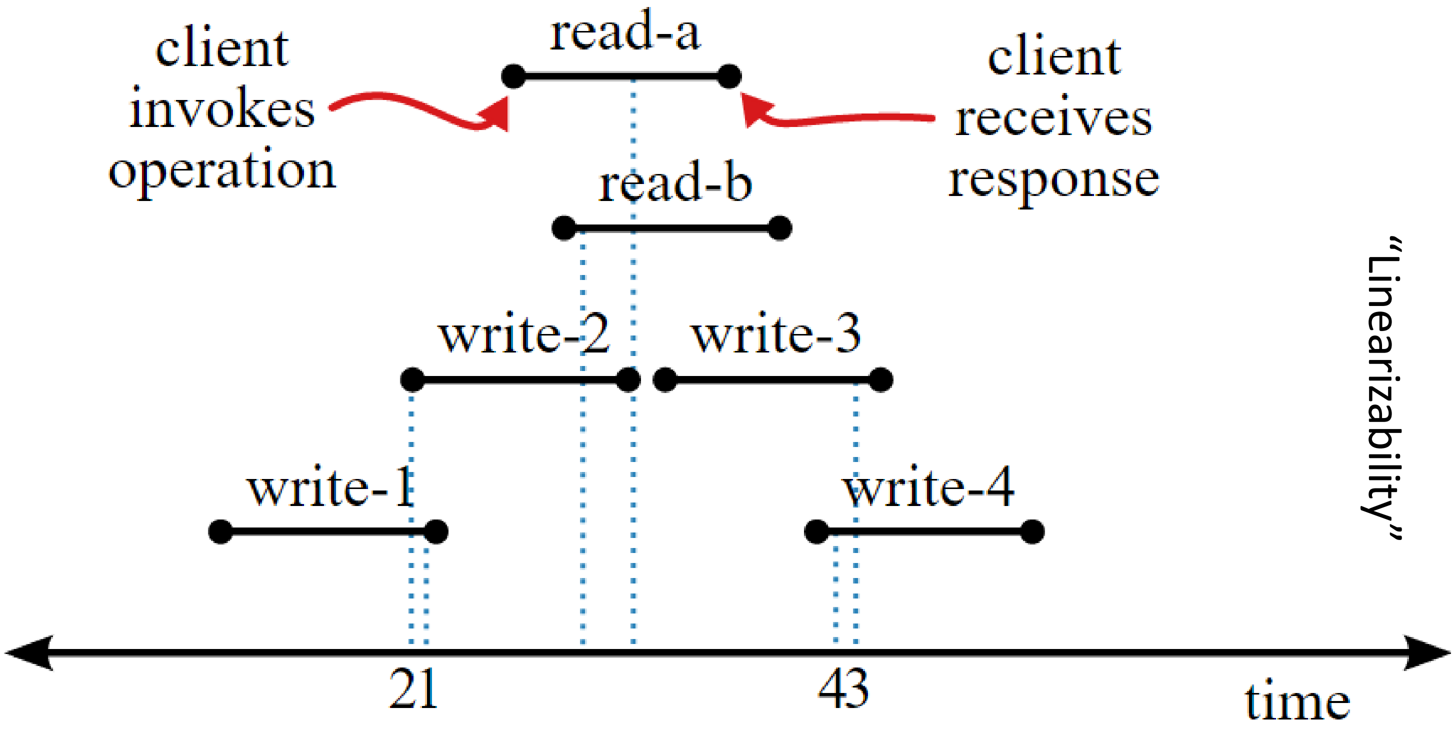
Replication & Consistency



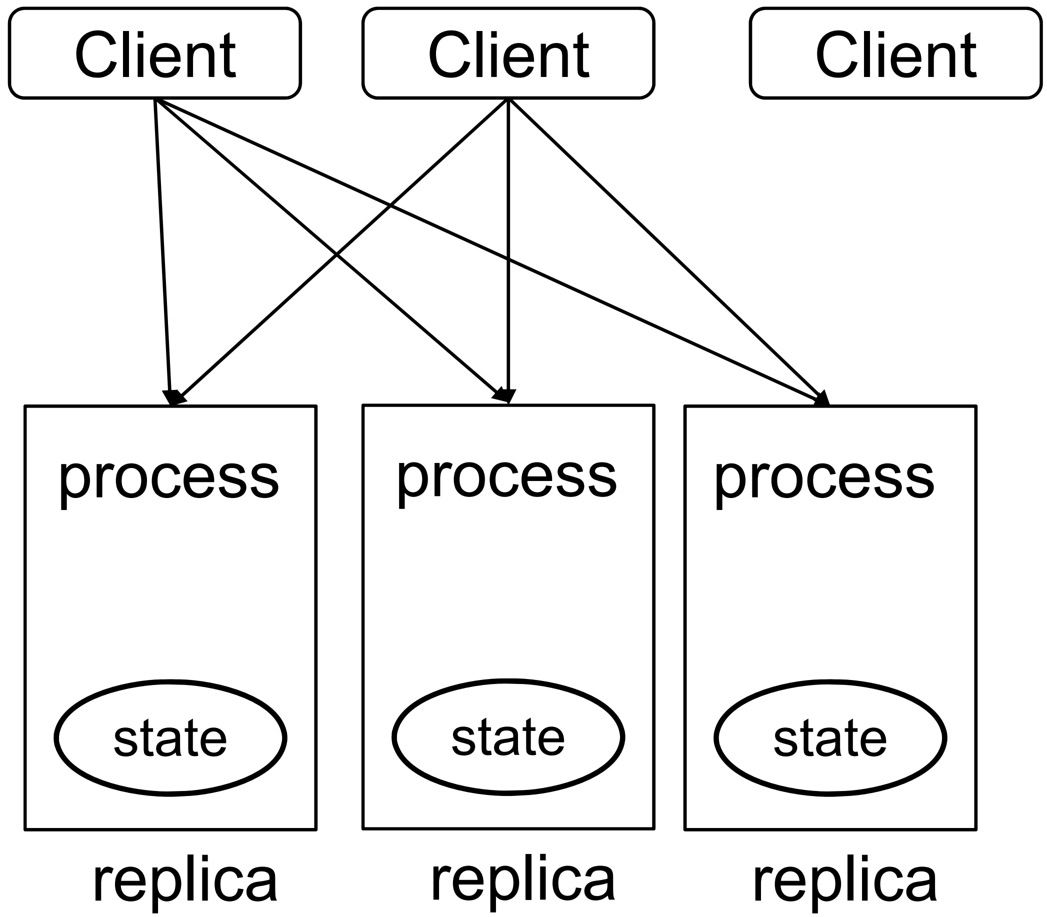
"Linearizability"

Agenda

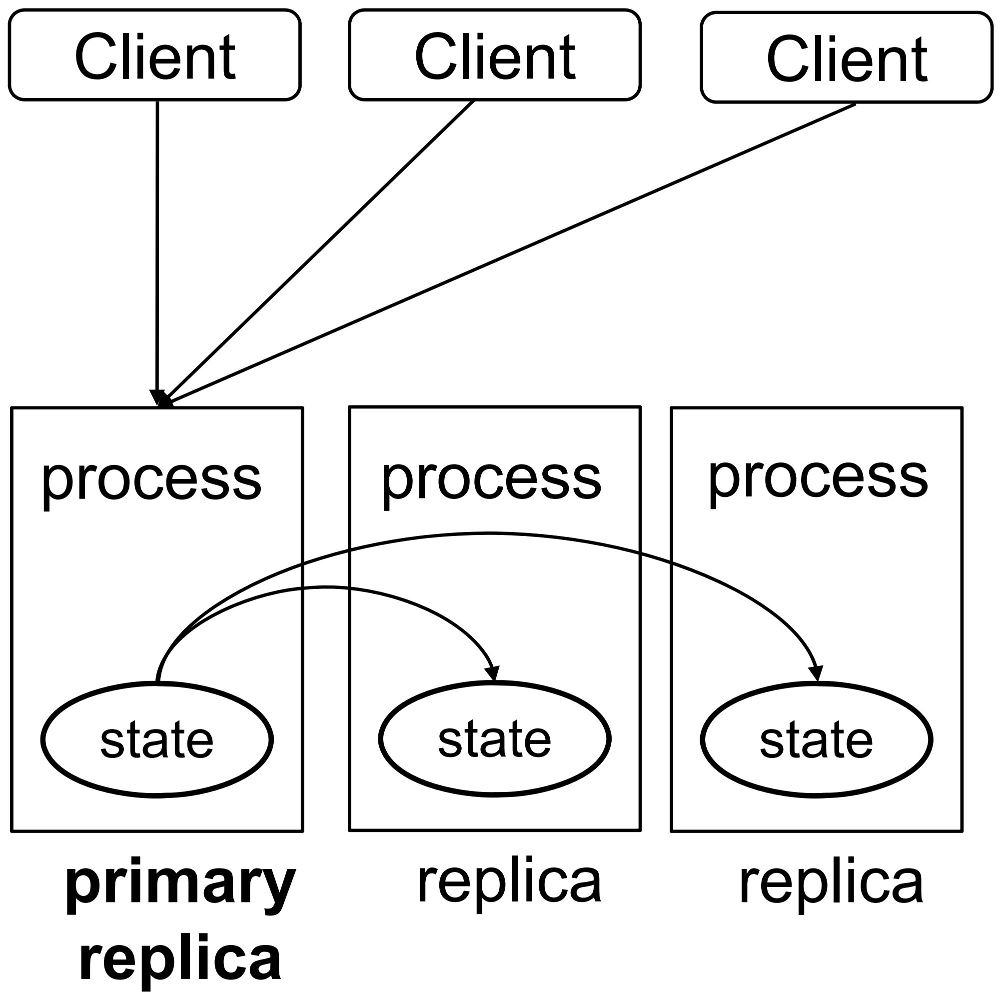
- Consistency models
- Data-centric consistency
- Client-centric consistency

