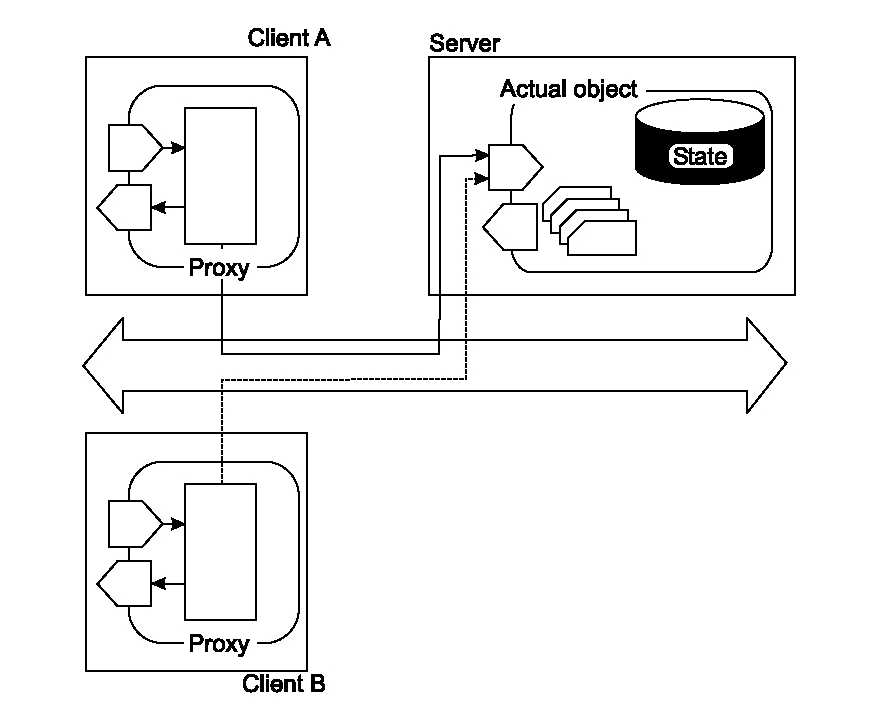
**Date Replication and Coherence**

**Sharing Replicated Data**

* Shared objects
* Replicated shared objects

# **Shared Objects**

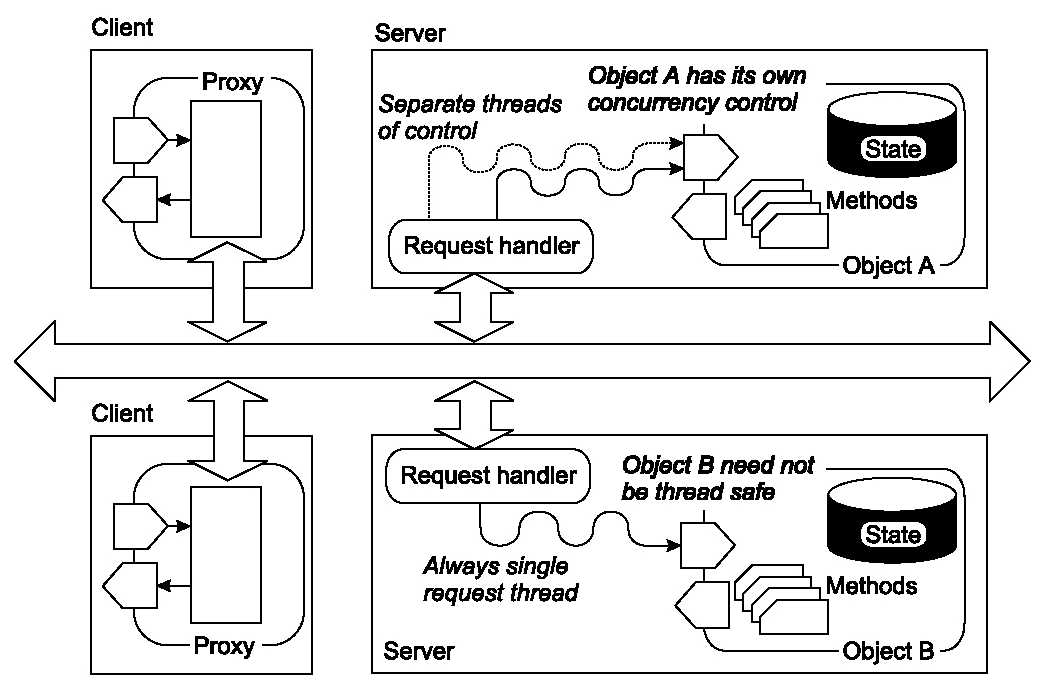
**Problem**: If objects (or data) are shared, we need to do something about concurrent accesses to guarantee state consistency.   
  



