

CS x55: DISTRIBUTED SYSTEMS [**CONSISTENCY**]

Shrideep Pallickara
Computer Science
Colorado State University



Topics covered in this lecture

- Consistent Ordering of Operations
 - ▣ Sequential consistency
 - ▣ Causal consistency
- Client-centric consistency models



CONSISTENT ORDERING OF OPERATIONS



Consistent ordering of operations

- Class of models from **concurrent programming**
- We will look at
 - ▣ Sequential consistency
 - ▣ Causal consistency

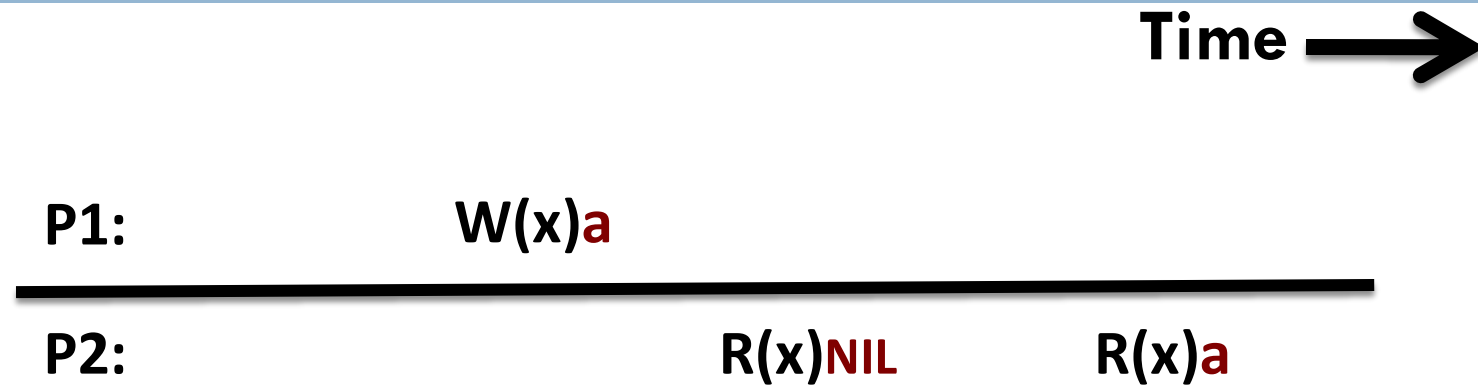


Sequential consistency: Notations

- Operations of processes depicted along time axis
- Write by a process P_i to data item x with value a
 - $W_i(x)a$
- Read by a process P_i of data item x that returns the value b
 - $R_i(x)b$
- All items are initially **NIL**



Two processes operating on the same data item



Time to propagate update of x to P2 is acceptable 😊



Sequential consistency

- Defined by Lamport
 - ▣ Context: Shared memory in multiprocessor setting
- When processes run concurrently
 - ▣ Any valid interleaving of read/write is acceptable
 - ▣ But all processes ***must see the same interleaving***



Sequential consistency example

Time →

P1: W(x)a

P2: W(x)b

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

P4: R(x)b R(x)a



Write operation of P2 appears to be **before** P1
This is acceptable



Sequential consistency: Example

Time →

P1: W(x)a

P2: W(x)b

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

P4: R(x)a R(x)b

P3 concludes final value is a
P4 concludes final value is b

Unacceptable



Sequential Consistency:

Another example

Process 1

x = 1
print(y,z)

Process 2

y = 1
Print(x,z)

Process 3

z = 1
Print(x,y)



Multiple interleaved sequences are possible

- With 6 statements there are
 - ▣ $6!$ possibilities = 720
 - ▣ Some of these **violate program order**
- 120 ($5!$) sequences begin with $x=1$
 - ▣ Half $\text{print}(x,z)$ before $y=1$
 - ▣ Half $\text{print}(x,y)$ before $z=1$
 - ▣ Only $\frac{1}{4}$ or 30 are valid
- Similarly, there are 30 that start with $y=1, z=1$
 - ▣ Total of 90 valid execution sequences



