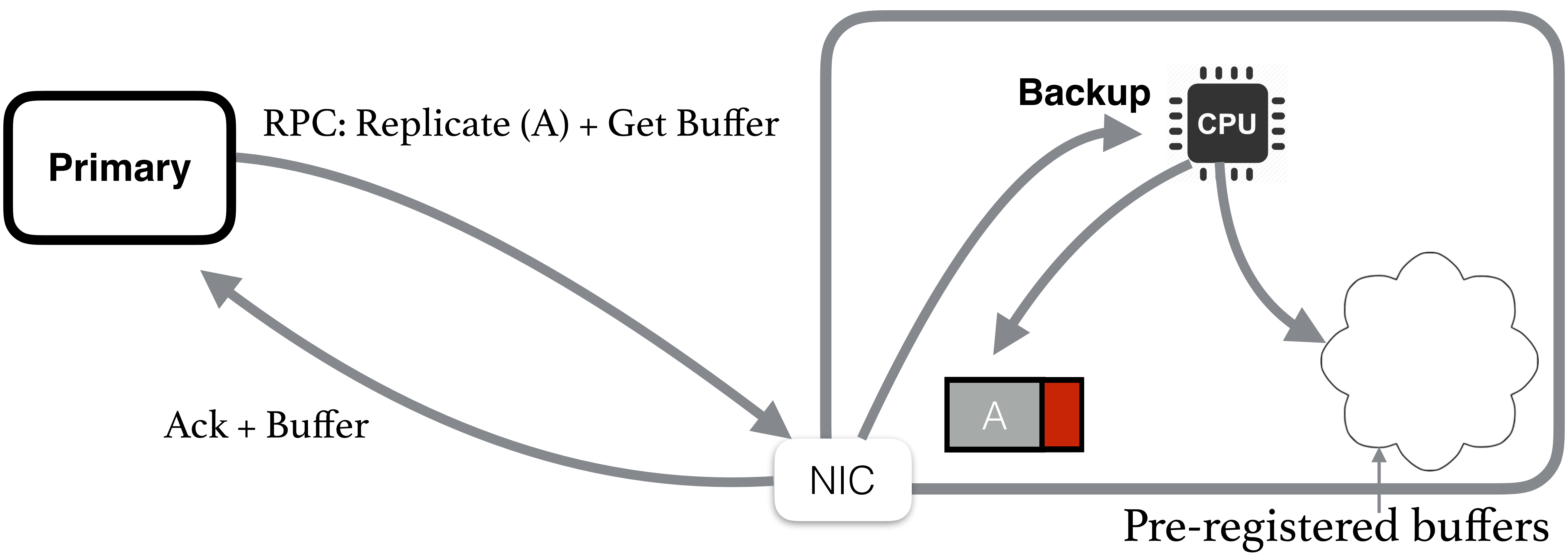
# RDMA Buffers Allocation

- A primary chooses a backup, and requests an RDMA buffer



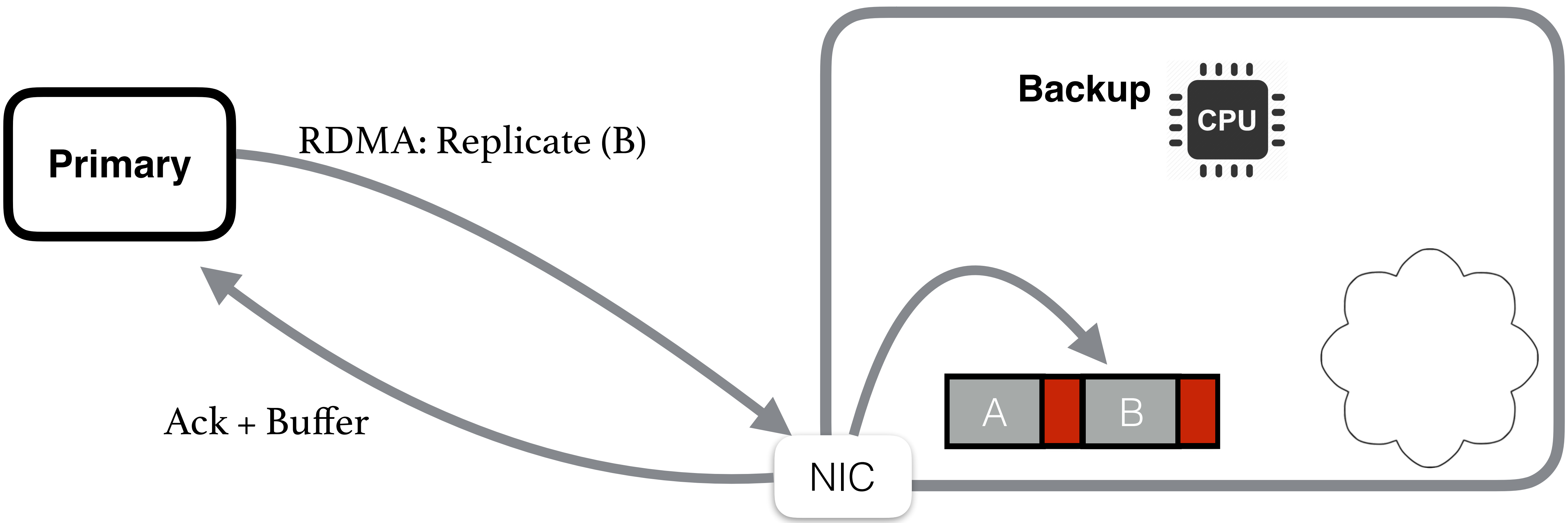
# RDMA Buffers Allocation

- A primary chooses a backup, and requests an RDMA buffer



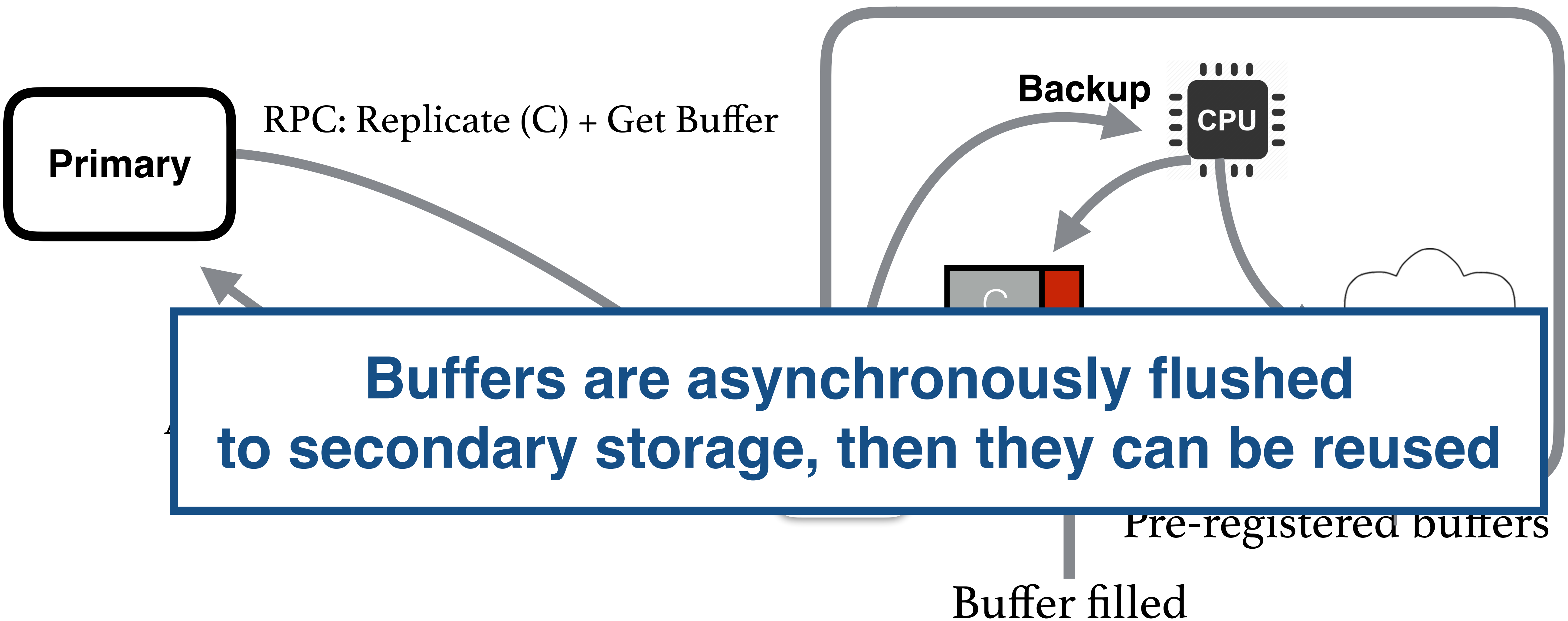
# RDMA Buffers Allocation

- All subsequent replication requests are performed with one-sided RDMA WRITEs



# RDMA Buffers Allocation

- When the primary fills a buffer, it will chose a backup and repeat all steps