# **Concurrency Control: Nonreplicated Shared Objects**

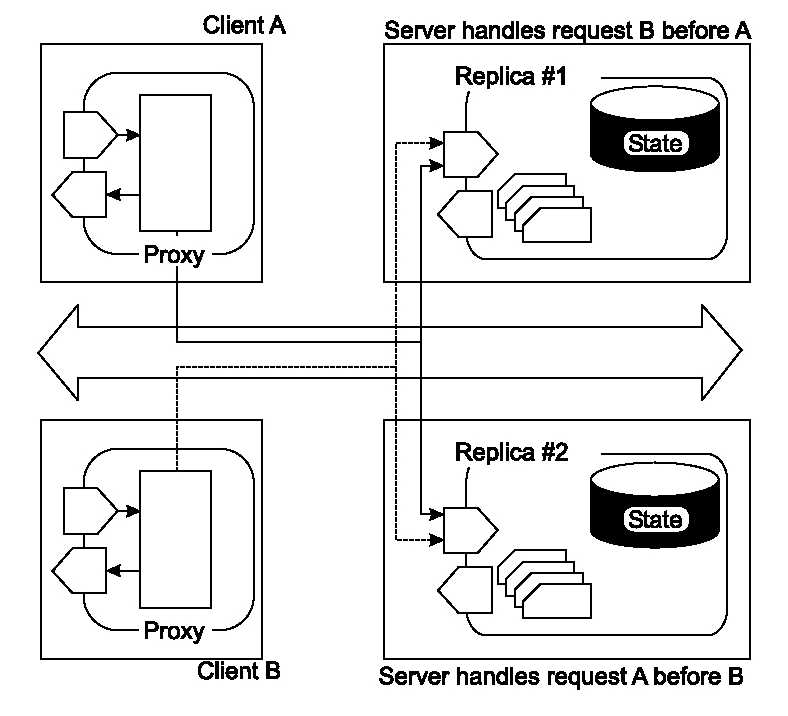
**Basic issue**: If the shared object resides in one address space, controlling concurrency is easy:

* Protect the object using local locks and condition variables
* Serialize access by the server



# **Concurrency Control: Replicated Shared Objects**

**Problem**: Having copies of shared objects (data), and concurrent updates, may get into serious consistency problems



**Performance and Scalability**

**Main issue:**To keep replicas consistent, we need to ensure that all conflicting operations are done in the same order everywhere

**Conflicting operations**: From the world of transactions:

* Read-write conflict: a read operation and a write operation act concurrently
* Write-write conflicts: two concurrent write operations

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

**Solution**: weaken consistency requirements so that, hopefully, global synchronization can be avoided

**-------------------------------------------------------------------------------------------------------**

# **Data-Centric Coherence Models**

**Strong consistency models**: Operations on shared data are synchronized:

* Strict consistency (related to time)
* Sequential consistency (what we are used to)
* Causal consistency (maintains only causal relations)
* PRAM consistency (maintains only individual ordering)

**Weak consistency models**: Synchronization occurs only when shared data is locked and unlocked:

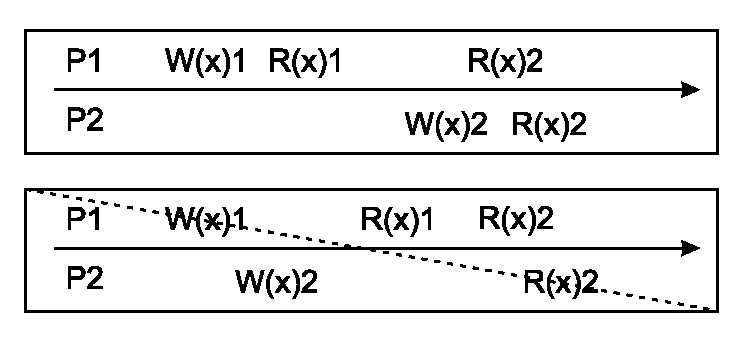
* General weak consistency
* Release consistency
* Entry consistency

**Observation**: The weaker the consistency model, the easier it is to build a scalable solution.

# **Strict Consistency**

*Any read to a shared data item X returns the value stored by the most recent write operation on X.*

**Observation**: It doesn’t make sense to talk about the most recent in a distributed environment.

  
*My Analysis: In Strict Consistency, each operation is done instantaneously, i.e. no interleaving is allowed.*

* *1st case 🡪 In processor P1, W(x)1 happened first hence immediate read returned 1. Then in P2, W(x)2 happened so R(x)2 in both P1 & P2 will return 2 as first write is overwritten*
* *2nd case 🡪 Once W(x)2 was written there is no way that P1 first read could return 1 in case of Strict Consistency as all operations are instantaneously performed*

  Assume all data items have been initialized to 0

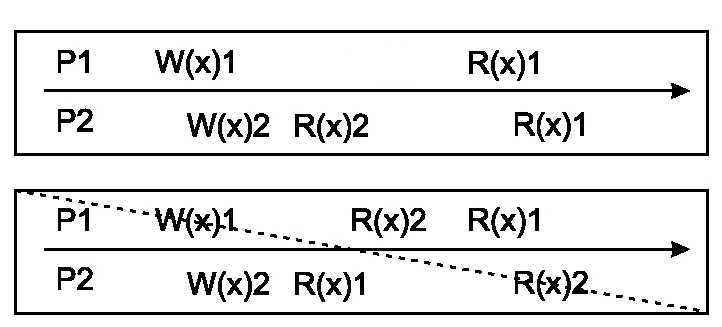
* W(x)1: value 1 is written to x
* R(x)1: reading x returns the value 1

**Note**: Strict consistency is what you get in the normal sequential case, where your program does not interfere with any other program.