CONSISTENCY MODELS

Active Replication



Passive Replication



Read-Write Conflict

- Alice and Bob buy tickets online, one ticket left
- Alice logs in, sees there is one ticket left, delays purchase
- Bob logs in, sees there is one ticker left, buys the ticket
- Alice resumes, tries to buy ticket now, purchase fails

Read-Write Conflict

- Process P1, P2 execute reads and writes concurrently

P1	P2
Read(A)	
	Read(A)
	Write(A)
Write(A)	

Write-Write Conflict

- Process P1, P2 execute writes concurrently

P1	P2
Write(A)	
	Write(B)
Write(B)	
	Write(A)

- Here, we have P1's version of B and P2's version of A
  – P1's version of A is lost and P2's version of B is lost
- NB: Outcome is not equivalent to any serial execution

Write-Write Conflict Example

(Initially A is 100; Deposit(x): A = A + x)

- Sequential execution to deposit 20:
  - Deposit(10)
  - Deposit(10)
- Implementation of Deposit:

  read A, add 10, store A

- Deposit1(10) || Deposit2(10)
- Deposit1:

  read A {100}, add 10 {110}
- Deposit2:

  read A {100}
- Deposit1:

  store A {110}
- Deposit2:

  add 10 {110}, store A {110}

Write-Write Conflict Example

(Initially A is 100; Deposit(x): A = A + x)

- …ion to
  - Deposit(10)
- Implementation of Deposit:
  …*tore A*

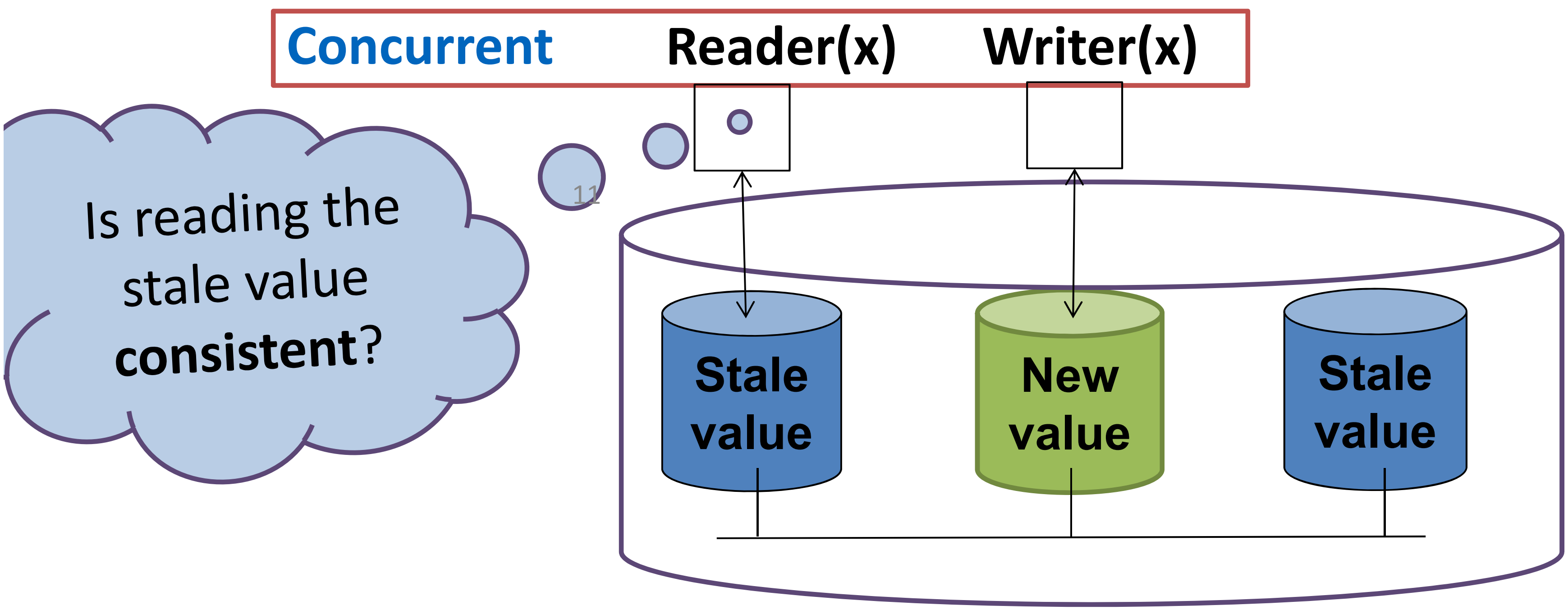
First call to deposit has no effect (overwritten by second store).

Update by first deposit is lost (**lost update problem**).

- Deposit1(10) || Deposit2(10)
- Deposit1:
  read A {100}, add 10 {110}
- Deposit2:
  read A {100}
- Deposit1:
  store A {110}
- Deposit2:
  add 10 {110}, store A {110}

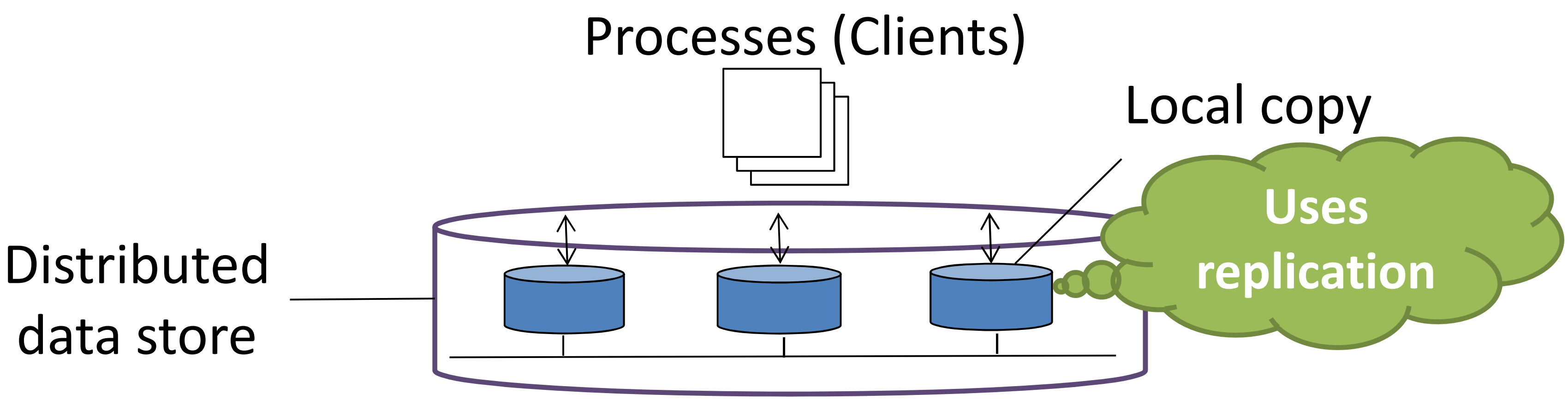
Concurrency and Replication

- All operations must be applied in a **specific order** to all replicas
- **Global ordering** is **too costly** and **not scalable** (e.g., using consensus, or single primary)
- **Solution**: Avoid global ordering using **weaker** consistency requirements **suitable** for the application



Consistency Models

- **Definition (Data-centric Consistency model)**
  - A **contract** between a distributed data store and a set of processes which specifies what the **results of concurrent read/write** operations are



Distributed data store as synonym for replicas, distributed database, shared memory, shared files, etc.

Data-Centric Consistency Models

- **Data-centric** consistency models dictate the outcome of concurrent reads and writes (r/w and w/w **conflicts**):
  - Strict consistency
  - Sequential consistency
  - Linearizable consistency
  - Causal consistency
  - FIFO consistency
  - Weak consistency

Strict Consistency

- **Definition**: Any ***read*** on a data item *x* returns a value corresponding to the result of the **most recent *write*** on *x*

- Uni-processor systems have traditionally observed strict consistency, …

  but what about multi-processor systems?

  – a = 1; a = 2; print(a); ***Output?***

- Definition assumes existence of **absolute global time** for unambiguous determination of "most recent“.

Interpretation of Strict Consistency

- Under strict consistent, all writes are **instantaneously visible to all processes** and **absolute global time order is maintained**

- If a **replica is updated**, …
  - all subsequent reads, **see the new value**, no matter how soon after the update the reads are done
  - and no matter which process is doing the reading and where it is located

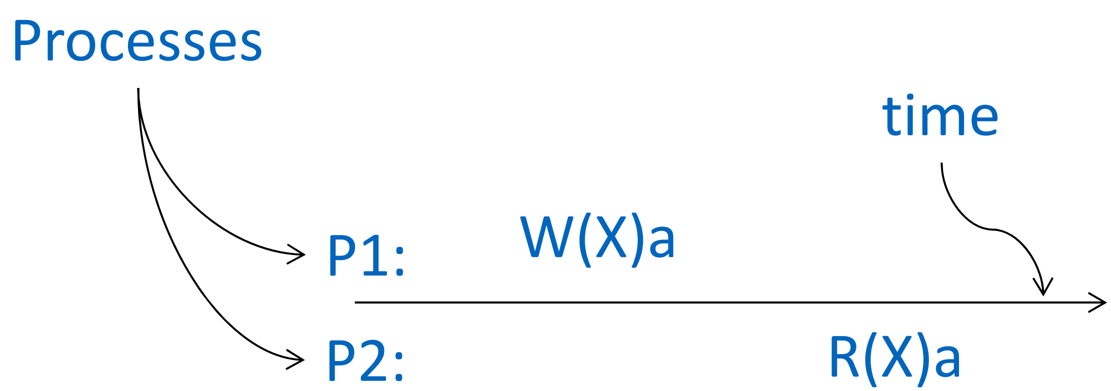
- Similarly, if a **read** is done, then it **gets the most recent value,** no matter how quickly the next write is done

