Distributed Systems: Consistency and Replication

Gian Pietro Picco
Dipartimento di Ingegneria e Scienza dell'Informazione
University of Trento, Italy
gianpietro.picco@unitn.it
http://disi.unitn.it/~picco

Why Replicate Data?

- Improve performance
  - Local access is faster than remote access
  - Sharing of workload
- Increase availability
  - Data may become unavailable due to failure
  - Data may be available only intermittently in a mobile setting
    - Replicas provide support for disconnected operations
- Achieve fault tolerance
  - Related to availability
  - It becomes possible to deal with incorrect behaviors through redundancy

A Note About Availability

Definition (Availability)

The probability that a system will provide its required service, or the ratio of the total time a system is capable of being used during a given interval to the length of the interval:

$$A = \frac{E[uptime]}{E[uptime + downtime]}$$

- Single server
- On average, crashes once per week, i.e., the mean time between failures is MTBF=10.080 min
- On average, 2 min to reboot

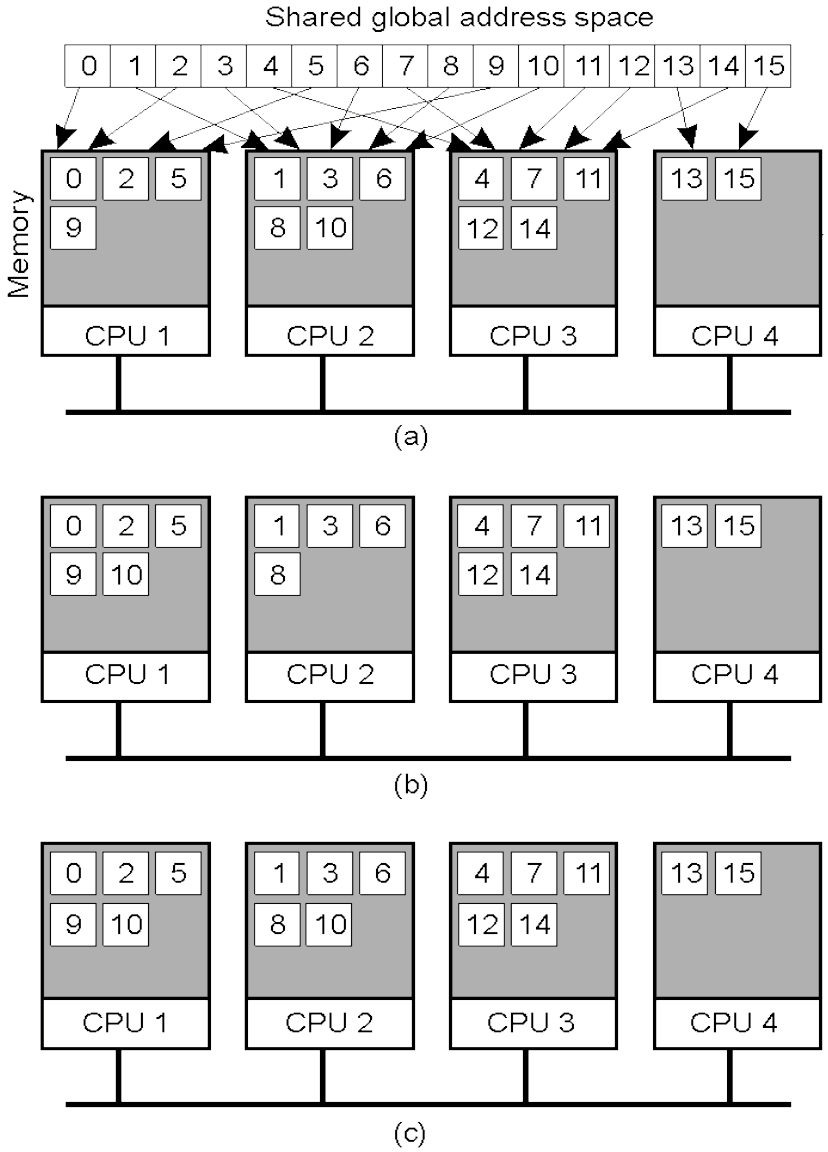
$$A = \frac{10080}{10080 + 2} = 0.\underline{9998}$$

- 10 servers
- one replica is sufficient to provide the service
- MTBF and reboot time as before

$$p_f = \frac{2}{10082}$$

$$A = 1 - (p_f)^{10} = 1 - 10^{-38}$$

An Example: Distributed Shared Memory



- Devised for small-scale, LAN scenarios
- Single virtual address space, spanning multiple computers
  - Powerful programming abstraction, minimizes differences with conventional programming
- Efficiency problems
  - "False sharing": pages may be transferred back and forth
- Replication improves performance (and scalability), by enabling local access
  - However, it introduces the problem of how to keep replicas consistent

Other Examples

- Web caches
  - Browsers store locally a copy of the page, to improve access time
  - Caches may be present also in the network
    - e.g., on Web proxies
  - Assumption: pages are changed infrequently
- Distributed file systems
  - File replication (e.g., in Coda) allows for faster access and disconnected operations
  - User files are reconciled against servers upon reconnection
  - Assumption: conflicts (files modified by more than a user) are infrequent
- Domain Name Service
- …
- ***Massively used in today's data centers!***