# **Sequential Consistency**

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

**Note**: We’re talking about interleaved executions: there is some total ordering for all operations taken together

   
 *r + w >= t,*where

* *r :*read time
* *w:*write time
* *t:*minimal packet transmission time between nodes

*My Analysis: In Sequential Consistency, all operations of p1 & p2 are re-organized to form a valid sequence, which can be interleaved. Then operations are executed in same sequence over all the replicas*

* *1st case 🡪*
* *W(x)1 can precedes W(x)2 (Seq1)*
* *W(x)2 can precede W(x)1(Seq2).*

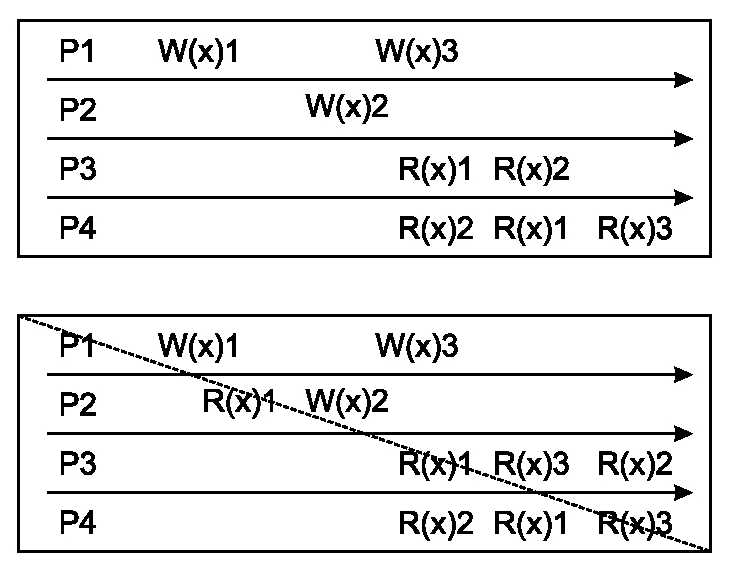
*Both are valid order in Sequential Consistency but not in Strict Consistency.  
But any one of the sequence can be executed across all the replica. If at one replica Seq1 is getting executed and at the other replica Seq2 is executed, then it does not obey Sequential Consistency rule and data can be inconsistent state  
In this case assumed W(x)2 precedes W(x)1. Hence, P2 first read will see 2 and second read will see 1 (by then W(x)1 have overwritten the previous value)  
P1 will read 1*

* *2nd case* 🡪
* *Assuming W(x)1 precede W(x)2 🡪 P2 read is correct. It sees 1 then 2. But P1 read is not correct.*
* *Assuming W(x)2 precede W(x)1 🡪 Now P1 read is correct by not P2*

*With respect to Strict Consistency, both cases are incorrect*

# **Causal Consistency**

*Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order by different processes.*



*My Analysis: In Causal Consistency, a read operation sand-witched between 2 write operations shall follow that order as they are causally related.   
E.g. W(x)1 🡪 R(x)1🡪W(x)2. All replicas shall follow this order*

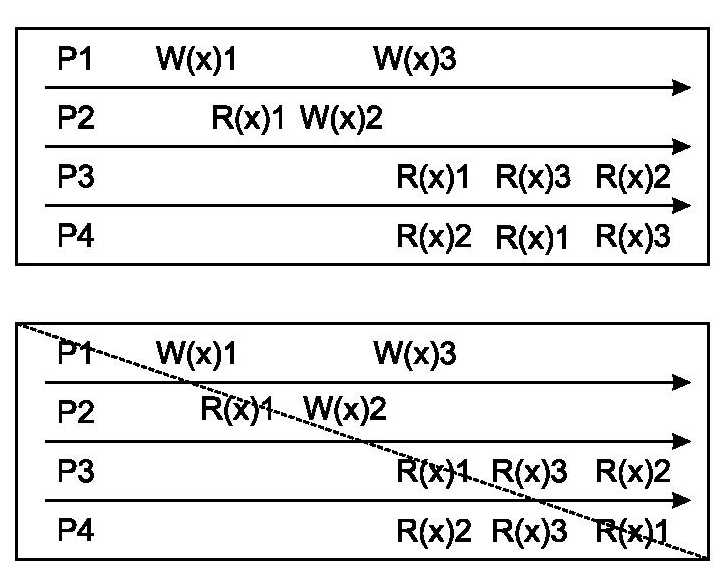
* *1st case 🡪 All 3 writes operations can happen in any order, i.e. 6 valid sequences*
* *W(x)1🡪W(x)2🡪W(x)3*
* *W(x)1🡪W(x)3🡪W(x)2*
* *W(x)2🡪W(x)1🡪W(x)3*
* *W(x)2🡪W(x)3🡪W(x)1*
* *W(x)3🡪W(x)1🡪W(x)2*
* *W(x)3🡪W(x)2🡪W(x)1*

