Consistency and Replication
Part I

CS403/534

Distributed Systems

Erkay Savas
Sabanci University

Consistency and Replication

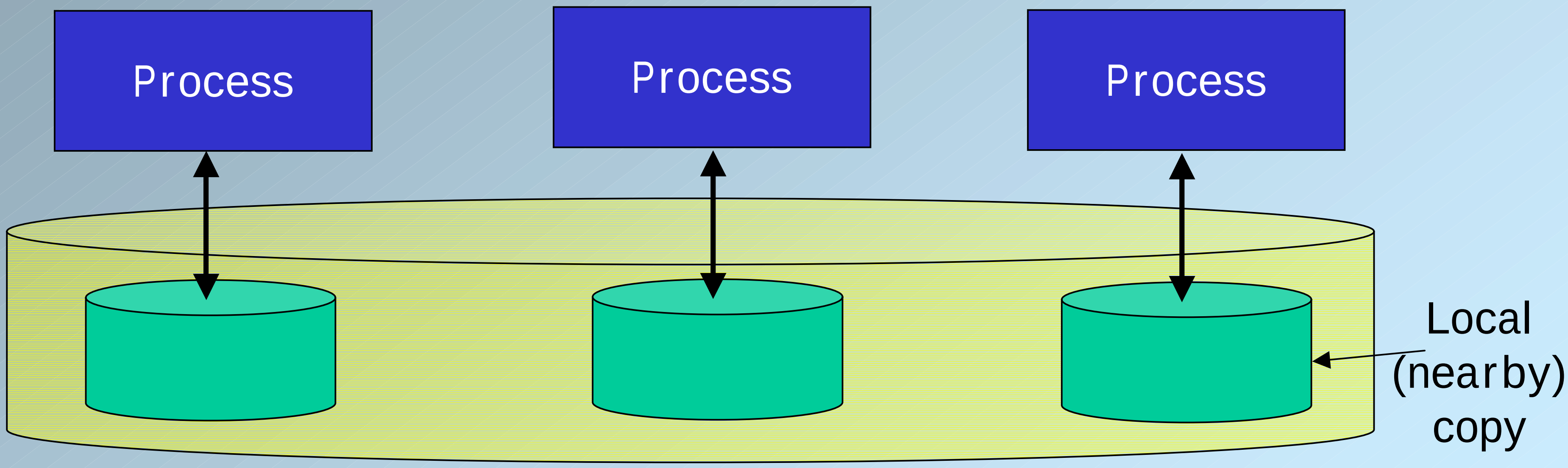
- Reasons
  1. Enhance reliability
  2. Improve performance
- Major problems:
  - Keep the replicated data consistent
  - More bandwidth between the replicas to communicate updates
- Overview
  - Consistency models: data-centric, client-centric
  - Implementation of consistency models

Replication as a Scaling Technique

- Main problem: to keep replicas consistent.
  - Updates must be propagated to every replica
  - All conflicting (update) operations to be performed in the same order everywhere
- Issues involved:
  - Negative impact on bandwidth requirements
  - Enforcing an ordered relationship between events require synchronization: physical or logical clocks
  - How seriously we should take the strict (tight) consistency approach.
  - To what extent consistency loosened highly depended
    - on the purpose for which those data are used
    - on the access & update patterns of the replicated data

Data-Centric Consistency Models

- **Data store:**
  - A term used to refer to a *physically distributed shared data*
  - Read and write operations are of concern
  - Each process has a replica nearby



