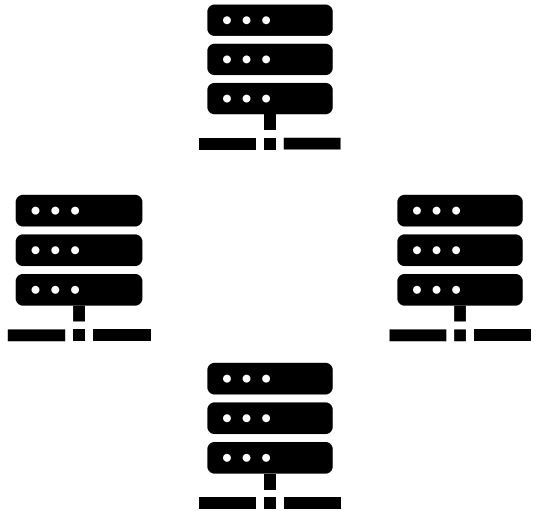
Stronger Consistency Models

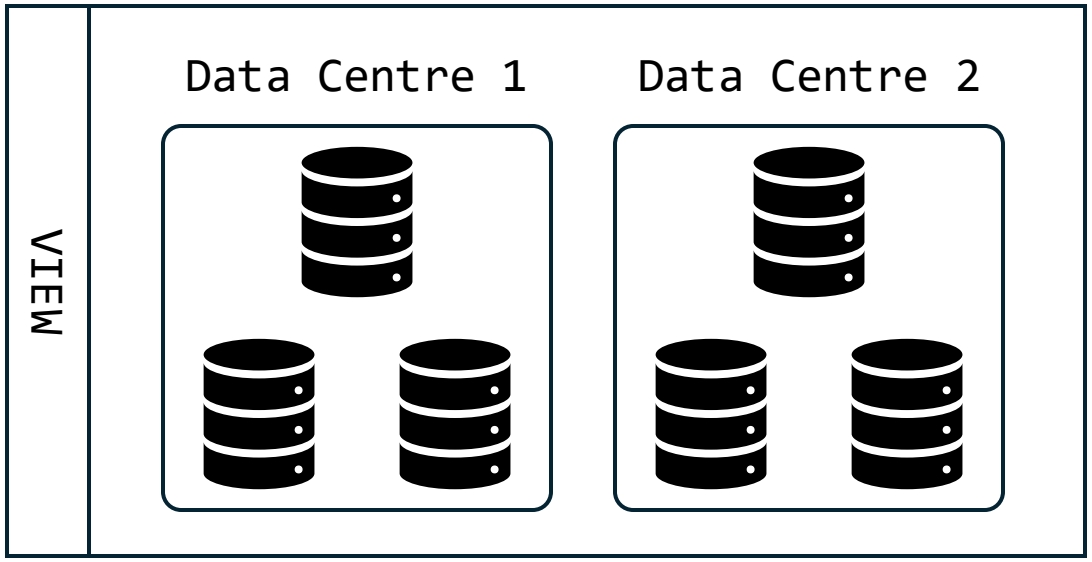
*stronger than eventual, weaker than serializable

Consistency in Distributed Systems

Every replica / node in the distributed system should have the same view of the data at a given point of time.



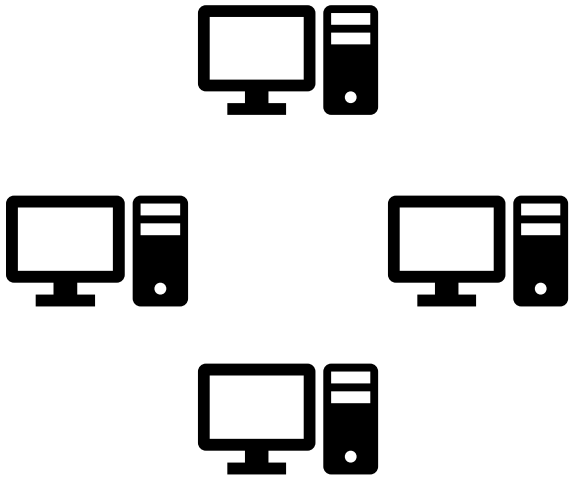
Application Replicas



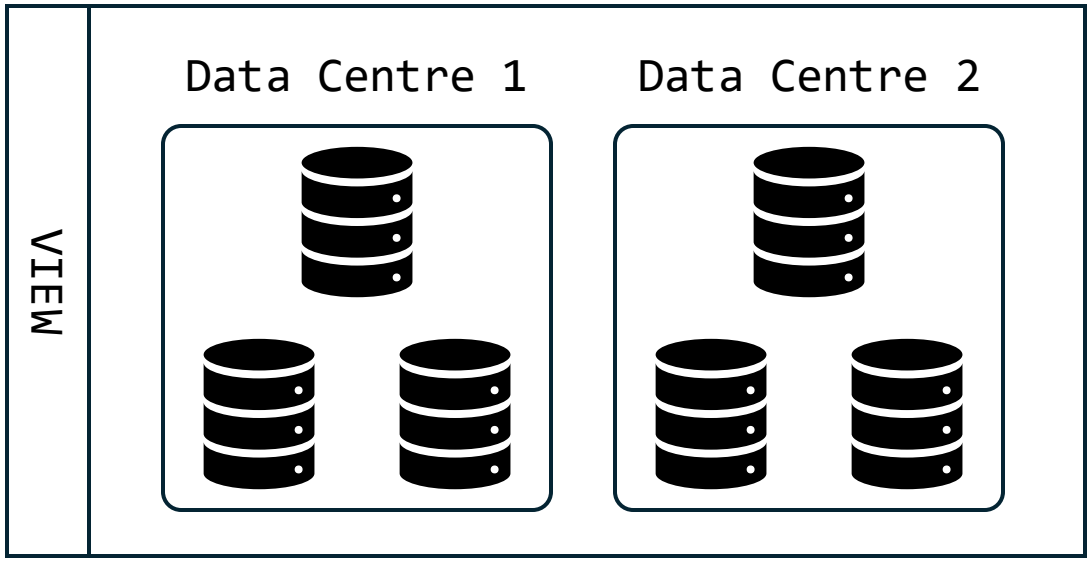
Data Storage

Consistency in Distributed Systems

Every client should have the same view of the data at a given point of time.



Clients



Data Storage

Need for Consistency Guarantees

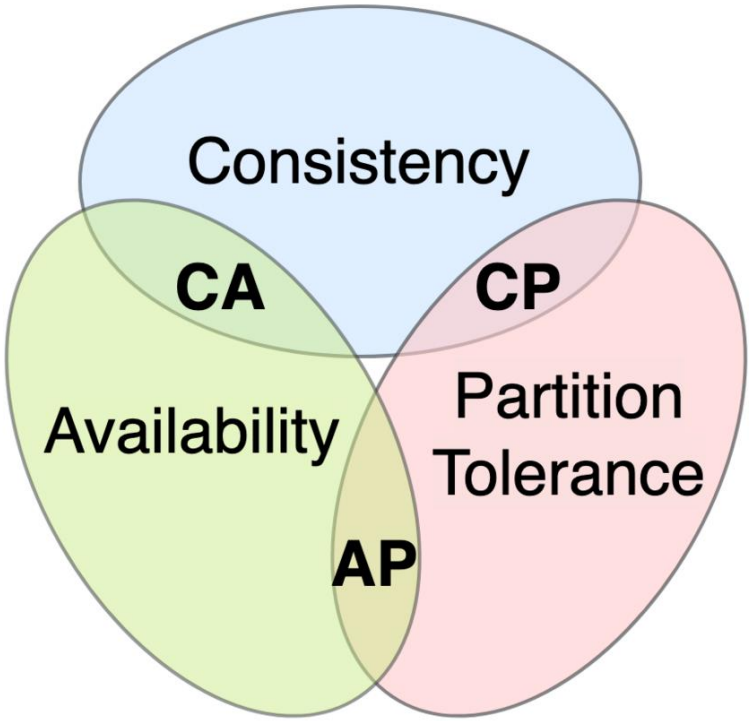
- In a distributed system, consistency conflicts arise due to **concurrent read / writes** to **different replicas**.
- To ensure consistency, system must **synchronize data** across the replicas.
- This can have a **significant overhead**, impacting the performance of a system.

Recap: CAP Theorem

- In distributed systems, **Partition Tolerance** is a must.

- We are left with two choices:

CP or **AP**



What is a Consistency Model?

- A consistency model defines the **rules and guarantees** about how data is seen and updated across multiple nodes in a distributed system.

- They are tradeoffs between:

Availability vs **Consistency**

Weak Consistency Models

- Sacrifice **consistency**
- Prioritize **availability**

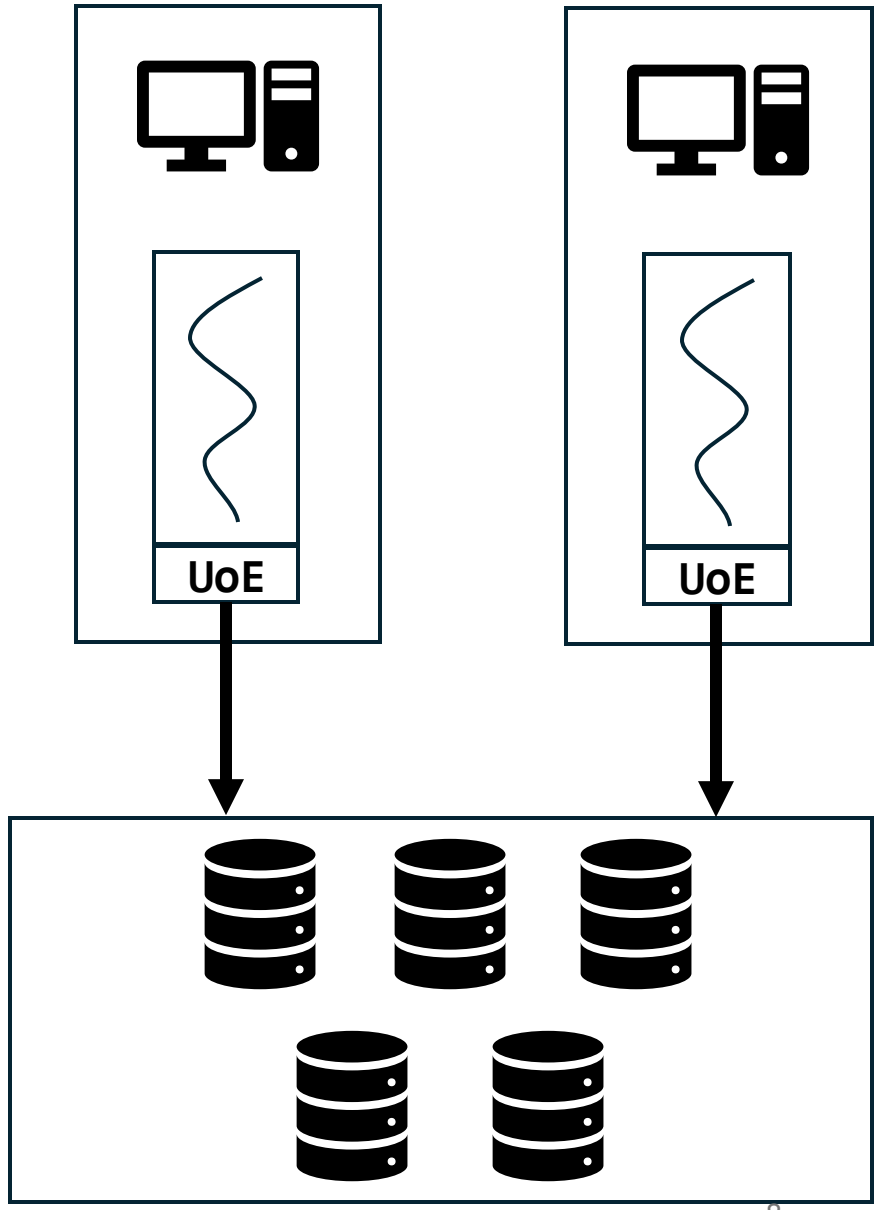
- Require analysis of the data model to determine the required consistency guarantees
- Careful analysis can allow us to achieve the optimal balance between consistency and availability

We will discuss consistency models in the light of their consistency guarantees w.r.t. applications

Some Terminology

- Unit of Execution

An independent thread (client)
making sequential requests to
an external distributed / non-distributed system



Some Terminology

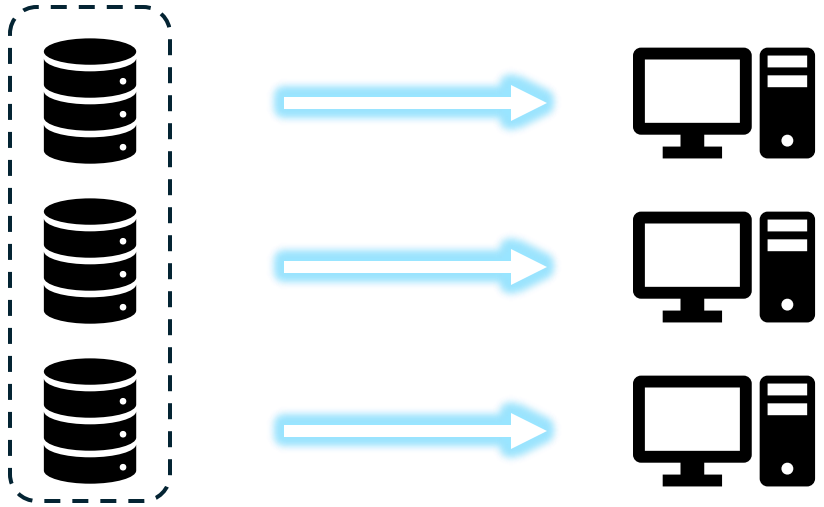
- Data-Centric Perspective
  - System is aware that multiple execution units are running
  - It synchronizes data access from all of them to guarantee consistent results

- Applications in Banking, Trading, Authentication and Authorization, where data consistency is critical.

Some Terminology

- Client-Centric Perspective
  - System only cares about concerned process operating on the given data
  - It assumes other execution units don’t exist or don’t have significant impact

- Applications in social media like YouTube where the priority is user experience.



Performance Metrics

Consistency

Availability

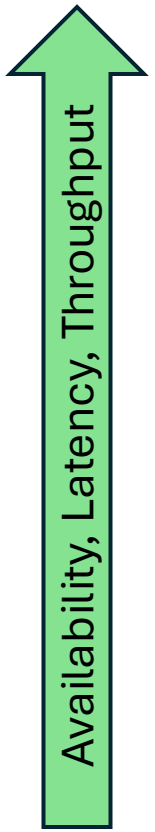
Latency

Throughput

Perspective

Consistency Guarantees Scale

1. Eventual
2. Consistent Prefix Read
3. Session
4. Causal
5. Bounded Staleness
6. Strong



Availability
Latency
Throughput

Agenda

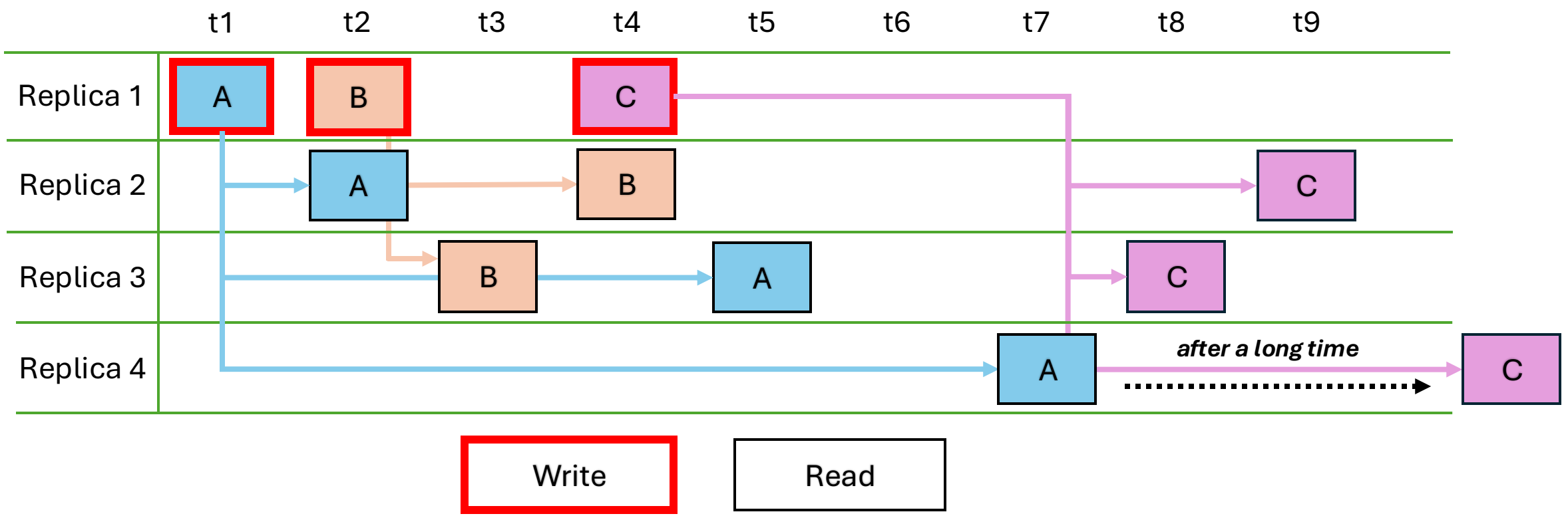
1. Eventual
2. Consistent Prefix Read
3. Session
4. Causal
5. Bounded Staleness
6. Strong

The weakest notion of consistency among all four models discussed

Eventual Consistency

- It guarantees that if no new writes are made to a data item, all replicas will converge to the same value *eventually*.

Eventual Consistency



The timeline shows writes and reads to a single data item

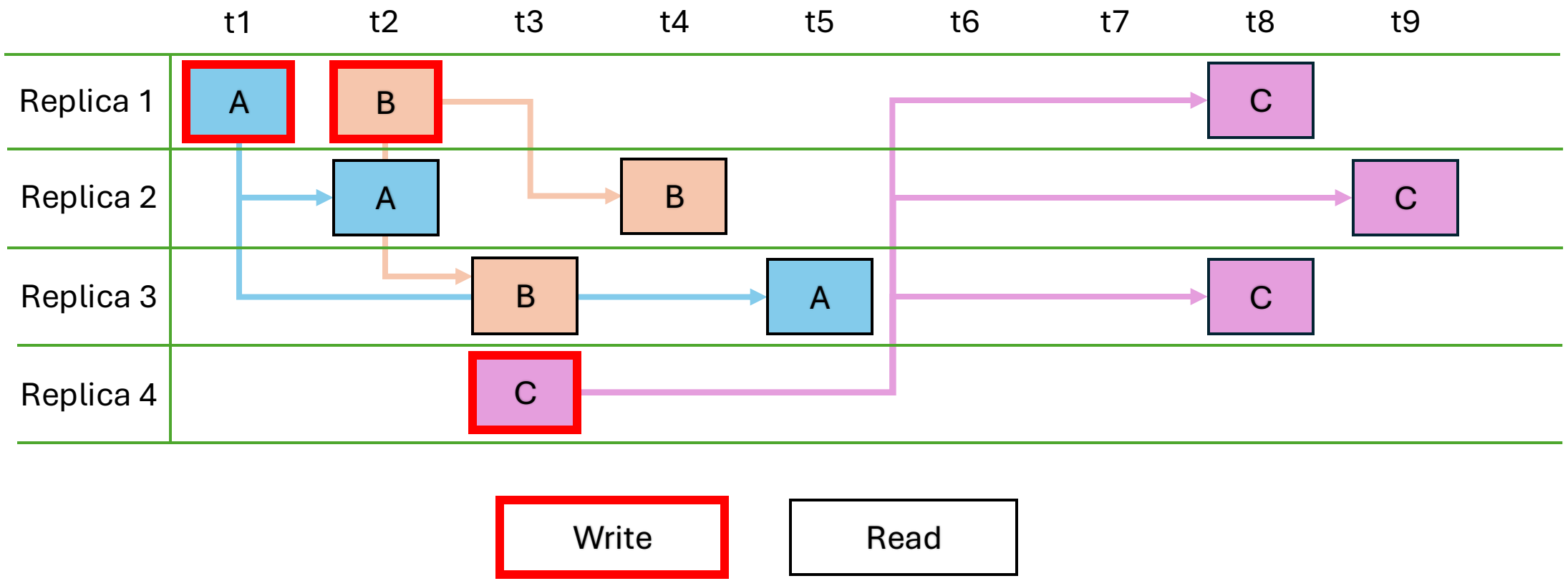
Eventual Consistency

- **No ordering guarantee** on the reads and writes
- Even the unit of execution which made the write may not observe it in the following writes (**Invisibility**)
- **No time bound** on convergence

- Only guarantee is that:
  - Data will converge to the same value **eventually**

- The value to which it converges depends on the **conflict resolution policy**
  - For example: Last Write Wins, Merging as in CRDT, etc.

Conflict Resolution - Last Write Wins



The timeline shows writes and reads to a single data item

Eventual Consistency

Suitable for applications in which immediate consistency and order of updates is not of concern but availability is essential.

- **DNS Servers**
  - Propagating DNS record updates across global servers
  - Changes to single record are infrequent

- **Views, likes, comments aggregation** in social media platforms

Eventual Consistency

Consistency	**Least**
Availability	**Highest**
Latency	**Low**
Throughput	**Highest**
Perspective	**Client-Centric**

Eventual | **CPR** | Session | Causal

Consistent Prefix Read

Provides record-level guarantee on read order

Eventual | CPR | **Session** | Causal

Provides four consistency guarantees to individual **units of execution**

Read Your Own Write (RYOW)

Monotonic Read (MR)

Write Follow Read (WFR)

Monotonic Write (MW)

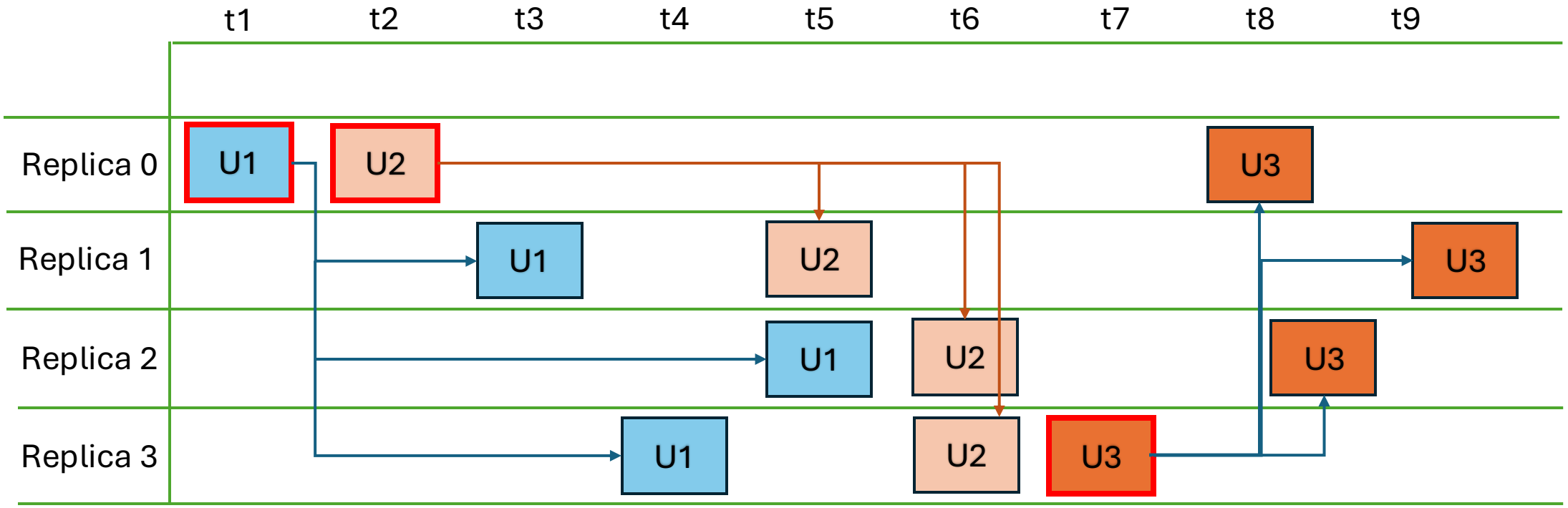
Consistency based on **causal dependencies** between reads and write

Consistent Prefix Read

Consistent Prefix Read guarantees that all replicas receive updates to a given data item in the same order.

That is, for a data item, the database system as a whole has received updates $\mathbf{u_1, u_2, \ldots, u_n}$ in that order, any replica will have received updates $\mathbf{u_1, u_2, \ldots, u_k}$ in that order for some $k \leq n$.

Consistent Prefix Read



These are all updates to a single data item.