*Hence, read operation for P3 🡪 indicates W(x)1🡪 W(x)2  
read operation for P4 🡪 indicates W(x)2🡪W(x)1🡪 W(x)3  
  
Again, these sequences are not valid with respect to Strict Consistency, as for that, there shall be one only one valid sequence*

* *2nd case 🡪 W(x)1 & W(x)2 are causally related, hence W(x)1 must precede W(x)2. W(x)3 can happen before W(x)1, between W(x)1 & W(x)2 or after W(x)2.   
  P3 read is correct but not P4*

# **Pipelined RAM Consistency**

*Writes done by a single process are received 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 .*



*My Analysis: Only one processor writes are guaranteed to occur in particular sequence*

* *1st case 🡪 W(x)1 precede W(x)3. That’s the only guarantee of FIFO consistency W(x)2 can come anywhere.  
  In both P3 & P4 , R(x)1🡪 R(x)3  
    
  Note, w.r.t to Casual Consistency, W(x)1 🡪 W(x)2, W(x)3 could have occurred anywhere*
* *2nd case 🡪 In P4, R(x)3 🡪 R(x)1, which is not allowed for FIFO consistency model*

**Weak Consistency**

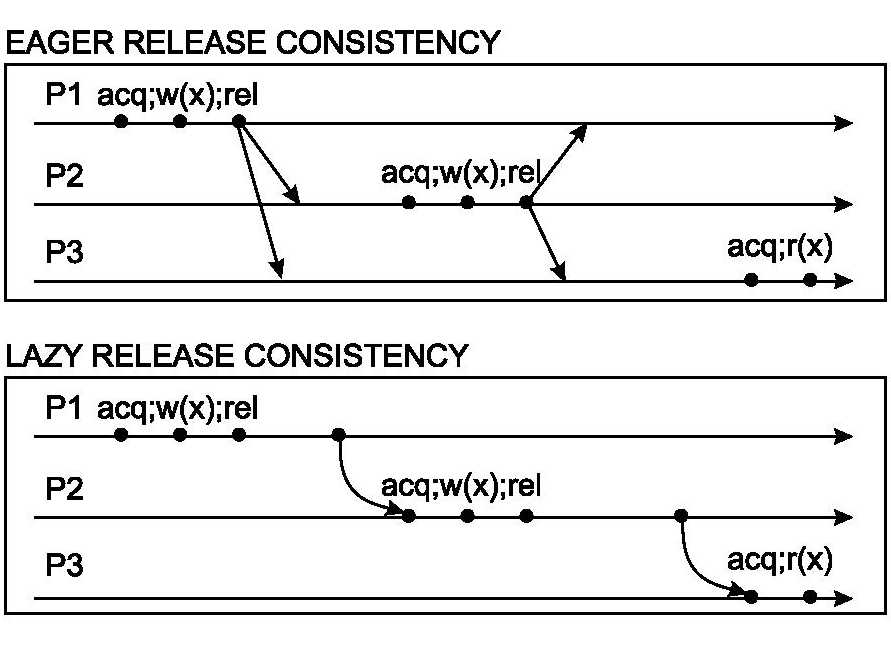
* Accesses to synchronization variables are sequentially consistent.
* No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
* No data access is allowed to be performed until all previous accesses to synchronization variables have been performed.

**Basic idea**: You don’t care that reads and writes of a series of operations are immediately known to other processes. You just want the effect of the series as a whole to be known.

**Observation**: Weak consistency implies that we need to lock and unlock data (implicitly or not).

**Release Consistency**

**Idea:**Divide access to a synchronization variable into two parts: an *acquire* and a *release* phase. Acquire forces a requester to wait until the shared data can be accessed; release sends requester’s local value to shared memory.



# **Entry Consistency**

* With release consistency, all local updates are propagated to other processors during release of shared variable.
* With entry consistency, each shared variable is associated with a synchronization variable.
* When acquiring the synchronization variable, the most recent values of its associated shared variables are fetched.

**Note:** Where release consistency affects all shared variables, entry consistency affects only those shared variables associated with a synchronization variable.

**Question**: What would be a convenient way of making entry consistency more or less transparent to programmers?

------------------------------------------------------------------------------------------------------------------------------------------

**Client-Centric Coherence Models**

* System model
* Read-your-writes
* Monotonic reads
* Write-follows-reads
* Monotonic writes

**Goal**: Show how we can perhaps avoid system-wide consistency, by concentrating on what specific clients want, instead of what should be maintained by servers.

