Distributed Systems

Lec 12: Consistency Models – Sequential, Causal, and Eventual Consistency

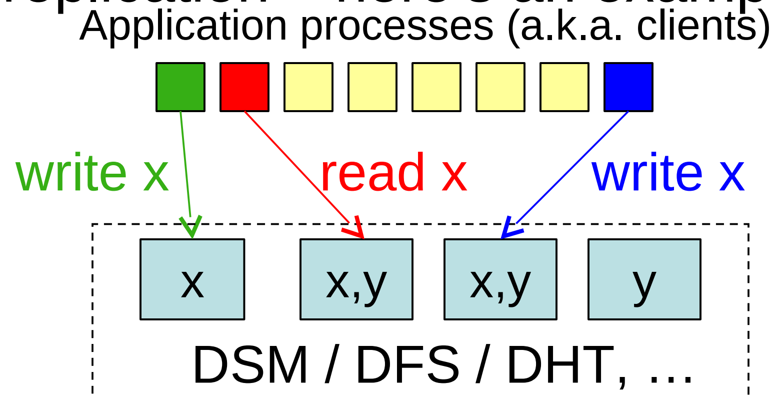
Slide acks: Jinyang Li

(http://www.news.cs.nyu.edu/~jinyang/fa10/notes/ds-eventual.ppt)

Consistency (Reminder)

- 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

- Let's focus on replication – here's an example:



Consistency (Reminder)

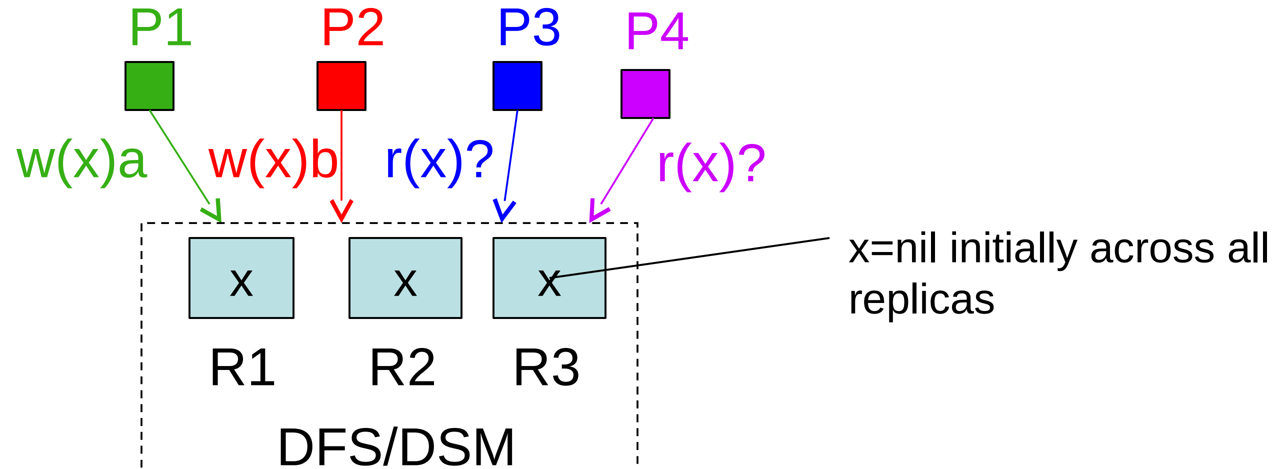
- What is a consistency model?
  - Contract between a distributed data system (e.g., DFS, DSM) and processes constituting its applications
  - E.g.: “If a process reads a certain piece of data, I (the DFS/DSM) pledge to return the value of the last write”

- What are some consistency models?
  - Strict consistency
  - Sequential consistency
  - Causal consistency
  - Eventual consistency

- Less intuitive, harder to program
- More feasible, scalable, efficient (traditionally)