Challenges

- Main problem: ***consistency***
  - Changing a replica demands changes to all the others
  - We must ensure that accessing a replica has the same effect as accessing the original
  - Goal: provide consistency with limited communication overhead
- Conflicting operations are key
  - two concurrent reads are not a problem
  - two concurrent writes, or concurrent reads and writes are problem
  - … remember transactions?
- Scalability vs. performance
  - Replication may actually degrade performance!
- Different consistency requirements depending on the application scenario
  - Data-centric vs. client-centric

The Consistency Problem

The goal

We generally need to ensure that all conflicting operations are done in the same order everywhere

The problem

Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability

The solution

Weaken consistency requirements so that hopefully global synchronization can be avoided

An Example

Example (Flight reservation database)

- At 9.36, all seats of flight 48 are booked
- At 9.37, Jane cancel its reservation on flight 48
- At 9.38, Michael tries to reserve a seat on flight 48
  - the answer is fully booked
- At 9.39, George tries to reserve a seat on flight 48
  - the seat is granted

- What do you think may be the problem?
- How do we deal with these situations?

An Authoritative Point of View

"Whether or not inconsistencies are acceptable depends on the client application. In all cases ***the developer must be aware*** *that consistency guarantees are provided by the storage systems and must be taken into account when developing applications."*

Werner Vogels

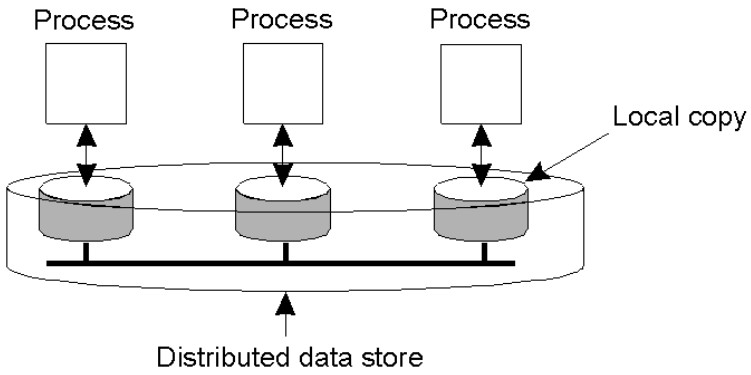
Amazon's vice-president and Chief Scientific Officer

Consistency Models

Definition (Consistency model)

A contract between a distributed data store and a set of processes, which specifies what the results of read/write operations are in the presence of concurrency

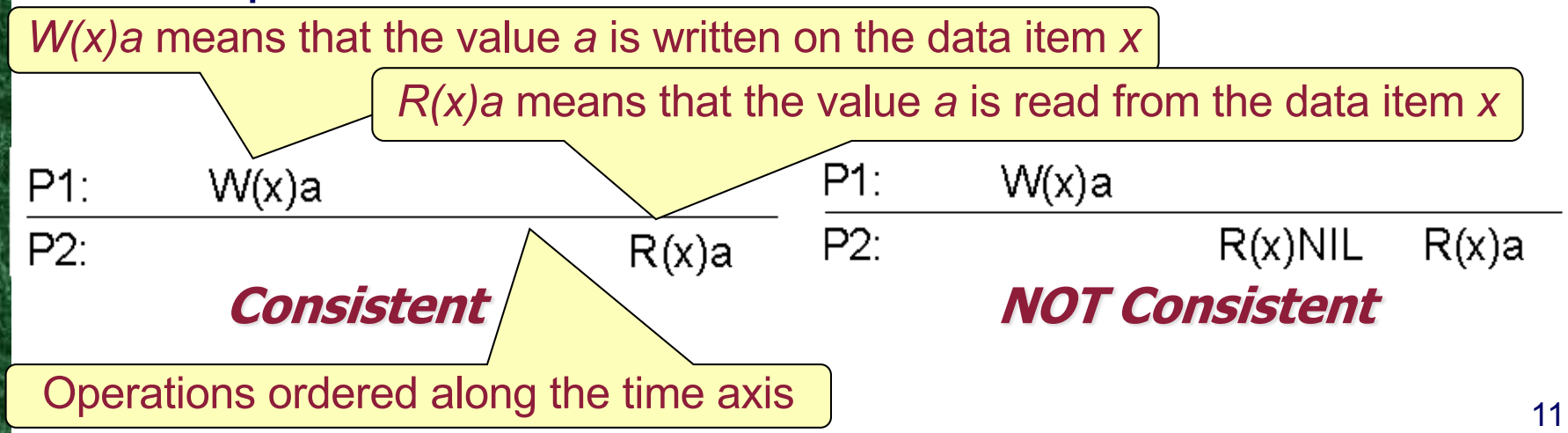
- Ideally, a read should show the result of the last write, as in the non-distributed case
- This is in general not possible, hence the need for a (weaker) consistency model
  - Tradeoffs: guarantees & ease of use vs. performance
- Focus on a (distributed) ***data store***
  - E.g., shared memory, filesystem, database, …
  - Each process has access to a local copy of the entire store



Strict Consistency

> ***"Any read on data item x returns the value of the most recent write on x"***