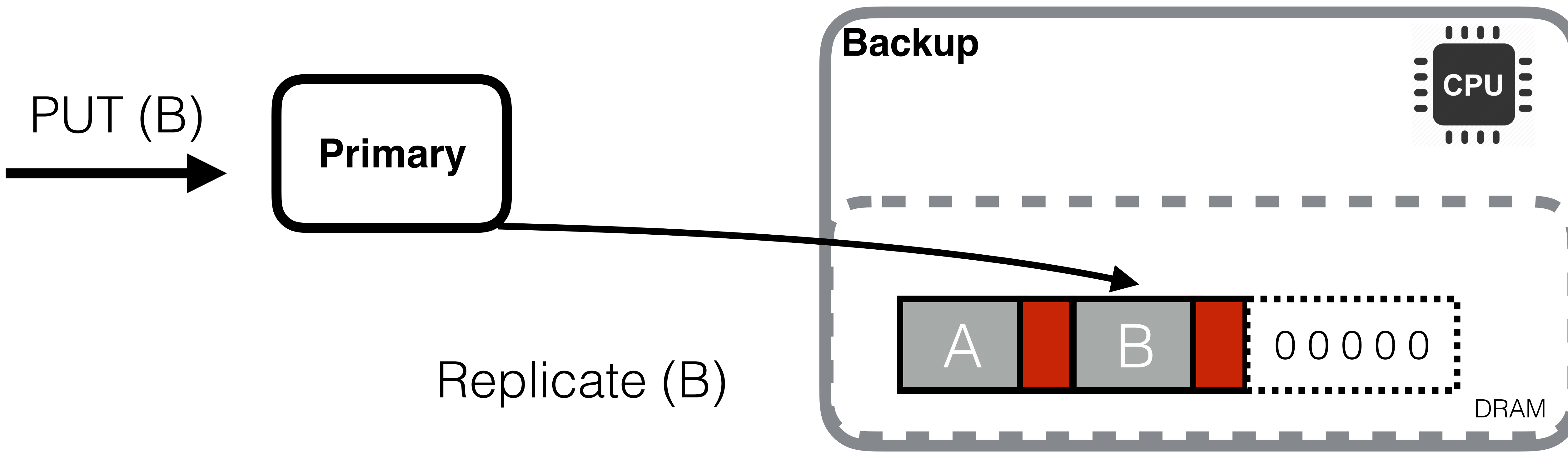

**Buffers are asynchronously flushed to secondary storage, then they can be reused**

# Tailwind: Failures

- Failures can happen at any moment
- $\boxed{\text{RDMA complicates primary replica failures}}$
- Secondary replica failures are naturally dealt with in storage systems

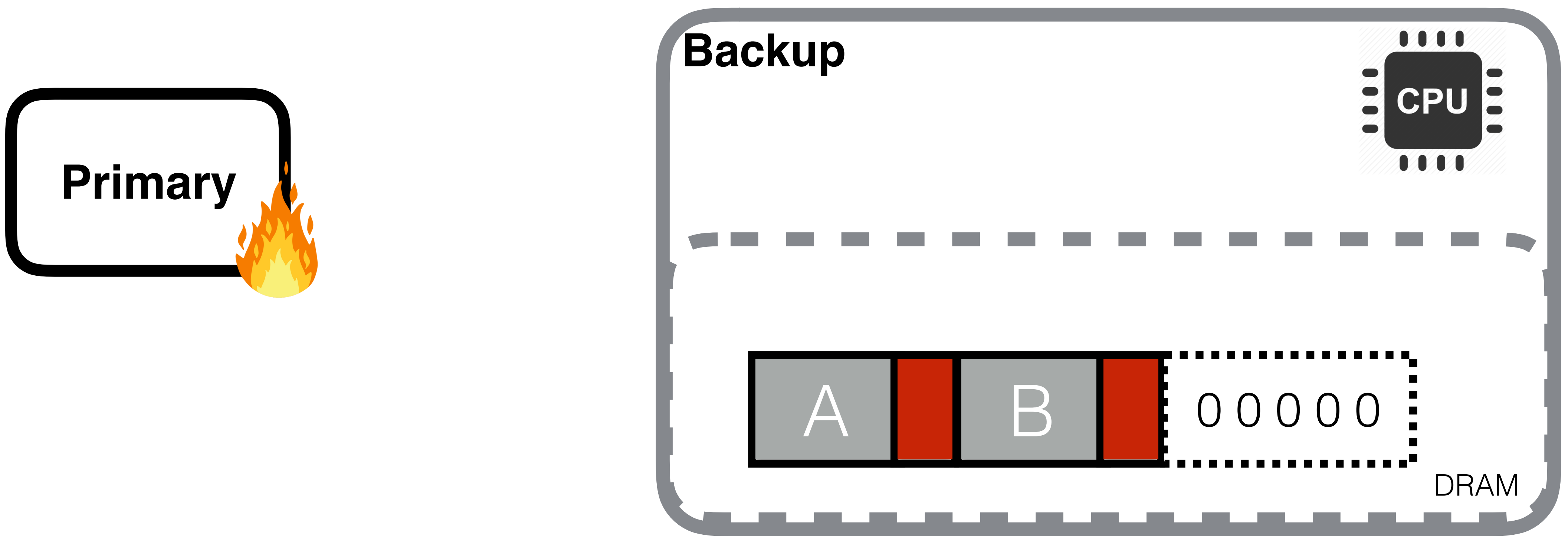
# Failure scenarios: Fully Replicated Objects