**In pure CPR implementations, it is assumed that there is a reliable mechanism (such as a central sequencer or consensus algorithm) that imposes a global order on updates. In scenarios where writes can be directed to any replica, these mechanisms ensure that all replicas eventually agree on the order.*

For the same example, these are the updates received by replicas (in order) at time stamps:

Replica	t1	t2	t3	t4	t5	t6	t7	t8	t9
R0	**U1**	U1, **U2**	U1, U2	U1, U2	U1, U2	U1, U2	U1, U2	U1, U2, U3	U1, U2, U3
R1	—	—	U1	U1	U1	U1, U2	U1, U2	U1, U2, U3	U1, U2, U3
R2	—	—	—	—	U1	U1, U2	U1, U2	U1, U2, U3	U1, U2, U3
R3	—	—	—	—	U1	U1, U2	U1, U2, **U3**	U1, U2, U3	U1, U2, U3

Ux indicates the first time any of the replicas receive a write (from client)

Eventual Consistency

- **No ordering guarantee** on the reads and writes
- Even the unit of execution which made the write may not observe it in the following writes (**Invisibility**)
- **No time bound** on convergence

Consistent Prefix Read

What we can’t do using **Eventual consistency**?

Eventual Consistency only guarantees that if no new updates are made to a data item, all replicas will converge to the same state over time. There is no guarantee on the order in which updates are seen in the meantime.

Consistent Prefix Read

- What we can't do using **Eventual consistency**?
- Consider an image sharing application where users can upload images and only their friends can view those images. Assume that this application uses replication for availability.

Consistent Prefix Read

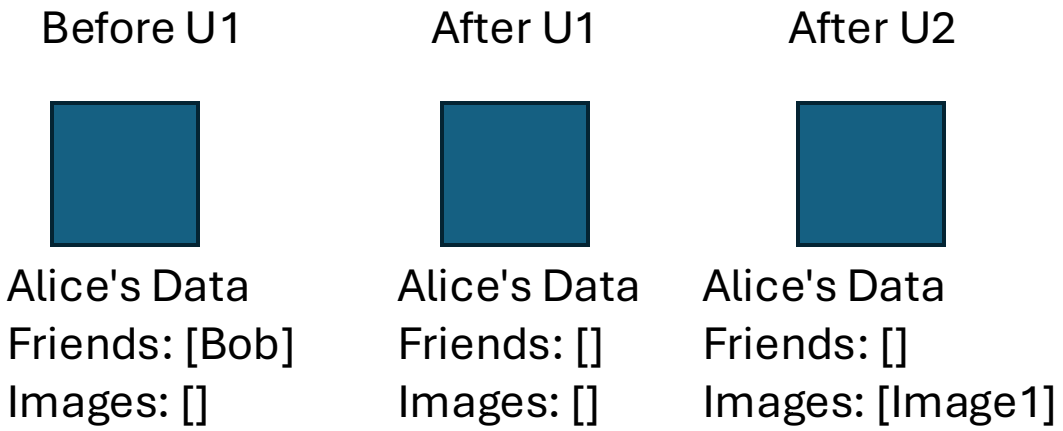
- What we can't do using **Eventual consistency**?
- Consider an image sharing application where users can upload images and only their friends can view those images. Assume that this application uses replication for availability.
- Suppose that Alice and Bob are initially friends. Alice wants to post an image but doesn't want Bob to see it. So, Alice will unfriend Bob first. After getting confirmation, will post the image.
- What can go wrong here, if we use a database that only guarantees Eventual consistency?

Consistent Prefix Read

What's a "read" in our context?

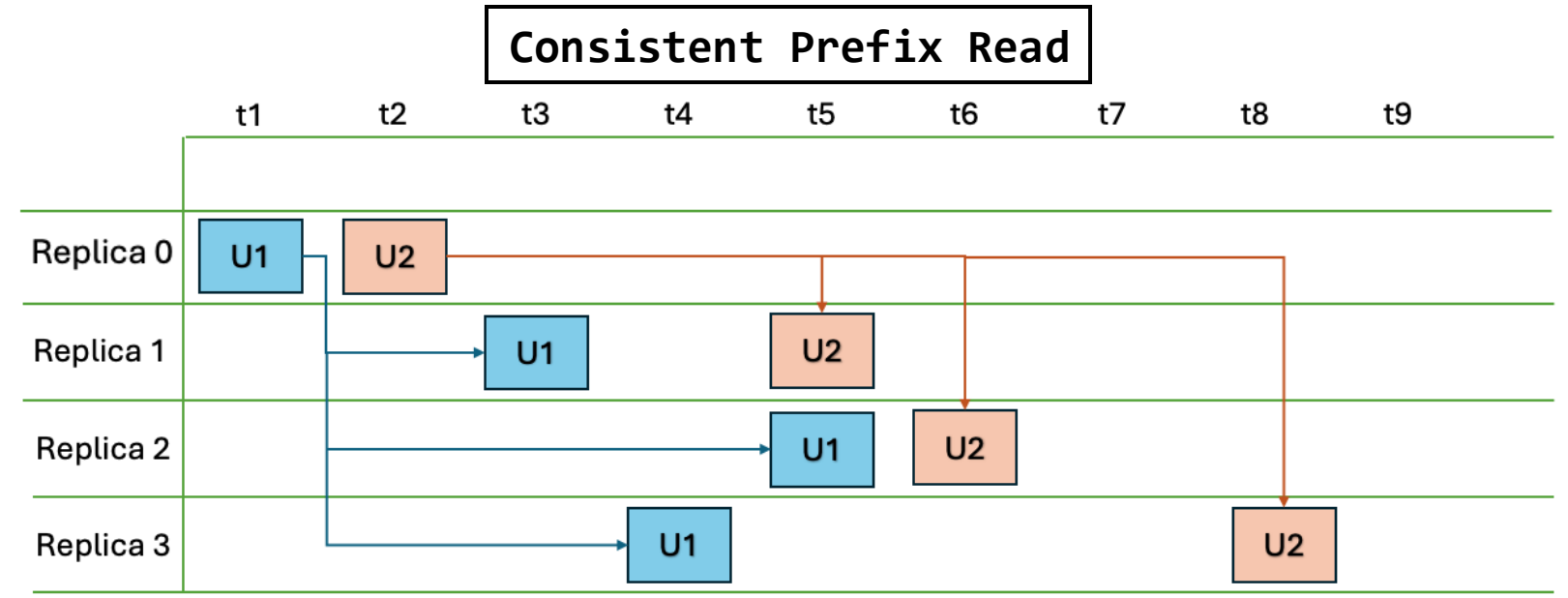
When a read request to comes to a replica for Alice's record (data item) from Bob, it will check if Bob is in the friend's list, if yes, he can view the images, otherwise, no.

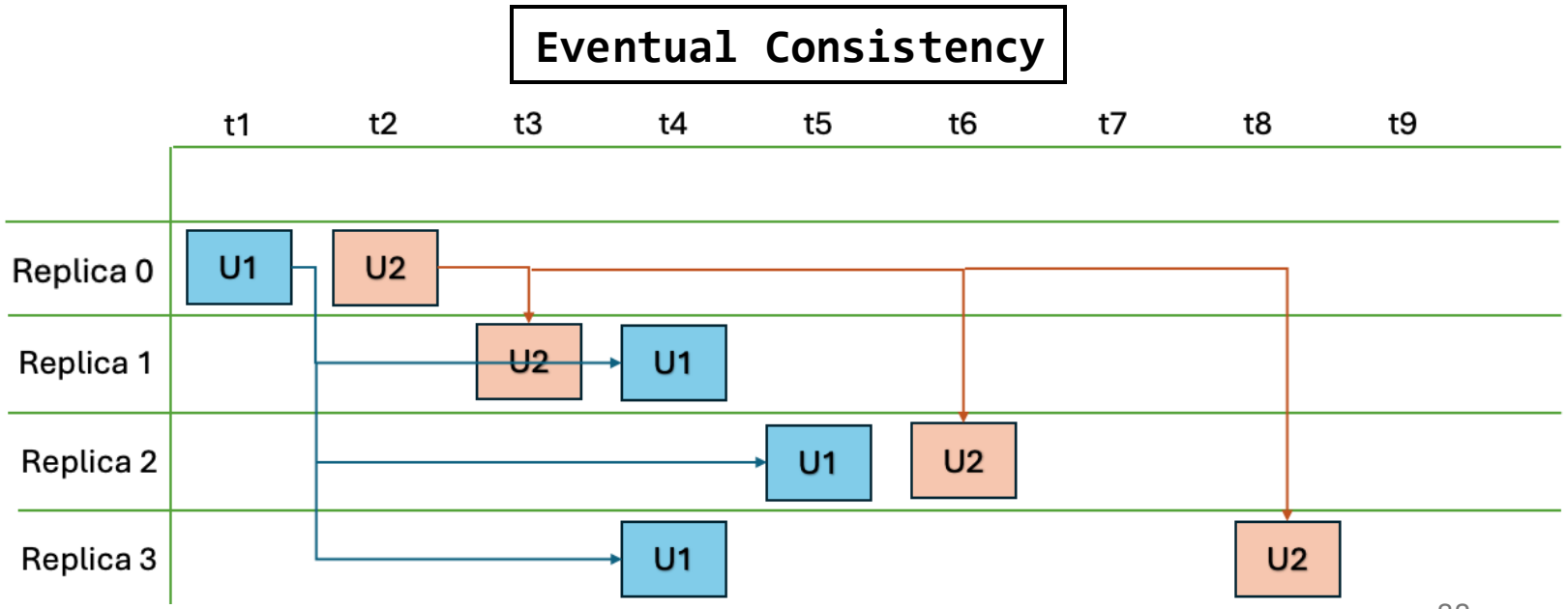
Types of Writes:
U1: Remove from friend list
U2: Post a picture



Types of Writes:
U1: Remove from friend list
U2: Post a picture

Read by Bob at t_i time





Consistent Prefix Read

Problem with **Eventual consistency**?

Due to the lack of ordering guarantees in eventual consistency, a replica that is slow to apply updates may process U2 (post image) before U1 (remove Bob), causing Bob's read of Alice's data to incorrectly include the newly posted image—even though he should no longer be a friend.

Consistent Prefix Read

- For this use-case, Eventual consistency isn't enough. We need a stronger notion of consistency.
- It guarantees the following
  - If the system **receives** the updates U1, U2 in that order for a data item. Then, all replicas will apply those updates to that data item in the same order.
- What doesn't it guarantee?
  - It doesn't have any bound on how stale the data can be.

PNUTS: Yahoo! Hosted Data Service Platform

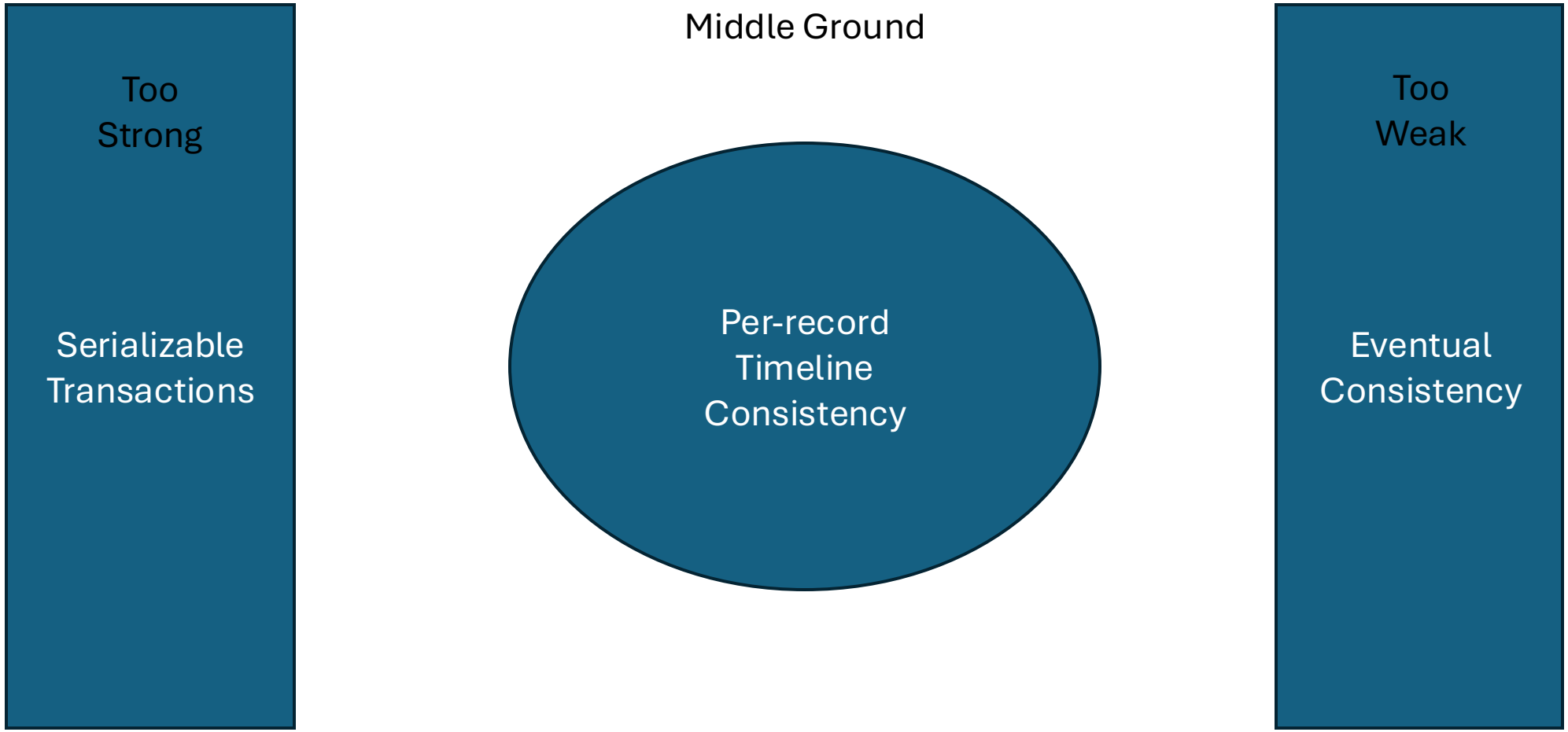
- **Massively Parallel & Distributed**: Supports web applications across multiple geographic regions.
- **Low Latency**: Efficient for large-scale concurrent queries and updates.
- **Automated Management**: Centralized, hosted service with load balancing and failover.

**Platform for Nimble Universal Table Storage*

PNUTS: Yahoo! Hosted Data Service Platform

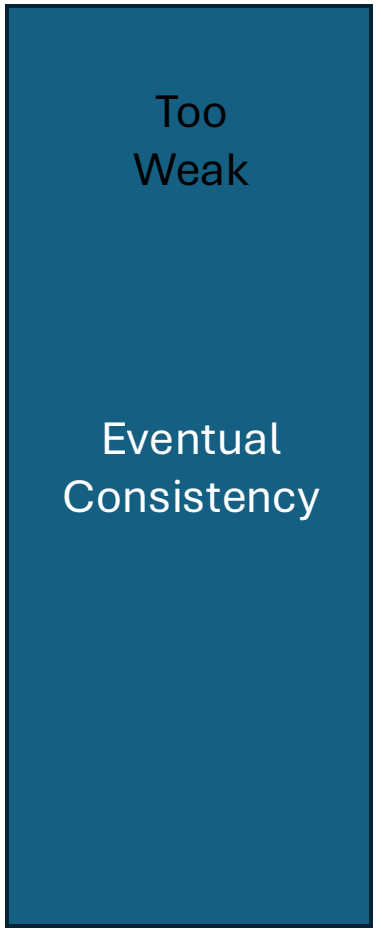
- **Massively Parallel & Distributed**: Supports web applications across multiple geographic regions.
- **Low Latency**: Efficient for large-scale concurrent queries and updates.
- **Automated Management**: Centralized, hosted service with load balancing and failover.
- **Per-Record Timeline Consistency**: Ensures updates are applied in the same order across replicas. Relaxed Consistency guarantees.

PNUTS: Yahoo! Hosted Data Service Platform

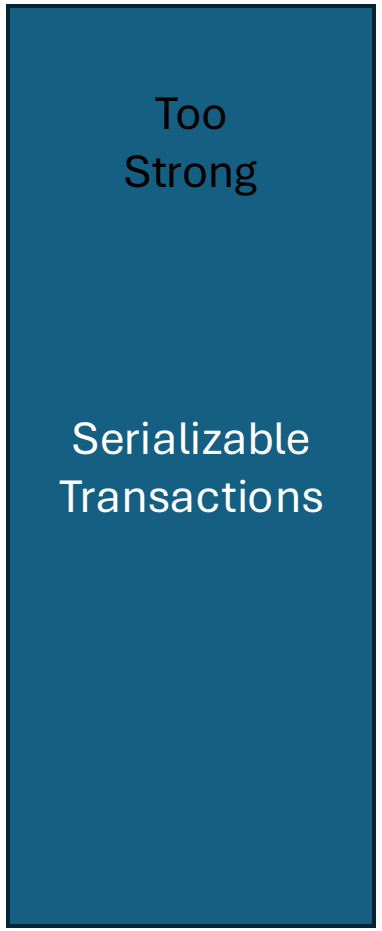


PNUTS: Yahoo! Hosted Data Service Platform

A client can update any replica of an object and all updates to an object will eventually be applied, but potentially in different orders at different replicas.



PNUTS: Yahoo! Hosted Data Service Platform

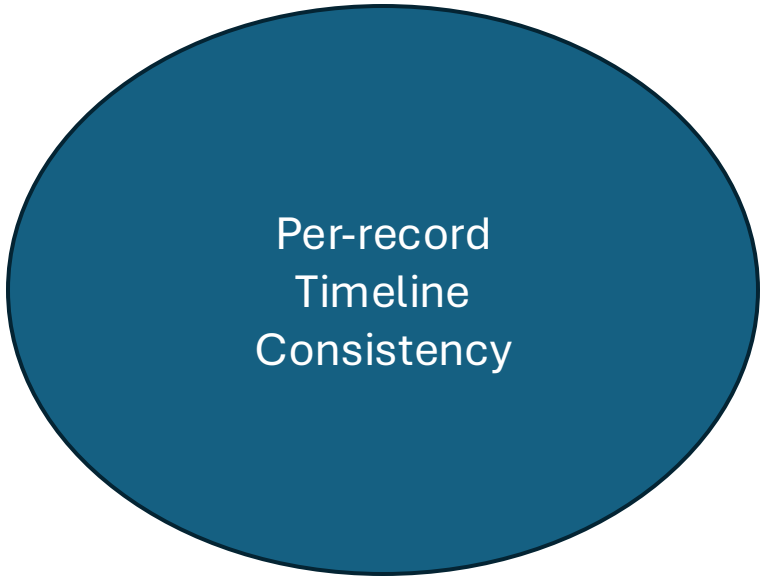


Serializable Transactions are the highest level of isolation in database systems. They ensure that transactions execute **as if they were run sequentially, one after the other**, even if they are executed concurrently.

PNUTS: Yahoo! Hosted Data Service Platform

Middle Ground

Ensures updates to each record are applied in the same order across replicas. This offers stronger guarantees than eventual consistency while avoiding the complexity of serializability.



Consistency per record instead of enforcing global order. Reduces delays and handles more data at once. This allows **concurrent reads and writes on different records**, unlike serializable systems that would process them one by one.

**PNUTS make no guarantees as to consistency for multi-record transactions.*

PNUTS: Per-record Timeline Consistency

- Different records may have activity with different geographic locality
- **All replicas of a given record apply all updates to the record in the same order**

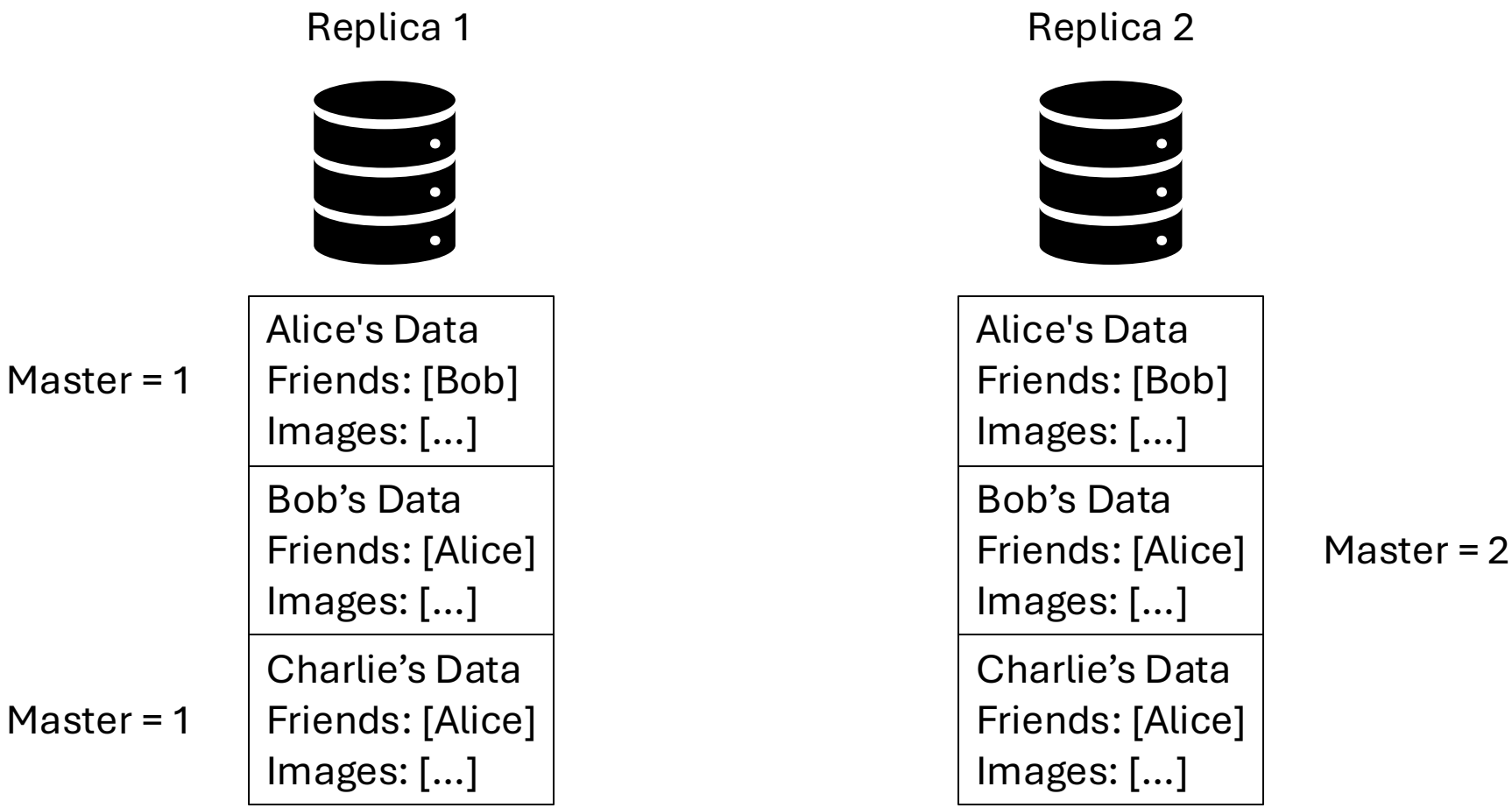
PNUTS: Per-record Timeline Consistency

- A read of any replica will return a consistent version from this timeline
- Replicas always move forward in the timeline.

PNUTS: Master based Implementation

- Each record is assigned a master replica.
- **All updates** to a record are forwarded to its master.