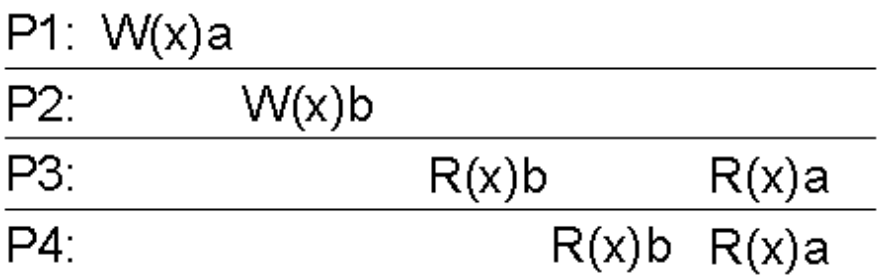
- All writes are instantaneously visible, global order is maintained
- "Most recent" is ambiguous without global time
- Even with global time, in some cases it can be impossible to guarantee
- In practice, possible only within a uniprocessor machine
- Example:



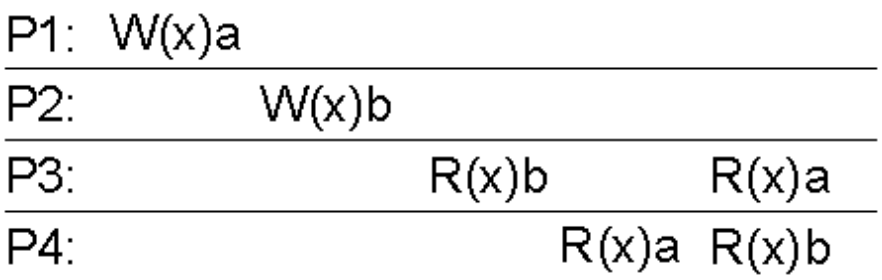
Sequential Consistency

> ***"The result is the same as if the operations by all processes were executed in some sequential order, and the operations by each process appear in this sequence in the order specified by its program"***

- In other words:
  - operations within a process may not be re-ordered
  - all processes see the same interleaving
- Does not rely on physical time



Consistent

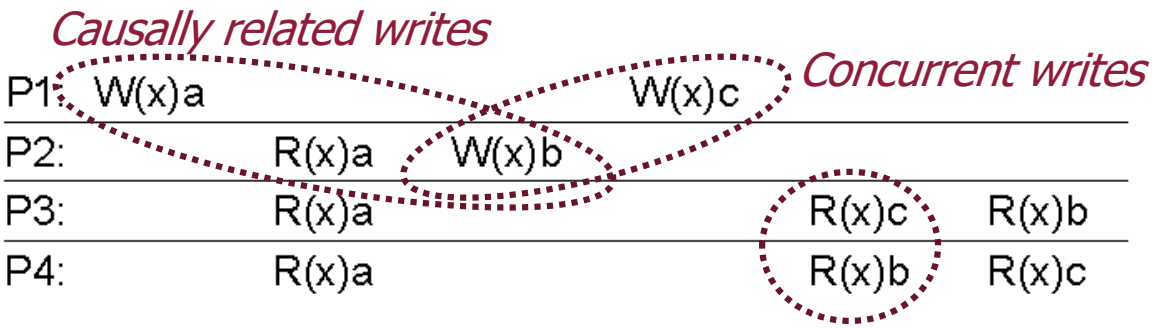


NOT Consistent

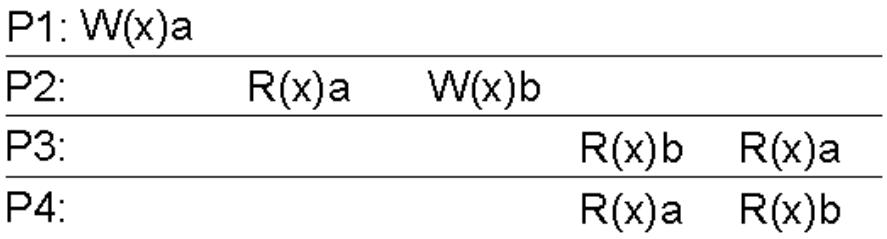
Causal Consistency

"Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in any order at different machines."

- Similar to the notion of causality in vector clocks
  - Here, events are read and write operations on the store
  - Reads are causally related to writes before and/or after them



Consistent



NOT Consistent

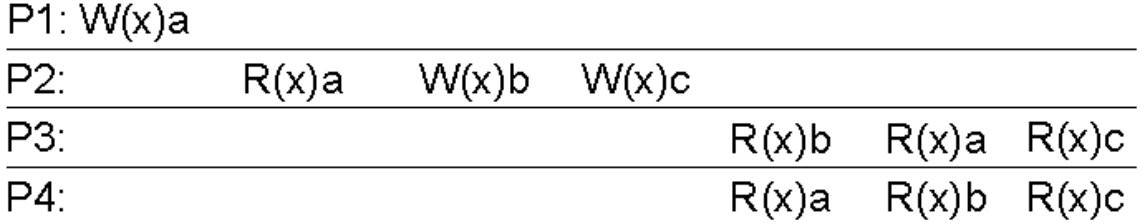
(it becomes so if P2's R(x) is removed)

FIFO Consistency

"Writes done by a single process are seen by all others in the order in which they were issued; writes from different processes may be seen in any order at different machines."