Notation

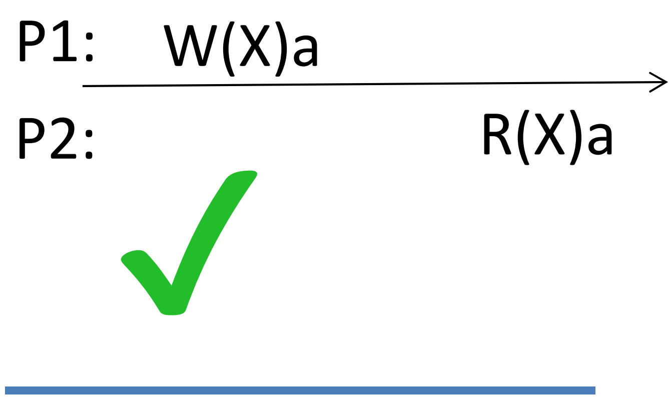
W(X)a: Represents **writing** the value ***a*** to data ***X*** *(memory)*

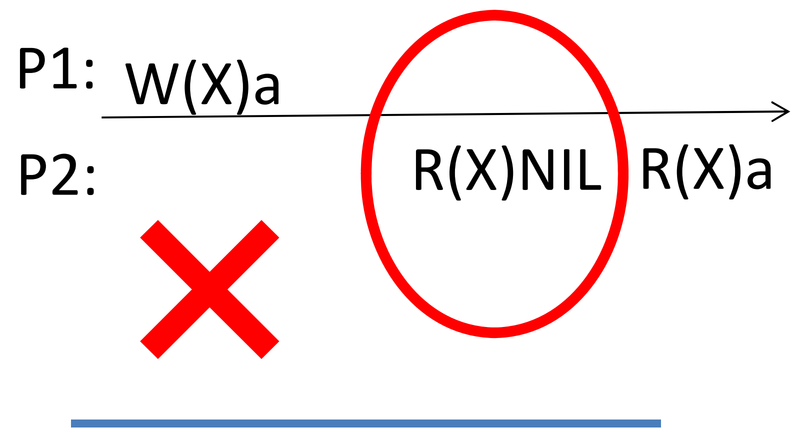
R(X)a: Represents **reading** data ***X***, which returns the value ***a***

Initial value of ***X*** is ***NIL***



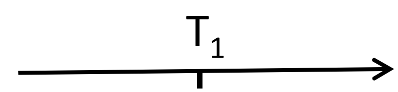
Strict Consistency Example



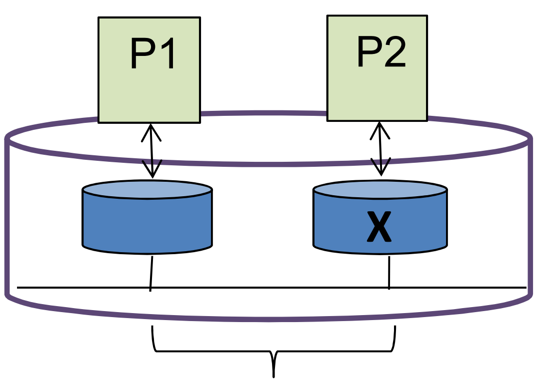


Strict Consistency: Thought Experiment

- To satisfy strict consistency, laws of physics may have to be violated! Obviously, not an option!!!
- Example:



@P1: W(X)a at T_1 **@P2: R(X)a** at T_1 + 1ns



3 meters away

To realize strict consistency in this case, *W(X)a* would have to travel at 10 times the speed of light!

Strict Consistency: The Bad News

- It is **impossible to perfectly** synchronize clocks
  - How to accurately determine the time of each operation?
- It is **impossible** to instantaneously replicate operations

Fallacy #2: Latency is zero

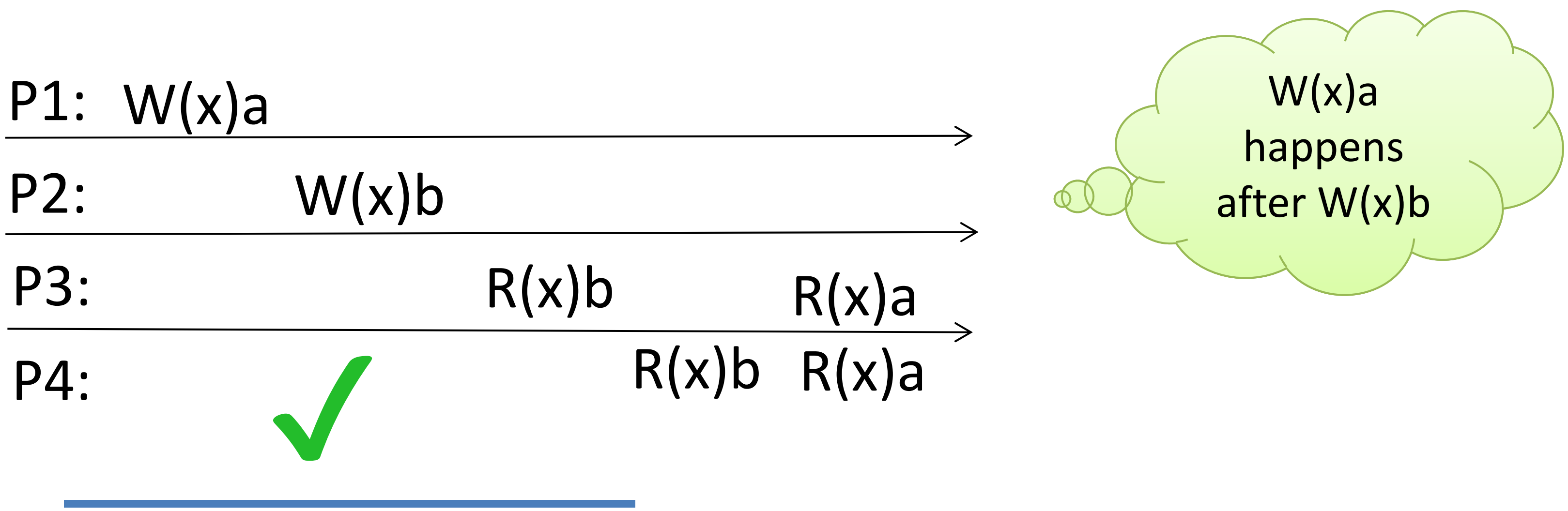
Definition of Sequential Consistency

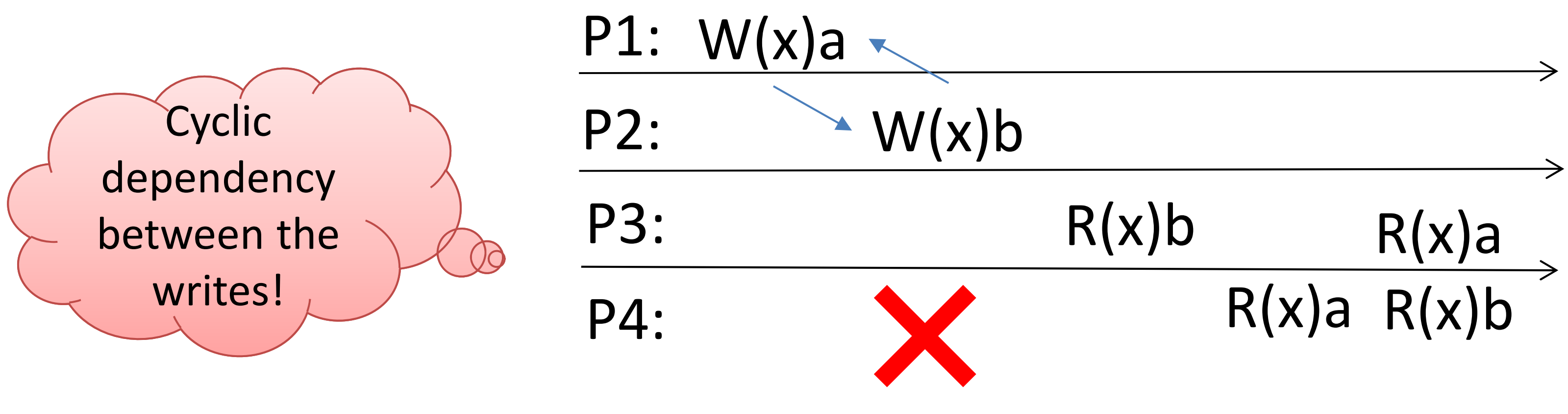
- The result of any execution is the same as if the operations by **all processes** on the data store were executed in some **sequential order** and …

- … the operations of each individual process appear in this **sequence in the order specified by its program**

- Weaker than strict consistency: **logical** time instead of **physical** time

Sequential Consistency Example





Definition of Linearizability

- The result of any execution is the same as if the (read and write) ope … ata store were executed i … e operations of each indiv … uence in the order specified by its program.

Sequential consistency

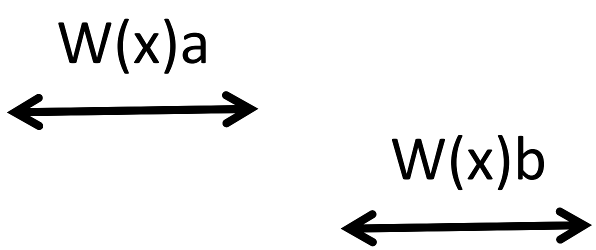
- In addition, if $\mathbf{ts_{OP1}(x) < ts_{OP2}(y)}$, then **operation OP1(x)** should **precede OP2(y)** in this sequence
- **$\mathbf{ts_{OP}(x)}$ denotes the timestamp** assigned to operation OP *that is performed on data item x,* and OP is either read (R) or write (W).

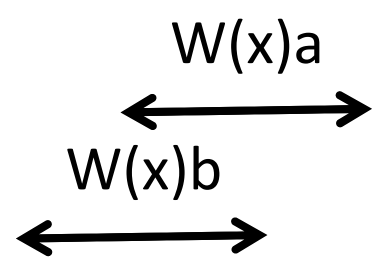
W(x)a

Time stamp is modeled as a duration.

