Consistency Models

Why replicate?

- Data replication: common technique in distributed systems
- Reliability
  - If one replica is unavailable or crashes, use another
  - Protect against corrupted data
- Performance
  - Scale with size of the distributed system (replicated web servers)
  - Scale in geographically distributed systems (web proxies)

- Key issue: need to maintain *consistency* of replicated data
  - If one copy is modified, others become inconsistent

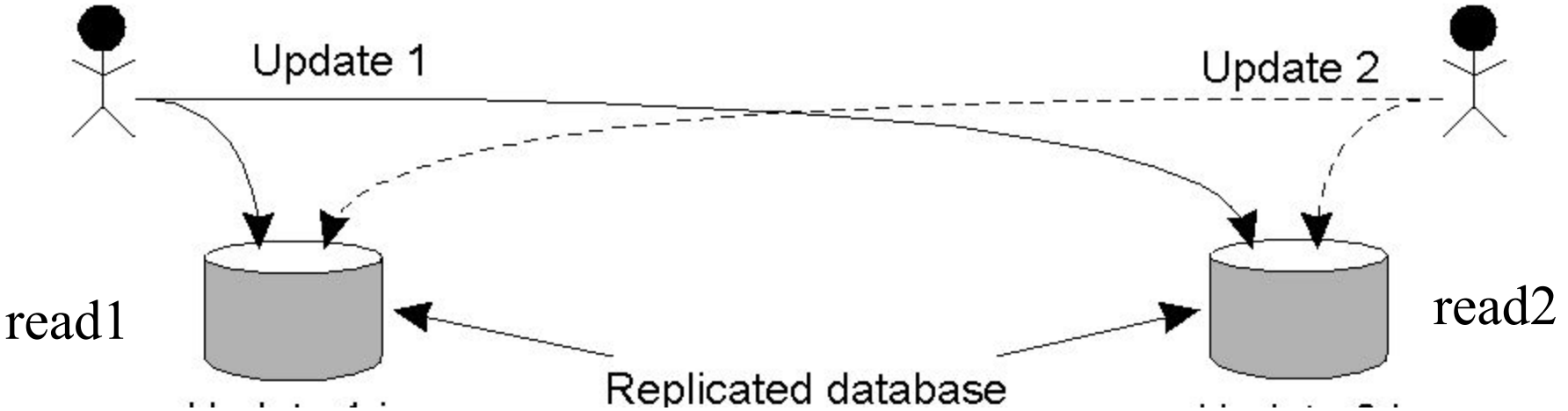
Cost of Replication

- Replicas must be kept consistent

Dilemma:

1. Replicate data for better performance
2. Modification on one copy triggers modifications on all other replicas
3. Propagating each modification to each replica can degrade performance

Consistency Models (cont)



- A process performs a read operation on a data item, expects the operation to return a value that shows the result of the last write operation on that data
- No global clock $\Rightarrow$ difficult to define the last write operation
- Consistency models provide other definitions
- Different consistency models have different restrictions on the values that a read operation can return

Consistency models

- Consistency requirements vary from application to application

- A *Consistency Model* refers to the degree of consistency that has to be maintained by the shared memory for the application to run correctly

  - Set of rules – an agreement between data store and the processes
  - Helps in improving the performance
  - Stronger consistency models support weaker consistency
  - Consistency in terms of read and write operations on shared data, called the data store

