# Large Scale Systems
# CS 410 / 510

**Lecture 6:**

**Replication and Consistency**

**Suyash Gupta**

Assistant Professor

Distopia Labs and ORNG

Dept. of Computer Science

(E) suyash@uoregon.edu

(W) gupta-suyash.github.io

# Assignment 1 Due Today!

- **Assignment 1 is out!**
- Please work with your groups to understand the underlying system.
- Assignment 1 report **deadline** → Today by 11:59pm.

# Last Class

- Last class we looked at:

- Need for Commit Protocols
- Two Phase Commit
- Three Phase Commit

# Replication

- Previously, we discussed briefly about replication.

- What are the **benefits** of replication?

# Replication

- Previously, we discussed briefly about replication.

- What are the **benefits** of replication?
  - Availability!

# Replication

- Previously, we discussed briefly about replication.

- What are the **benefits** of replication?
  - Availability!

- What is the key **challenge** for replication?

# Replication

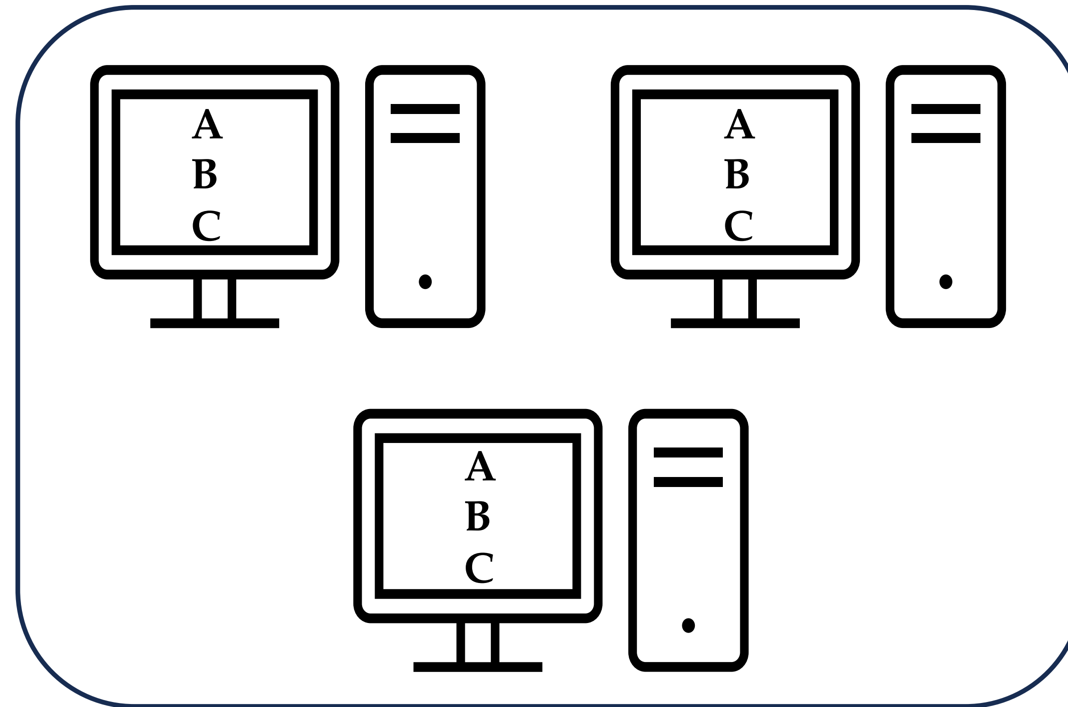
- Previously, we discussed briefly about replication.

- What are the **benefits** of replication?
  - Availability!

- What is the key **challenge** for replication?
  - Consistency

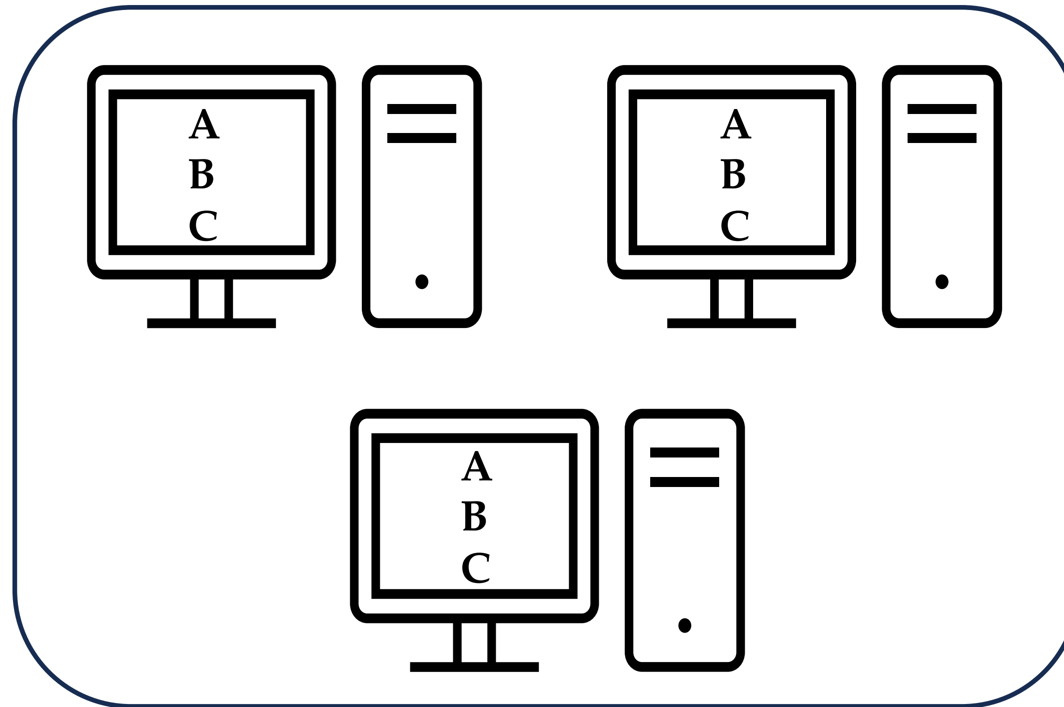
# Replication Terminology

- Recall that in a replicated database, we have multiple copies of each data item.
- Specifically, multiple nodes act as replicas and store the same data.



# Replication Terminology

- Recall that in a replicated database, we have multiple copies of each data item.
- Specifically, multiple nodes act as replicas and store the same data.




- From a client's perspective, there exists only a single copy of each data item; client does not observe replication semantics → **one-copy equivalence**.

# Execution Challenge

- Till now, we focused on sharded/partitioned systems.

- If a transaction accesses a data item, that item will be present at a specific partition.
  - Only that specific partition will execute the data item.

# Execution Challenge

- Till now, we focused on sharded/partitioned systems.

- If a transaction accesses a data item, that item will be present at a specific partition.
  - Only that specific partition will execute the data item.

- In a replicated system, which replica stores, executes and updates the data item?

# Execution Challenge

- Till now, we focused on sharded/partitioned systems.

- If a transaction accesses a data item, that item will be present at a specific partition.
  - Only that specific partition will execute the data item.

- In a replicated system, which replica stores, executes and updates the data item?
  - Store → every replica.
  - Update → every replica.
  - Execute → Generally, every replica, but only one can also execute and relay the results to others (may not work under failures)!

# Design Decisions for Replication

- **Consistency:**
  - **Strongly Consistent replicas** → Replicas that are mutually consistent after every update.
  - **Weakly Consistent replicas** → There is an allowed delay in making replicas mutually consistent.

# Design Decisions for Replication

- **Updates Priority:**
  - **Centralized** → Replicas are split into primary (master) and backup (slave).
    - Every update is performed first on the primary replica and asynchronously trickled down to backups.
    - **Primary replica** vs. **primary copy** for each data item, so no single primary.

# Design Decisions for Replication

- **Updates Priority:**
  - **Centralized** → Replicas are split into primary (master) and backup (slave).
    - Every update is performed first on the primary replica and asynchronously trickled down to backups.
    - **Primary replica** vs. **primary copy** for each data item, so no single primary.
  - **Distributed** → Any replica can perform update.
    - One possible design → allow all replicas to perform updates in parallel.

# Design Decisions for Replication

- **Updates Priority:**
  - **Centralized** → Replicas are split into primary (master) and backup (slave).
    - Every update is performed first on the primary replica and asynchronously trickled down to backups.
    - **Primary replica** vs. **primary copy** for each data item, so no single primary.
  - **Distributed** → Any replica can perform update.
    - One possible design → allow all replicas to perform updates in parallel.
    - **Lazy propagation** vs. **Eager propagation** of updates.

# Eager Propagation Protocols

# Eager Propagation Protocols

- Perform all the updates within the context of the transaction that initiated the write operations.
- When the transaction commits, its updates should have applied to all the replicas.

# Eager Propagation Protocols

- Perform all the updates within the context of the transaction that initiated the write operations.
- When the transaction commits, its updates should have applied to all the replicas.
- Enforces strong mutual consistency among the replicas.

# Eager Propagation Protocols

- Perform all the updates within the context of the transaction that initiated the write operations.
- When the transaction commits, its updates should have applied to all the replicas.
- Enforces strong mutual consistency among the replicas.
- Essentially, read one, write all (ROWA) protocols.
  - Sufficient to read from just one replica as all the replicas have same state.
- What is the **major disadvantage**?

# Eager Propagation Protocols

- Perform all the updates within the context of the transaction that initiated the write operations.
- When the transaction commits, its updates should have applied to all the replicas.
- Enforces strong mutual consistency among the replicas.
- Essentially, read one, write all (ROWA) protocols.
  - Sufficient to read from just one replica as all the replicas have same state.
- What is the **major disadvantage**?
  - High response time for the client → Update all replicas before sending a response to the client.
  - If a replica has fails or crashes → Challenge to terminate the protocol.

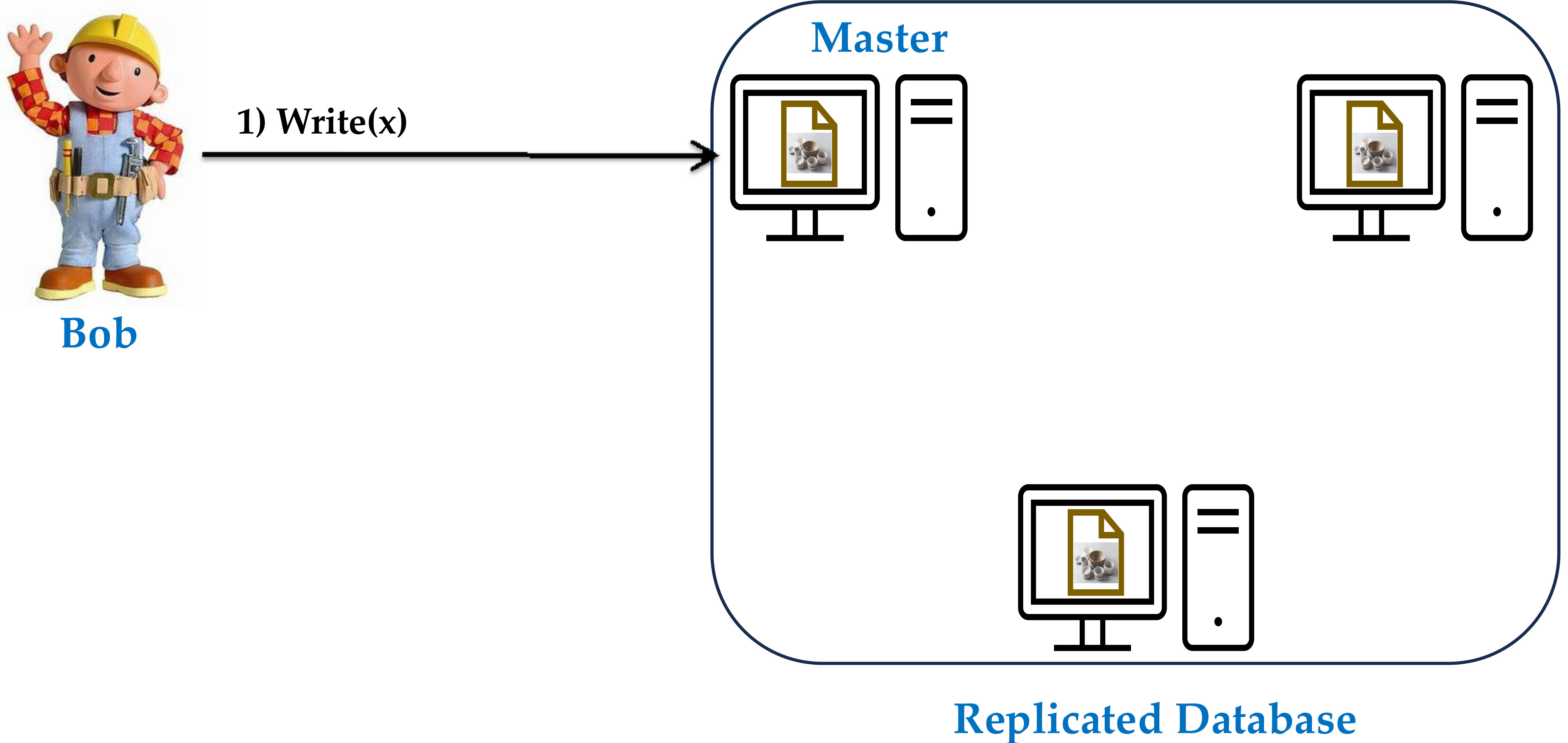
# Lazy Propagation Protocols

# Lazy Propagation Protocols

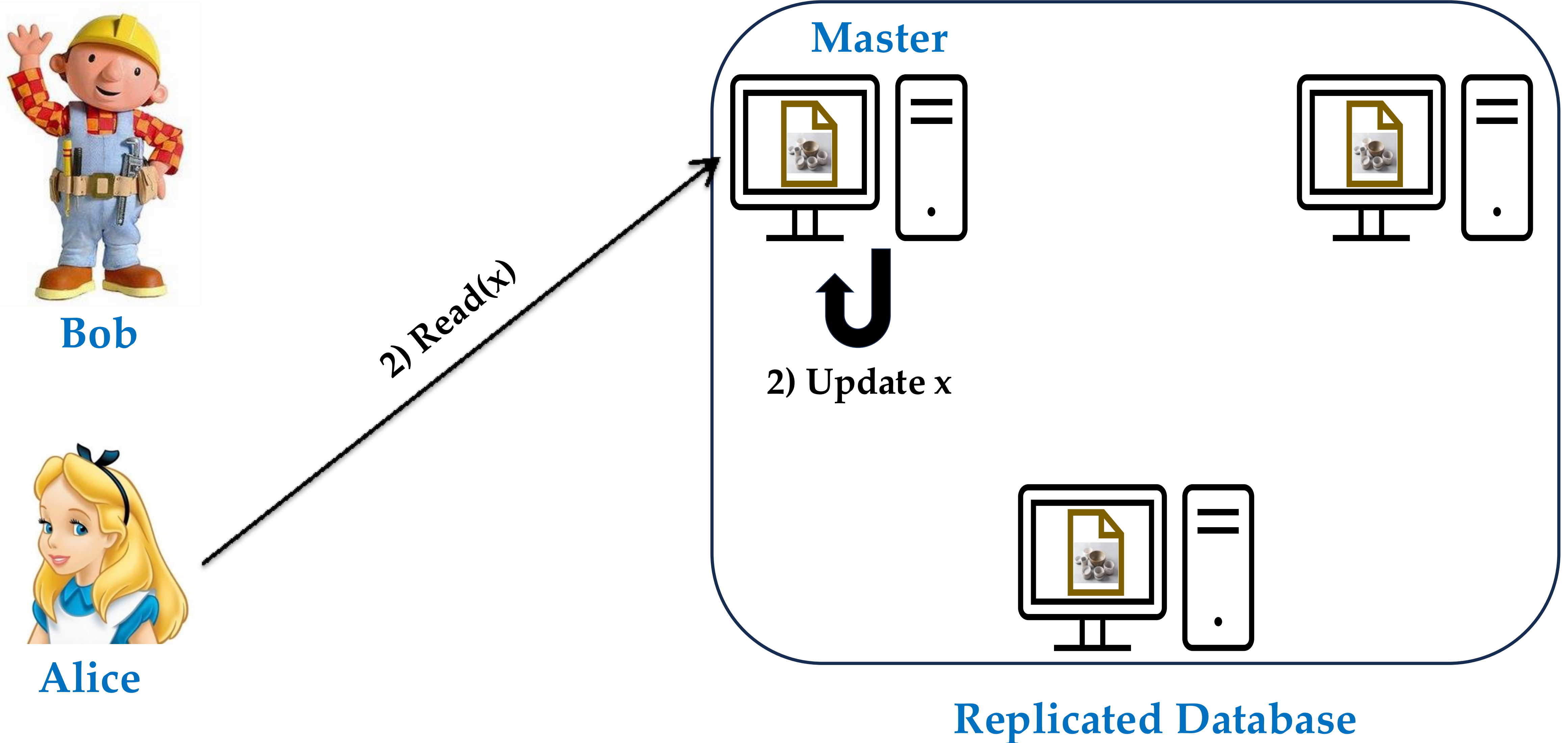
- The transaction does not need to wait for its updates to be applied to all the replicas.
- Transaction commits as soon as one replica is updated.
- Other replicas are updated asynchronously.
- Low response time but replicas are not mutually consistent!