**Background**: Most large-scale distributed systems (i.e., databases) apply replication for scalability, but can support only weak consistency:

* **DNS**: Updates are propagated slowly, and inserts may not be immediately visible.
* **NEWS**: Articles and reactions are pushed and pulled throughout the Internet, such that reactions can be seen before postings.
* **Lotus Notes**: Geographically dispersed servers replicate documents, but make no attempt to keep (concurrent) updates mutually consistent.
* **WWW**: Caches all over the place, but there need be no guarantee that you are reading the most recent version of a page.

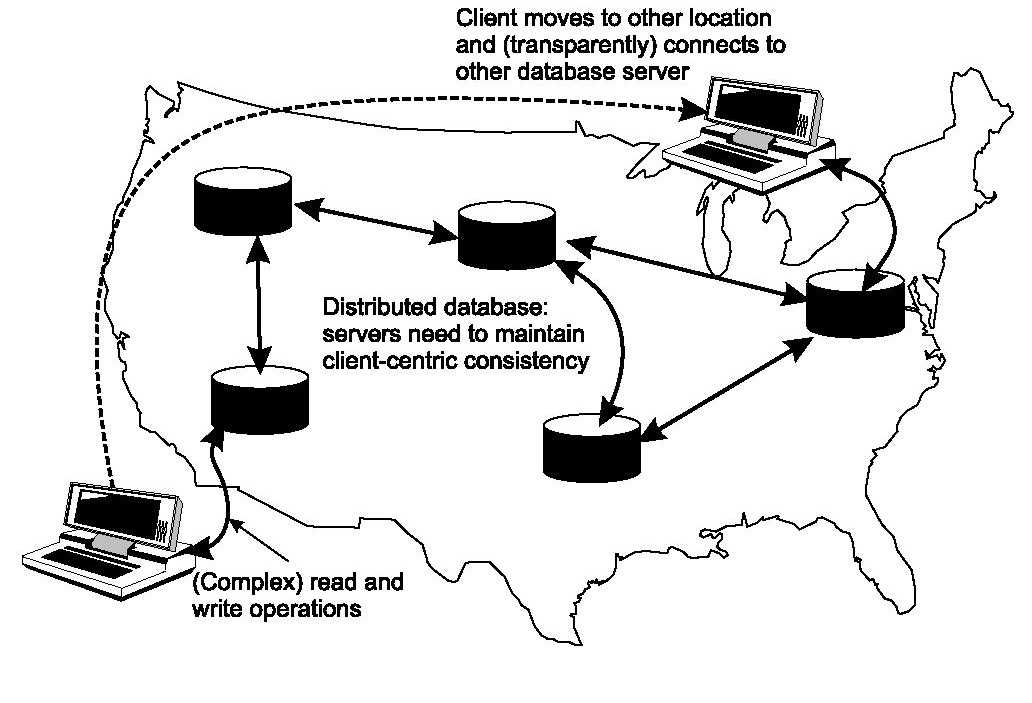
**Consistency for Mobile Users**

**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:
  + your updates at *A* may not have yet been propagated to *B*
  + you may be reading newer entries than the ones available at *A*
  + your updates at *B* may eventually conflict with those at *A*

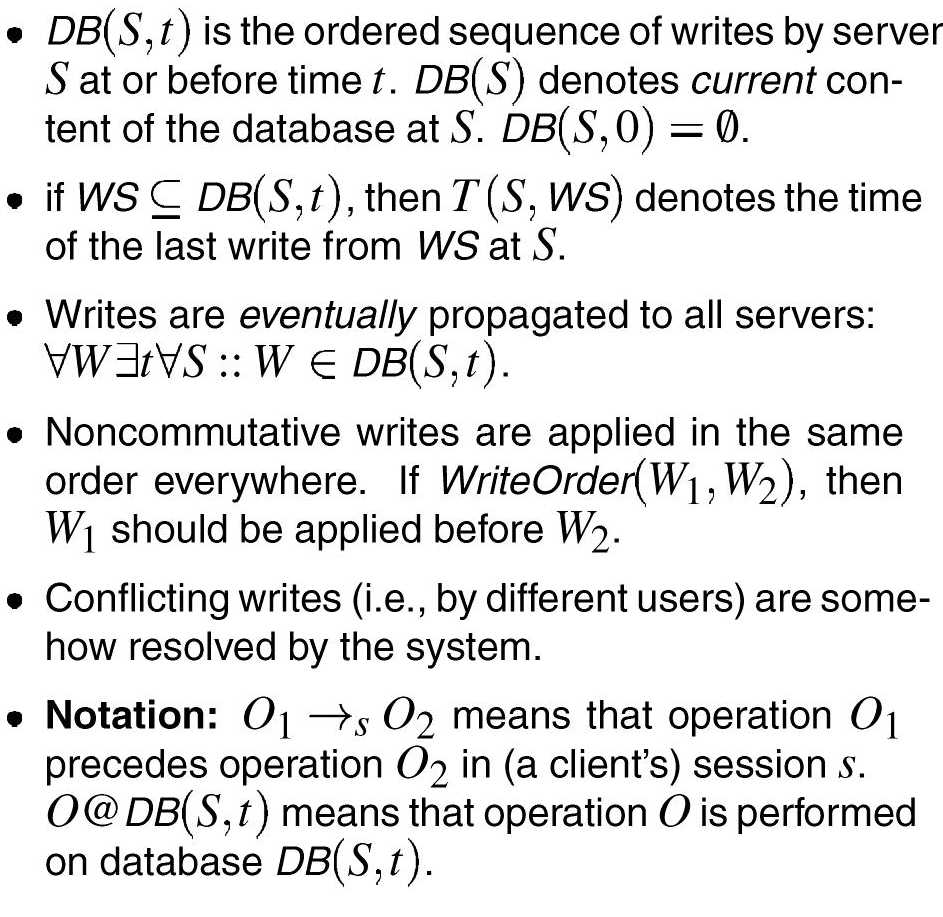
Note: The only thing you really want is that the entries you updated and/or read at *A*, are in *B* the way you left them in *A*. In that case, the database will appear to be consistent to you.

