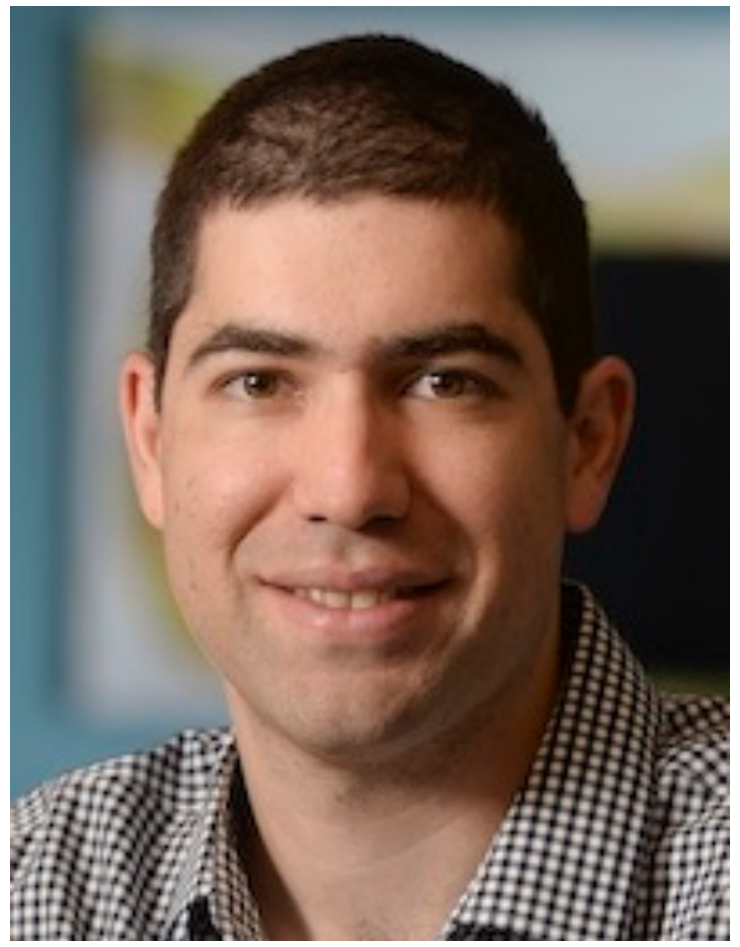
The Complexity of Memory Checking with Covert Security

Neekon Vafa (MIT)

Eurocrypt 2025
May 5, 2025

Based on joint work with:



Elette Boyle
Reichman University
& NTT Research

Ilan Komargodski
Hebrew University
& NTT Research

Remote Cloud Storage

Remote Cloud Storage

- **Your goal**: Perform computation that requires lots of storage.

Remote Cloud Storage

- **Your goal**: Perform computation that requires lots of storage.
- **Problem**: You don't have enough storage yourself (even to store the input data!)

Remote Cloud Storage

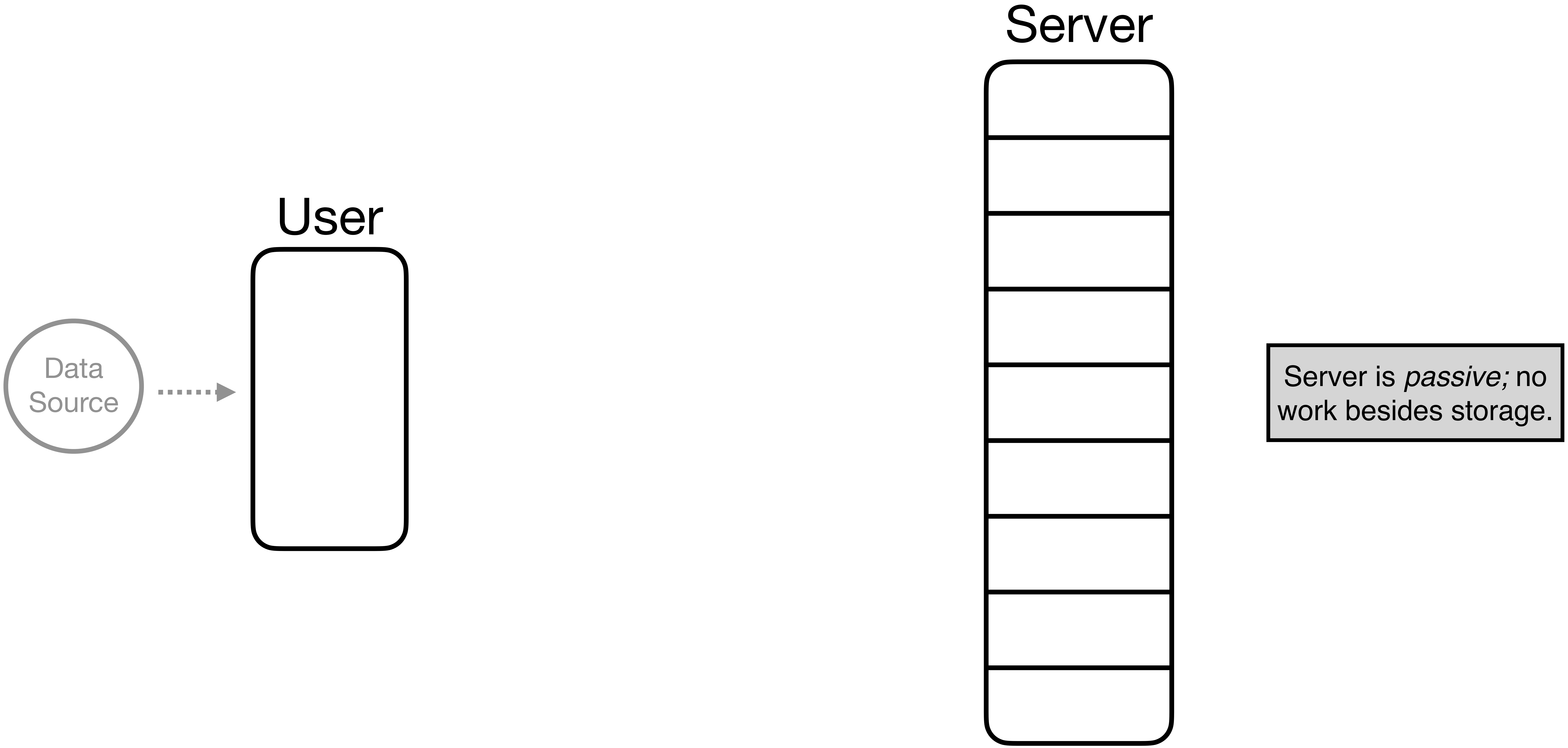
- **Your goal**: Perform computation that requires lots of storage.
- **Problem**: You don't have enough storage yourself (even to store the input data!)
  - Examples: file storage, experiment with lots of data, analytics, …

Remote Cloud Storage

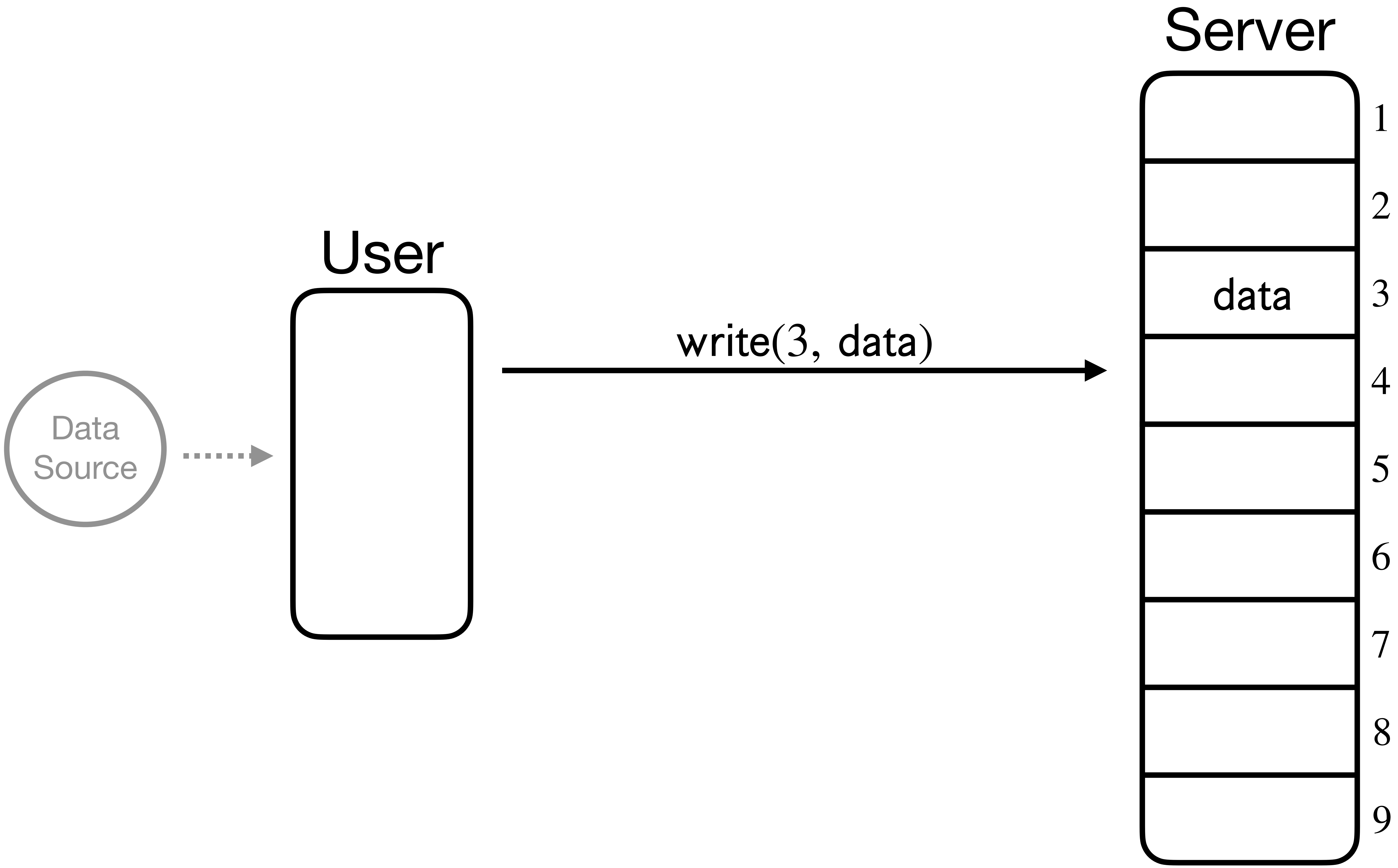
- **Your goal**: Perform computation that requires lots of storage.
- **Problem**: You don't have enough storage yourself (even to store the input data!)
    - Examples: file storage, experiment with lots of data, analytics, …
- **Common solution**: Run computation using **remote cloud** as storage.

Basic Setup

Basic Setup

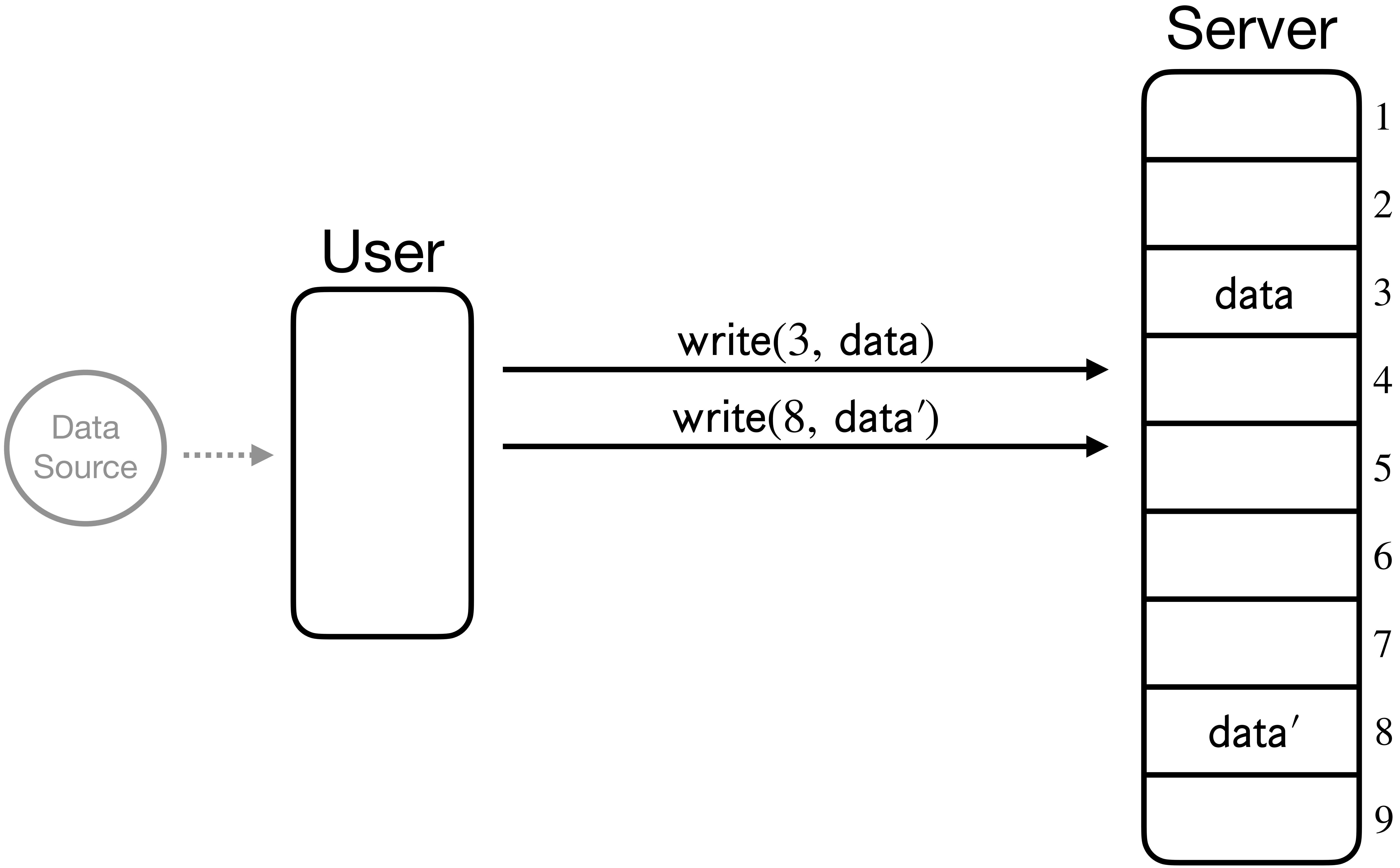

Server
User
Data
Source
Server is *passive;* no
work besides storage.

Basic Setup



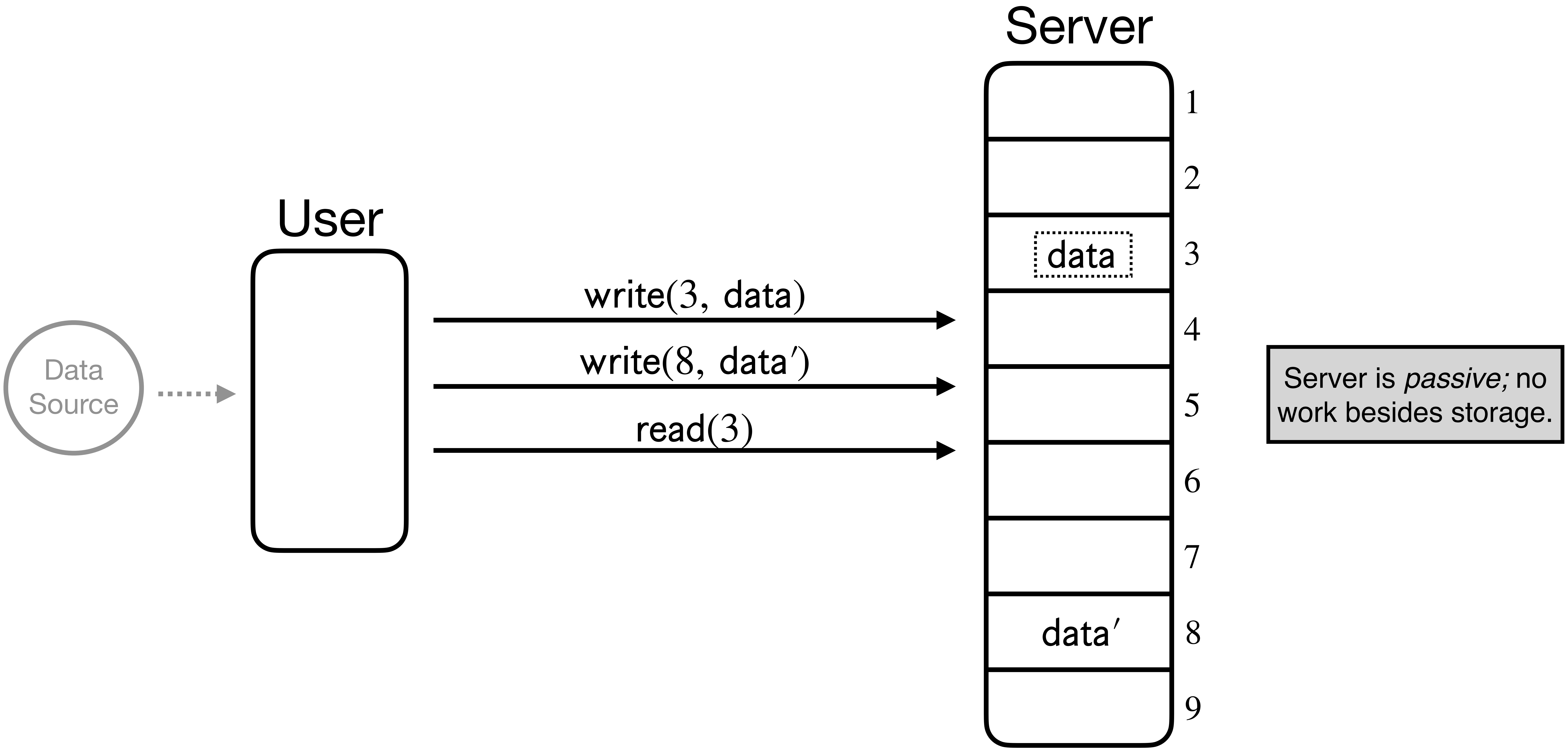
Server is *passive;* no work besides storage.

Basic Setup

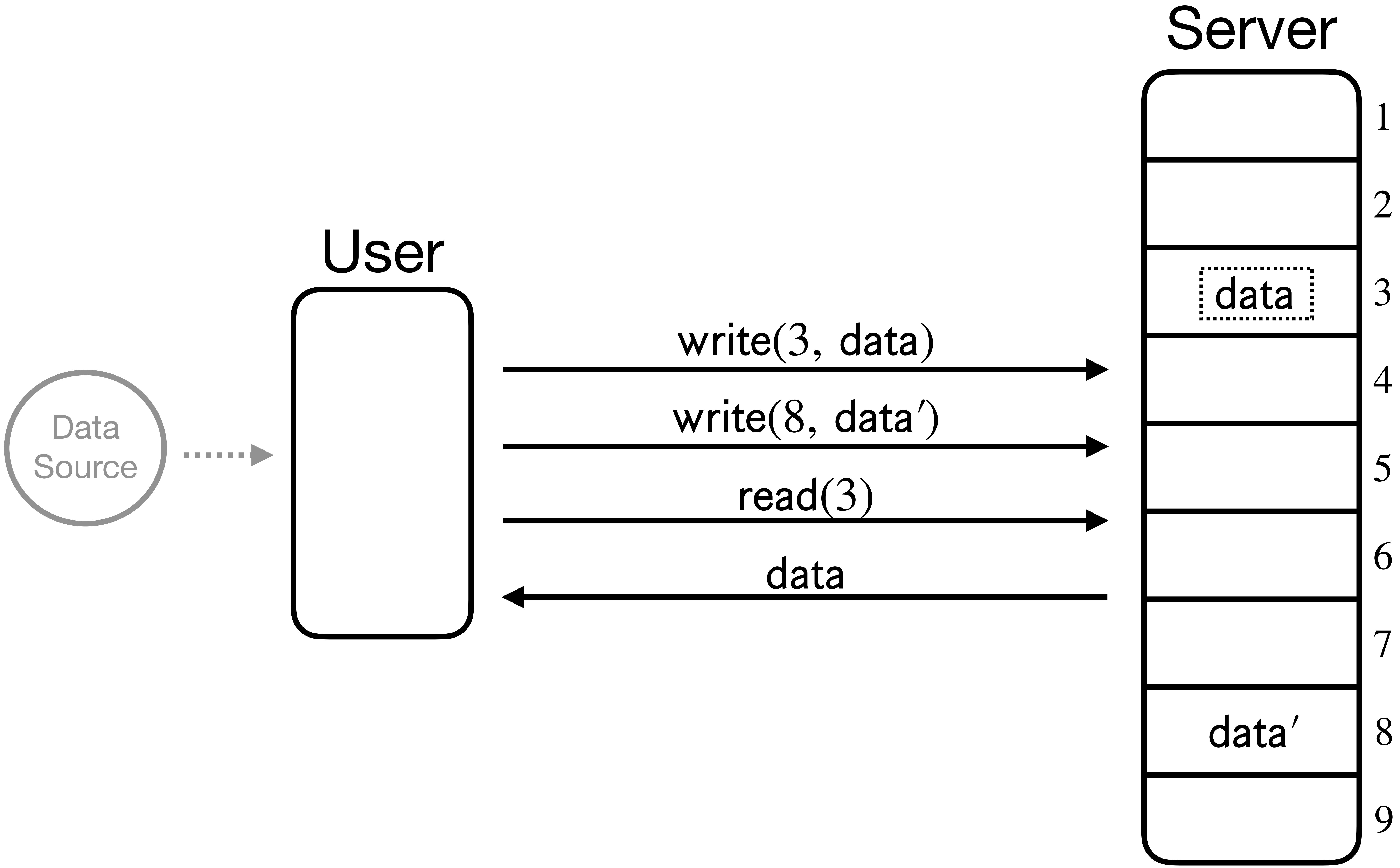


Server is *passive;* no work besides storage.