PNUTS: Records across Replicas

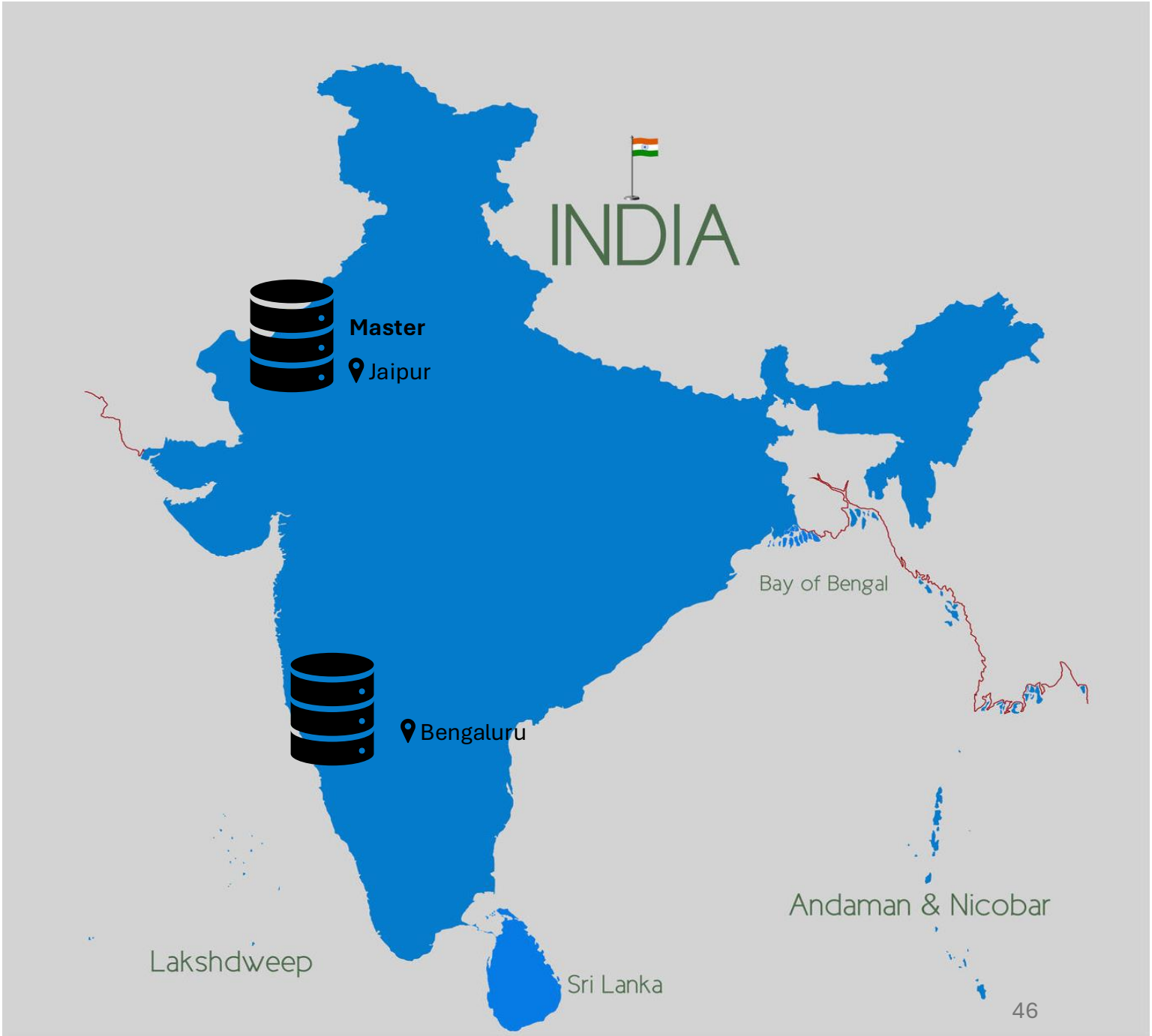


PNUTS: Master Selection

- The master replica is **adaptively changed** based on the workload.
- The replica receiving the **majority of write requests** for a specific record becomes the master.

Replica Location MATTERS!

For people in Jaipur, read and write requests go to Jaipur replica if it's not down and vice versa.



PNUTS: Role of Master Replica

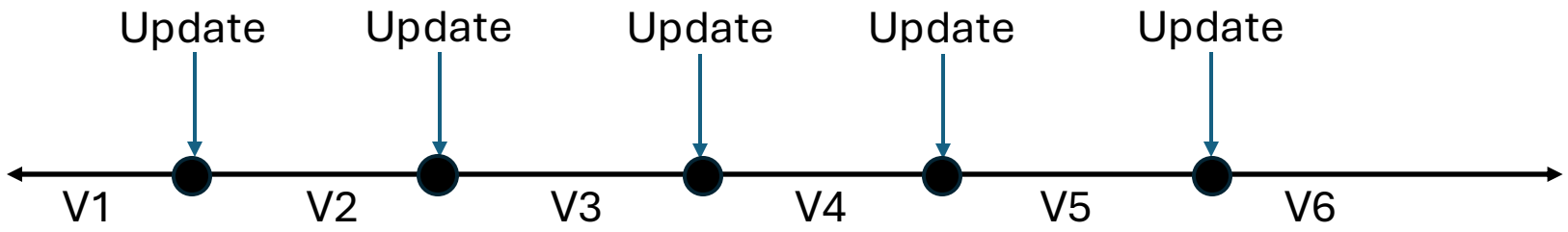
- Master ensures a **single, global write order**.
- **Local master = faster writes** for nearby users.
- Avoids **latency from cross-city/cross-country communication**.
- Improves **read freshness** and **user experience**.
- Jaipur users benefit if **Jaipur is the master**.

* *A **fresh read** reflects the **latest committed updates**.*

PNUTS: Ordering

- Each record has a **sequence number** that increments with every write maintained by master node.
- The sequence number consists of **Version**: Represents an update (each update creates a new version).

Per-record Timeline Consistency



PNUTS: Guarantees

- Replicas always move forward in the timeline.
- **No Reordering of Updates:** Replicas will apply updates in the correct order, ensuring consistency in the view of data across different replicas.
- **Monotonic Progress:** Once a replica has applied a particular version, it will not revert to an older version. It only moves to a newer version.
- **Read Consistency:** When a read request is made, it will return a version that is guaranteed to be consistent with the latest applied version in that replica's timeline.

PNUTS API: Read Any

- Read the record from any replica (usually geographically nearest), which might be non-master. We are guaranteed to get a consistent version from the timeline (although it might be stale).
- Even after a successful write, a stale version may still be seen.
- This call offers **lower latency** compared to stricter read guarantees.
- Example: In a **social networking app**, displaying a friend's status doesn’t require the most up-to-date value, making **read-any** suitable.

Consistent Prefix Read: Metrics

Consistency	**Low**
Availability	**Moderate**
Latency	**Low**
Throughput	**High**
Perspective	**Data-Centric**

Agenda

Session Guarantees

- Introduction
- Data storage model and terminology
- Read-Your-Own-Write (RYOW)
- Monotonic Read
- Writes Follow Reads
- Monotonic Write (MW)

Implementation

Strong vs Weak Consistent Systems

Strongly Consistent System

Ensures that replicas in the system have same view of data.

Pros:

- Prevents anomalies like stale reads or lost updates
- Easier to reason about correctness in applications

Cons:

- High latency: Requires coordination across nodes, increasing response times.
- High concurrency leads to frequent locking. Increased chances of deadlocks. So, low availability.

Weakly Consistent System

Allows different replicas to have different states temporarily, but guarantees eventual convergence

Pros:

- Scalability: Easier to distribute across global regions
- Better performance: Lower latency as replicas don’t need immediate synchronization.
- High availability

Cons:

- User might see stale data.
- Some applications can't work as intended without any guarantees on the order of operations.

Session Guarantees

- By incorporating some extra logic on top of a weakly consistent system, we can make the system provide some guarantees with high availability
- These guarantees are only applicable within a session (i.e these are not global guarantees). We will talk about 4 such guarantees
  - Read your own Write
  - Monotonic Read
  - Write Follows Read
  - Monotonic Write

Terminology

- A ***session*** is an abstract concept to correlate multiple Read and Write operations together as a group from the same unit of execution.
- Example: when you login to Amazon for shopping, a session is created internally which keeps track of your activities and browsing history, cart updates for that session.

Terminology

- Sessions can be identified with a unique ID called session ID. The life time of a session could be few seconds to days or more depending on the business use case.
- When a group of Read / Write operations are to be performed, we can bind all these operations together within a session, the client can keep on passing the corresponding session ID with all requests to help correlate them.

Terminology

- Each Write to our distributed database system has a globally unique identifier, called a **"*WID*".**
- **DB(S,t)** to be the ordered sequence of Writes(WID) that have been received by replica **S at or before time t.**
- Let **WriteOrder(W1,W2)** be a boolean predicate indicating whether Write W1 should be ordered before Write W2.

Read Your Own Write

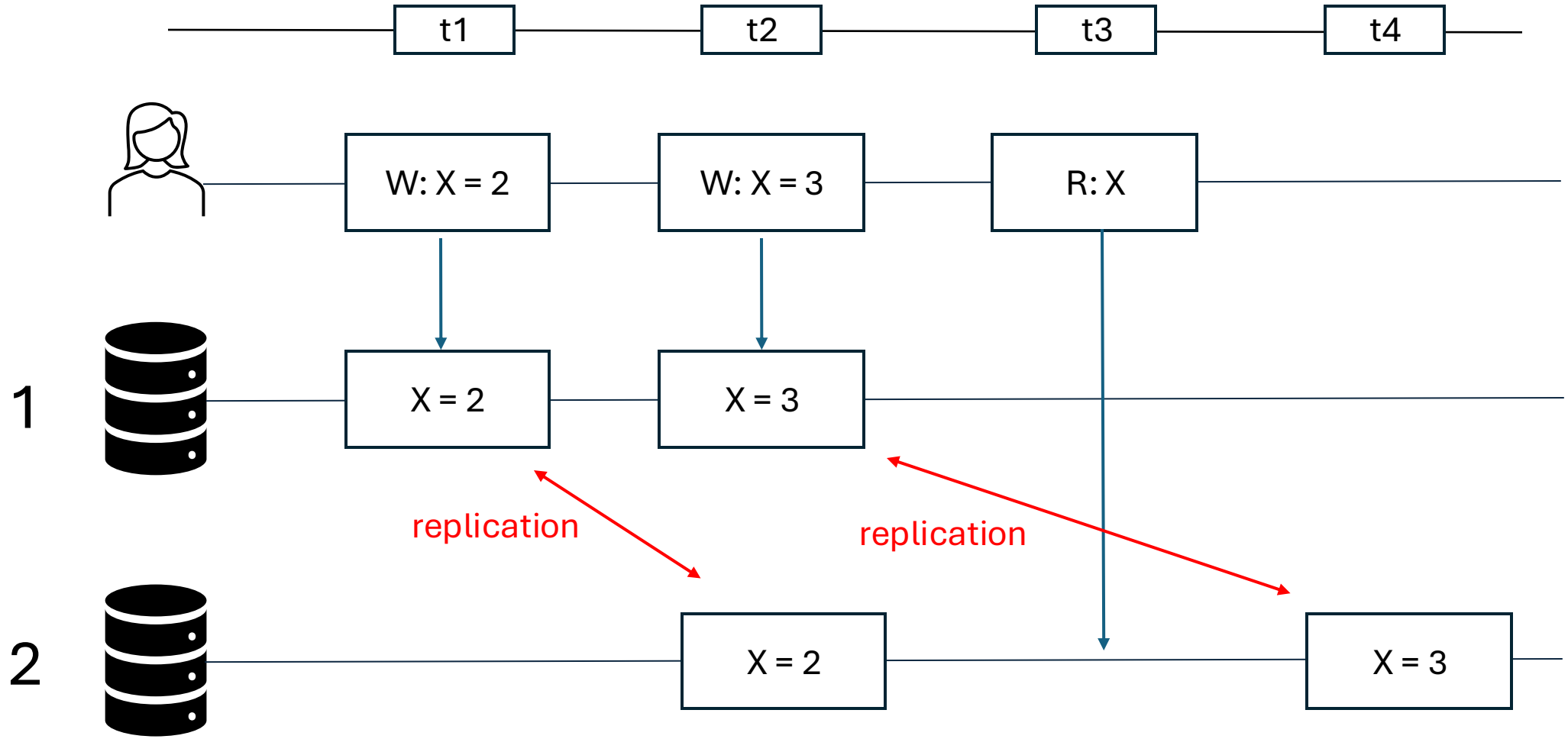
This guarantee ensures that the effects of any Writes made within a session are visible to Reads within that session.

RYOW-guarantee: If Read R follows Write W in a session and R is performed within same session at replica S at time t, then W is included in DB(S,t).

Example1:- If you delete a mail from Gmail and you get the confirmation, upon refreshing the page, you should not see the mail again in RYOW consistency.

Example2:- After updating the account password the user should be able to login with the new password.

Example



How to guarantee RYOW consistency

There could be couple of ways, we will discuss one of them

- Make all requests from same session hit the same replica
- We can choose the replica based on the hash of session ID
- Eventually, all replicas will converge to same state

Monotonic Reads

Informal Definition:-

If a Read R0 observes Write W for a piece of data, further Read R1 by the same unit of execution for the same data in the same session should at least observe W or more recent value.

In case of a distributed system having multiple replicas, the client can reach out to any replica as long as they have enough updates at least till W for the concerned data.

Note: In our discussion server and replica is interchangeable.

Terminologies

- DB(S,t) represent the ordered sequence of Writes(WID) that have been received by server S at or before time t. If t is known to be the current time, then it may be omitted leaving DB(S) to represent the current ordered sequence of Writes(WID) of the server's database.
- A set of Writes completely determines the result of a Read if the set includes "enough" of the database's Writes so that the result of executing the Read against this set is the same as executing it against the whole database.
- Example: Assume in a server S the DB(S, t) = (W1,W2,W3,W4,W5). And we say the write set (W1,W3) completely determines the result of a Read R1 then it mean If we perform this read R1 operation in a server which have only this write set (W1,W3) then the read result will still be the same as this read R1 perform on the whole write set DB(S, t)

Monotonic Reads

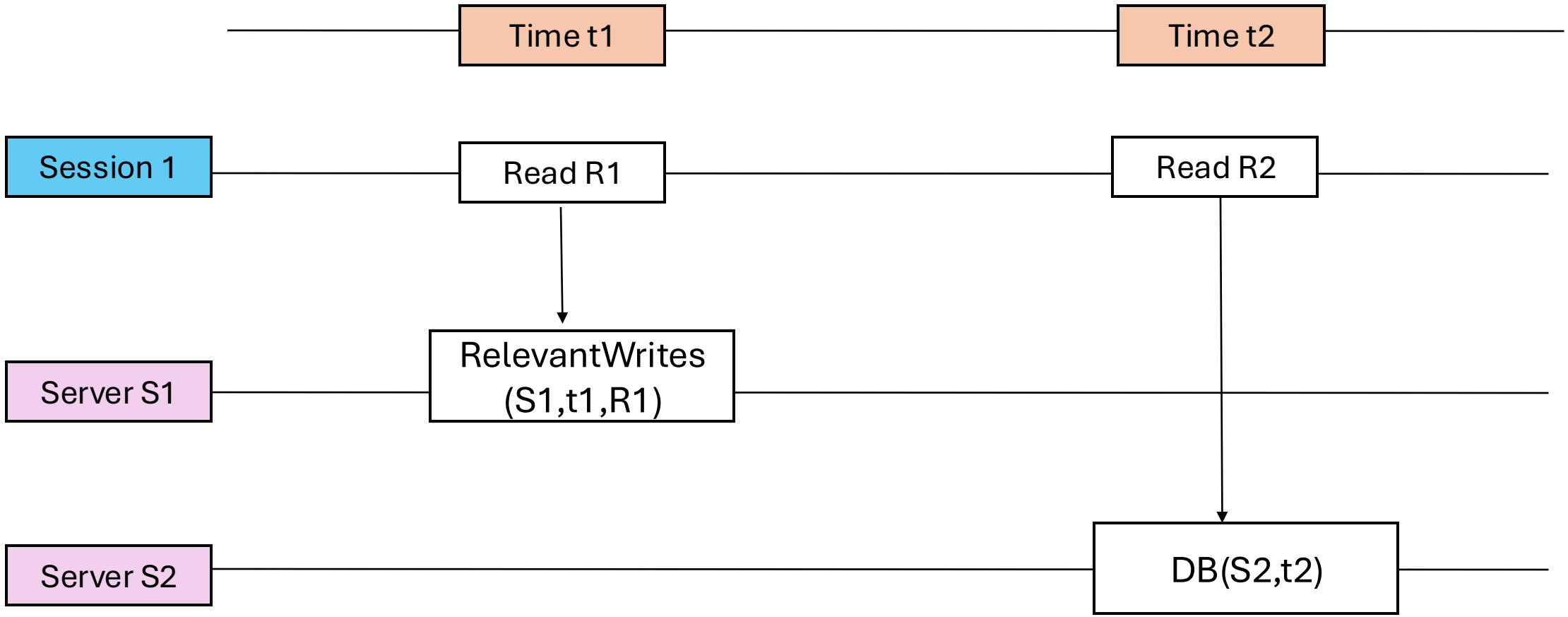
- *RelevantWrites(S,t,R)* denote the function that returns the smallest set of Writes(WID) that is complete for Read R and DB(S, t)
- Intuitively, RelevantWrites(S,t,R) is a smallest set that is "enough" to completely determine the result of R

Formal Definition:-

- If Read R1 occurs before R2 in a session and R1 accesses server S1 at time t1 and R2 accesses server S2 at time t2, then the RelevantWrites(S1,t1,R1) is a subset of DB(S2,t2).

Note: we guess that a read may have multiple relevant writes. if the read reads an entire object, and each of the writes populates one of the fields of the object.

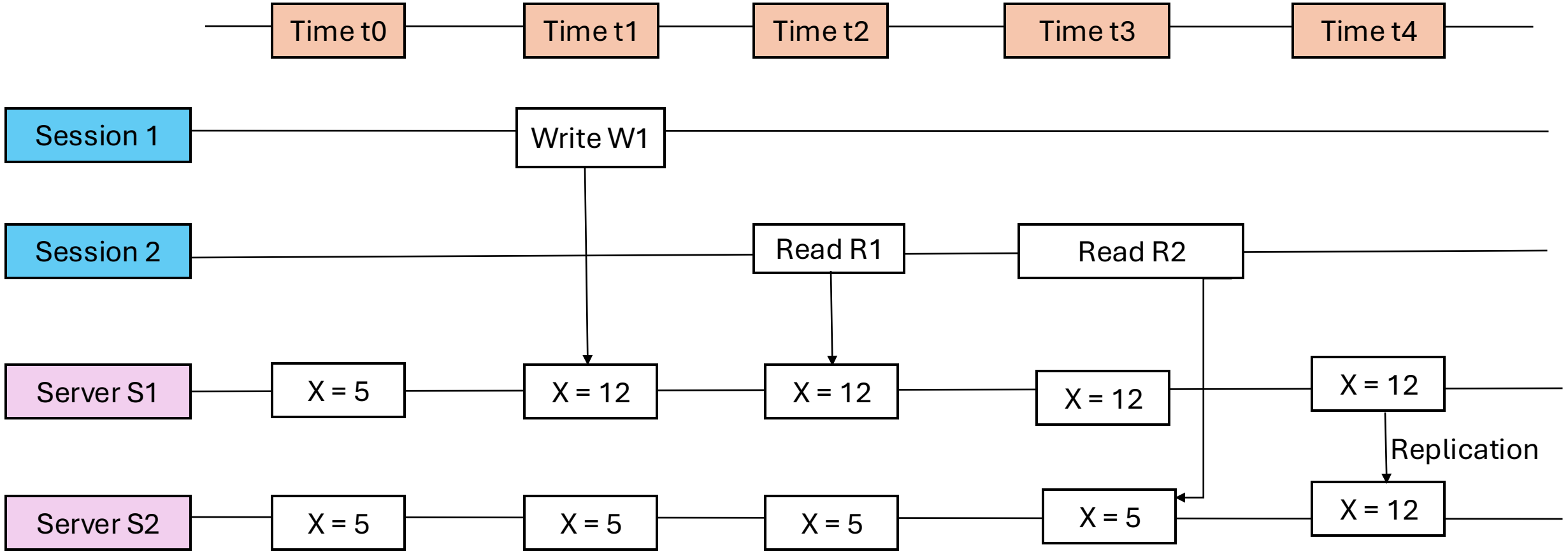
Monotonic Read



RelevantWrites (S1,t1,R1) ⊆ DB(S2,t2)

Monotonic Read guarantee:- DB(S2, t2) should be subset of RelevantWrites(S1,t1,R1)

Monotonic Read Example



RelevantWrites (S1,t2,R1) = {W1} $\not\subseteq$ DB(S2,t3) = {}

Read R2 is not monotonic read

Monotonic Reads

Why it matters?

- Consider a replicated electronic mail database.
- The mail reader issues a query to retrieve all new mail messages and displays summaries of these to the user.
- When the user issues a request to display one of these messages, the mail reader issues another Read to retrieve the message's contents.
- The Monotonic Read-guarantee can be used by the mail reader to ensure that the second Read is issued to a replica that holds a copy of the message. Otherwise, the user, upon trying to display the message, might incorrectly be informed that the message does not exist.

Monotonic Reads

How it differs from RYOW?

- In RYOW, the writes were made by the User1 themselves in their session.
- Here in the example of the mailbox, the new unread mails are not writes made by the User1, but by some other person (user2) in a different session.
- These writes made by User2 now fall in the *RelevantWrites* of User1 first read.
- Monotonic Read guarantees ensure that these RelevantWrites are also present in the replicas accessed by User1 for the second read.
- RYOW did not consider these RelevantWrites.

Monotonic Reads

Properties:

- If we assume, with every write, the version of an object increases, each subsequent Read should see the same or monotonically increasing version of the object.
- As long as the same server handle the requests for the same unit of execution in the same session, Monotonic Read should be guaranteed.

Write Follows Read(WFR)

- The Writes Follow Reads guarantee ensures that traditional Write/Read dependencies are preserved in the ordering of Writes at all servers.
- That is, in every copy of the database, Writes made during the session are ordered after any Writes whose effects were seen by previous Reads in the session(Will see in detail).

Write Follows Read

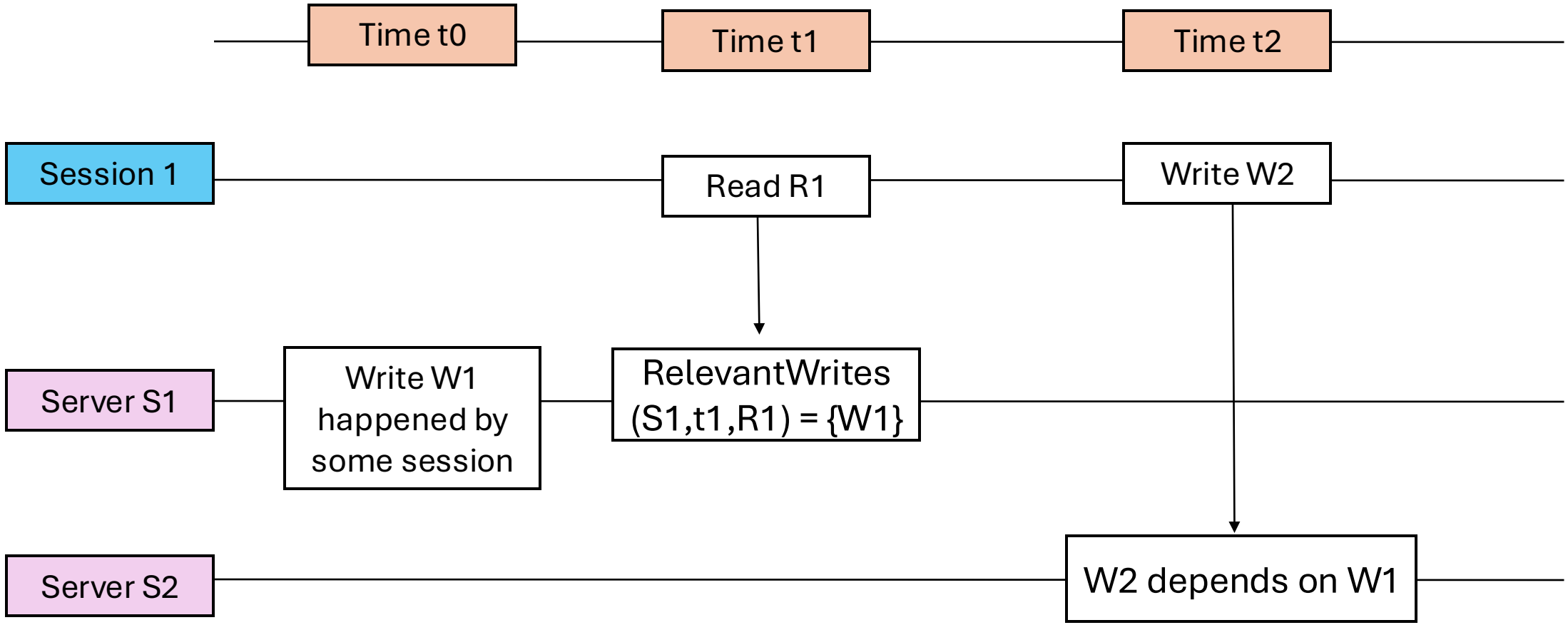
- Let’s say, In a replica S1 at time t0 Write W1 happened on a data.
- And in a session on the same data Read R1 occurs and that R1 accesses server S1 at time t1, i.e., R1 has seen the effect of W1.
- In the same session Write W2 happened and that W2 accesses server s2 at time t2.
- And W2 depends on R1.