# Let’s look at some Replication Designs

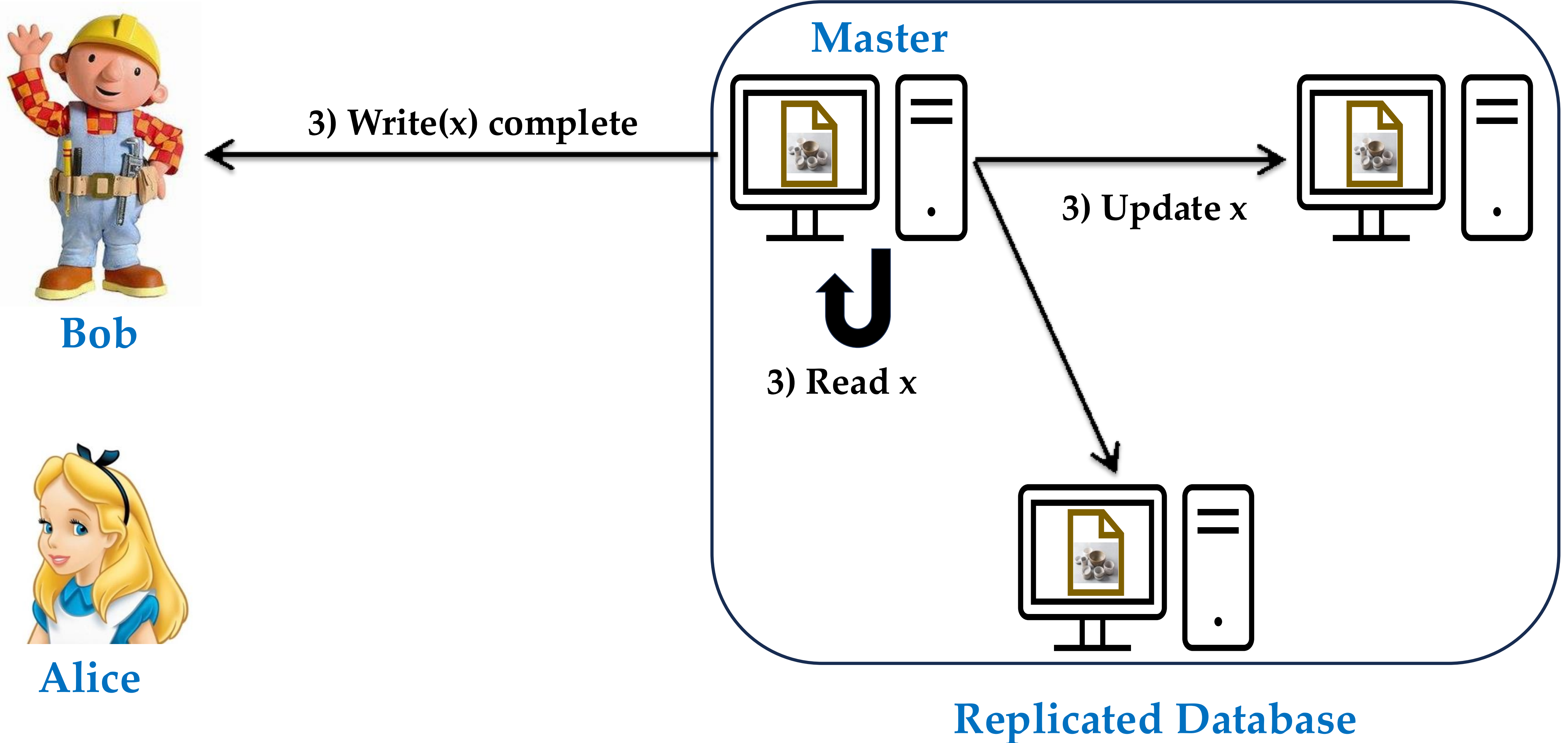
# Single Master with Lazy Replication (I)



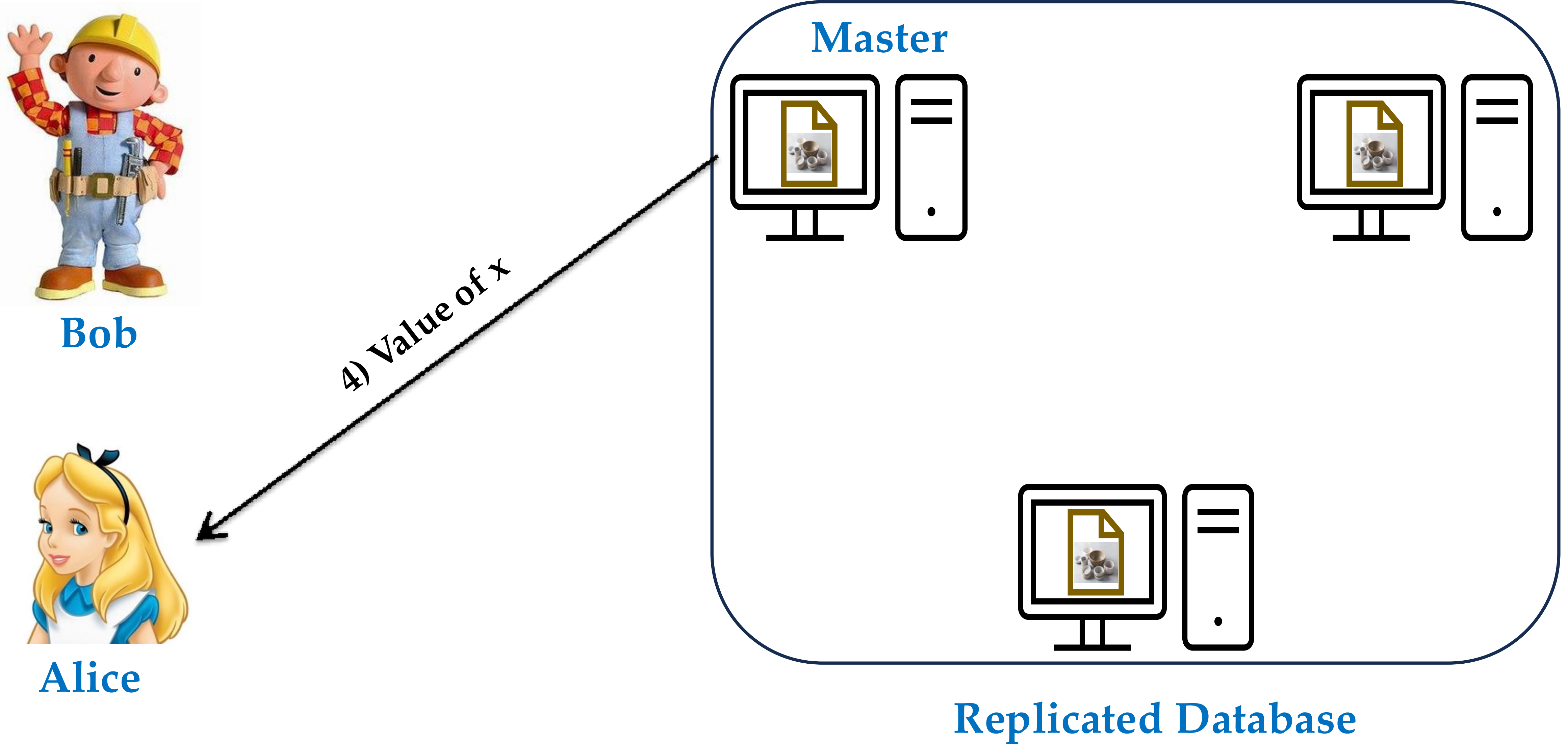
# Single Master with Lazy Replication (I)



# Single Master with Lazy Replication (I)



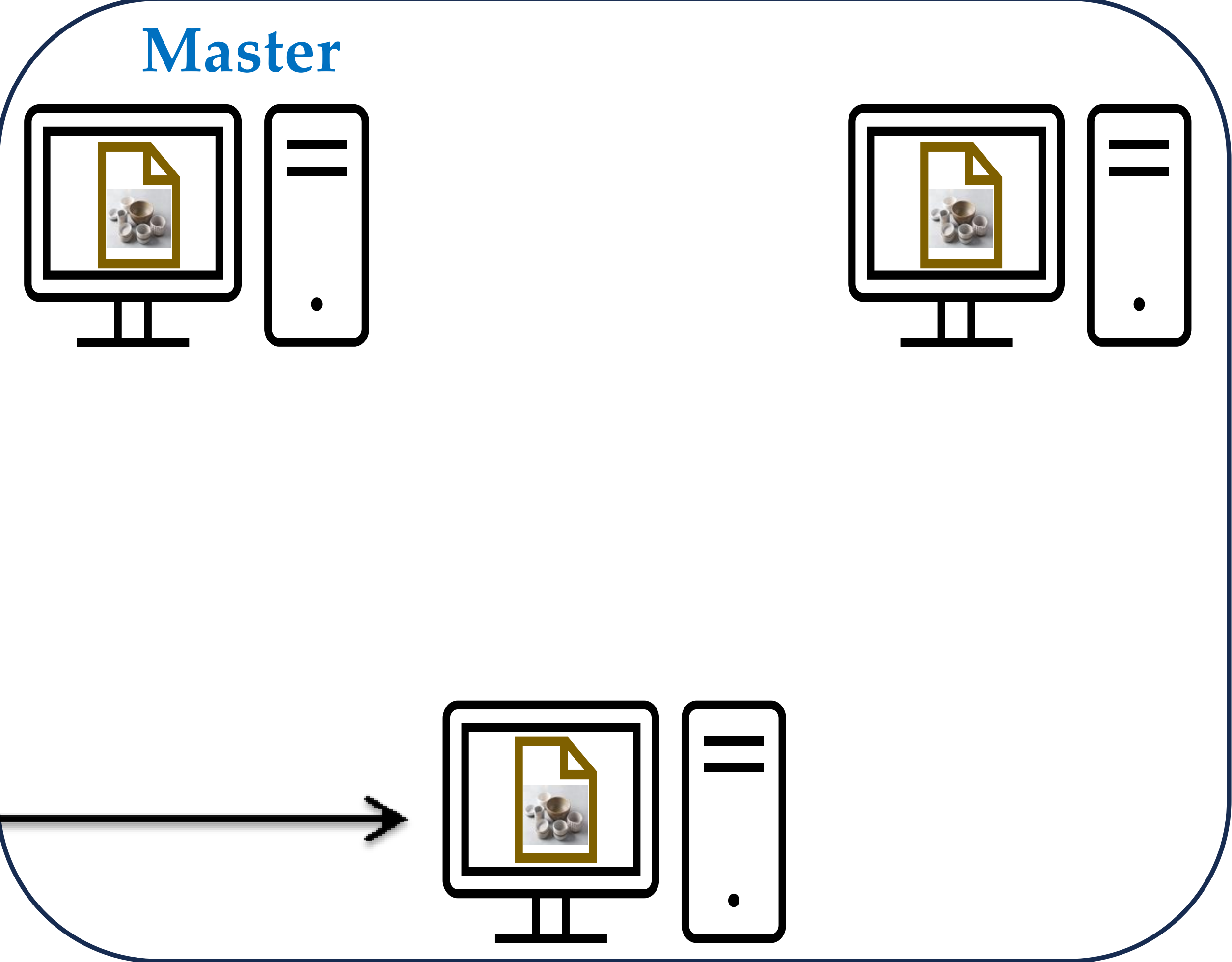
# Single Master with Lazy Replication (I)