Different, but valid interleaving of the statements

Signature is the concatenation of the outputs of P1, P2 and P3

```
x = 1
print(y,z)
y = 1
print(x,z)
z = 1
print(x,y)
```

Prints: 001011
Signature: 001011

```
x = 1
y = 1
print(x,z)
print(y,z)
z = 1
print(x,y)
```

Prints: 101011
Signature: 101011

```
y = 1
z = 1
print(x,y)
print(x,z)
x = 1
print(y,z)
```

Prints: 010111
Signature: 110101

```
y = 1
x = 1
z = 1
print(x,z)
print(y,z)
print(x,y)
```

Prints: 111111
Signature: 111111



Contract between processes and shared data store

- Processes must accept **all valid results**
- Must work if any of them occurs



Invalid sequences in signature patterns

□ 000000?

- Print statements ran before assignments
- **Violates** program order

Process 1

x = 1
print(y,z)

Process 2

y = 1
Print(x,z)

Process 3

z = 1
Print(x,y)

□ 001001?

- {00} y and z were 0 when **P1** did its printing
 - **P1** executes its statements *before* **P2** and **P3** start
- {10} **P2** ran after **P1** started, but before **P3** started
- {01} **P3** must complete *before* **P1** starts
 - Not possible!



CAUSAL CONSISTENCY



Causal consistency

- **Weakens** sequential consistency
- Makes **distinction** between events that are *causally* related
 - ▣ If event b caused/is-influenced by event a
 - Everyone must see a before b
- Operations not causally related: **concurrent**




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**



Note: This is NOT ALLOWED in sequential consistency



Causal consistency example:

Example 2

Time →

P1: $W(x)a$

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

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

P4: $R(x)a$ $R(x)b$

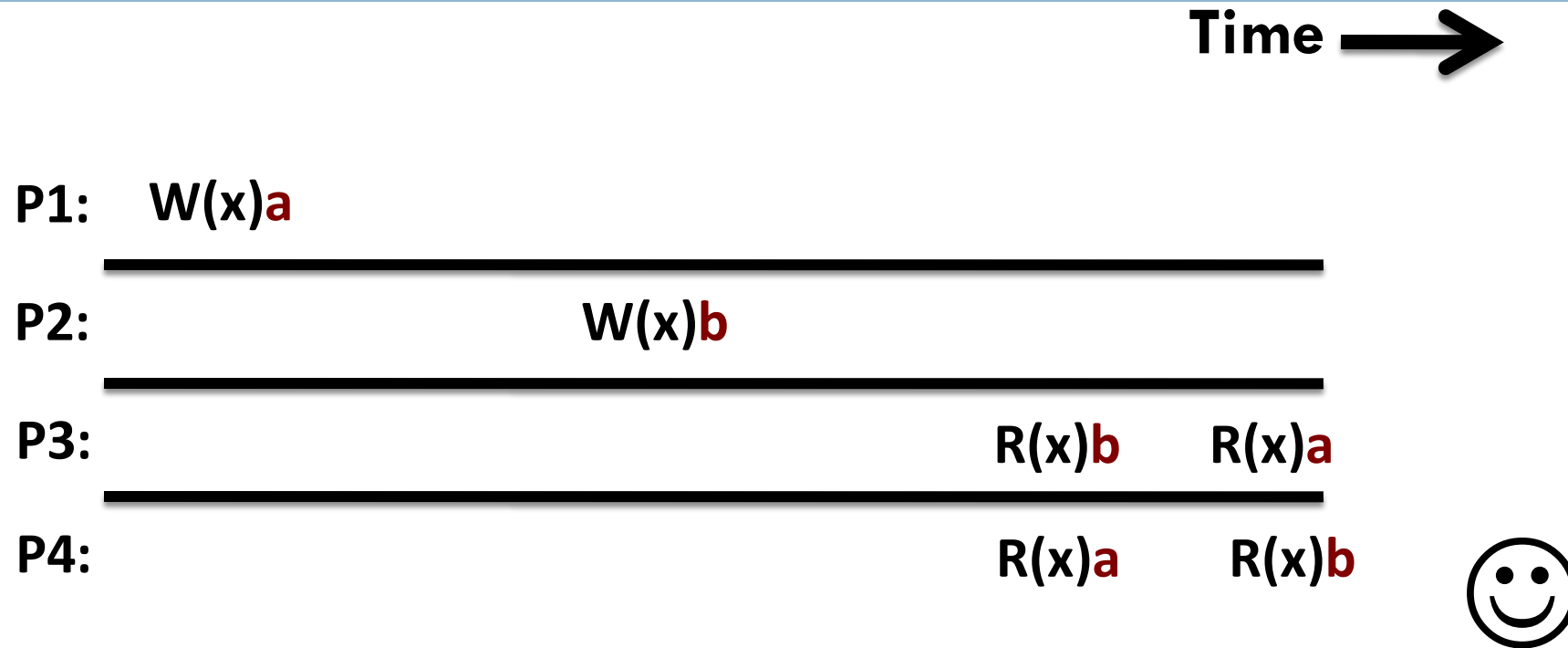


Writes $W_1(x)a$ and $W_2(x)b$ are **causally related**
Process must see them in the same order



Causal consistency example:

Example 3



Writes $W_1(x)a$ and $W_2(x)b$ are **concurrent writes**
Process can see them in different orders



GROUPING OPERATIONS



Concurrency using synchronization operations

- Operations bracketed by
 - ENTER_CS
 - LEAVE_CS
 - CS: Critical Section
- Semantics enforced using shared **synchronization** variables



Critical sections and synchronization variables

- Each synchronization variable has an **owner**
- Owner may repeatedly enter or exit critical section
- Process that does not own a synchronization variable
 - ▣ Must own it before it can enter critical section
 - ▣ **Acquire** by sending a message to the owner