Basic Setup

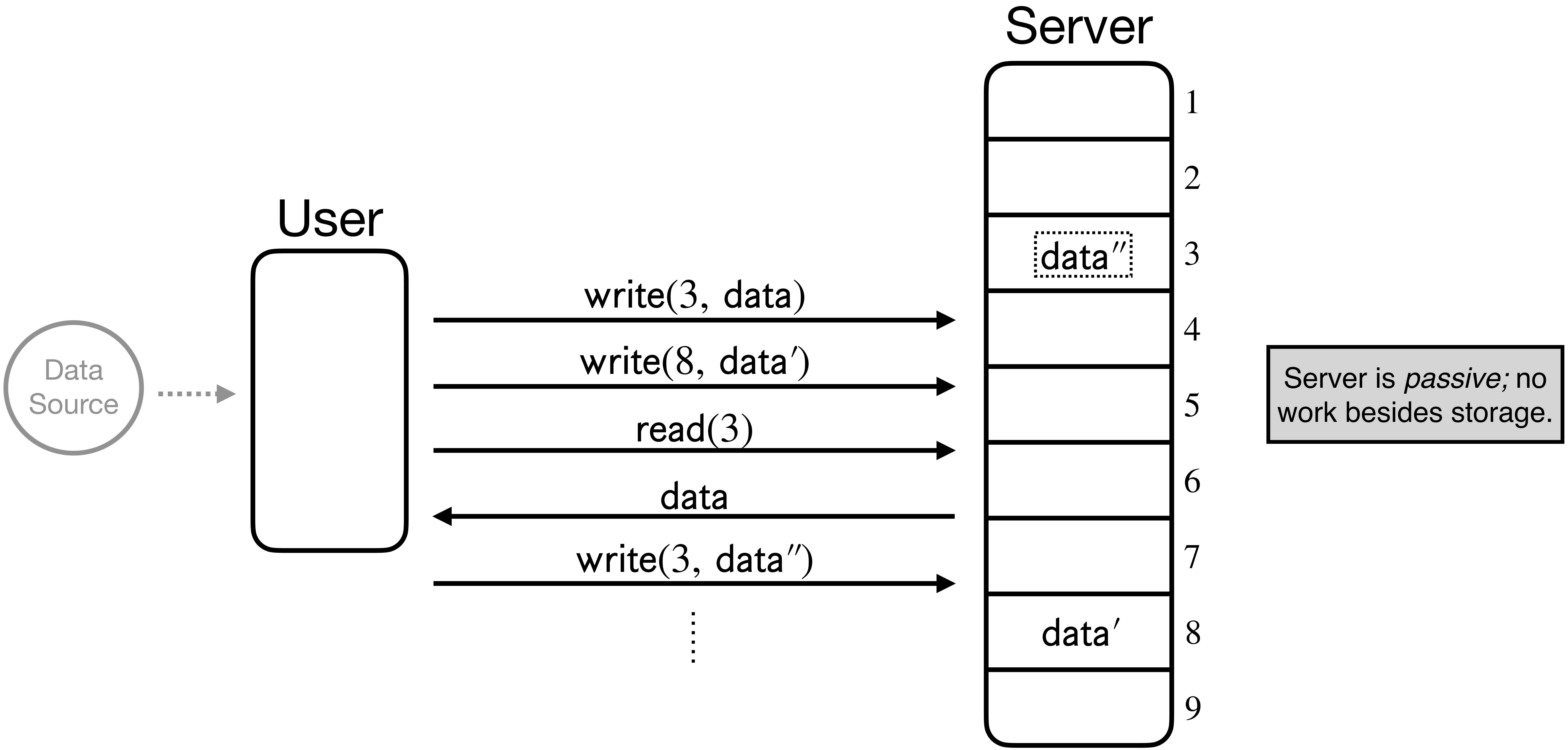

Server
User
Data Source
write(3, data)
write(8, data′)
read(3)
data
data′
1
2
3
4
5
6
7
8
9
Server is *passive;* no work besides storage.

Basic Setup



Server is *passive;* no work besides storage.

Basic Setup


Server
User
Data
Source
write(3, data)
write(8, data′)
read(3)
data
write(3, data″)
data″
data′
1
2
3
4
5
6
7
8
9
Server is *passive;* no
work besides storage.

Trust Concerns

Trust Concerns

- **Privacy #1**: The server may see your data!

Trust Concerns

- **Privacy #1**: The server may see your data!

Solution: Secret-key encryption (or secret sharing)

Trust Concerns

- **Privacy #1**: The server may see your data!

Solution: Secret-key encryption (or secret sharing)

- **Privacy #2**: The server can see *where* you're accessing!

Trust Concerns

- **Privacy #1**: The server may see your data!

Solution: Secret-key encryption (or secret sharing)

- **Privacy #2**: The server can see *where* you're accessing!

Solution: Oblivious RAM (ORAM) [Goldreich, Ostrovsky '89, '90, '96]

Trust Concerns

- **Privacy #1**: The server may see your data!

Solution: Secret-key encryption (or secret sharing)

- **Privacy #2**: The server can see *where* you're accessing!

Solution: Oblivious RAM (ORAM) [Goldreich, Ostrovsky '89, '90, '96]

- **Integrity**: An active, malicious server may modify your data!

Trust Concerns

- **Privacy #1**: The server may see your data!

  Solution: Secret-key encryption (or secret sharing)

- **Privacy #2**: The server can see *where* you're accessing!

  Solution: Oblivious RAM (ORAM) [Goldreich, Ostrovsky '89, '90, '96]

- **Integrity**: An active, malicious server may modify your data!

  Ideally: Verify that the server is behaving honestly

Trust Concerns

- **Privacy #1**: The server may see your data!

  Solution: Secret-key encryption (or secret sharing)

- **Privacy #2**: The server can see *where* you're accessing!

  Solution: Oblivious RAM (ORAM) [Goldreich, Ostrovsky '89, '90, '96]

- **Integrity**: An active, malicious server may modify your data!

  $\boxed{\text{Ideally: Verify that the server is behaving honestly}}$

- (**Privacy + Integrity**: All simultaneously!)

Integrity: Verifying Honest Server Behavior

Integrity: Verifying Honest Server Behavior

- Can we prevent adversary from…

Integrity: Verifying Honest Server Behavior

- Can we prevent adversary from…
  - …modifying data?

Integrity: Verifying Honest Server Behavior

- Can we prevent adversary from…
  - …modifying data? **No!**

Integrity: Verifying Honest Server Behavior

- Can we prevent adversary from…
  - …modifying data? **No!**
  - …*undetectably* modifying data?

Integrity: Verifying Honest Server Behavior

- Can we prevent adversary from…
  - …modifying data? **No!**
  - …*undetectably* modifying data? **Yes!**

Integrity: Verifying Honest Server Behavior

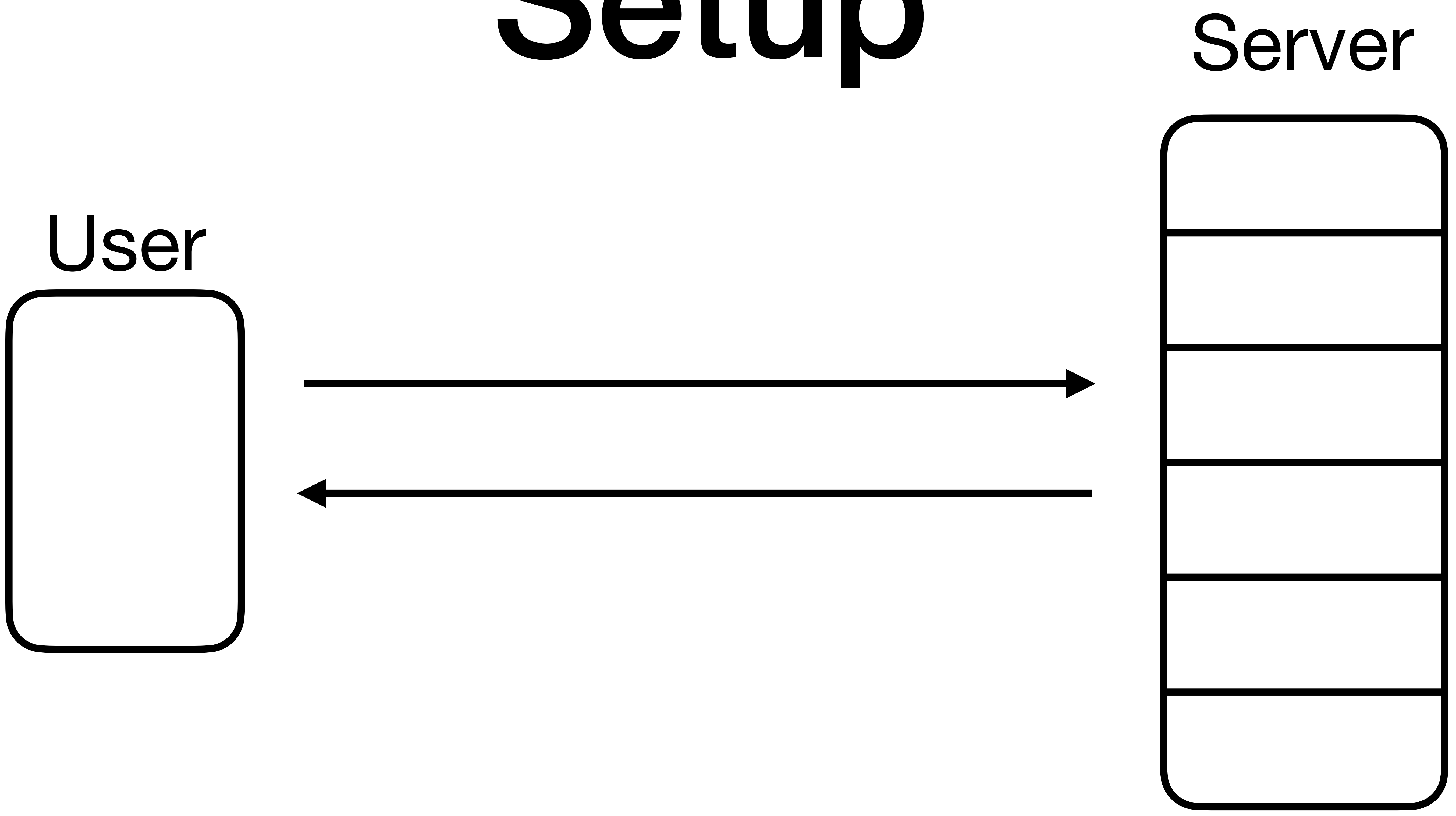
- Can we prevent adversary from…
  - …modifying data? **No!**
  - …*undetectably* modifying data? **Yes!**
- Name for this: **memory checker**

Memory Checking

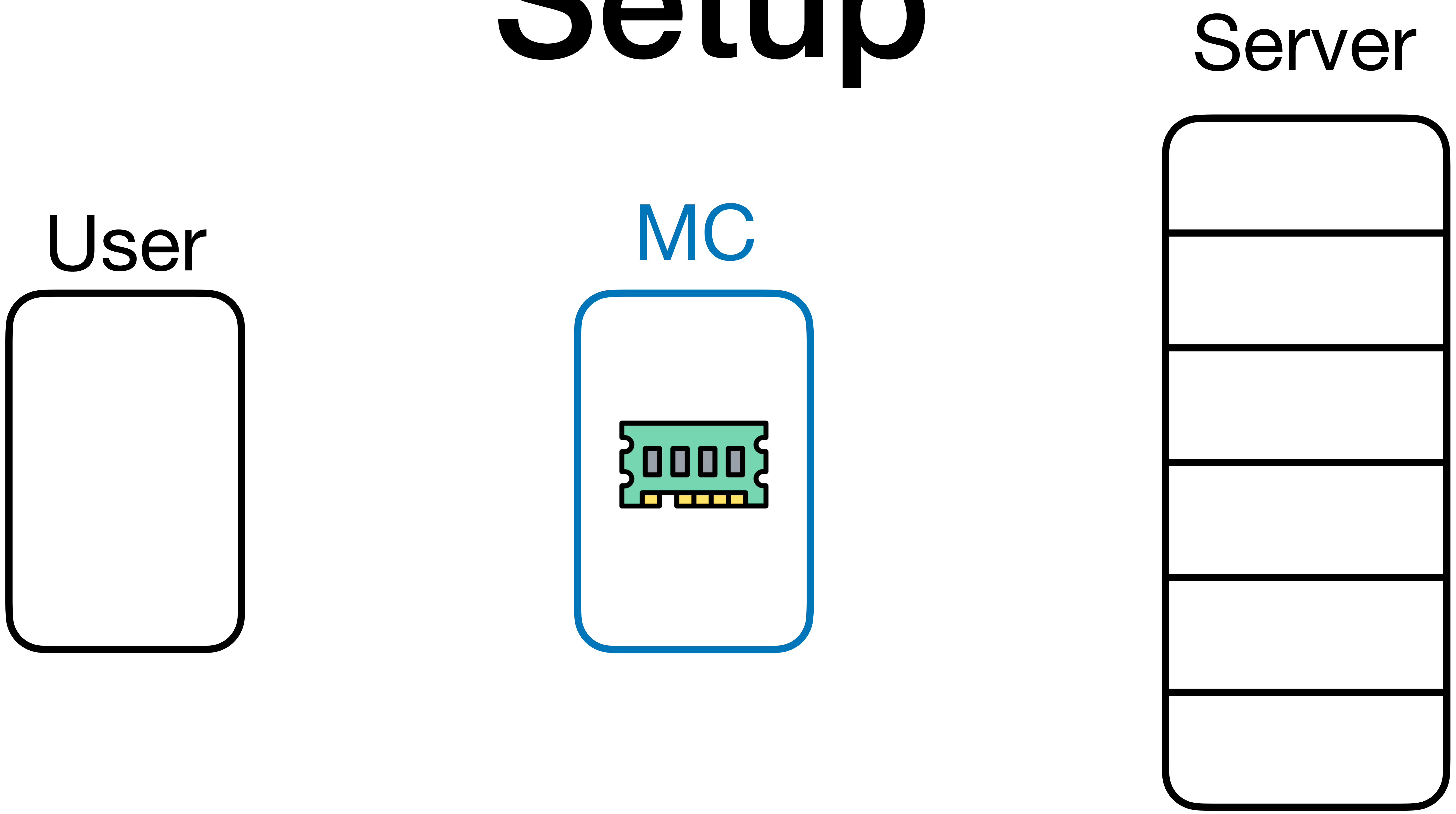
A **memory checker** (MC) is a protocol that prevents adversaries from **undetectably** modifying cloud data.

[FOCS '91, Blum, Evans, Gemmell, Kannan, Naor]