# Single Master with Lazy Replication (II)

Bob

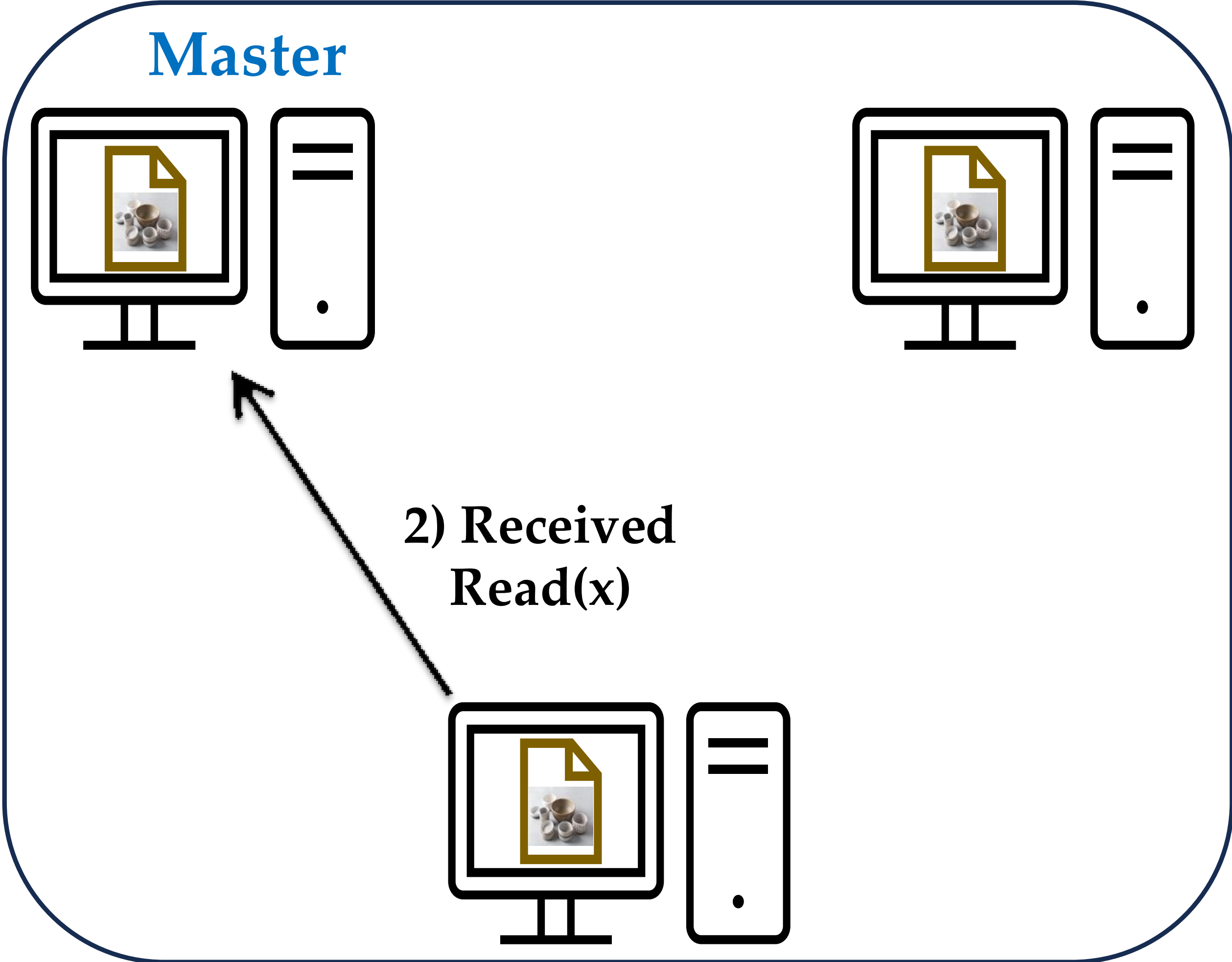
Alice



# Single Master with Lazy Replication (II)

Bob

Alice

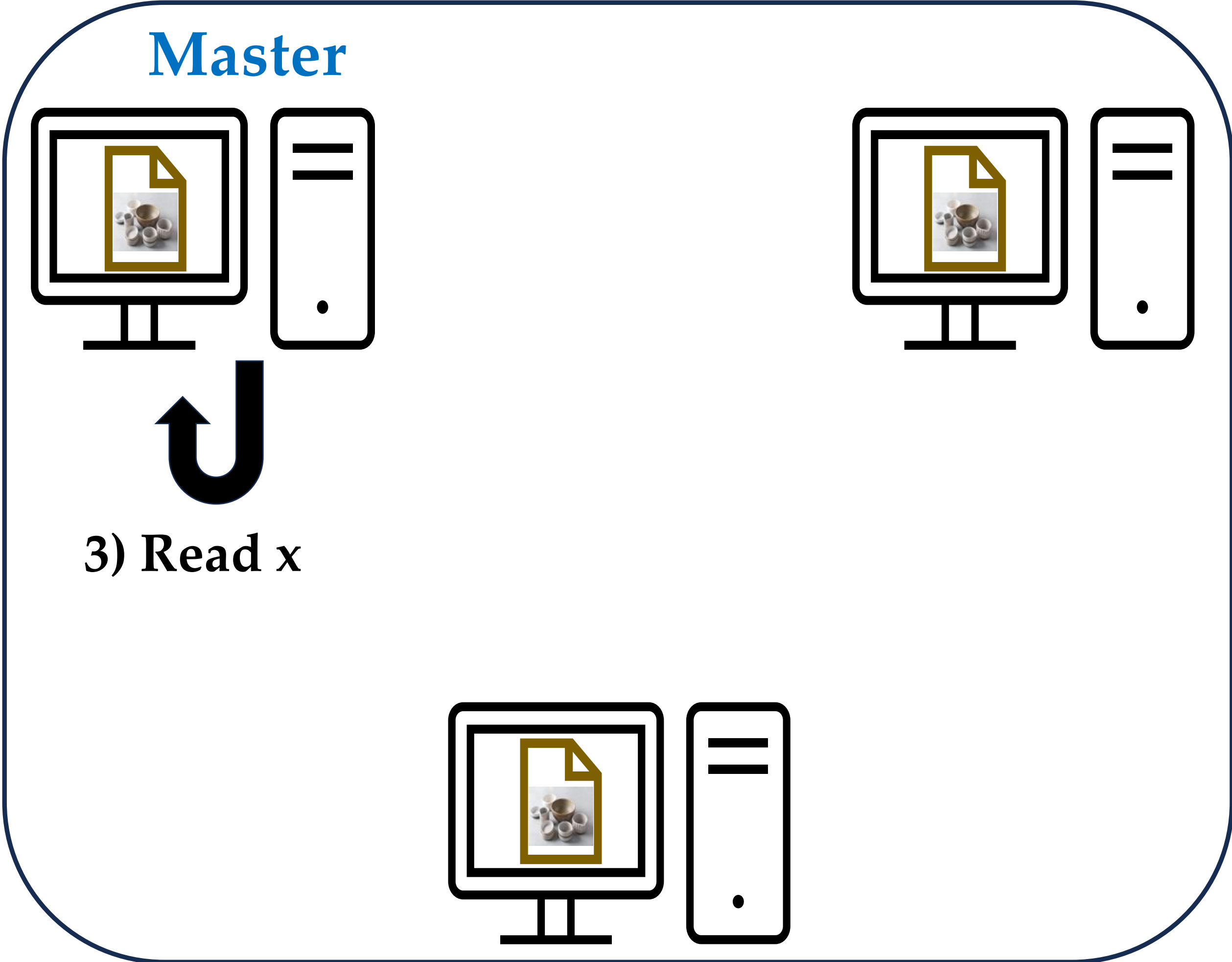



Replicated Database

# Single Master with Lazy Replication (II)

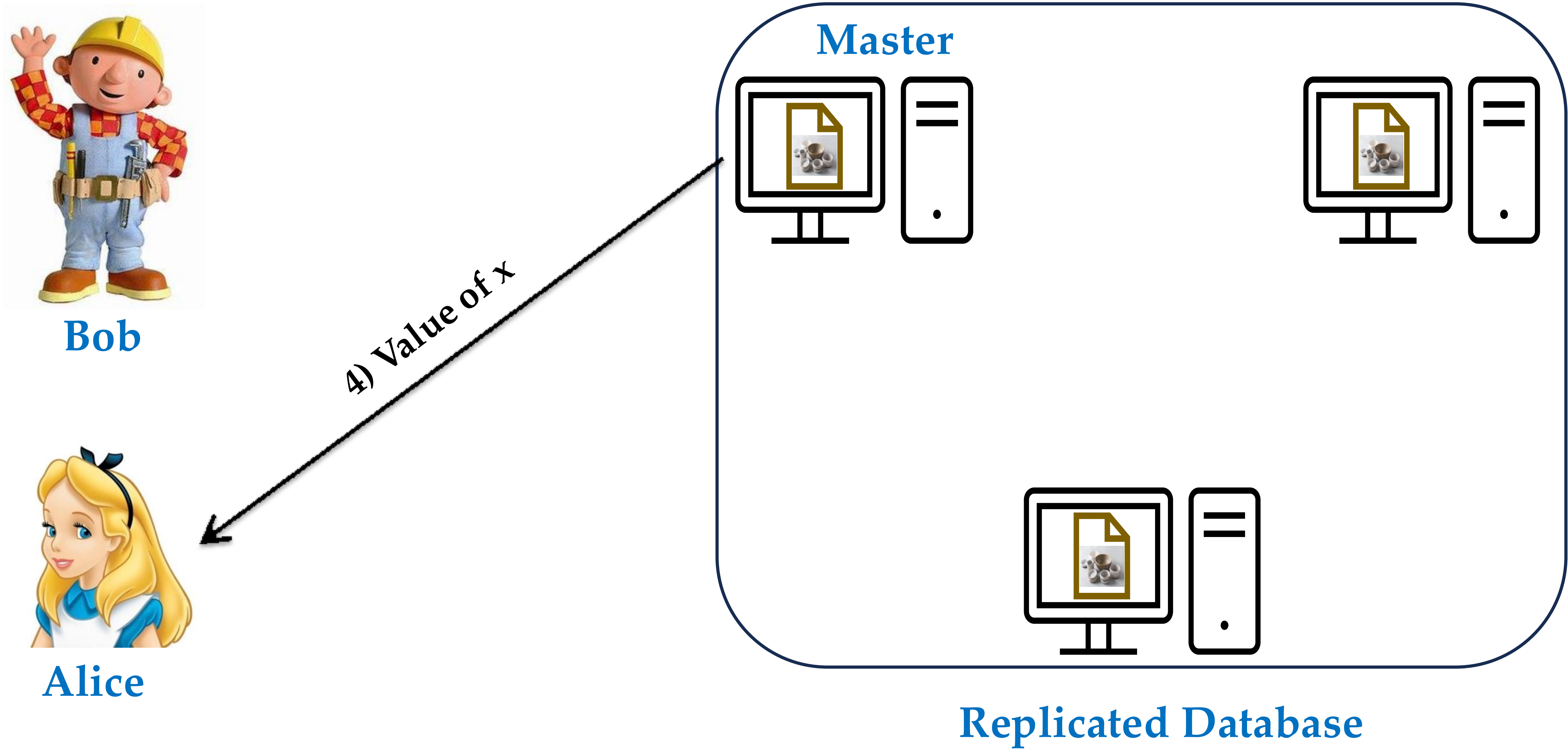
Bob

Alice




Replicated Database

# Single Master with Lazy Replication (II)



# Single Master with Lazy Replication (III)

Bob

Alice

1) Read(x)

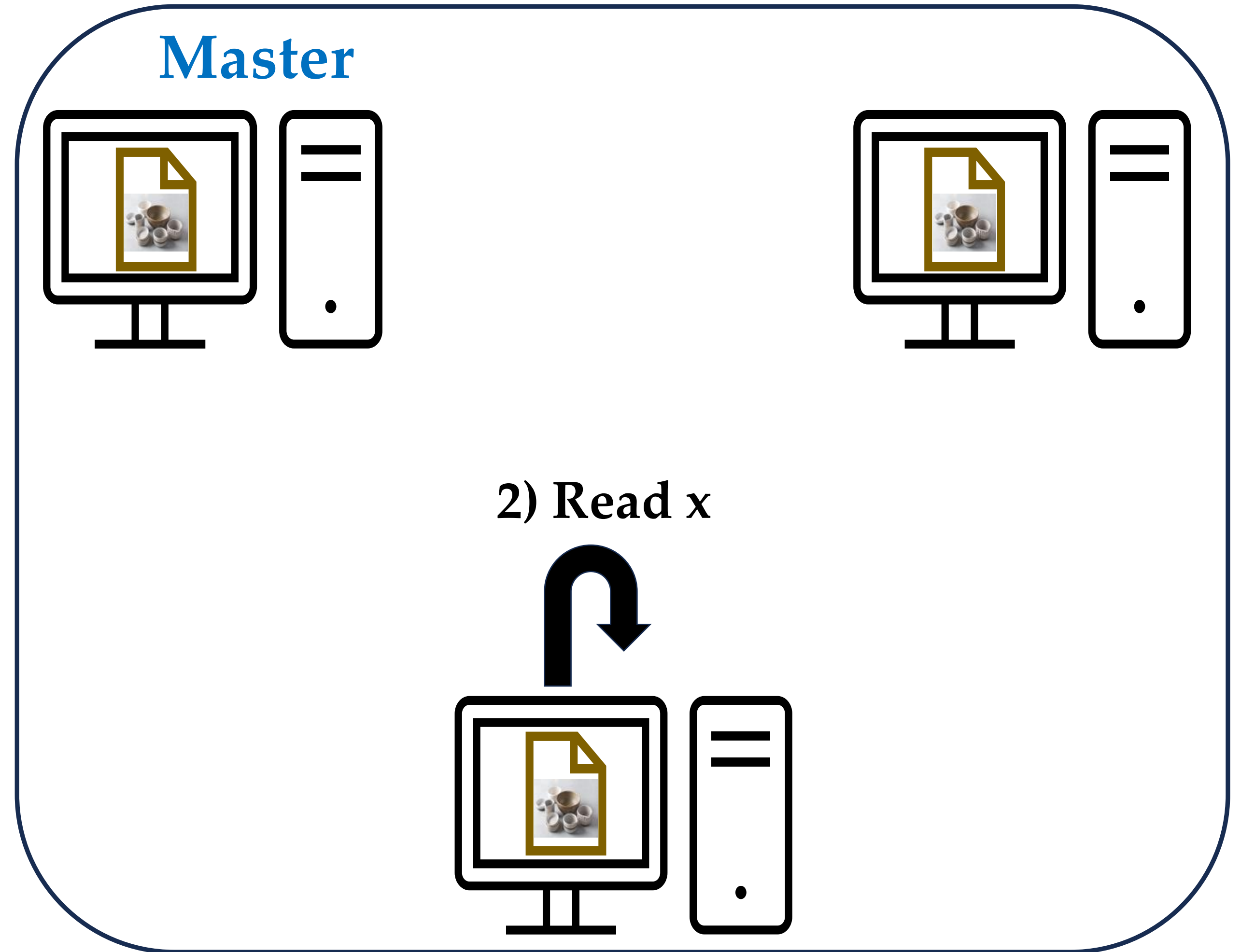
Master

Replicated Database

# Single Master with Lazy Replication (III)

Bob

Alice

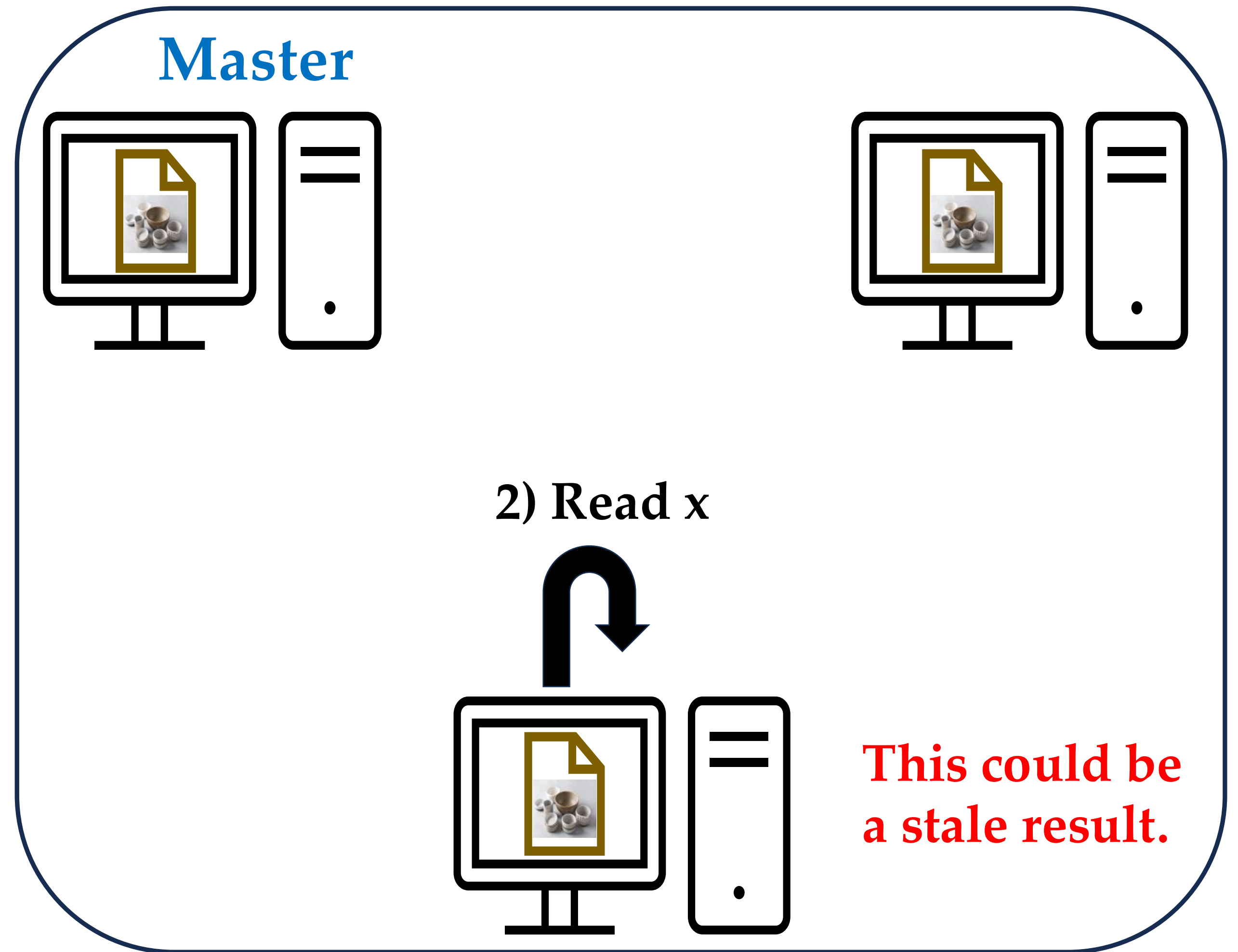



Replicated Database

# Single Master with Lazy Replication (III)

Bob

Alice



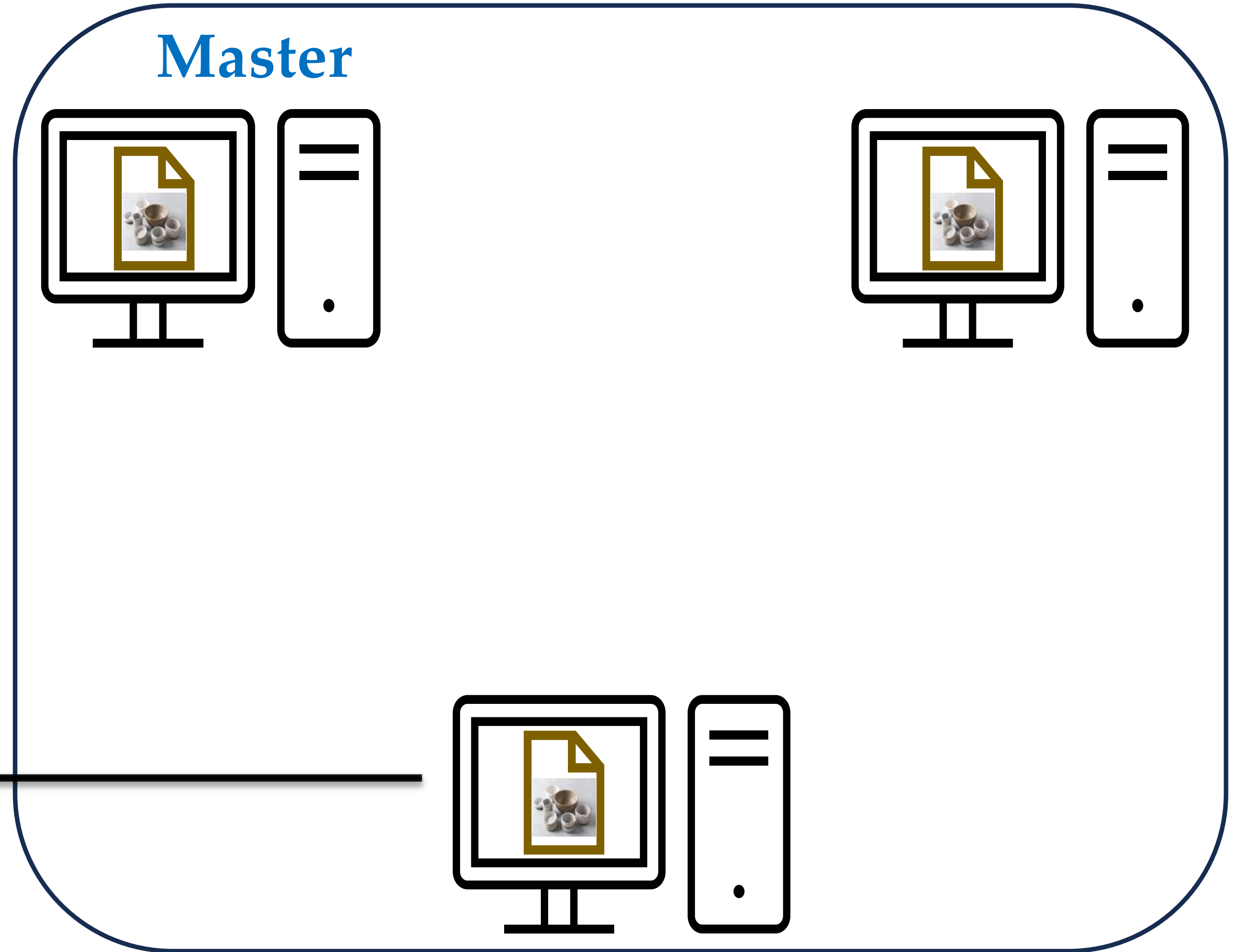

Replicated Database

# Single Master with Lazy Replication (III)

Bob

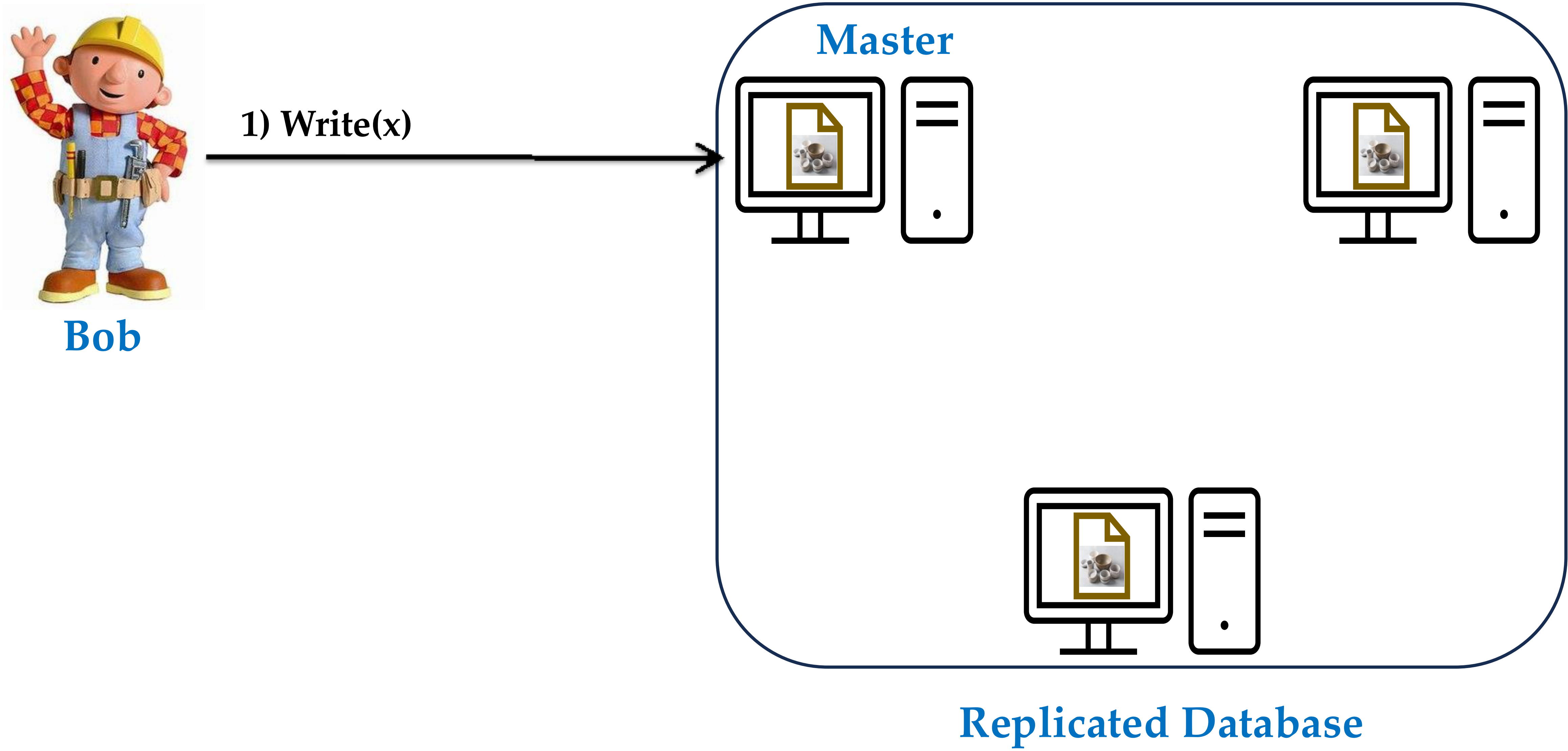
Alice

3) Value of x




Replicated Database

# Single Master with Eager Replication (I)



# Single Master with Eager Replication (I)

Bob

Alice

2) Read(x)

