Distributed Systems -Consistency Models

Introduction

- What is consistency?
  - What processes can expect when RD/WR shared data concurrently
- When do consistency concerns arise?
  - With replication and caching
- Why are replication and caching needed?
  - For performance, scalability, fault tolerance, disconnection
- Consistency
  - is a property of the distributed system which says that every node/replica has the same view of data at a given point in time. This is irrespective of whichever client has updated the data
- What is a consistency model?
  - Contract between a distributed data system (e.g., DFS, DSM) and processes constituting its applications

Types of consistency

Type	Definition
strict	✓ The is the strongest form of memory coherence, having the most stringent consistency requirements. ✓ A shared-memory system is said to support the strict consistency model if the value returned by a read operation on a memory address is always the same as the value written by the most recent write operation to that address, irrespective of the locations of the processes performing the read and write operations.
sequential	✓ By Lamport ✓ A shared-memory system is said to support the sequential consistency model if all processes see the same order of all memory access operations on the shared memory.
causal	✓ relaxes the requirement of the sequential model for better concurrency. ✓ all processes see only those memory reference operations in the same (correct) order that are potentially causally related

Type	Definition
FIFO	✓ 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. ✓ "FIFO consistency is called *PRAM consistency* in the case of distributed shared memory systems
Pipelined Random-Access Memory (PRAM)	✓ It only ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed as if all the write operations performed by a single process are in a pipeline. ✓ Write operations performed by different processes may be seen by different processes in different orders."
Weak	✓ Synchronization accesses (accesses required to perform synchronization operations) are sequentially consistent. Before a synchronization access can be performed, all previous regular data accesses must be completed. ✓ Before a regular data access can be performed, all previous synchronization accesses must be completed. This essentially leaves the problem of consistency up to the programmer. ✓ The memory will only be consistent immediately after a synchronization operation."

Types of consistency

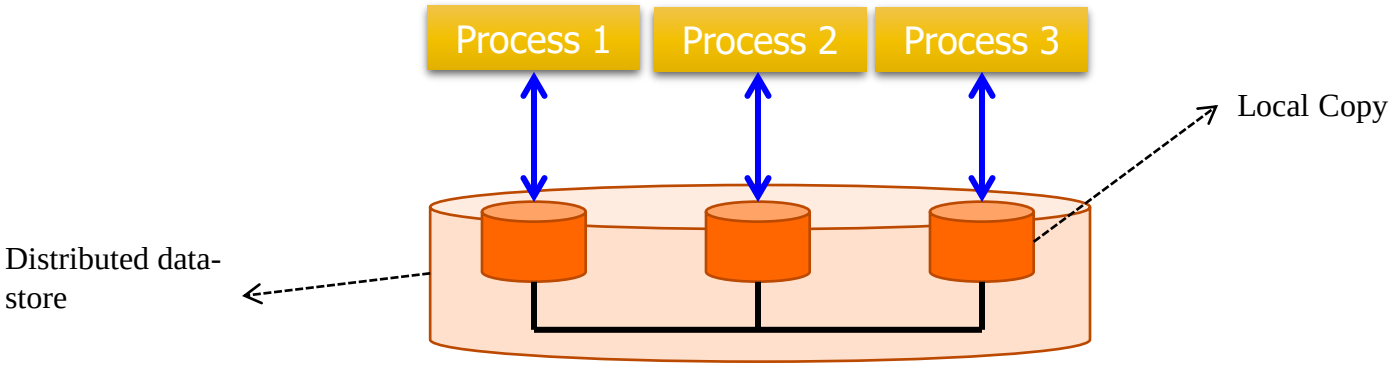
Type	Definition
Release	✓ Essentially the same as weak consistency, but synchronization accesses must only be processor consistent with respect to each other. ✓ Synchronization operations are broken down into *acquire* and *release* operations. ✓ All pending acquires (e.g., a lock operation) must be done before a release (e.g., an unlock operation) is done. Local dependencies within the same processor must still be respected.
entry	✓ Requires each ordinary shared data item to be associated with some synchronization variable, such as a lock or barrier. ✓ If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks. ✓ When an acquire is done on a synchronization variable, only those data guarded by that synchronization variable are made consistent."