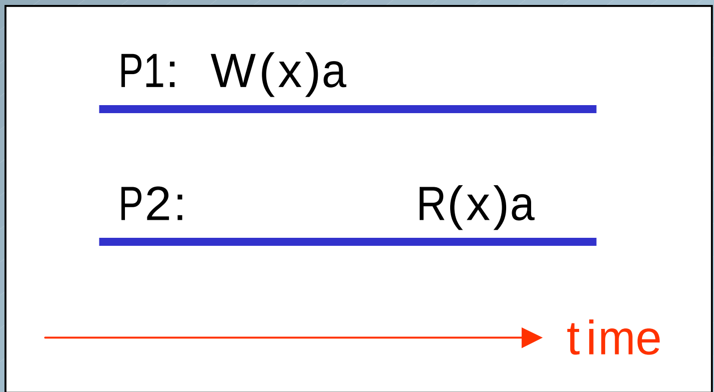
Distributed data store

Consistency Model

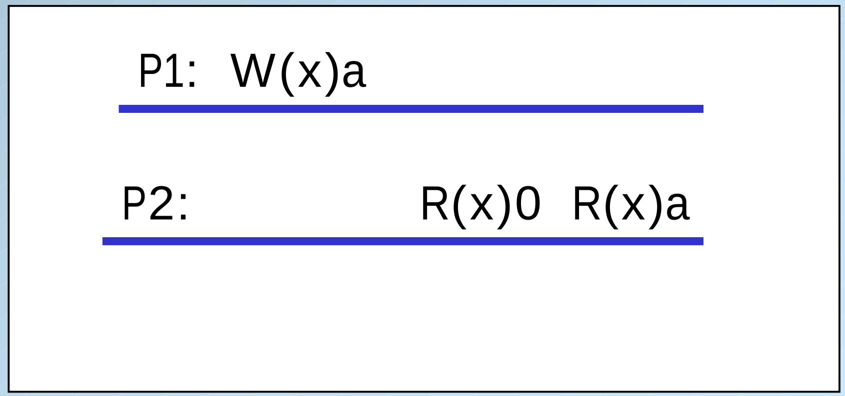
- It is a contract between the processes and data store
- If processes agree to obey certain rules, the store promises to work correctly.
- For example, when a process wants to read a data item, then it expects to have the value that is the result of the last write operation on the data item.
- Without a global clock, it is hard to decide which write operation is the last.

Strong Models: Strict Consistency

- *Any read on a shared data item x returns a value corresponding to the result of the most recent write on x.*
- All writes are instantaneously visible to all processes
- It is impossible to implement it in distributed systems since strict consistency relies on *absolute global time*



a) A strictly consistent store.



b) A store that is not strictly consistent.

Sequential Consistency

- *When processes run concurrently on (possibly) different machines, any valid order of read and write operations is acceptable, as long as all processes observe the same order of operations*
- No reference to “most recent” write
- No reference to physical time
- No reference to logical time
- Somewhat a weaker consistency model

Sequential Consistency: Example

P1: W(x)a

P2: W(x)b

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

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

a) A sequentially consistent data store.

P1: W(x)a

P2: W(x)b

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

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

b) A data store that is not sequentially consistent.

Sequential Consistency: Linearizability

- **Linearizability** is similar to sequential consistency; but stronger than sequential consistency (but weaker than strict consistency).
- Operations are assumed to receive a timestamp using a <u>globally available clock</u>
  - <u>one with finite precision</u>
- If $\mathbf{ts_{OP1}(x) < ts_{OP2}(x)}$ then **OP1(x)** should precede **OP2(x)** in the sequence
- Linearizable data is also sequentially consistent
- Processes use loosely synchronized clocks

Sequential Consistency : Example (1)

Process P1	Process P2	Process P3
`x = 1;`	`y = 1;`	`z = 1;`
`print (y, z);`	`print (x, z);`	`print (x, y);`

- Three concurrently executing processes: **`P1, P2, P3`**.
- Initial setting: **`x = y = z = 0`**
- Assignment →write
- print →read
- With six operations, there are 6! = 720 possible execution sequences, (however, some of these violate program order)
- Only, 90 of them preserve the program order.

Sequential Consistency : Example(2)

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time. The signature is a string that is the concatenation of outputs of P1, P2, P3 in that order.

`x = 1;`	`x = 1;`	`y = 1;`	`y = 1;`
`print (y, z);`	`y = 1;`	`z = 1;`	`x = 1;`
`y = 1;`	`print (x,z);`	`print (x, y);`	`z = 1;`
`print (x, z);`	`print(y, z);`	`print (x, z);`	`print (x, z);`
`z = 1;`	`z = 1;`	`x = 1;`	`print (y, z);`
`print (x, y);`	`print (x, y);`	`print (y, z);`	`print (x, y);`
Prints: 001011	**Prints: 101011**	**Prints: 010111**	**Prints: 111111**
Signature: 001011	**Signature: 101011**	**Signature: 110101**	**Signature: 111111**
(a)	**(b)**	**(c)**	**(d)**

Time

Sequential Consistency : Example(3)

- Not all the signatures are allowed.
- Is the **Signature 001001** allowed?
- P1: 00 → P1 executes before P2 and P3
- P2: 10 → P2 executes after P1 but before P3 (OK)
- P3: 01 → P3 executes before P1 starts (Contradiction!)