- Case 1: The object + its metadata are correctly transferred



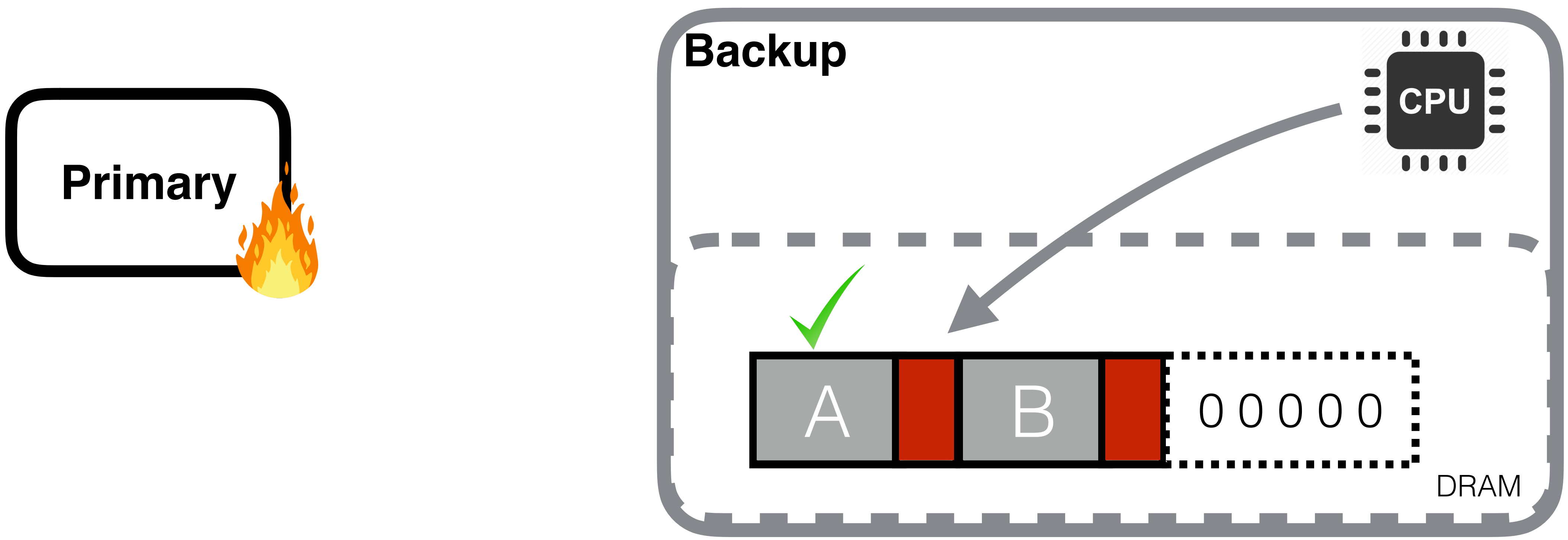
# Failure scenarios: Fully Replicated Objects

- Case 1: The object + its metadata are correctly transferred



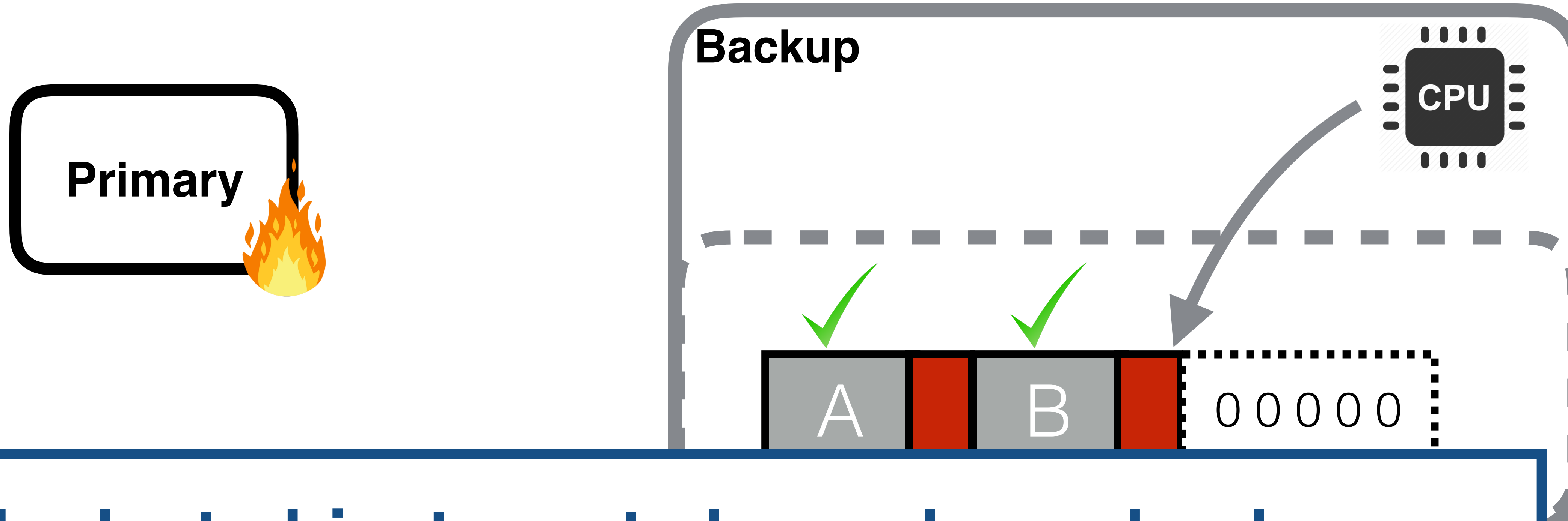
# Failure scenarios: Fully Replicated Objects

- Case 1: The object + its metadata are correctly transferred



# Failure scenarios: Fully Replicated Objects

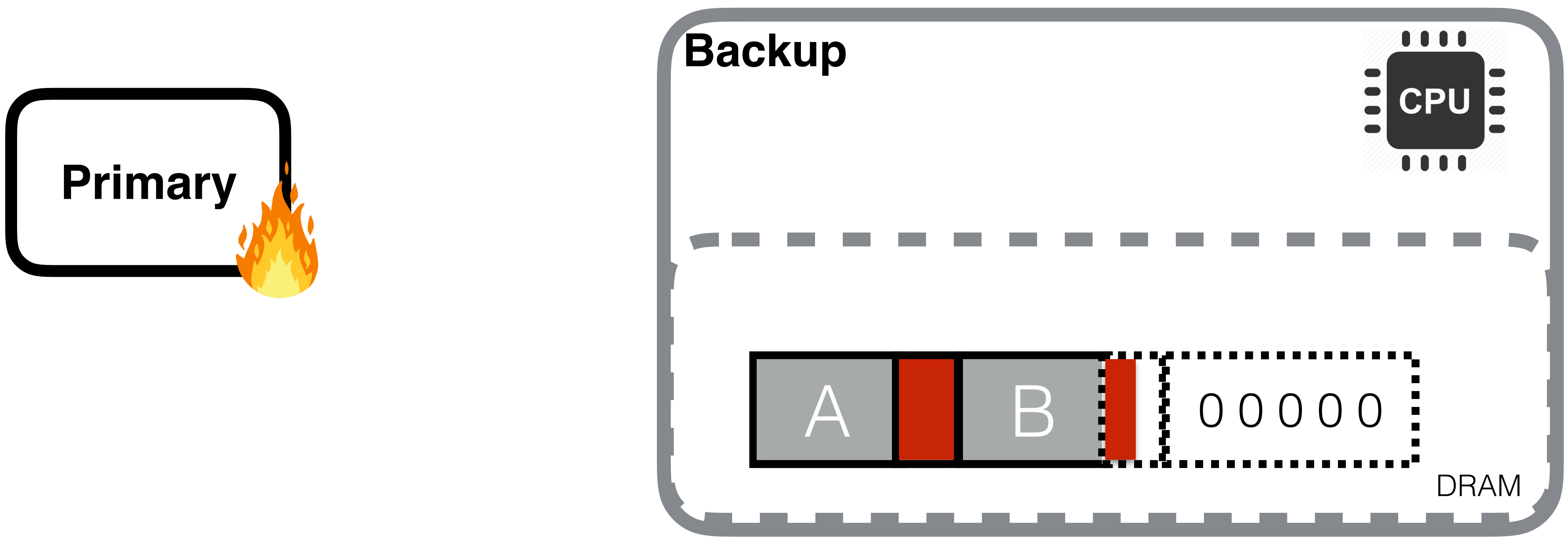
- Case 1: The object + its metadata are correctly transferred