Types of consistency

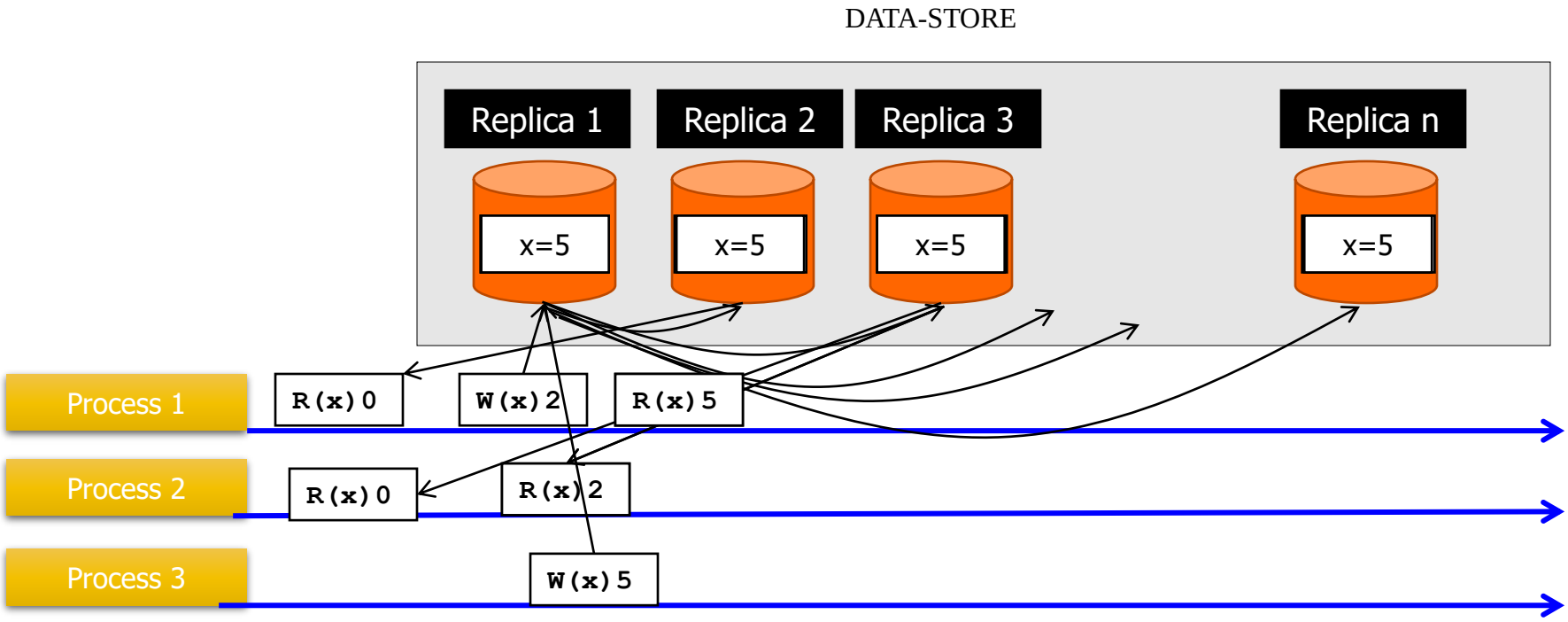
Type	Definition
Processor	✓ Writes issued by a processor are observed in the same order in which they were issued. ✓ However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. ✓ That is, two simultaneous reads of the same location from different processors may yield different results.
General	✓ A system supports *general consistency* if all the copies of a memory location eventually contain the same data when all the writes issued by every processor have completed.

Introduction to Consistency and Replication

- In a distributed system, shared data is typically stored in distributed shared memory, distributed databases or distributed file systems.
  - The storage can be distributed across multiple computers
  - Simply, we refer to a series of such data storage units as *data-stores*
- Multiple processes can access shared data by accessing any replica on the data-store
  - Processes generally perform read and write operations on the replicas



Maintaining Consistency of Replicated Data

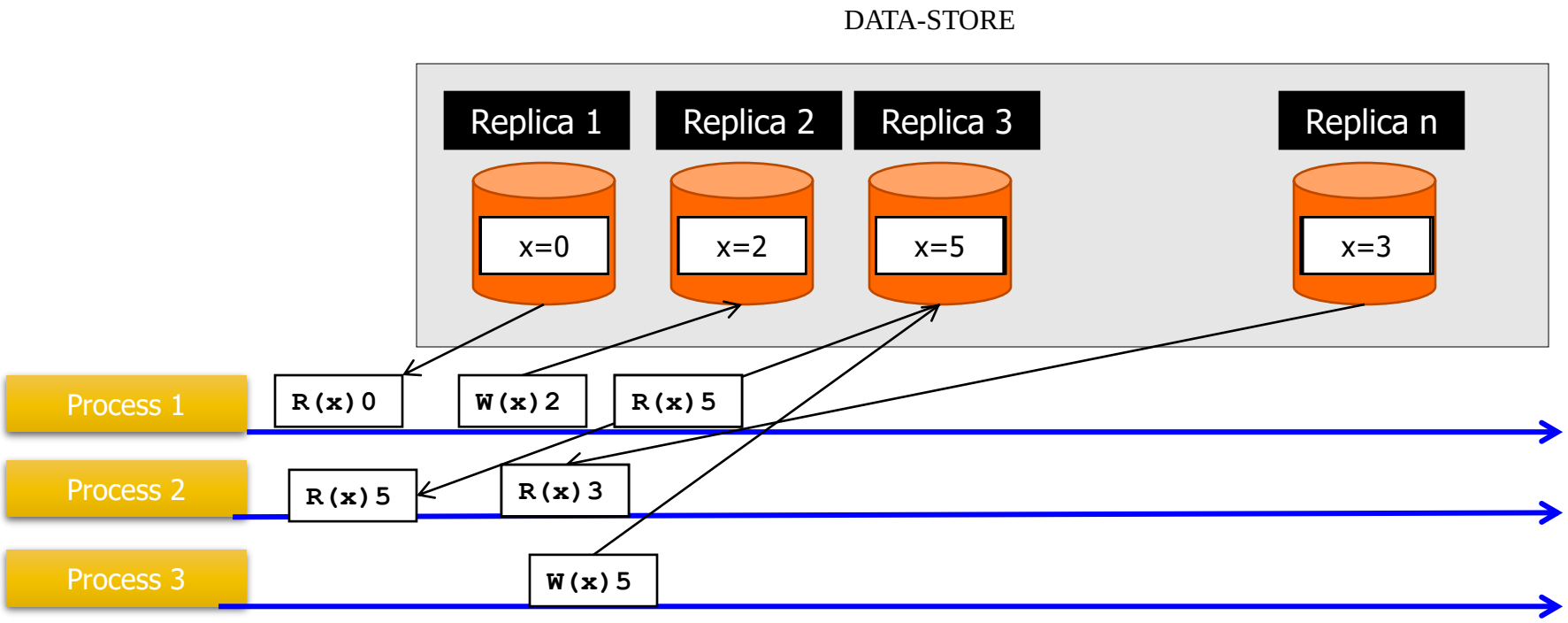


Strict Consistency

- Data is always fresh
  - After a write operation, the update is propagated to all the replicas
  - A read operation will result in reading the most recent write
- If there are occasional writes and reads, this leads to large overheads