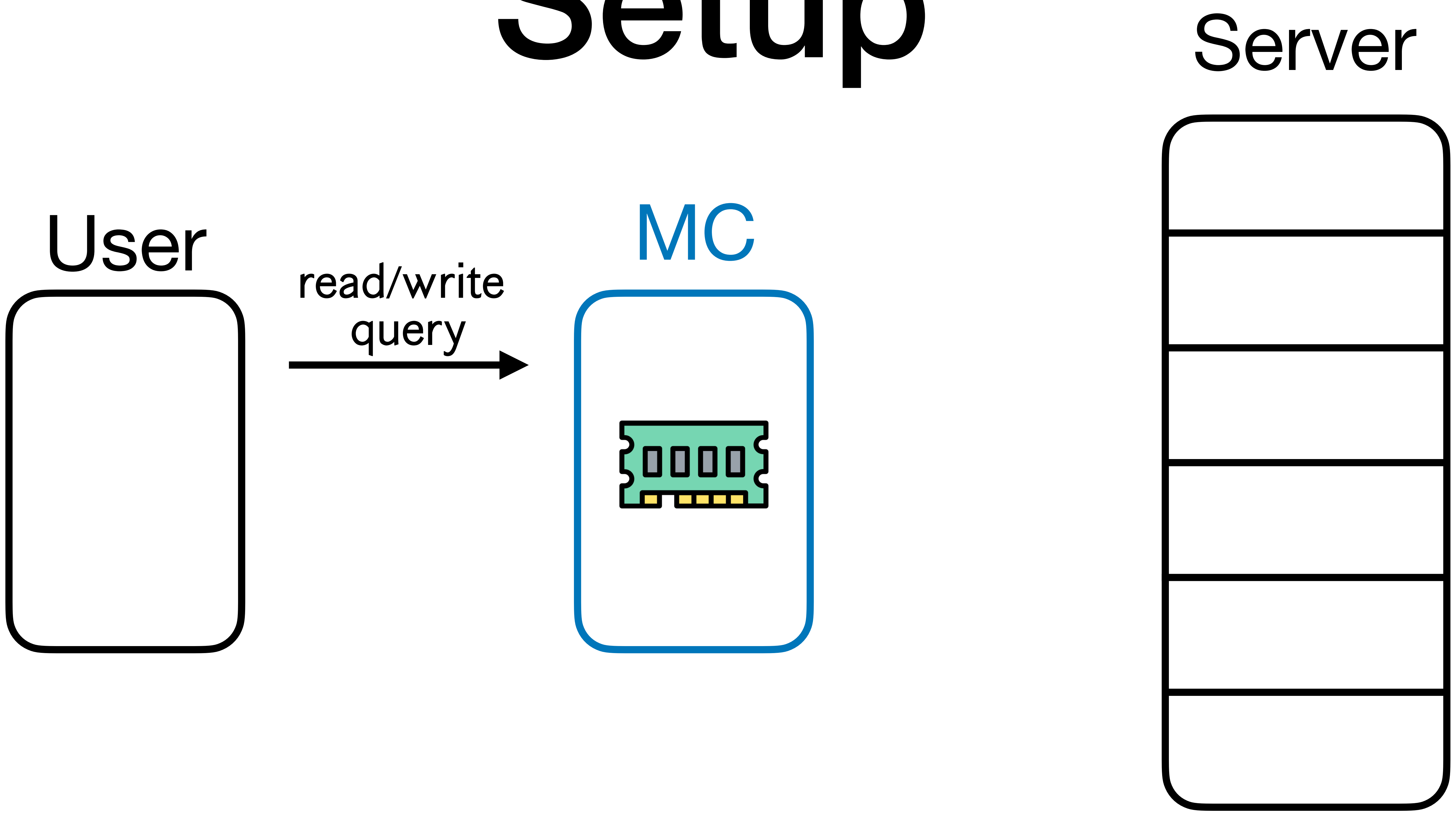
Setup



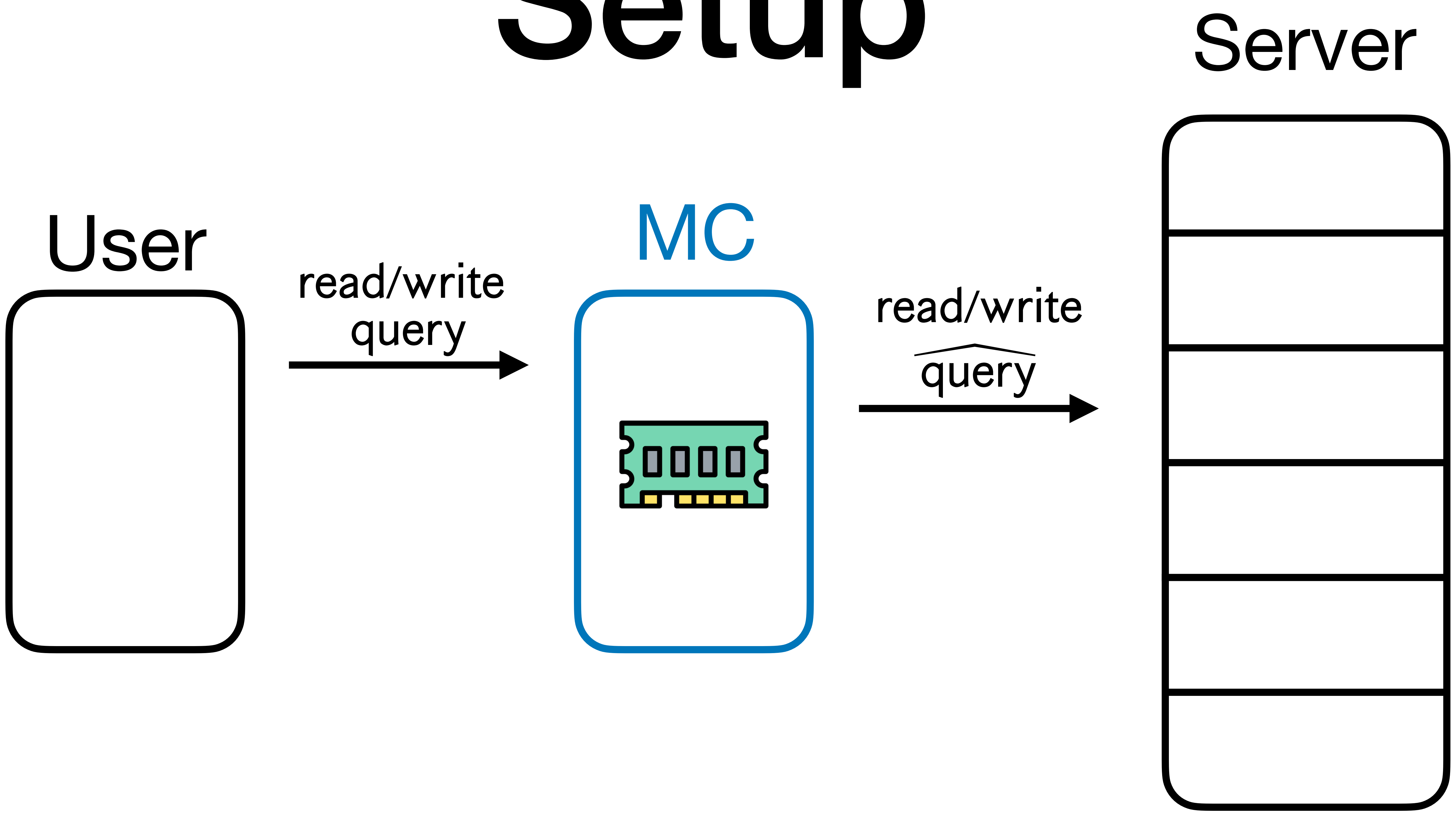
Setup



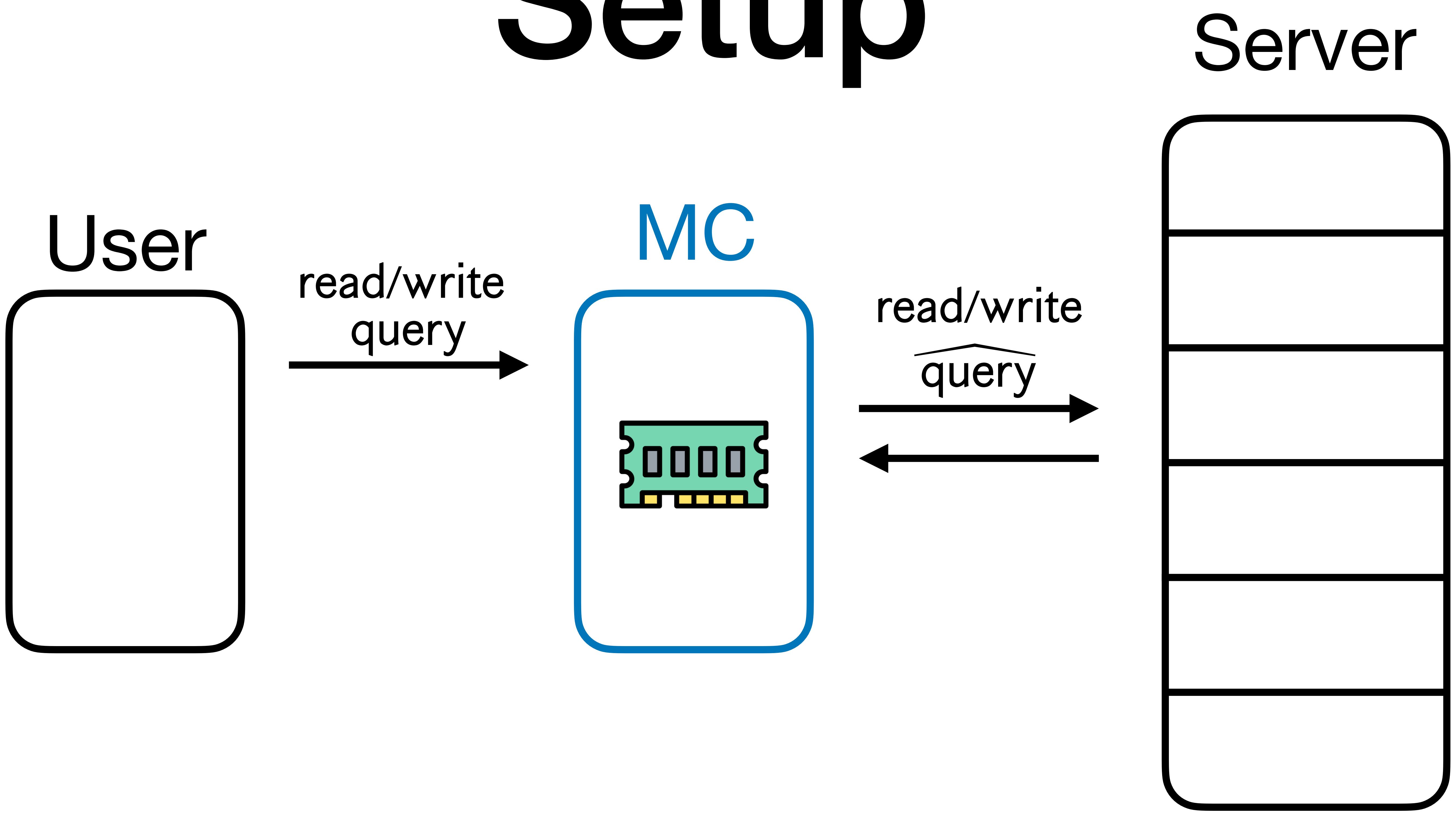
Setup



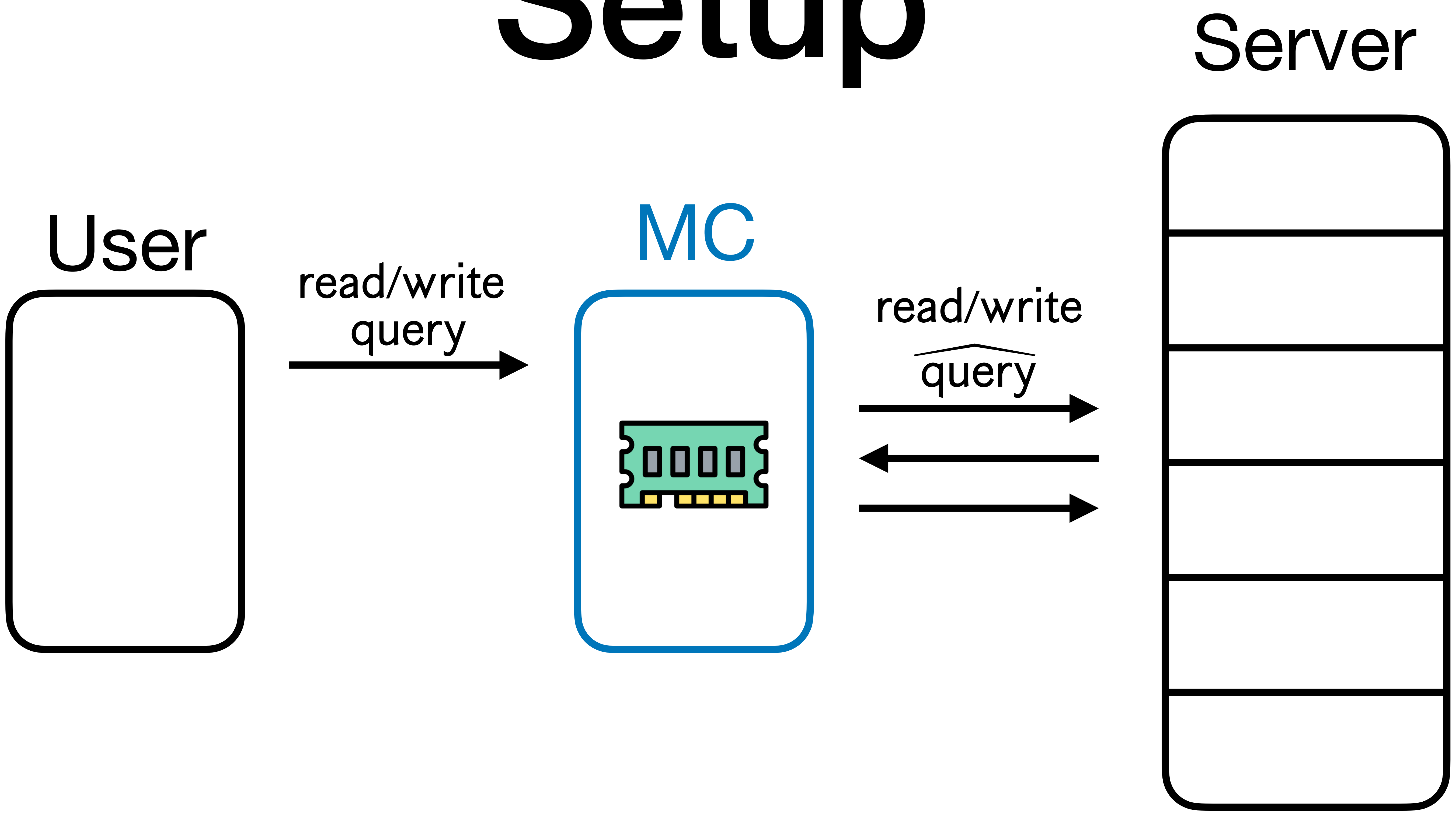
Setup


Server
User
read/write
query
MC
read/write
$\widehat{\text{query}}$

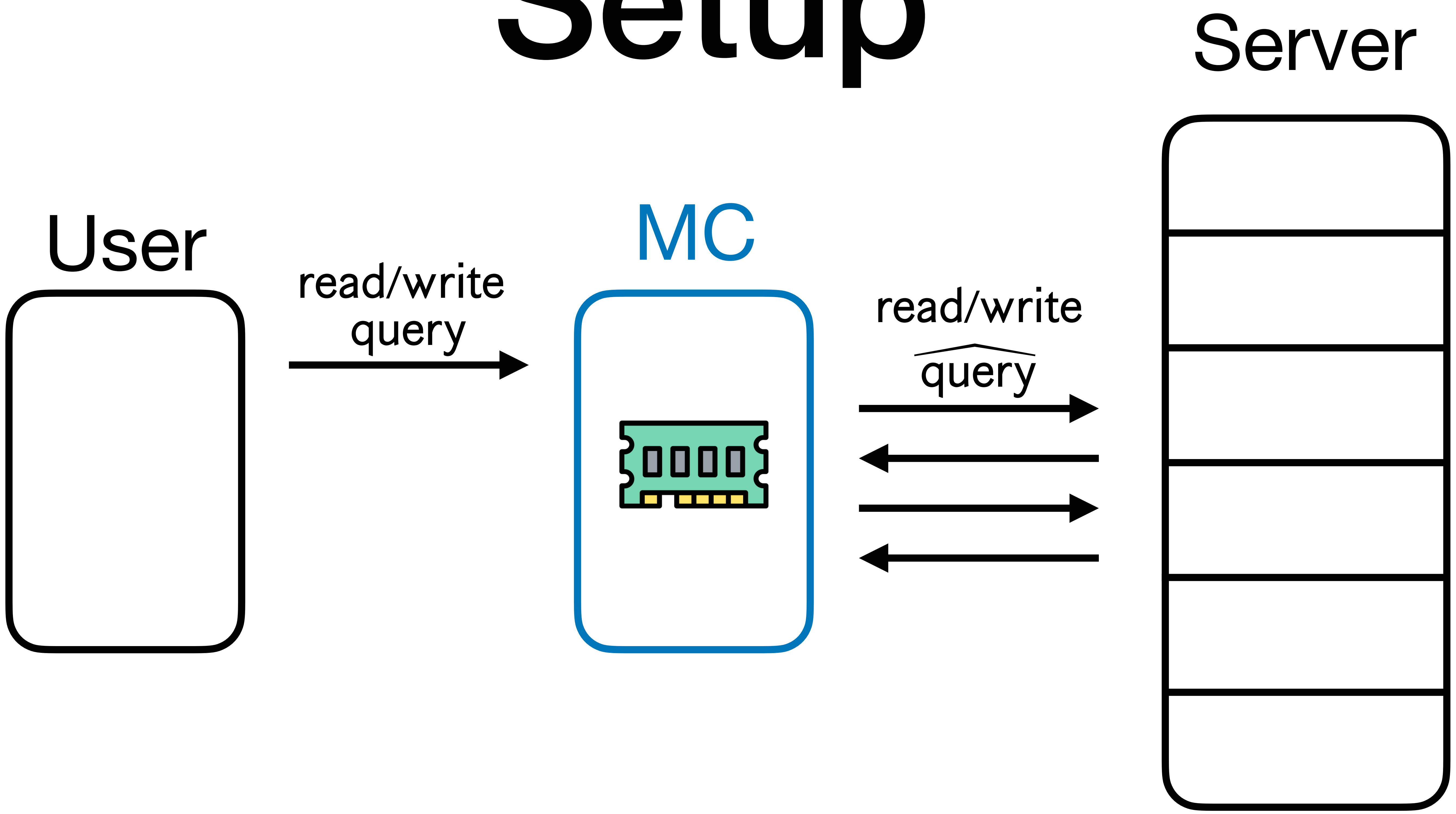
Setup


User
read/write
query
MC
read/write
$\widehat{\text{query}}$
Server

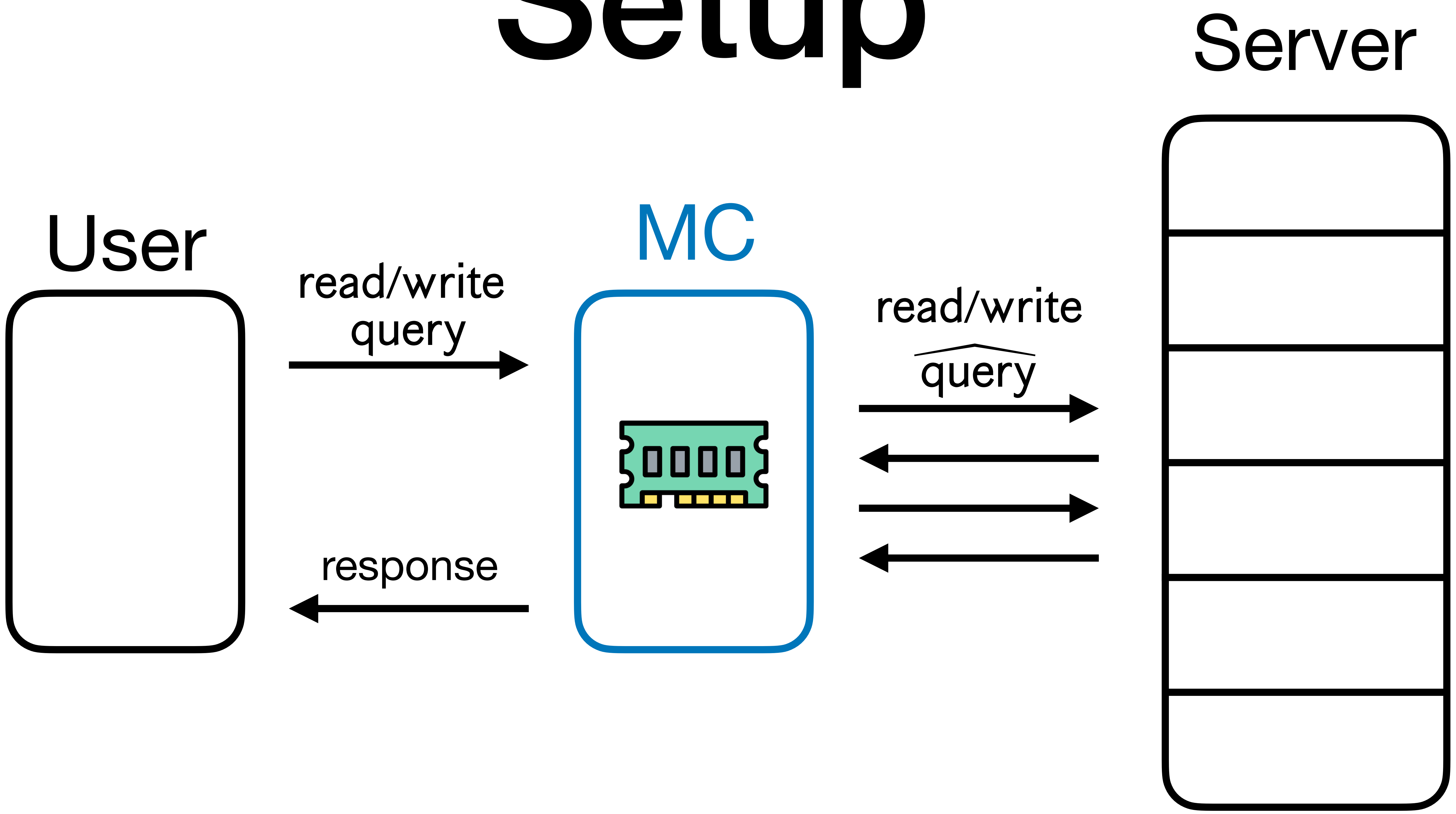
Setup



Setup

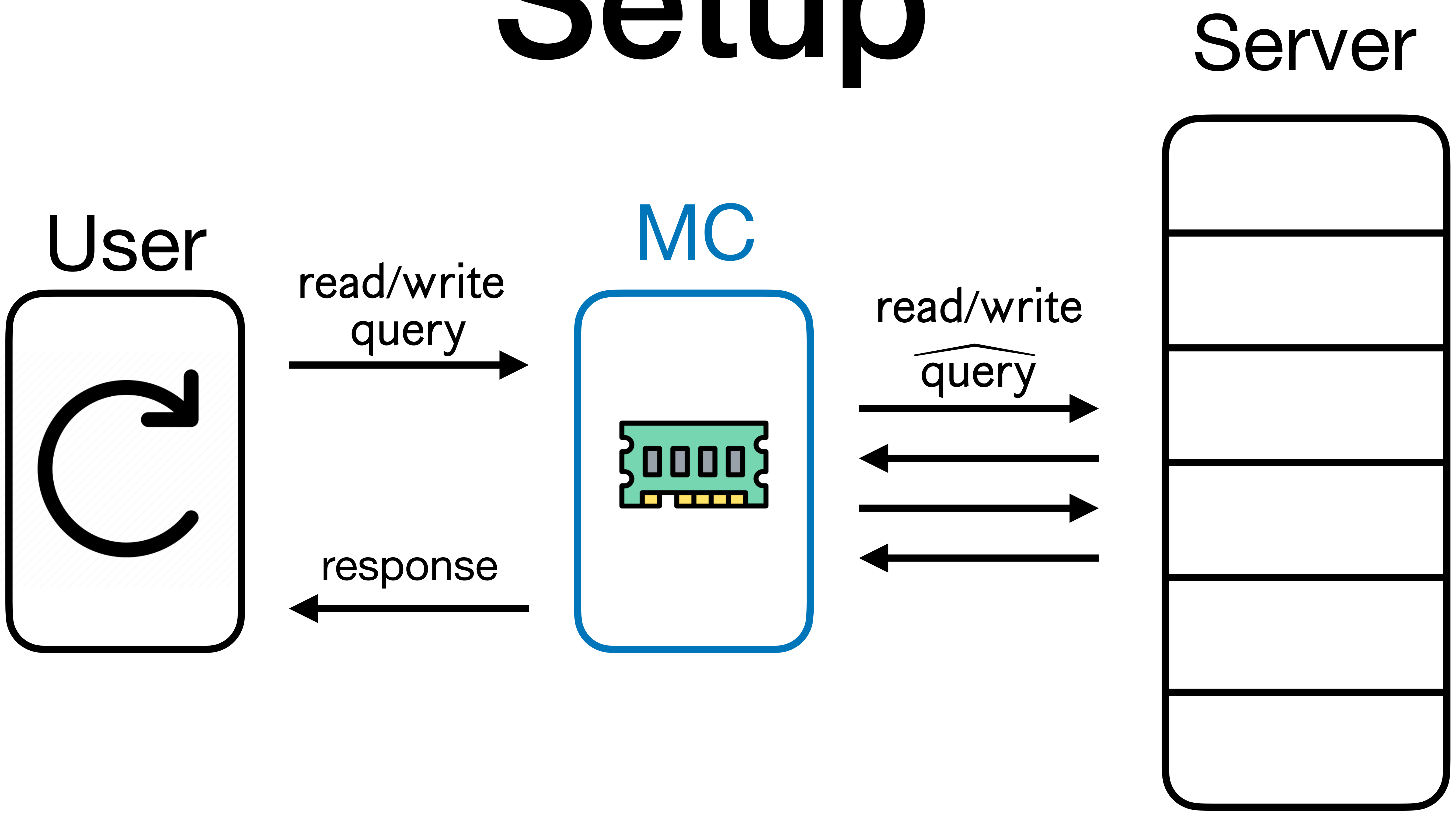


Setup

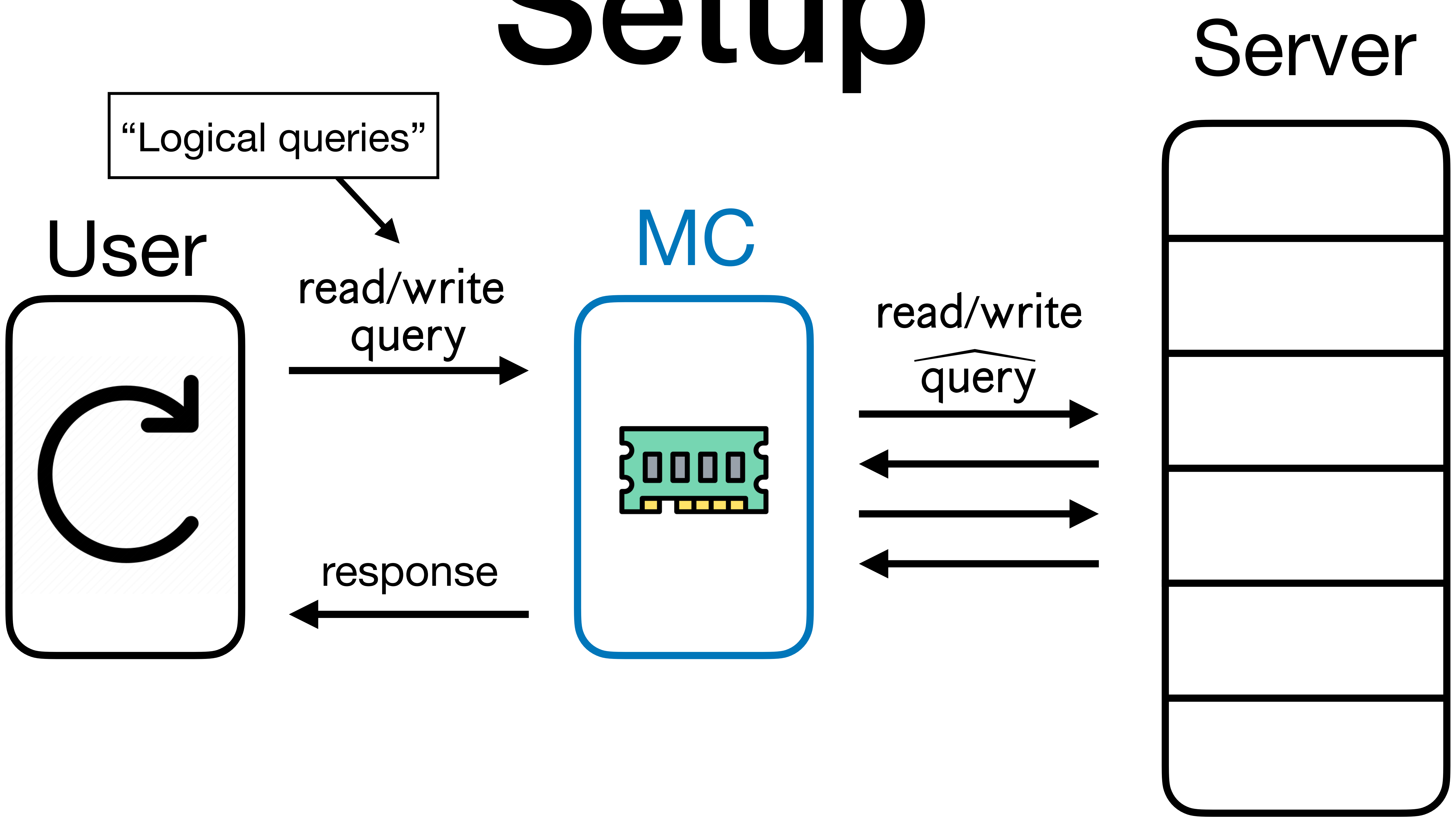


Server
User
read/write
query
MC
read/write
$\widehat{\text{query}}$
response

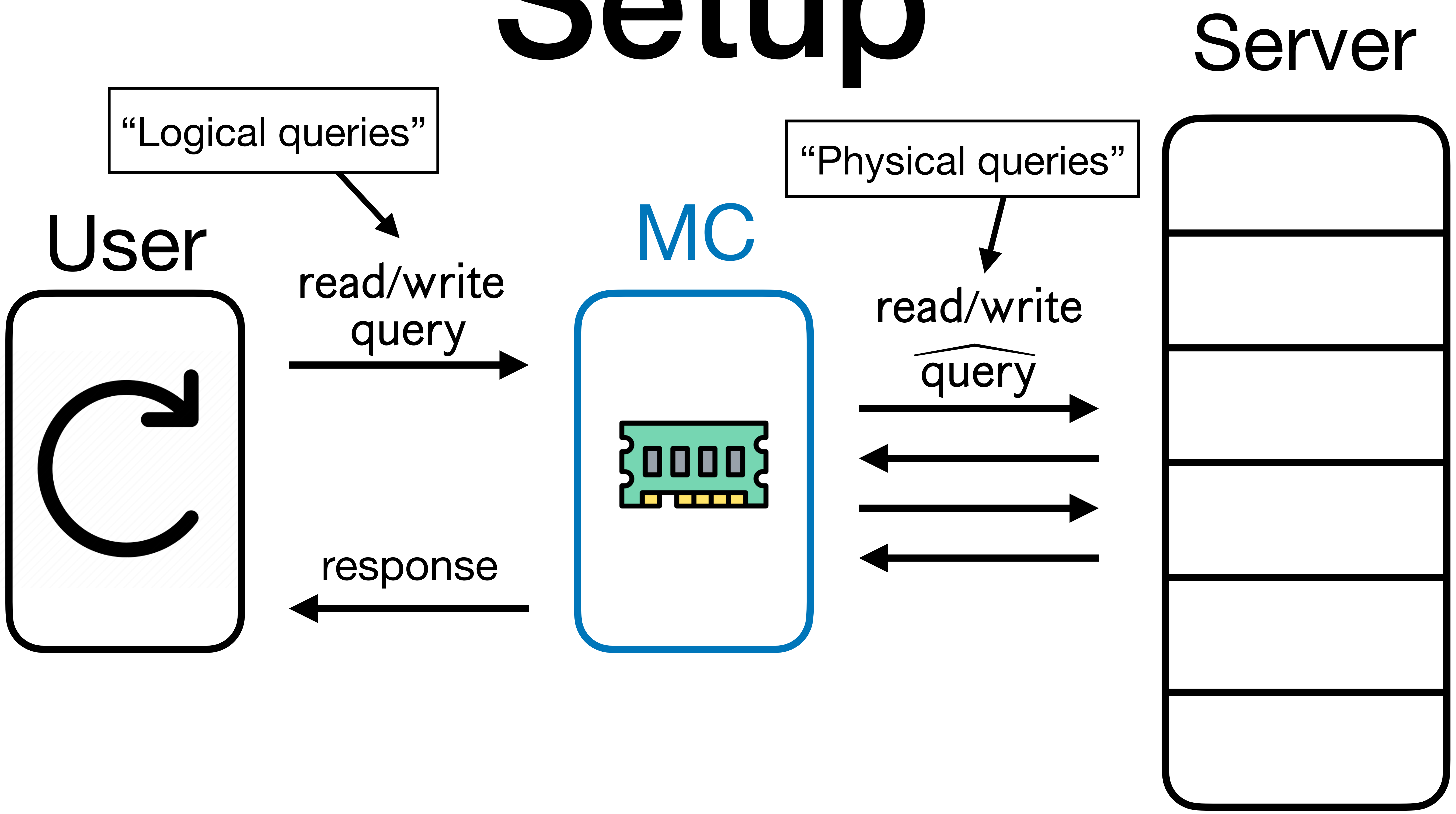
Setup