**The last object must always be a checksum + Checksums are not allowed to be zeroed**

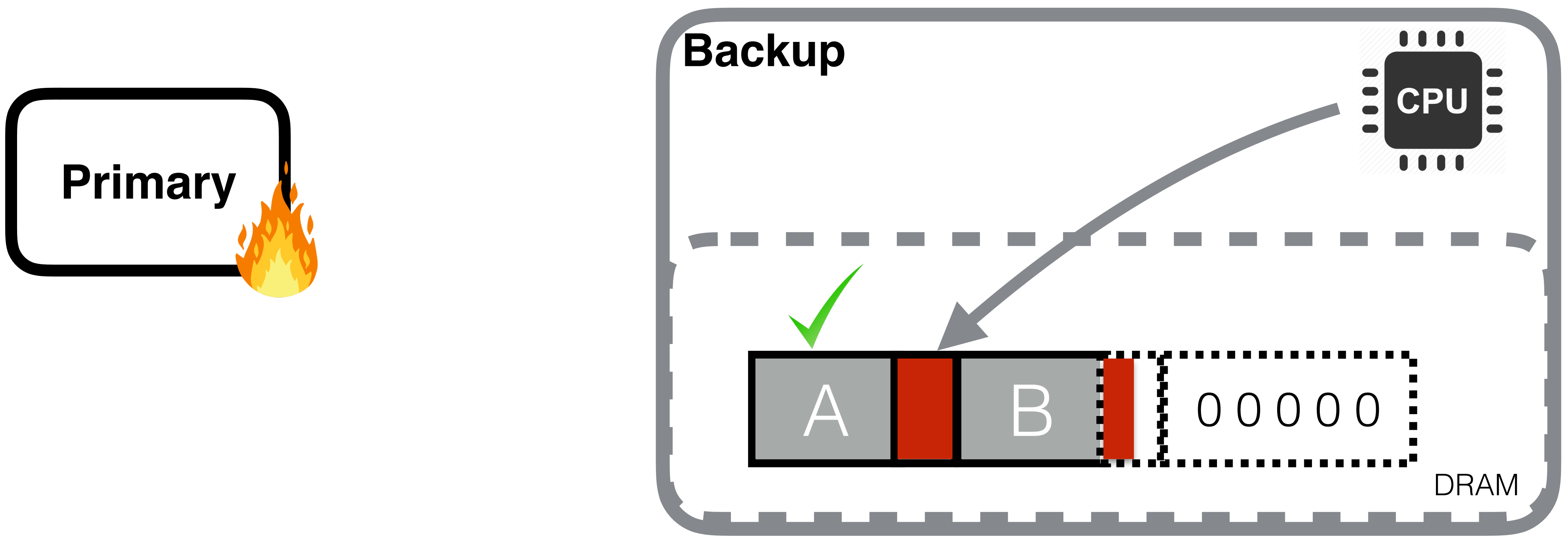
# Failure scenarios: Partially Written Checksum

- Case 2: Partially transferred checksum



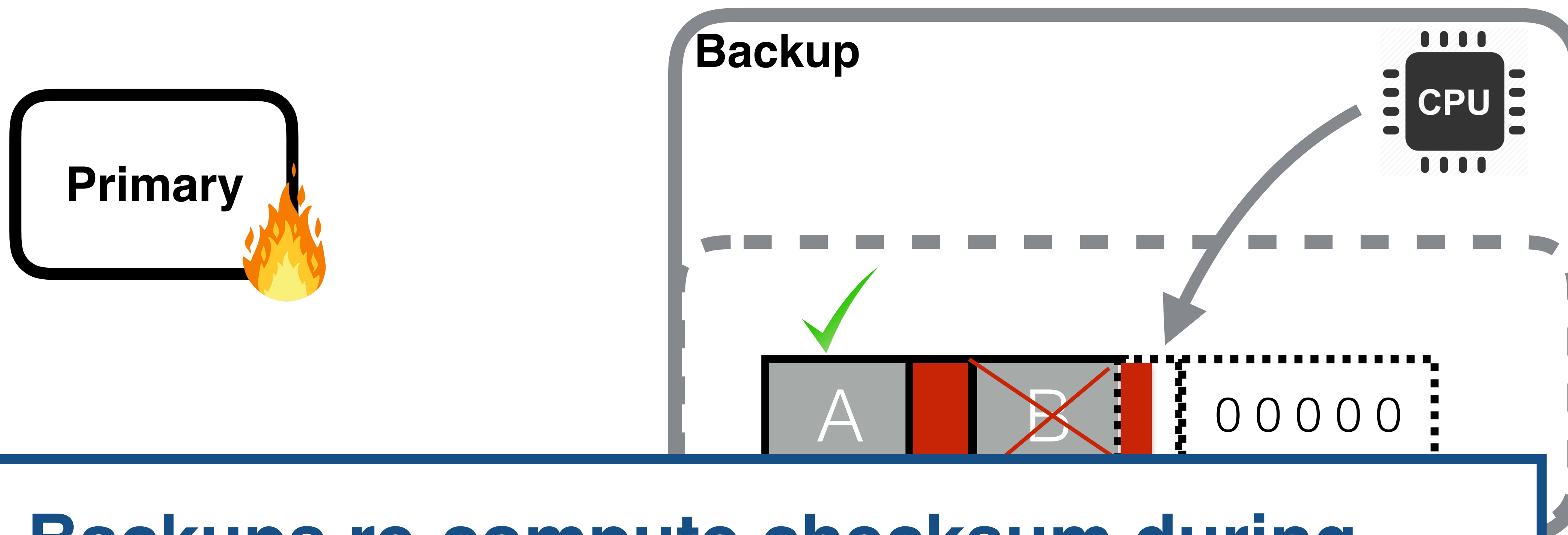
# Failure scenarios: Partially Written Checksum