P1 =Process P1 → =Timeline at P1 R(x)b =Read variable x; Result is b W(x)b = Write variable x; Result is b

Maintaining Consistency of Replicated Data

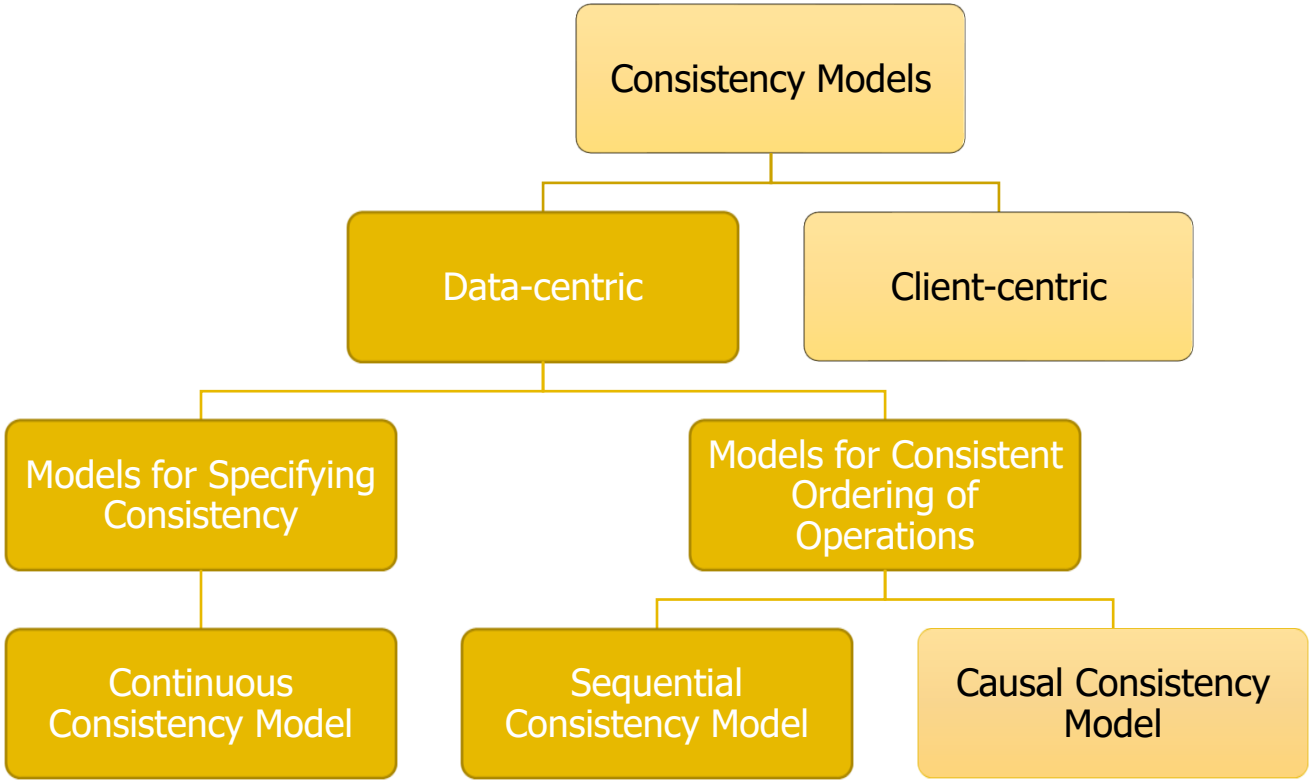


Loose Consistency

- Data might be stale
  - A read operation may result in reading a value that was written long back
  - Replicas are generally out-of-sync
- The replicas may sync at coarse grained time, thus reducing the overhead

P1 =Process P1 → =Timeline at P1 R(x)b =Read variable x; Result is b W(x)b = Write variable x; Result is b