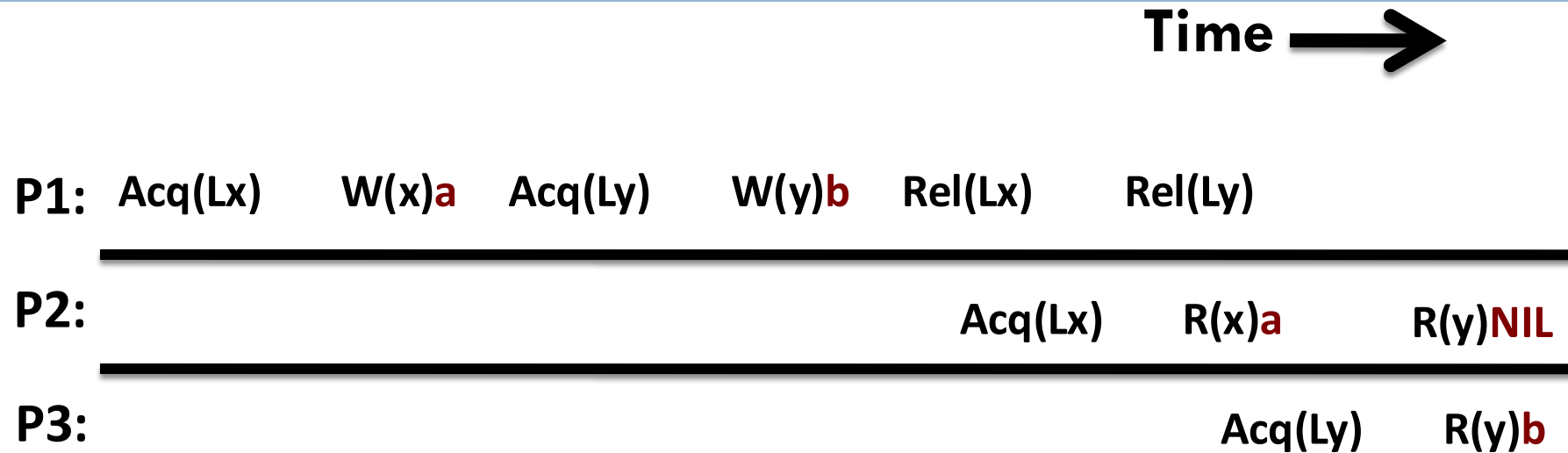


Rules for critical sections

- Acquire cannot complete until all guarded shared data is up to date
- Before *updating* a shared item
 - ▣ Enter critical section in **exclusive mode**
- If a process enters a critical region in non-exclusive mode
 - ▣ **Fetch recent copies** of the shared guarded data from owner



Entry consistency example



P2 does an acquire for x, but not y: *MAY* read NIL



CLIENT CENTRIC CONSISTENCY MODELS



Applications have different requirements about:

- Concurrency
- Consistency



Often only one or a few processes can perform updates

- How **fast** should these be propagated to processes that only read?
- DNS: Different domains managed by naming authority
 - Owner of that domain
 - **Write-write conflicts** never occur
 - Write-write conflicts result in overwriting uncommitted data (lost updates)
 - Read-write conflicts may occur
 - But it is still OK to do **lazy updates**
 - Read-write conflicts are also known as unrepeatable reads



Often only one or a few processes can perform updates

- Web pages updated by authors
 - ▣ Write-write conflicts *never* occur
 - ▣ Read-write conflicts *may* occur
 - Browsers or proxies cache these pages
 - Several users find this inconsistency acceptable



The DNS and Web page examples can be viewed as large (distributed) databases

- That tolerate a high degree of inconsistency
- If no updates take place for a long time
 - ▣ All replicas *gradually* become consistent
 - ▣ **Eventual consistency**