- In other words, causality across processes is dropped
- Also called PRAM consistency (Pipelined RAM)
  - If writes are put onto a pipeline for completion, a process can fill the pipeline with writes, not waiting for early ones to complete
- Easy to implement (sequence numbers on writes)
- Maybe counterintuitive:

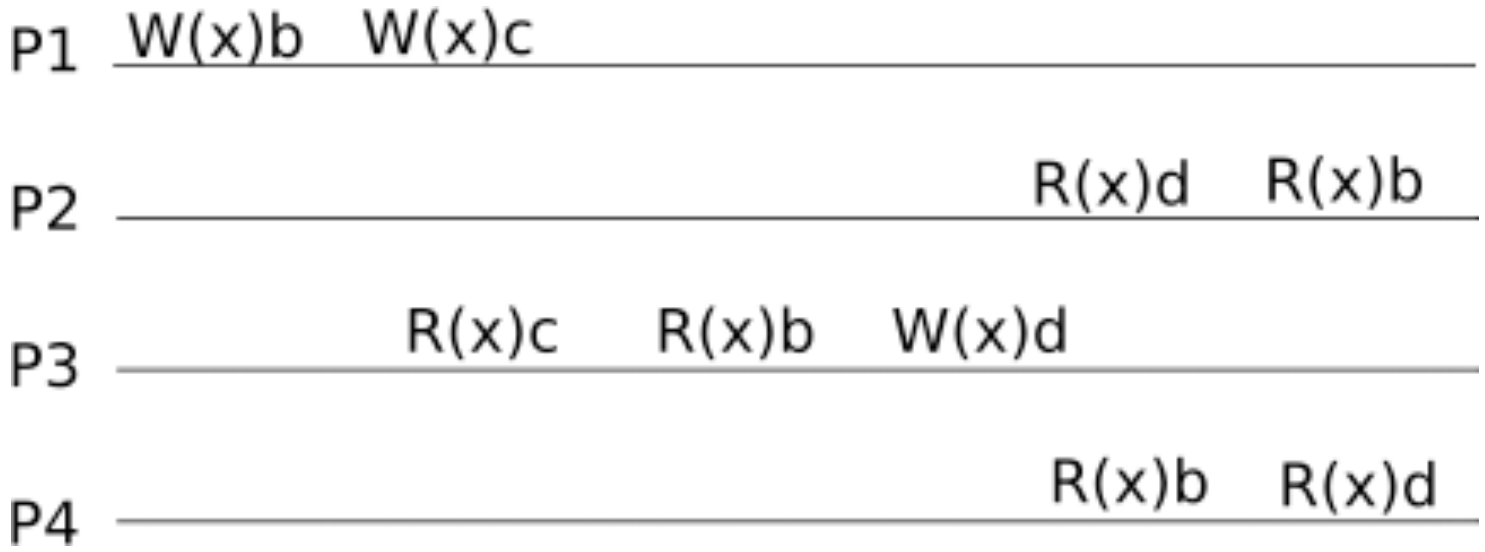
Process P1	**Process P2**
x = 1;	y = 1;
if (y == 0) kill (P2);	if (x == 0) kill (P1);



Consistent

Exercise

- For each of the consistency criteria (FIFO, causal, sequential) say
  - whether the schedule below satisfies the criteria
  - If not, show the minimum number of modifications (no removal, only reordering) of the read operations to satisfy the criteria



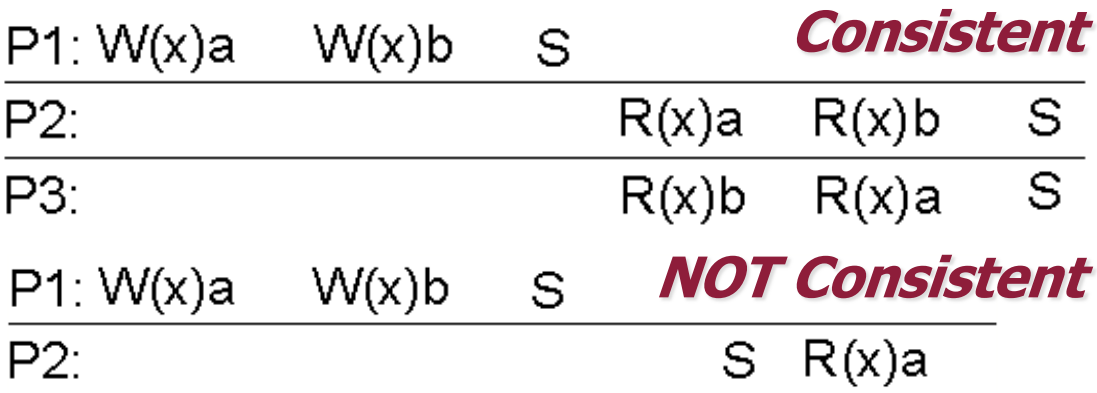
Consistency Models and Synchronization

- FIFO consistency still requires all writes to be visible to all processes, even those that do not care
- Moreover, not all writes need be seen by all processes
  - e.g., those within a transaction/critical section
- Some consistency models introduce the notion of ***synchronization variables***
  - Writes become visible only when processes explicitly request so through the variable
  - Appropriate constructs are provided, e.g., **`synchronize(S)`**
- It is up to the programmer to force consistency when is really needed, typically:
  - At the end of a critical section, to distribute writes
  - At the beginning of a “reading session” when writes need to become visible