Master

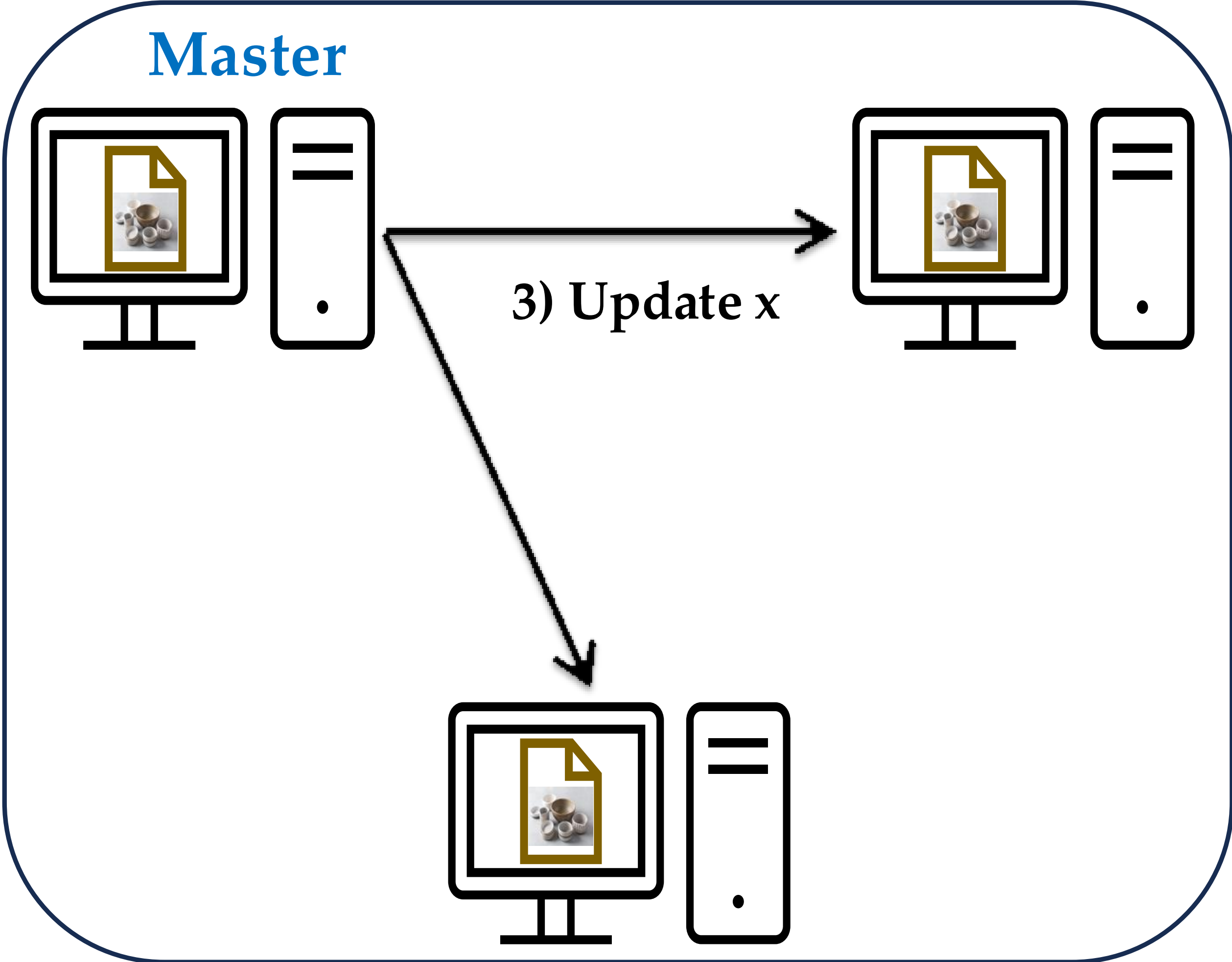
2) Update x

Replicated Database

# Single Master with Eager Replication (I)

Bob

Alice




Replicated Database

# Single Master with Eager Replication (I)

Bob

4) Write(x) complete

Master

4) Read x

Alice

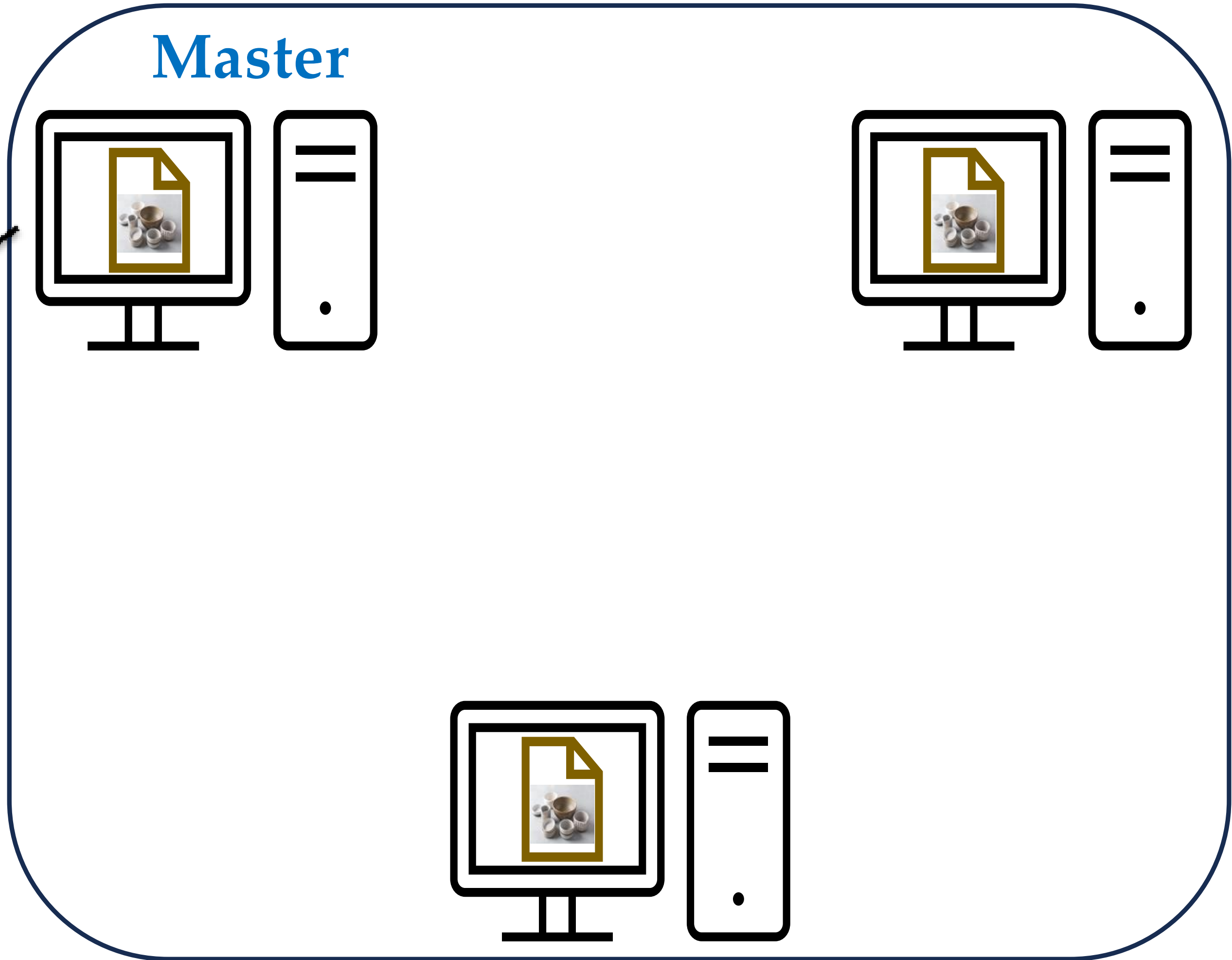
Replicated Database

# Single Master with Eager Replication (I)

Bob

5) Value of x

Alice




Replicated Database

# Single Master with Full Replication

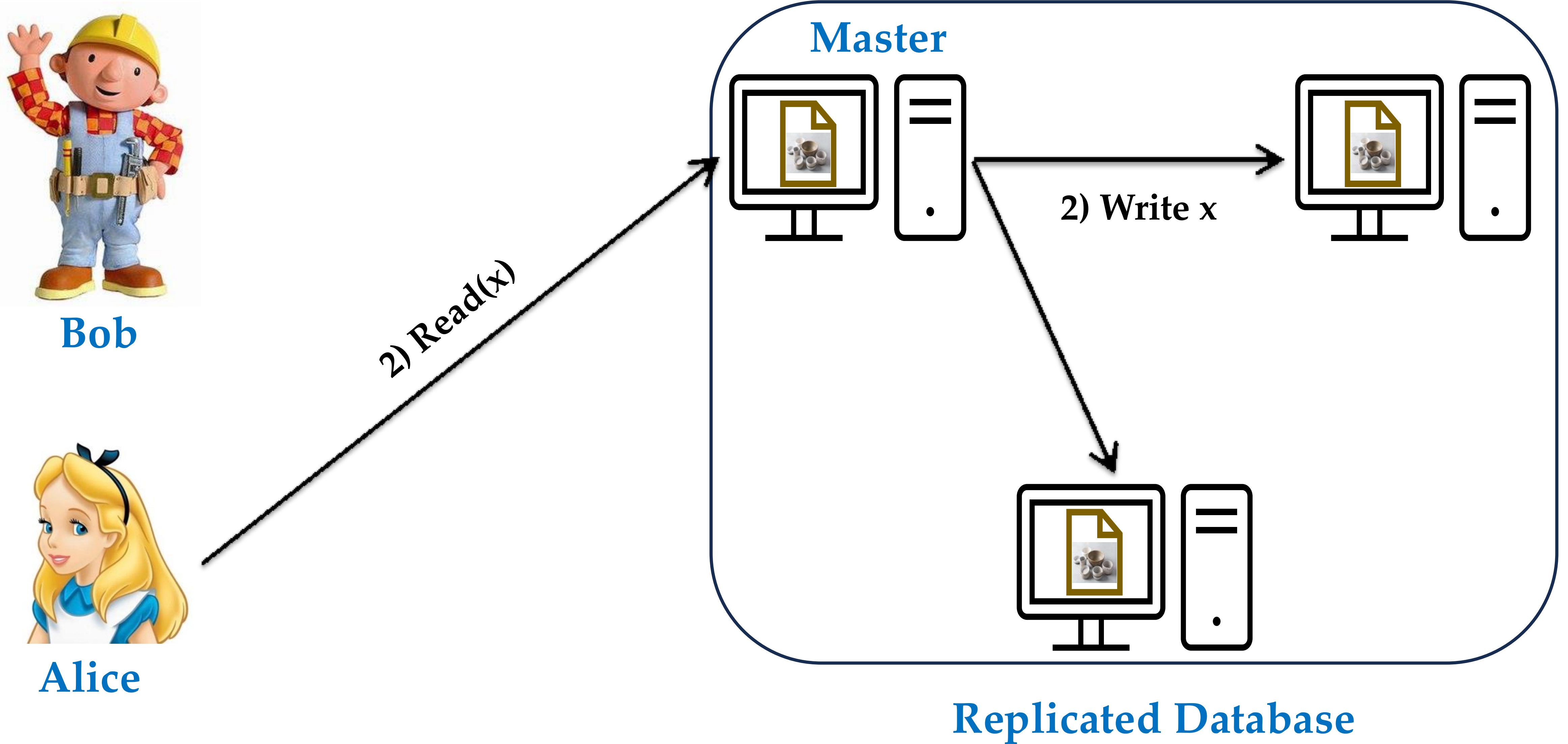
Bob

1) Write(x)