In such a case, WFR provides 2 guarantees:

- Ordering guarantee
- Write propagation guarantee

SET UP

Note:- R1 depends on W1. And W2 depends on R1. So, W2 depends on W1

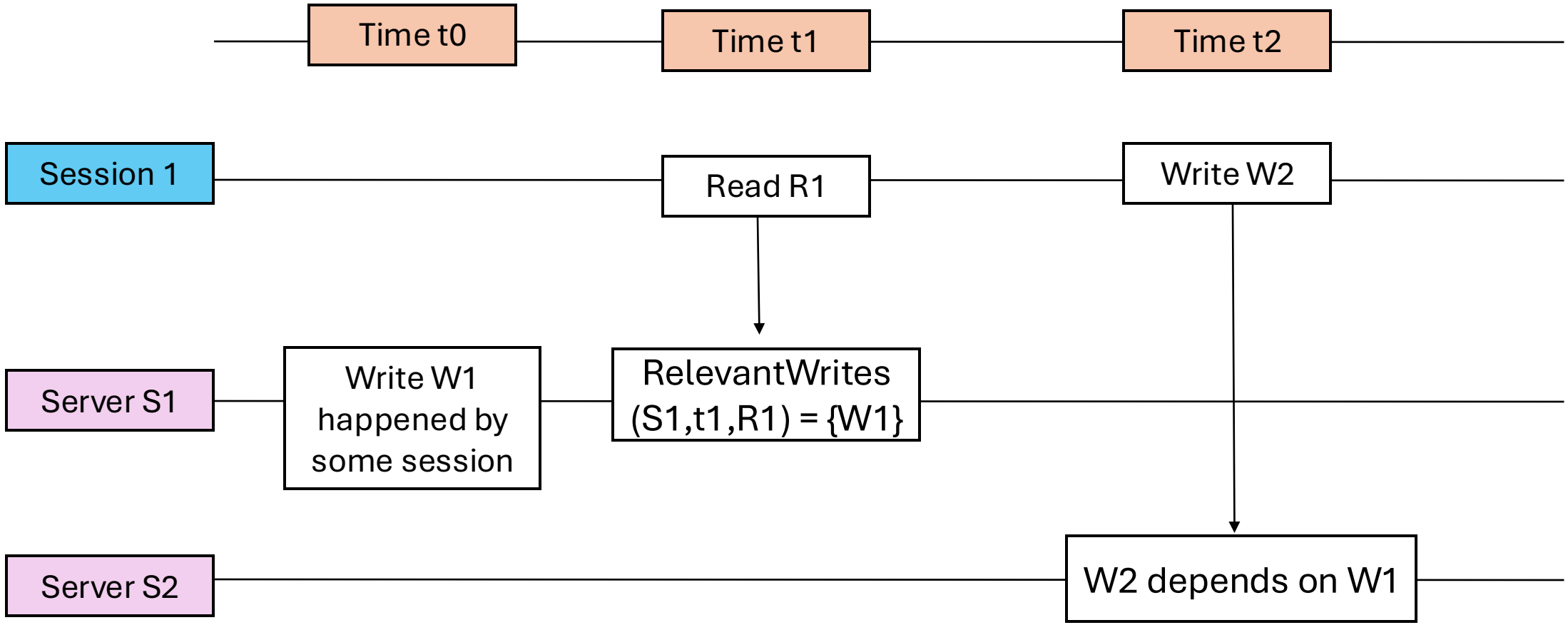


Write Follows Read

- **Ordering guarantee**: W2 should follow all those Writes **which caused** R1 (**all relevant Writes** to R1). In our case, W1 is the only relevant Write for R1. Hence, the Write order should place W2 after W1 in S2. **This guarantee applies within the session**.

Ordering Guarantee

Note:- R1 depends on W1. And W2 depends on R1. So, W2 depends on W1



Ordering guarantee:- W2 should place after W1 in S2

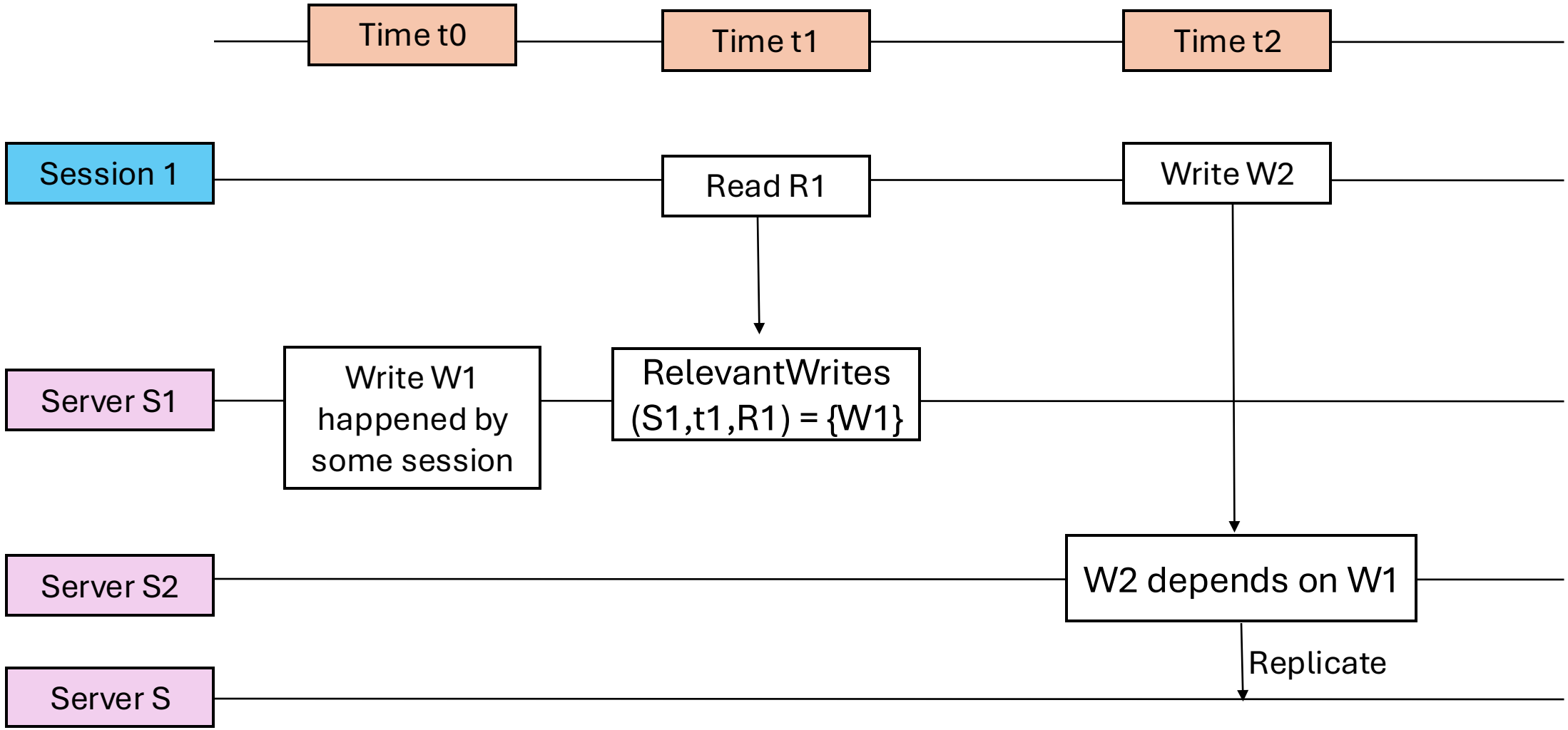
Write Follows Read

- **Write propagation guarantee**: If R1 occurs in replica S1 at time t1, then for any replica S, if S has seen W2 at time t2 > t1, S must have seen all relevant Writes to R1 i.e; in our example, W1 prior to W2. It means all other replicas in the system should apply a Write after they have seen all the previous Writes on which it depends. **This guarantee applies outside of session**.

- **Note:** propagating the changes may take some time depending on implementation, **WFR does not require any real time or instantaneous propagation of changes**.

Write propagation guarantee

Note:- R1 depends on W1. And W2 depends on R1. So, W2 depends on W1



Propagation guarantee:- any replica S, the S must see W1 prior to W2

Write Follows Read

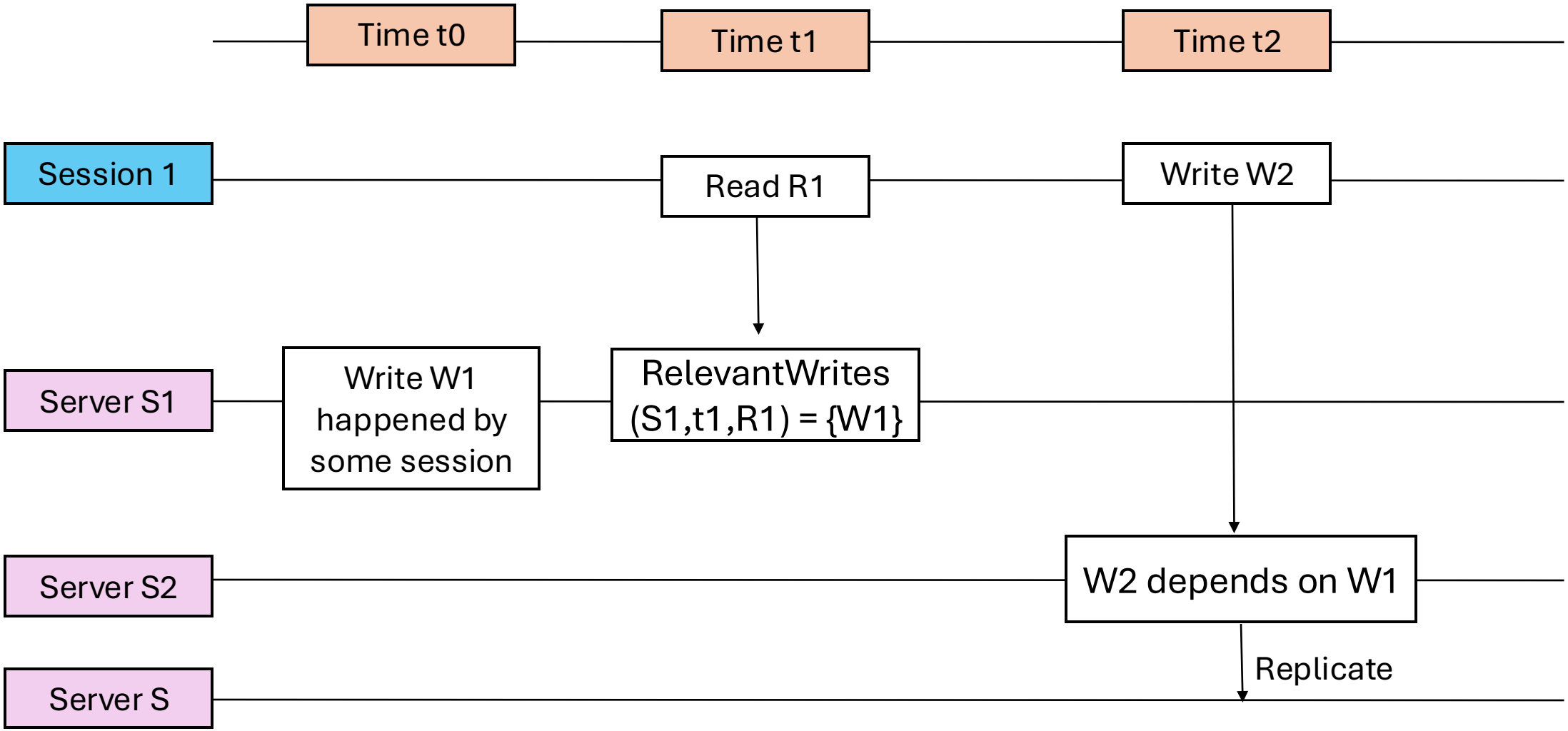
Formal definition:

If Read R1 precedes Write W2 in a session and R1 is performed at server S1 at time t1, then, for any server S, if W2 is in DB(S) then any W1 in RelevantWrites(S1,t1,R1) is also in DB(S) and WriteOrder(W1,W2) is true.

- This guarantee is different in nature from the previous two guarantees in that it affects users outside the session.
- Not only does the session observe that the Writes it performs occur after any Writes it had previously seen, but also all other clients will see the same ordering of these Writes regardless of whether they are from same session.

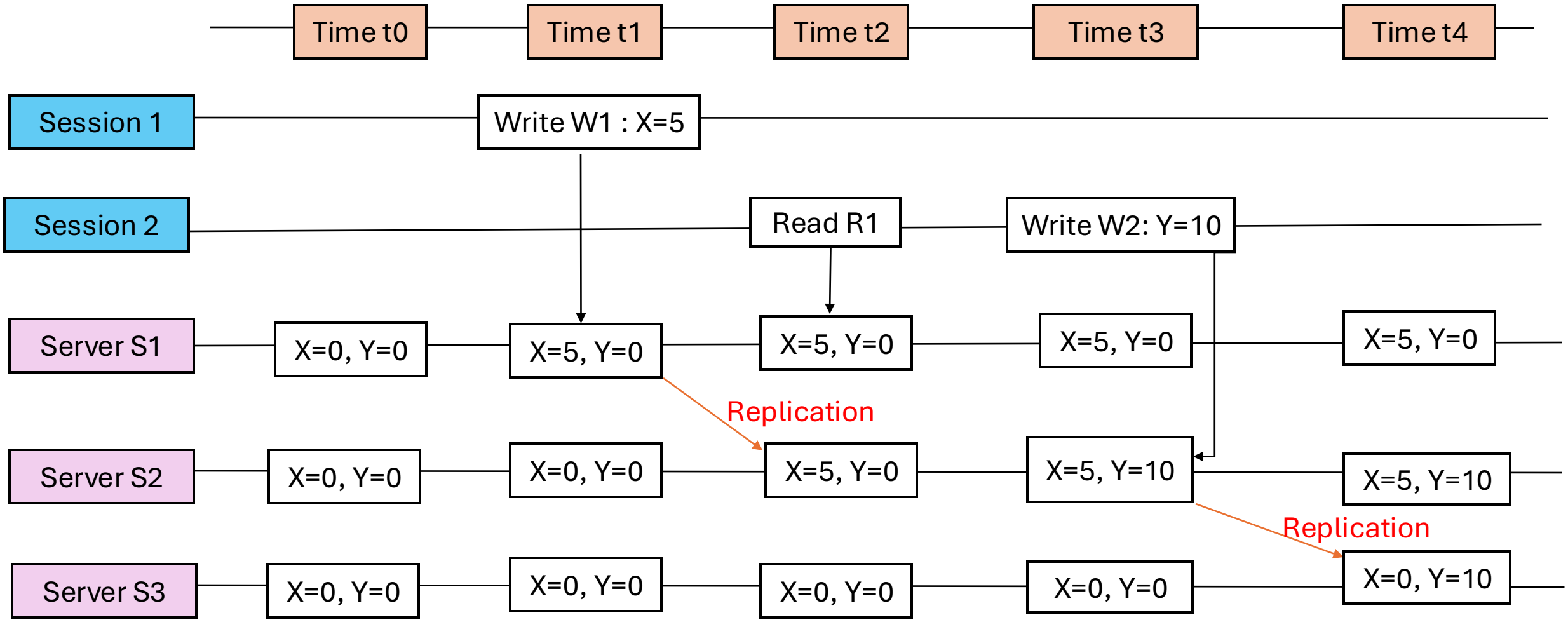
Write Follows Read

Note:- R1 depends on W1. And W2 depends on R1. So, W2 depends on W1



WFR guarantee:- ∀ Server S, If W2 in S then any W1 in RelevantWrites (S1,t1,R1) ∈ DB(S) and WriteOrder(W1,W2) is true

Write Follows Read Example



RelevantWrites(S1,t2,R1) = {W1}

At Time t3 in S2 WriteOrder(W1,W2) = True

At Time t4 in S3 WriteOrder(W1,W2) = False

Write Follows Read not there in S3 at time t4

Write Follows Read

Properties

- Ordering first applies within the involved session.
- Outside of the client's session, propagation of writes in order of their occurrence should be done to guarantee WFR.
- The propagation can be lazy or non-real time thus making the guarantee weak.
- This consistency is also called **session causality**.

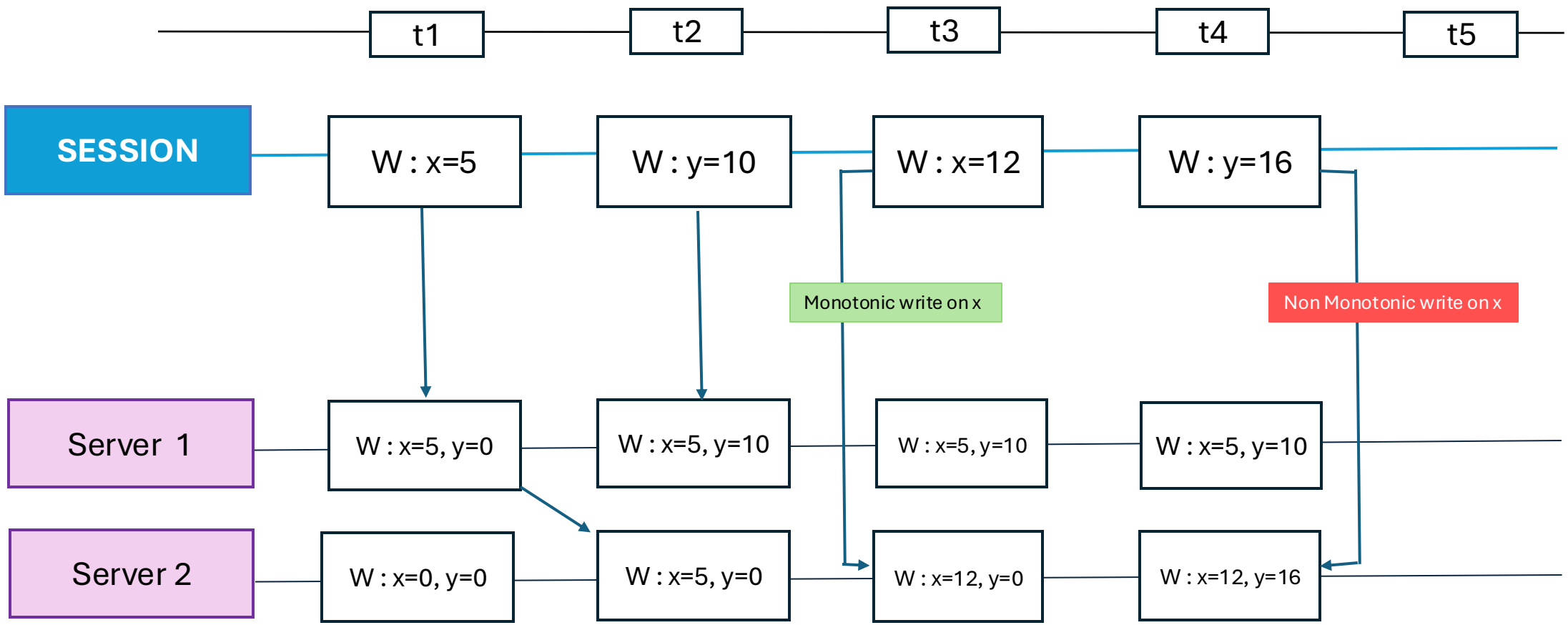
Write Follows Read

Example:-

- Consider replying to a tweet.
- You can only do that when the tweet is already written to the system and is visible to you.
- Both reading and replying could be done in the same session.

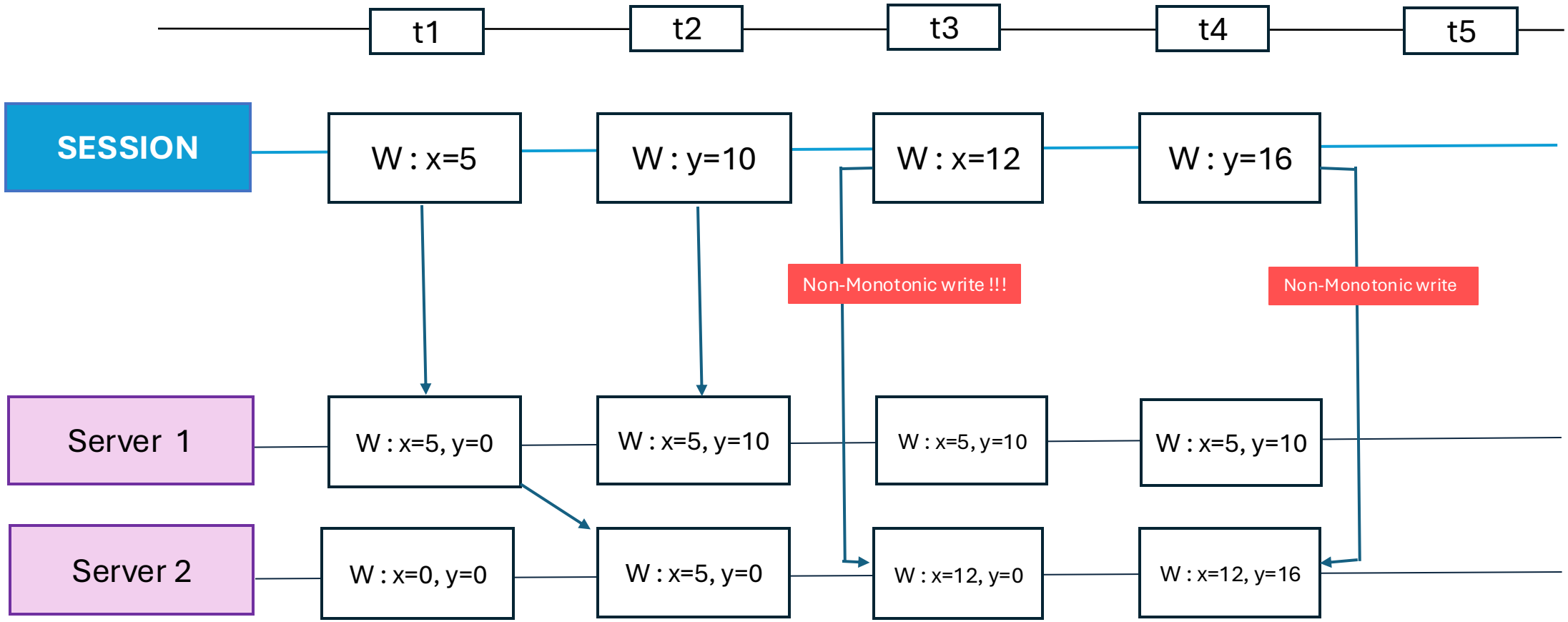
Monotonic Write

A write operation by a session on a data item x is completed before any successive write operation on x during the session. Implies a copy must be up to date before performing a write on it.



Stricter form of monotonic write *(as per Terry paper)*

If Write W1 precedes Write W2 in a session, then, for any server S2, if W2 in DB(S2) then W1 is also inDB(S2) and WriteOrder(W1,W2).



Breaking into its Principle Components

Ordering Guarantee:

A session should **see its own successive updates on a particular variable / object in the order of their occurrence**. This guarantee applies within the session.

Propagation Guarantee:

Eventually, all other replicas should see the writes on the object in the same order. **This applies outside of the session**.

When and why do you need monotonic write

The database contains software source code and is replicated across multiple servers.

Upgrading a Library (Upward Compatibility)

- A programmer updates a library, adding new functionality in an **upward compatible** way.
- Since the update does not break existing client software, it can be **propagated lazily** to other servers.

Updating an Application (Dependency Issue)

- The programmer also updates an application to use the **new functionality** from the library.
- If the updated application is written to servers **before** they receive the updated library, it will **fail to compile** due to missing dependencies.

Ensuring Consistency with MW-Guarantee

- To prevent this issue, the programmer can create a **new session** with an **MW-guarantee** (Monotonic Writes).
- This ensures that **both** the updated library and the updated application are written within the same session.
- As a result, all servers will have the required dependencies, preventing compilation failures.

Implementation of SESSIONAL GUARANTEES

The rules on how the session's read or write requests should be handled to give sessional guarantees

SETUP

The implementations require cooperation from the servers that process Read and Write operations.

Specifically, a server must be willing to return

1. **the unique identifier (WID) assigned to a new Write**
2. **the set of WIDs for Writes that are relevant to a given Read, and the set of WIDs for all Writes in its database**.