Consistency Model



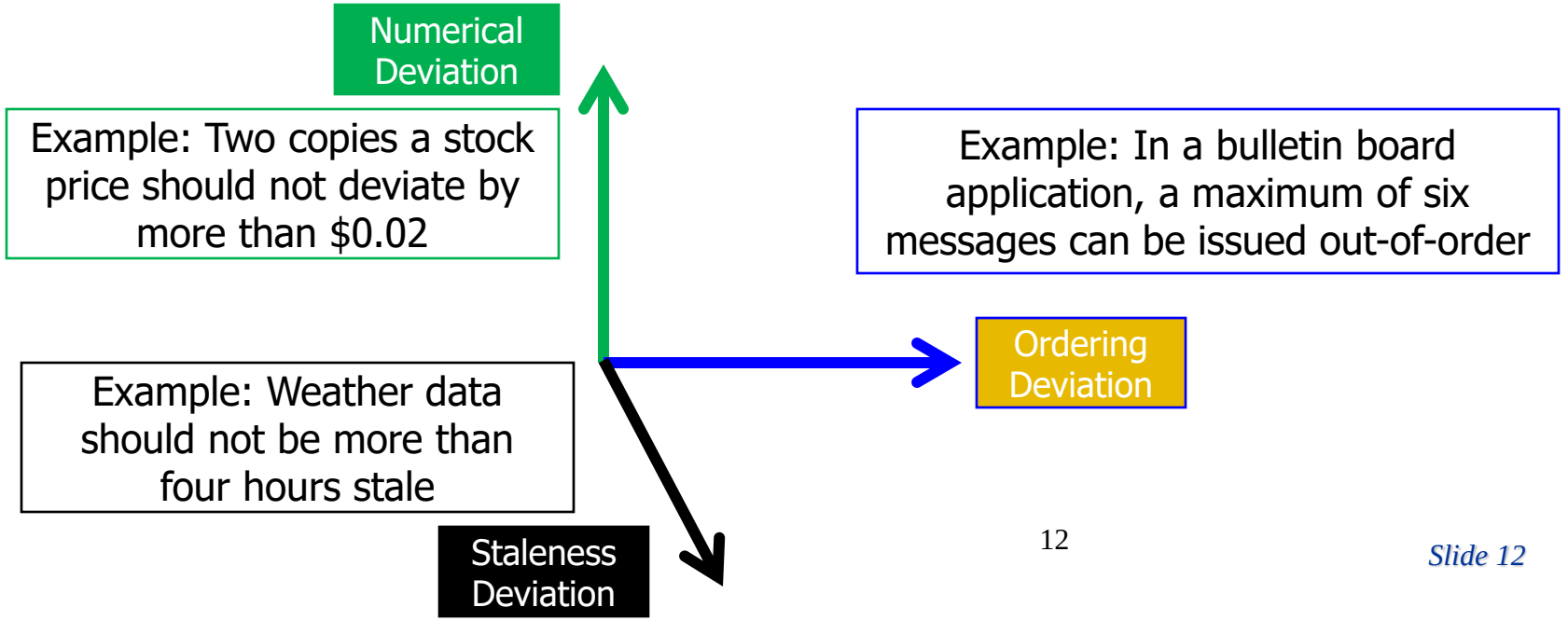
Data-centric Consistency Models

- Data-centric Consistency Models describe how the replicated data is kept consistent, and what the process can expect

- Under Data-centric Consistency Models, we study two types of models:
  - Consistency Specification Models:
    - These models enable specifying the consistency levels that are tolerable to the application
  - Models for Consistent Ordering of Operations:
    - These models specify the order in which the data updates are propagated to different replicas

Continuous Consistency Ranges

- Level of consistency is defined over three independent axes:
  - Numerical Deviation: Deviation in the numerical values between replicas
  - Order Deviation: Deviation with respect to the ordering of update operations
  - Staleness Deviation: Deviation in the staleness between replicas



Types of Ordering

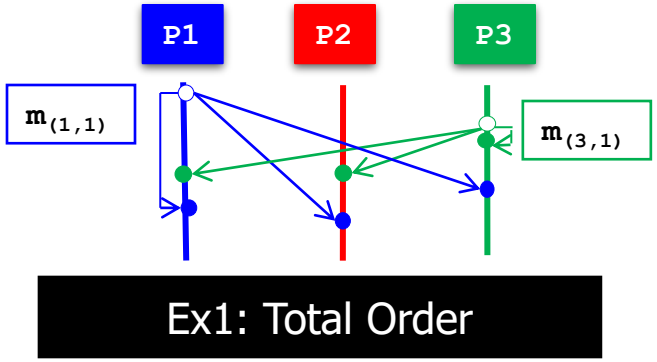
- We will study three types of ordering of messages that meet the needs of different applications:
  1. Total Ordering
  2. Sequential Ordering
     i. Sequential Consistency Model
  3. Causal Ordering
     i. Causal Consistency Model

Types of Ordering

1. Total Ordering
2. Sequential Ordering
3. Causal Ordering

Total Ordering

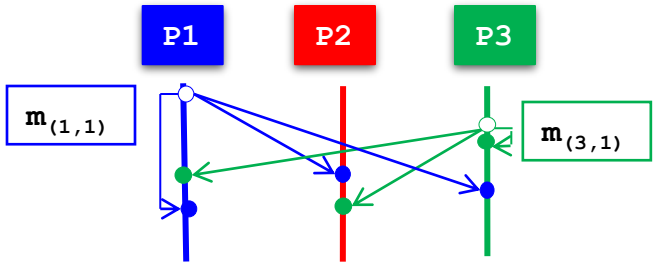
- Total Order
  - If process P_i sends a message m_i and P_j sends m_j, and if one correct process delivers m_i before m_j then every correct process delivers m_i before m_j



Ex1: Total Order

- Messages can contain replica updates, such as passing the read or write operation that needs to be performed at each replica
  - In the example Ex1, if P_1 issues the operation $m_{(1,1)}$: `x=x+1;` and
  - If P_3 issues $m_{(3,1)}$: `print(x);`
  - Then, at all replicas P_1, P_2, P_3 the following order of operations are executed

```
print(x);
x=x+1;
```