Master

Replicated Database

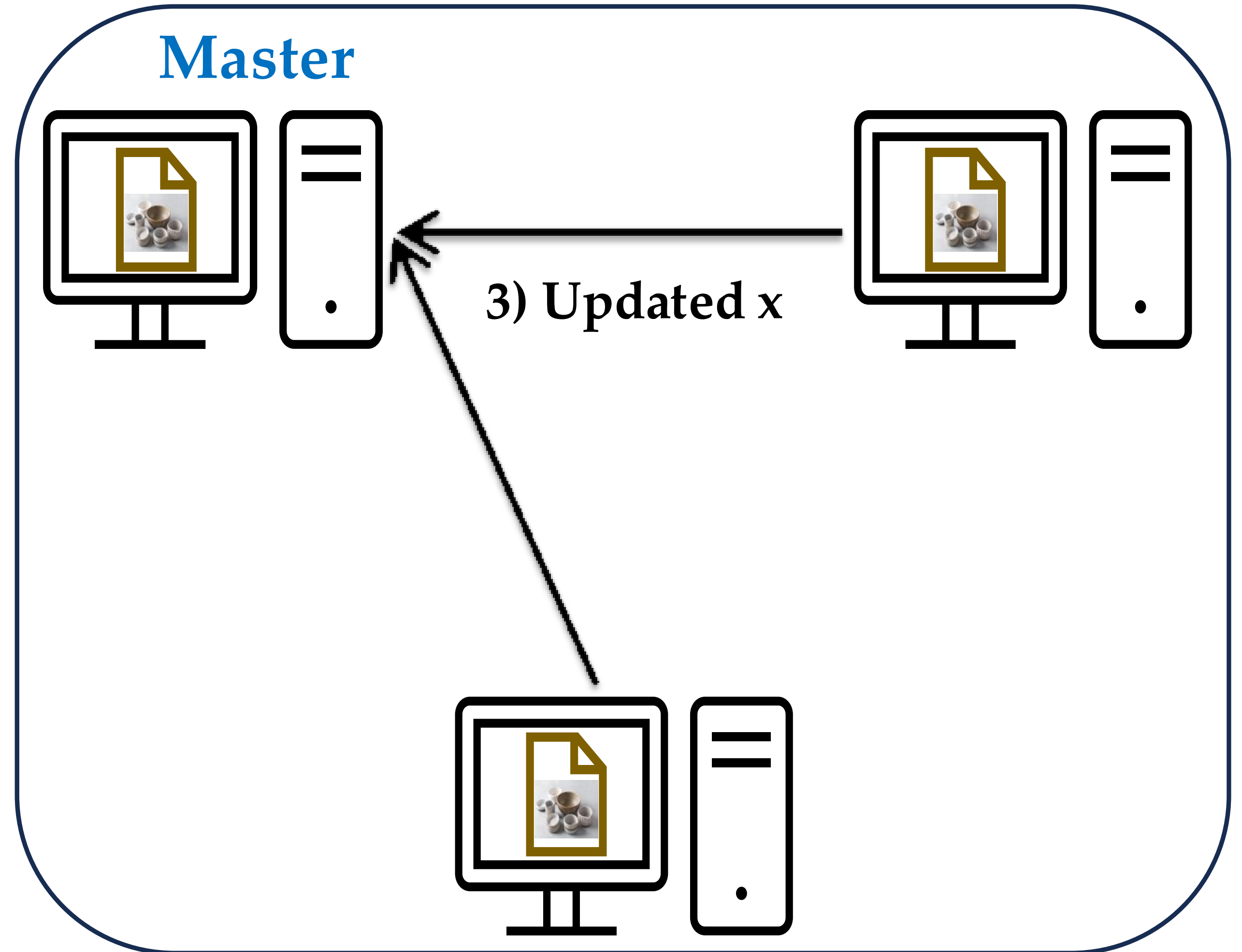
# Single Master with Full Replication



# Single Master with Full Replication

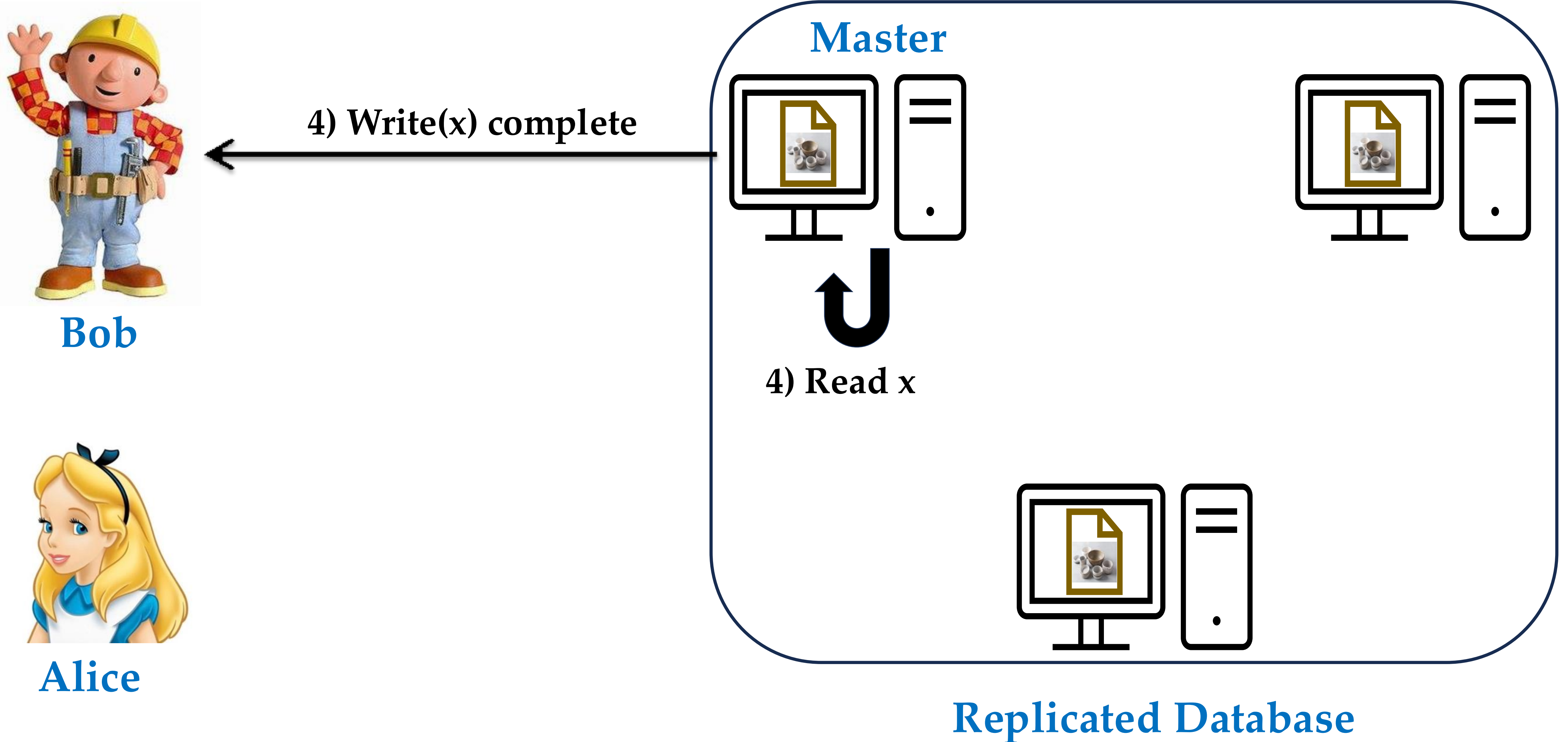
Bob

Alice




Replicated Database

# Single Master with Full Replication



# Single Master with Full Replication

Bob

5) Value of x

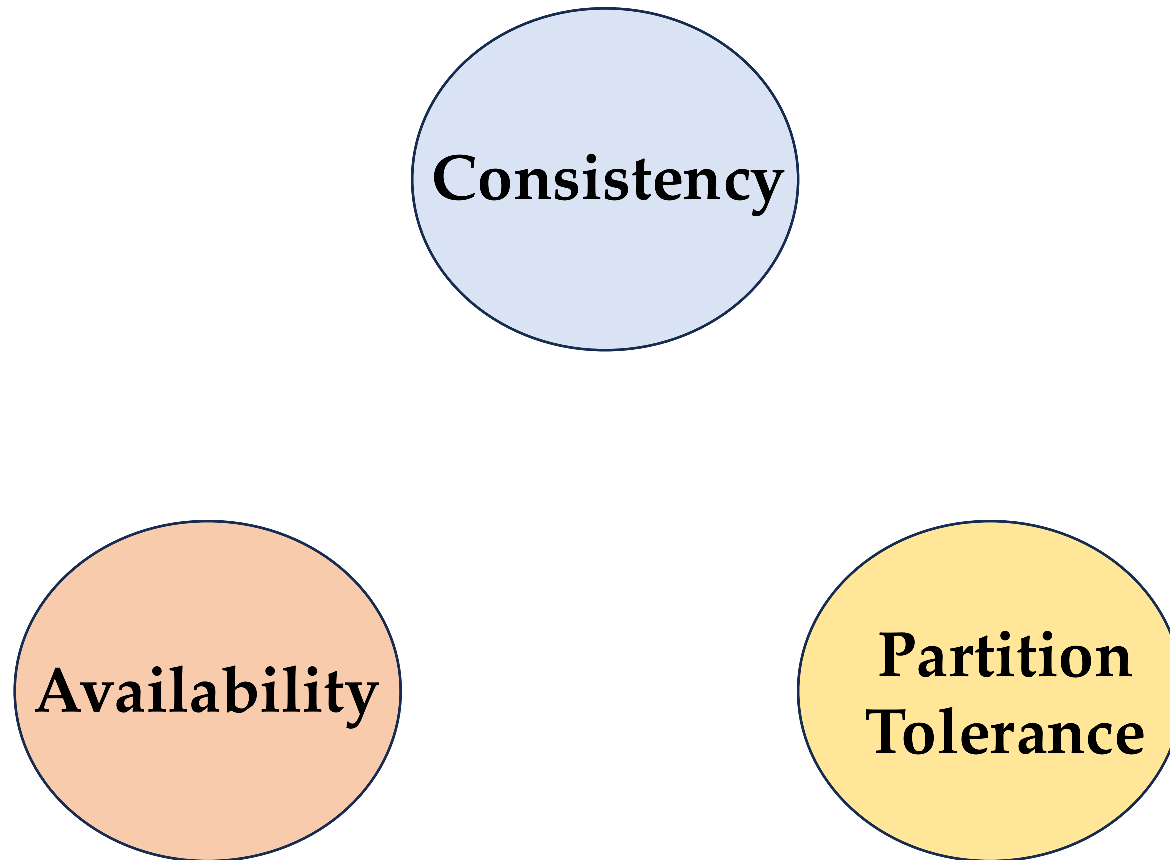
Alice

Master

Replicated Database

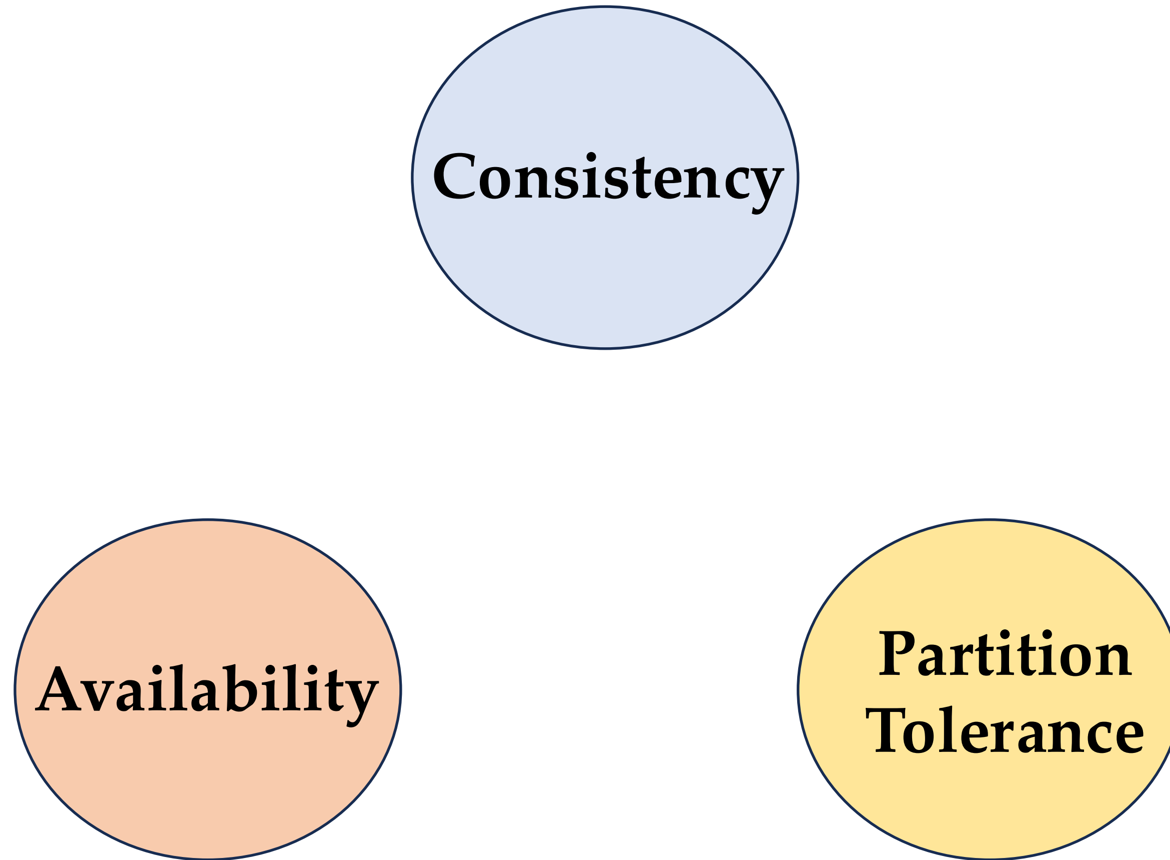
# CAP Theorem

# CAP Theorem




CAP Theorem states that a distributed system can deliver only two of the three desirable properties: consistency, availability, and partition tolerance.

# CAP Theorem




When you design a distributed system, you sacrifice something or make some assumptions to ensure your system can partially attain all these properties.

# Distributed Consistency

# Distributed Consistency Levels

- Next, we look at different types of distributed consistency levels.

- Based on the choice of consistency level we select, we fix the maximum performance our system can yield.

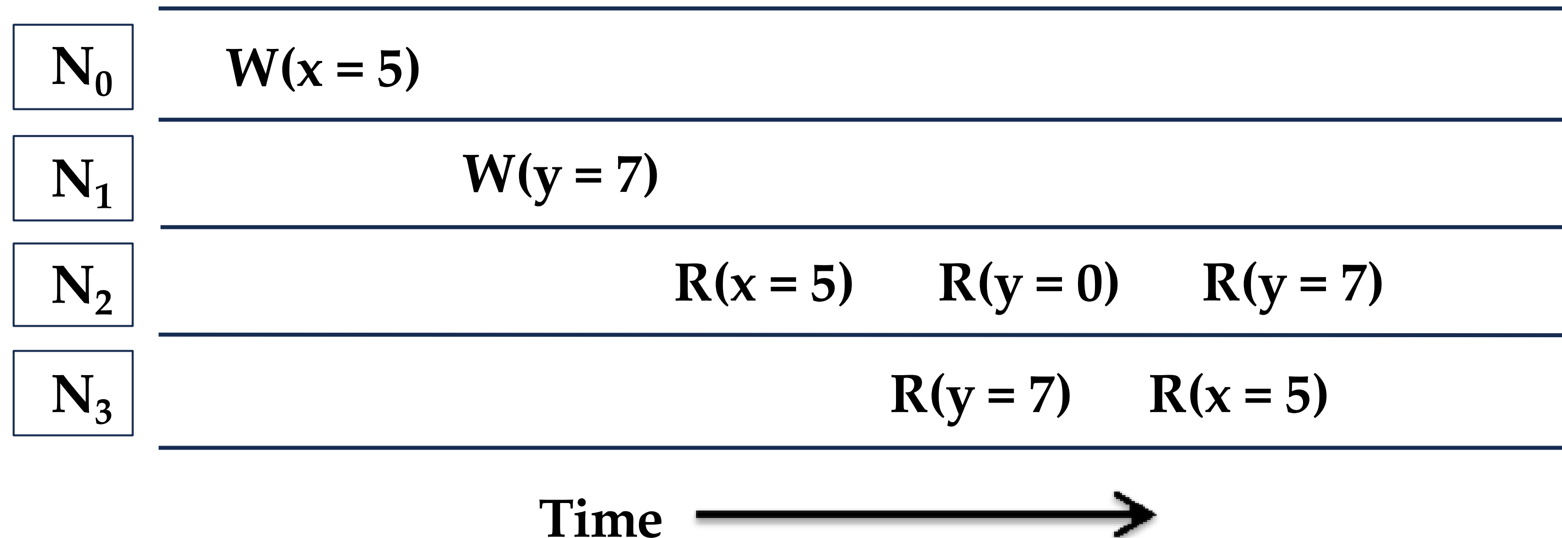
# Sequential Consistency

# Sequential Consistency

- All write operations are globally ordered.
    - No matter which thread/process/node makes the write and no matter which data item was written.
    - Ex: If a node first observes a write to a variable X and then observes a write to variable Y, then all the nodes should observe the same order.
- ***Note:*** While checking whether a schedule of transactions is sequentially consistent or not, you can reorder the operations of different nodes/processes but not operations of a single node.

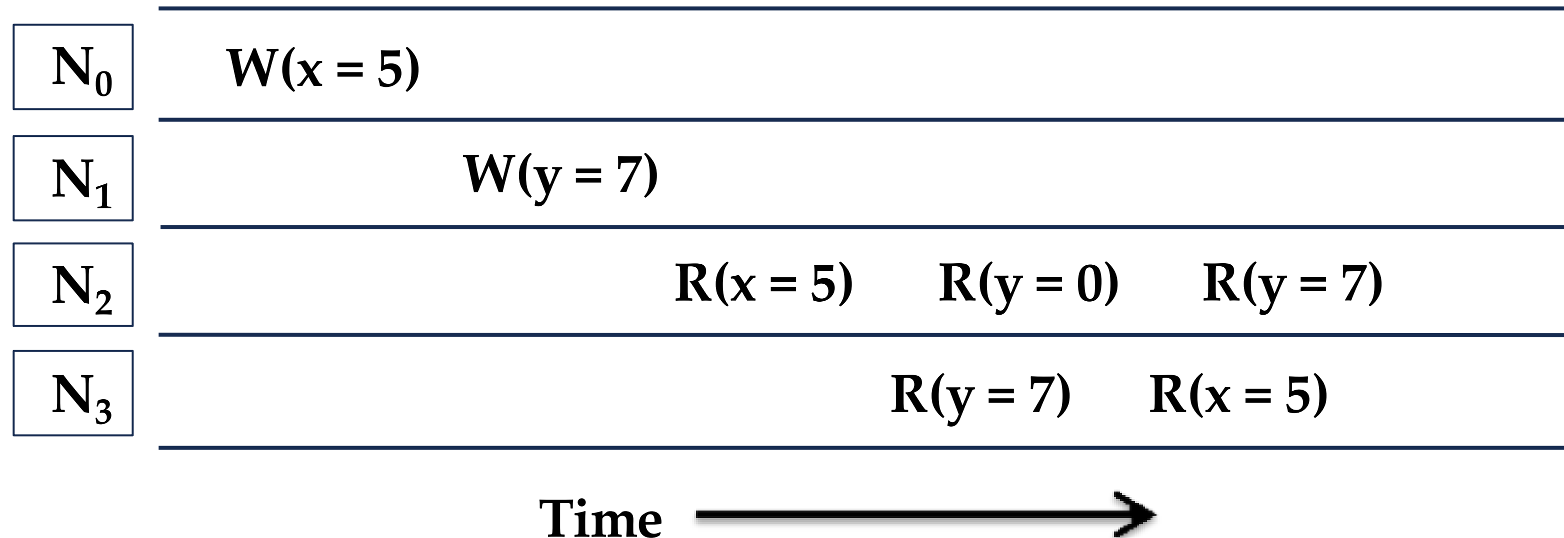
# Sequential Consistency (I)

Can we create a sequentially consistent schedule out of this?



# Sequential Consistency (I)

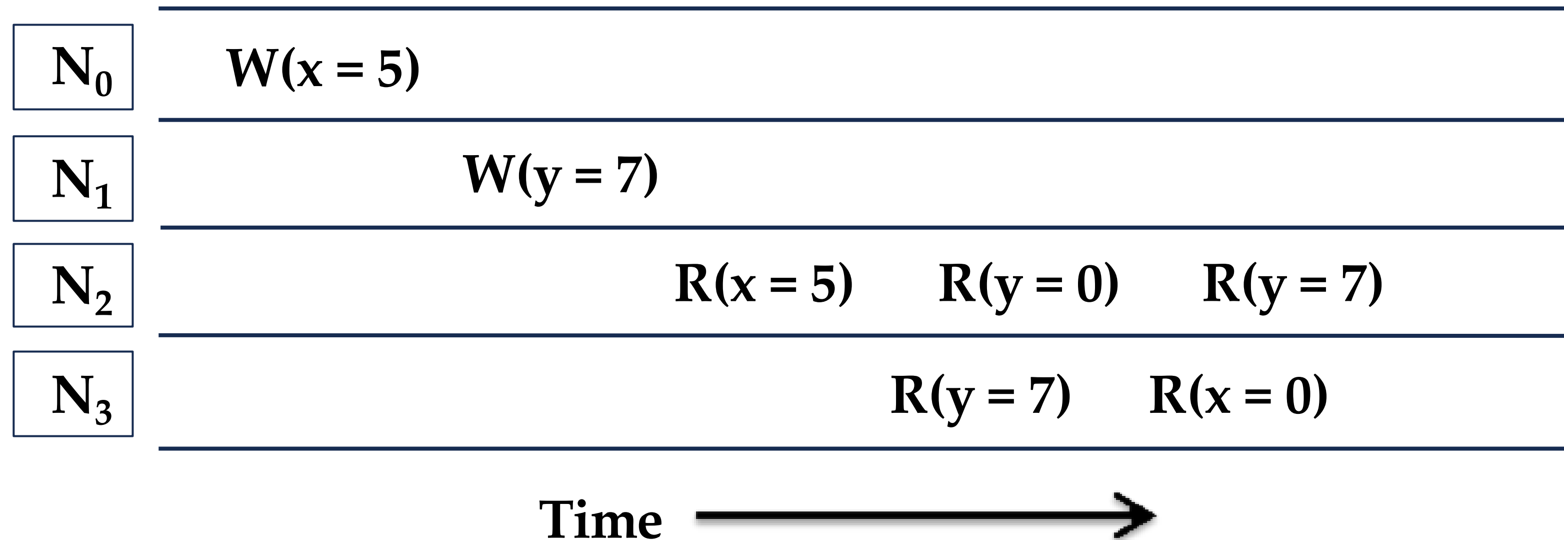
Can we create a sequentially consistent schedule out of this?