- Case 2: Partially transferred checksum



# Failure scenarios: Partially Written Checksum

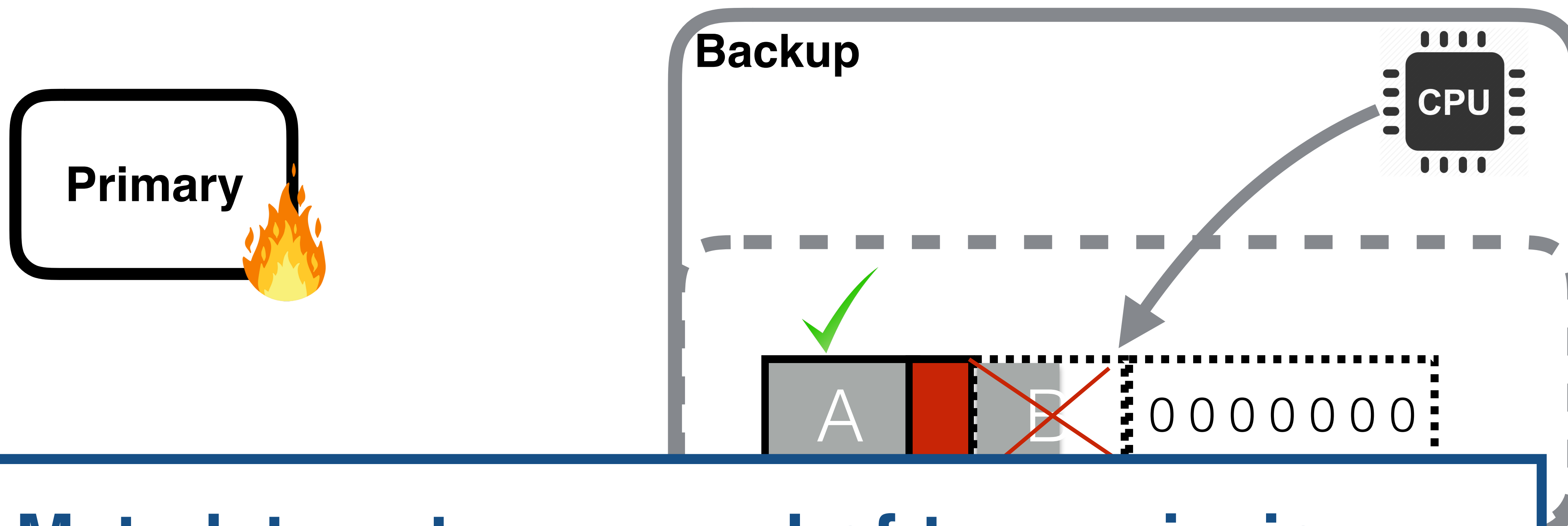
- Case 2: Partially transferred checksum




**Backups re-compute checksum during recovery and compare it with the stored one**

# Failure scenarios: Partially Written Object

- Case 3: Partially transferred object




**Metadata act as an end-of-transmission marker**

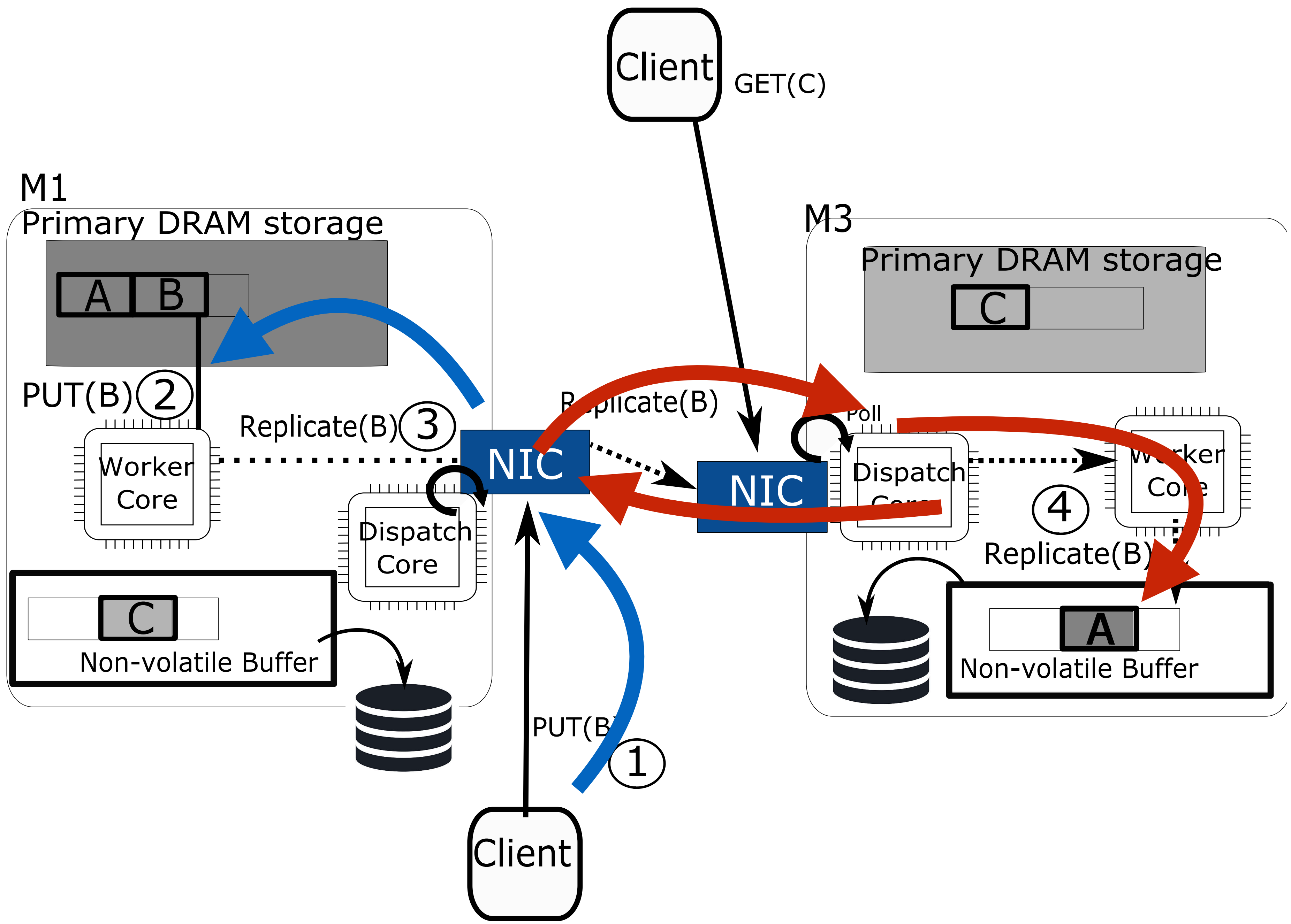
# Outline

- Context
- Existing RDMA-based replication protocols
- Tailwind's design
- Evaluation
- Conclusion

# Implementation

- Implemented on the RAMCloud in-memory kv-store
- Low latency, large scale, strongly consistent
- RPCs leverage fast networking and kernel bypass
- Keeps all data in memory, durable storage for backups

# RAMCloud Threading Architecture



# Evaluation Configuration

- Yahoo! Cloud Serving Benchmark (Workloads A (50%PUT), B(5%PUT), WRITE-ONLY)
- 20 million - 100 bytes objects + 30 byte/key
- Requests generated with a Zipfian distribution
- RAMCloud replication vs Tailwind (**3-way replication**)