**Basic Architecture**



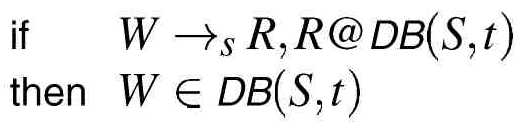
**System Model**

* Each write has a globally unique identifier *WID*



**Read Your Writes**

**Definition**: if read *R* follows write *W* in a session, and *R* is performed on *DB(S,t)*, then *W* should have been in *DB(S,t)*:



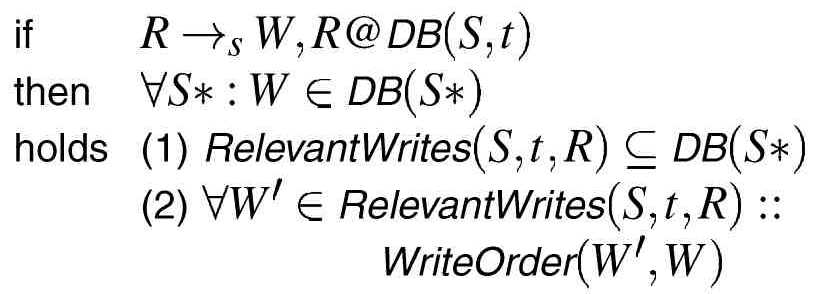
**Note**: There is no guarantee that *R* returns *W*: there may have been writes from other clients at *S* between *W* and *R*.

**Example**: Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.

**Writes Follows Reads**

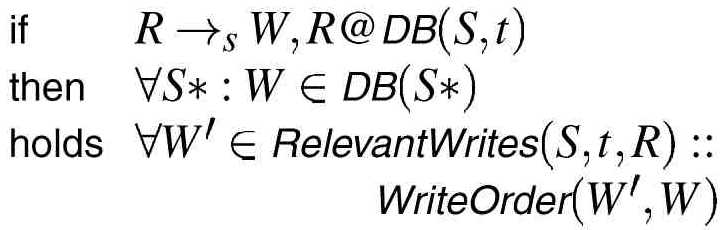
**Intuitively**: If a read precedes a write (in a session), then that write is performed after all writes that preceeded the read.

**Definition**: if a read *R* precedes a write *W*, and *R* is performed at server *S*, then if *W* is performed at server *S\**, all relevant writes for R are also performed at *S\**, and before *W*:



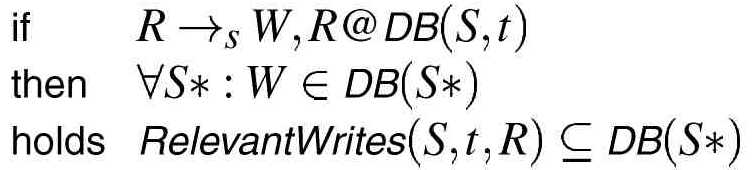
**Note**: We are imposing two conditions which need not always be relevant:

* Drop (1): See locally updated data items without having to see all the writes at other servers that formally preceded the update. However, you do not want old writes to undo your local update, so keep (2). **WFRO**:

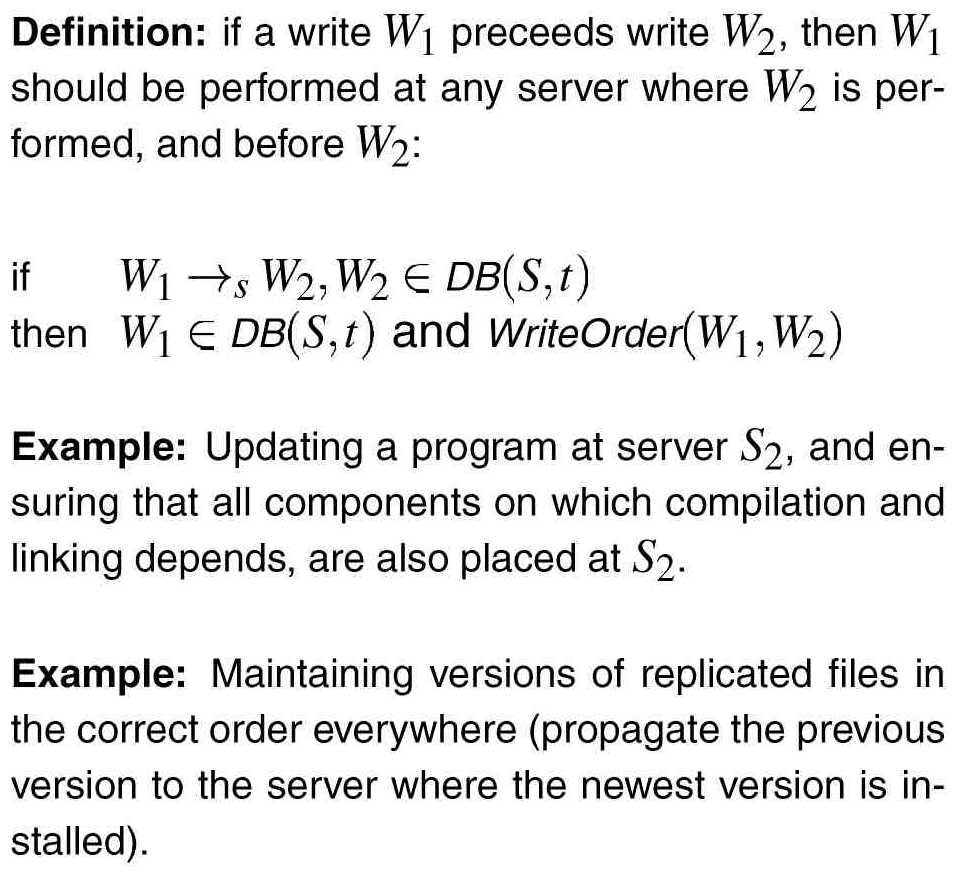
 

* Drop (2): See reactions to posted articles only if you have the original posting, but don’t worry about the ordering between reaction and posting,

so keep (1). **WFRP**:



**Monotonic Writes**



**--------------------------------------------------------------------------------------------------------**

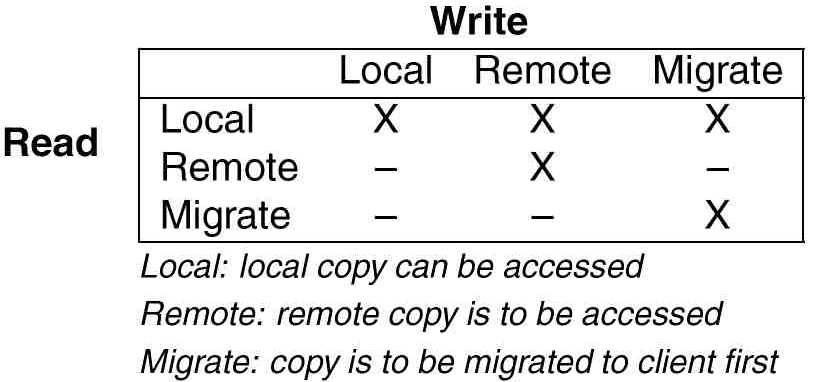
**Replica Coherence Protocols**

* Sequential consistency protocols
* General design issues

**Algorithms for Sequential Consistency**

**Observation**: In the end we always want sequential consistency, whether or not it is implemented using weak consistency in combination with synchronization variables (locks).

**Observation**: If we discard whether all data is globally consistent (as is the case with release consistency), or specific data is consistent (entry consistency), we need to distinguish three access methods.



**Read Remote, Write Remote**

* A single process is responsible for maintaining the shared data.
* All read and write requests are forwarded to the server.
* The server does the read/write operation, and returns a reply to the client.
* **Adv**: Really simple implementation, and very easy to maintain sequential consistency (for free).
* **Disadv**: Server can become a bottleneck, but this can be partially alleviated by distributing shared data using some data-address hashing scheme.

**Read Migrate, Write Migrate**

* Data is always migrated to the process that wants to access it
* No distinction between read or write accesses.
* Better watch out for **thrashing**: data keeps migrating between different processes so that shared data access rate drops too much.
* In some cases , it can easily be integrated with virtual memory techniques: a page fault gives the local OS full control, so that fetching data can be handled transparently.
* You’ll have to do something about locating data. Either multicasting or keeping track of shared data through chains of forwarding pointers. Doesn’t really scale.

**Read Local, Write Remote**

* Essentially, get your own local copy of the data and read it as long as it’s valid.
* Writing data means (1) forwarding your update to a primary, (2) having all copies invalidated, and (possibly) (3) doing the write locally after receiving ACK from primary.
* Experience shows that this is a very reasonable model, but that more needs to be done in order to achieve competitive efficiency.

**Read Local, Write Migrate**

* Get your own local copy of the data and read it as long as it’s valid.
* Writing data means (1) becoming primary (fetching the most recent state), (2) having all copies invalidated, and (3) doing the update locally.
* Interesting observation: do an immediate write, and delay the invalidation (affects the consistency, but improves the performance).
* Experience shows that this is a very reasonable model, but that more needs to be done in order to achieve competitive efficiency.