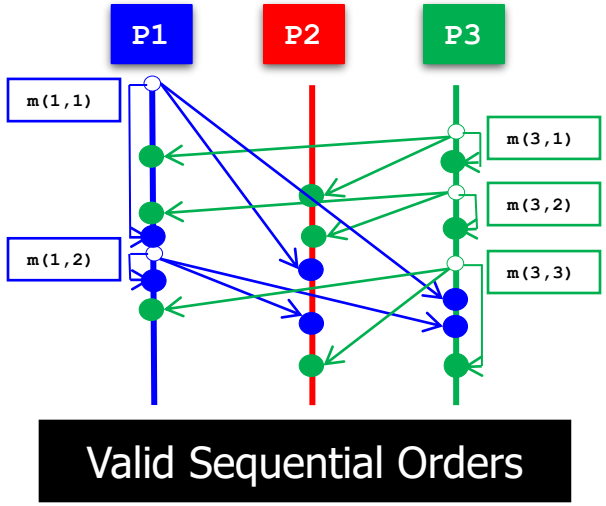
Ex2: Not in Total Order

Types of Ordering

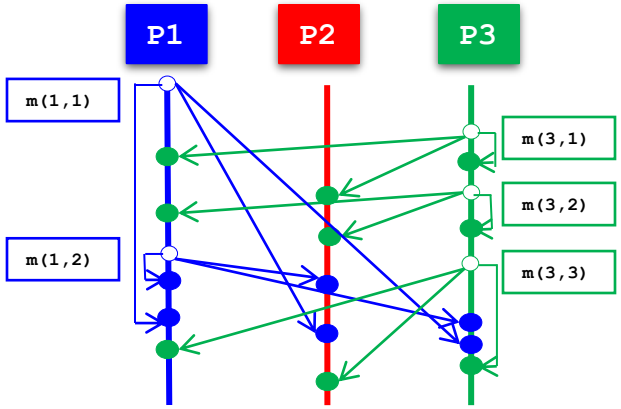
1. Total Ordering
2. Sequential Ordering
3. Causal Ordering

Sequential Ordering

- If a process **Pi** sends a sequence of messages $m_{(i,1)}, \ldots, m_{(i,ni)}$, and
- Process **Pj** sends a sequence of messages $m_{(j,1)}, \ldots, m_{(j,nj)}$,
- Then, :
  - At any process, the set of messages received are in some sequential order
  - Messages from each individual process appear in this sequence in the order sent by the sender
    - At every process, $m_{i,1}$ should be delivered before $m_{i,2}$, which is delivered before $m_{i,3}$ and so on...
    - At every process, $m_{j,1}$ should be delivered before $m_{j,2}$, which is delivered before $m_{j,3}$ and so on...



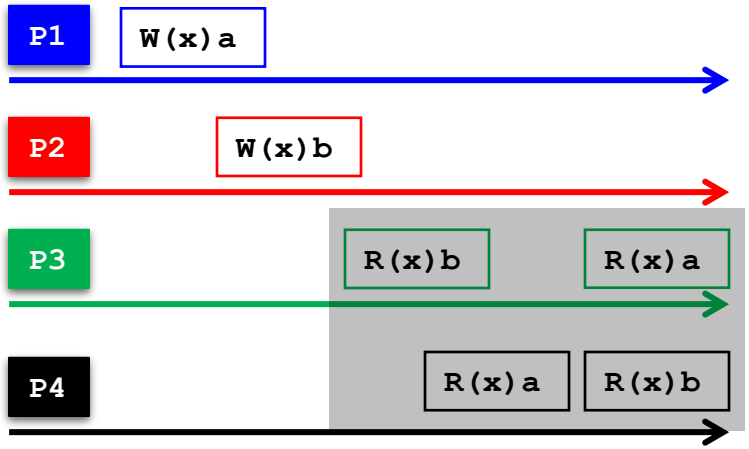
Valid Sequential Orders



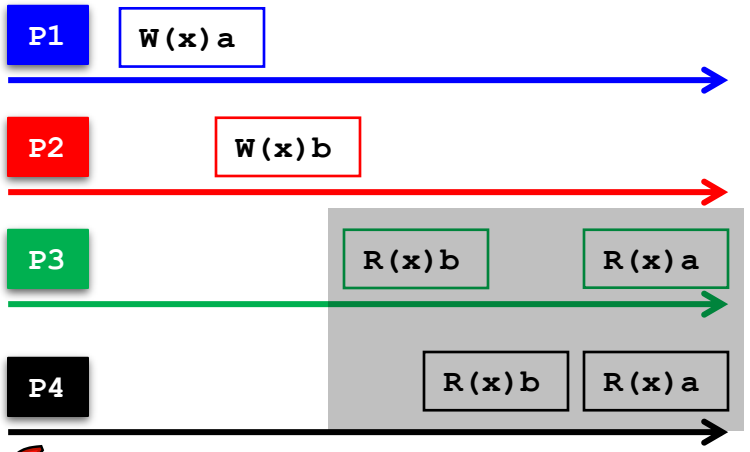
Invalid Sequential Orders, but Valid Total Order

Sequential Consistency Model

- Sequential Consistency Model enforces that all the update operations are executed at the replicas in a sequential order
- Consider a data-store with variable **x** (Initialized to **NULL**)
  - In the two data-stores below, identify the sequentially consistent data-store