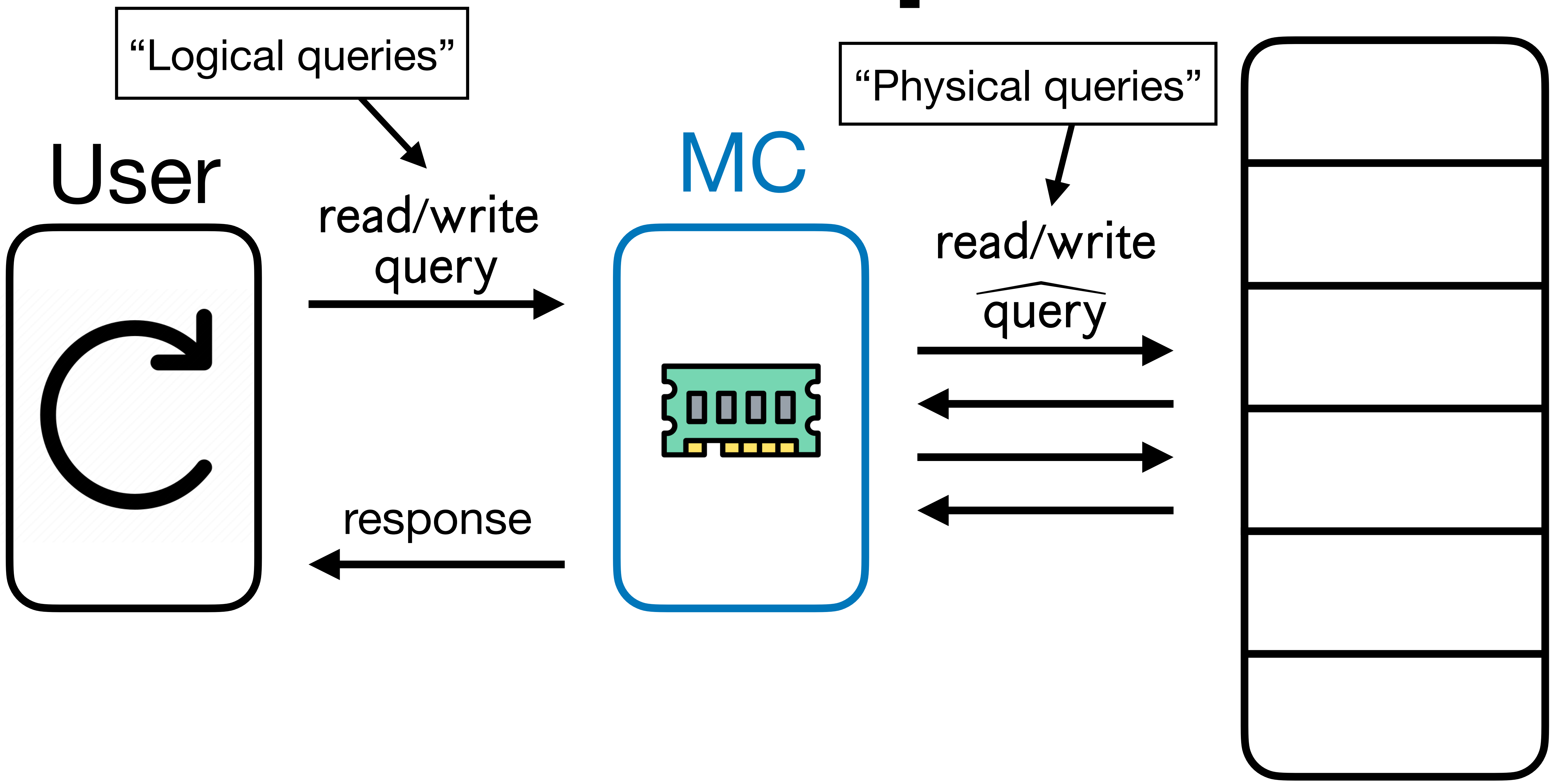
Setup


Server
“Logical queries”
User
read/write
query
MC
read/write
query
response

Setup

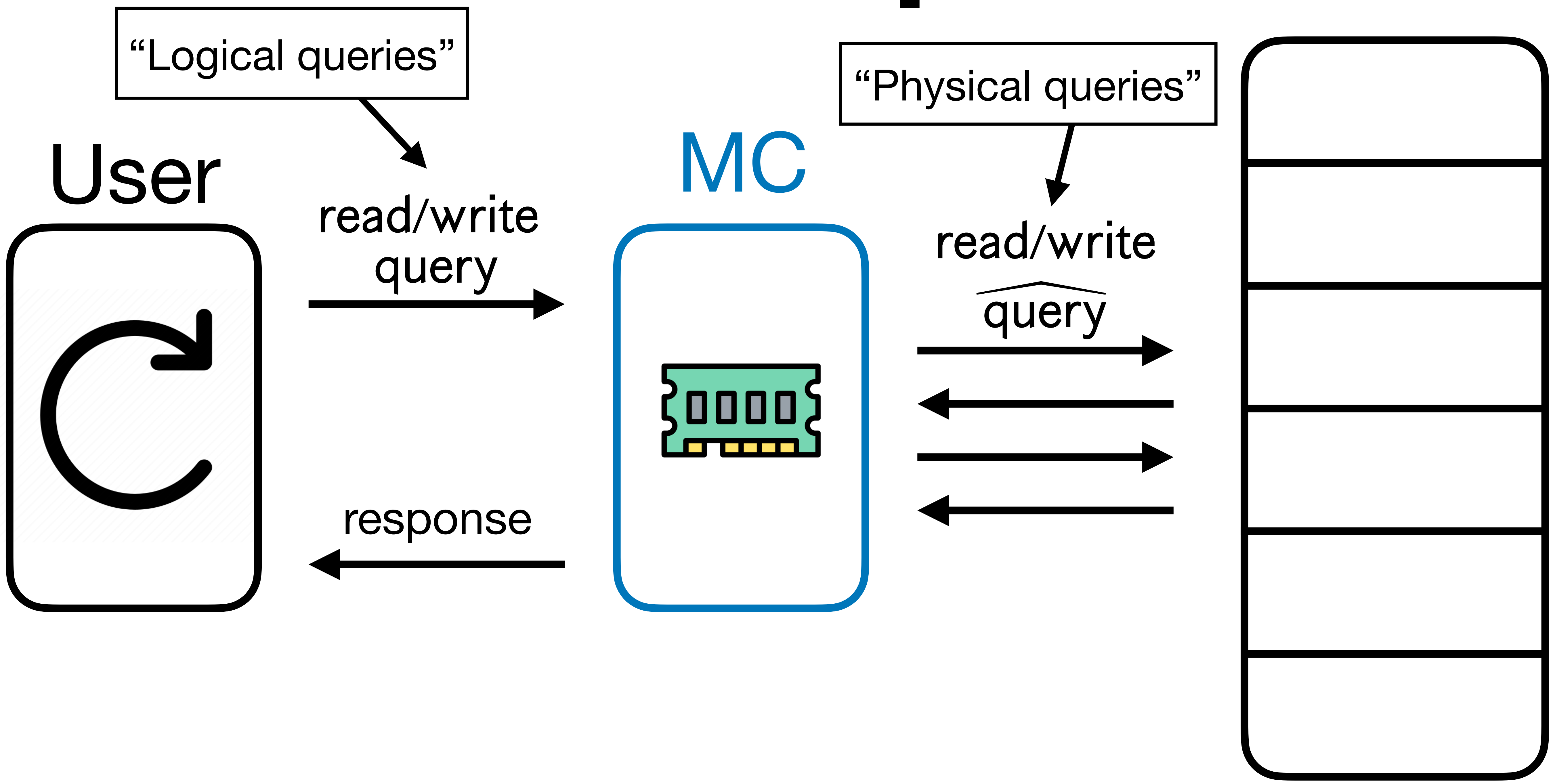

Server
"Logical queries"
"Physical queries"
User
read/write query
MC
read/write
$\widehat{\text{query}}$
response

Setup



"Read-Only Reads" Assumption:
No read query ever invokes a write $\widehat{\text{query}}$.

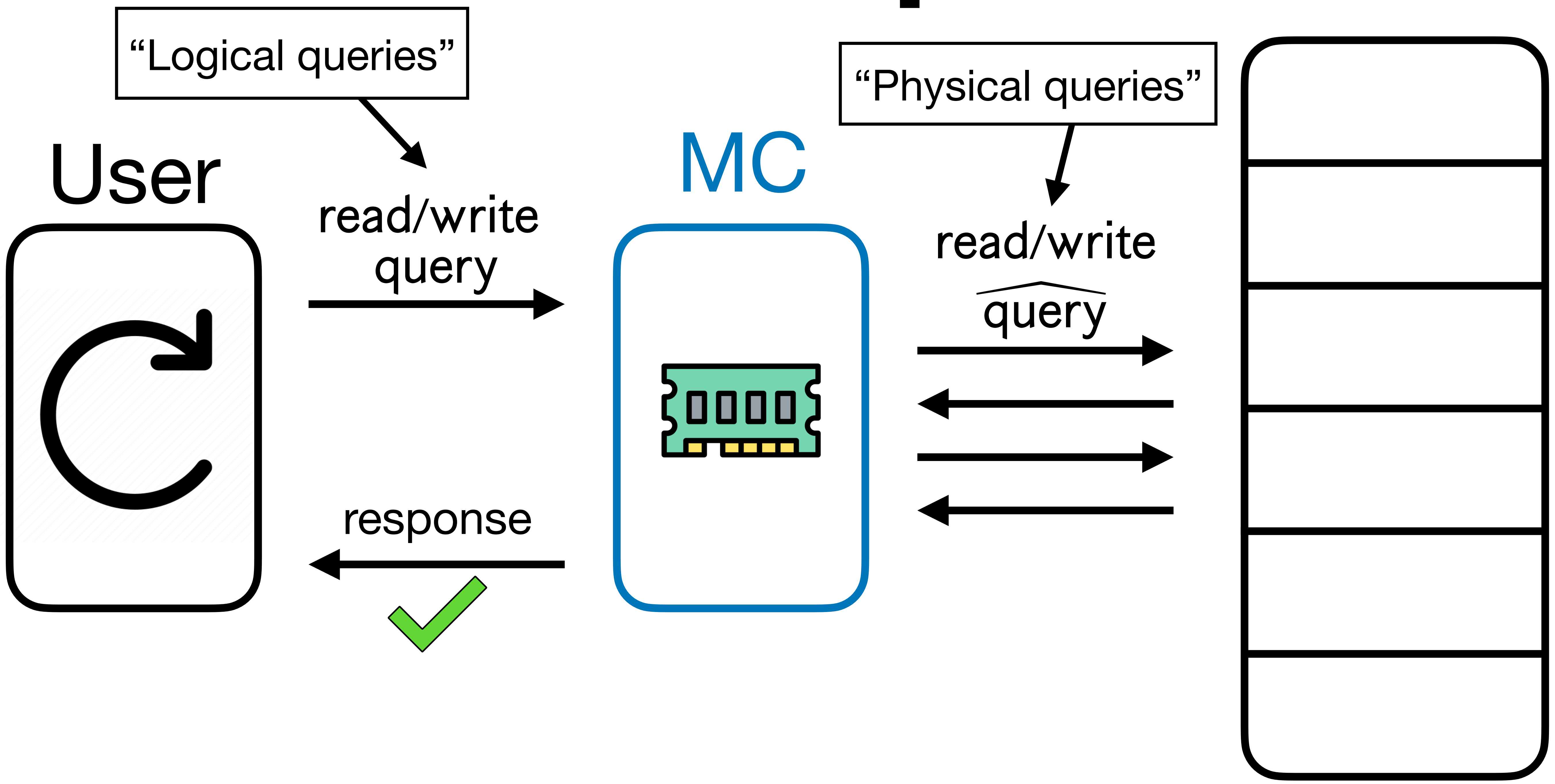
Setup



"Read-Only Reads" Assumption: No read query ever invokes a write $\widehat{\text{query}}$.

- **Completeness**: If the server behaved honestly (i.e., $\widehat{\text{query}}$ are all correct), then MC gives correct response.