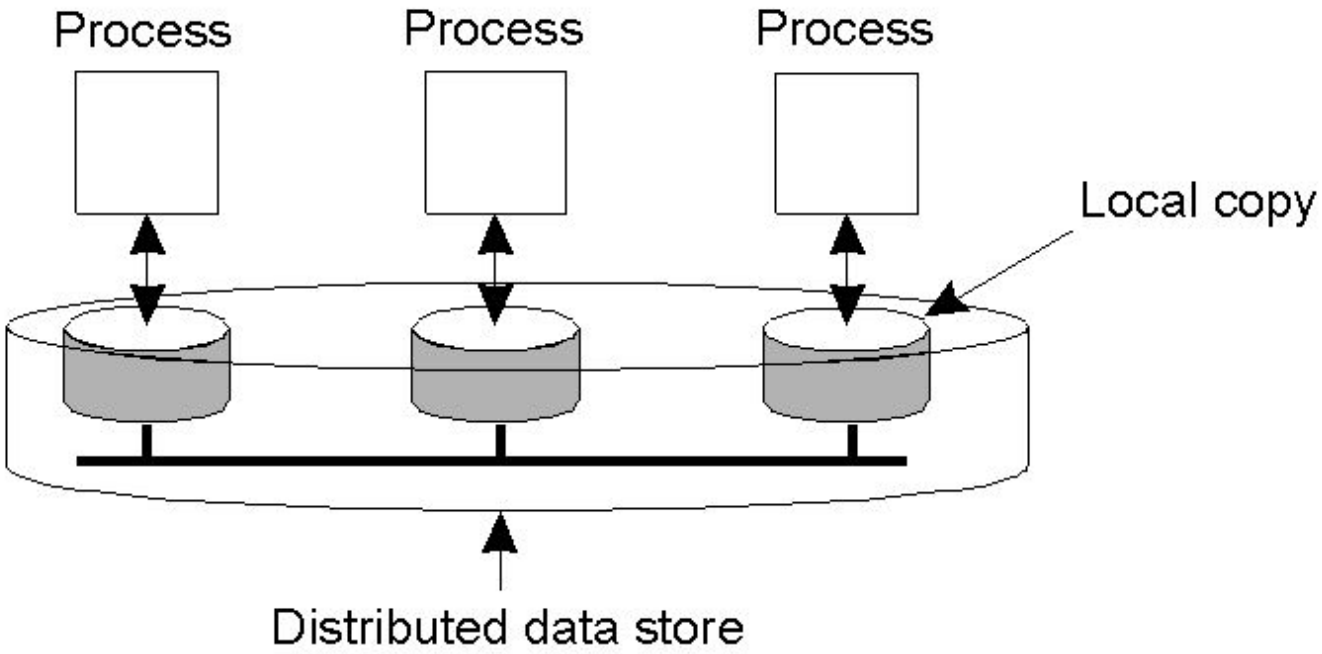
Consistency Models

- A *Consistency Model* is a contract between the software and the memory
  - it states that the memory will work correctly but only if the software obeys certain rules
- The issue is how we can state rules that are not too restrictive but allow fast execution in most common cases
- These models represent a more general view of sharing data than what we have seen so far!

Conventions we will use:

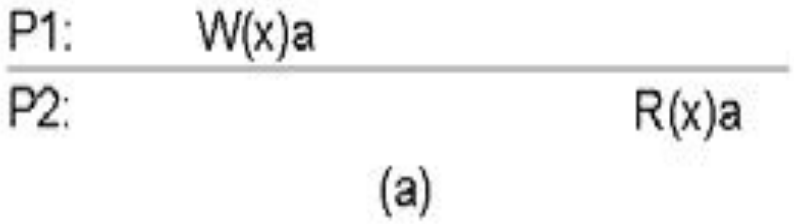
- ***W(x)a* means "a *write* to *x* with value *a*"**
- ***R(y)b* means "a *read* from *y* that returned value *b*"**

Data-Centric Consistency Models

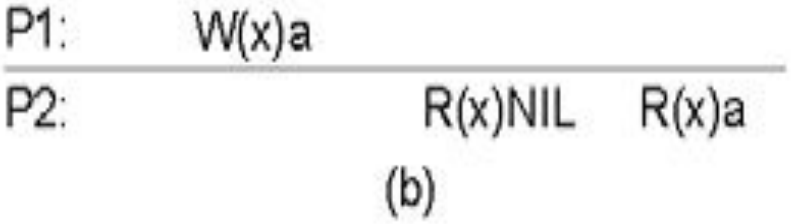


- Consistency model (*consistency semantics or constraints*)
  - Contract between processes and the data store
    - If processes obey certain rules, data store will work correctly
  - All models attempt to return the results of the last write for a read operation
    - Differ in how “last” write is determined/defined

Strict Consistency



(a)

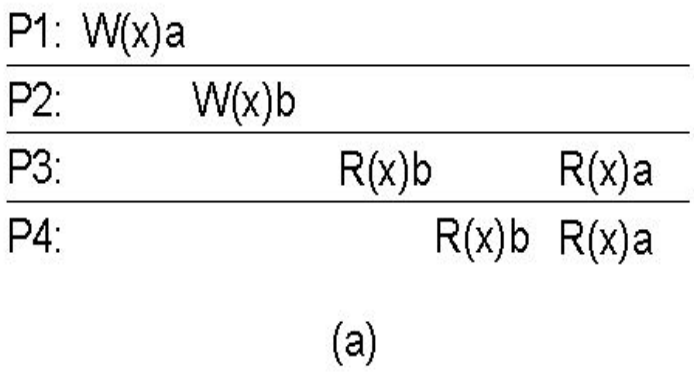


(b)