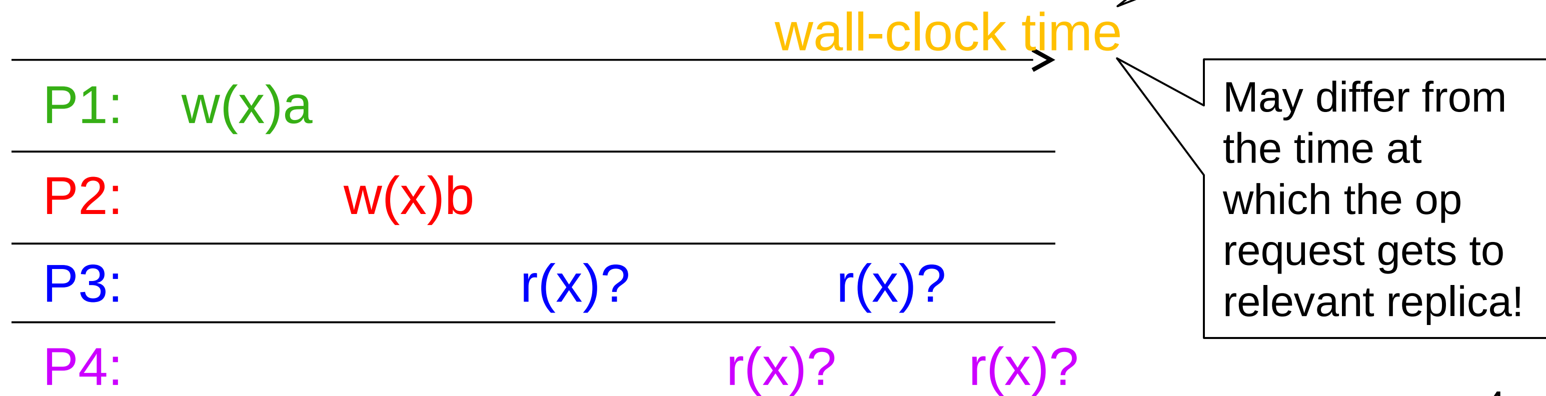
- Variations boil down to:
  - The allowable staleness of reads
  - The ordering of writes across all replicas

Example



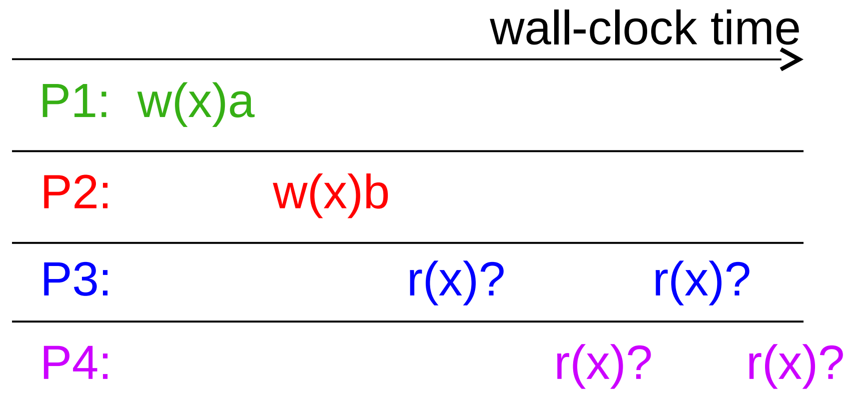
- Consistency model defines what values reads are admissible by the DFS/DSM

Time at which client process issues op



Strict Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in order of wall-clock time at which they were issued
- Therefore:
  - Reads are never stale
  - All replicas enforce wall-clock ordering for all writes
- If DSM were strictly consistent, what can these reads return?