Linearizability

- Like strict consistency, **assumes global time,** but not absolute global time
- Assumes processes in the system have **physical clocks** synchronized to within an **bounded error** (captured by duration timestamp)
- If ***W(x)b*** was the **most recent** write operation and there is no other write operation **overlapping** with *W(x)b*, then any **later** read should return ***b***
- If *W(x)a* and *W(x)b* were two **most-recent overlapping** write operation, then **any later read** should **return either *a* or *b*,** not something else

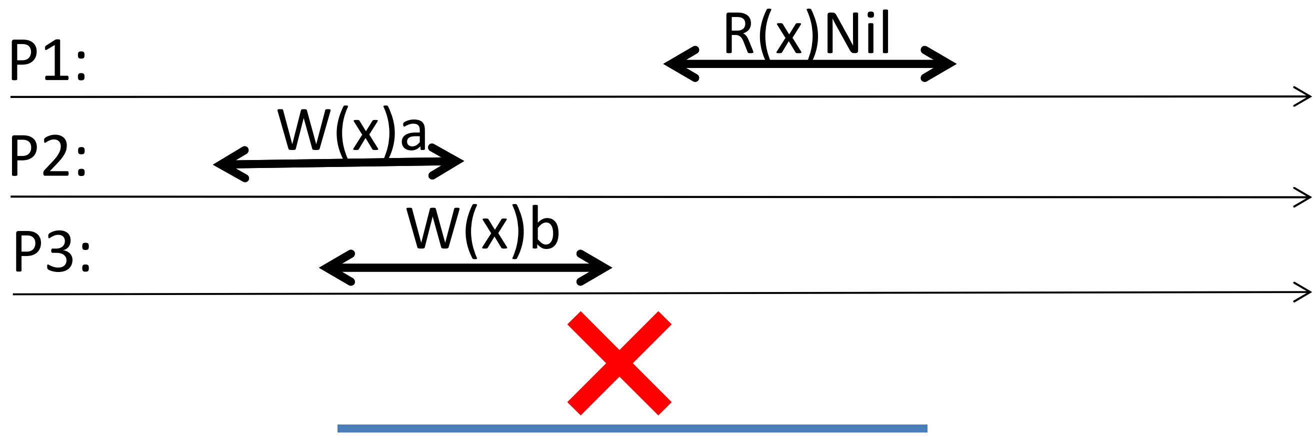


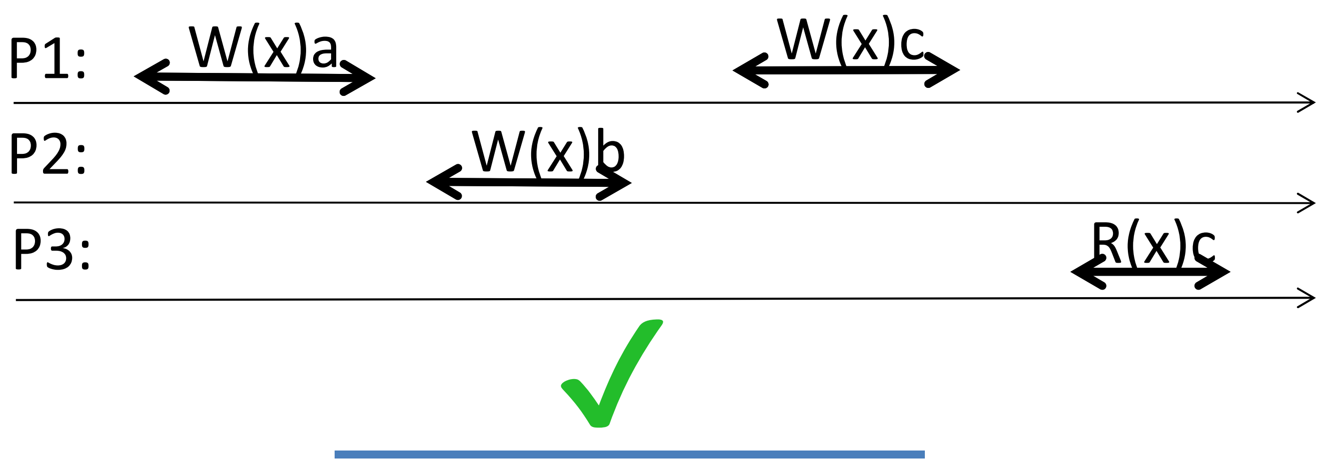


Linearizability

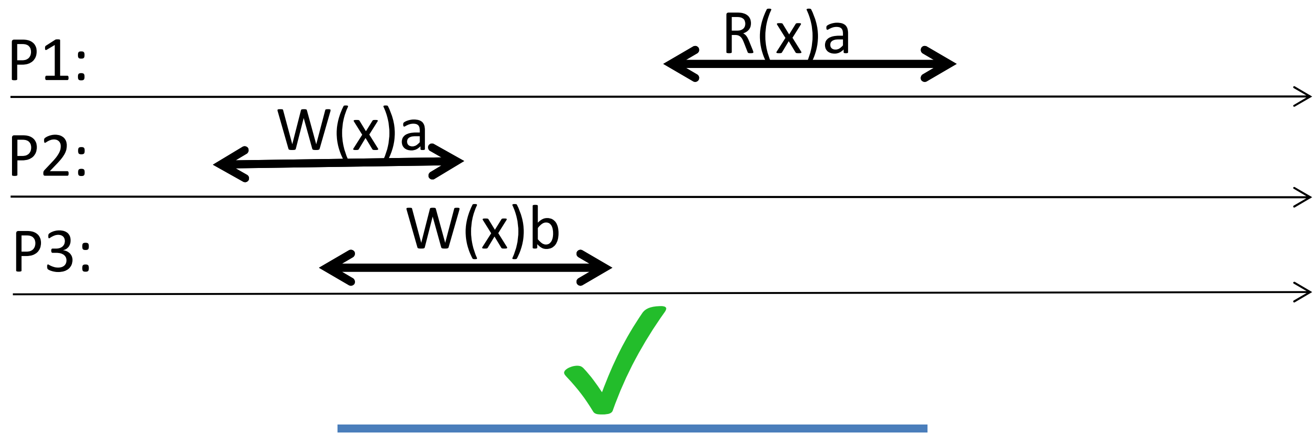
- A linearizable data store is also **sequentially consistent**
- I.e., linearizability is more restrictive (**stronger**)
- Difference is the ordering according to a set of synchronized clocks
- Linearizability prevents **stale reads**, since it guarantees correct reads after a write is completed