**Yes**, because in any way you reorder the nodes, if $N_2$ observes x=5 and y=7, then so does $N_3$.

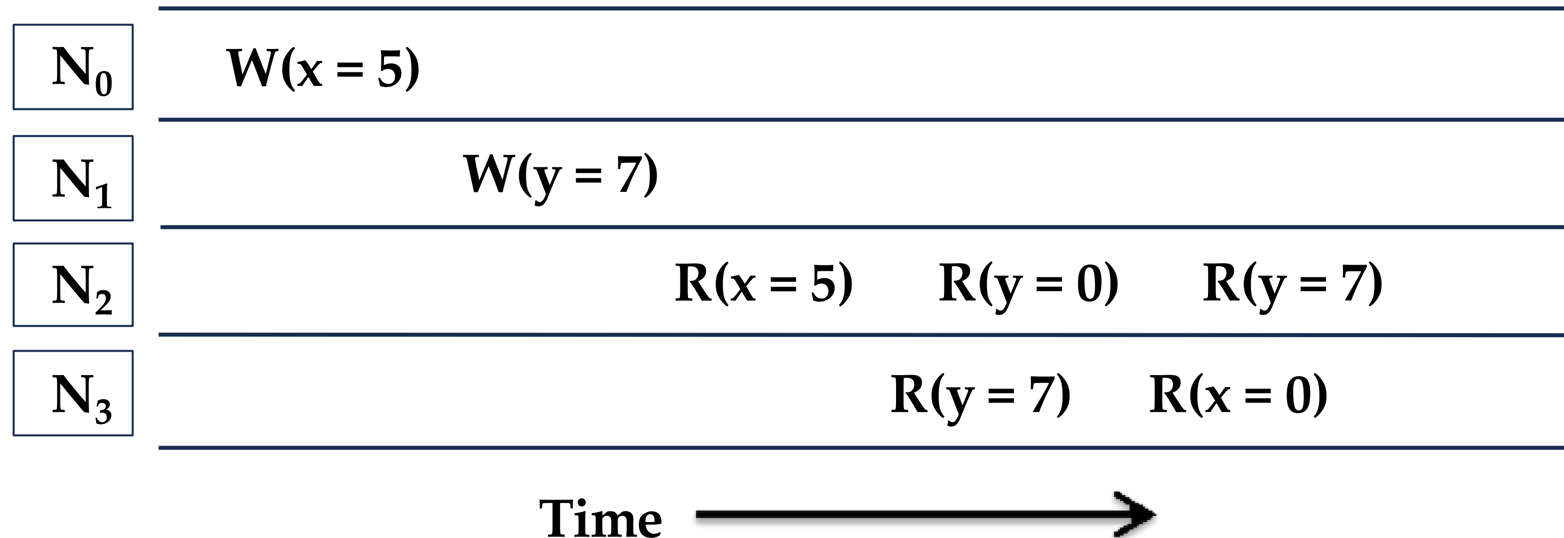
# Sequential Consistency (II)

Can we create a sequentially consistent schedule out of this?



# Sequential Consistency (II)

Can we create a sequentially consistent schedule out of this?

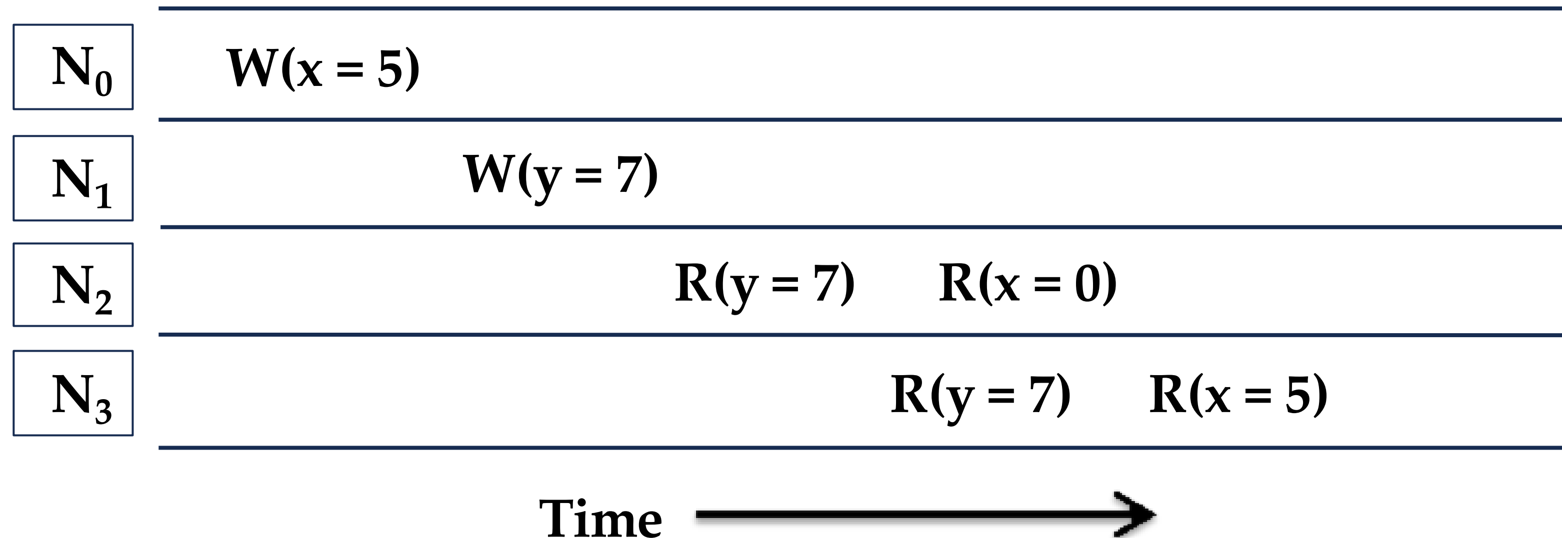



**No!**

# Sequential Consistency (III)

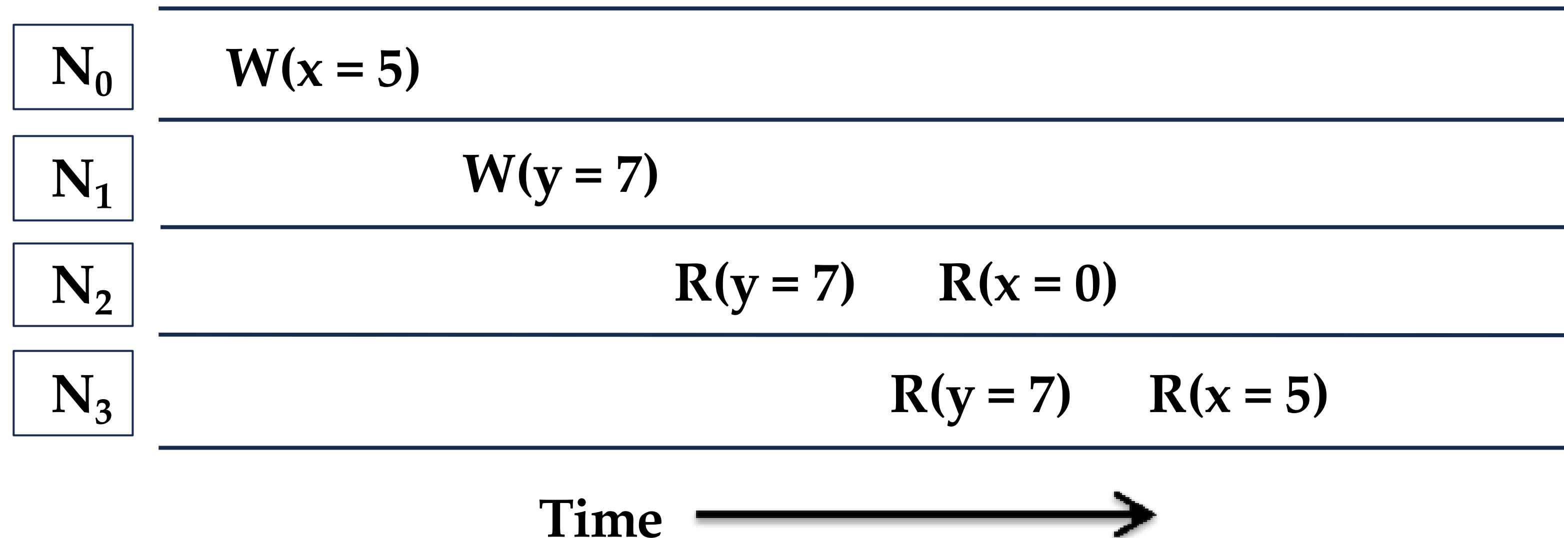
Can we create a sequentially consistent schedule out of this?

**Initially: x=0 and y=0**



# Sequential Consistency (III)

Can we create a sequentially consistent schedule out of this?




**Yes!**

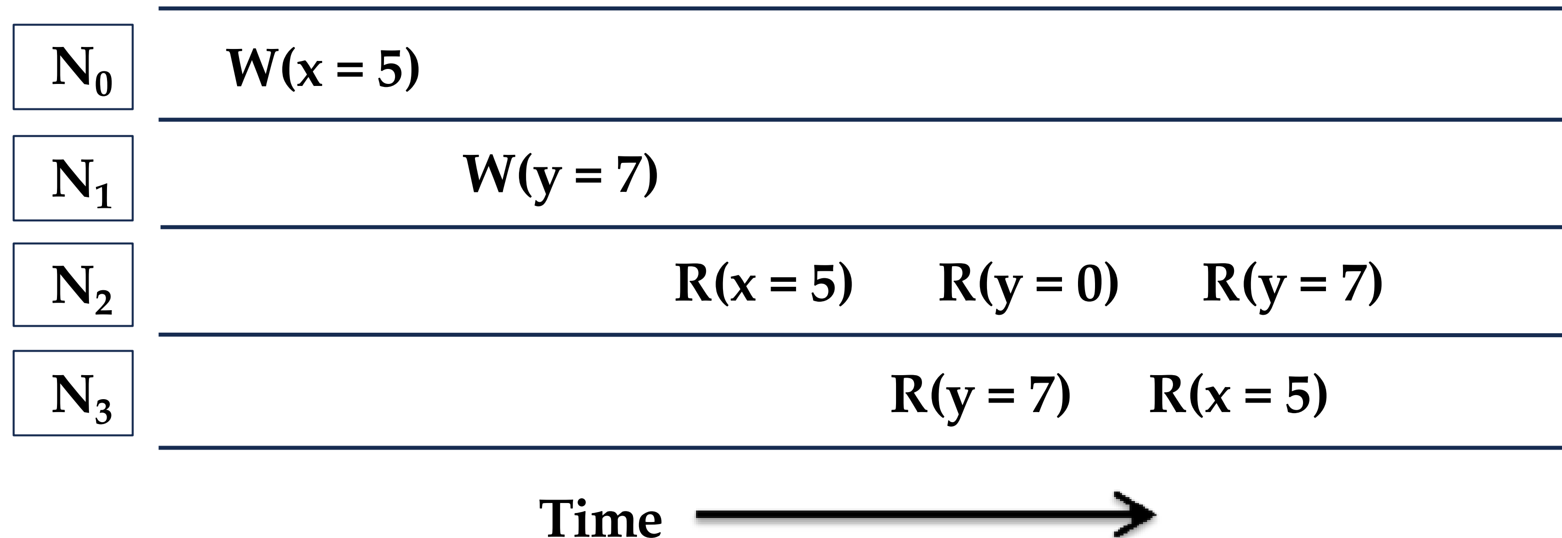
# Linearizability

# Linearizability

- Stricter and more restrictive than sequential consistency.
- Cares about time.
- The only complication is concurrency or overlapping operations.
- For non-overlapping operations, every read should return the last write.

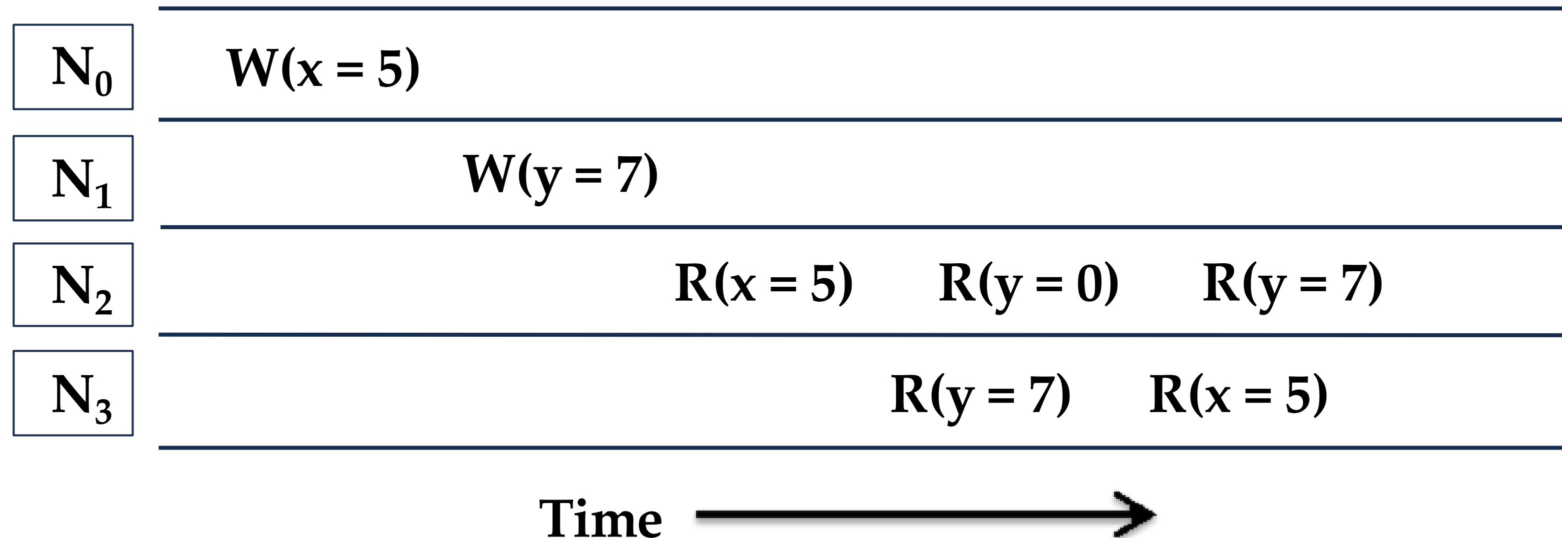
# Linearizability (I)

Can we create a linearizable schedule out of this?



# Linearizability (I)

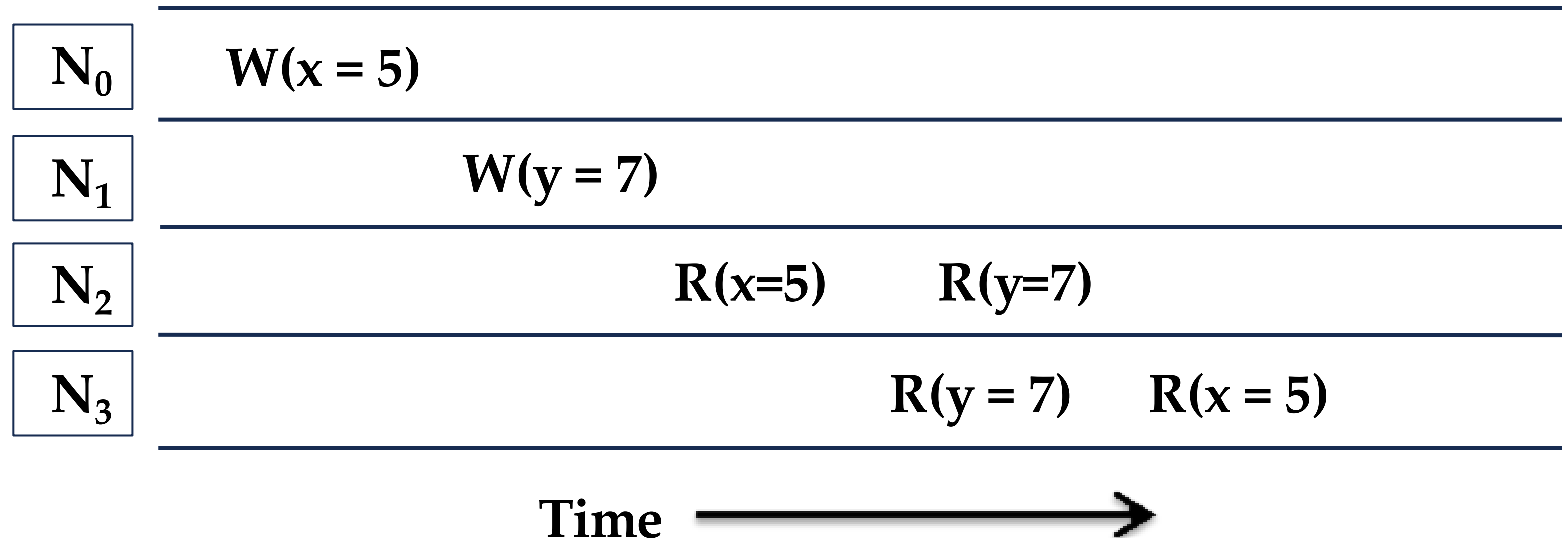
Can we create a linearizable schedule out of this?