Casual Consistency

- Necessary condition:
  - *Writes that are potentially casually related must be seen by all processes in the same order*.
  - Concurrent writes (that are not causally related) may be seen in a different order on different machines.
- Two writes are causally related to each other through a read operation.
- Vector timestamps are used to implement casual consistency

Casual Consistency: Example (1)

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**

- This sequence is allowed with a casually-consistent store, but not with sequentially or strictly consistent store.

Casual Consistency: Example (2)

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

(a)

a) A violation of a casually-consistent store.

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

(b)

b) A correct sequence of events in a casually-consistent store.

FIFO Consistency

- Necessary Condition:
  - Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.
- Weaker form of strong consistency
- All writes generated by different processes are concurrent

FIFO Consistency: Example (1)

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

- A valid sequence of events of FIFO consistency

FIFO Consistency: Example (2)

P1:	P2:	P3:
`x = 1;`	`y = 1;`	`z = 1;`
`print (y, z);`	`print (x, z);`	`print (x, y);`

(a)	(b)	(c)
`x = 1;`	`x = 1;`	`y = 1;`
`print (y, z);`	**`y = 1;`**	`print (x, z);`
`y = 1;`	**`print(x, z);`**	**`z = 1;`**
`print(x, z);`	`print ( y, z);`	**`print (x, y);`**
`z = 1;`	`z = 1;`	`x = 1;`
`print (x, y);`	`print (x, y);`	`print (y, z);`
`Prints: 00`	`Prints: 10`	`Prints:  01`

- Statement execution as seen by the three processes from the previous slide. The statements in bold are the ones that generate the output shown.

FIFO Consistency: Example (3)

Process P1	Process P2
`x = 1;`	`y = 1;`
`if (y == 0) kill (P2);`	`if (x == 0) kill (P1);`

- Two concurrent processes.
- Both processes can be killed in FIFO consistency

Weak Consistency Model

- Basic idea:
  - individual read & write operations are not immediately made known to other processes.
  - Final effect is communicated (as in transactions)
  - <u>synchronization variable</u> **`(S)`** (a lock or barrier)
  - A synchronization variable has only one operation: **`synchronize(S)`**
  - A process gains exclusive access to a critical region through synchronization operation
  - While a process is in the critical region, the inconsistencies will happen.
  - **`synchronize`** operation pushes local updates to other replicas + brings about the remote updates to the local replica

Weak Consistency: Properties

1. Access to synchronization variable is sequential
   - If processes P1 and P2 call **`synchronize(S)`**, the execution order of these operations will be the same everywhere
2. Synchronization flushes the pipeline
   - It forces all writes *that are in progress or partially completed or completed at some local copies but not all* to complete everywhere.
3. When data items are accessed, either for reading or writing, all previous synchronization will have been completed.
   - By doing synchronization before reading a shared data, a process can be sure of getting the most recent values.

Weak Consistency: Properties

- A good consistency model when isolated accesses to shared data is rare.
- With weak consistency, sequential consistency is enforced between groups of operations
- Synchronization variables to delimit those groups.
- Weak consistency models tolerates a greater degree of inconsistency for a limited amount of time.