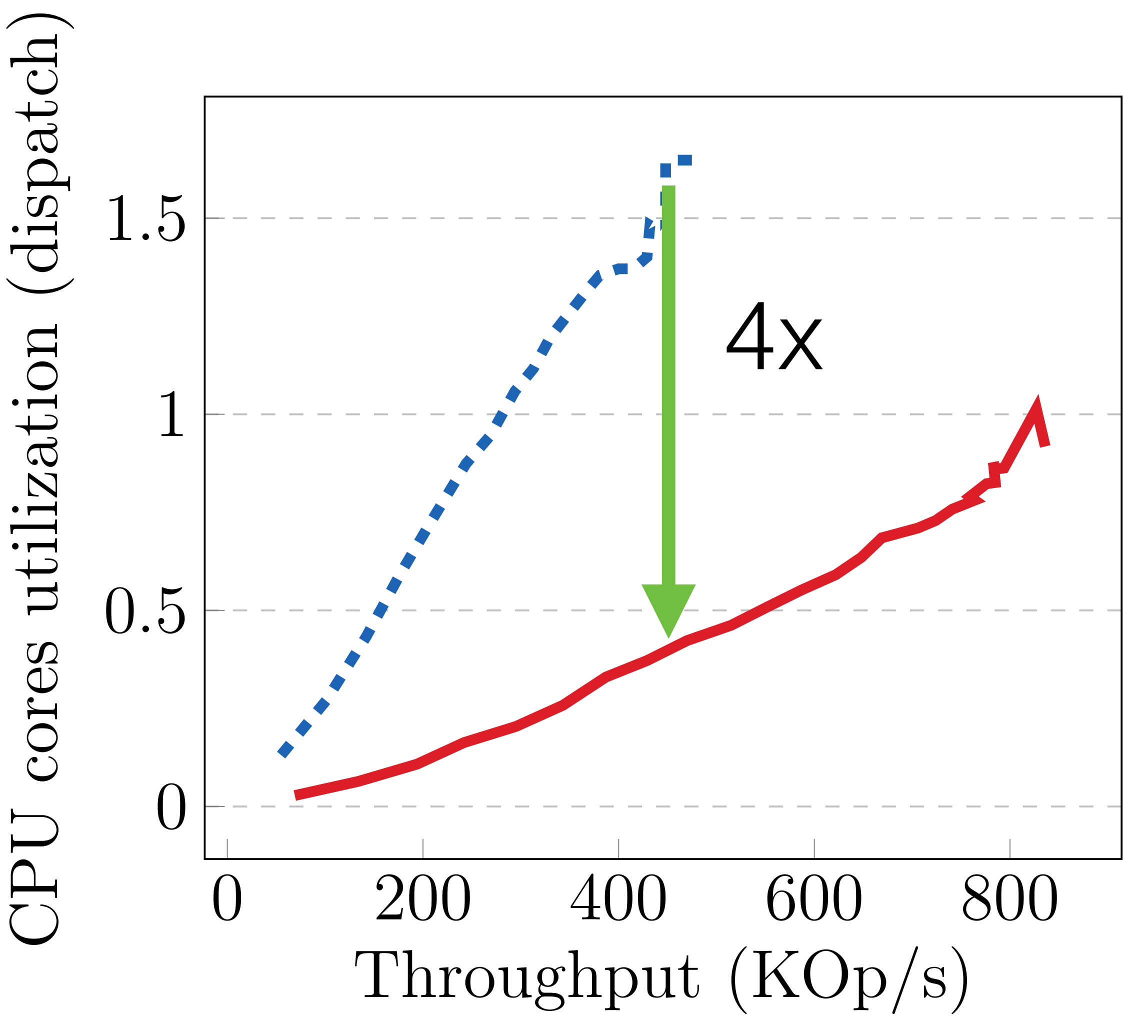
| | |
|---|---|
| **CPU** | Xeon E5-2450 2.1 GHz 8 cores, 16 hw threads |
| **RAM** | 16 GB 1600 MHz DDR3 |
| **NIC** | Mellanox MX354A CX3 @ 56 Gbps |
| **Switch** | 36 port Mellanox SX6036G |
| **OS** | Ubuntu 15.04, Linux 3.19.0-16,<br>MLX4 3.4.0, libibverbs 1.2.1 |

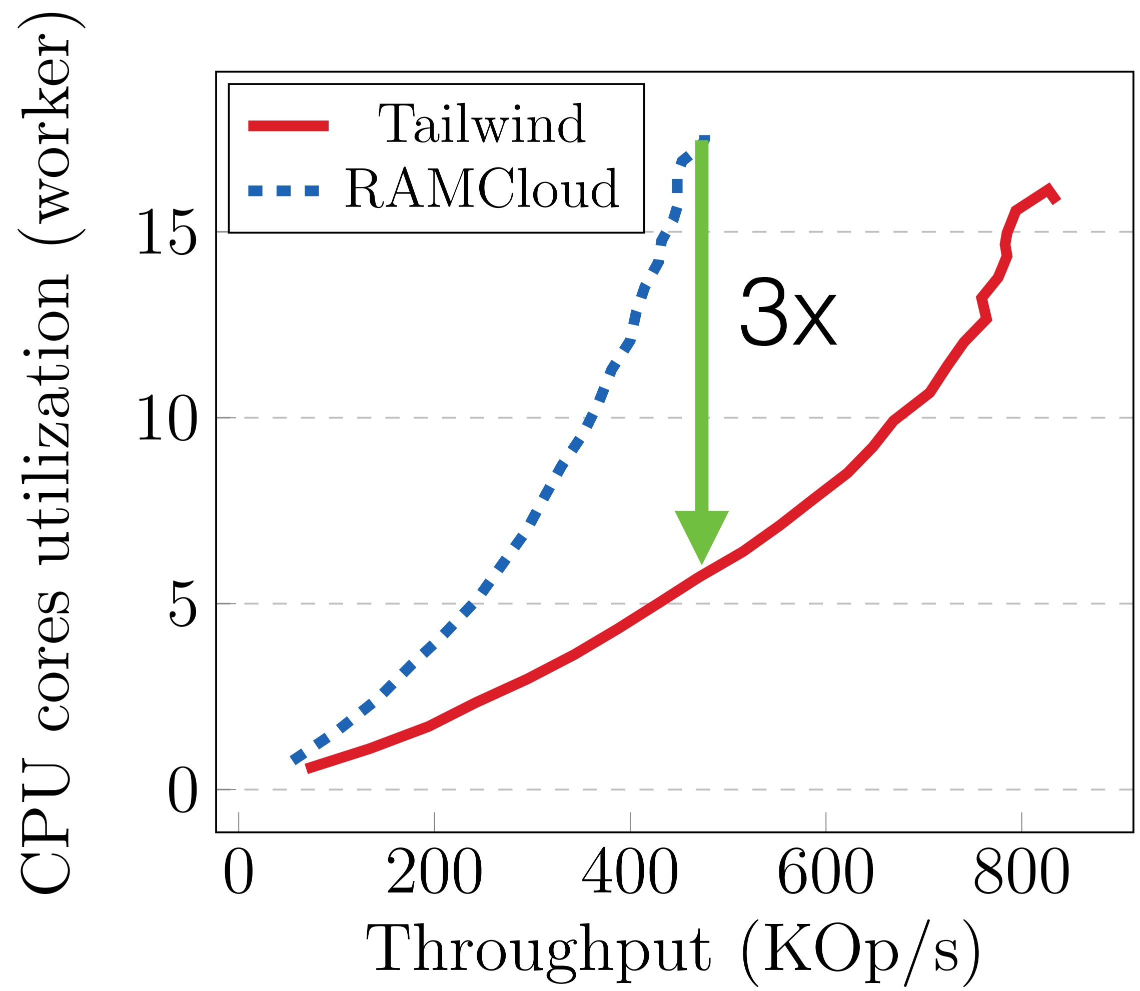
# Evaluation Goals

- How much CPU cycles can Tailwind save?
- Does it improve performance?
- Is there any overhead?