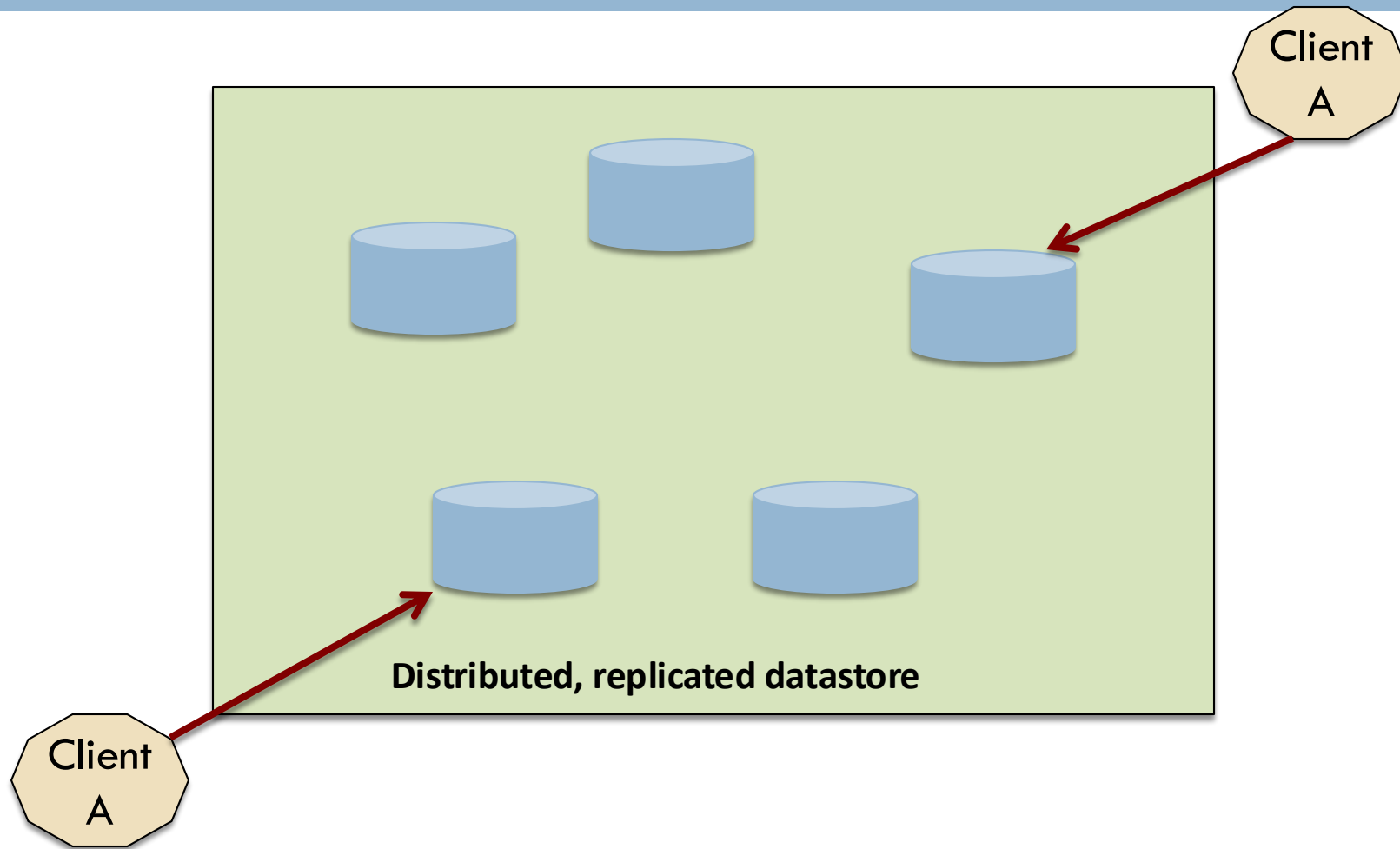


The caveat for eventual consistency

- Works fine as long as clients access the **same replica**
- **Problems** when you access *different replicas* within a short interval



An example of a mobile user accessing different replicas



CLIENT-CENTRIC CONSISTENCY MODELS



Client-centric consistency models

- Provides guarantees for a **single** client accessing the store
- No guarantees for **concurrent** accesses of store by **different** clients



Client-centric consistency models

- Monotonic read
- Monotonic write
- Read-your-writes
- Writes-follow-read



Notations for client-centric consistency

- Version of data item x at local copy L_i at time t
 - $x_i[t]$
- $x_i[t]$ is the **result** of a **series** of operations at L_i since initialization
 - ▣ This set of operations: $WS(x_i[t])$
 - ▣ Operation at L_i at t_1 and at L_j at time t_2
 - $WS(x_i[t_1]; x_j[t_2])$



Monotonic read consistency

- If a process reads a value of x , any successive read on x by that process returns either:
 - ▣ *Same* value
 - OR
 - ▣ **More recent** value
- If process sees a value of x at time t
 - ▣ It *never* sees an older version



A mailbox example of monotonic read consistency

- Each user's mailbox is replicated & distributed
- **Lazy/on-demand** updates
 - ▣ When copies need data for consistency the updates are propagated
- User reads mail in San Francisco ... goes to NYC
- Monotonic consistency
 - ▣ Messages in mailbox in SF are also there in NYC