Weak Consistency: Example

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

P2: S R(x)b S

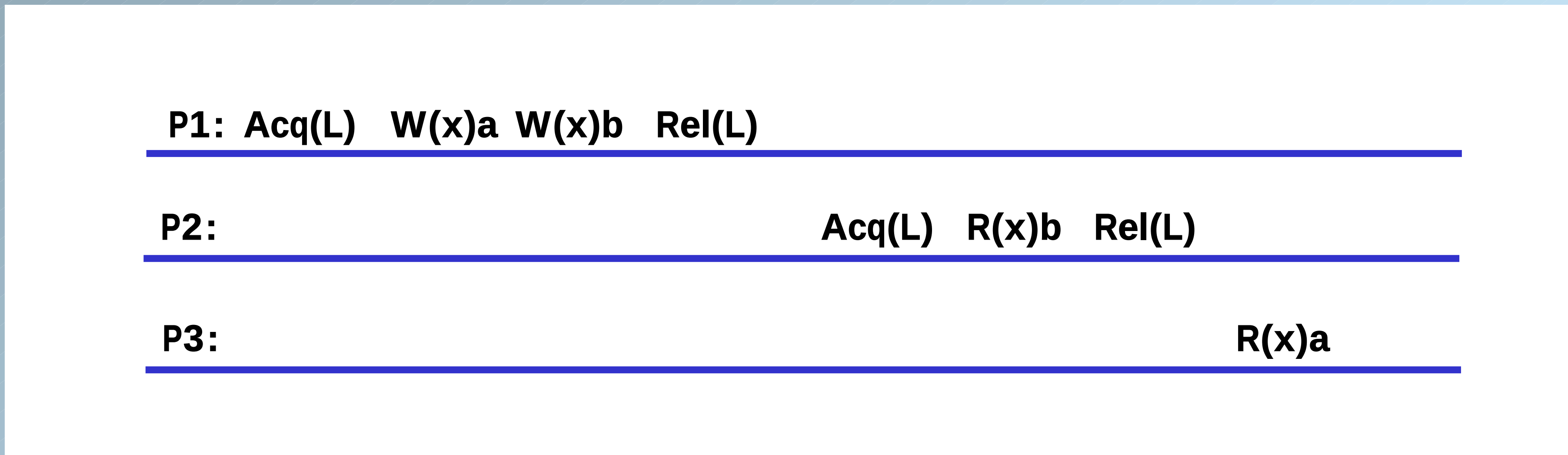
P3: R(x)a R(x)b S

Release Consistency (1)

- <u>Drawbacks of weak consistency</u>: When a synchronization variable is accessed by a process, the data store does not know if
  - the process is finished writing data (exiting critical region)
  - or it is just about to read data (entering critical region)
- Therefore, when an access to a synchronization variable is initiated, the local data store does two things:
  1. All locally initiated writes have been propagated to all other copies
  2. Gather in all writes from other copies
- If data store makes the distinction between entering or exiting from critical regions, problems will be solved

Release Consistency: Example

- Two types of synchronization operations are used:
  1. **Acquire**
  2. **Release**
- Programmer is responsible to call these operations before entering and exiting critical region



- A valid event sequence for release consistency.

Release Consistency (2)

- Eager release consistency: release operation pushes out all the modified data to all other process
  - It does not matter whether the other processes need that data
- Lazy release consistency: release operation does send nothing
  - Acquiring process must come and get them.

Entry Consistency

- With release consistency, all local updates are made available to all copies during the release of the lock
- With entry consistency, each individual shared data item is associated with some synchronization variable (e.g. lock or barrier)
- When *acquiring* the synchronization variable, the most recent values of its associated shared data item must be fetched
- **Note:** where release consistency affects *all* shared data, entry consistency affects only those associated with a synchronization variable.

Entry Consistency: Conditions

1. At an acquire, all remote changes to the guarded data must be made visible
2. Before updating a shared data item, a process must enter the critical region in exclusive mode
3. If a process wants to enter a critical region in nonexclusive mode, it must first check with the owner of the synchronization variable to fetch the most recent copies of shared data.

Entry Consistency: Example

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)

P2: Acq(Lx) R(x)b R(y)NIL

P3: Acq(Ly) R(y)b

- A valid event sequence for entry consistency.
- **Questions:** What would be a convenient way of making entry consistency more or less transparent to programmers with distributed objects?

Summary of Consistency Models

Consistency models not using synchronization operations (Strong models)

Consistency	Description
Strict	Absolute time ordering of all shared accesses.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a 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 used. Writes from different processes may not always be seen in that order

Summary of Consistency Models

Models with synchronization operations.