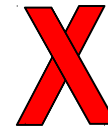


Strict Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in order of wall-clock time at which they were issued
- Therefore:
  - Reads are never stale
  - All replicas enforce wall-clock ordering for all writes
- If DSM were strictly consistent, what can these reads return?

wall-clock time →

P1:	w(x)a				
P2:		w(x)b			
P3:			r(x)b		r(x)b	
P4:				r(x)b		r(x)b

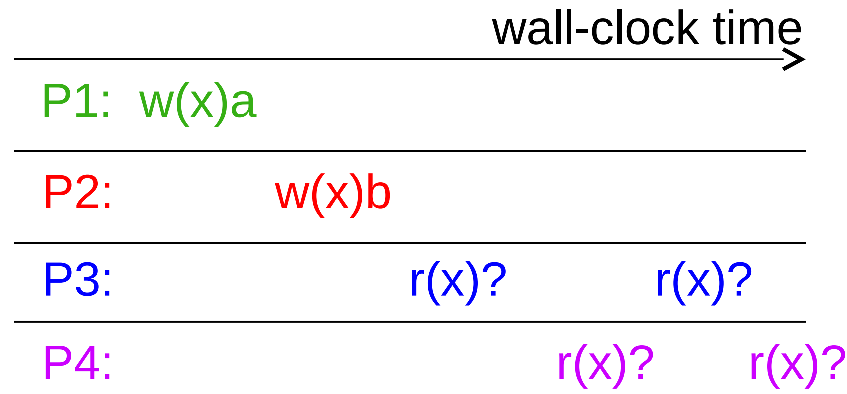


wall-clock time →

P1:	w(x)a				
P2:		w(x)b			
P3:			r(x)a		r(x)b	
P4:				r(x)b		r(x)b

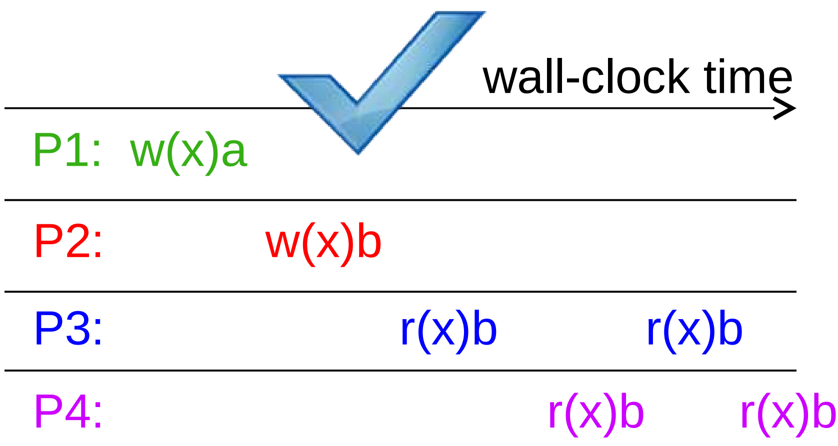
Sequential Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in some global ordering, and the ops of each client process appear in the order specified by its program
- Therefore:
  - Reads may be stale in terms of real time, but not in logical time
  - Writes are totally ordered according to logical time across all replicas
- If DSM were seq. consistent, what can these reads return?



Sequential Consistency (Last Time)

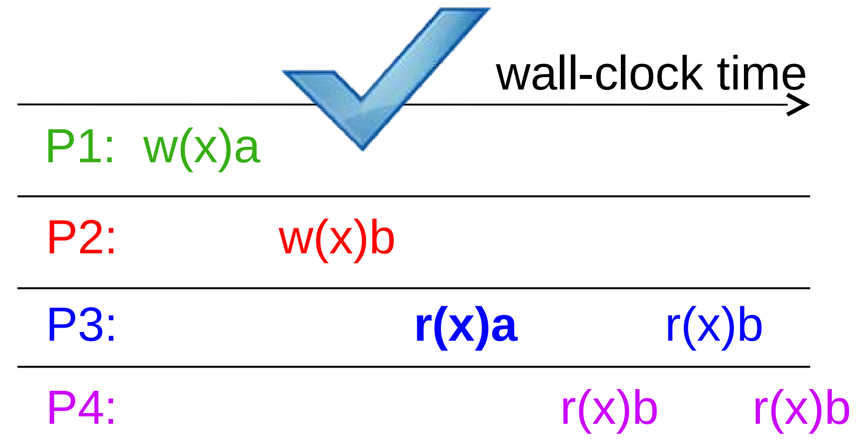
- Any execution is the same as if all read/write ops were executed in some global ordering, and the ops of each client process appear in the order specified by its program