Weak Consistency

1. ***Access to synchronization variables is sequentially consistent;***
2. ***No operation on a synchronization variable is allowed until all previous writes have completed everywhere;***
3. ***No read or write to data are allowed until all previous operations to synchronization variables have completed.***

- It enforces consistency on a ***group*** of operations
- It limits only the ***time*** when consistency holds, rather than the form of consistency
- Data may be inconsistent in the meanwhile



We assume only the operation immediately before/after the S benefits from synchronization

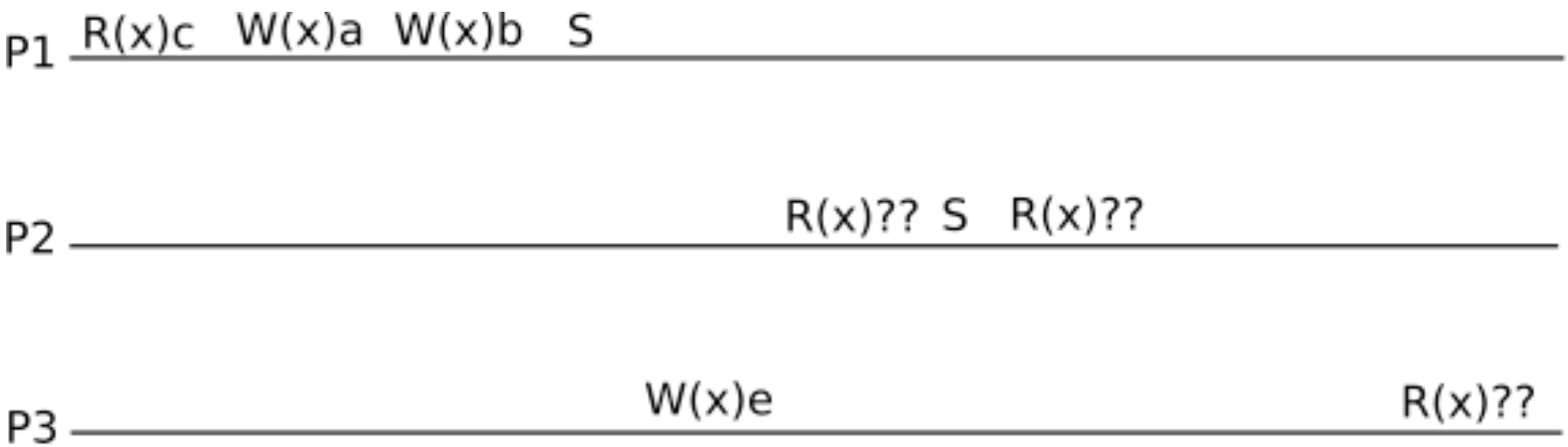
More complex variants differentiate entry vs. exit from critical section

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters
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
Weak	Shared data can be counted on to be consistent only after a synchronization is done

Exercise

- Complete the schedule below with the appropriate values, so that it is compatible with the weak consistency criterion



Exercise

- Consider the following schedule. W(x)a means that the value a is written into variable x, and similarly for a read R(X)a. S1–S4 are the calls on the synchronization variable S, for weak consistency.

P1	W(x)a											W(x)d	S3	
P2		W(x)b							R(x)b	R(x)a				R(x)?
P3				R(x)?	R(x)?	S1	W(x)c	S2						
P4				R(x)a	R(x)b								S4	R(x)?