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

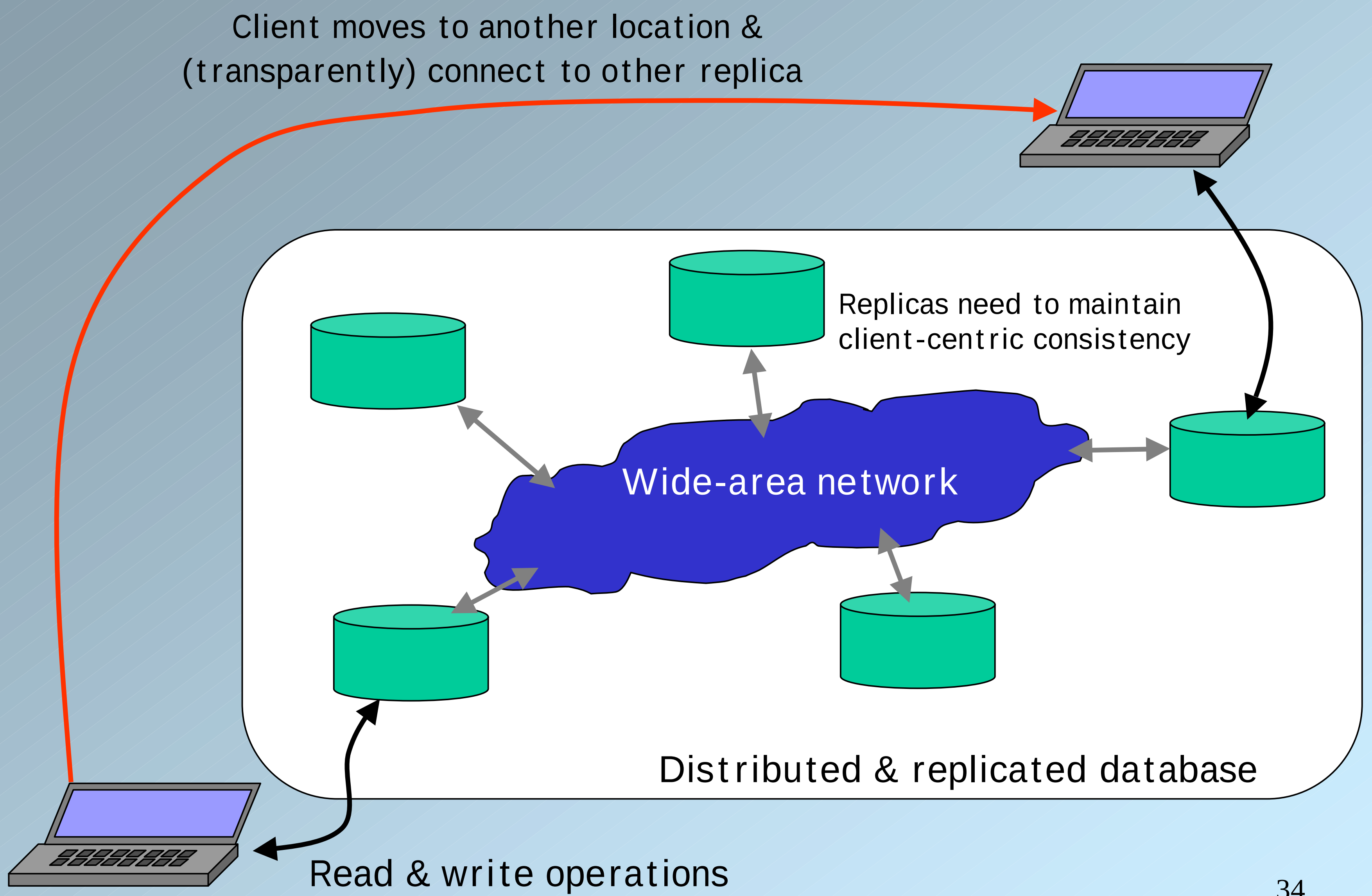
Client-Centric Consistency Models

- **Data-centric consistency:** guarantees systemwide consistency on data store
- **Client-centric consistency:** provides guarantee of consistency for accesses of a single client
  - Most large-scale distributed systems apply replication for scalability,
  - Simultaneous updates are rare; and if they happen they are easy to resolve
  - **DNS:** Updates are done by one processes in a domain (no write-write conflicts); propagate slowly
  - **WWW:** Caches all over the place, but there need be no guarantee that you are reading the most recent version of a page