For each session, it maintains two sets of WIDs:

read-set : set of WIDs for the Writes that are relevant to session Reads

write-set : set of WIDs for those Writes performed in the session

relevant-writes**:** is a smallest set that is "enough" to completely determine the result of R

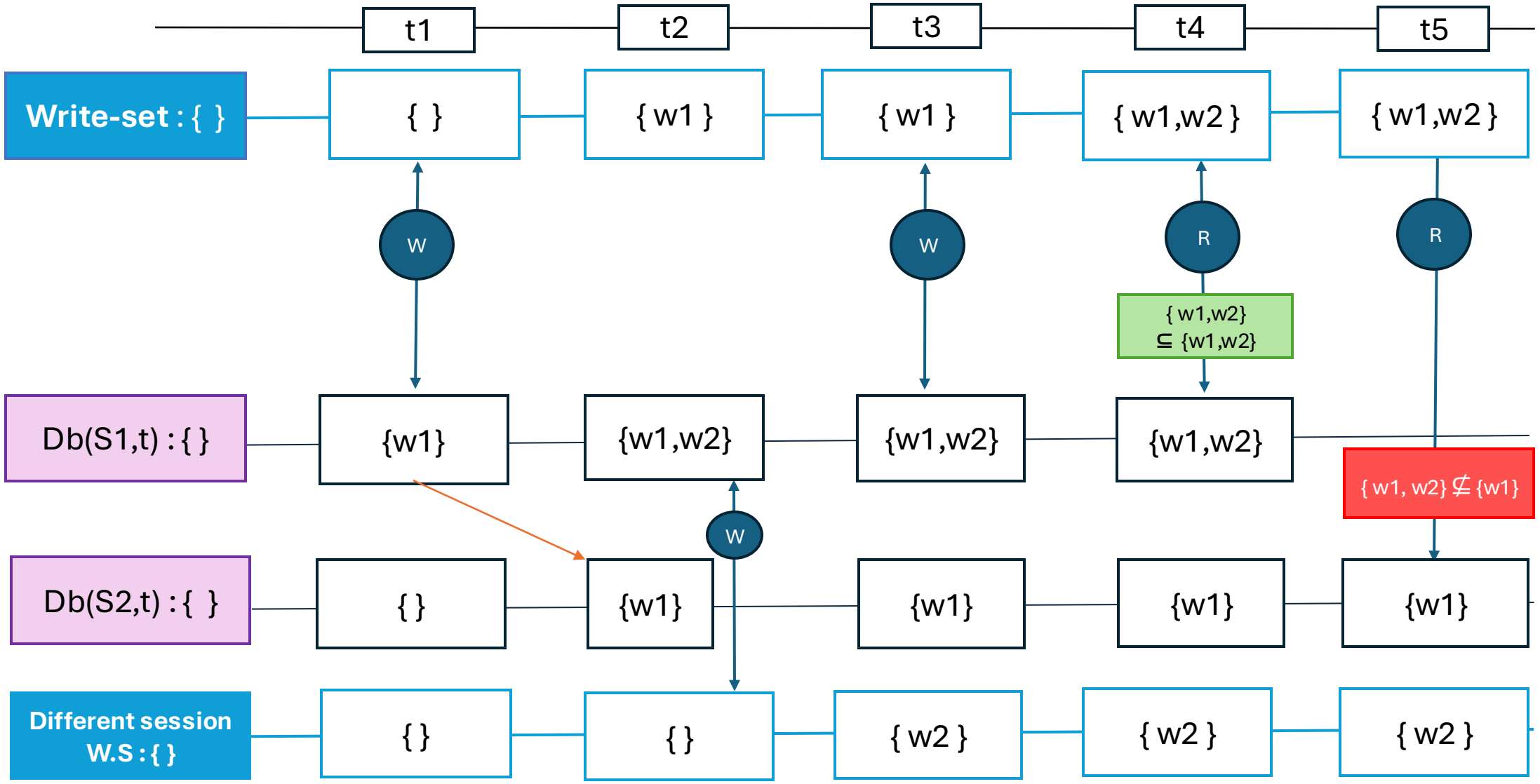
Table 1: Read/Write guarantees

Guarantee	session state updated on	session state checked on
Read Your Writes	Write	Read
Monotonic Reads	Read	Read
Writes Follow Reads	Read	Write
Monotonic Writes	Write	Write

Read Your Write Implementation Steps

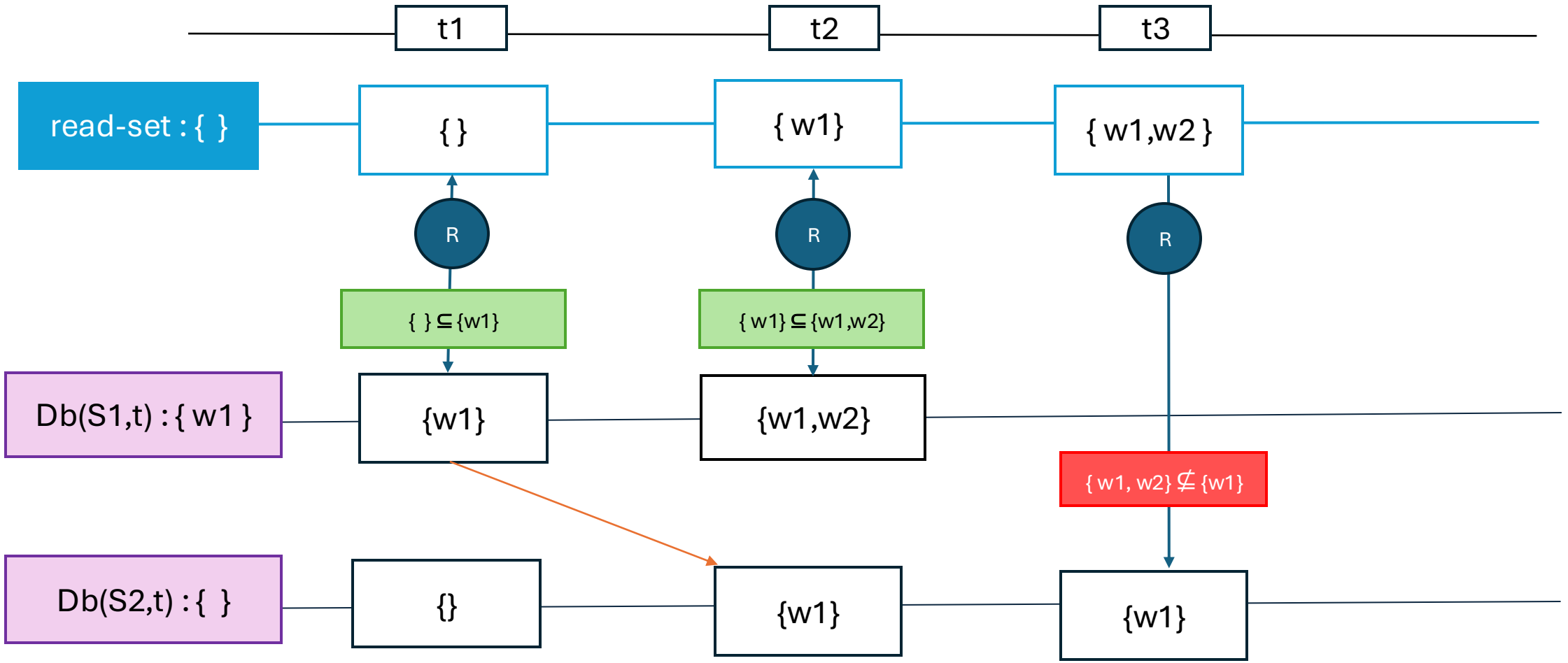
1. On writes wid added to write set (ws=ws+wid)
2. **Before** read check ws ⊆ Db(S,t)

RYW-guarantee:
If Read R follows Write W in a session and R is performed at server S at time t, then W is included in DB(S,t)



MR implementation

1. **Before** read check rs ⊆ Db(S,t)
2. Add the relevent writes to the read set (rs=rs+wid) . W2 is from another session

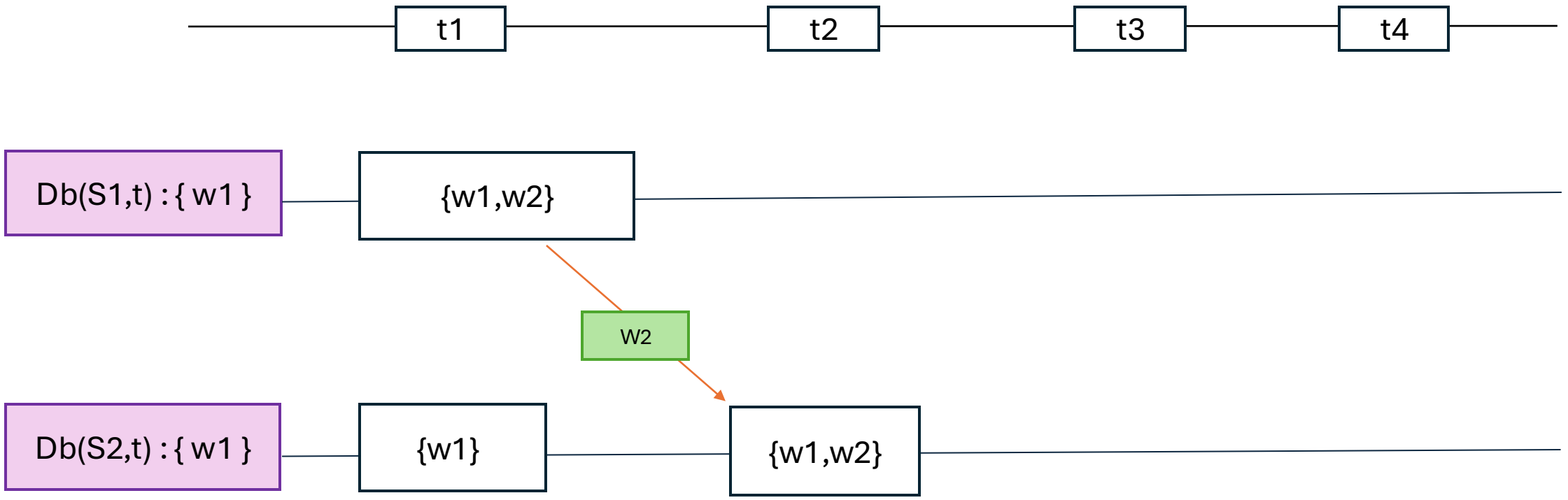


MR-guarantee:
If Read R1 occurs before R2 in a session and R1 accesses server S1 at time t1 and R2 accesses server S2 at time t2, then RelevantWrites(S1,t1,R1) is a subset of DB(S2,t2).

Additional Constraints For Write Follows Read and Monotonic Write

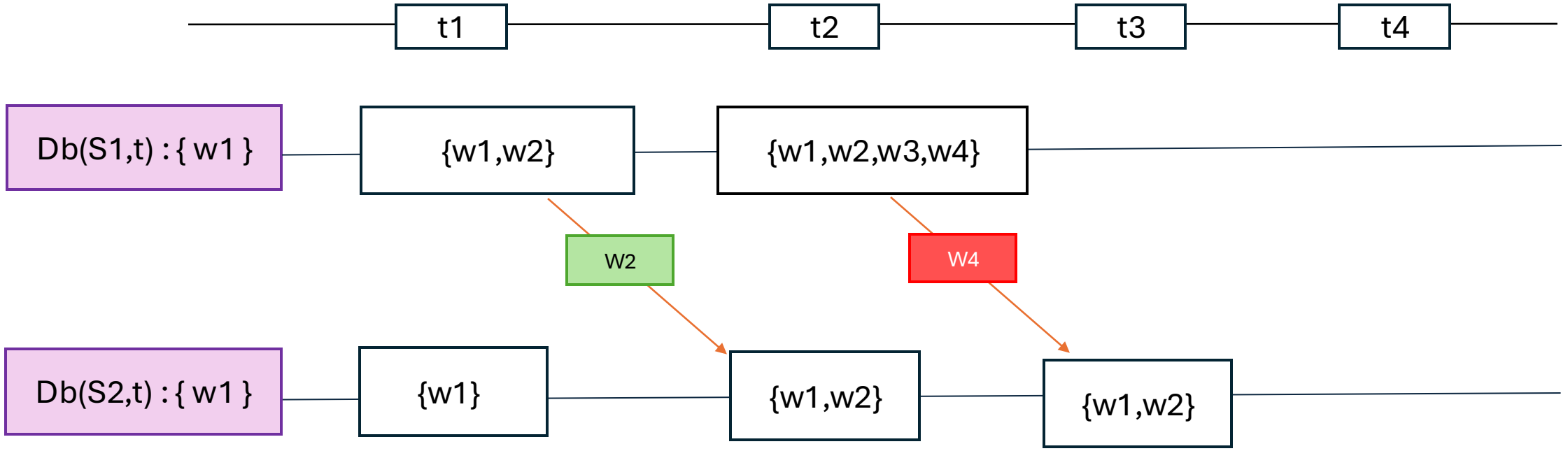
C1. When a server S accepts a new Write W2 at time t, it ensures that WriteOrder(W1,W2) is true for any W1 already in DB(S,t). That is, new Writes are ordered after Writes that are already known to a server

C2. Propagation of writes is performed such that if W2 is propagated from server S1 to server S2 at time t then any W1 in DB(S1,t) such that WriteOrder(W1,W2) is also propagated to S2



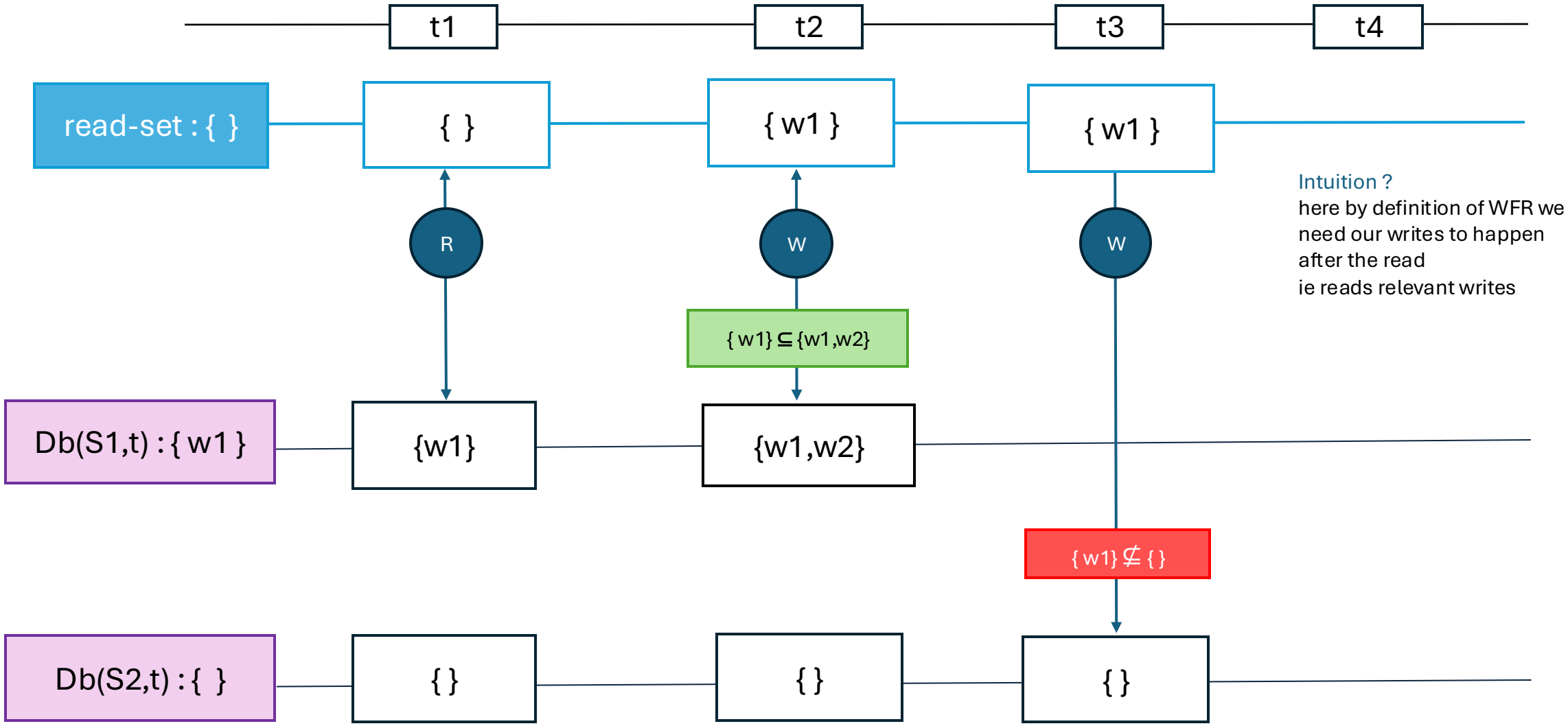
C2. Propagation of writes is performed such that if W2 is propagated from server S1 to server S2 at time t then any W1 in DB(S1,t)such that WriteOrder(W1,W2) is also propagated to S2

Similarly here writes are performed by same or different sessions



WFR IMPLEMENTION STEPS

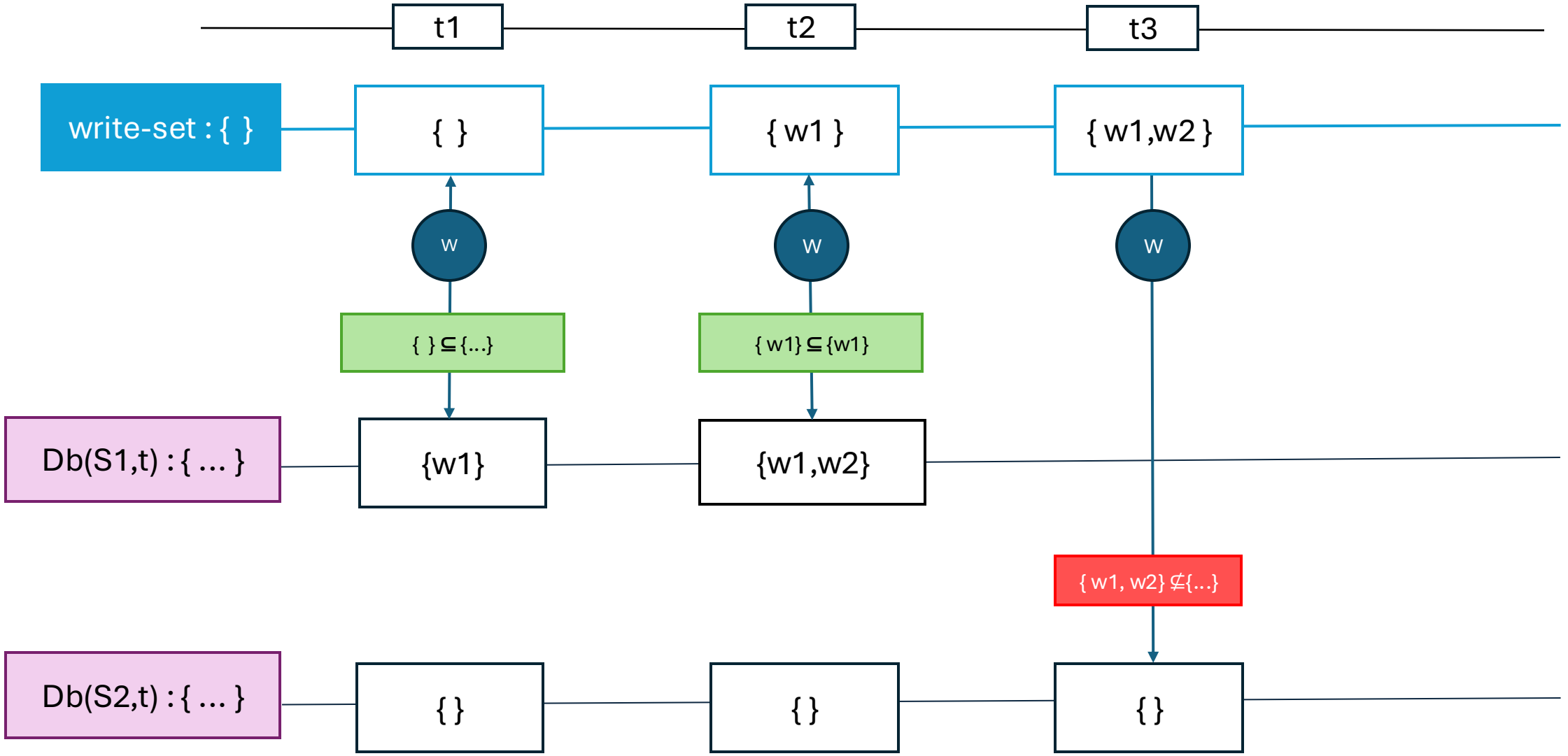
1. **Before** write check Read Set ⊆ Db(S,t)
2. if read successful update read set with relevant-writes of read data item



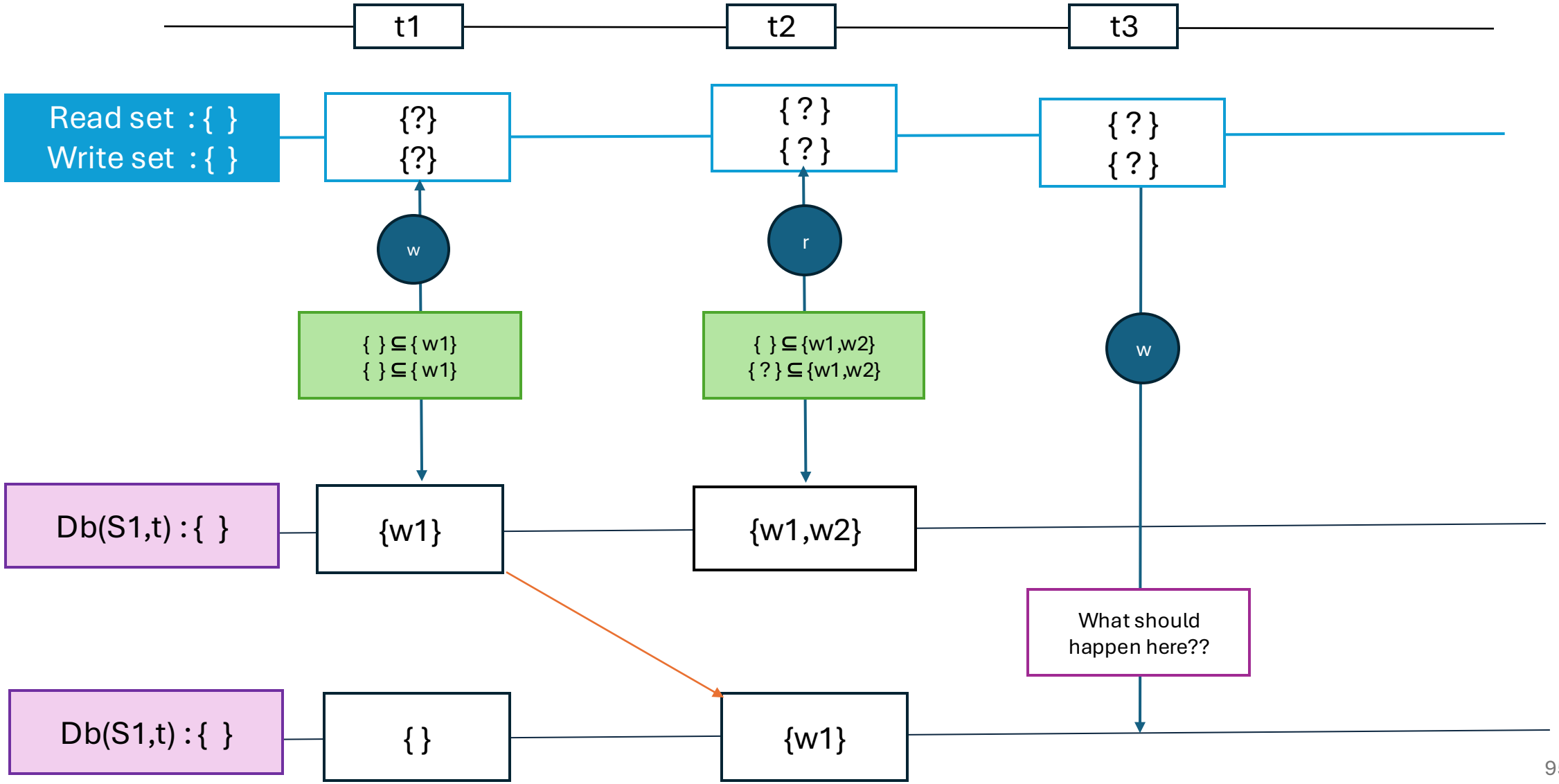
WFR-guarantee: If Read R1 precedes Write W2 in a session and R1 is performed at server S1 at time t1, then, for any server S2, if W2 is in DB(S2) then any W1 in RelevantWrites(S1,t1,R1) is also in DB(S2) and WriteOrder(W1,W2)