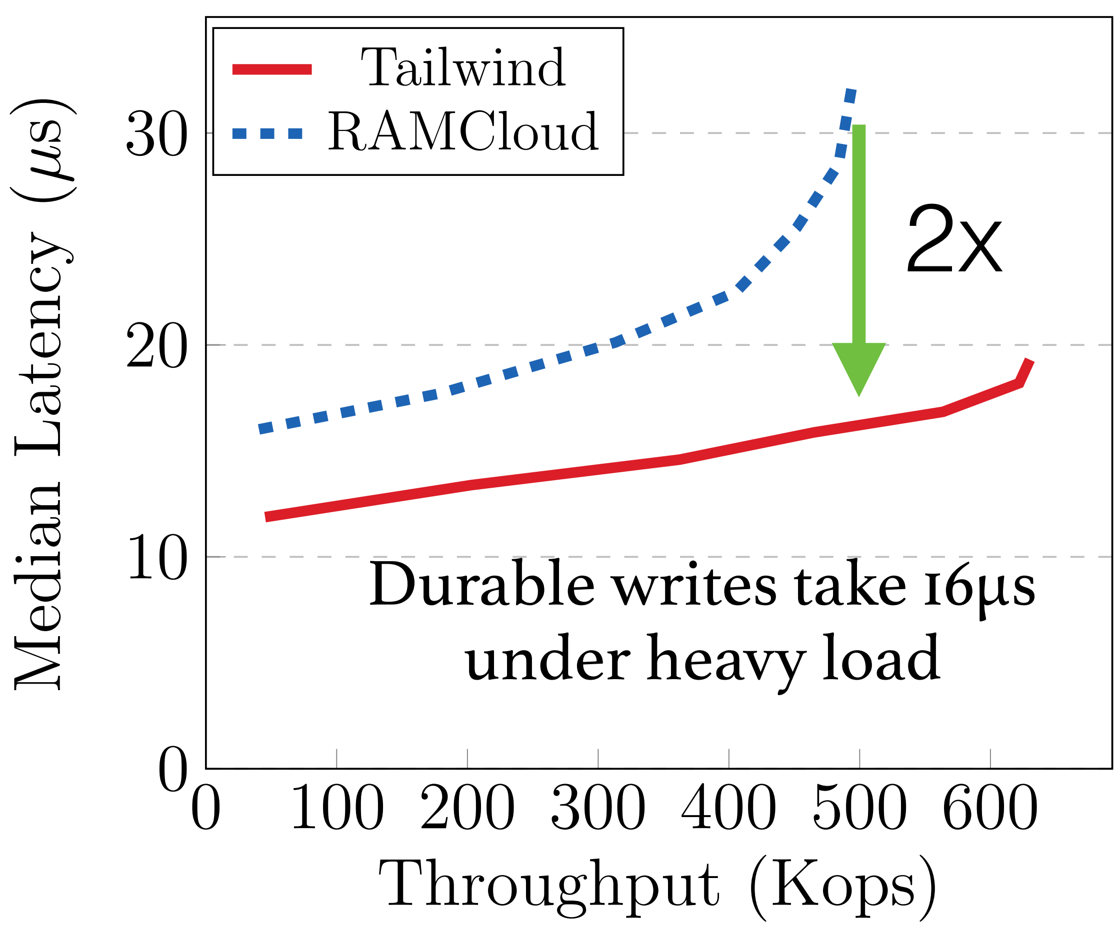
# Evaluation: CPU Savings

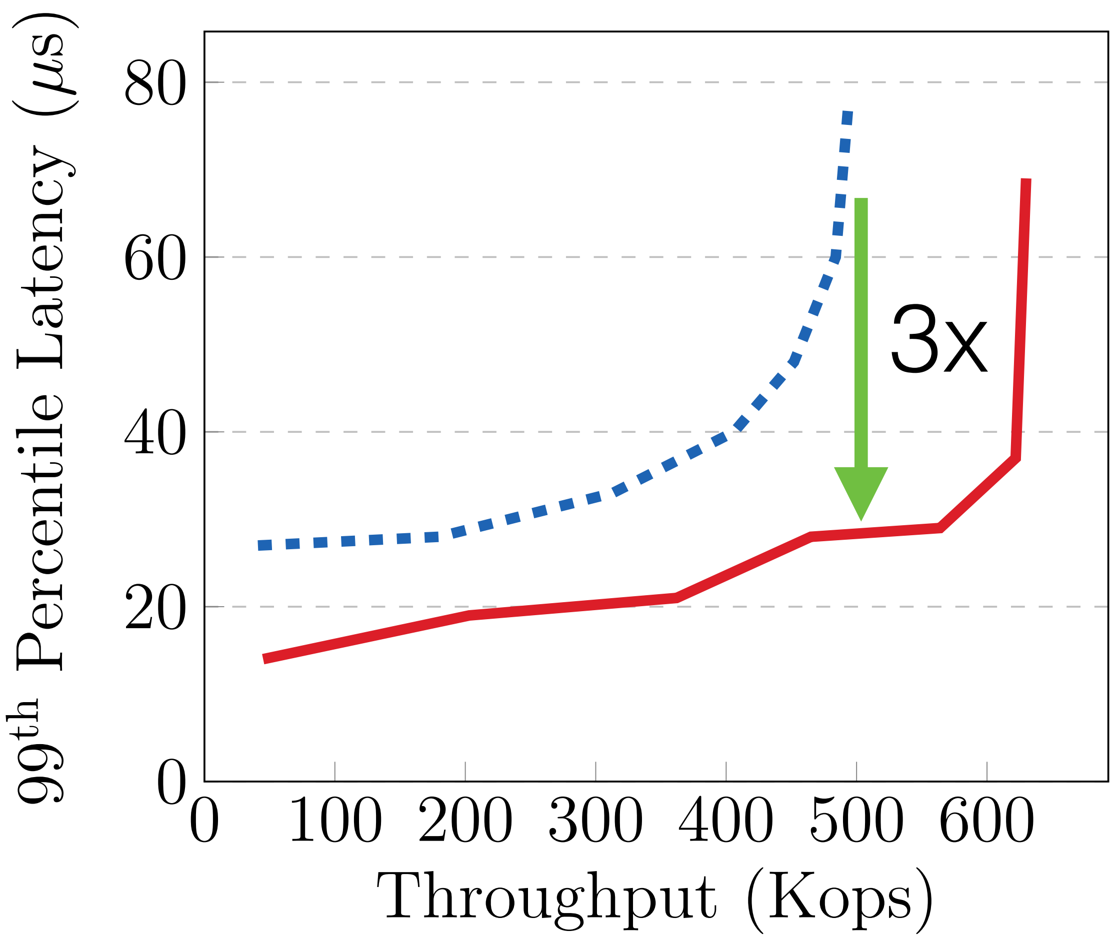
**Values are aggregated over a 4-node cluster WRITE-ONLY workload**