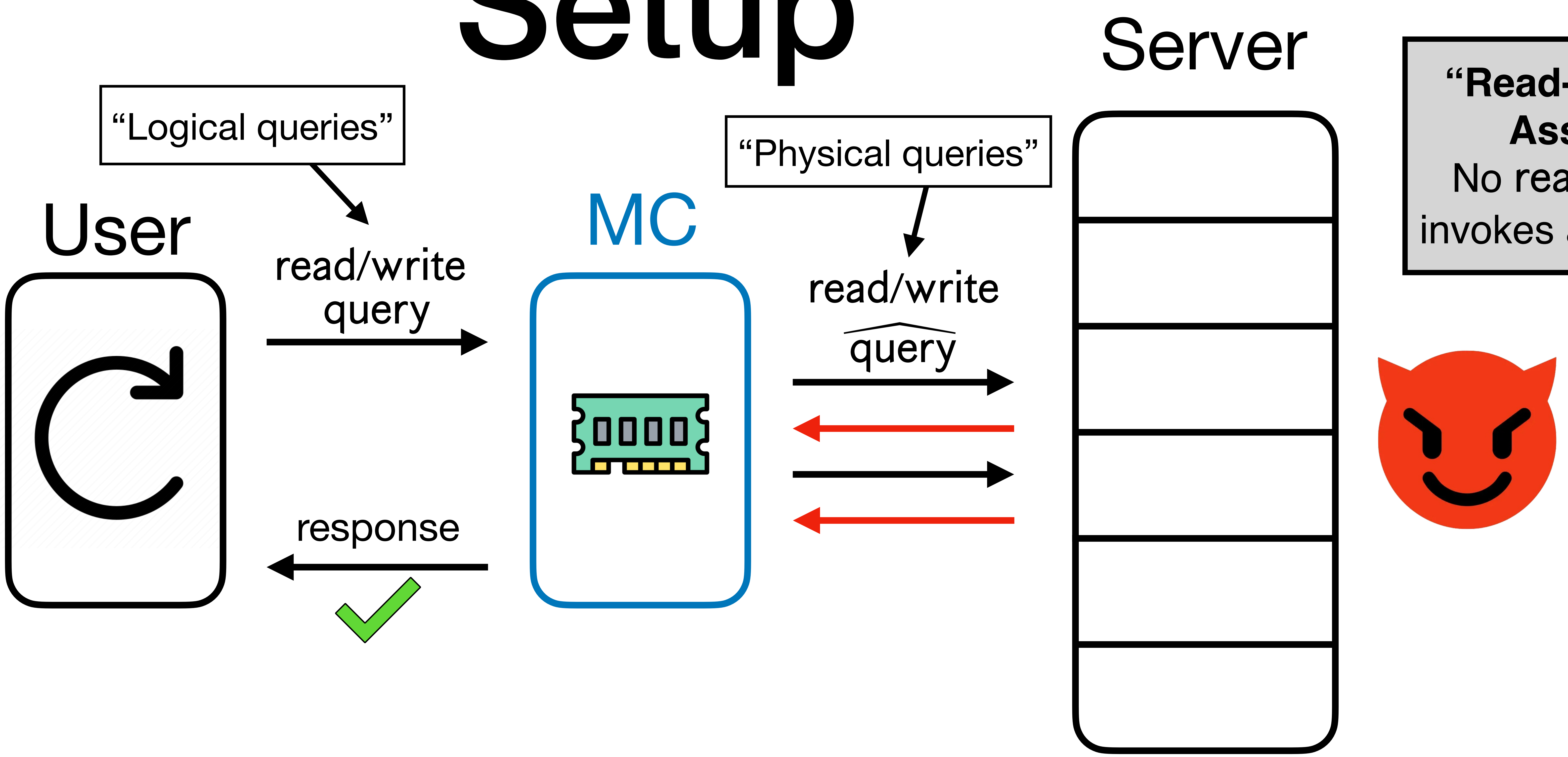
Setup



“Read-Only Reads” Assumption:
No read query ever invokes a write $\widehat{\text{query}}$.

- **Completeness**: If the server behaved honestly (i.e., $\widehat{\text{query}}$ are all correct), then MC gives correct response.

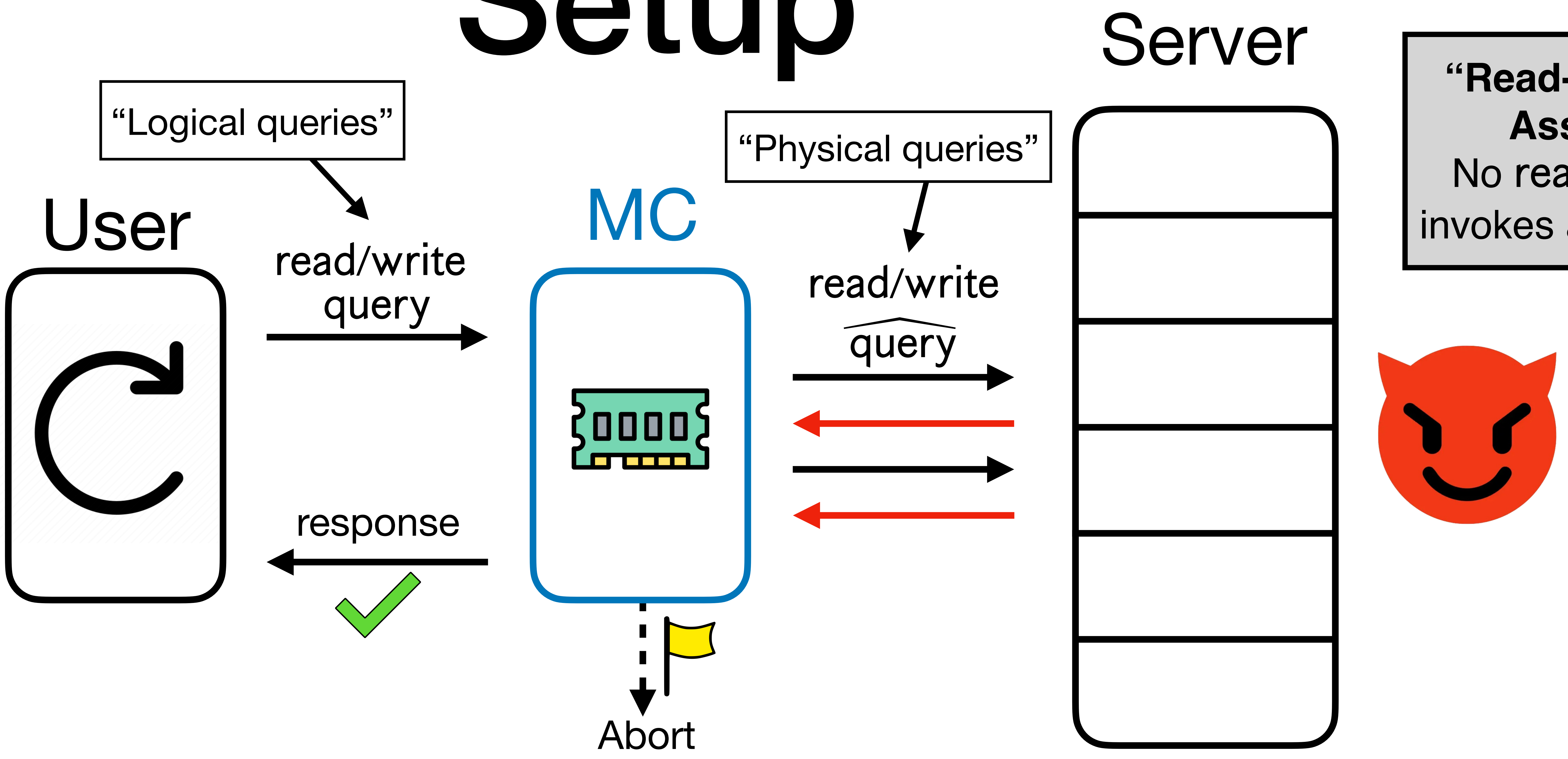
Setup



"Read-Only Reads" Assumption: No read query ever invokes a write $\widehat{\text{query}}$.

- **Completeness**: If the server behaved honestly (i.e., $\widehat{\text{query}}$ are all correct), then MC gives correct response.
- **Soundness**: For any PPT malicious server and any sequence of user queries, the probability that the MC gives an incorrect response without **aborting** is at most p, where p is negligible.

Setup

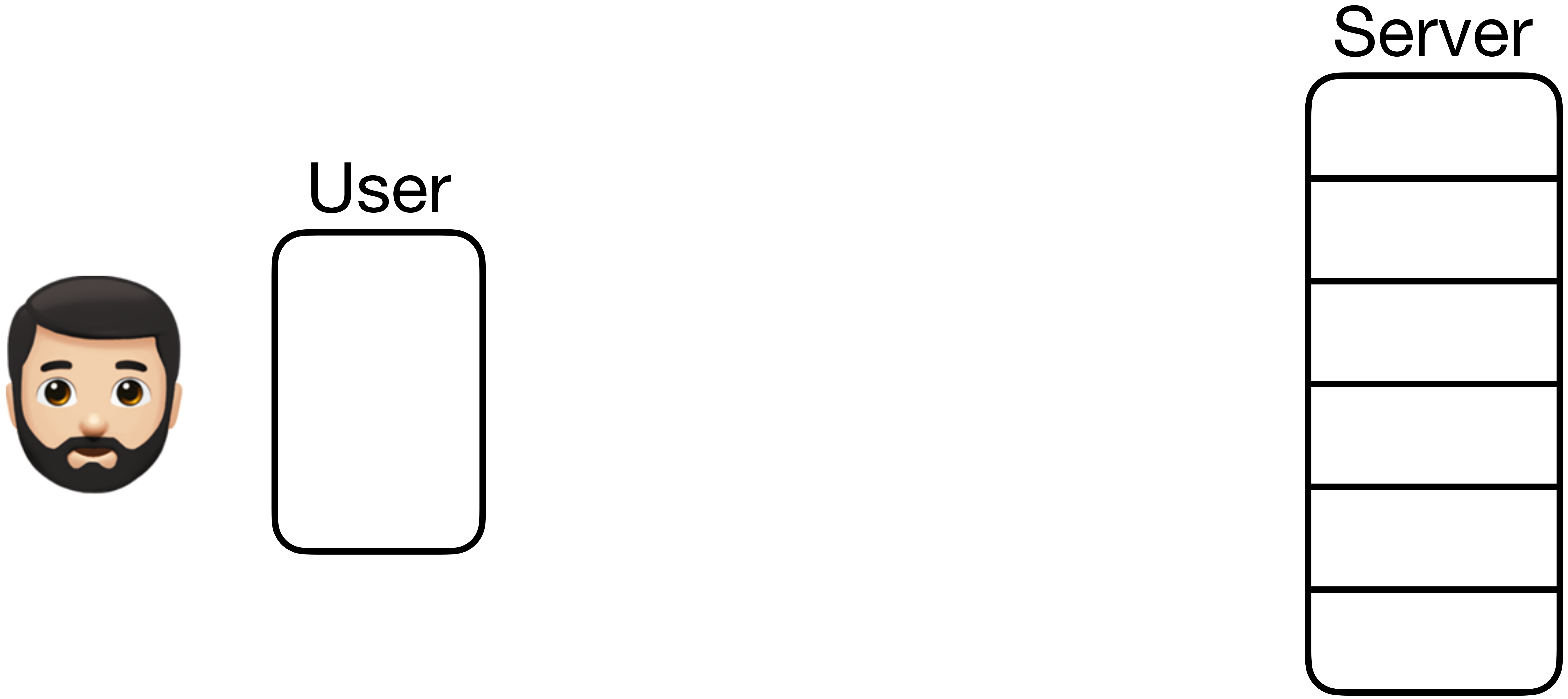


“Read-Only Reads” Assumption: No read query ever invokes a write $\widehat{\text{query}}$.

- **Completeness**: If the server behaved honestly (i.e., $\widehat{\text{query}}$ are all correct), then MC gives correct response.

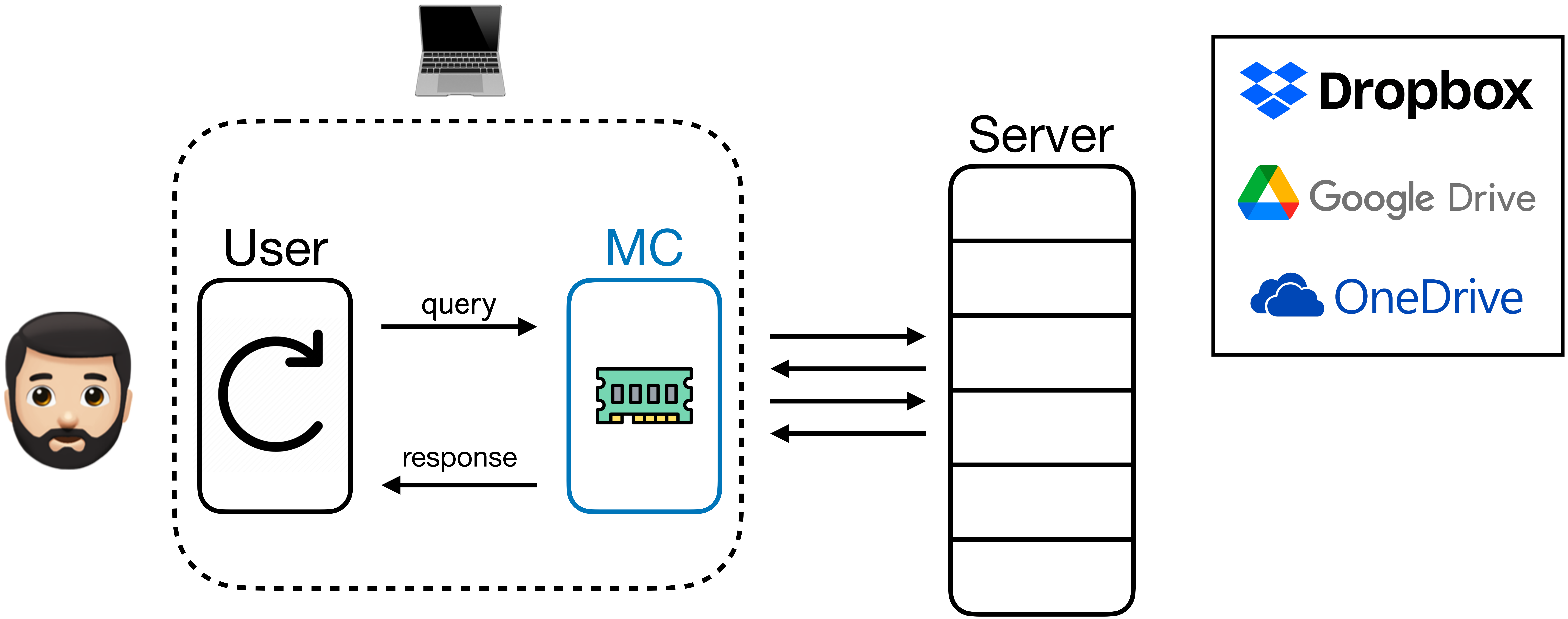
- **Soundness**: For any PPT malicious server and any sequence of user queries, the probability that the MC gives an incorrect response without **aborting** is at most p, where p is negligible.

Application: File Storage Platforms

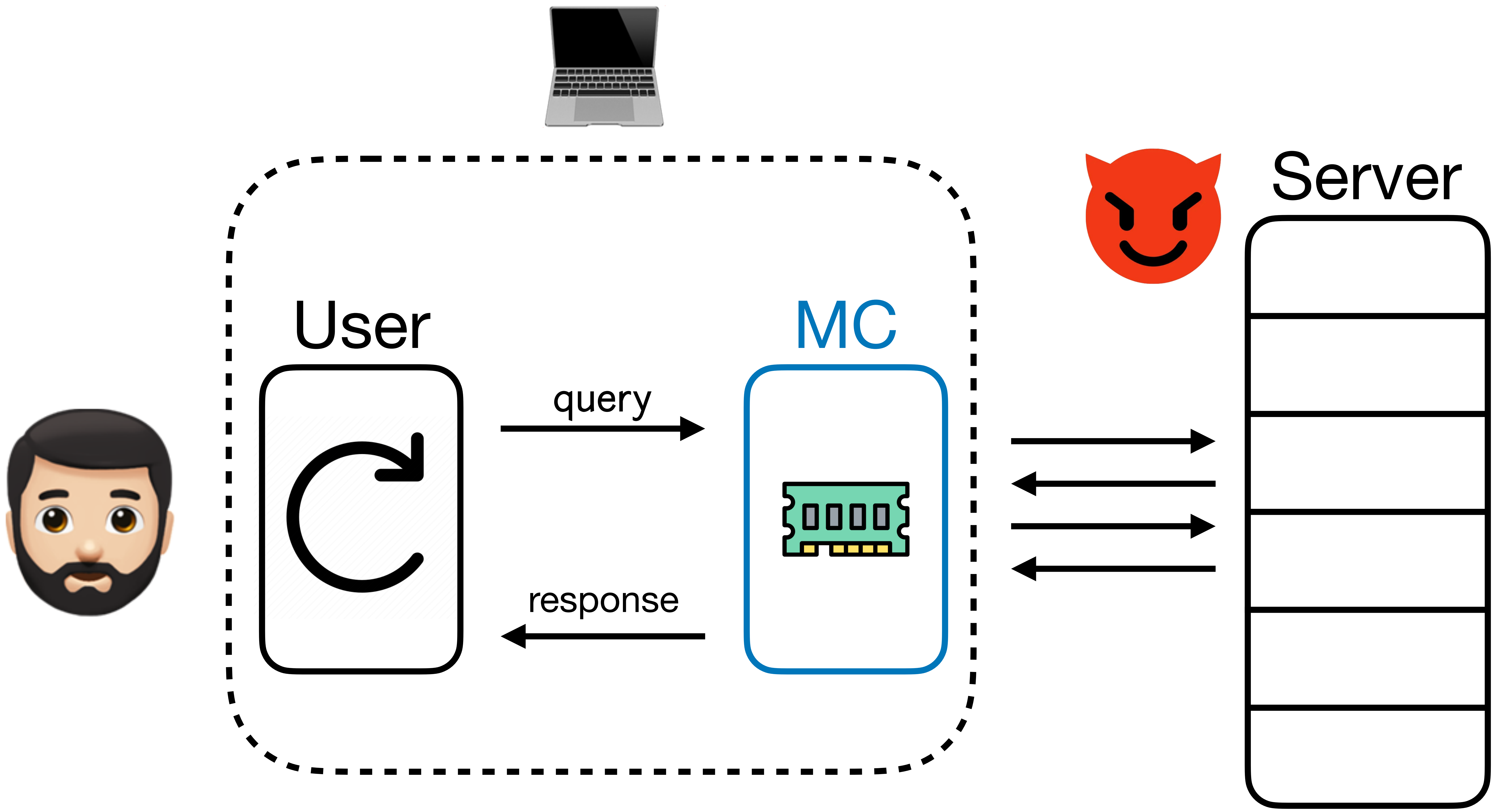


Dropbox
Google Drive
OneDrive

Application: File Storage Platforms



Application: File Storage Platforms

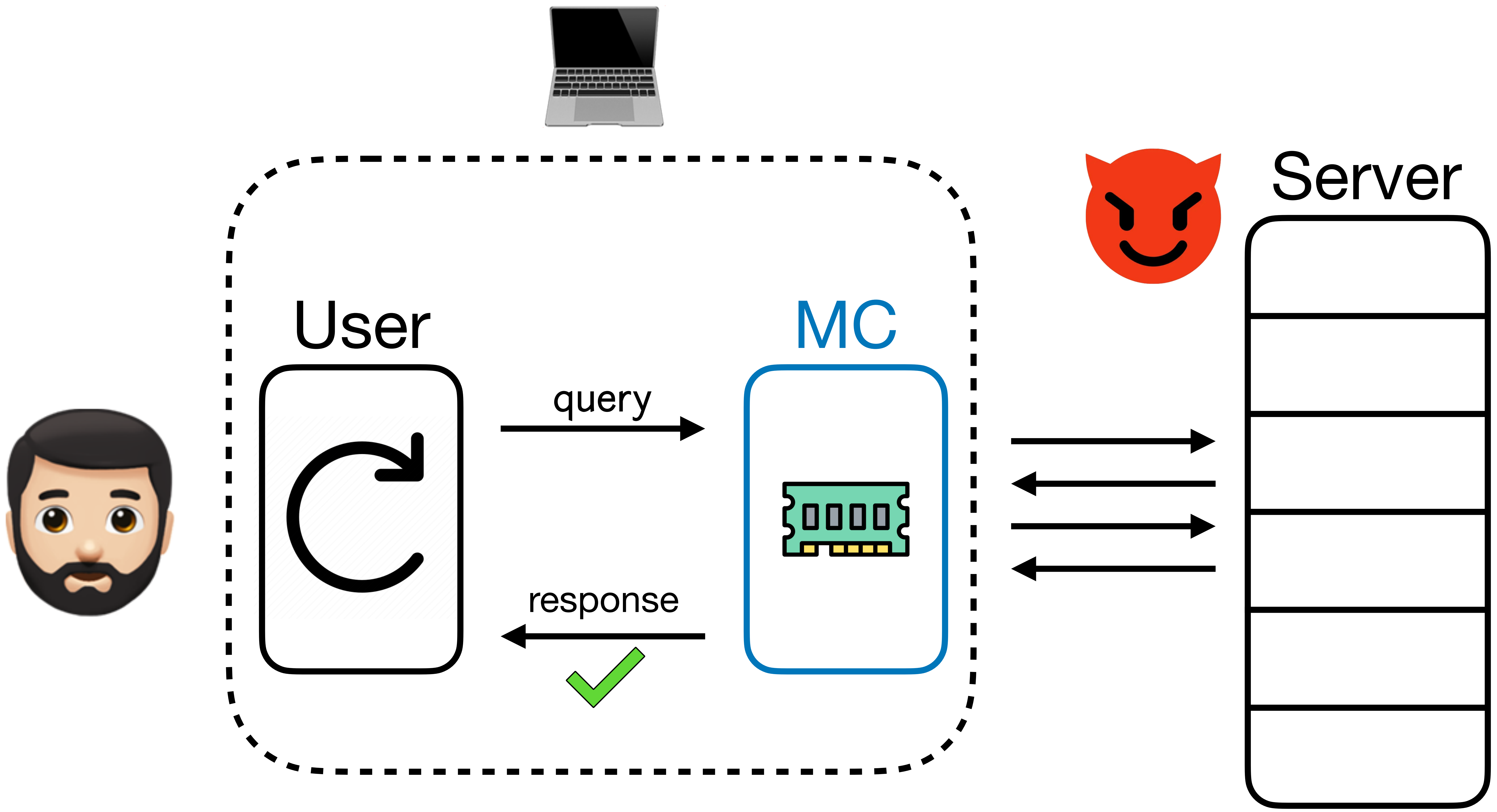


Dropbox

Google Drive

OneDrive

Application: File Storage Platforms



Dropbox

Google Drive

OneDrive

Related Applications

Related Applications

- Secure hardware (enclaves)

Related Applications

- Secure hardware (enclaves)
- Provable data possession and retrievability systems

Related Applications

- Secure hardware (enclaves)
- Provable data possession and retrievability systems
- **Offline** memory checking

Related Applications

- Secure hardware (enclaves)
- Provable data possession and retrievability systems
- **Offline** memory checking
    - Verifiable computation (SNARKs) [Setty20, BCHO22, AST23, STW23, …]

Related Applications

- Secure hardware (enclaves)
- Provable data possession and retrievability systems
- **Offline** memory checking
  - Verifiable computation (SNARKs) [Setty20, BCHO22, AST23, STW23, …]
  - Accumulation schemes [BC24, …]

Efficiency

Efficiency

Two main complexity measures:

Efficiency

Two main complexity measures:

1. **Local Space**: Amount of private space the MC can store locally.