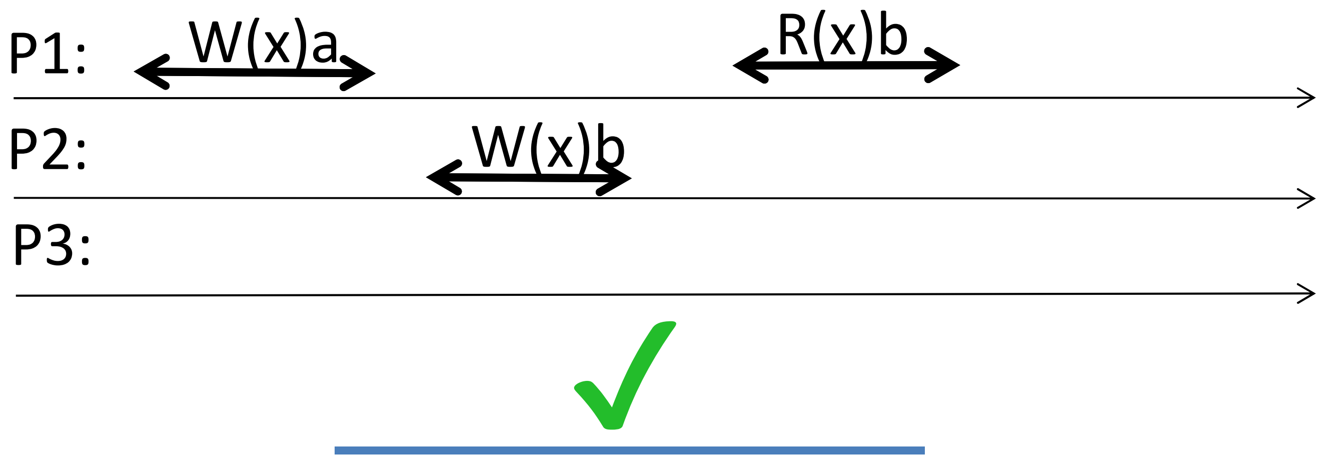
Example 1



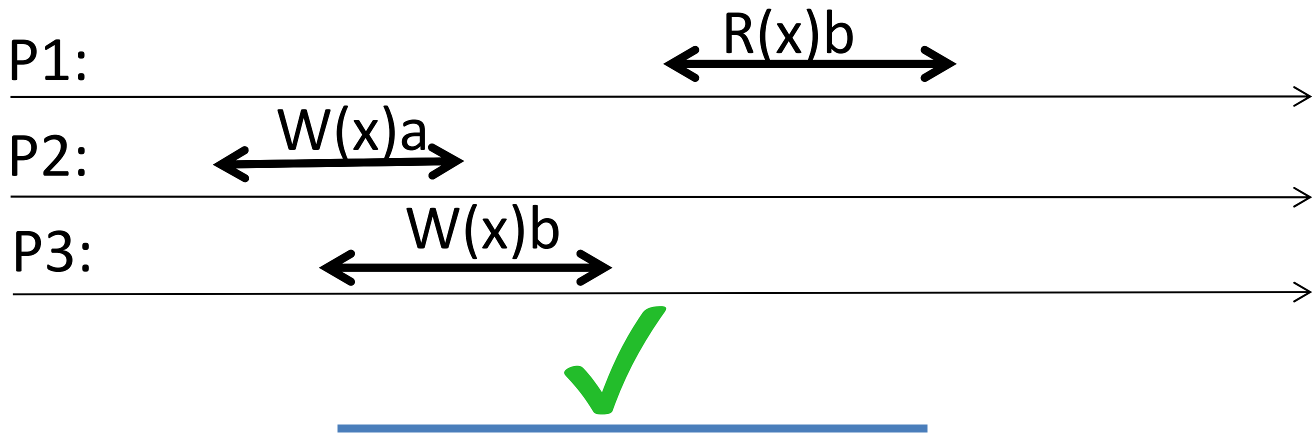


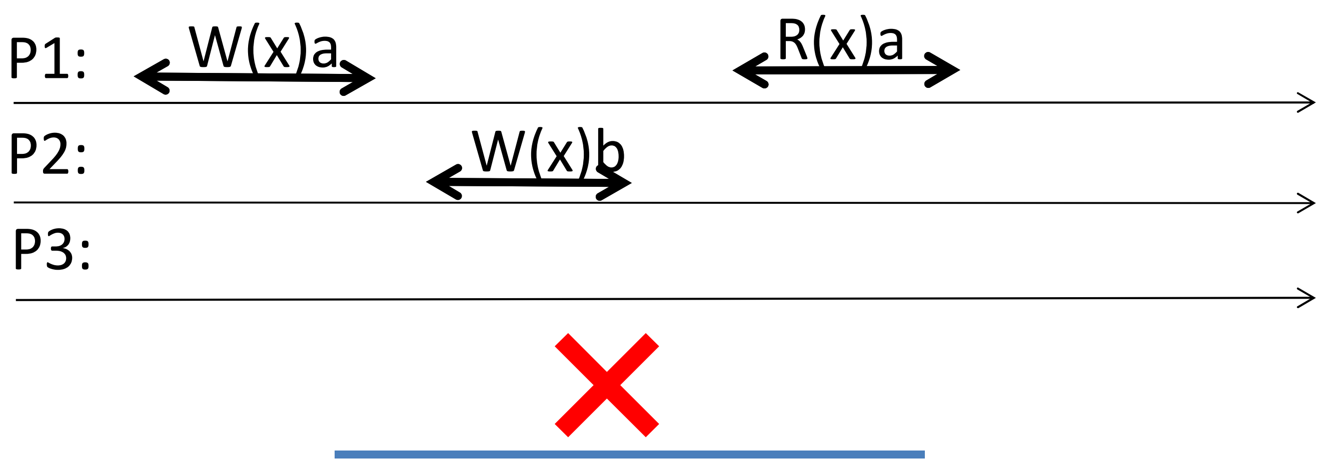
Example 2



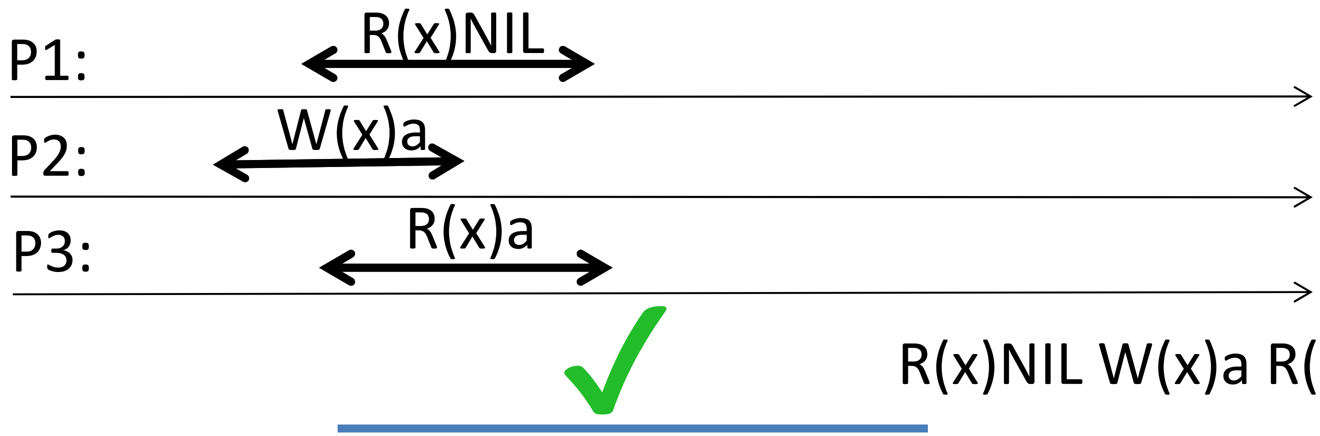


Example 3

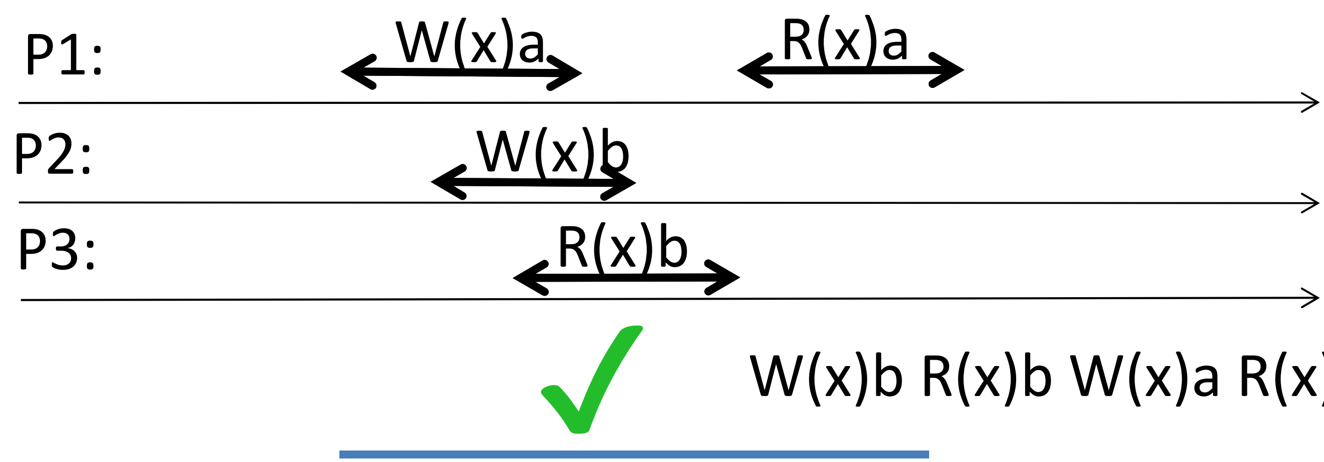




Example 4

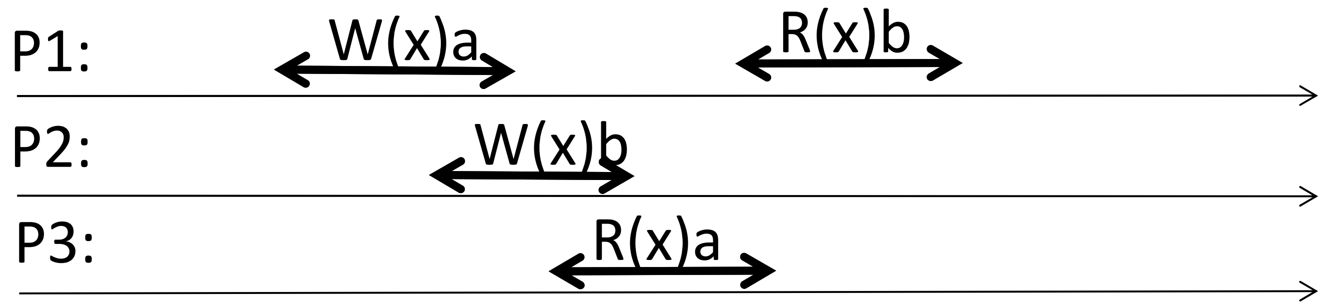


R(x)NIL W(x)a R(x)a

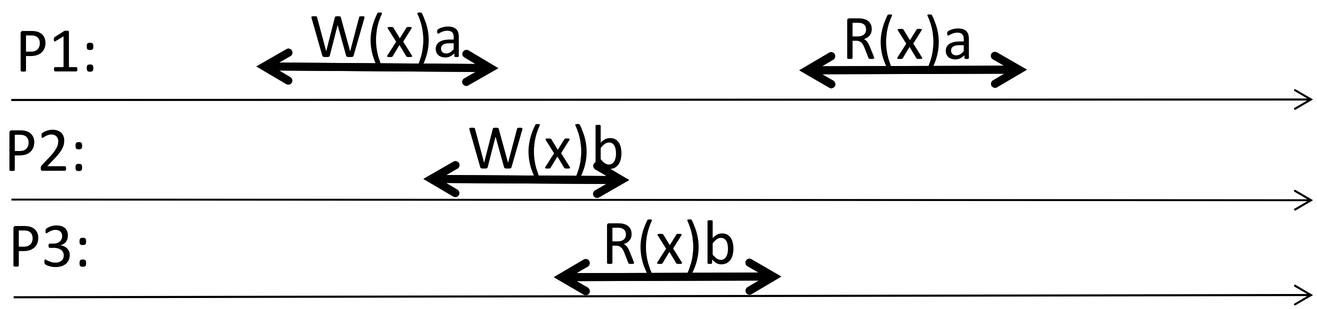


W(x)b R(x)b W(x)a R(x)a

Example 5



W(x)a R(x)a W(x)b R(x)b



W(x)a W(x)b R(x)b ~~R(x)a~~
W(x)b W(x)a R(x)a ~~R(x)b~~
W(x)b W(x)a ~~R(x)b~~

Intuition for Causal Consistency

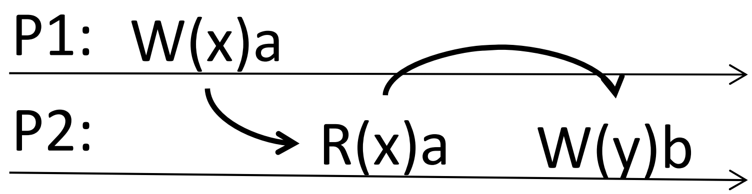
- **Weaker** than sequential consistency
- Distinguish events that are potentially **causally related and those that are not** (concurrent events)
- Similar to the **happens-before** relationship (cf. Lamport clock), but with reads and writes instead of messages
- If Event *B* is influenced (caused) by an earlier Event A (A→B), causal consistency requires that **every process first sees *A* then *B***
- **Concurrent events** (i.e., writes) may be seen in a different order on different machines