Eventual Consistency

- Large-scale distributed and replicated databases can tolerate a relatively high degree of inconsistency.
- *If no updates take place for a long time, all replicas will gradually become* 100% *consistent*.
- This model guarantees that updates to be propagated to all replicas *eventually*.
- Write-write conflicts are often relatively easy to solve assuming only a small group of processes can perform updates.
- Very inexpensive to implement.

Consistency for Mobile Client



Notation

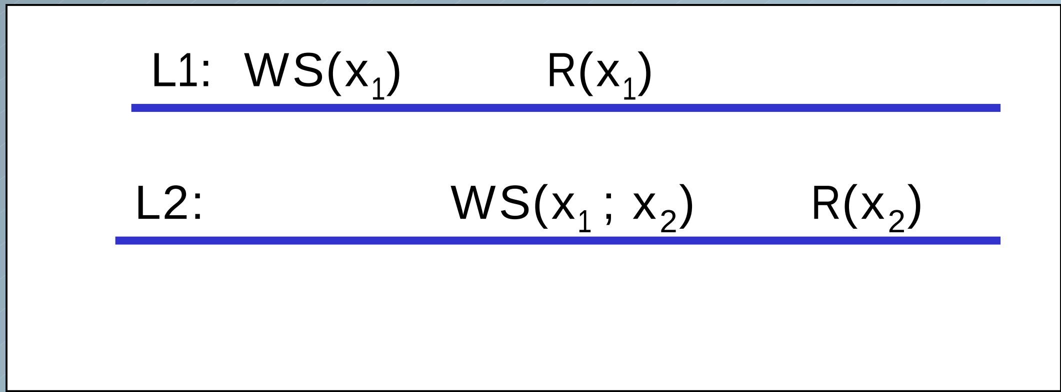
- **Assumption:** Data items have an associated owner
- $\mathbf{x_i[t]}$: the version of data item $\mathbf{x}$ at local copy $\mathbf{L_i}$ at time $\mathbf{t}$.
- $\mathbf{WS(x_i[t])}$: write operations that took place at $\mathbf{L_i}$ since the initialization.
- $\mathbf{WS(x_i[t_1];\ x_j[t_2])}$**:** write operations in $\mathbf{WS(x_i[t_1])}$ have been also performed at local copy $\mathbf{L_j}$ at a later time $\mathbf{t_2}$.
- Time index is dropped from the notation if the timing or ordering of operations are obvious from the context.

Models: Monotonic Reads

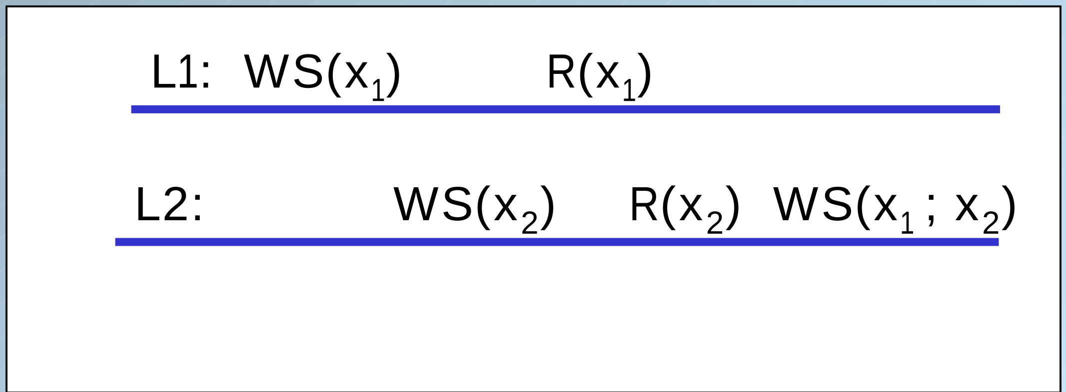
- *If a process reads the value of a data item* **x**, *any successive read operation on* **x** *by that process will always return that same or a more recent value*
- This guarantees that if a process has seen a value **x** at time **t,** it will never see an older version of **x** afterwards.
- Example: Distributed e-mail database
  - Updates are propagated in a lazy fashion
  - Reading (not modifying) incoming **e-mails** while you are on the move.
  - Each time you connect to a different e-mail server; that server fetches (*at least*) all the updates from the server you previously connected.