What's a global sequential order that can explain these results?

wall-clock ordering

This was also strictly consistent



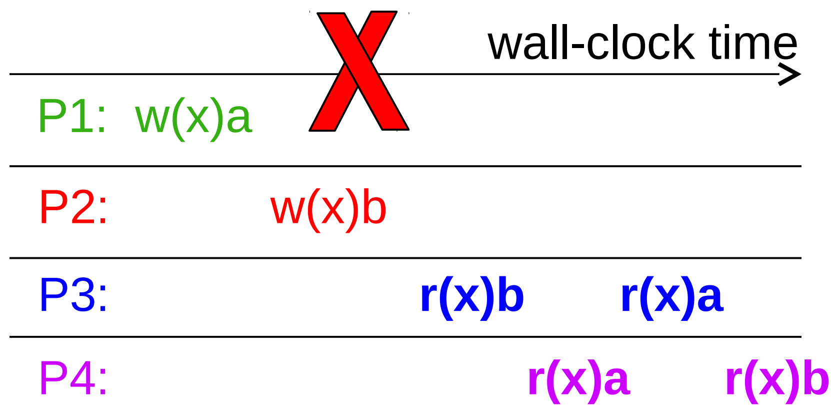
What's a global sequential order that can explain these results?

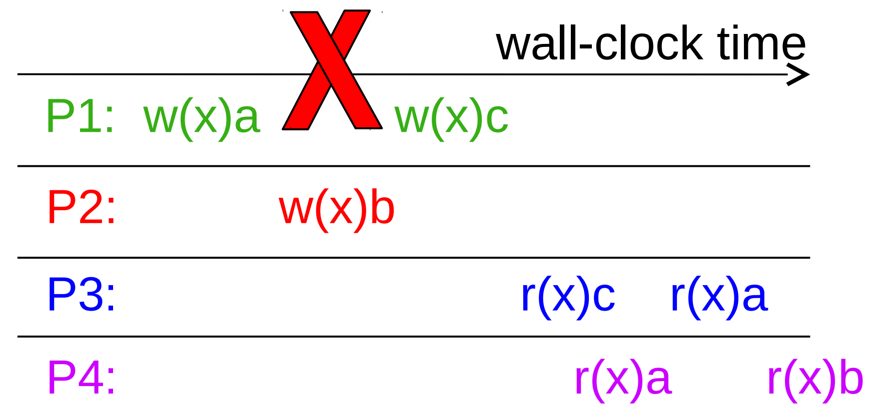
w(x)a, r(x)a, w(x)b, r(x)b, …

This wasn't strictly consistent

Sequential Consistency (Last Time)

- Any execution is the same as if all read/write ops were executed in some global ordering, and the ops of each client process appear in the order specified by its program