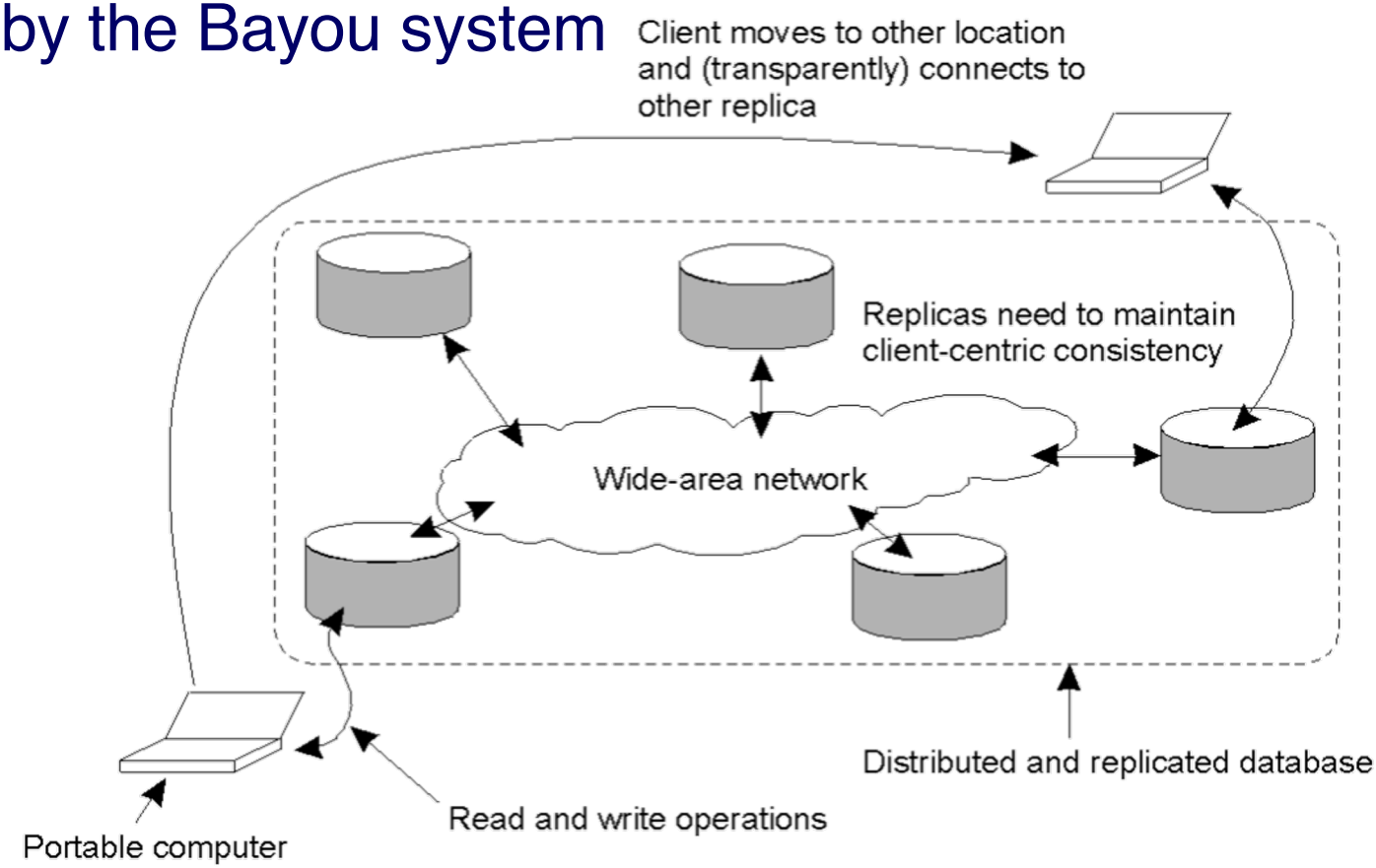
- For each operation with a question mark (?) on the value, specify what are the values allowed for the schedule to comply with the weak consistency model.
- The weak consistency model states that accesses to the synchronization variable must be sequentially consistent. Explain why with reference to the figure. For instance, say how your answer to the previous question would change in the case where the global order of accesses is
- i. S1, S2, S3, S4
  ii. S1, S2, S4, S3

Eventual Consistency

- The models considered so far are ***data-centric***
  - Provide a system-wide consistent data view in the face of simultaneous, concurrent updates
- However, there are situations where there are:
  - no simultaneous updates (or can be easily resolved)
  - mostly reads
- Examples: Web caches, DNS, USENET news
- In these systems, ***eventual consistency*** is often sufficient
  - Updates are guaranteed to eventually propagate to all replicas
- However, eventual consistency is easy to implement only if reads involve always the same replica …

Client-Centric Consistency

- The problem can be tackled by introducing ***client-centric consistency*** models
  - Provide guarantees about accesses to the data store ***from the perspective of a single client***
- Pioneered by the Bayou system



Monotonic Reads

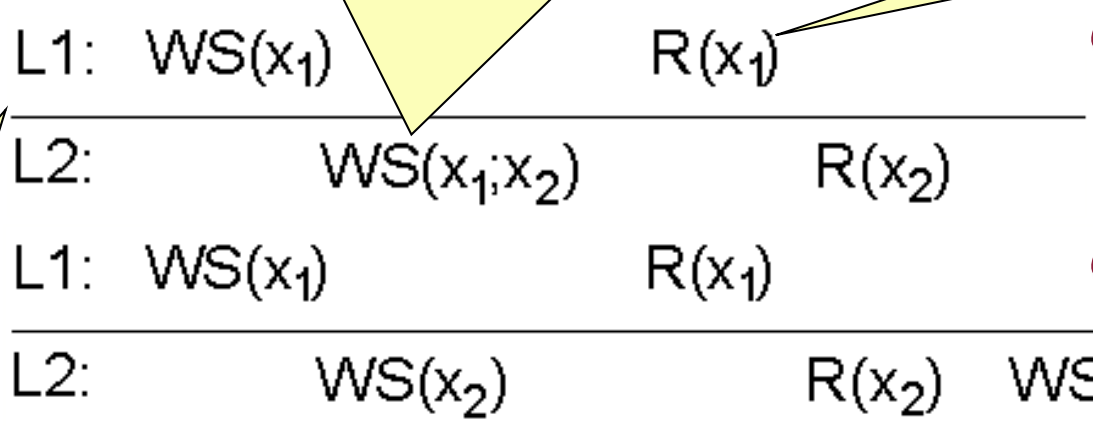
"If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value."