Monotonic Reads: Example

The read operations performed by a single process *P* at two different local copies of the same data store.



a) A monotonic-read consistent data store



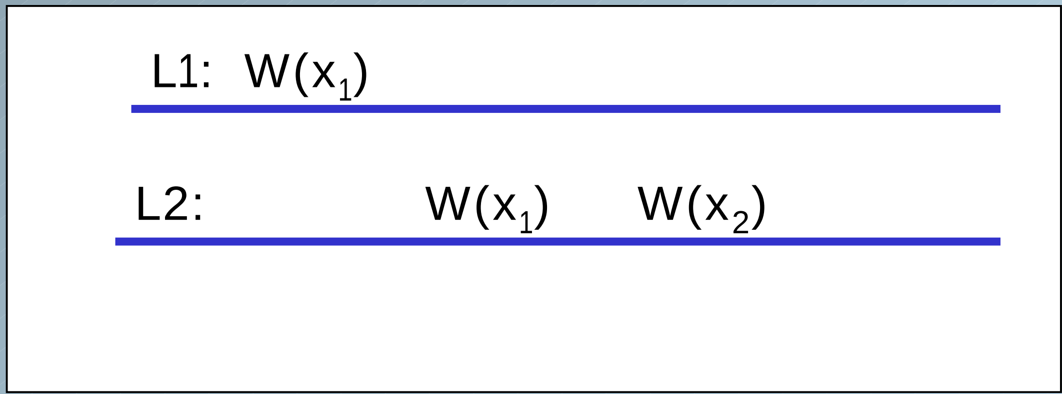
b) A data store that does not provide monotonic reads.

Monotonic Writes

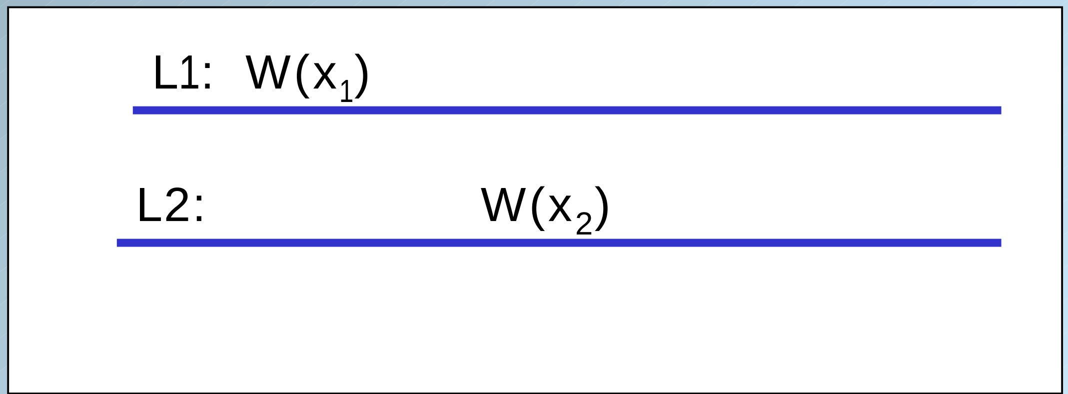
- *A write operation by a process on a data item* **x** *is completed before any successive write operation on* **x** *by the same process*
  - Resembles the FIFO consistency; different in the sense that monotonic writes model cares what the single process, which is performing writes, sees.
  - A write operation on a copy of data item **x** is performed only if that copy has been brought up to date by means of any preceding write operation that may have taken place on other copies of **x** by the same process.
  - **Example:** Updating a C library at server S_i and ensuring that all previous updates on all components of the library on which compilation and linking depends, are also propagated to S_i.