**No**, because $N_2$ observes y=0 after $N_1$ has written y=7.

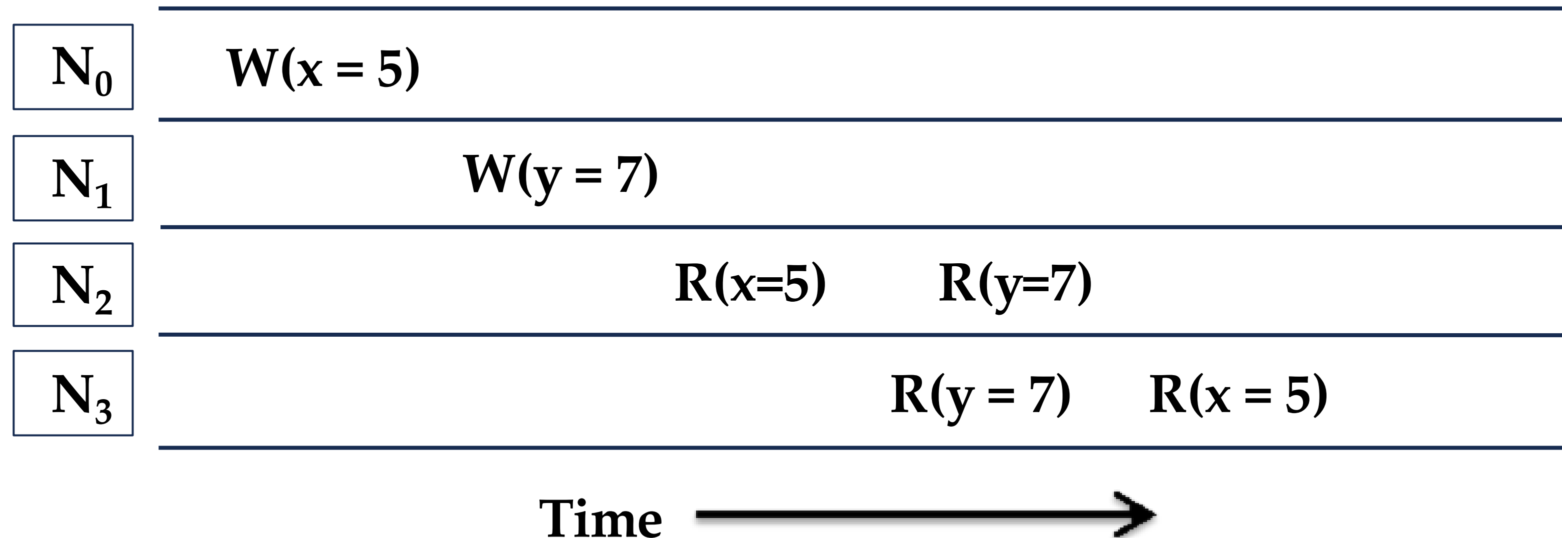
# Linearizability (II)

Can we create a linearizable schedule out of this?



# Linearizability (II)

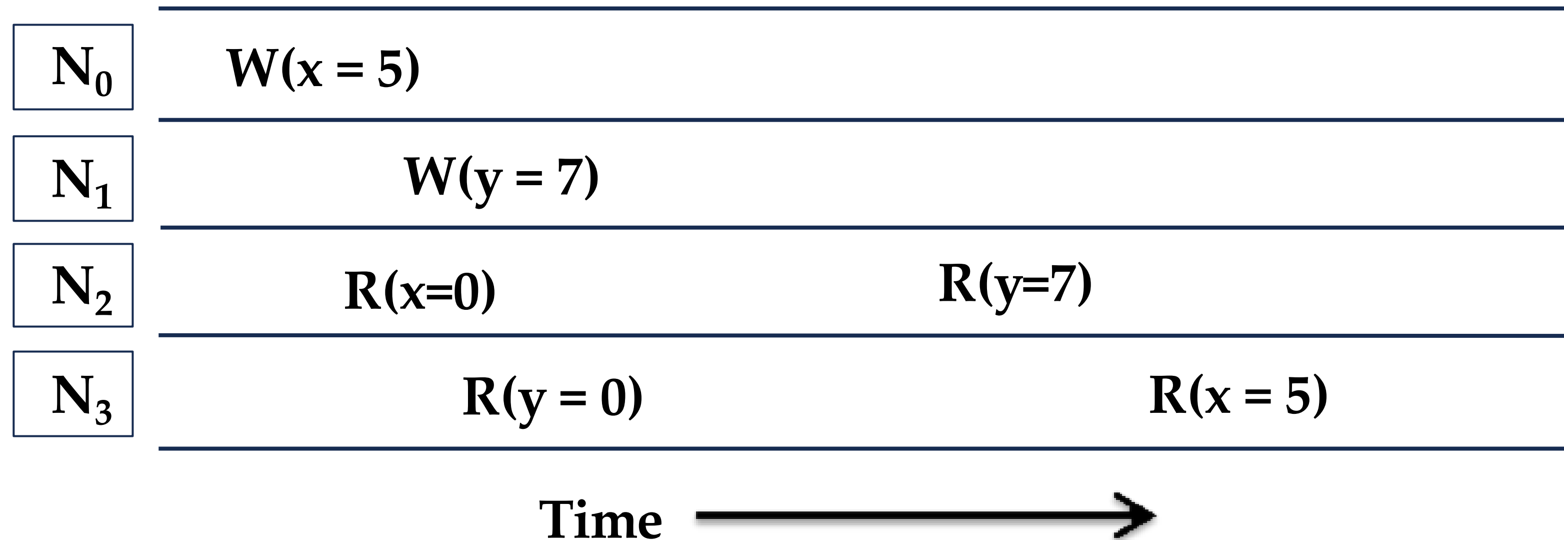
Can we create a linearizable schedule out of this?




**Yes!**

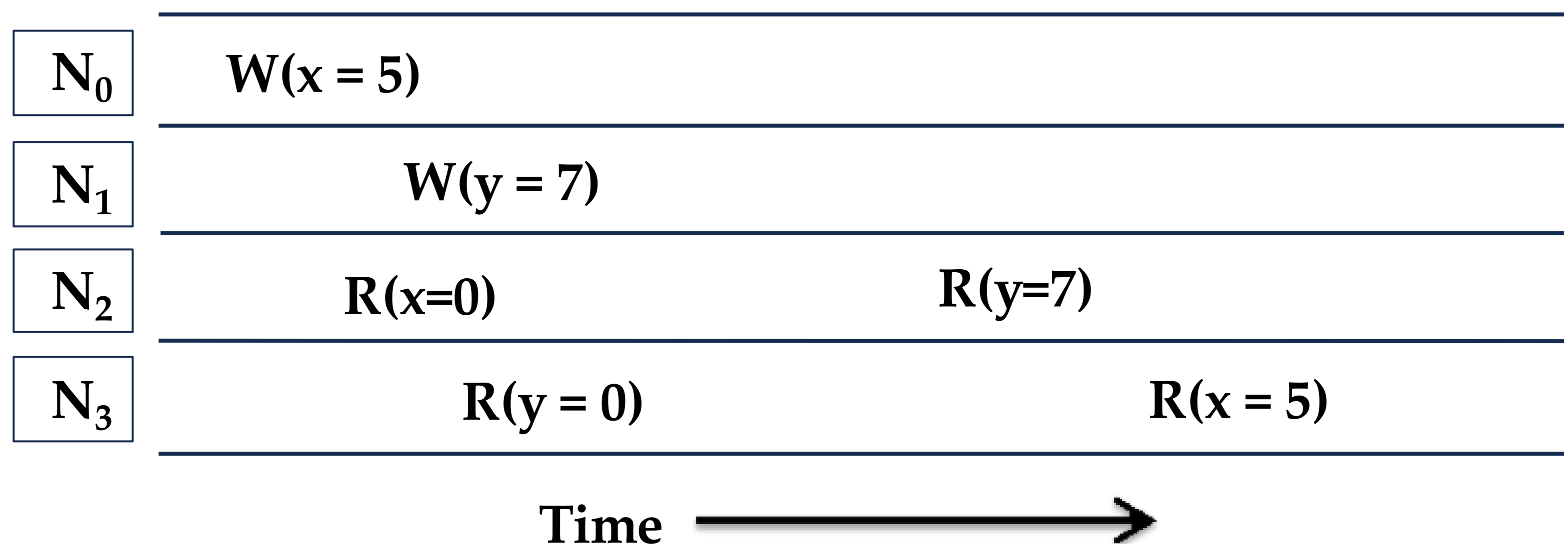
# Linearizability (III)

Can we create a linearizable schedule out of this?



# Linearizability (III)

Can we create a linearizable schedule out of this?




Yes!

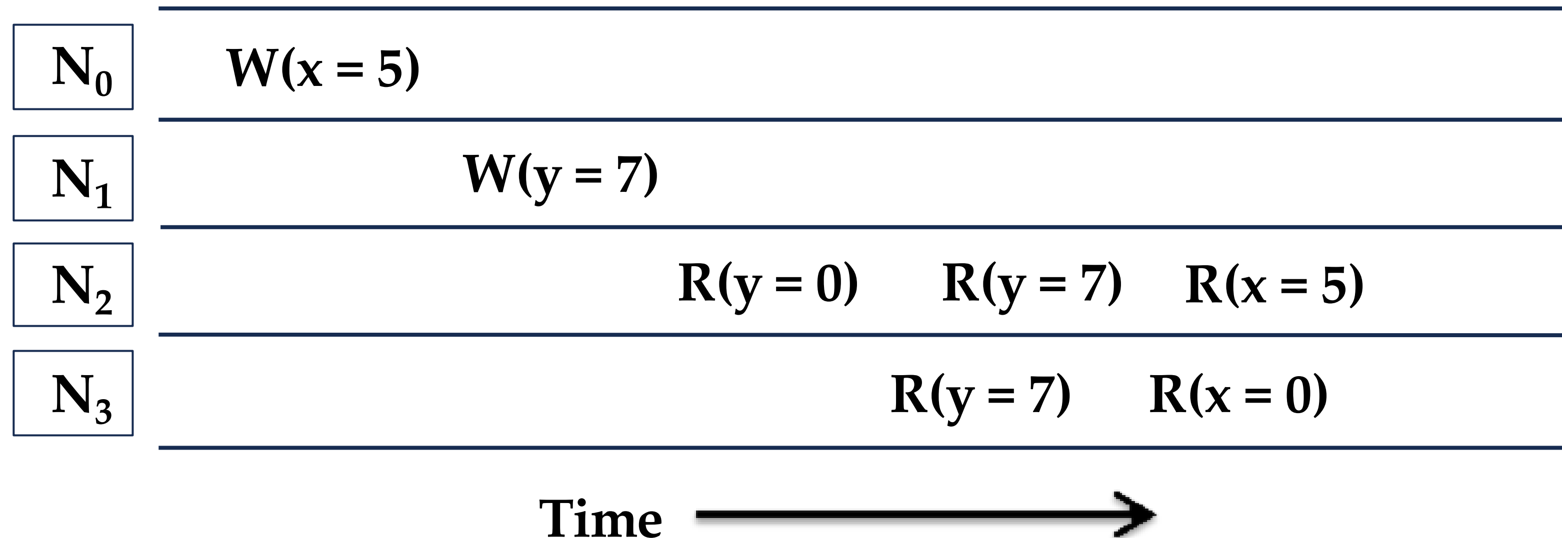
# Causal Consistency

# Causal Consistency

- Follows Lamport clocks Happens Before Relationship.
- Causal consistency only cares about causal relationship between dependent operations.
- No restrictions on concurrent operations.
- Only ensure that if a process/node observes a causally dependent operation, then it also observes the operation that caused that causally dependent operation.

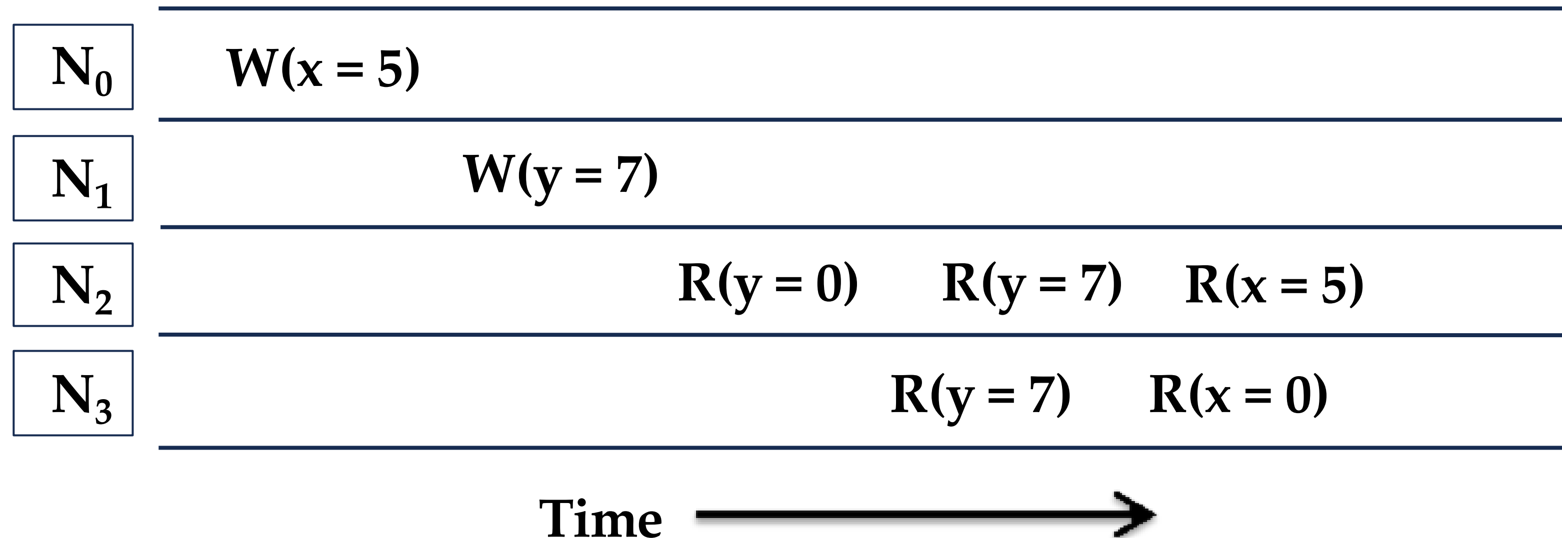
# Causal Consistency (I)

Can we create a causally consistent schedule out of this?



# Causal Consistency (I)

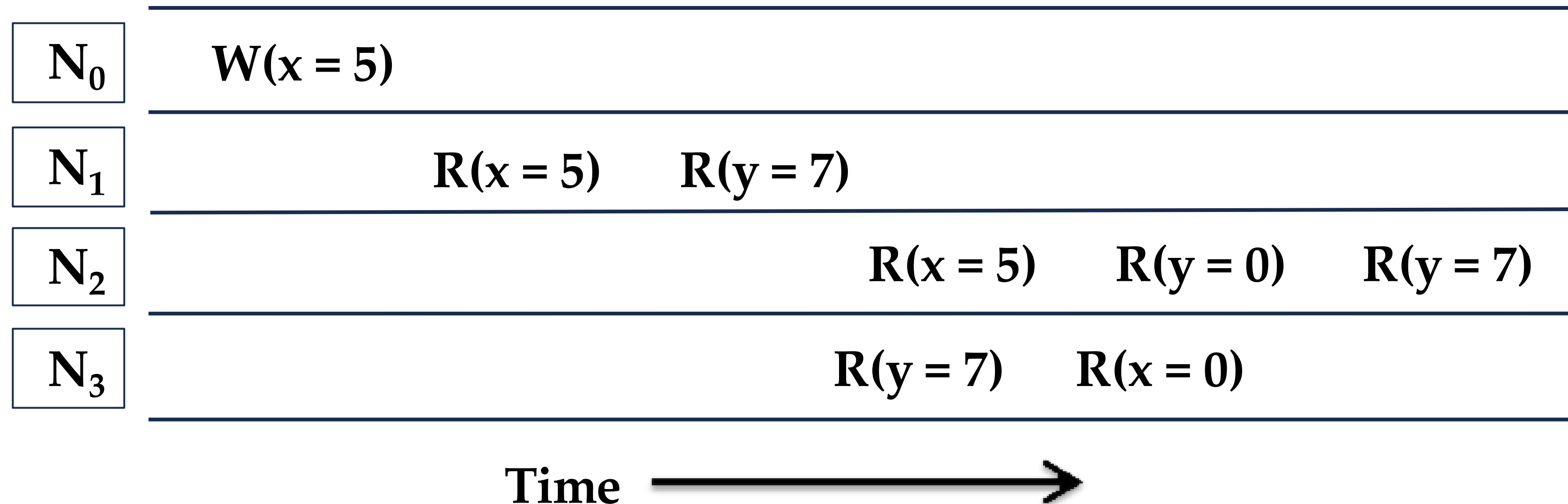
Can we create a causally consistent schedule out of this?