Causal Relationship

- **Read followed by write** in the same process, the two are causally related
  - **Example**: R(x)a → W(y)b (e.g., it may be that y=f(x))
- Read is **causally related** to write that provided the value read got (across processes)
  - **Example**: W(x)a → R(x)a
- Transitivity: if Op1 → Op2, Op2 → Op3, then Op1 → Op3
- Independent writes by two processes on a variable are not causally related (they are concurrent)
  - **Example:** W(x)a || W(x)b

"Potentially" Causally Related

- **Example:** Say, $W_1(x)$, then $R_2(x)$ and $W_2(y)$

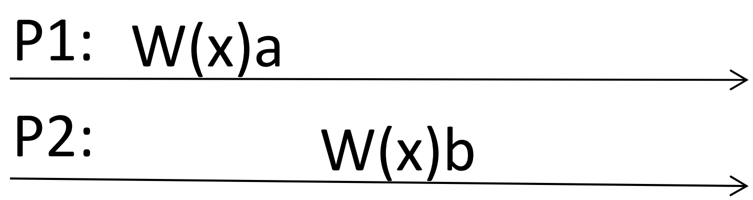


- Reading of *x* and writing of *y* are **potentially** causally related
- Computation of *y* may have depended on value of *x* read by P2 (written by P1); e.g., $y = f(x)$
- On the other hand, *y* may not have depended on *x*, yet potential causality still holds in our formalization!

Definition of Causal Consistency

- "Potentially" causally related writes must be seen by all processes in the same order

- Concurrent writes may be seen in a different order by different processes



Causal Consistency Example I

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

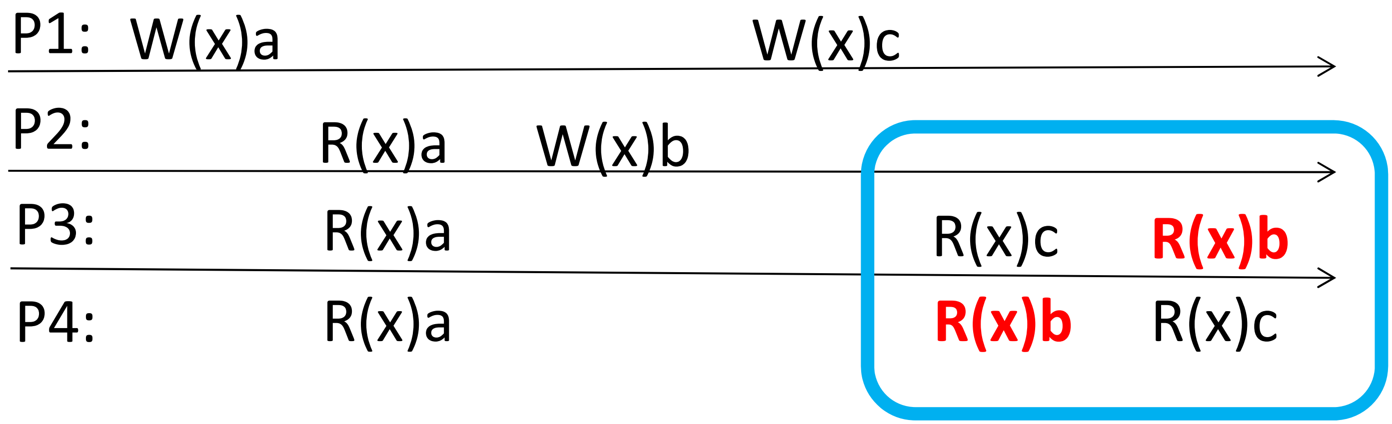
Is this strictly, sequentially or causally consistent?

Causal Consistency Example I

P1:	W(x)a			W(x)c	
P2:		R(x)a	W(x)b		
P3:		R(x)a		R(x)c	**R(x)b**
P4:		R(x)a		**R(x)b**	R(x)c

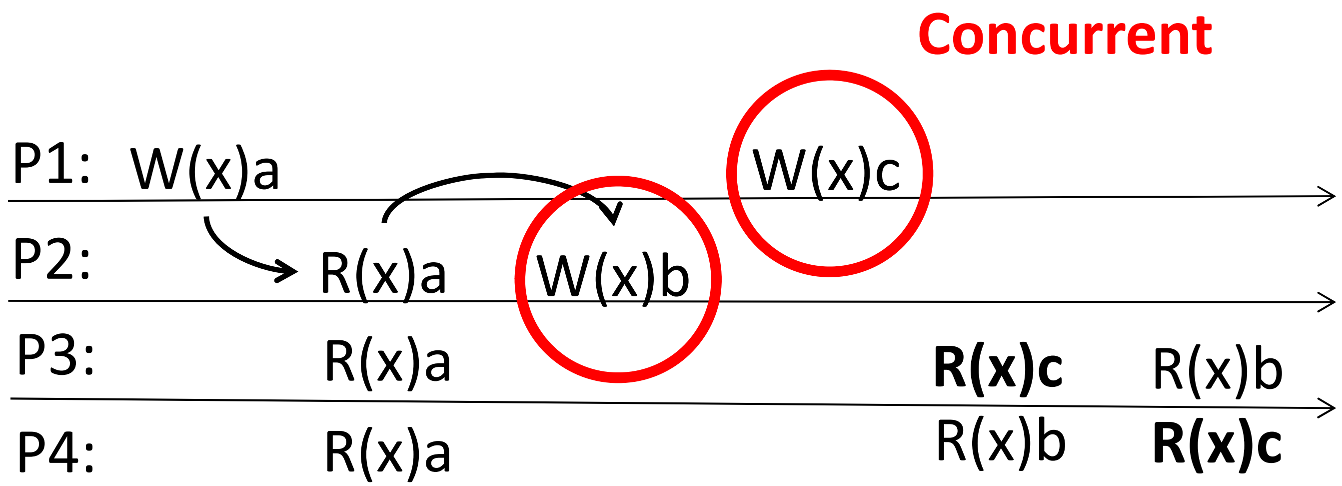
Neither **strictly**, nor sequentially consistent, but causally consistent.

Causal Consistency Example I



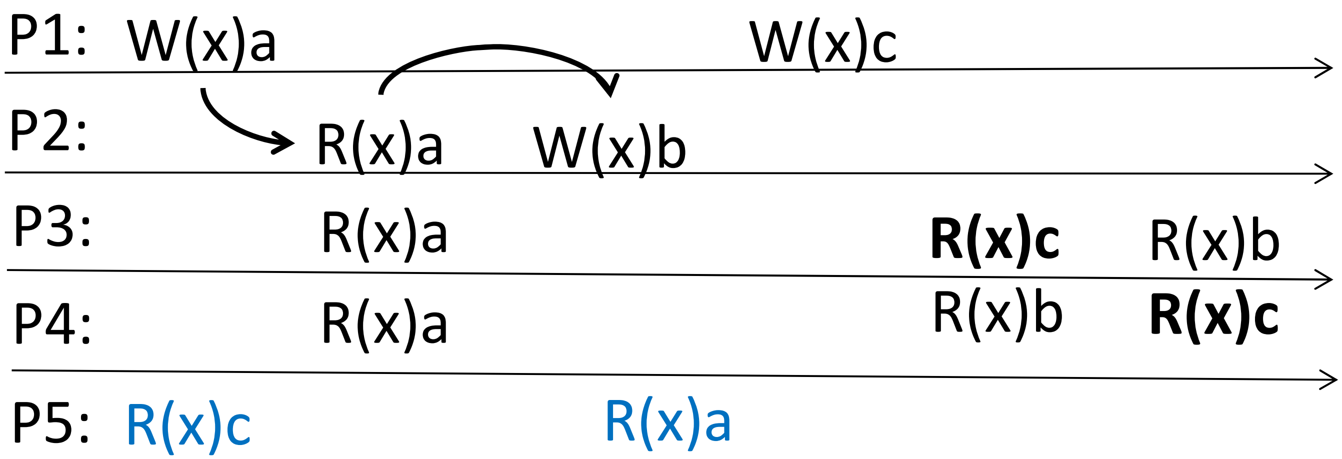
Neither **strictly**, nor sequentially consistent,
but causally consistent.