Results while operating on DATA-STORE-1

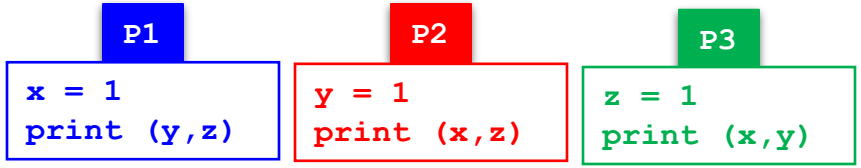


Results while operating on DATA-STORE-2

P1 =Process P1 → =Timeline at P1 R(x)b =Read variable x; Result is b W(x)b = Write variable x; Result is b

Sequential Consistency (cont'd)

- Consider three processes P_1, P_2 and P_3 executing multiple instructions on three shared variables **x**, **y** and **z**
  - Assume that **x**, **y** and **z** are set to zero at start



- There are many valid sequences in which operations can be executed at the replica respecting sequential consistency
  - Identify the output

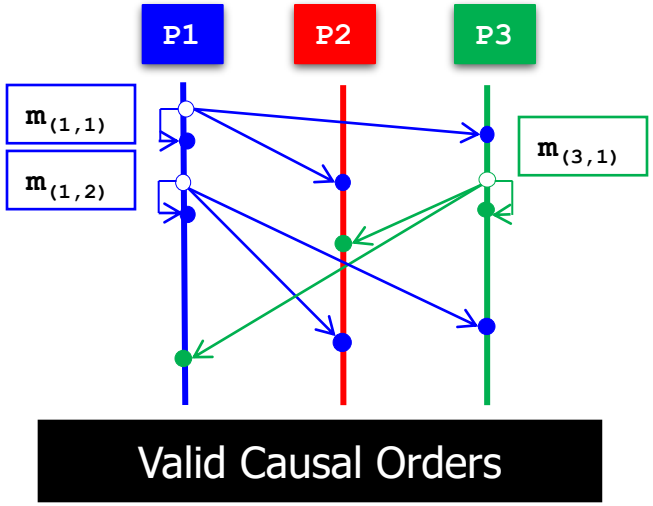
	x = 1 print (y,z) y = 1 print (x,z) z = 1 print (x,y)	x = 1 y = 1 print (x,z) print (y,z) z = 1 print (x,y)	z = 1 print (x,y) print (x,z) y = 1 x = 1 print (y,z)	y = 1 z = 1 print (x,y) print (x,z) x = 1 print (y,z)
Output	001011	101011	000111	010111

Causal Ordering

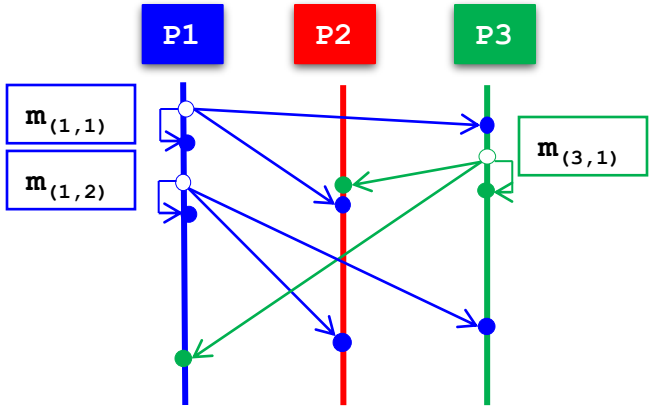
- Causal Order
  - If process P_i sends a message m_i and P_j sends m_j, and if $m_i \rightarrow m_j$ (operator '→' is Lamport's **happened-before** relation) then any correct process that delivers m_j will deliver m_i before m_j

- In the example, $m_{(1,1)}$ and $m_{(3,1)}$ are in Causal Order

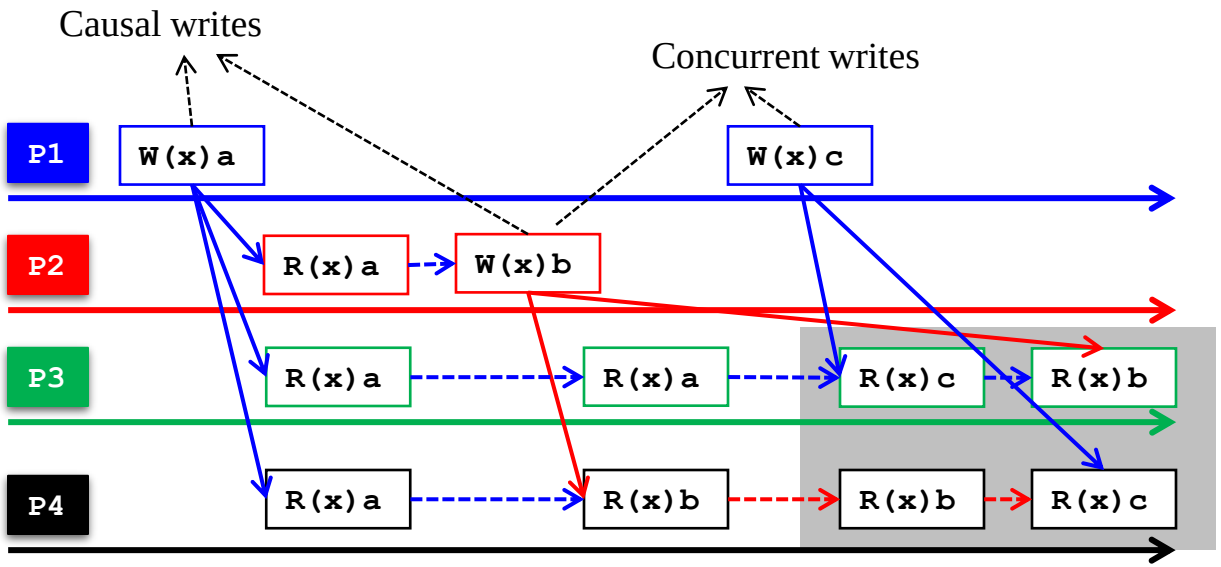
- Drawback:
  - The **happened-before** relation between m_i and m_j should be induced before communication