Monotonic Writes: Example

The write operations performed by a single process *P* at two different local copies of the same data store



a) A monotonic-write consistent data store.

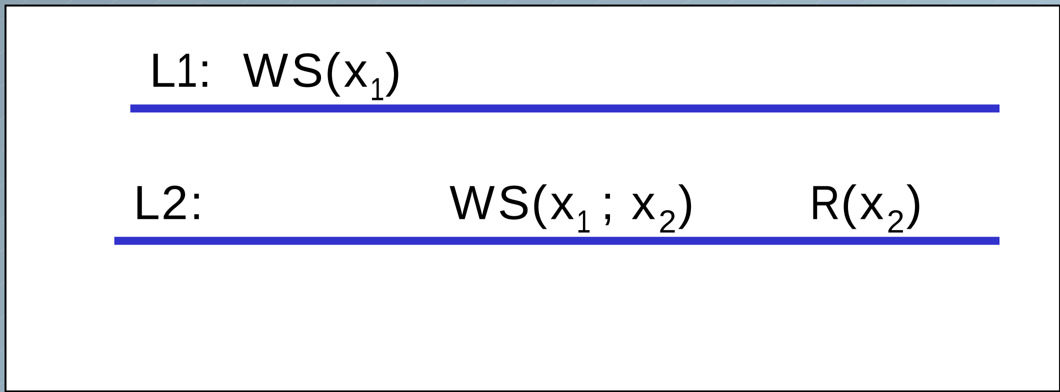


b) A data store that does not provide monotonic-write consistency.

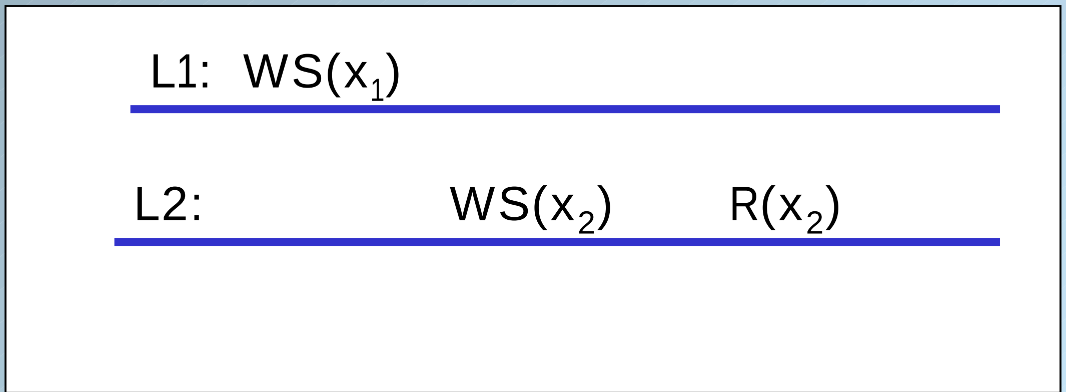
Read Your Writes

- *The effect of a write operation by a process on data item* **x***, will always be seen by a successive read operation on* **x** *by the same process*
- A write operation is always completed before a successive read operation by the same process, no matter where this read operation takes place.
- **Example:** Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.
- **Example:** updating passwords on password server

Read Your Writes: Example



a) A data store that provides read-your-writes consistency.



b) A data store that does not.

Client-Centric Consistency: Implementation (1)

- Straightforward implementation
  - Each write operation is assigned a globally unique id
  - For each client, we keep track of two sets of write ids: read set and write set
  - Read set consists of write ids of write operations that is relevant for the read operations by the client
  - Write set consists of write identifiers for write operations performed by the client
  - Example: Monotonic read consistency
  - A client wants to perform a read operation at a server
  - The client gives its read set to the server
  - Server checks if all the identified write operations are performed locally.
  - If not it contacts other servers

Client-Centric Consistency: Implementation (2)

- Example: Monotonic read consistency (cont)
  - Contacted server must have logged all the write operations so that it can be replayed at another server
  - Write operations must be performed in the order they were initiated.
  - Lamport timestamps can be included in the write id.
- Drawback: Costly
  - Read and write sets may get too large
- Better Approach
  - Use vector timestamps

Monotonic Reads with Vector Timestamps