Causal Consistency Example I



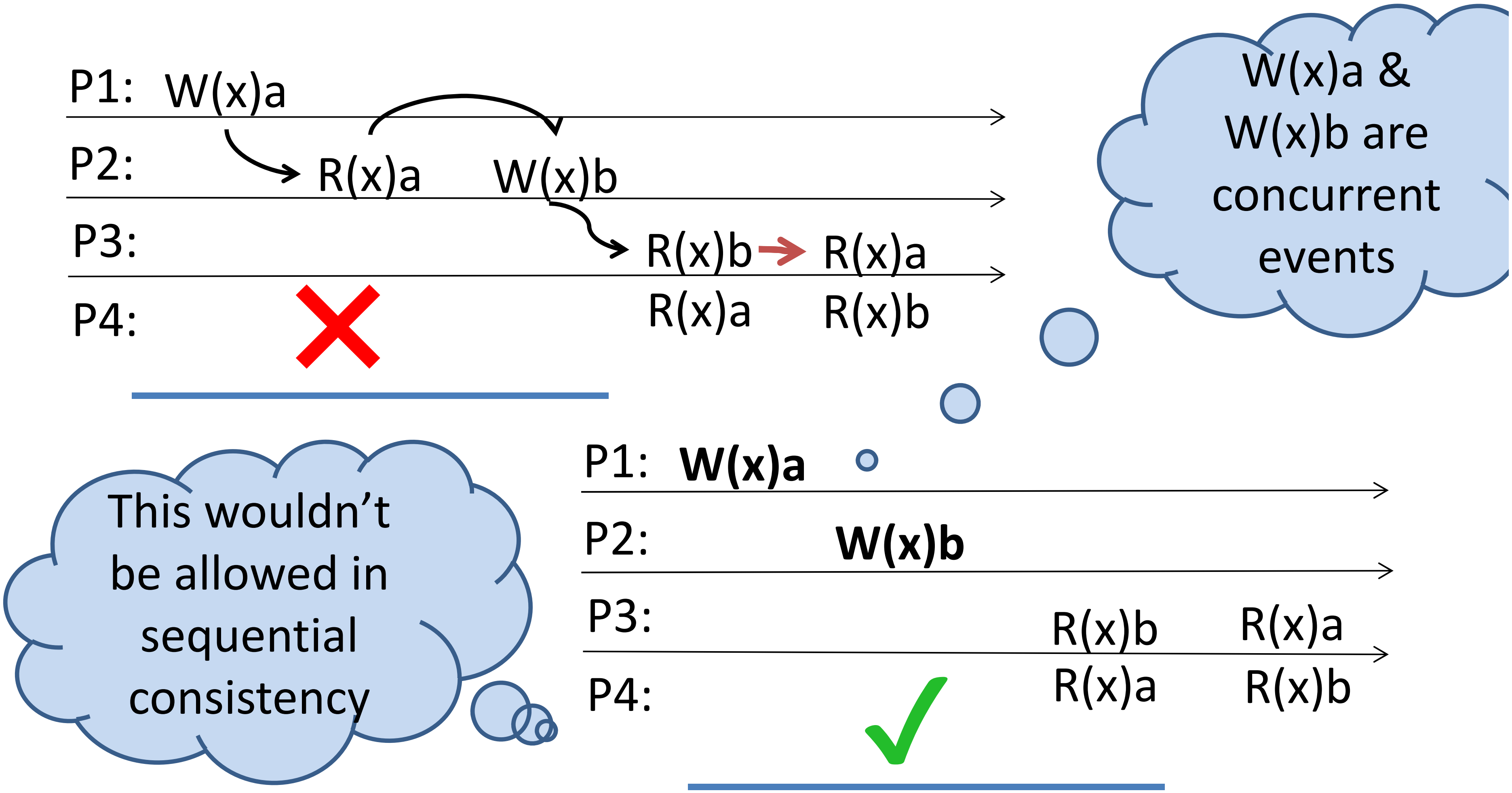
Neither strictly, nor sequentially consistent,
but causally consistent!

Causal Consistency Example I



Not causally consistent, W(x)a happens before W(x)c in P1

Causal Consistency Example II



Definition of FIFO Consistency

- Writes by a single process are seen by all other processes in the order in which they were issued
- Writes from different processes may be seen in a different order by different processes
- Easy to maintain: simply send writes in FIFO order from each process to each replica (e.g. using TCP)

FIFO Consistency

P1: W(x)a

P2: R(x)a **W(x)b** **W(x)c**

P3: **R(x)b** R(x)a **R(x)c**

P1: W(x)a

P2: R(x)a **W(x)b** **W(x)c**

P3: **R(x)c** R(x)a **R(x)b**

Weak Consistency

- Not all processes need to see all writes, let alone in the same order
- Based on **synchronization variable *S*** with single operation ***Synchronize(S)***
- A distributed critical section to access shared resources
- Any process can perform read/write operations and synchronize
- Order of operations before **synchronization** is not consistent
- After a synchronization, all processes see the **same outcome of operations** preceding the synchronization point

Weak Consistency Properties

- Access to synchronization variables are sequentially consistent
- No operation on a synchronization variable is allowed to be performed until **all previous writes have been completed at all replicas**
- No read or write operation are allowed to be performed until all previous operations to synchronization variables have been performed

Weak Consistency Interpretation

- Process forces the just written value out to all replicas
- Process can be sure to get the most recent value written before it reads
- Model enforces **consistency on a group of operations** as opposed to individual reads and writes
- Care about the **effect of a group** of reads and writes

Weak Consisten

P2 & P3 have yet to synchronize, no guarantees about values read

P1: W(x)a W(x)b S

P2: R(x)a R(x)b S

P3: R(x)b R(x)a S

P1 propagates value b

P1: W(x)a W(x)b S

P2: S R(x)a

Weak Consistency

Agreement on value is implementation dependent.

P1 & P2 agree on value a

P1: W(x)a W(x)b S R(x)a

P2: W(x)a S R(x)a

P3: R(x)b R(x)a S

✓

P1 & P2 agree on value b

P1: W(x)a W(x)b S R(x)b

P2: W(x)a S R(x)b

P3: R(x)b R(x)a S

✓

Weak Consistency

P1: W(x)a W(x)b S R(x)a
P2: W(x)a S R(x)b
P3: R(x)b R(x)a S

P1: W(x)a W(x)b S R(x)b
P2: W(x)a S R(x)a
P3: R(x)b R(x)a S

Summary of Consistency Models

Strongest

Consistency	Description
Strict	**Absolute time** ordering of all shared accesses
Linear-izability	All processes see all **shared accesses in the same order**. Accesses **ordered** according to a (non-unique) global timestamp
Sequential	All processes see all **shared accesses in the same order**. Accesses are **not ordered in time**
Causal	All processes see **causally-related shared accesses in the same order**
FIFO	All processes **see writes from each other in the order they were issued**. Writes from different processes may not always be seen in that order
Weak	Shared data can be counted on to be consistent only after a **synchronization** is done

Weakest

When to use…?

- **Linearizability**: Strongest distributed solution, possible with eager replication (synchronous), Hbase, Bigtable
- **Sequential**: System-wide consistent reads, e.g., everyone sees replies to a post in same order
- **Causal**: Everyone sees posts before replies
- **FIFO**: Reading all messages from each friend in order but not across friends
- **Weak**: Responsibility left to the developer who must explicitly enforce synchronization

CLIENT-CENTRIC CONSISTENCY

Client-Centric Consistency

- So far, the goal was to maintain consistency in presence of **concurrent read** and **write** operations
- There are use cases with **no (few) concurrent writes,** or consistency of write operations are secondary
  - **DNS**: **No write-write conflicts** since there is a single authority updating each domain (disjoint partitions)
  - **Key-value stores**: **Usually no write-write conflicts** since updates partitioned by keys, e.g., Dynamo, Cassandra (also, optimistic concurrency control)
  - **WWW**: Heavy use of client-side caching, reading stale pages is acceptable in many cases

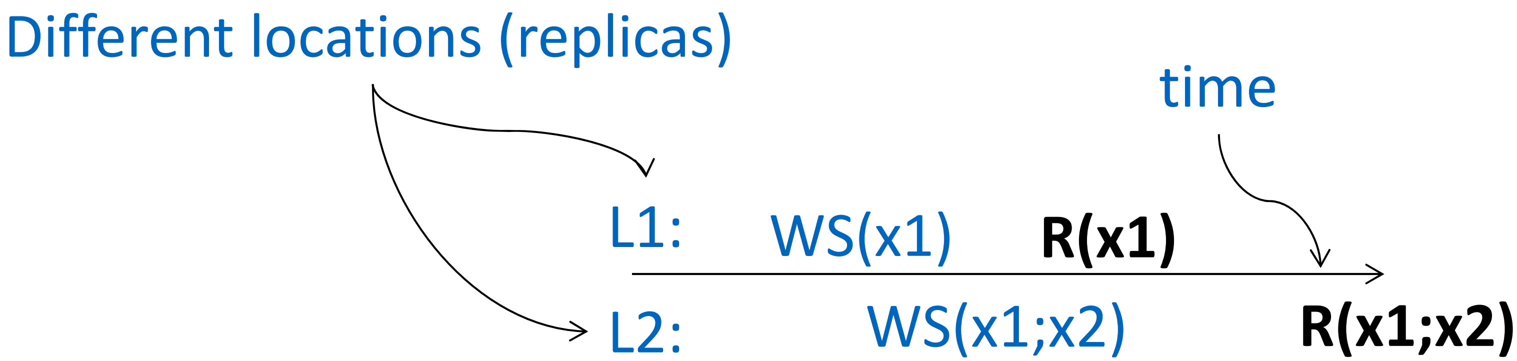
Client-Centric Consistency

- **Client-centric** consistency puts the emphasis on maintaining a **consistent view** for a **single process**, instead of on the data stored in the system
- Emphasis on **read-write conflicts**, and we assume **write-write** conflicts do not exist (i.e., no concurrent writes)
- Client-centric consistency models describe what happens when a **single client** writes/reads from **multiple replicas**

Eventual Consistency

- Eventual consistency states all replicas **eventually** converge when write operations stop
  - E.g., lazy replication using gossiping (cf. replication)
- Very weak form of consistency with no time bound, but **highly available** (i.e., always return a value, but could be stale)
- Works fine if a client always reads from the **same** replica…
- …but gives "weird" results if client reads from **multiple replicas**:
  - In a mobile scenario
  - During replica failure
- Client-centric consistency models describe what happens when a **single** client writes/reads from **multiple replicas**