MW IMPLEMENTATION STEPS

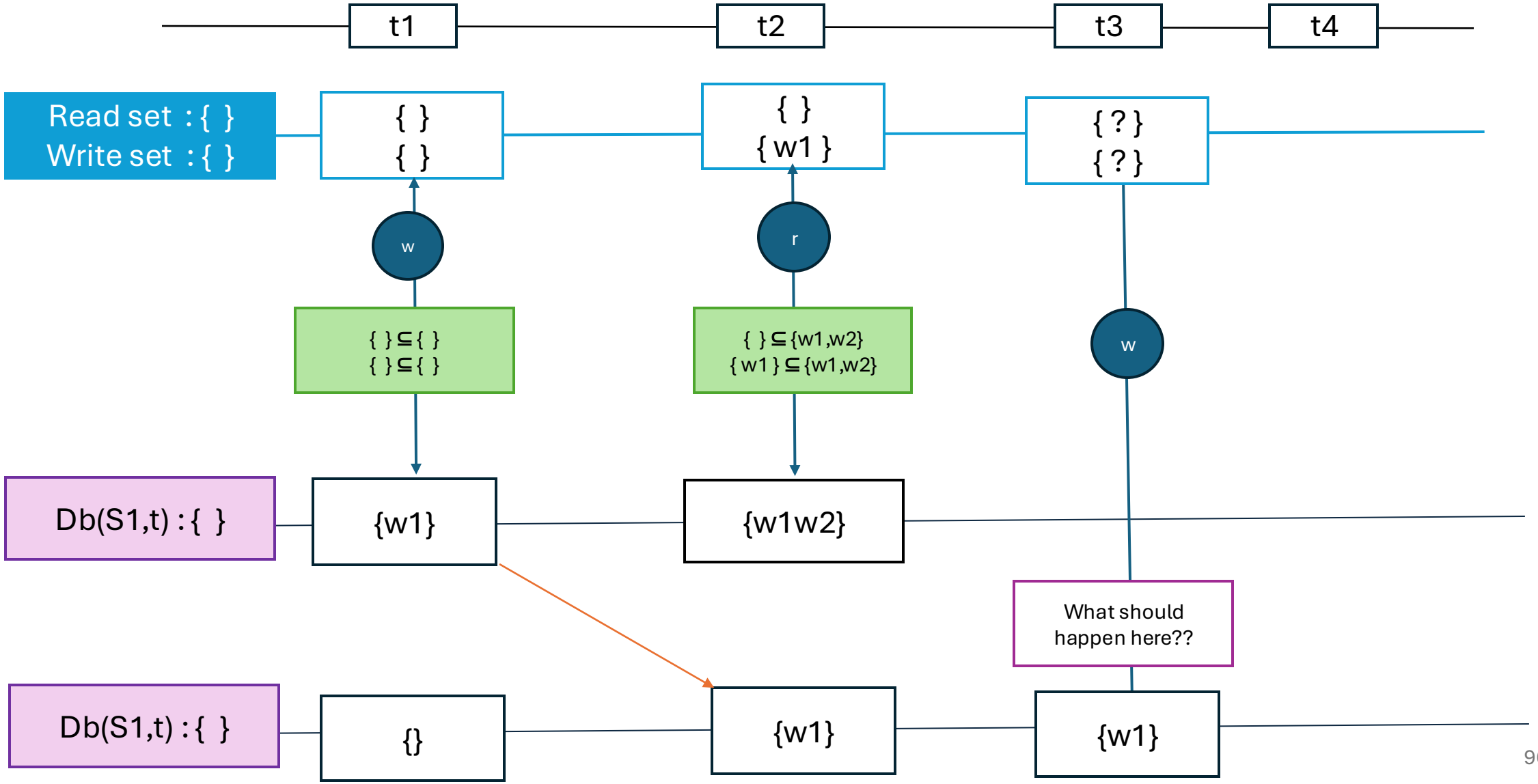
1. On write check write set ⊆ Db(S,t)
2. After write accepted by server add wid to the write set (ws=ws+wid)



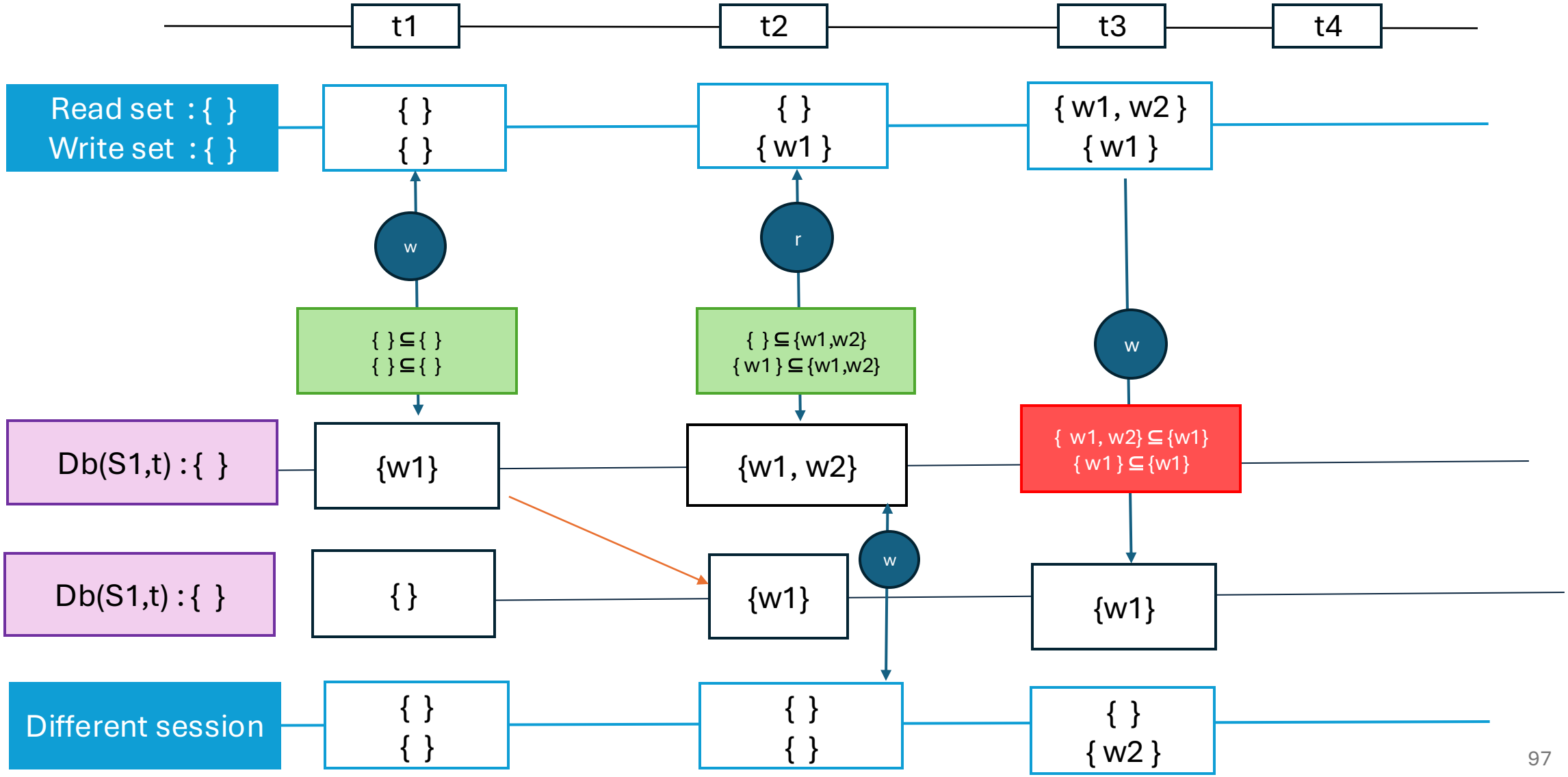
Example with full consistency with operation on same data item



Example with full consistency with operation on same data item



Example with full consistency with operation on same data item

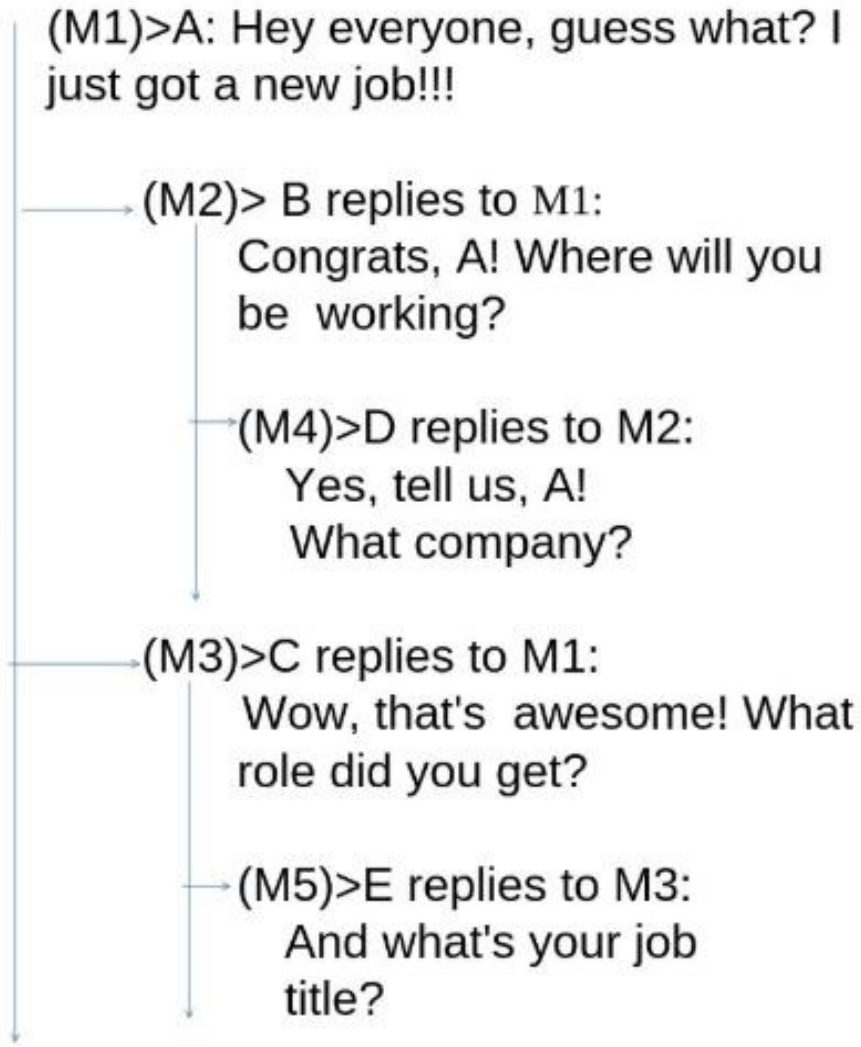


A Chat Between Friends

Consider chat in a group between 5 friends:

A,B,C,D,E.

Let the corresponding messages be M1,M2,M3,M4,M5



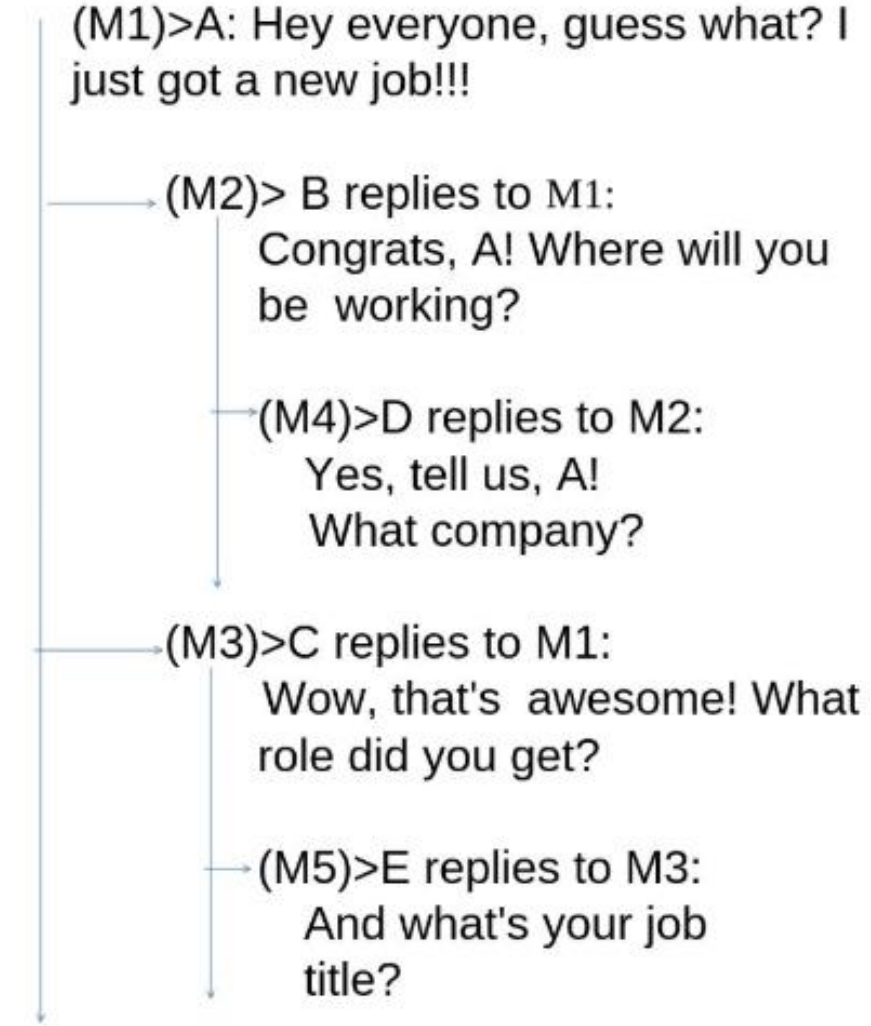
A Chat Between Friends

Consider chat in a group between 5 friends:

A,B,C,D,E.

Let the corresponding messages be M1,M2,M3,M4,M5

Q: Now, Does it make sense to read M5 without first reading M3 and M3 without reading M1?

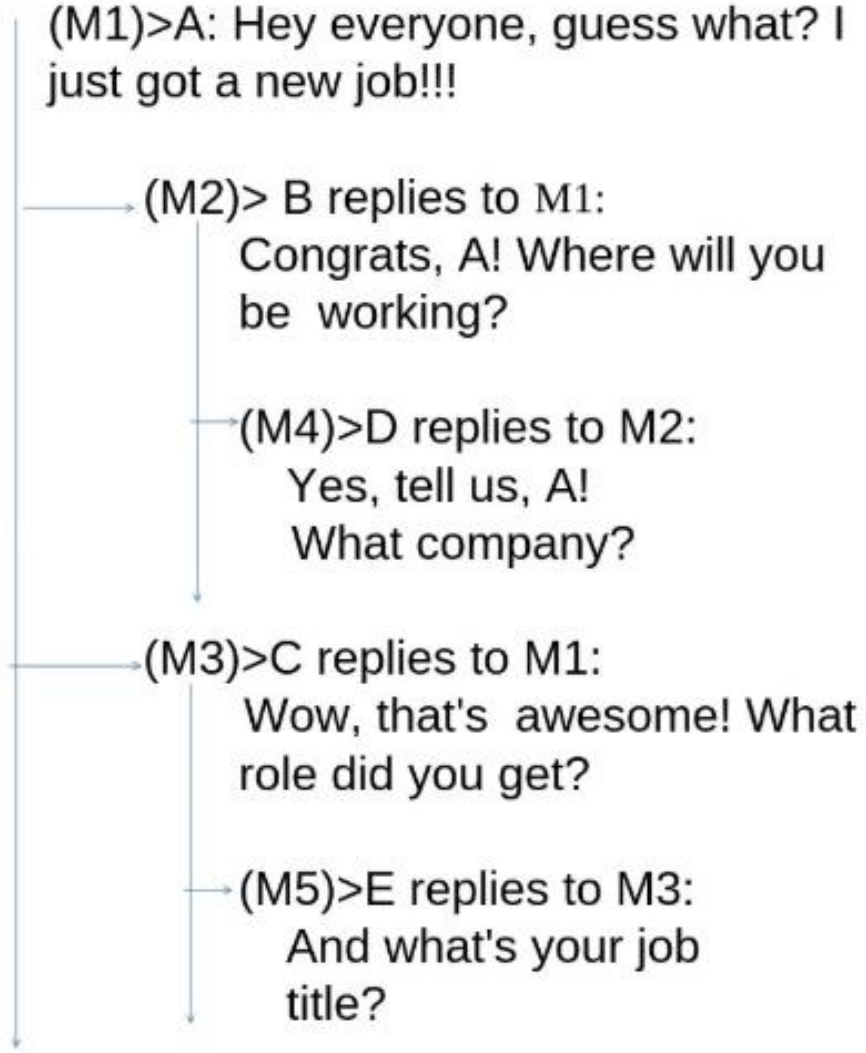


A Chat Between Friends

Q: Now, Does it make sense to read M5 without first reading M3 and M3 without reading M1?

Ans: No, because M1 is the **cause** of M3, M3 is the **cause** of M5, etc.

Remark: To any client (say friend F), it wouldn't make sense to read M5 without reading M3 and M3 wouldn't make sense without reading M1.



Requirement: Capture the sense of cause-and-effect

Requirement: Capture the sense of cause-and-effect

Causal Ordering comes to the Rescue

Causal Ordering

Let ***E*** be the **set of events** in a distributed system. Define the relation → **(called happens-before)** on E as follows:

Local Ordering:
If event a occurs before event b and both the events are initiated by the same UOE, then

$$a \rightarrow b$$

Ordering across UoEs:
When one UOE performs a write (event a) and then if another UOE later observes that update and performs a subsequent write (event b), we say that event a "happens-before" event b:

$$a \rightarrow b$$

Transitivity: For all $a,b,c \in E$, if $a \rightarrow b$ and $b \rightarrow c$, then:

$$a \rightarrow c$$

A Chat Between Friends

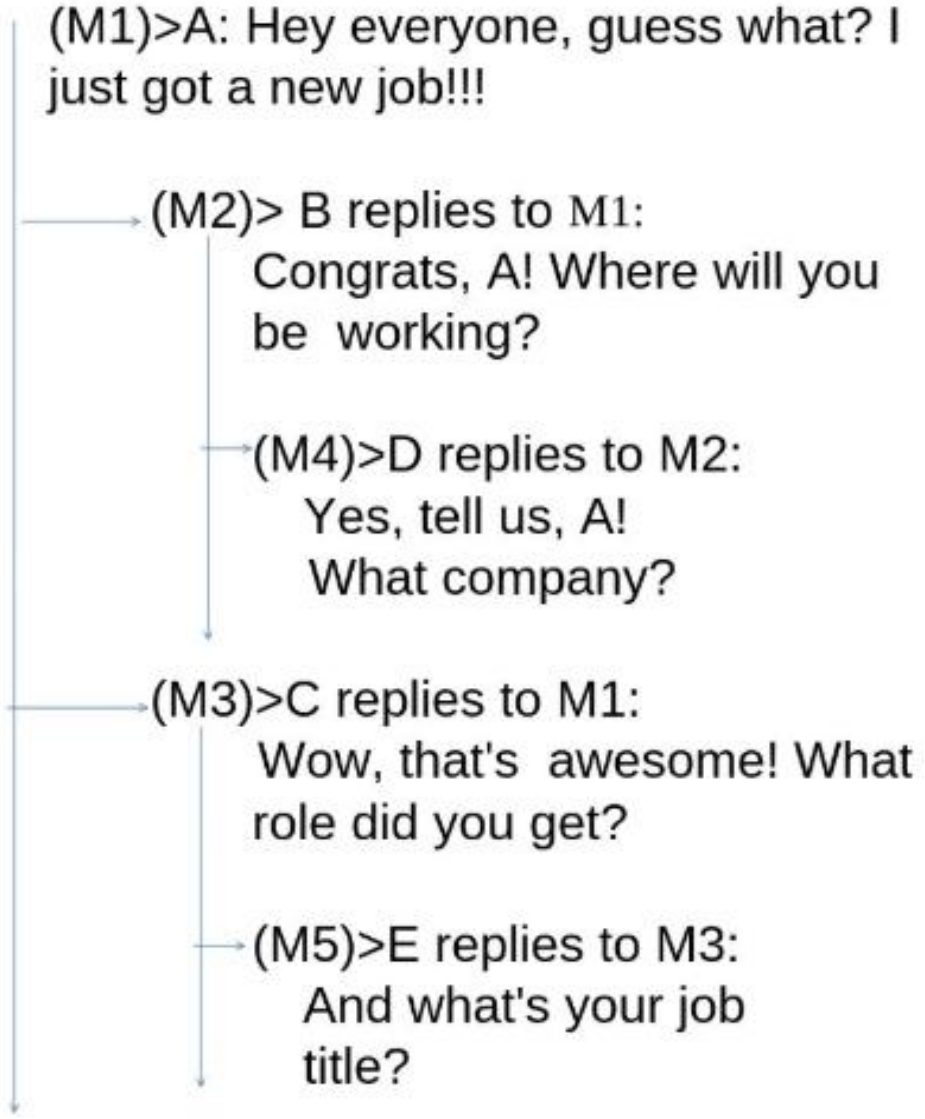
Consider chat in a group between 5 friends:

A,B,C,D,E.

Let the corresponding messages be M1,M2,M3,M4,M5

Using the previously 'happens-before' relationship we can define these causal Orderings:

M1 → M2 → M4



A Chat Between Friends

Consider chat in a group between 5 friends:

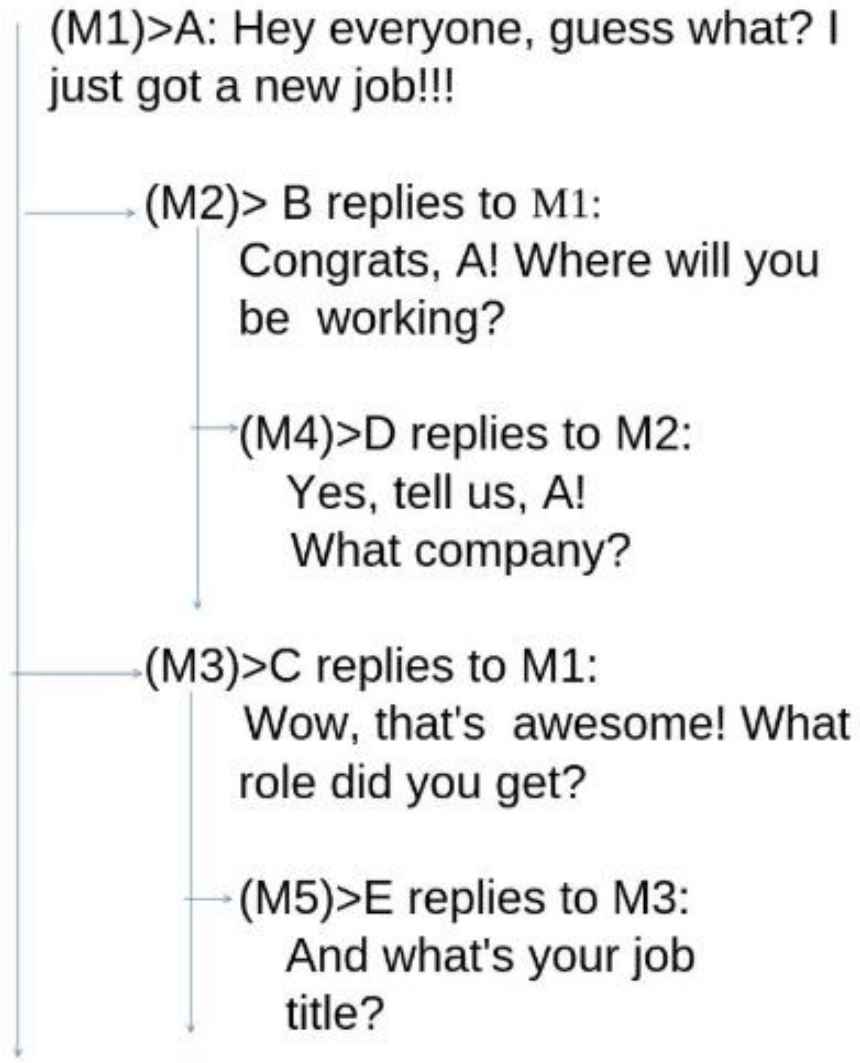
A,B,C,D,E.

Let the corresponding messages be M1,M2,M3,M4,M5

Using the previously 'happens-before' relationship we can define these causal Orderings:

M1 → M2 → M4

M1 → M3 → M5



Requirement: A Consistency Model that enforces Causal Ordering

Causal Consistency Comes to Rescue!

Causal Consistency

Definition:

*Causal consistency enforces ordering of **only related writes** as observed by any UoE*

A system is causally consistent if, for any two writes W1 and W2 :

- If W1→W2 (i.e. W1 causally precedes W2), then every UoE observes W1 before W2.
- If W1 and W2 are concurrent (i.e. neither W1→W2 nor W2→W1), different UoE may observe them in different orders.

What doesn't it guarantee?

Ans: **Total Order of Operations** – Only causally related operations are ordered, meaning two causally independent writes can be observed in different orders by different UoEs.

(M1)>A: Hey everyone, guess what? I just got a new job!!!