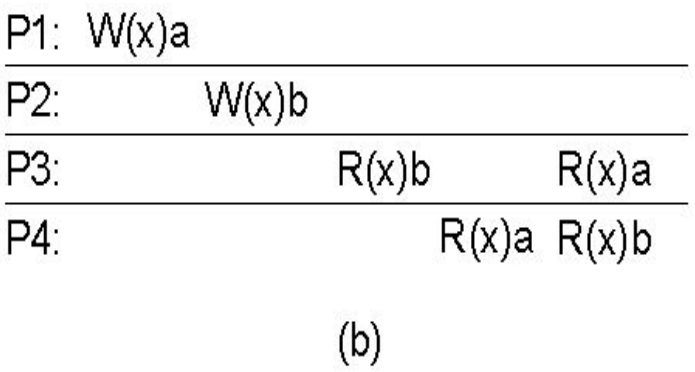
- Any read on a data item x always returns the result of the most recent write on x
  - **Implicitly assumes the presence of a global clock**
  - **A write is immediately visible to all processes**
    - **Difficult to achieve in real systems as network delays can be variable**

Sequential Consistency

- Sequential consistency: weaker than strict consistency
  - Assumes all operations are executed in some sequential order and each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order



(a)



(b)

Sequential Consistency (Lamport, 1979): The result of any execution is the same as if the read and write 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 :
This means that all processes see the same interleaving of operations

Linearizability

- Example:

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

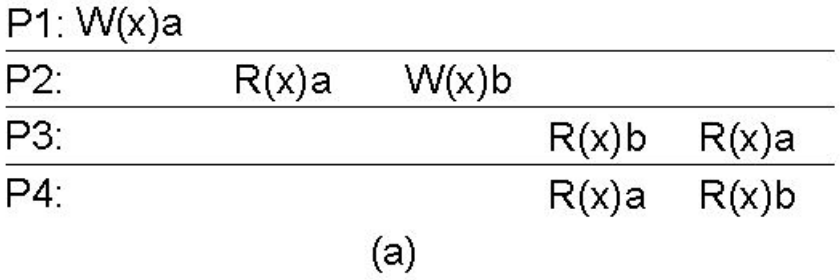
Sequential Consistency Example

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

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)**

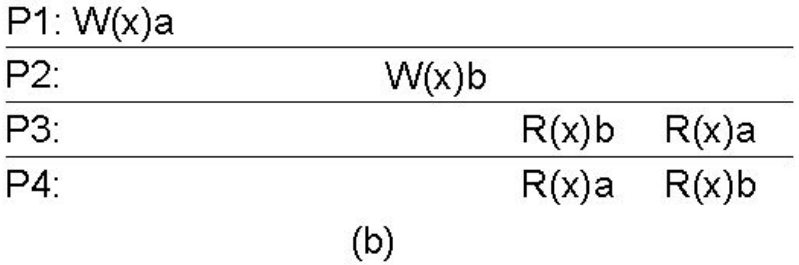
Causal consistency

- Causally related writes must be seen by all processes in the same order.
  - Concurrent writes may be seen in different orders on different machines



(a)

Not Valid



(b)

Valid

Causal consistency

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.

Pipelined RAM Consistency (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.
- Processes do not have to stall before the next writing

Sometimes Counter-intuitive

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

Pipelined RAM Consistency

- Simple and easy to implement
- Better performance than previous models
- Difference between Sequential Consistency and Pipelined RAM Consistency ?
- P1 W_{11} W_{12}
- P2 W_{21} W_{12}
- P3 W_{11} W_{12} W_{21} W_{22}
- P4 W_{21} W_{22} W_{11} W_{12}

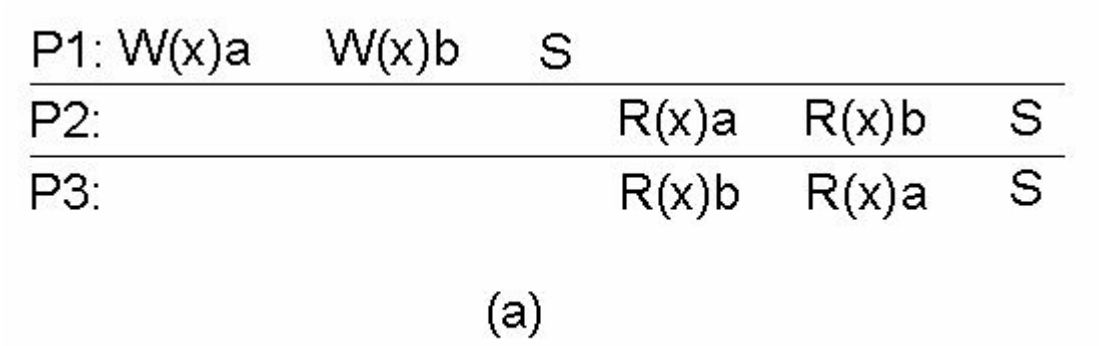
Weak consistency

- It is not necessary to show the change in memory done by every write by every process
- The combined result can be sent when others require it
  - Ex : a process in a critical region
- Pattern of accesses to shared variables – no isolated accesses
- When to show the changes performed?
- Use a *synchronization variable*