Valid Causal Orders



Invalid Causal Order



A Causally Consistent Data-Store

But not a Sequentially Consistent Data-Store

P1 =Process P1

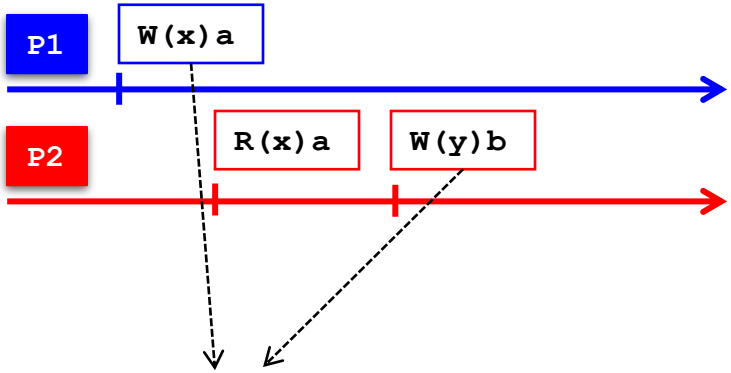
=Timeline at P1

R(x)b =Read variable x; Result is b

W(x)b = Write variable x; Result is b

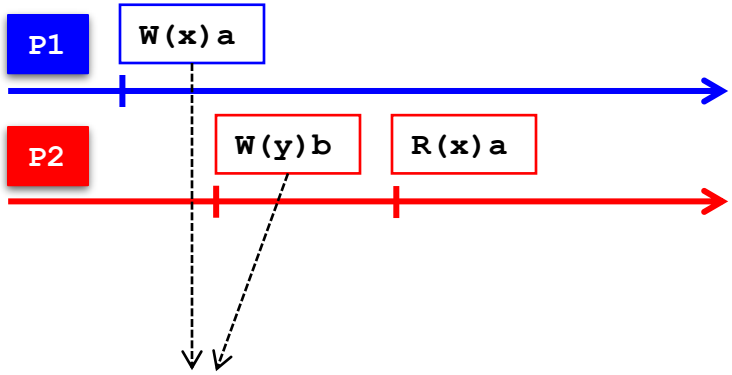
Causal vs. Concurrent events

- Consider an interaction between processes P_1 and P_2 operating on replicated data **x** and **y**



Events are causally related
Events are not concurrent

- Computation of **y** at P_2 may have depended on value of **x** written by P_1



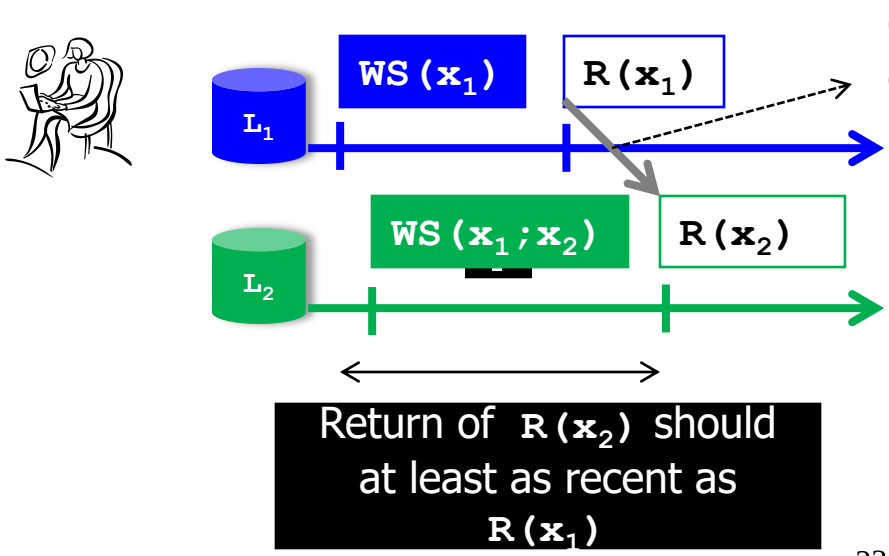
Events are not causally related
Events are concurrent

- Computation of **y** at P_2 does not depend on value of **x** written by P_1

P1 =Process P1 → =Timeline at P1 R(x)b =Read variable x; Result is b W(x)b = Write variable x; Result is b

Monotonic Reads

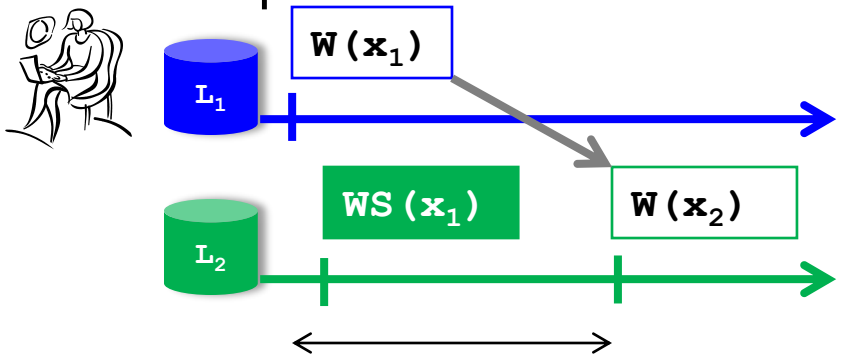
- The model provides guarantees on successive reads
- If a client process reads the value of data item **x**, then any successive read operation by that process should return the same or a more recent value for **x**