No global ordering can explain these results…

=> not seq. consistent

No global *sequential* global ordering can explain these results…

E.g.: the following global ordering doesn't preserve P1's ordering

w(x)c, r(x)c, w(x)a, r(x)a, w(x)b, …

Today

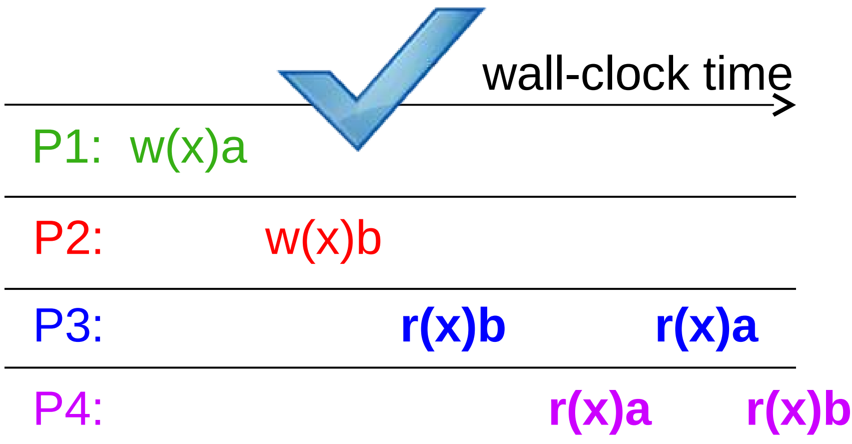
- Causal consistency
- Eventual consistency
- Implementing eventual consistency

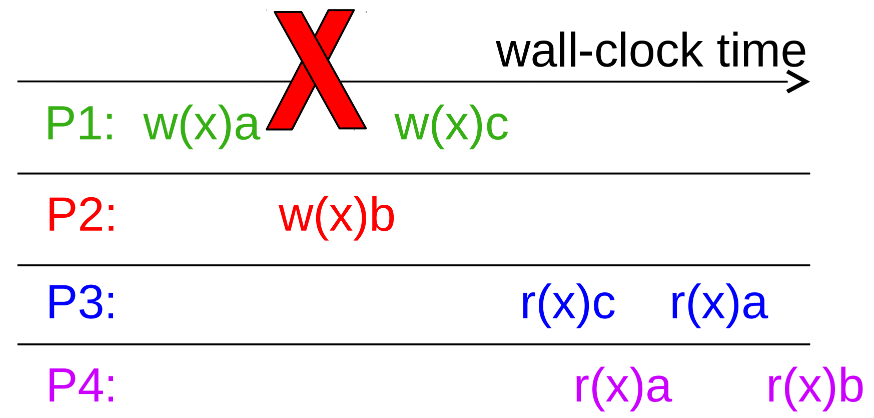
Causal Consistency

- Remember causality notion from Lamport (logical) clocks?
  – That's what causal consistency enforces
- Causal consistency: Any execution is the same as if all causally-related read/write ops were executed in an order that reflects their causality
  – All concurrent ops may be seen in different orders
- Therefore:
  – Reads are fresh only w.r.t. the writes that they are causally dependent on
  – Only causally-related writes are ordered by all replicas in the same way, but concurrent writes may be committed in different orders by different replicas, and hence read in different orders by different applications

Causal Consistency: (Counter)Examples

- Any execution is the same as if all causally-related read/write ops were executed in an order that reflects their causality
  - All concurrent ops may be seen in different orders





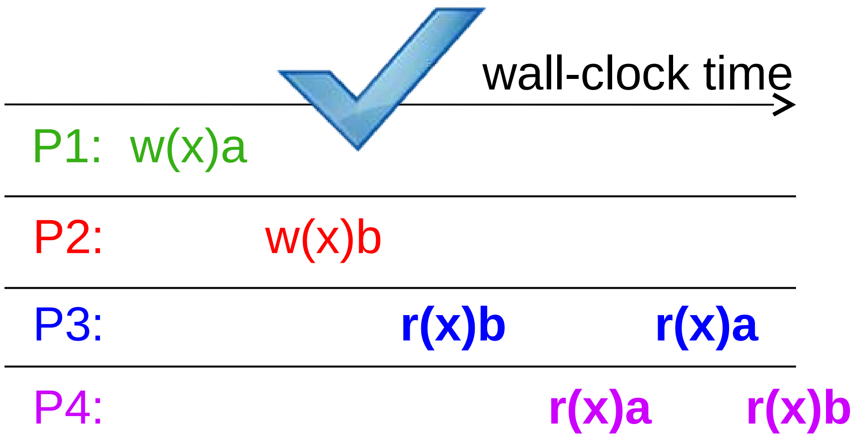
Only per-process ordering restrictions:
w(x)b < r(x)b; r(x)b < r(x)a; …
w(x)a || w(x)b, hence they can be seen in ≠ orders by ≠ processes