- Once a process reads a value from a replica, it will never see an older value from a read at a different replica
- Example: e-mail mirrors

The set of writes known at a data store contain the update of x at L1 and the update of x at L2, in this order

Read on x_1, the value of x at data store L1

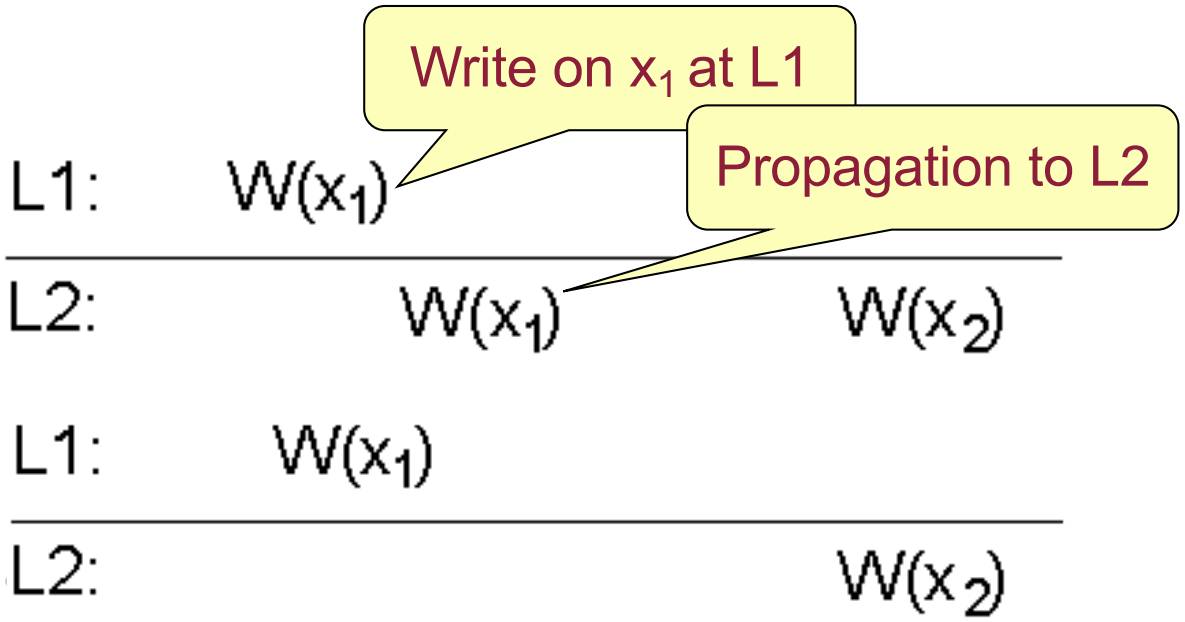
L1 and L2 are local copies of the data store, accessed by the same process



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."***

- Similar to FIFO consistency, although this time for a single process
- A weaker notion where ordering does not matter is possible if writes are commutative
- x can be a large part of the data store (e.g., a code library)