**Read Replicate, Write Replicate**

* Again, get your own local copy of the data and read it as long as it’s valid.
* Writing data means (1) getting a local copy if you don’t have it yet, (2) write to your local copy, and (3) update all other copies.
* The hard part: to achieve sequential consistency, we need a totally ordered multicast protocol. We’re back where we started. It turns out this is doable, for local distributed systems with hardware multicasting facilities (e.g., Amoeba on Ethernet).

**Replication Strategies (1/3)**

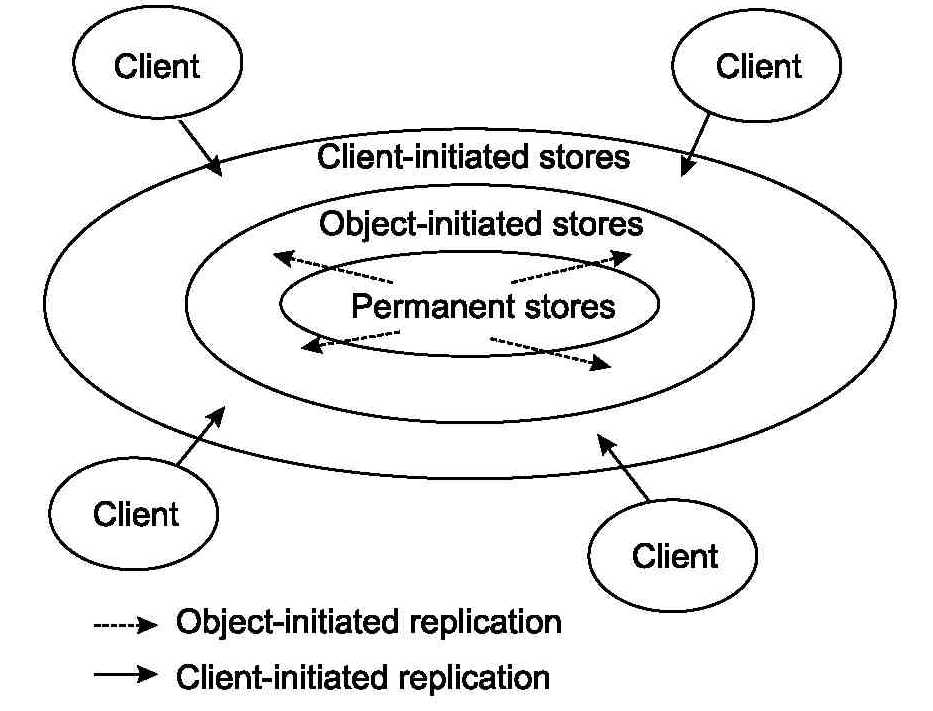
**Observation**: Choosing one of the basic algorithms for (sequential) consistency is only the first step. There are many more refinements to make!

**Model**: We consider objects (and don’t worry whether they contain just data or code, or both)

**Distinguish different stores**: A store is capable of hosting a replica of an object:

* **Permanent store**: Server always having a replica
* **Object-initiated store**: Server that can dynamically host a replica upon request of the object
* **Client-initiated store**: Server that can dynamically host a replica upon request of a client

**Replication Strategies (2/3)**



**Replication Strategies (3/3)**

**Change distribution**: What is distributed between the replicas:

* Notification/in validation messages
* Transfer of state (full state, or only the differences)
* Method shipping (send the operation that caused the change of state)

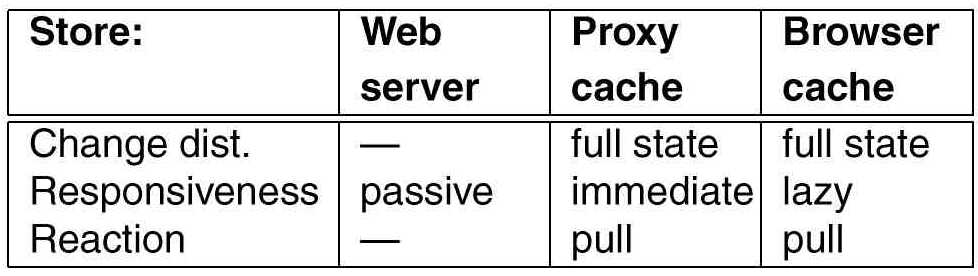
**Store responsiveness**: When does a replica take action when it notices inconsistency:

* Immediate reaction
* Lazy reaction (e .g., periodic, or after some event)
* Remains passive (e .g., because it is most-up-to-date)

**Store reaction**: What does a replica actually do:

* Pull (i.e. fetch change)
* Push (i.e. propagate change)

**Example Consistency Protocol**

   
 **Observation**: Basically, all stores in the Web apply the same replication strategy regardless the content of the Web document (there are a few exceptions)

**Observation**: Caching is becoming more and more problematic in the Web, as the number of documents grows exponentially.

**Again**: It is the consistency requirements that determines the applicability of scaling techniques.