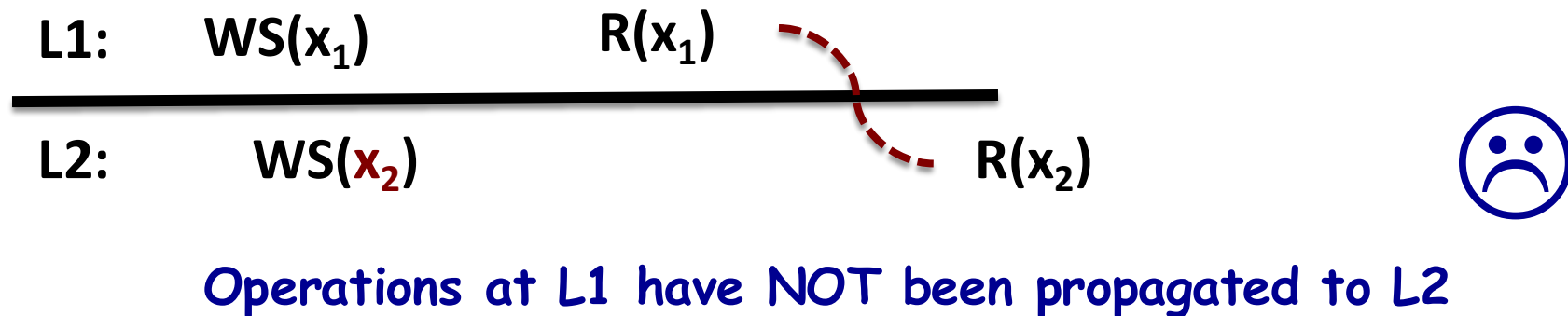
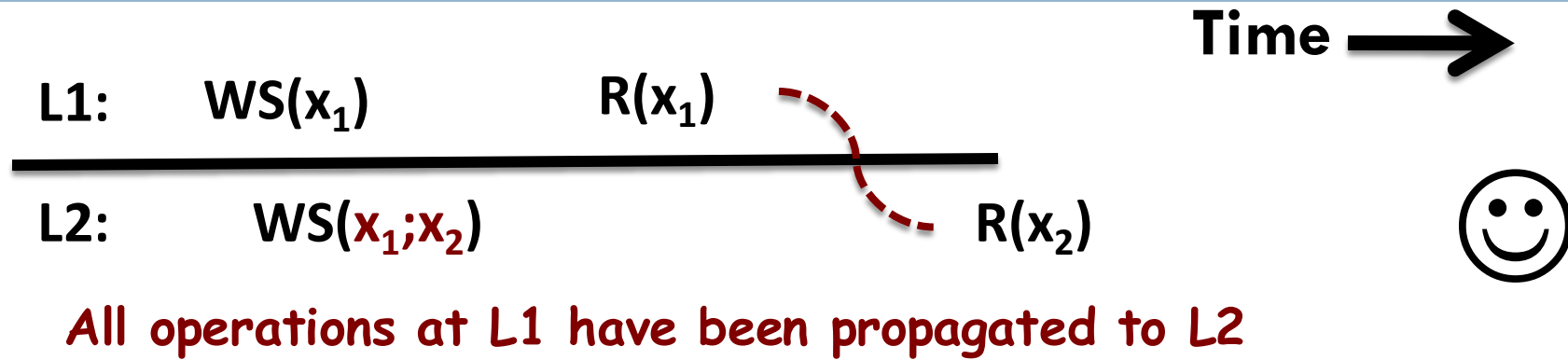


Representing client-centric consistency

- Time is along horizontal axis
- Different copies of a replica on the vertical axis
- Operations are carried out by a **single process**



Monotonic Read Consistency: Operations by a single process P



Monotonic Writes

[1 / 2]

- Write operation on data item x is completed
 - ▣ *Before any successive* write operation on x by the **same** process
- Copy on which write is performed
 - ▣ Reflects affect of a *previous* write
 - ▣ Irrespective of *where* it was initiated

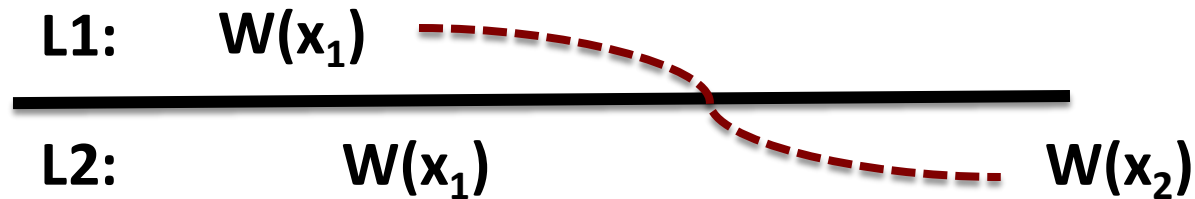


- When each write **completely overwrites** x
 - ▣ Getting things up to date is easier
- In most cases we perform **partial updates**; for e.g. x could be software library
 - ▣ We update functions etc. to get to the next version
 - ▣ If an update is performed to library
 - All **preceding updates must *first*** be performed