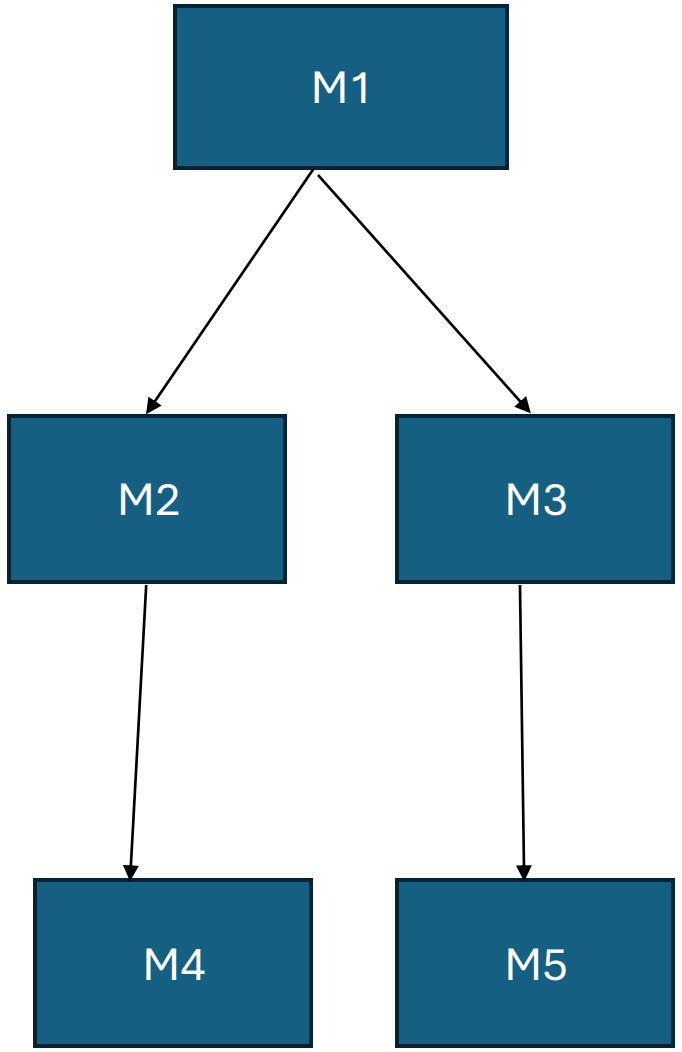
(M2)> B replies to M1:
Congrats, A! Where will you be working?

(M4)>D replies to M2:
Yes, tell us, A!
What company?

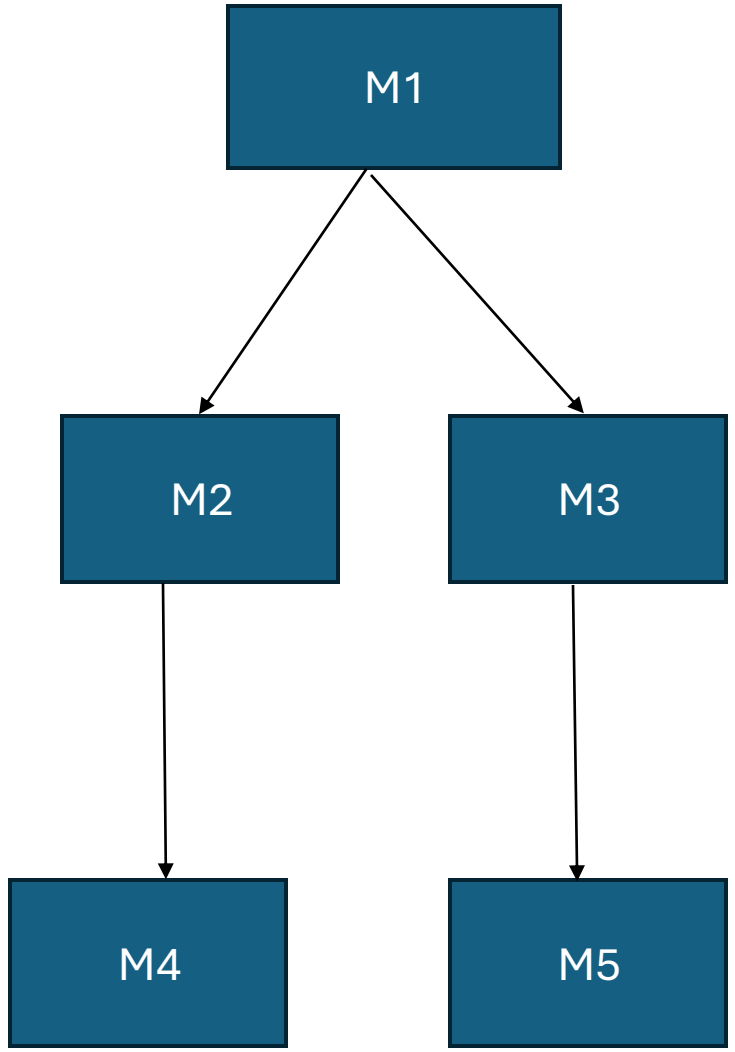
(M3)>C replies to M1:
Wow, that's awesome! What role did you get?

(M5)>E replies to M3:
And what's your job title?

Mi → Mj
if Mj is a reply to Mi



The chat can be represented as this tree.
And the arrows represent the replies.



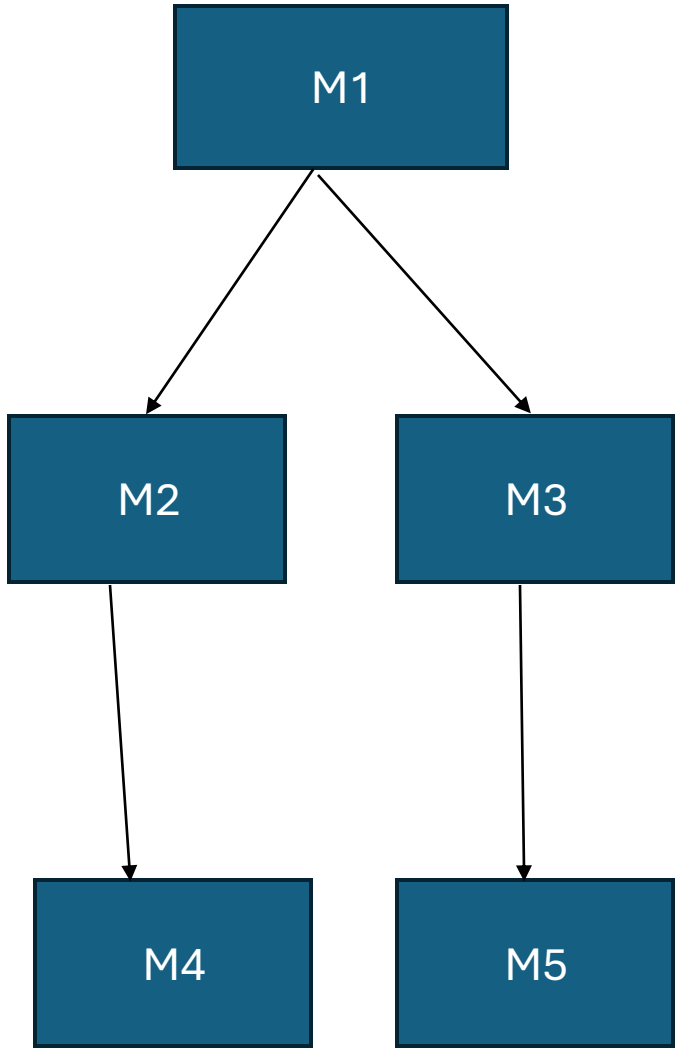
Now, as established earlier, we saw that the

M1 → M2 → M4
M1 → M3 → M5

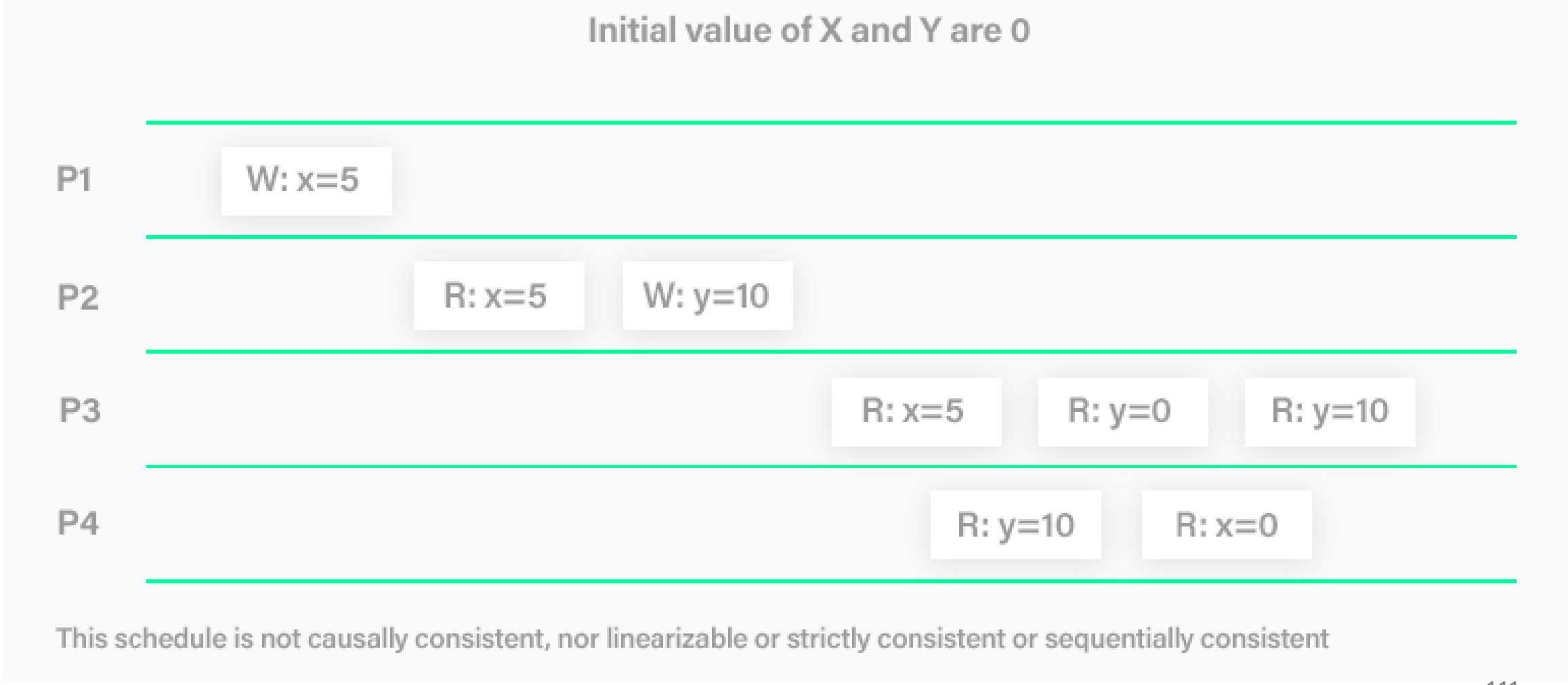
So, any UoE that observes the series of messages, must observe M1 at the very first.

The messages at the same level of the tree, (M2 and M3) or (M4 and M5) can be processed in any order.

Notably among M2, M3 and M5; M2 can be processed before, after, or even in between M3 and M5, as there is no causal ordering between M2 and either of M3 and M5.



Counter-Example: Broken Causal Consistency

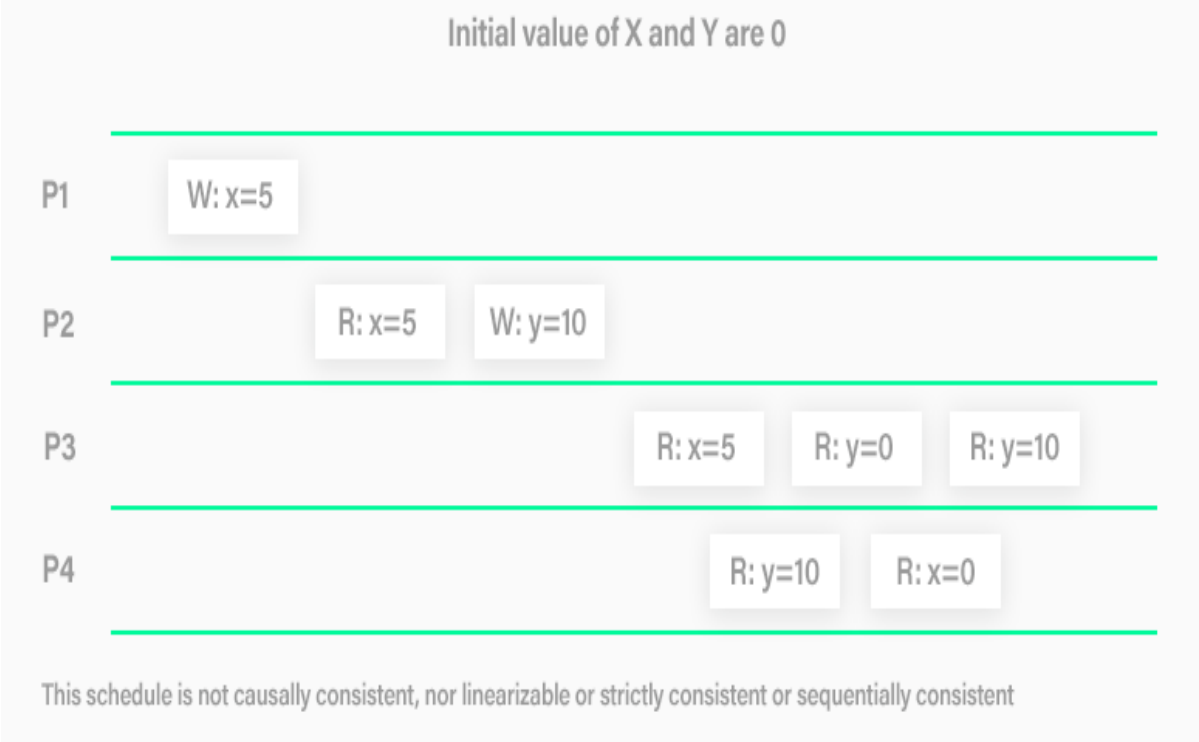


Understanding Causal Consistency
Scenario:

1. **P1 writes** x = 5 to the system.
2. **P2 reads** x = 5 and then **writes** y = 10.

y depends on **x**, so the write of **y** by **P2** is causally dependent on the earlier write of **x** by **P1**.

Symbolically: (W: x=5) → (W: y=10)

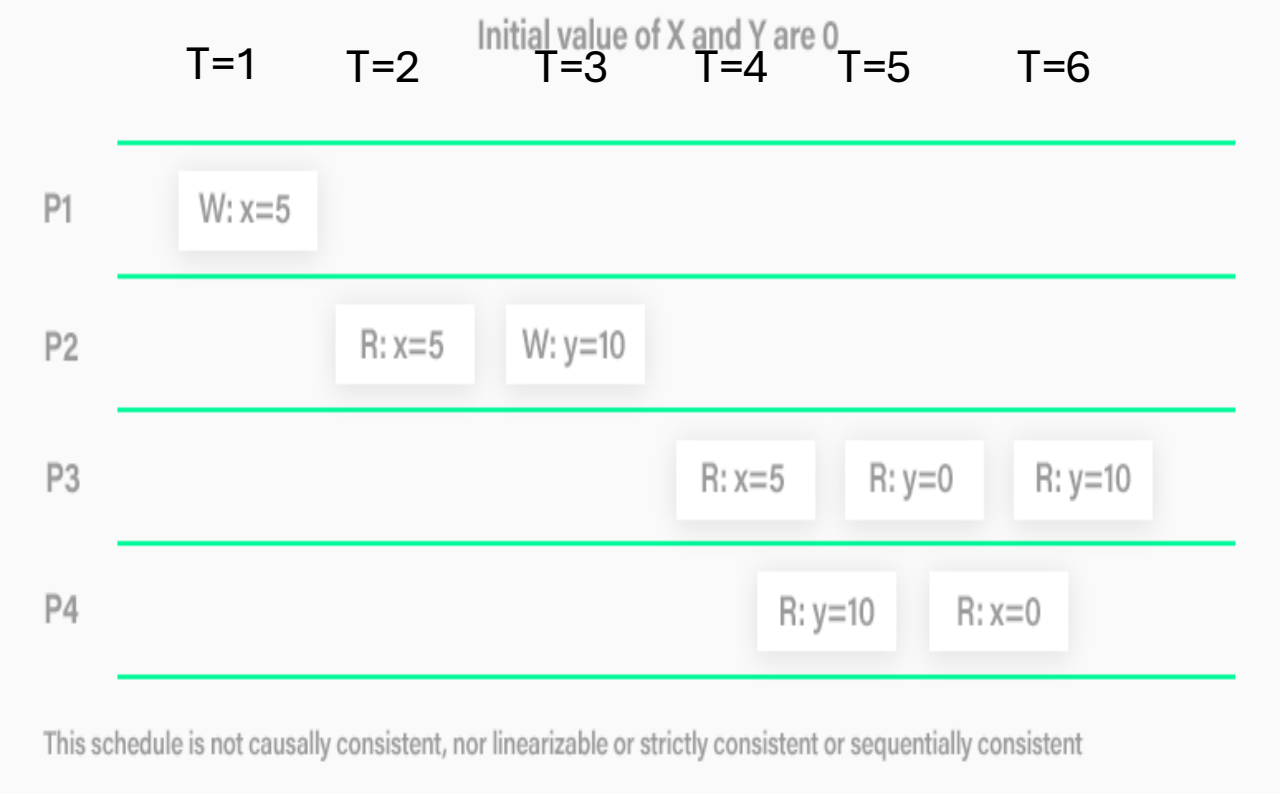


Understanding Causal Consistency Scenario:

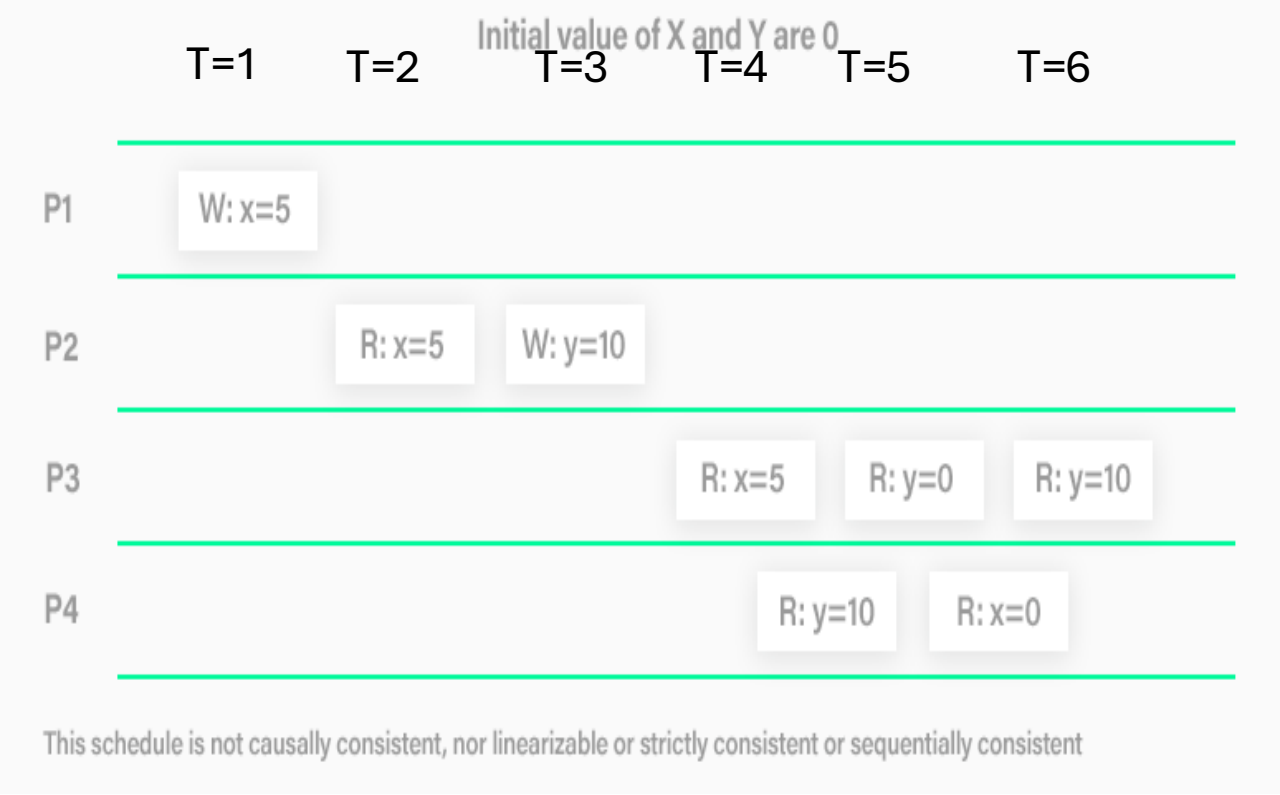
Now, lets assume that another UOE **P3** reads
x and y in this sequence:

- **At T=4 :** x=5
- **At T=5:** y=0
- **At T=6 :** y=10

Q: Is the Causal Consistency Preserved in this Scenario?



Understanding Causal Consistency Scenario:



Now, lets assume that another UOE **P3** reads **x and y** in this sequence:

- **At T=4 :** x=5
- **At T=5:** y=0
- **At T=6 :** y=10

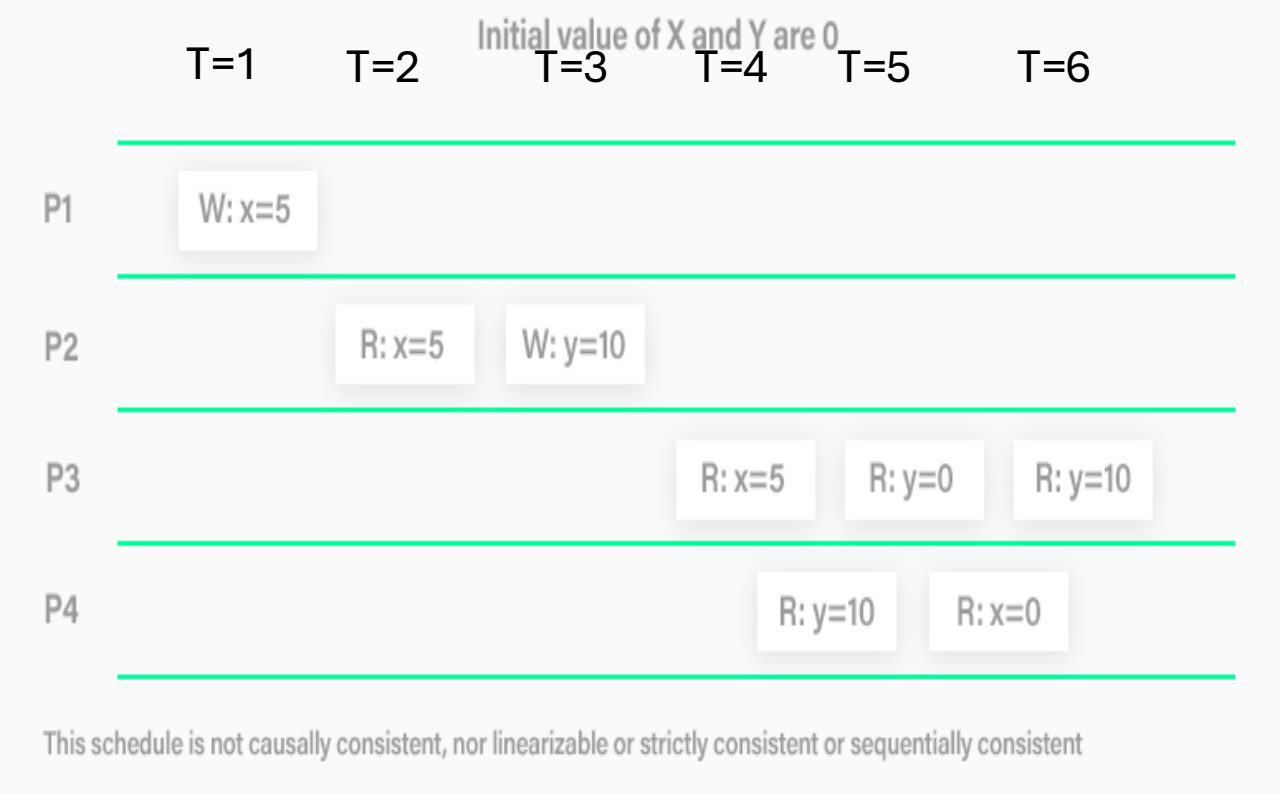
Q: Is the Causal Consistency Preserved in this Scenario?

A: Yes, because the value of (y=10) is read only after the value of (x=5) is read. The fact that y=0, which is stale, was read earlier , does not concern us because, according to the causal relation established earlier, only this is ensured: (W:x=5) -> (W:y=10), (Not the Other way around).