**Yes!**

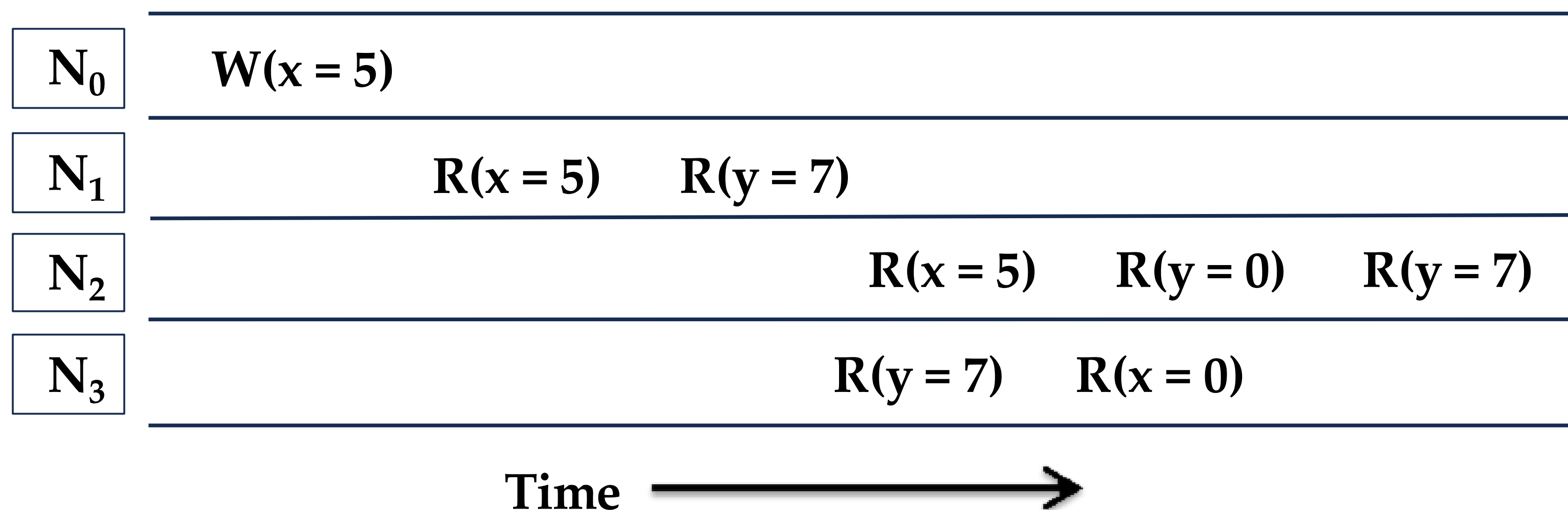
# Causal Consistency (II)

Can we create a causally consistent schedule out of this?



# Causal Consistency (II)

Can we create a causally consistent schedule out of this?




**No**, because $N_3$ observes y=7 but does not observe x=5, which caused y=7.

# Eventual Consistency

# Eventual Consistency

- Traditionally, the weakest form of consistency.

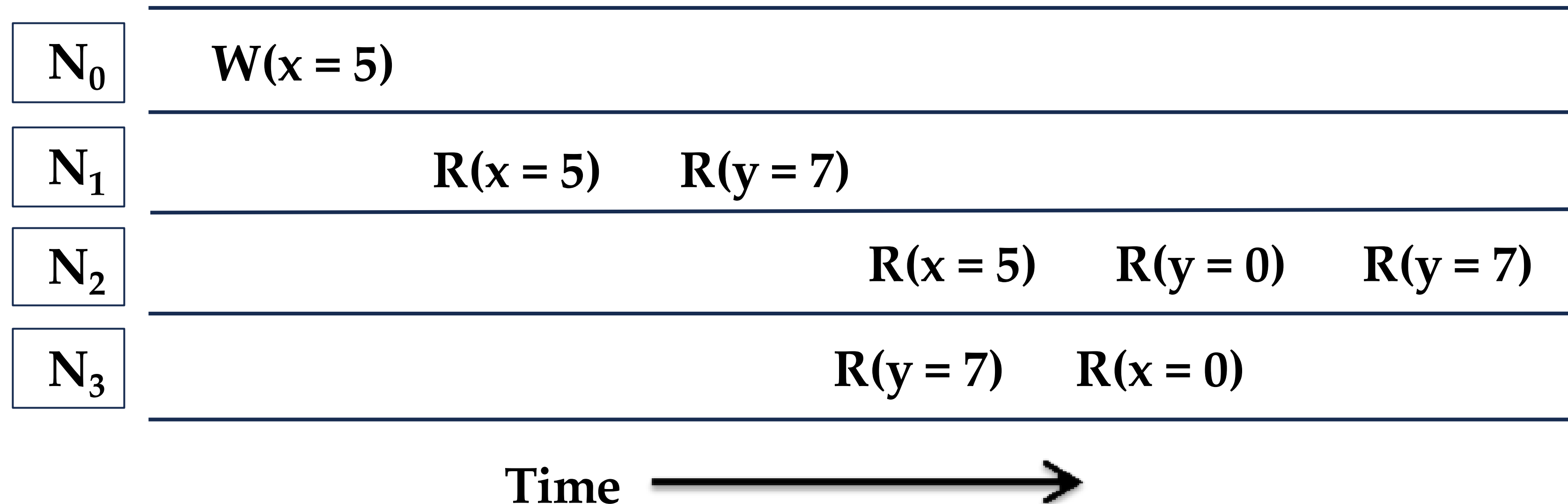
- The only guarantee it gives is that if there are no writes for a ***long period*** of time, then every node will agree on the value of last write.

- Essentially, eventual agreement.

- The value of ***long period of time*** is system dependent.

# Eventual Consistency

This schedule will be eventually consistent if $N_3$ observes x=5 after some time has passed.



# Transaction Isolation Levels

# Conflicting Transactions

- Interleaving concurrent transactions can lead to the following three anomalies:
  - Read-Write Conflicts (**R-W**)
  - Write-Read Conflicts (**W-R**)
  - Write-Write Conflicts (**W-W**)

# Queries Isolation

| Time | T1: | T2: |
|---|---|---|
| → | **Begin** | **Begin** |
| | **select count(*) as** cnt<br>**from** cs_cmployees<br>**where** age > 30 | |
| | | **insert into** cs_employees<br>**values** (anakin, 70, 500) |
| | **select count(*) as** cnt<br>**from** cs_cmployees<br>**where** age > 30 | **Commit** |
| ↓ | **Commit** | |

```
create table cs_employees
    (       name    varchar(20),
            age     int,
            salary  int
    );
```

**Is there any problem with this schedule?**

# Queries Isolation

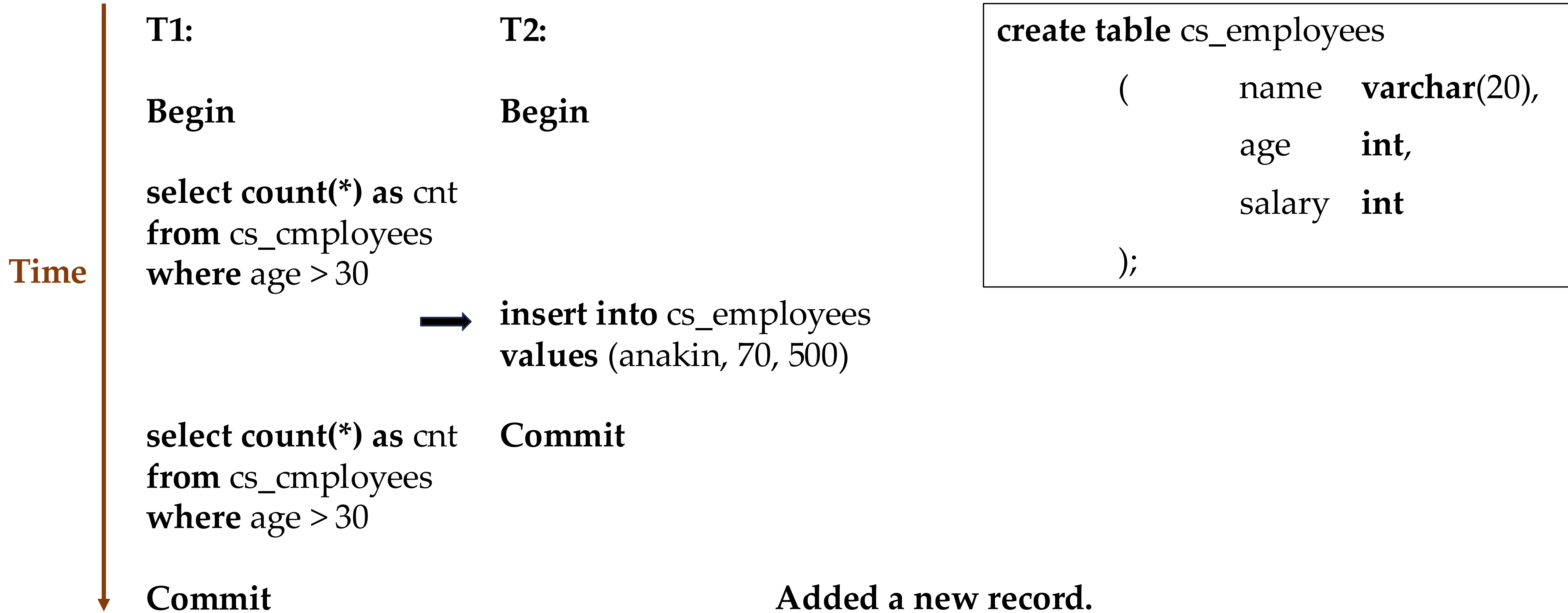
Time

| T1: | T2: |
|---|---|
| **Begin** | **Begin** |
| **select count(*) as** cnt **from** cs_cmployees **where** age > 30 | |
| | **insert into** cs_employees **values** (anakin, 70, 500) |
| **select count(*) as** cnt **from** cs_cmployees **where** age > 30 | **Commit** |
| **Commit** | |

**create table** cs_employees
( name **varchar**(20),
age **int**,
salary **int**
);

**Say, output = 10**

# Queries Isolation



# Queries Isolation

Time

**T1:**

**Begin**

**select count(*) as** cnt
**from** cs_cmployees
**where** age > 30

**T2:**

**Begin**

**insert into** cs_employees
**values** (anakin, 70, 500)

**Commit**

T1:

**select count(*) as** cnt
**from** cs_cmployees
**where** age > 30

**Commit**

```
create table cs_employees
    (   name    varchar(20),
        age     int,
        salary  int
    );
```

**Now, output = 11**

# Queries Isolation

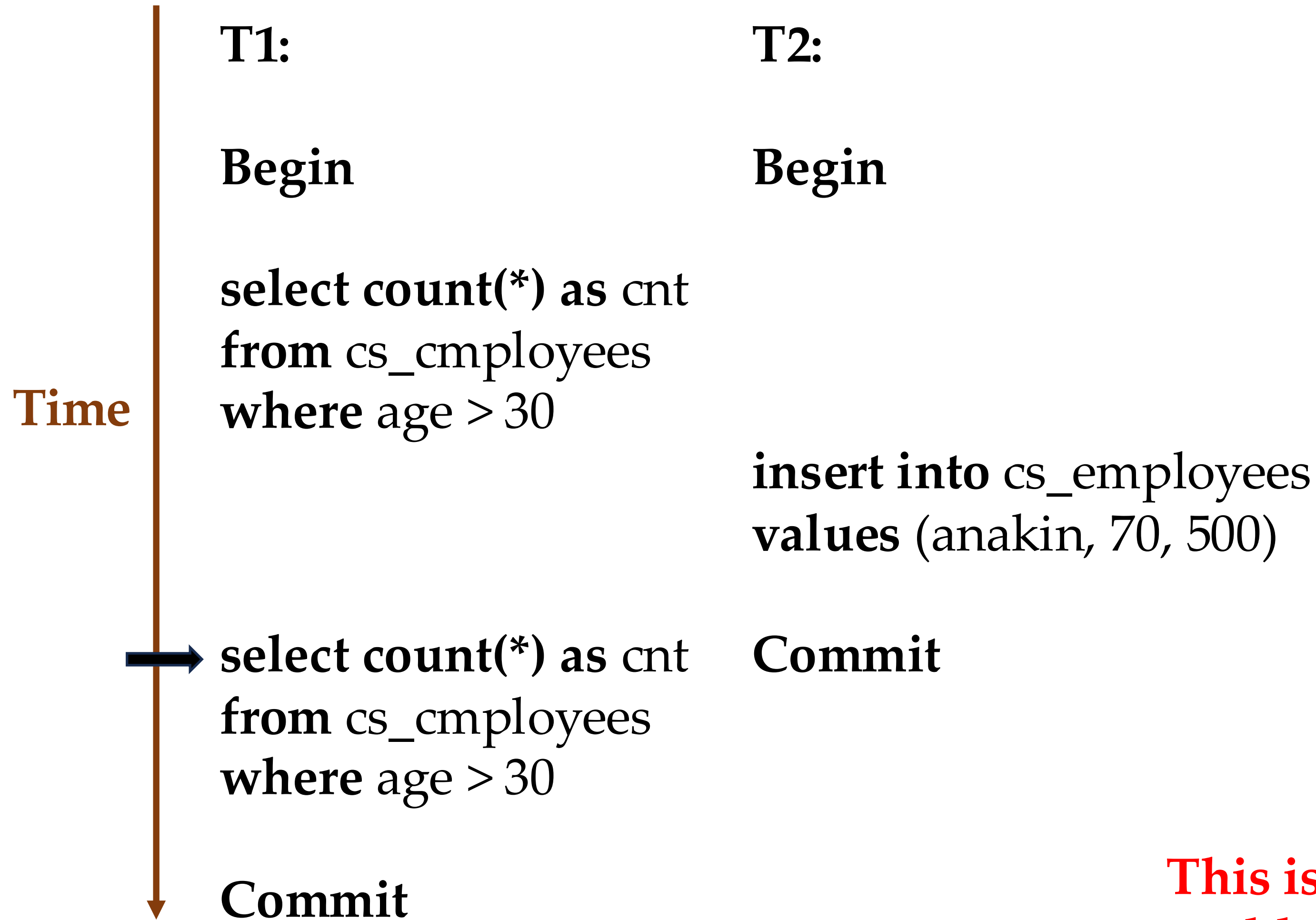
Time

| T1: | T2: |
| --- | --- |
| **Begin** | **Begin** |
| **select count(*) as** cnt<br>**from** cs_cmployees<br>**where** age > 30 | |
| | **insert into** cs_employees<br>**values** (anakin, 70, 500) |
| **select count(*) as** cnt<br>**from** cs_cmployees<br>**where** age > 30 | **Commit** |
| **Commit** | |

**create table** cs_employees
( name **varchar**(20),
age **int**,
salary **int**
);

**The output of the two queries changed!**

# Phantom Problem




```
create table cs_employees
        (       name    varchar(20),
                age     int,
                salary  int
        );
```

**This is also termed as a phantom problem.**

# Phantom Problem

Time

| T1: | T2: |
| --- | --- |
| **Begin** | **Begin** |
| **select count(*) as** cnt<br>**from** cs_cmployees<br>**where** age > 30 | |
| | **insert into** cs_employees<br>**values** (anakin, 70, 500) |
| **select count(*) as** cnt<br>**from** cs_cmployees<br>**where** age > 30 | **Commit** |
| **Commit** | |

```
create table cs_employees
    (   name    varchar(20),
        age     int,
        salary  int
    );
```

**Violates our traditional definition of 2PL?**
**T1 cannot take a lock on something that does not exist!**

# Why Phantom Problem?

# Why Phantom Problem?

- We took read/write locks on existing records and our locking scheme assumed a static system.

- But, real-world databases are dynamic.

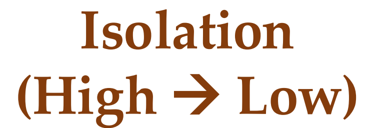
- Concurrent transactions can add new records and our locking scheme does not consider insertions, deletions, and updates.

# Weaker Levels of Isolation

- Isolation Levels control the extent to which a transaction is exposed to the actions of other concurrent transactions.

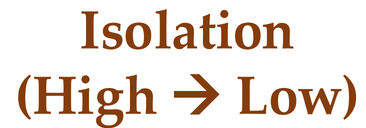
- Providing greater concurrency leads to several challenges:
  - Dirty Reads (W-R)
  - Unrepeatable Reads (R-W)
  - Lost Updates (W-W)
  - Phantom Reads

# Weaker Levels of Isolation



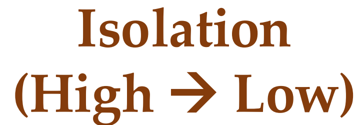

- **Serializable:** no phantoms, all reads repeatable, no dirty reads.

# Weaker Levels of Isolation



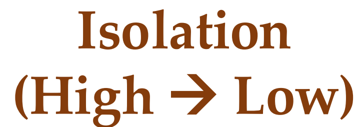

- **Serializable:** no phantoms, all reads repeatable, no dirty reads.
- **Repeatable Reads:** phantoms may happen.

# Weaker Levels of Isolation



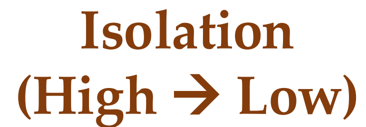

- **Serializable:** no phantoms, all reads repeatable, no dirty reads.
- **Repeatable Reads:** phantoms may happen.
- **Read Committed:** phantoms, unrepeatable reads, and lost updates may happen.

# Weaker Levels of Isolation




- **Serializable:** no phantoms, all reads repeatable, no dirty reads.
- **Repeatable Reads:** phantoms may happen.
- **Read Committed:** phantoms, unrepeatable reads, and lost updates may happen.
- **Read Uncommitted:** all anomalies may happen.

# Weaker Levels of Isolation




- **Serializable:** Strong Strict 2PL with phantom protection (example index locks
- **Repeatable Reads:** Same as above, but without phantom protection.
- **Read Committed:** Same as above, but S-Locks are released immediately.
- **Read Uncommitted:** Same as above but allows dirty reads (no S-Locks).

# Isolation Levels vs. Consistency Levels

| Isolation Levels | Consistency Levels |
| --- | --- |
| • **Correspond** to the **I in ACID**.<br>• **Database isolation** is the ability of a database to allow a transaction to execute as if there are no other concurrently running transactions.<br>• Greater the guaranteed isolation among the transactions, lesser the system performance.<br>• **Isolation levels** trade off isolation guarantees for improved performance. | • **Do not correspond to C in ACID**.<br>• Unlike the C in ACID, the **database consistency** refers to the rules that make a **concurrent**, **distributed system** appear as a **single-threaded, centralized system**.<br>• **Reads** at a particular point in time **must reflect the most recently completed write** (in real-time) of that data item, no matter which server processed that write.<br>• **Consistency levels** trade off read results for improved performance. |