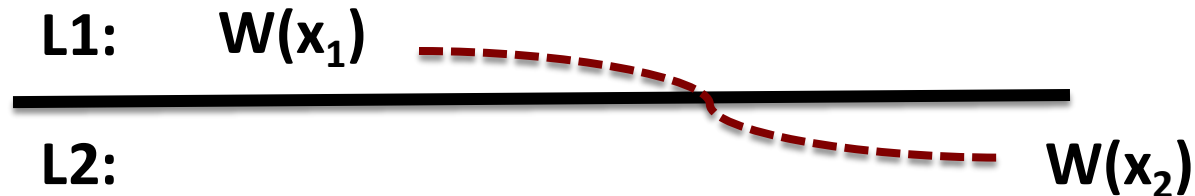


Monotonic Write Consistency: Operations by a single process P

Time →



Previous write at L1 has been propagated to L2



Write at L1 has NOT been propagated to L2



Read your writes

- Effect of a write operation on data item x
 - ▣ **Seen** by successive reads on x by the same process
- Write operation is always **completed before** a successive read operation
 - ▣ By same process
 - ▣ No matter *where* operations are performed



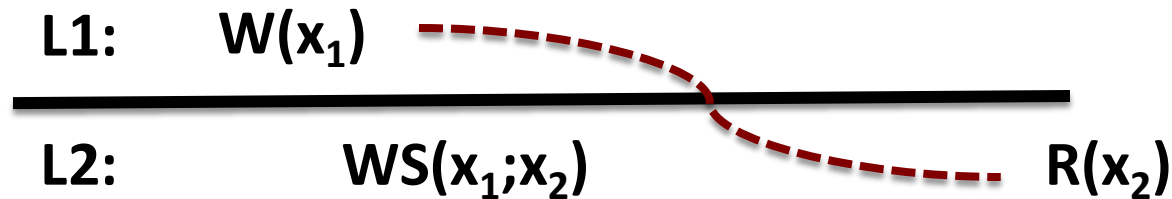
Example of inability to enforce read-your-write consistency

- Web designer creates a web page
- Tries to view it
- But browser/proxy has cached the older version
- With a read-your-write consistent browser
 - ▣ Cache is invalidated when page is updated
- Other example: Updating passwords

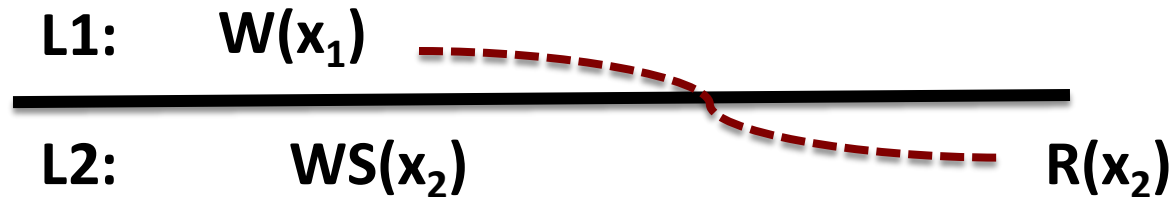


Read-your-Writes Consistency: Operations by a single process P

Time →



Previous write at L1 has been propagated to L2



Write at L1 has NOT been propagated to L2



Write Follow Reads

- Write operation by process on data item x
 - ▣ Following a previous read on x by the *same* process
 - Will take place on the **same (or more recent)** value of x
- Write operation on item x will be performed on a copy that is up to date
 - ▣ With value (most) recently read by process