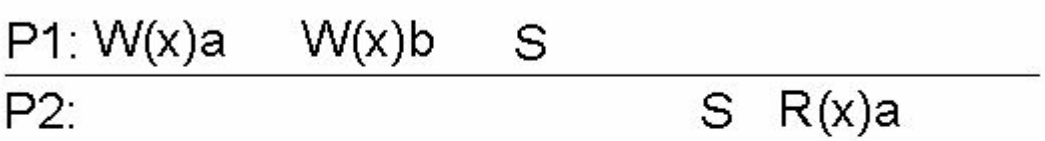
Weak Consistency

- Weak consistency uses synchronization variables to propagate writes to and from a machine at appropriate points:
  - accesses to synchronization variables are sequentially consistent
  - no access to a synchronization variable is allowed until all previous writes have completed in all processors
  - no data access is allowed until all previous accesses to synchronization variables (by the same processor) have been performed
- That is:
  - accessing a synchronization variable “flushes the pipeline”
  - at a synchronization point, all processors have consistent versions of data

- **Weak consistency**
  - Accesses to synchronization variables associated with a data store are sequentially consistent
  - No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
  - No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.
- **Entry and release consistency**
  - Assume shared data are made consistent at entry or exit points of critical sections



(a)



(b)

a) A valid sequence of events for weak consistency.
b) An invalid sequence for weak consistency

Release Consistency

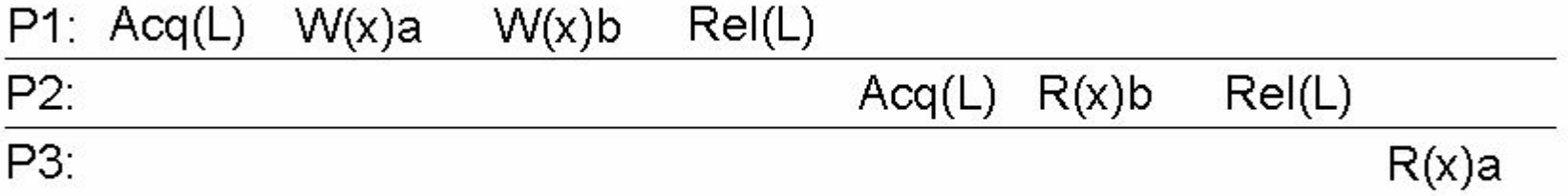
- Release consistency is like weak consistency, but there are two operations “lock” and “unlock” for synchronization
  - (“acquire/release” are the conventional names)
  - doing a “lock” means that writes on other processors to protected variables will be known
  - doing an “unlock” means that writes to protected variables are exported
  - and will be seen by other machines when they do a “lock” (lazy release consistency) or immediately (eager release consistency)

Release Consistency

Acquire: copy remote data to local stores

Release: copy local date to remote stores

- Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
- Before a release is allowed to be performed, all previous reads and writes by the process must have completed
- Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).

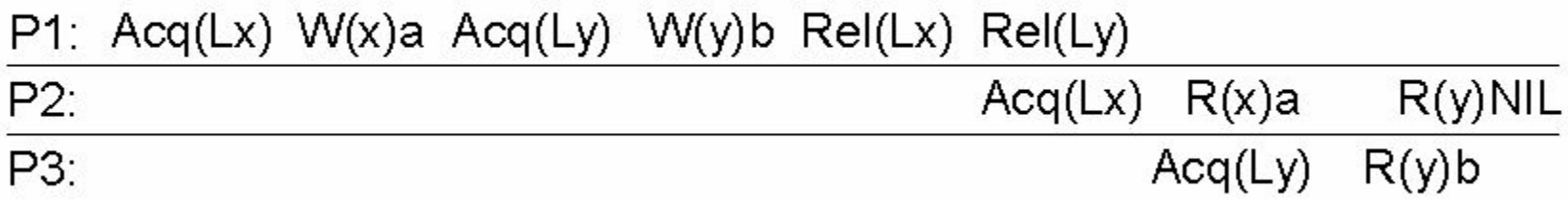


Entry Consistency

Fine-Grained Lazy Release Consistency Conditions:

- An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
- Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
- After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Entry Consistency



A valid event sequence for entry consistency.

Extra overhead and complexity of associating every shared data item with some synchronization variable.

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) 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

(a)

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.

(b)

Example: Consistency for Mobile Users

Example: Consider a distributed database to which you have access through your notebook.

Assume your notebook acts as a front end to the database.

- At location A you access the database doing reads and updates.
- At location B you continue your work, but unless you access the same server as the one at location A, you may detect inconsistencies:
  - you may be reading newer entries than the ones available at A
  - your updates at B may eventually conflict with those at A
  - your updates at A may not have yet been propagated to B

Note: The only thing you really want is that the entries you updated and/or read at A, are in B the way you left them in A. In that case, the database will appear to be consistent to you.

Idea: the database will appear to be consistent to the user