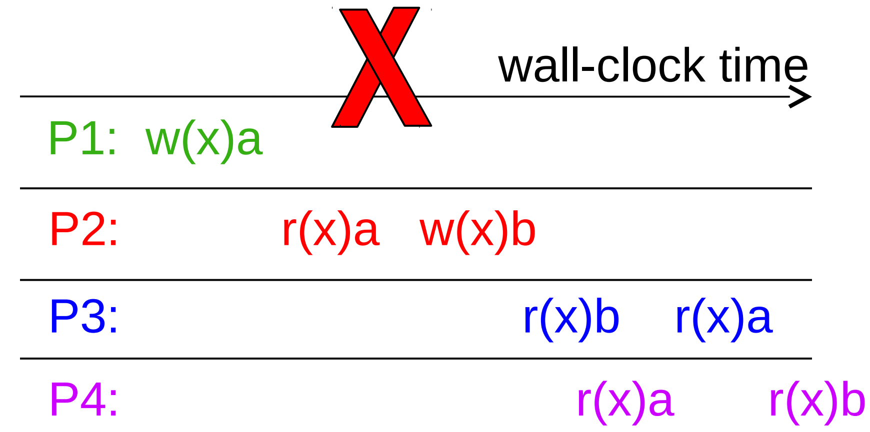
This wasn't sequentially consistent

Having read c (r(x)c), P3 must continue to read c or some newer value (perhaps b), but can't go back to a, b/c w(x)c was conditional upon w(x)a having finished

Causal Consistency: (Counter)Examples

- Any execution is the same as if all causally-related read/write ops were executed in an order that reflects their causality
  - All concurrent ops may be seen in different orders





w(x)b is causally-related on r(x)a, which is causally-related on w(x)a. Therefore, system must enforce w(x)a < w(x)b ordering. But P3 violates that ordering, b/c it reads a after reading b.

Why Causal Consistency?

- Causal consistency is strictly weaker than sequential consistency and can give weird results, as you've seen
  - If system is sequentially consistent => it is also causally consistent
- BUT: it also offers more possibilities for concurrency
  - Concurrent operations (which are not causally-dependent) can be executed in different orders by different people
  - In contrast, with sequential consistency, you need to enforce a global ordering of all operations
  - Hence, one can get better performance than sequential
- From what I know, not very popular in industry
  - So, we're not gonna focus on it any more

Eventual Consistency (Overview)

- Allow stale reads, but ensure that reads will eventually reflect previously written values
  - Even after very long times

- Doesn’t order concurrent writes as they are executed, which might create conflicts later: which write was first?