Efficiency

Two main complexity measures:

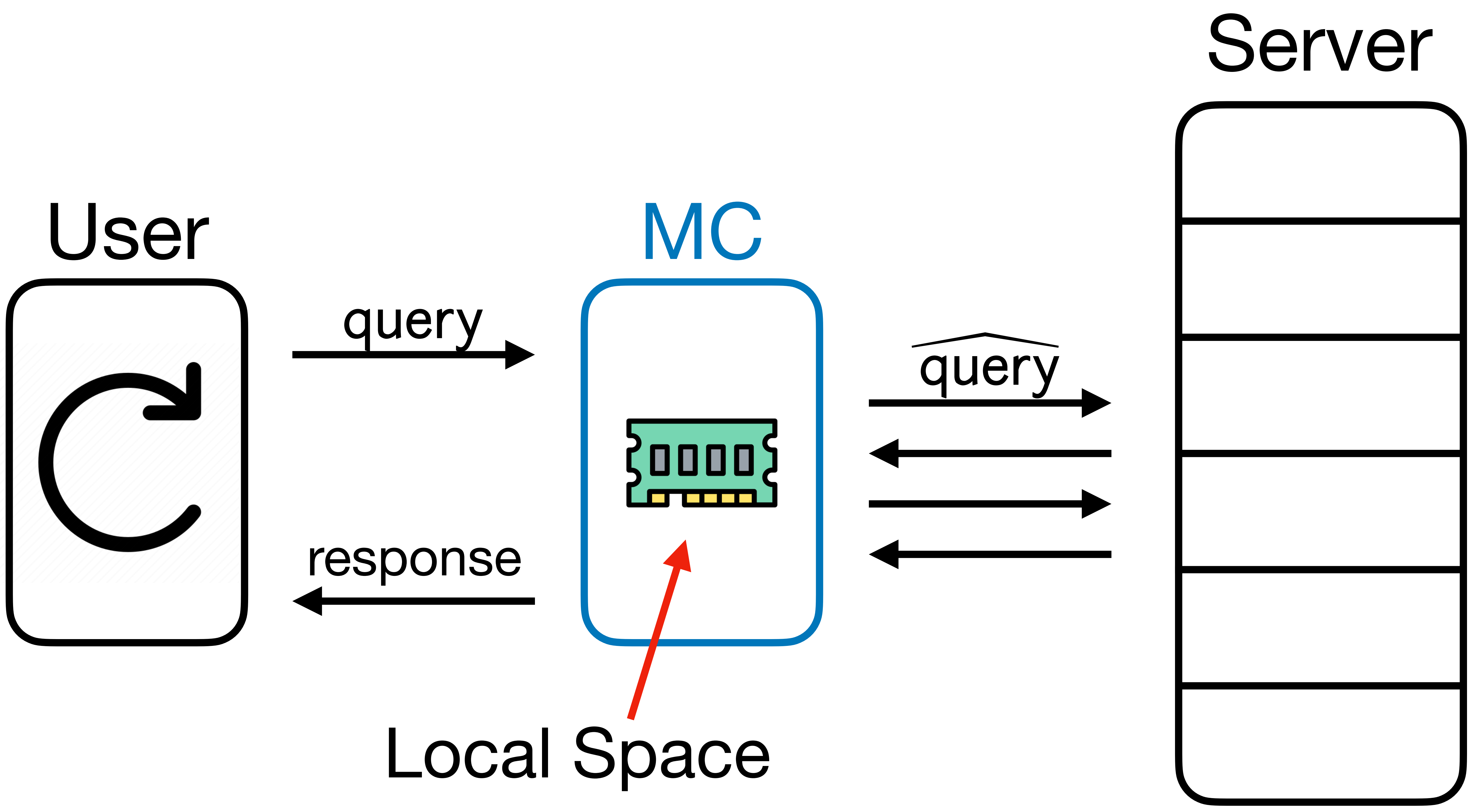
1. **Local Space**: Amount of private space the MC can store locally.
    - For storing n entries, space n is trivial (can store the full RAM itself).

Efficiency

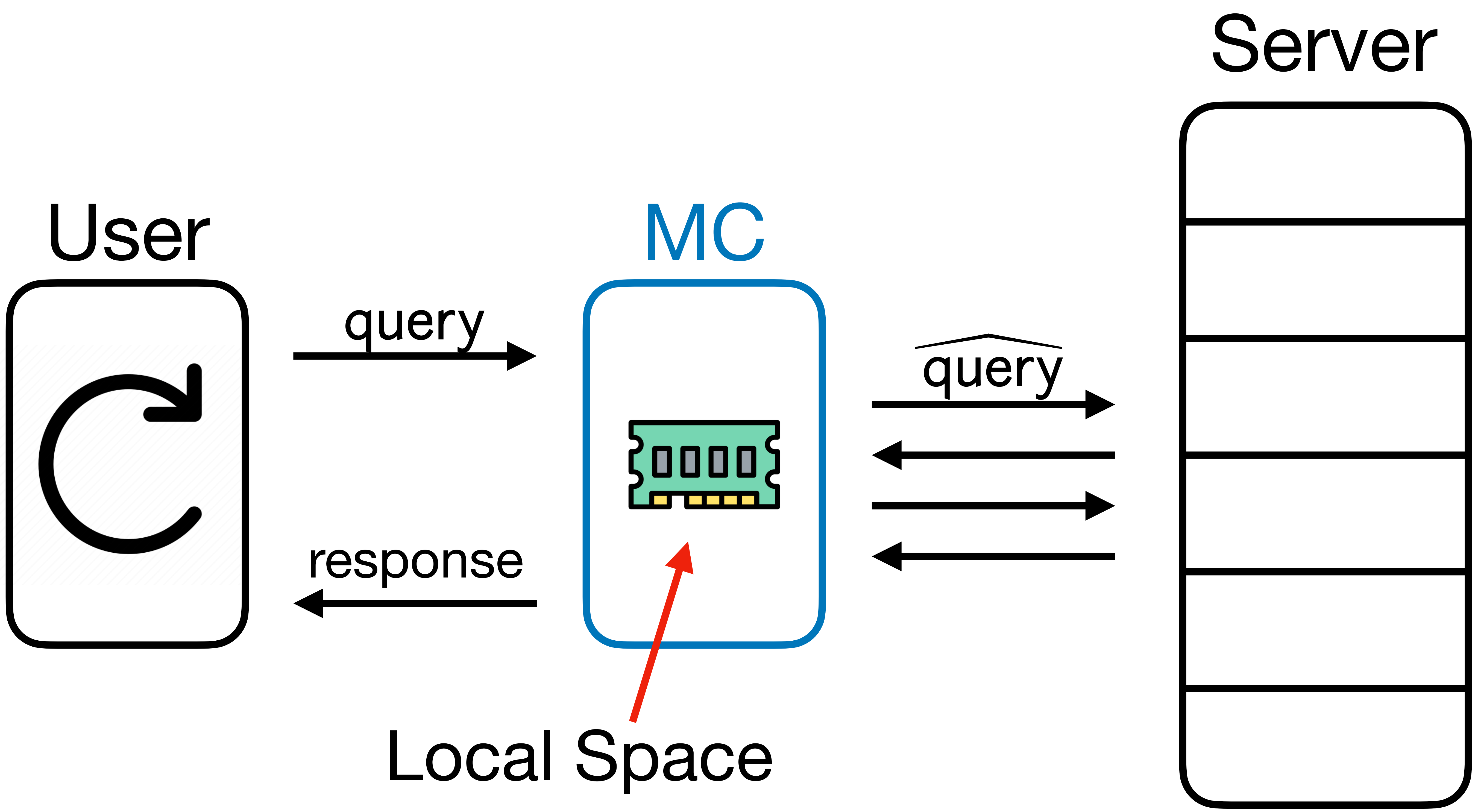
Two main complexity measures:

1. **Local Space**: Amount of private space the MC can store locally.
   - For storing n entries, space n is trivial (can store the full RAM itself).
   - For the rest of the talk, assume space at most $n^{1-\varepsilon}$ for some $\varepsilon > 0$.

Efficiency

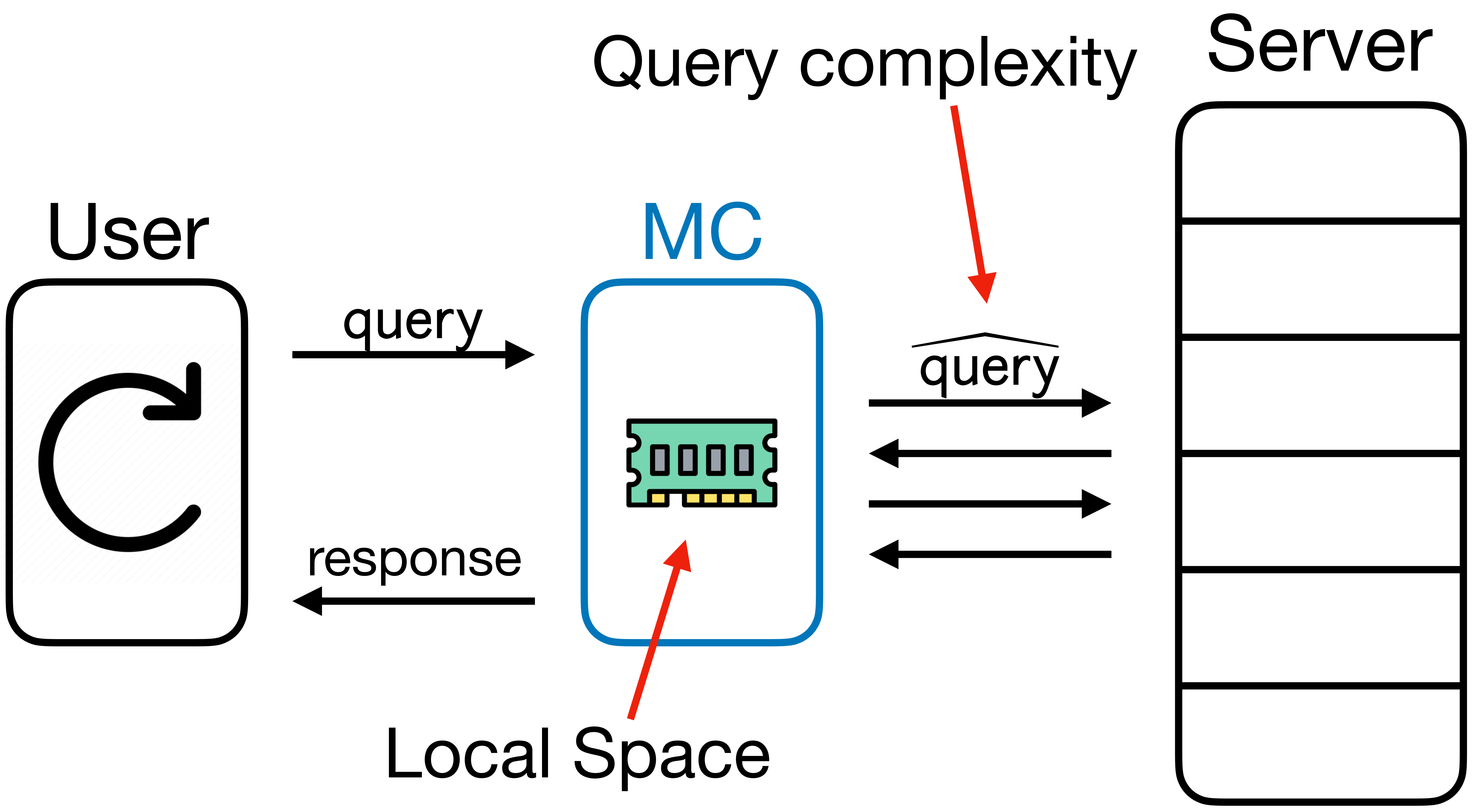


Efficiency



2. **Query complexity/overhead:** Number of physical queries made to the server per logical query. Ideally as small as possible!

Efficiency



2. **Query complexity/overhead:** Number of physical queries made to the server per logical query. Ideally as small as possible!

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound

Memory Checking: What's Known

Database of size n, word size $\mathrm{polylog}(n)$, local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical		

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	n^{ε} [Naor-Rothblum '05]

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	n^{ε} [Naor-Rothblum '05]

Tight!

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	n^{ε} [Naor-Rothblum '05]

Tight!

More generally, local space $\times$ queries $= \Theta(n)$

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	n^{ε} [Naor-Rothblum '05]
Computational		

Tight!

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	n^{ε} [Naor-Rothblum '05]
Computational	$\log n$ [Merkle '79, Blum et al. '91]	

Tight!

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	n^{ε} [Naor-Rothblum '05]
Computational	$\log n / \log\log n$ [Papamanthou-Tamassia '11]	

Tight!

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	n^{ε} [Naor-Rothblum '05]
Computational	$\log n / \log \log n$ [Papamanthou-Tamassia '11]	$\log n / \log \log n$ [Dwork-Naor-Rothblum-Vaikuntanathan '09, Boyle-Komargodski-**V.** '24]

Tight!

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound	
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	n^{ε} [Naor-Rothblum '05]	Tight!
Computational	$\log n / \log\log n$ [Papamanthou-Tamassia '11]	$\log n / \log\log n$ [Dwork-Naor-Rothblum-Vaikuntanathan '09, Boyle-Komargodski-**V.** '24]	Tight!

Memory Checking: What's Known

Database of size n, word size polylog(n), local space $n^{1-\varepsilon}$

Soundness	Upper Bound	Lower Bound
Statistical	n^{ε} [Blum et al. '91, Naor-Rothblum '05]	n^{ε} [Naor-Rothblum '05]
Computational	$\log n / \log\log n$ [Papamanthou-Tamassia '11]	$\log n / \log\log n$ [Dwork-Naor-Rothblum-Vaikuntanathan '09, Boyle-Komargodski-**V.** '24]

Tight!

Tight!

Limitations of the Computational Lower Bounds

[Dwork-Naor-Rothblum-Vaikuntanathan ’09]

[Boyle-Komargodski-**V.** ’24]

Limitations of the Computational Lower Bounds

[Dwork-Naor-Rothblum-Vaikuntanathan '09]

- Lower bound applies only to **deterministic and non-adaptive** memory checkers. **Big restriction**:

[Boyle-Komargodski-**V.** '24]

Limitations of the Computational Lower Bounds

[Dwork-Naor-Rothblum-Vaikuntanathan '09]

- Lower bound applies only to **deterministic and non-adaptive** memory checkers. **Big restriction**:
  - For every logical user query to $i \in [n]$, physical query locations must be fixed; depend only on i.

[Boyle-Komargodski-**V.** '24]

Limitations of the Computational Lower Bounds

[Dwork-Naor-Rothblum-Vaikuntanathan '09]

- Lower bound applies only to **deterministic and non-adaptive** memory checkers. **Big restriction**:
  - For every logical user query to $i \in [n]$, physical query locations must be fixed; depend only on i.

[Boyle-Komargodski-**V.** '24]

- Only rules out memory checkers with **inverse polynomial soundness error**, roughly $p \approx 1/n$.

Limitations of the Computational Lower Bounds

[Dwork-Naor-Rothblum-Vaikuntanathan ’09]

- Lower bound applies only to **deterministic and non-adaptive** memory checkers. **Big restriction**:
  - For every logical user query to $i \in [n]$, physical query locations must be fixed; depend only on i.

[Boyle-Komargodski-**V.** ’24]

- Only rules out memory checkers with **inverse polynomial soundness error**, roughly $p \approx 1/n$.
- Doesn’t rule out super-efficient MCs with **larger soundness error**.

Covert Security [Aumann-Lindell '07]

Covert Security [Aumann-Lindell ’07]

- In many settings (e.g., commercial, political, social), **malicious adversaries don’t want to get caught**.

Covert Security [Aumann-Lindell ’07]

- In many settings (e.g., commercial, political, social), **malicious adversaries don’t want to get caught**.

- Negligible soundness **overkill**!

Covert Security [Aumann-Lindell '07]

- In many settings (e.g., commercial, political, social), **malicious adversaries don't want to get caught**.

- Negligible soundness **overkill**!

- $\Omega(1)$ soundness error is sufficient. Detecting adversaries with 90% probability, instead of (100 - negl)%, is **enough of a disincentive**.

Covert Security [Aumann-Lindell ’07]

- In many settings (e.g., commercial, political, social), **malicious adversaries don’t want to get caught**.

- Negligible soundness **overkill**!

- $\Omega(1)$ soundness error is sufficient. Detecting adversaries with 90% probability, instead of (100 - negl)%, is **enough of a disincentive**.

- This relaxation has enabled asymptotic **efficiency gains** in terms of computational overhead and communication. (e.g, [Aumann-Lindell ’07, Goyal-Mohassel-Smith ’08, Hazay-Lindell ’10])

Covert Security [Aumann-Lindell ’07]

- In many settings (e.g., commercial, political, social), **malicious adversaries don’t want to get caught**.

- Negligible soundness **overkill**!

- $\Omega(1)$ soundness error is sufficient. Detecting adversaries with 90% probability, instead of (100 - negl)%, is **enough of a disincentive**.

- This relaxation has enabled asymptotic **efficiency gains** in terms of computational overhead and communication. (e.g, [Aumann-Lindell ’07, Goyal-Mohassel-Smith ’08, Hazay-Lindell ’10])

- Naturally fits into memory checking setting: file storage cloud server doesn’t want to harm their reputation!

Main Question

Main Question

Can MCs, relaxed to covert security (soundness $\Omega(1)$), have query complexity $q \ll \log n / \log \log n$? $O(1)$?

Main Question

Can MCs, relaxed to covert security (soundness $\Omega(1)$), have query complexity $q \ll \log n / \log \log n$? $O(1)$?

Concrete Example: Is there a MC with 5% soundness error and $q = 2$?

Main Result

Main Result

- We show:

Main Result

- We show:

<u>Theorem</u>: Every memory checker*, even with $\Omega(1)$ soundness error, must have $q = \Omega(\log n / \log \log n)$.

*Assuming it has read-only reads

Main Result

- We show:

<u>Theorem</u>: Every memory checker*, even with $\Omega(1)$ soundness error, must have $q = \Omega(\log n / \log \log n)$.

- **Tight** up to constant factors. [Papamanthou-Tamassia '11]

*Assuming it has read-only reads

Main Result

- We show:

Theorem: Every memory checker*, even with $\Omega(1)$ soundness error, must have $q = \Omega(\log n / \log \log n)$.

- **Tight** up to constant factors. [Papamanthou-Tamassia '11]

- **Unconditional**. Holds regardless of any computational assumptions.

*Assuming it has read-only reads

Main Result

- We show:

<u>Theorem</u>: Every memory checker*, even with $\Omega(1)$ soundness error, must have $q = \Omega(\log n / \log \log n)$.

- **Tight** up to constant factors. [Papamanthou-Tamassia '11]
- **Unconditional**. Holds regardless of any computational assumptions.
- Handles **randomized** and **adaptive** memory checkers.

*Assuming it has read-only reads

Main Result

- We show:

<u>Theorem</u>: Every memory checker*, even with $\Omega(1)$ soundness error, must have $q = \Omega(\log n / \log \log n)$.

- **Tight** up to constant factors. [Papamanthou-Tamassia ’11]
- **Unconditional**. Holds regardless of any computational assumptions.
- Handles **randomized** and **adaptive** memory checkers.
- **An Interpretation**: Unlike many other MPC functionalities, covert security **does not** enable efficiency gains for memory checking.

*Assuming it has read-only reads

Technical Overview

Our Approach

Our Approach

- Just like [Boyle-Komargodski-**V.**'24], we can use a MC that's *too efficient* to compress random bits.

Our Approach

- Just like [Boyle-Komargodski-**V.**'24], we can use a MC that's *too efficient* to compress random bits.
- Will use following style of compression lemma:

Our Approach

- Just like [Boyle-Komargodski-**V.**'24], we can use a MC that's *too efficient* to compress random bits.
- Will use following style of compression lemma:
  - Transmitting uniformly random $S \subseteq [n]$ from Alice to Bob where $|S| = k$ requires $\log \binom{n}{k}$ bits, even with shared indep. randomness.