Notation



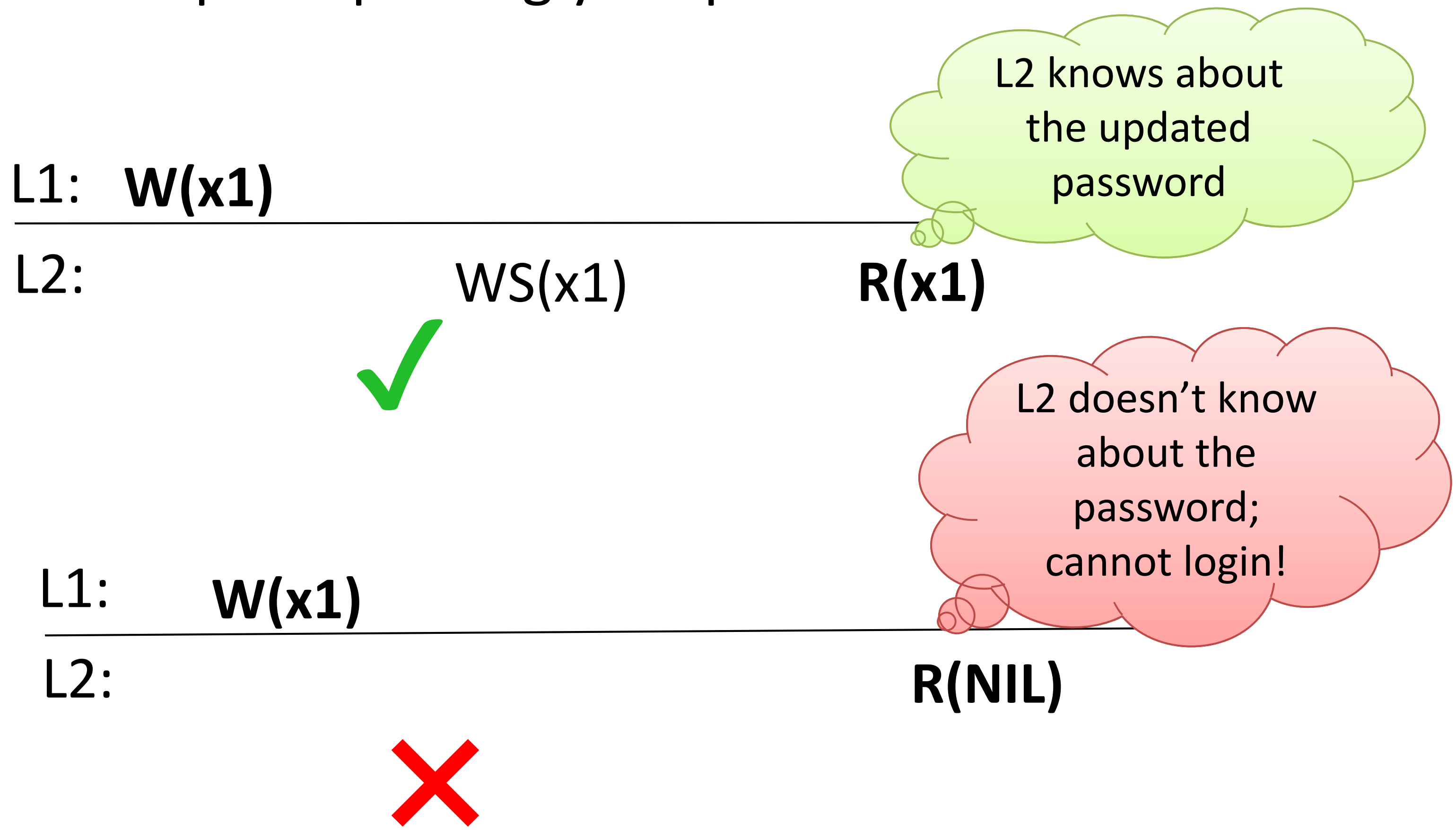
- **Reads** and **writes** by **one client** at **different locations**
- *WS* represents a series of write operations at L_i
- E.g., *WS(x1)* denotes that only op *x1* has been written; *WS(x1;x2)* denotes that op *x1* and then later *op x2* have been written
- R(x1;x2) means the read returns a value formed by operation x1 followed by x2
- W(x1), R(x1) are write to and read from x1

Read-Your-Writes Consistency

- Informally, ... *W* ... *R* ... (by same process!)
- If a read *R* follows a write then *W* is included in the set of writes read by *R*
- **A write operation is always completed before a successive read operation by the same process**, no matter where the read operation takes place

Read Your Writes

Example: Updating your password on a cluster



Monotonic-Reads Consistency

- … *WS1(..) R* … *WS2(..) R* … (s.t. $WS1(..) \subseteq WS2(..)$)
- If a process reads a value formed by a set of operations, *WS*, any successive read operation of that process will always return a **value** formed from a **superset** of *WS*
- A process always sees **more recent data** (but not necessarily fresh!)
- If a process reads again from another replica that replica must have already received the relevant older operations

Example: Monotonic Reads

Example: E-mail client
Each read returns the list of emails,
Each write adds one e-mail

L1: WS(x1) **R(x1)**

L2: WS(x1;x2) **R(x1;x2)**

L2 knows about all e-mails

L1: WS(x1) **R(x1)**

L2: WS(x2) **R(x2)**

L2 doesn't know about the e-mail x1!

Writes-Follow-Reads Consistency

- *… WS(X1, …, Xn)* ***R1*** *… WS(X1, …, Xn)* ***W2*** *…*
- If a read *R1* precedes a write *W2* then at any replica, if *W2* is written, it is preceded by WS, the write set read by *R1*
- Any successive write operation by a process will be performed on a state that is **up to date** with the value **most recently read** by that process

Writes Follow Reads

Example: Facebook Wall

Reading a post

Replying to a post

L1: WS(x1) **R(x1)**

L2: WS(x1) **W(x2)**

L2 knows, writes a reply to a known post

✓

L1: WS(x1) **R(x1)**

L2: WS(NIL) **W(x2)**

L2 doesn't know about the post to reply to!

Monotonic-Writes Consistency

- Informally, *... W1 ... W2 ...*

- If a write *W1* precedes a write *W2* then on any replica, *W1* must precede *W2*

- In other words, the **writes by the same process** are performed in the same order at every replica

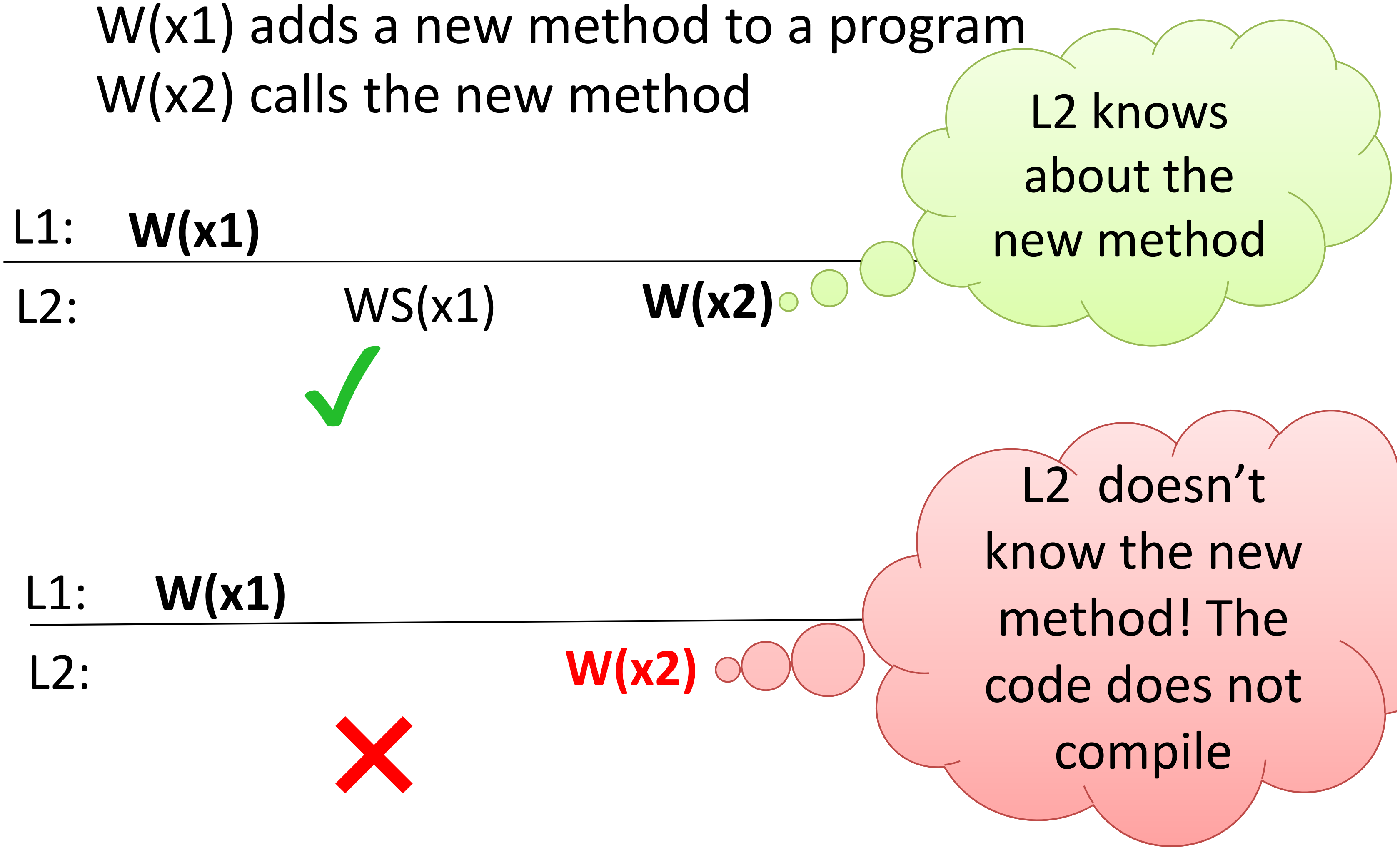
- Resembles FIFO consistency model

Monotonic Writes

Example: Replicating software code

W(x1) adds a new method to a program

W(x2) calls the new method



Sketch of Mechanisms Realizing CCMs

- Use **optimistic** concurrency control to verify that the replica has the needed information
- Client keeps track of the **read-set** and **write-set**, depending on which consistency model is used
- When processing an operation, the two sets are **passed** to the replica, who must **verify** those operations are already processed
- Otherwise, the incoming operation is **queued or rejected**

Summary: Client-centric consistency

- Eventual consistency can be used when:
  - **Few** concurrent write occurs
  - Used in Dynamo, Cassandra, Riak
- Eventual consistency provides **high availability** and scalability
- Client-centric consistency dictates how reads and writes should look from the client's perspective
  - Read your writes
  - Monotonic reads
  - Monotonic writes
  - Writes follow reads