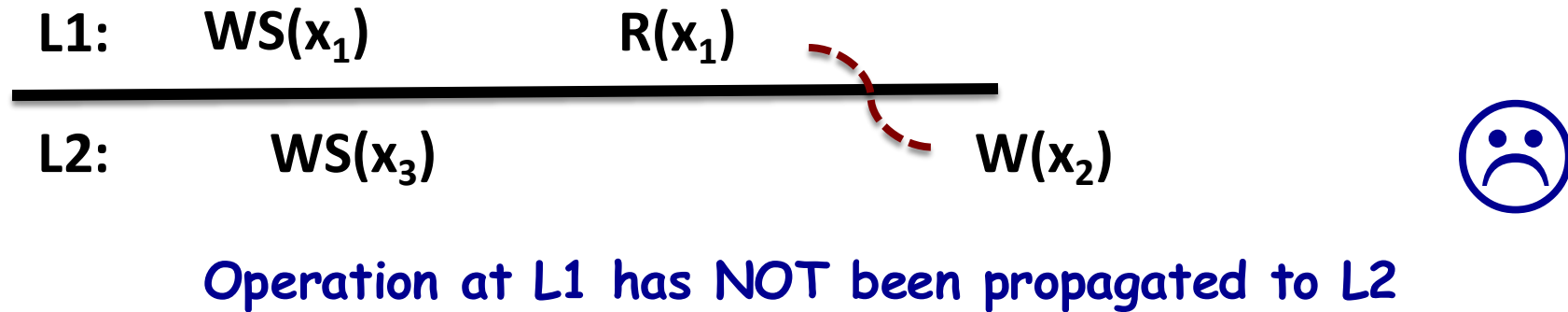
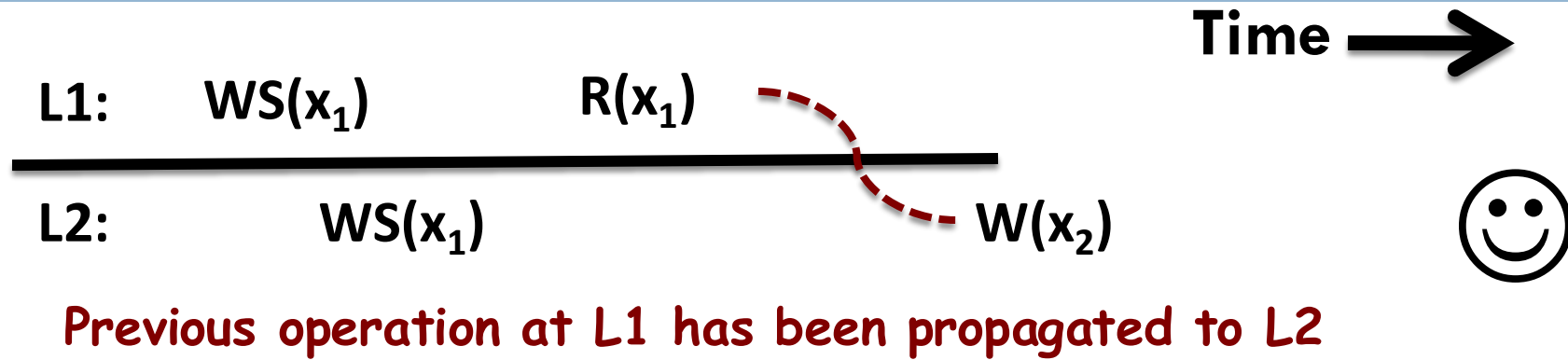


Write-follows-reads

- User reads an article **A**
- Reacts by posting article **B**
- Write follows reads consistency
 - ▣ **B** will be posted to a copy of the newsgroup
 - Only after **A** has been written



Writes-Follow-Reads Consistency: Operations by a single process P



The contents of this slide-set are based on the following references

- *Distributed Systems: Principles and Paradigms. Andrew S. Tanenbaum and Maarten Van der Steen. 2nd Edition. Prentice Hall. ISBN: 0132392275/978-0132392273.*
[Chapter 7]