Protocol

Knows $S \subseteq [n]$

Wants to recover S

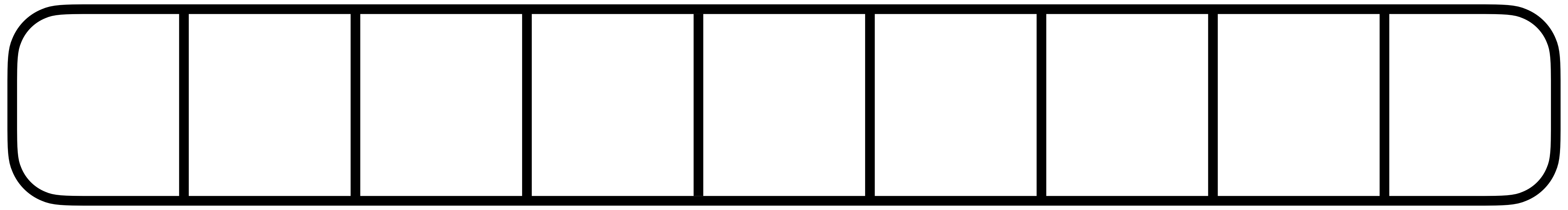
Publicly initialize MC:

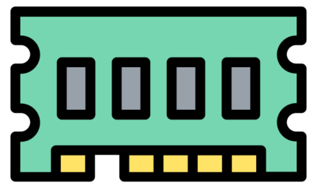
(by performing write$(i, 0)$ for all $i \in [n]$)

Knows $S \subseteq [n]$

Protocol

DB_0



st_0 

Wants to recover S

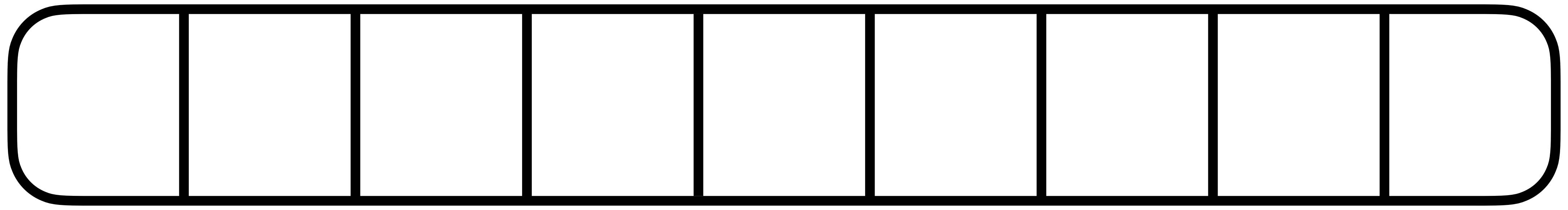
Publicly initialize MC:

(by performing write$(i,\ 0)$ for all $i \in [n]$)

Protocol

Knows $S \subseteq [n]$

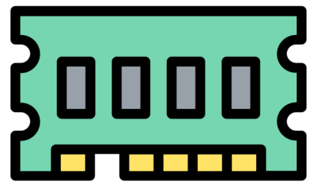
DB_0



Wants to recover S

Publicly initialize MC:

(by performing $\mathsf{write}(i, 0)$ for all $i \in [n]$)

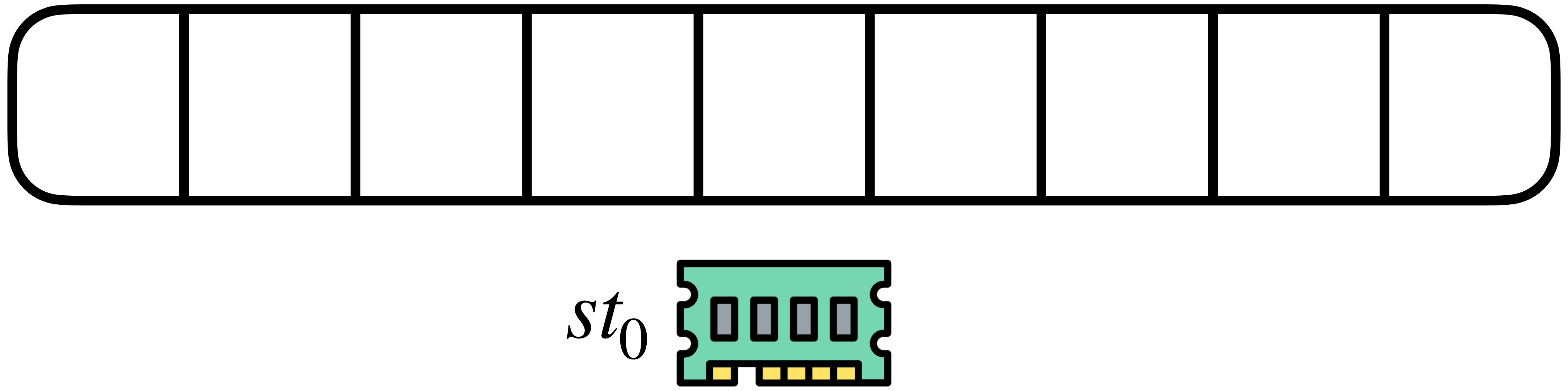
st_0 

For each $i \in S$:

Protocol

Knows $S \subseteq [n]$

DB_0



st_0

Wants to recover S

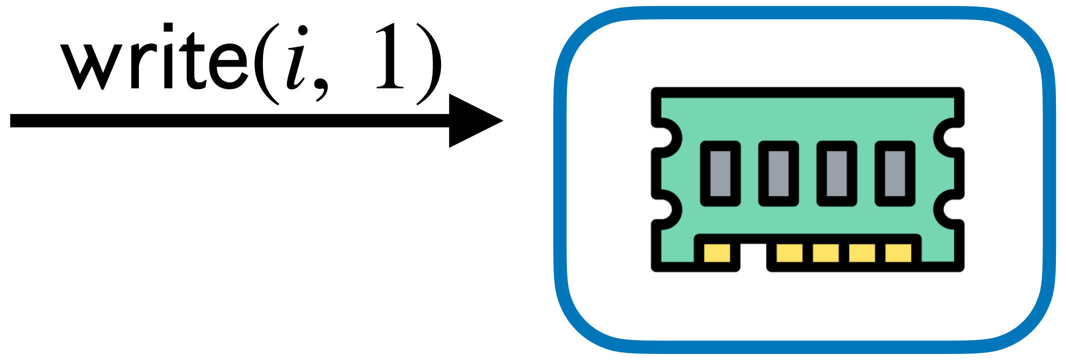
Publicly initialize MC:

(by performing write$(i,\ 0)$ for all $i \in [n]$)

For each $i \in S$:

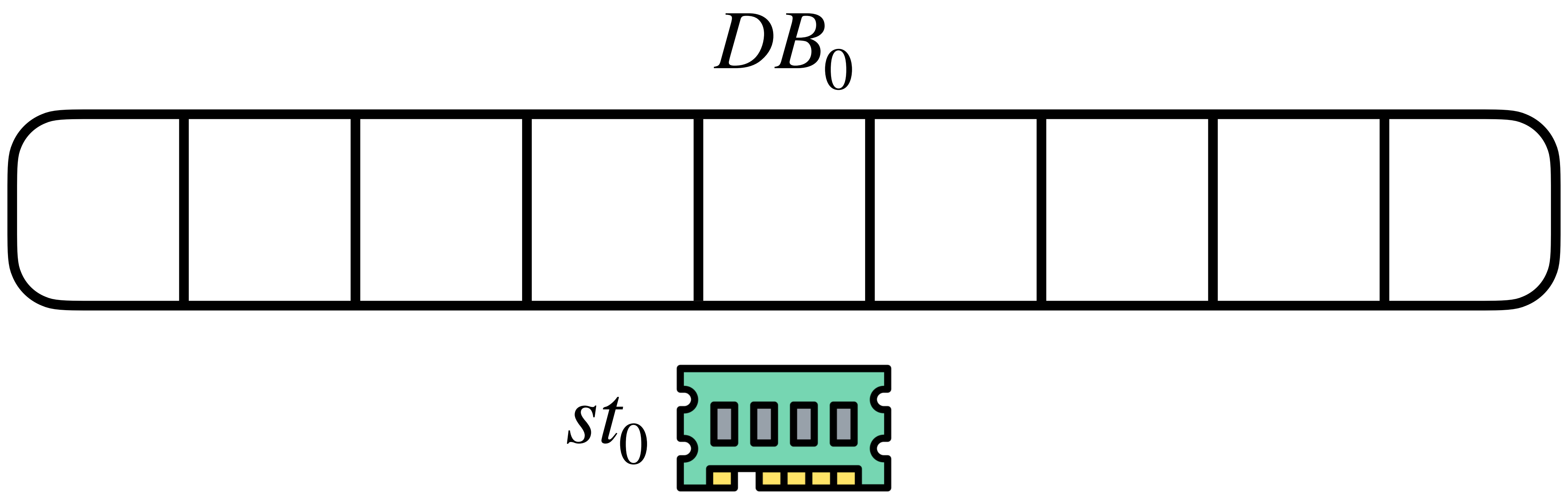
MC

write$(i,\ 1)$



Protocol

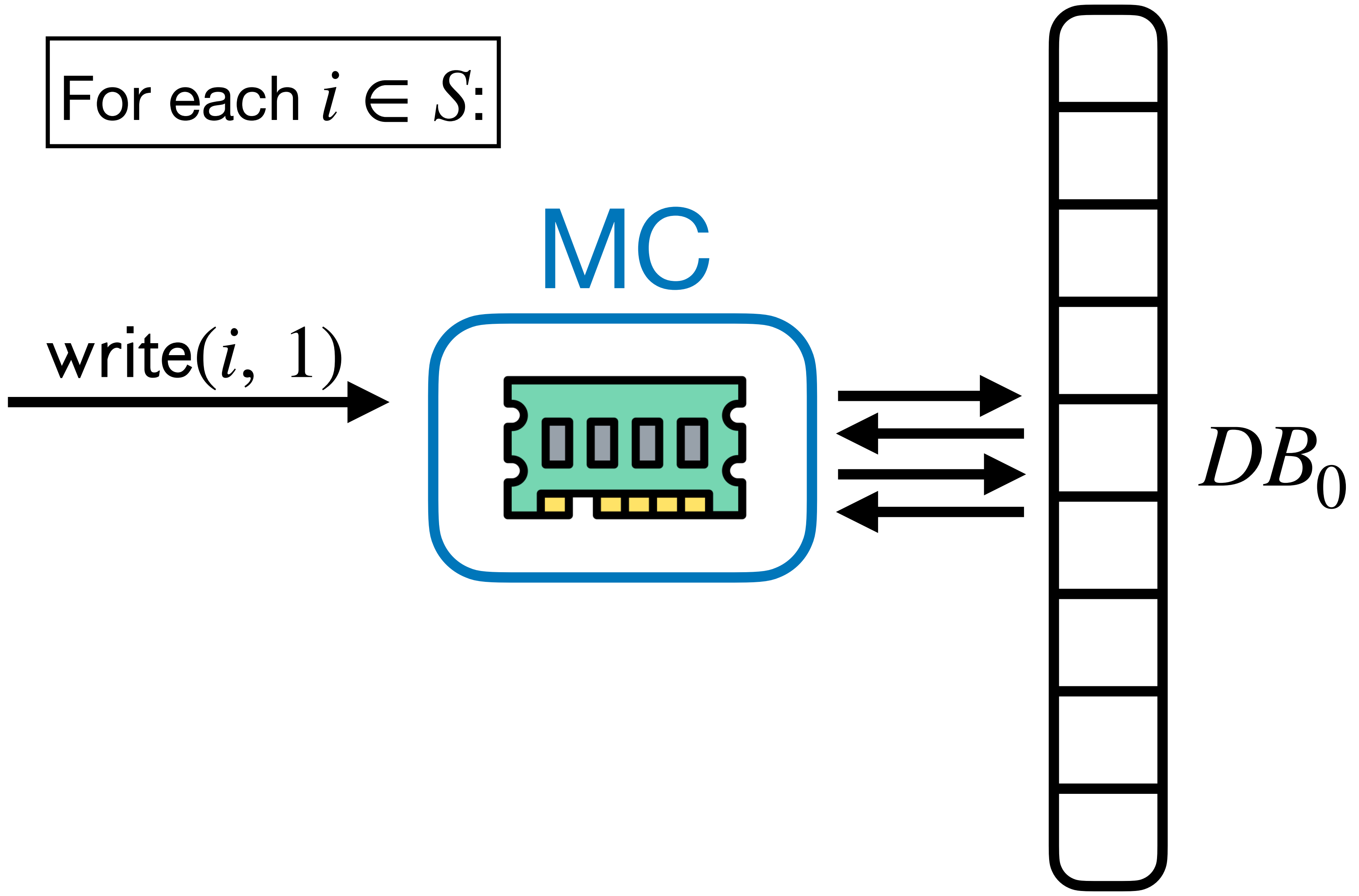
Knows $S \subseteq [n]$



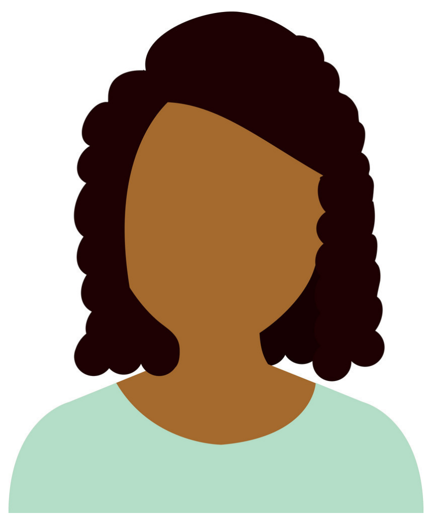
Wants to recover S

Publicly initialize MC:
(by performing write$(i,\ 0)$ for all $i \in [n]$)

For each $i \in S$:



Protocol

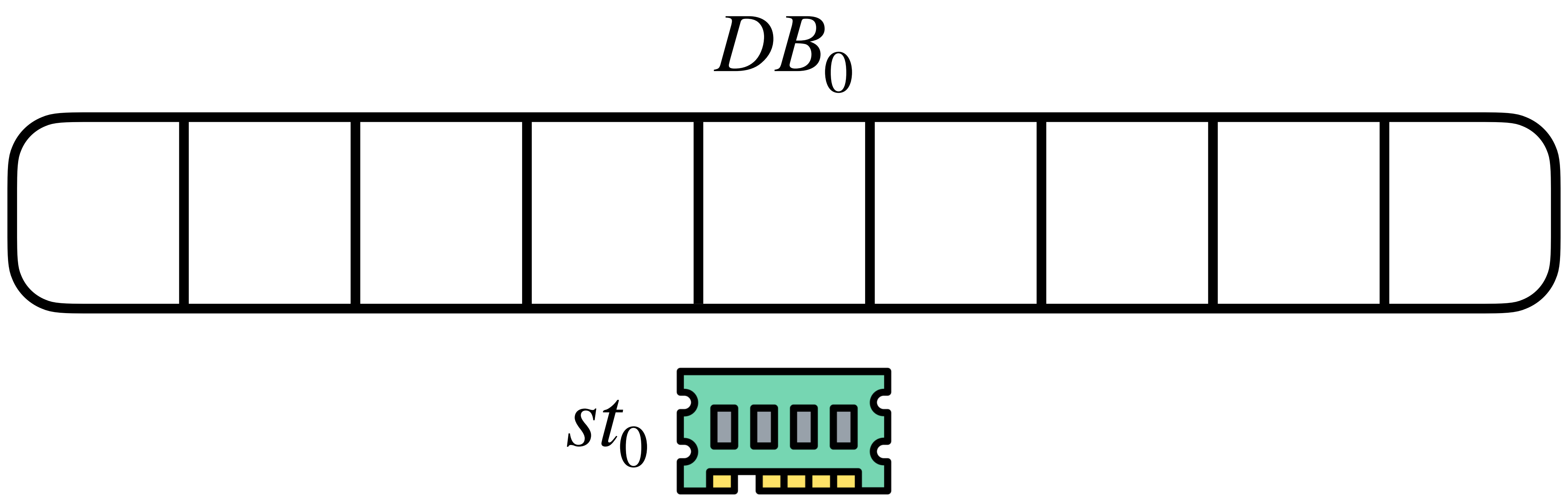


Knows $S \subseteq [n]$

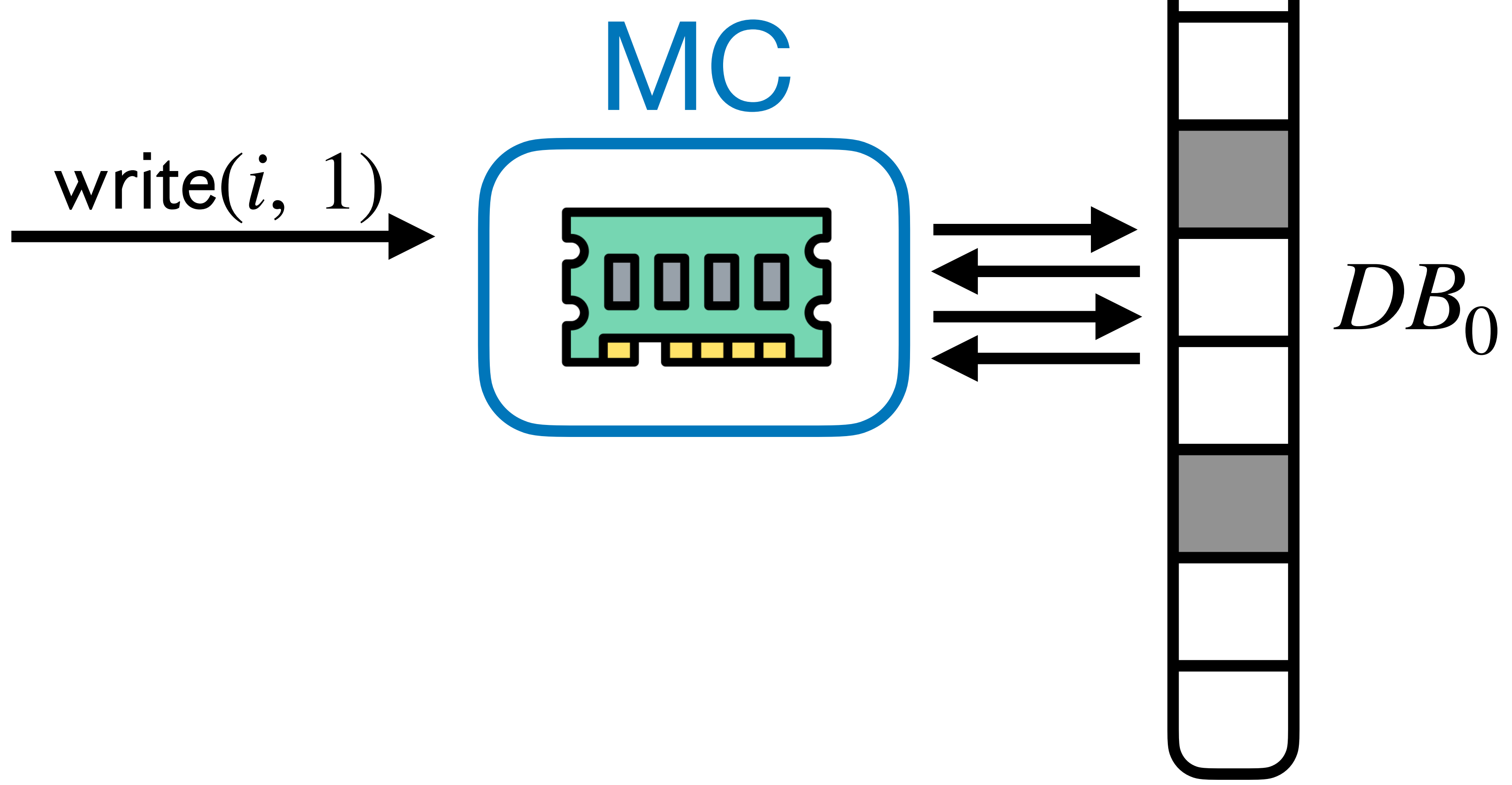
Wants to recover S

Publicly initialize MC:

(by performing write$(i,\ 0)$ for all $i \in [n]$)