Understanding Causal Consistency Scenario:

Now, lets assume that another UOE **P4** reads **x and y** in this sequence:

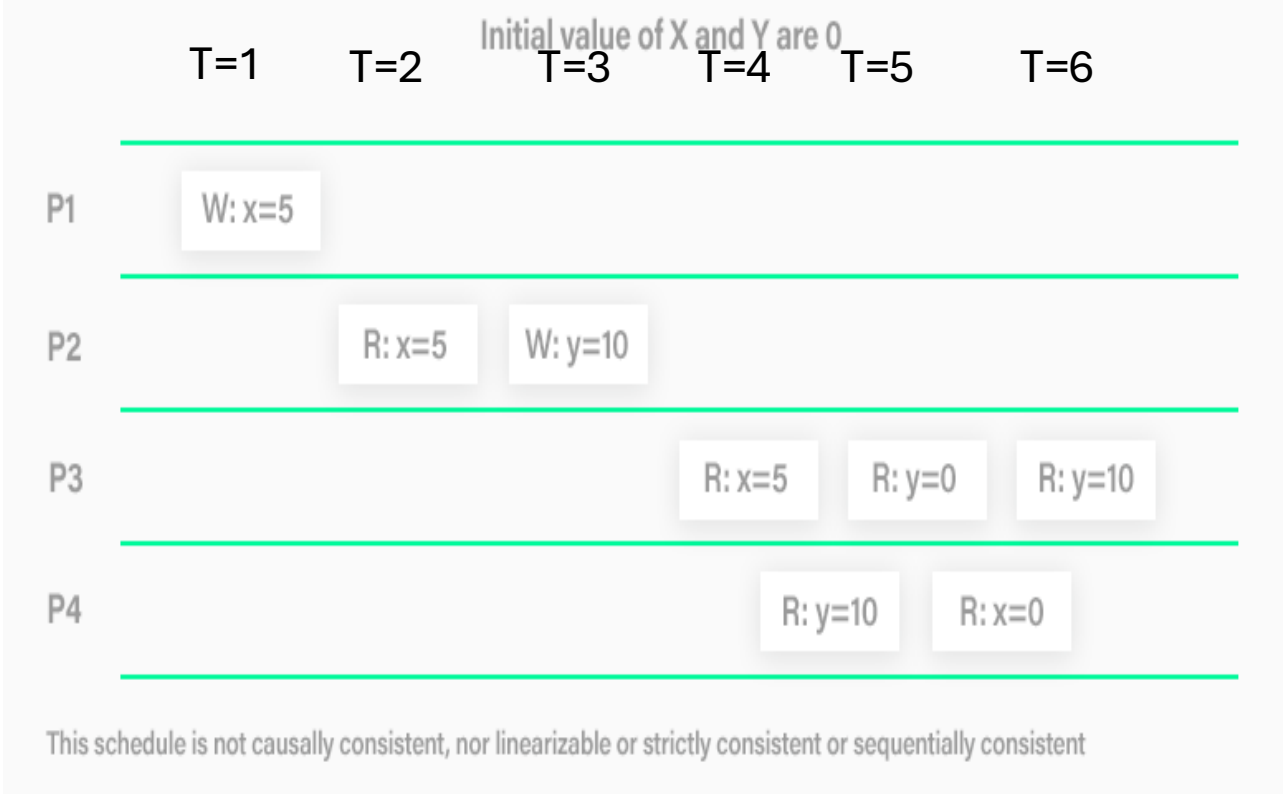
- **At T=5 :** y=10
- **At T=6:** x=0



Q: Is the Causal Consistency Preserved in this Scenario?

A:

Understanding Causal Consistency Scenario:



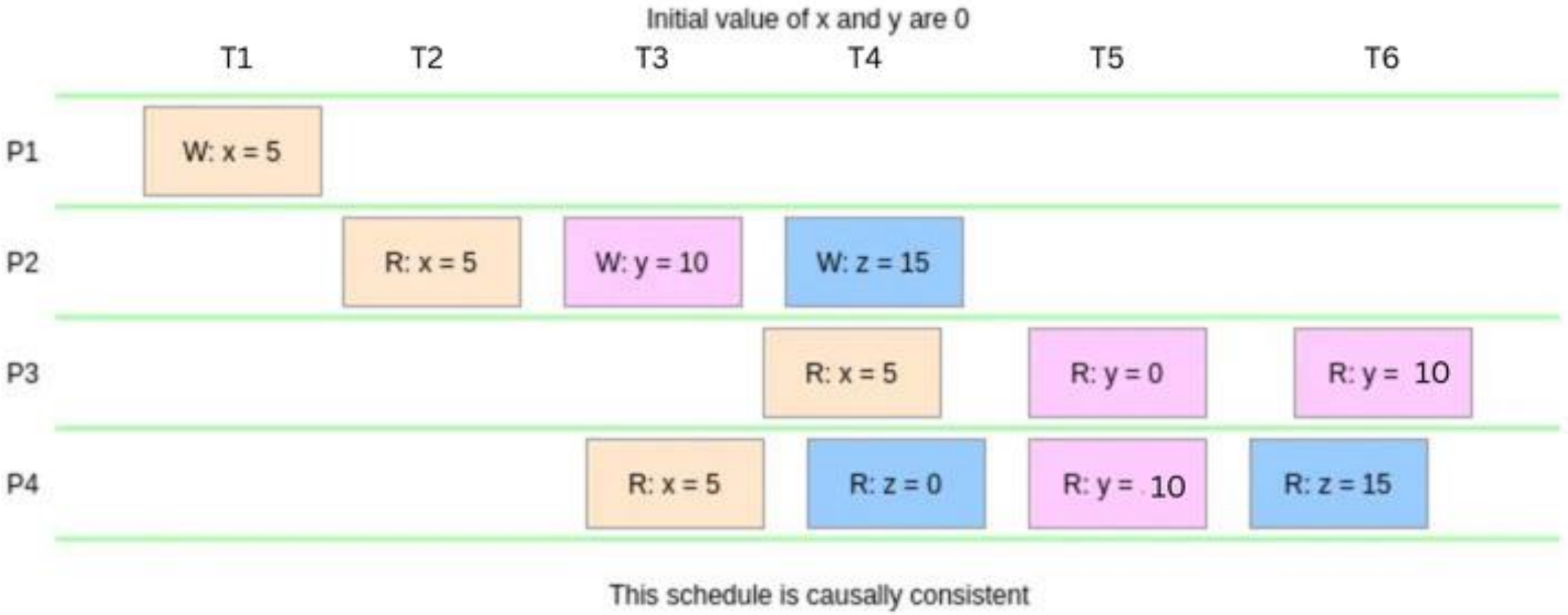
Now, lets assume that another UOE **P4** reads **x and y** in this sequence:

- **At T=5 :** y=10
- **At T=6:** x=0

Q: Is the Causal Consistency Preserved in this Scenario?

A: No. It gives an impression to P4 that y=10 is written to the system before x=5 which is actually incorrect. Hence it violates causal consistency guarantee.

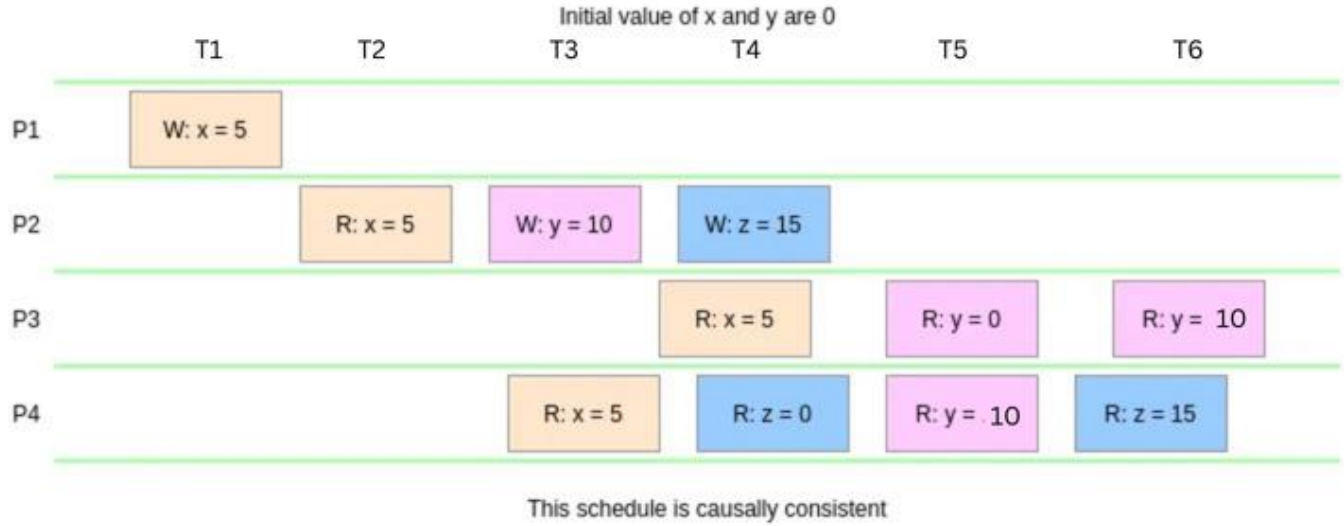
Example of Causal Consistency



Example of Causal Consistency

At first, **P1** writes **X = 5.**

Next, **P2** writes **Y=10** and **Z= 15** , based on the value of **X** being **5**.



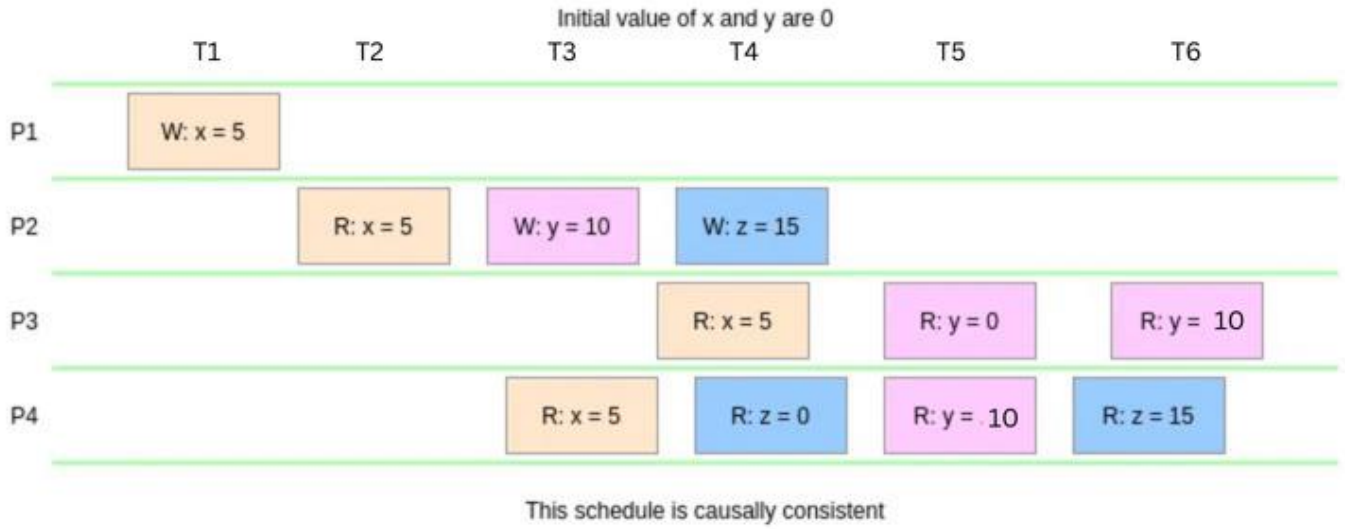
Example of Causal Consistency

At first, **P1** writes **X = 5.**

Next, **P2** writes **Y=10** and **Z= 15** , based on the value of **X** being **5**.

This establishes a causal relationship between the write of **X=5** and **Y=10** and **Z = 15** as follows:

(W:X=5) → (W: Y = 10)
(W: X=5) → (W: X = 15)



Example of Causal Consistency

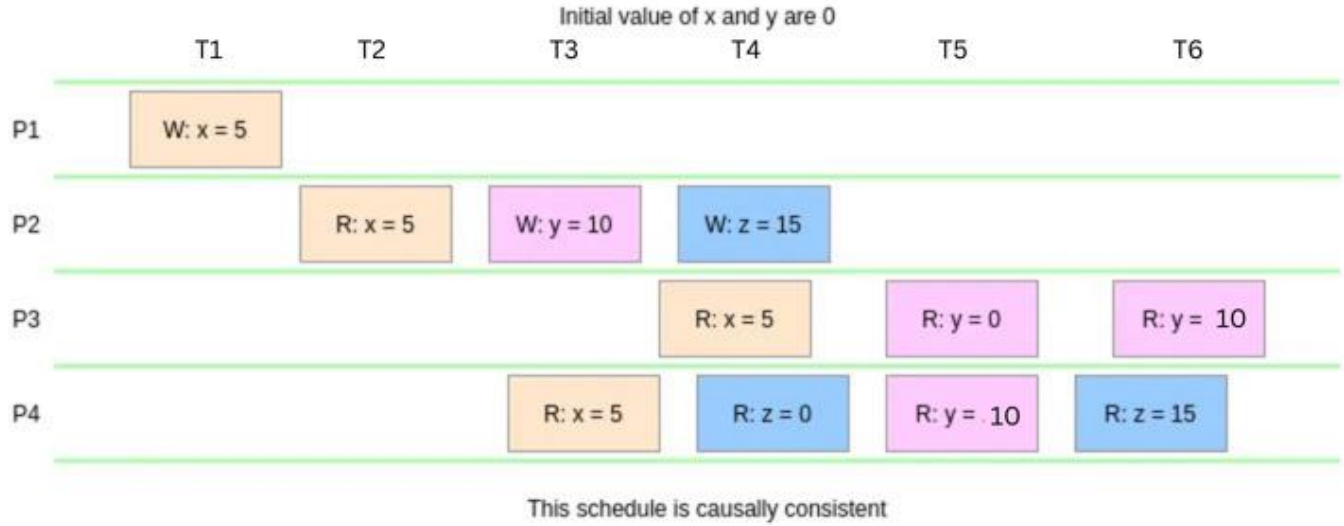
At first, **P1** writes **X = 5.**

Next, **P2** writes **Y=10** and **Z= 15** , based on the value of **X** being **5**.

This establishes a causal relationship between the write of **X=5** and **Y=10** and **Z = 15** as follows:

(W:X=5) → (W: Y = 10)
(W: X=5) → (W: X = 15)

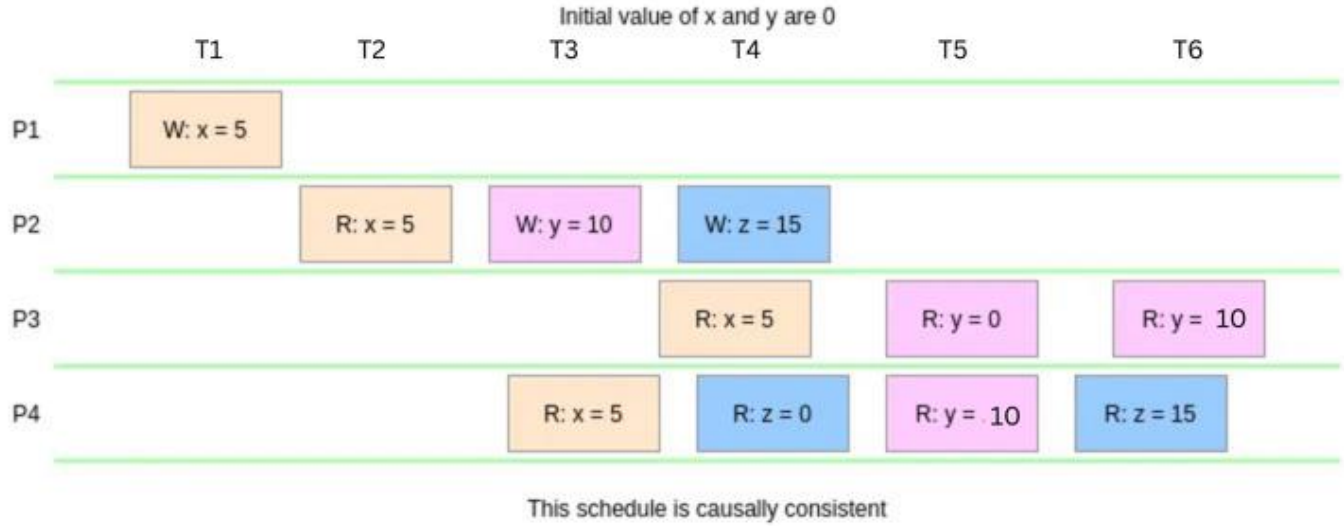
Notably, there is no causal relation between **(W: Y = 10)** and **(W:Z = 15)**.



Example of Causal Consistency

Now, suppose P3 , another UOE, performs the following read operations:

At T4: Reads X = 5
At T5: Reads Y=10
At T6: Reads Y = 15



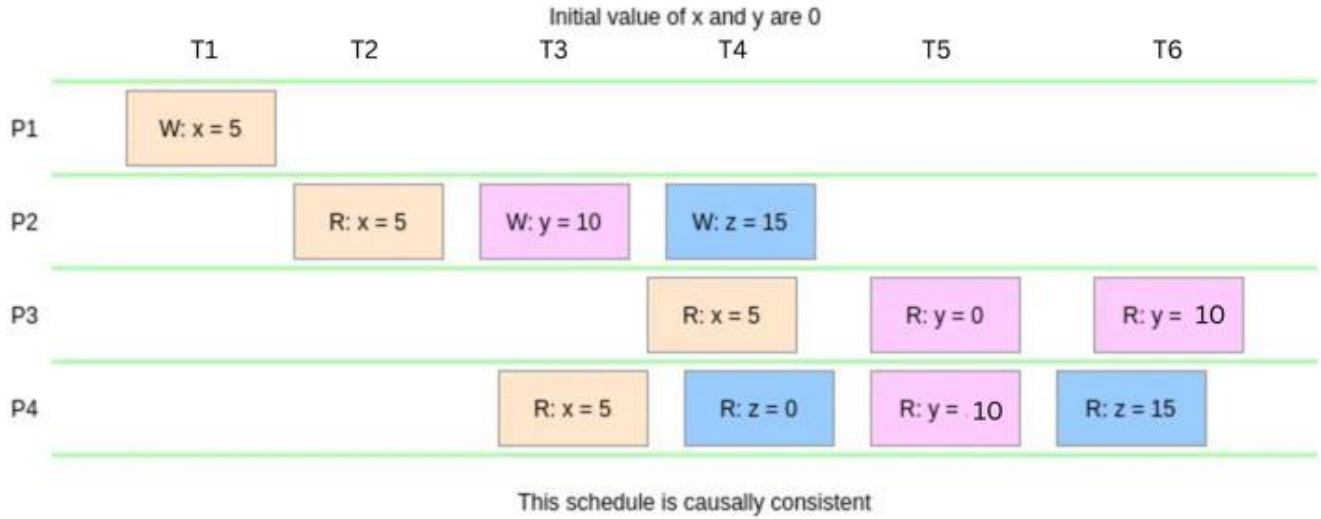
Q: Does this violate Causal Consistency because after X=5 was read, Y initially had the value of 0?

A:

Example of Causal Consistency

Now, suppose P3 , another UOE, performs the following read operations:

At T4: Reads X = 5
At T5: Reads Y=10
At T6: Reads Y = 15



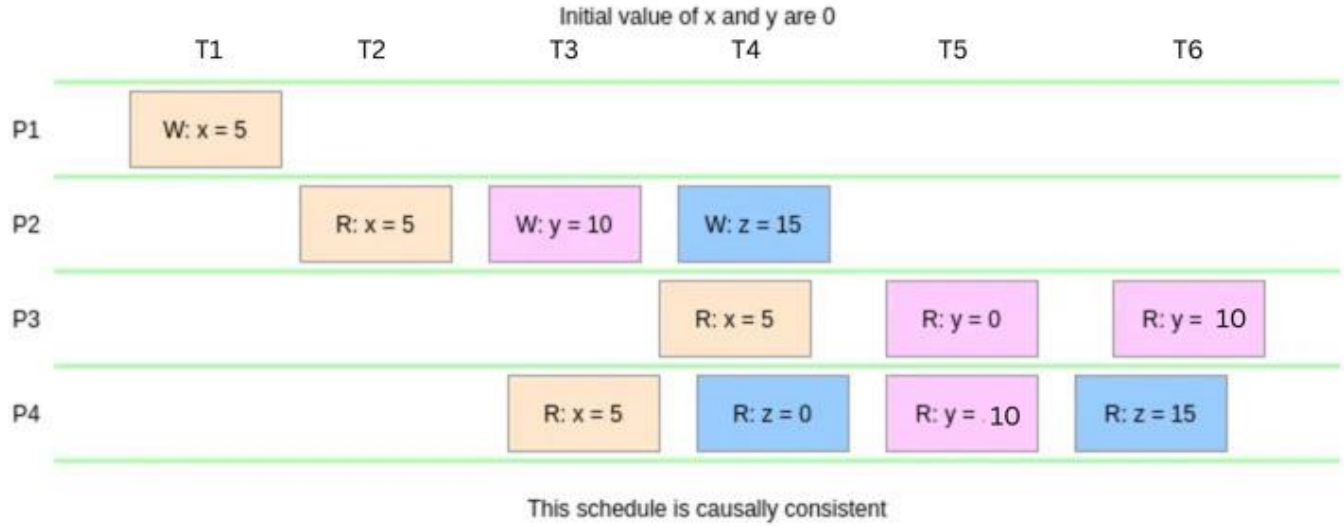
Q: Does this violate Causal Consistency because after X=5 was read, Y initially had the value of 0?

A: No, **because** the causal relation established is: (W: X=5) → (W: Y=10). This means that (W: X=5) must appear on UOE P4 before (W: Y=10). It doesn't say that all subsequent reads of Y must immediately reflect W: Y=10, as long as the causal order is preserved. Since P3 reads Y=10 only after reading X=5, the causal dependency is maintained, making this causally consistent.

Example of Causal Consistency

Now, suppose P4 , another UOE, performs the following read operations:

At T3: Reads X = 5
At T4: Reads Z = 0
At T5: Reads Y=15
At T6: Reads Z = 15

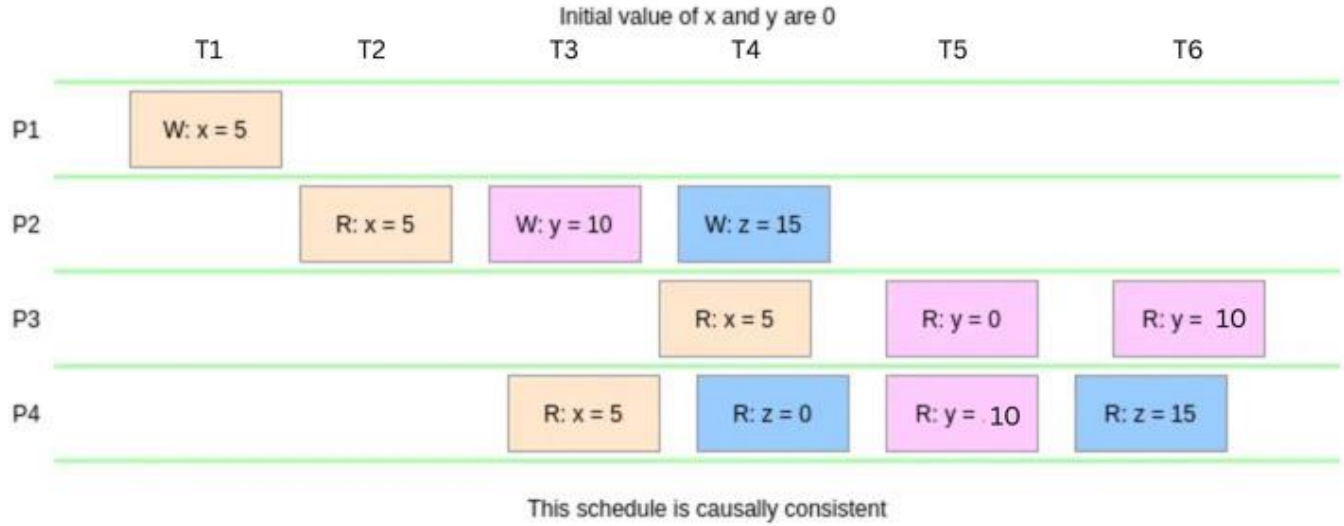


Q: Does this violate causal consistency because P4 reads **Z = 0** at T4 and **Y = 10** at T5, even though **W: Z = 15** happened before **W: Y = 10 in P1**?

Example of Causal Consistency

Now, suppose P4 , another UOE, performs the following read operations:

At T3: Reads X = 5
At T4: Reads Z = 0
At T5: Reads Y=15
At T6: Reads Z = 15



Q: Does this violate causal consistency because P4 reads **Z = 0** at T4 and **Y = 10** at T5, even though **W: Z = 15** happened before **W: Y = 10 in P1**?

A: No, because there is no causal relationship between (W:Y=10) and (W:Z=15). So, they can appear in any order in P4. And, as we saw earlier, after reading (W:X=5), it is not necessary that we read the latest values of Y immediately, because Causal Consistency doesn't enforce that.

Properties of Causal Consistency in Brief

✓ **Only Related Writes Are Ordered**

- Writes that are causally related are observed in the same order across UoEs.
- Unrelated writes can be observed in different orders.
- **No global ordering** is enforced.

✓ **No Real-Time Constraints**

- The system does **not** impose constraints based on real-time clocks.

Properties of Causal Consistency in Brief

✓ **Observation Order Matters More Than Values**

- What matters is the sequence in which updates are observed, not their absolute values.

✓ **Different Observers, Different Views**

- UoEs may observe different causally consistent sequences at the same time.

✓ **Causal Order Is Transitive**

- If **A → B** and **B → C**, then **A → C** holds.