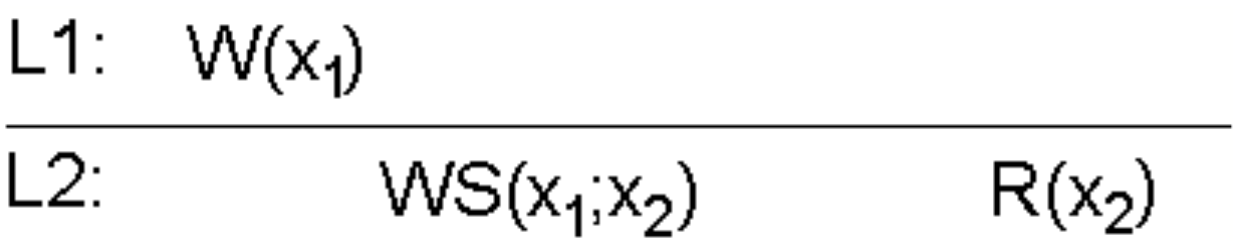
Consistent

NOT Consistent

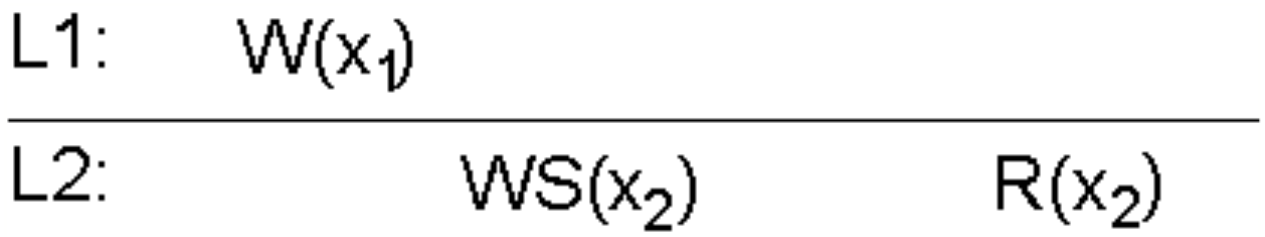
Read Your Writes

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

- Examples: updating a Web page, or a password



Consistent

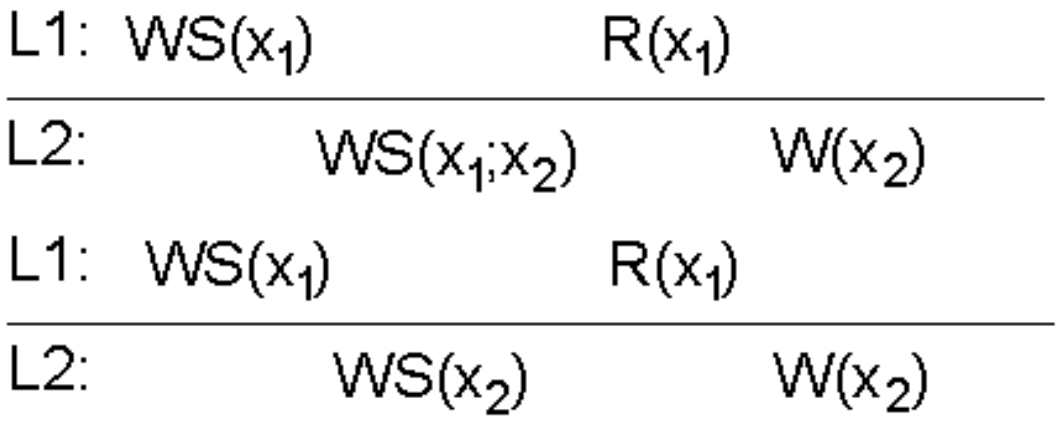


NOT Consistent

Writes 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 more recent value of x that was read."***

- Example: guarantee that users of a newsgroup see the posting of a reply only after seeing the original article



Consistent

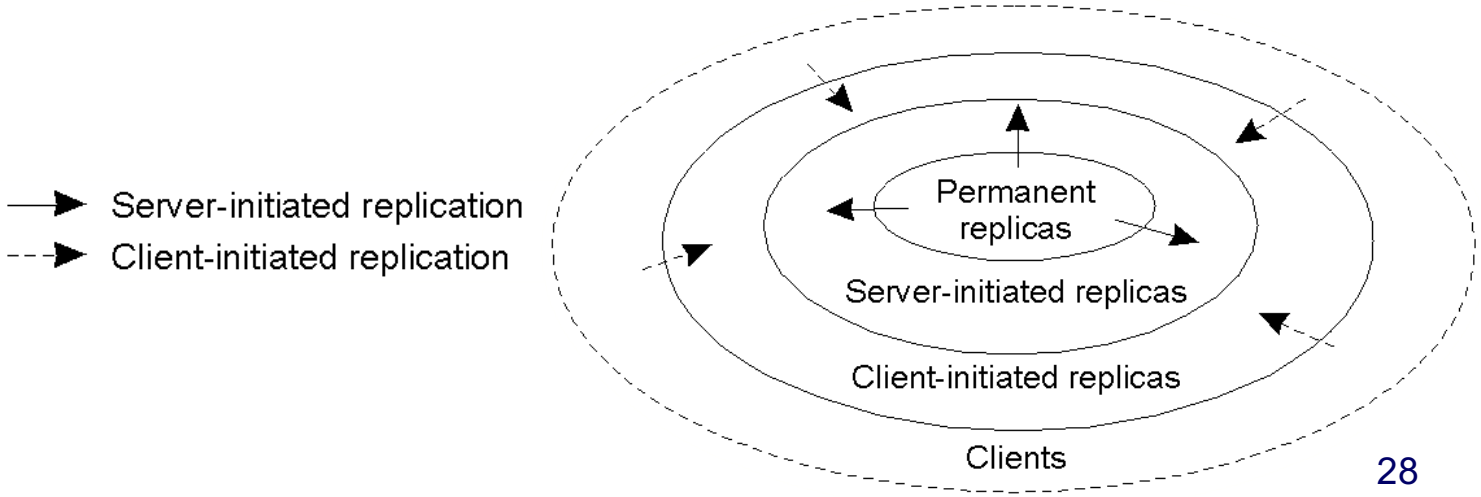
NOT Consistent

Implementing Replication

- How to distribute updates, which in turn involves
  - How/where to place replicas
  - How to propagate updates between them
- How to keep replicas consistent

Replica Placement

- Permanent replicas
  - Statically configured, e.g., Web mirrors
- Server-initiated replicas
  - Created dynamically, e.g., to cope with access load
  - Goal: move data closer to clients
  - Based on access profiling, often requires topological knowledge
- Client-initiated replicas
  - Rely on a client cache, where items are stored temporarily
  - Depending on the application scenario, caches can be shared among clients for enhanced performance
  - In principle, no involvement of the data store (but often useful)



Update Propagation

- What to propagate?
  - Perform the update and propagate only a notification
    - Used in conjunction with invalidation protocols: avoids unnecessarily propagating subsequent writes
    - Small communication overhead
    - Works best if #reads << #writes
  - Transfer the modified data to all copies
    - Works best is #reads >> #writes
  - Propagate information to enable the update operation to occur at the other copies
    - Also called ***active replication***
    - Very small communication overhead, but may require unnecessary processing power if the update operation is complex

Update Propagation

- How to propagate?
  - ***Push-based approach (or server-based)***
    - the update is propagated to all replicas, regardless of their needs
    - typically used to preserve a high degree of consistency
    - may benefit of multicast facilities
    - more convenient if #reads >> #writes
  - ***Pull-based approach (or client-based)***
    - an update is fetched on demand when needed
    - typically used to manage client caches, e.g., for the Web
    - usually done through unicast communication
    - more convenient if #reads << #writes
  - ***Leases*** can be used to switch between the two
    - push is performed only until timeout expiration: pull afterwards

Issue	**Push-based**	**Pull-based**
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Comparison assuming one server and multiple clients, each with their own cache