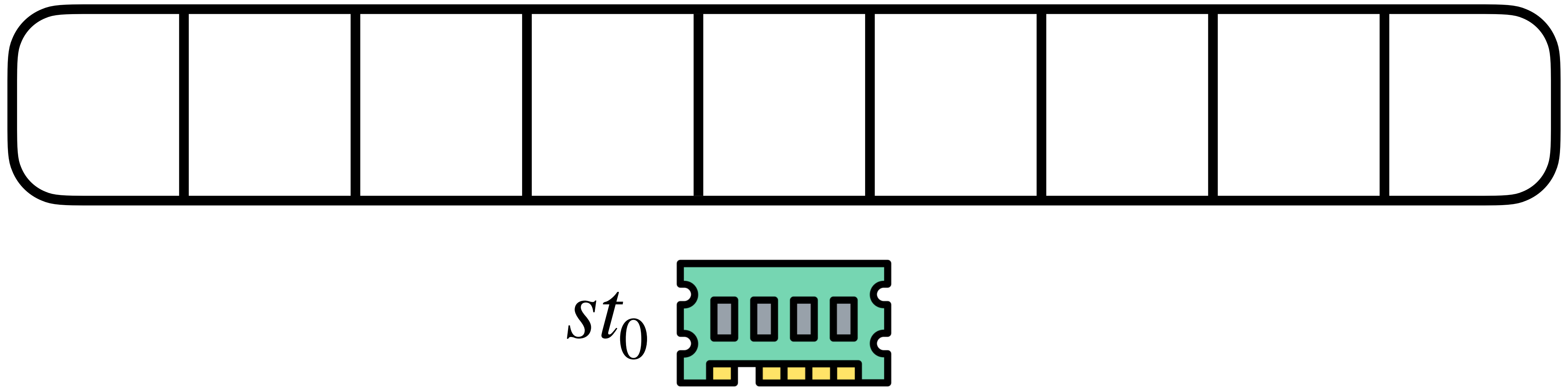
For each $i \in S$:



Protocol

Knows $S \subseteq [n]$

DB_0



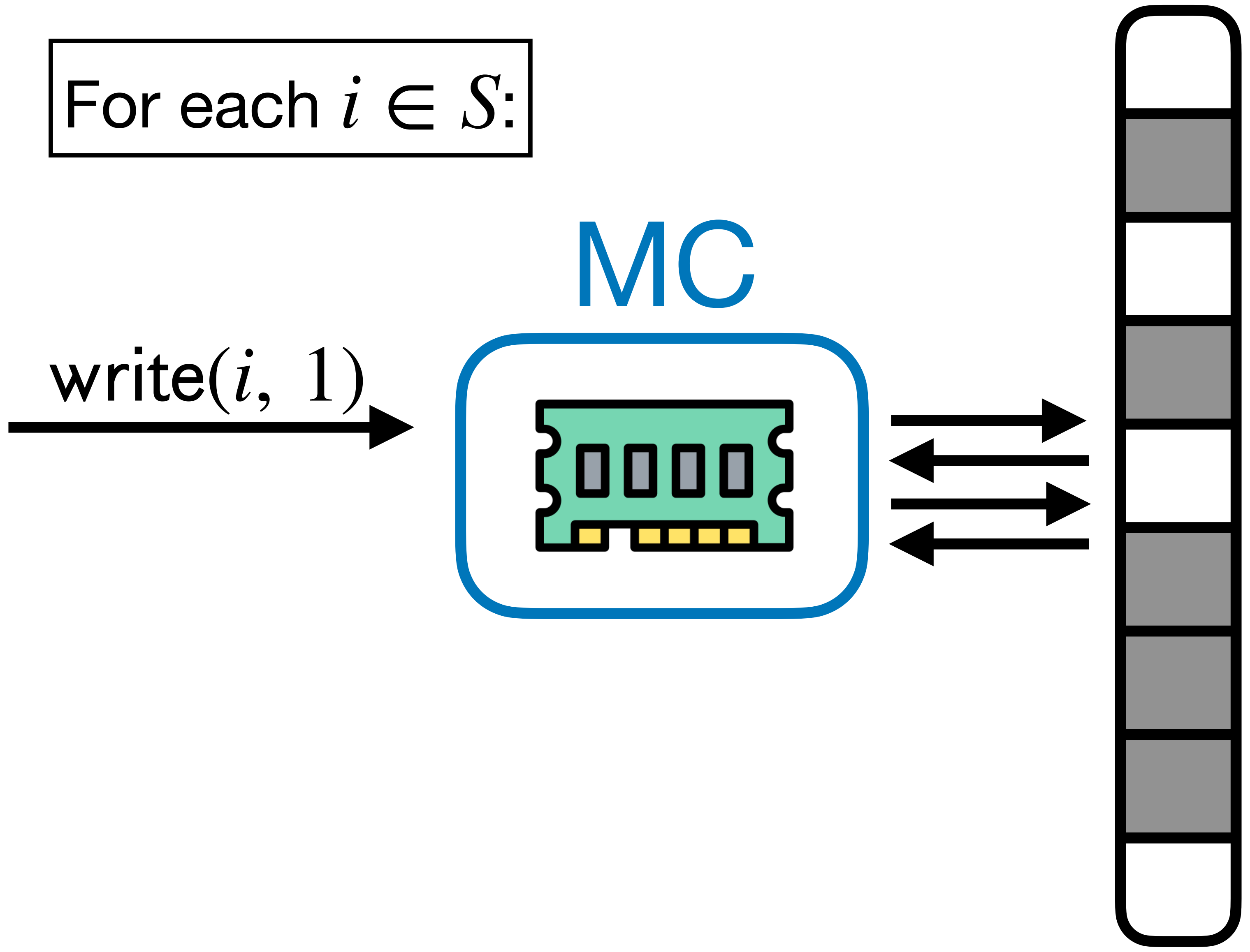
st_0

Wants to recover S

Publicly initialize MC:

(by performing write$(i,\ 0)$ for all $i \in [n]$)

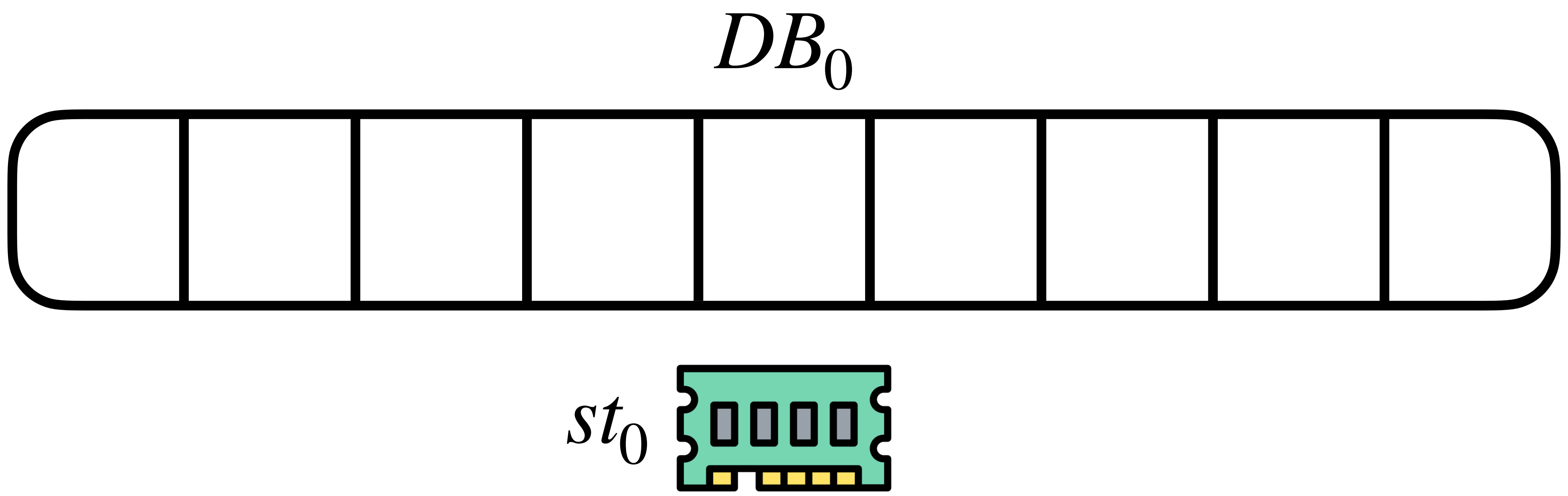
For each $i \in S$:



Protocol

Knows $S \subseteq [n]$

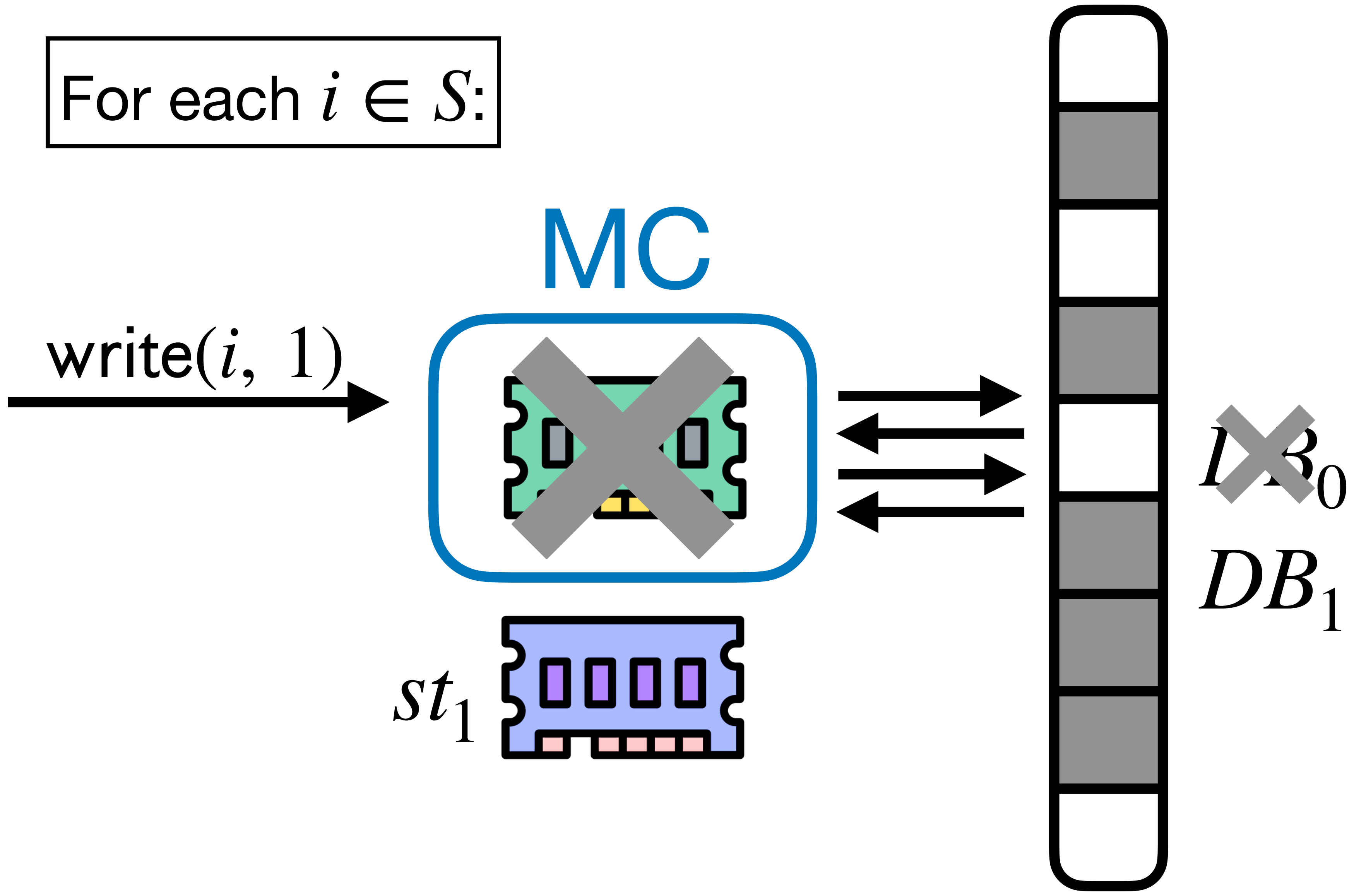
Wants to recover S



Publicly initialize MC:

(by performing write$(i,\ 0)$ for all $i \in [n]$)

For each $i \in S$:




Protocol
Knows $S \subseteq [n]$
Wants to recover S
DB_0
Publicly initialize MC:
(by performing write$(i, 0)$ for all $i \in [n]$)
st_0
For each $i \in S$:
MC
write$(i, 1)$
$\cancel{DB_0}$
DB_1
st_1
st_1


Protocol
Knows $S \subseteq [n]$
Wants to recover S
DB_0
Publicly initialize MC:
(by performing write$(i, 0)$ for all $i \in [n]$)
st_0
For each $i \in S$:
MC
write$(i, 1)$
st_1
$\cancel{DB_0}$
DB_1
st_1
For each $i \in [n]$:

Protocol


Knows $S \subseteq [n]$
Publicly initialize MC:
(by performing write$(i,\ 0)$ for all $i \in [n]$)
DB_0
st_0
Wants to recover S
For each $i \in S$:
MC
write$(i,\ 1)$
st_1
$\cancel{DB_0}$
DB_1
st_1
For each $i \in [n]$:
read(i)
MC


Protocol
Knows $S \subseteq [n]$
Wants to recover S
DB_0
Publicly initialize MC:
(by performing write$(i, 0)$ for all $i \in [n]$)
st_0
For each $i \in S$:
MC
write$(i, 1)$
st_1
$\cancel{DB_0}$
DB_1
st_1
For each $i \in [n]$:
MC
read(i)

Protocol
Knows $S \subseteq [n]$
Wants to recover S
DB_0
Publicly initialize MC:
(by performing write(i, 0) for all $i \in [n]$)
st_0
For each $i \in S$:
MC
write(i, 1)
st_1
$\cancel{DB_0}$
DB_1
st_1
For each $i \in [n]$:
MC
read(i)
DB_0
$\cancel{DB_1}$

Protocol

Knows $S \subseteq [n]$
Publicly initialize MC:
(by performing write$(i, 0)$ for all $i \in [n]$)
DB_0
st_0
Wants to recover S
For each $i \in S$:
MC
write$(i, 1)$
st_1
$\cancel{DB_0}$
DB_1
st_1
For each $i \in [n]$:
MC
read(i)
$r_i \in \{0, 1\}$
DB_0
$\cancel{DB_1}$


Protocol
Knows $S \subseteq [n]$
Wants to recover S
DB_0
Publicly initialize MC:
(by performing write$(i, 0)$ for all $i \in [n]$)
st_0
For each $i \in S$:
MC
write$(i, 1)$
st_1
$\cancel{DB_0}$
DB_1
st_1
For each $i \in [n]$:
MC
read(i)
$r_i \in \{0, 1\}$
Great!
$r_i = 1 \iff i \in S$
by **soundness** of MC.
DB_0
$\cancel{DB_1}$


Protocol
Knows $S \subseteq [n]$
Wants to recover S
DB_0
Publicly initialize MC:
(by performing write$(i, 0)$ for all $i \in [n]$)
st_0
For each $i \in S$:
MC
write$(i, 1)$
DB_0
DB_1
st_1
st_1
For each $i \in [n]$:
MC
read(i)
$r_i \in \{0, 1\}$
Great!
$r_i = 1 \iff i \in S$
by **soundness** of MC.
Abort: $r_i = \perp$
DB_0
DB_1


Protocol
Knows $S \subseteq [n]$
Wants to recover S
Publicly initialize MC:
(by performing write$(i, 0)$ for all $i \in [n]$)
DB_0
st_0
For each $i \in S$:
MC
write$(i, 1)$
st_1
$\cancel{DB_0}$
DB_1
st_1
For each $i \in [n]$:
MC
read(i)
$r_i \in \{0, 1\}$
Great!
$r_i = 1 \iff i \in S$
by **soundness** of MC.
Abort: $r_i = \perp$
Can't conclude anything…
DB_0
$\cancel{DB_1}$

Key [BKV. ’24] Idea: Partition the Server’s Memory

Key [BKV. '24] Idea: Partition the Server's Memory

- Analyze the query distribution of $\mathsf{read}(i)$ (where $i \leftarrow [n]$):

Key [BKV. '24] Idea: Partition the Server's Memory

- Analyze the query distribution of read(i) (where $i \leftarrow [n]$):
  - **Heavy** set H: Small set, all have high probability mass.
  - **Medium** set M: "Total" guarantee of low mass.
  - **Light** set L: "Point-wise" guarantee of low mass.

Protocol

Knows $S \subseteq [n]$
Wants to recover S
DB_0
Publicly initialize MC:
(by performing write$(i, 0)$ for all $i \in [n]$)
st_0
For each $i \in S$:
MC
write$(i, 1)$
st_1
$\cancel{DB_0}$
DB_1
st_1
For each $i \in [n]$:
MC
read(i)
$r_i \in \{0, 1\}$
DB_1
$\cancel{DB_0}$
Abort: $r_i = \perp$
Can't conclude anything…

Protocol

Knows $S \subseteq [n]$
DB_0
Wants to recover S
Publicly initialize MC:
(by performing write$(i,\ 0)$ for all $i \in [n]$)
st_0
For each $i \in S$:
MC
write$(i,\ 1)$
$\cancel{DB_0}$
DB_1
st_1
st_1, H, $DB_1|_H$
$H \cap$ =
$M \cap$ =
$L \cap$ =
For each $i \in [n]$:
MC
read(i)
$r_i \in \{0,\ 1\}$
Abort: $r_i = \perp$
Can't conclude anything…
$DB_1|_H$ +
$DB_0|_{L \sqcup M}$


Protocol
Knows $S \subseteq [n]$
DB_0
Wants to recover S
Publicly initialize MC:
(by performing write$(i, 0)$ for all $i \in [n]$)
st_0
For each $i \in S$:
MC
write$(i, 1)$
st_1
$\cancel{DB_0}$
DB_1
st_1, H, $DB_1|_H$
For each $i \in [n]$:
MC
read(i)
$r_i \in \{0, 1\}$
$DB_1|_H$ +
$DB_0|_{L \sqcup M}$
$H \cap$ =
$M \cap$ =
$L \cap$ =
Abort: $r_i = \bot$
Can't conclude anything...
Now, will usually prevent this!

Issue: Computing the Partition

Issue: Computing the Partition

- The compression argument relies on the adversary being able to **compute** which queries are **heavy** to do a replay attack on it (during one logical read).

Issue: Computing the Partition

- The compression argument relies on the adversary being able to **compute** which queries are **heavy** to do a replay attack on it (during one logical read).
  - [BK**V.** ’24] approach: Randomly guess whether each query is heavy.

Issue: Computing the Partition

- The compression argument relies on the adversary being able to **compute** which queries are **heavy** to do a replay attack on it (during one logical read).
  - [BK**V**. '24] approach: Randomly guess whether each query is heavy.
  - **Problem: Incurs 2^q security loss.**

Issue: Computing the Partition

- The compression argument relies on the adversary being able to **compute** which queries are **heavy** to do a replay attack on it (during one logical read).
  - [BK**V**. '24] approach: Randomly guess whether each query is heavy.
  - **Problem: Incurs 2^q security loss.**
- Can we avoid this guessing?

Issue: Computing the Partition

- The compression argument relies on the adversary being able to **compute** which queries are **heavy** to do a replay attack on it (during one logical read).
    - [BK**V.** ’24] approach: Randomly guess whether each query is heavy.
    - **Problem: Incurs 2^q security loss.**
- Can we avoid this guessing?
    - H could depend on **private, internal randomness** of the MC.

Issue: Computing the Partition

- The compression argument relies on the adversary being able to **compute** which queries are **heavy** to do a replay attack on it (during one logical read).
  - [BK**V.** ’24] approach: Randomly guess whether each query is heavy.
  - **Problem: Incurs 2^q security loss.**
- Can we avoid this guessing?
  - H could depend on **private, internal randomness** of the MC.
  - H could **adaptively** change as queries are sent to the MC.

Our Idea in a Nutshell

Our Idea in a Nutshell

The adversary can **efficiently learn** an approximation of H by making many dummy read(i) queries.

Our Idea in a Nutshell

The adversary can efficiently learn an approximation of H by making many dummy read(i) queries.

- By “read-only reads” property, making read(i) queries doesn’t change H.

Our Idea in a Nutshell

The adversary can efficiently learn an approximation of H by making many dummy read(i) queries.

- By "read-only reads" property, making read(i) queries doesn't change H.
- Making poly(n) queries is sufficient to learn a sufficiently good approximation of H.

Our Idea in a Nutshell

The adversary can efficiently learn an approximation of H by making many dummy read(i) queries.

- By "read-only reads" property, making read(i) queries doesn't change H.
- Making poly(n) queries is sufficient to learn a sufficiently good approximation of H.
- Analysis follows from multiplicative and additive Chernoff bounds.

Summary

Summary

- **Memory Checkers** (MCs) remove need for **trusting integrity** when using remote cloud storage.

Summary

- **Memory Checkers** (MCs) remove need for **trusting integrity** when using remote cloud storage.
- We prove **tight, unconditional** lower bounds for MCs, showing that Merkle-style constructions are optimal **even when relaxing to covert security.**

Summary

- **Memory Checkers** (MCs) remove need for **trusting integrity** when using remote cloud storage.

- We prove **tight, unconditional** lower bounds for MCs, showing that Merkle-style constructions are optimal **even when relaxing to covert security.**

  - Previously known only for **deterministic and non-adaptive** MCs or for MCs with inverse-polynomial soundness.

Open Questions

Open Questions

- Is there a more general framework to understand when relaxing covert security will enable efficiency gains or not?

Open Questions

- Is there a more general framework to understand when relaxing covert security will enable efficiency gains or not?
- Is there any way to avoid “read-only reads” assumption?

Open Questions

- Is there a more general framework to understand when relaxing covert security will enable efficiency gains or not?
- Is there any way to avoid “read-only reads” assumption?

Thanks!