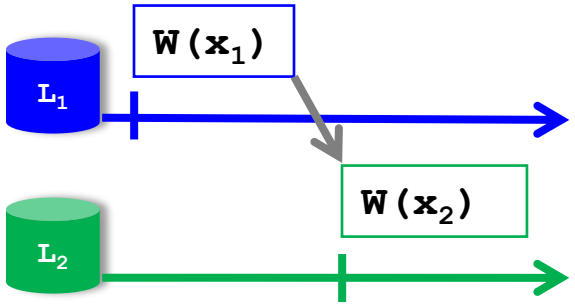
Order in which client process carries out the operations

Monotonic Writes

- This consistency model assures that writes are monotonic
- A write operation by a client process on a data item **x** is completed before any successive write operation on **x** by the same process
  - A new write on a replica should wait for all old writes on any replica



W(x2) operation should be performed only after the result of **W(x1)** has been updated at **L2**

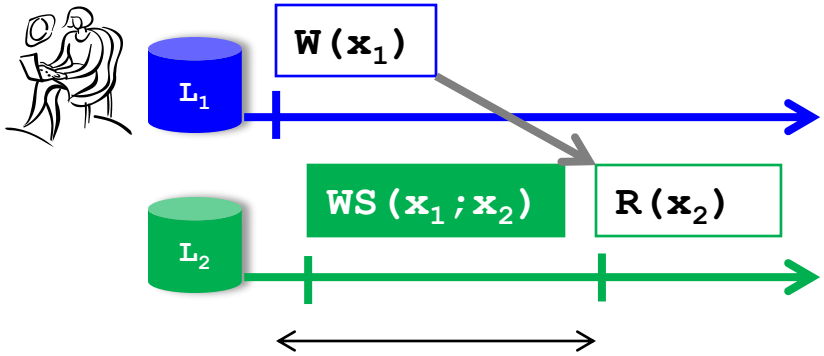


The data-store does not provide monotonic write consistency

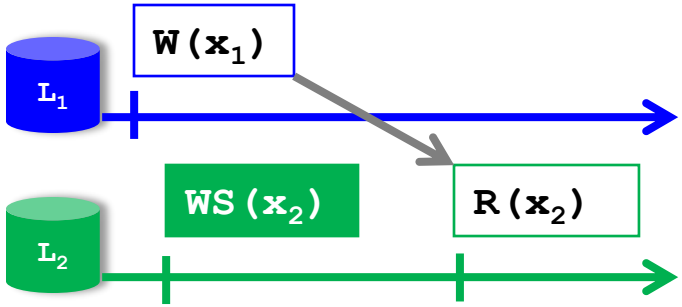
Read Your Writes

- The effect of a write operation on a data item **x** by a process will always be seen by a successive read operation on **x** by the same process

- Example scenario:
  - In systems where password is stored in a replicated data-base, the password change should be seen immediately



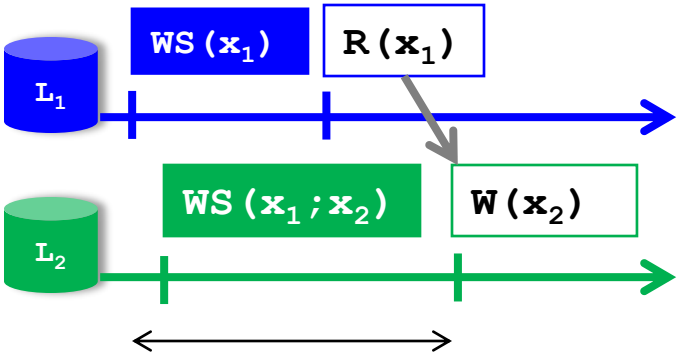
R(x2) operation should be performed only after the updating the Write Set **WS(x1)** at **L2**

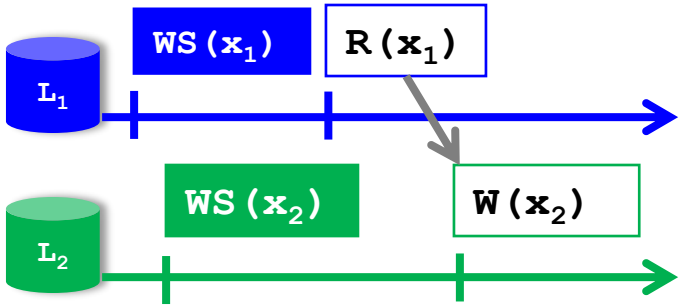


A data-store that does not provide *Read Your Write* consistency

Write Follow Reads

- A write operation by a process on a data item **x** following a previous read operation on **x** by the same process is guaranteed to take place on the same or a more recent value of **x** that was read
- Example scenario:
  - Users of a newsgroup should post their comments only after they have read all previous comments





W(x_2) operation should be performed only after the all previous writes have been seen

A data-store that does not guarantee Write Follow Read Consistency Model