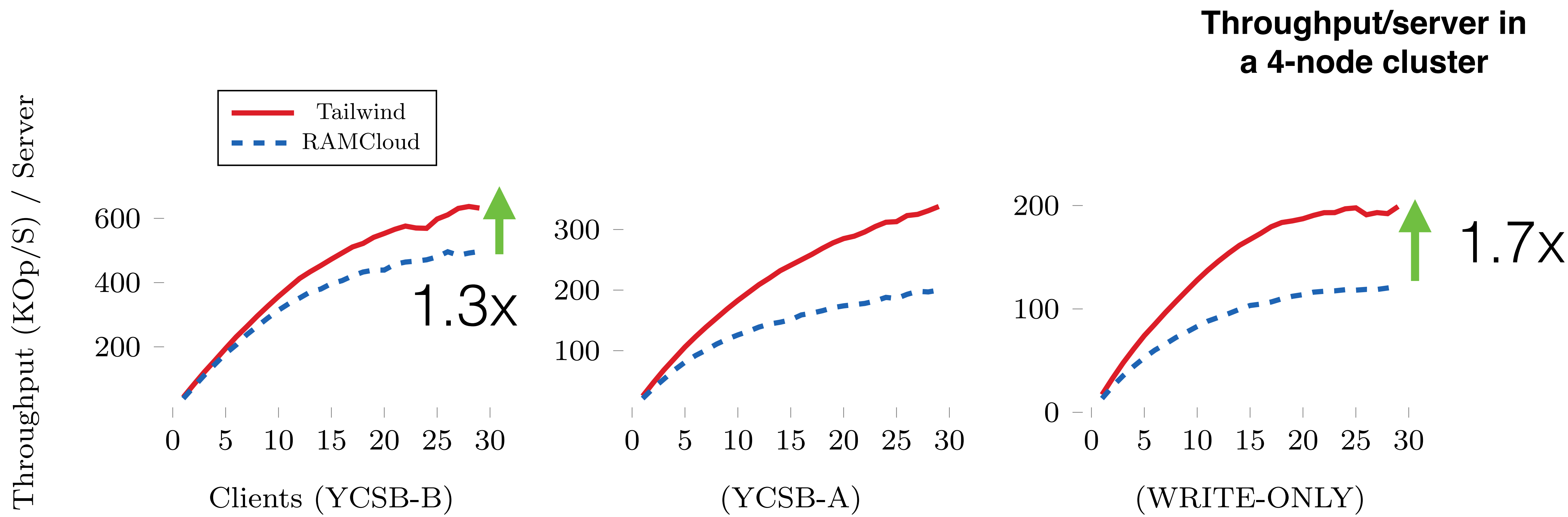



# Evaluation: Latency Improvement



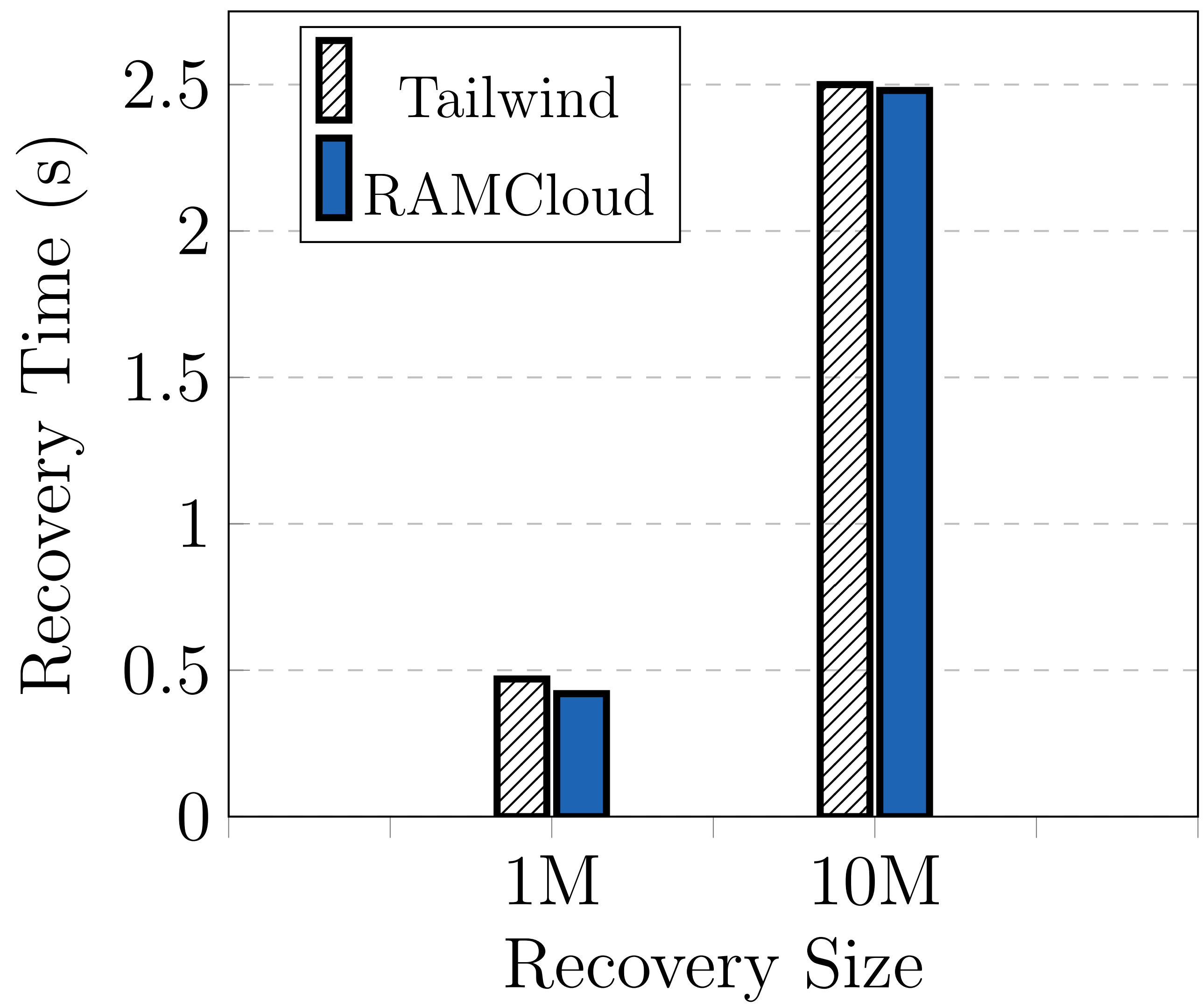



# Evaluation: Throughput




Even with 5% of PUTs,
Tailwind increase throughput by 30%

# Evaluation: Recovery Overhead




Recoveries with up to 10 million objects

# Related Work

- One-sided RDMA systems: Pilaf ATC'13, HERD SIGCOMM'14, FaRM NSDI'14, DrTM SOSP'15, DrTM + R Eurosys'16, …
- Mitigating replication overheads/Tuning consistency: RedBlue OSDI'12, Correctables OSDI'16
- Tailwind reduces replication CPU footprint and improves performance without sacrificing durability, availability, or consistency

# Conclusion

- Tailwind leverages one-sided RDMA to perform replication and leaves backups completely **idle**
- Provides backups with a protocol to protect against failure scenarios
- Reduces replication induced CPU usage while improving performance and latency
- Tailwind preserves client-facing RPC

**Thank you! Questions?**