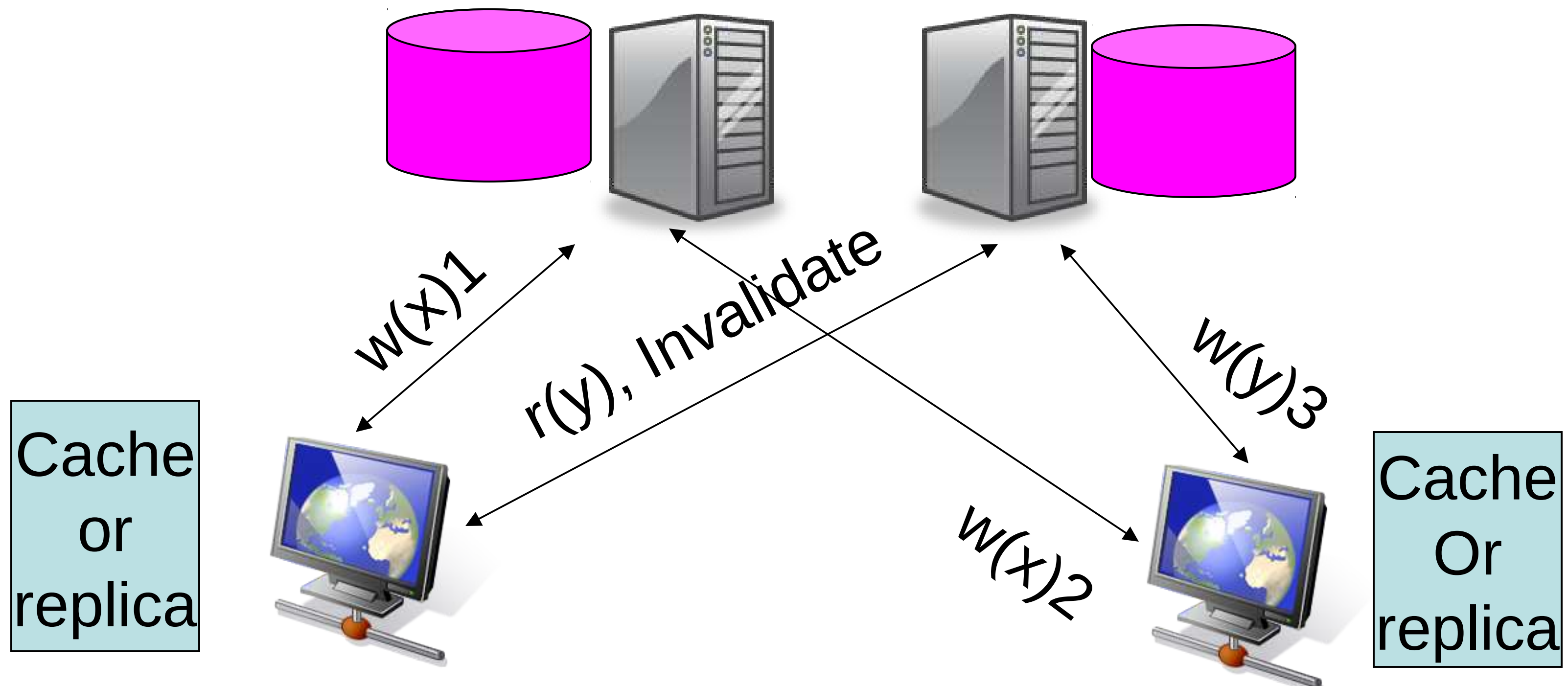
- Used in Amazon’s Dynamo, a key/value store
  - Plus a lot of academic systems
  - Plus file synchronization ← familiar example, we’ll use this

Why Eventual Consistency?

- More concurrency opportunities than strict, sequential, or causal consistency
- Sequential consistency requires highly available connections
  - Lots of chatter between clients/servers
- Sequential consistency many be unsuitable for certain scenarios:
  - Disconnected clients (e.g. your laptop goes offline, but you still want to edit your shared document)
  - Network partitioning across datacenters
  - Apps might prefer potential inconsistency to loss of availability

Case-in-Point: Realizing Sequential Consistency

- All reads/writes to address X must be ordered by one memory/storage module responsible for X (see Ivy, Lec10)
- If you write data that others have, you must let them know
- Thus, everyone must be online all the time



Why (Not) Eventual Consistency?

- ✓ Support disconnected operations or network partitions
  - Better to read a stale value than nothing
  - Better to save writes somewhere than nothing
- ✓ Support for increased parallelism
  - But that’s not what people have typically used this for
- ☒ Potentially anomalous application behavior
  - Stale reads and conflicting writes…

Sequential vs. Eventual Consistency

- Sequential: pessimistic concurrency handling
  - Decide on update order as they are executed

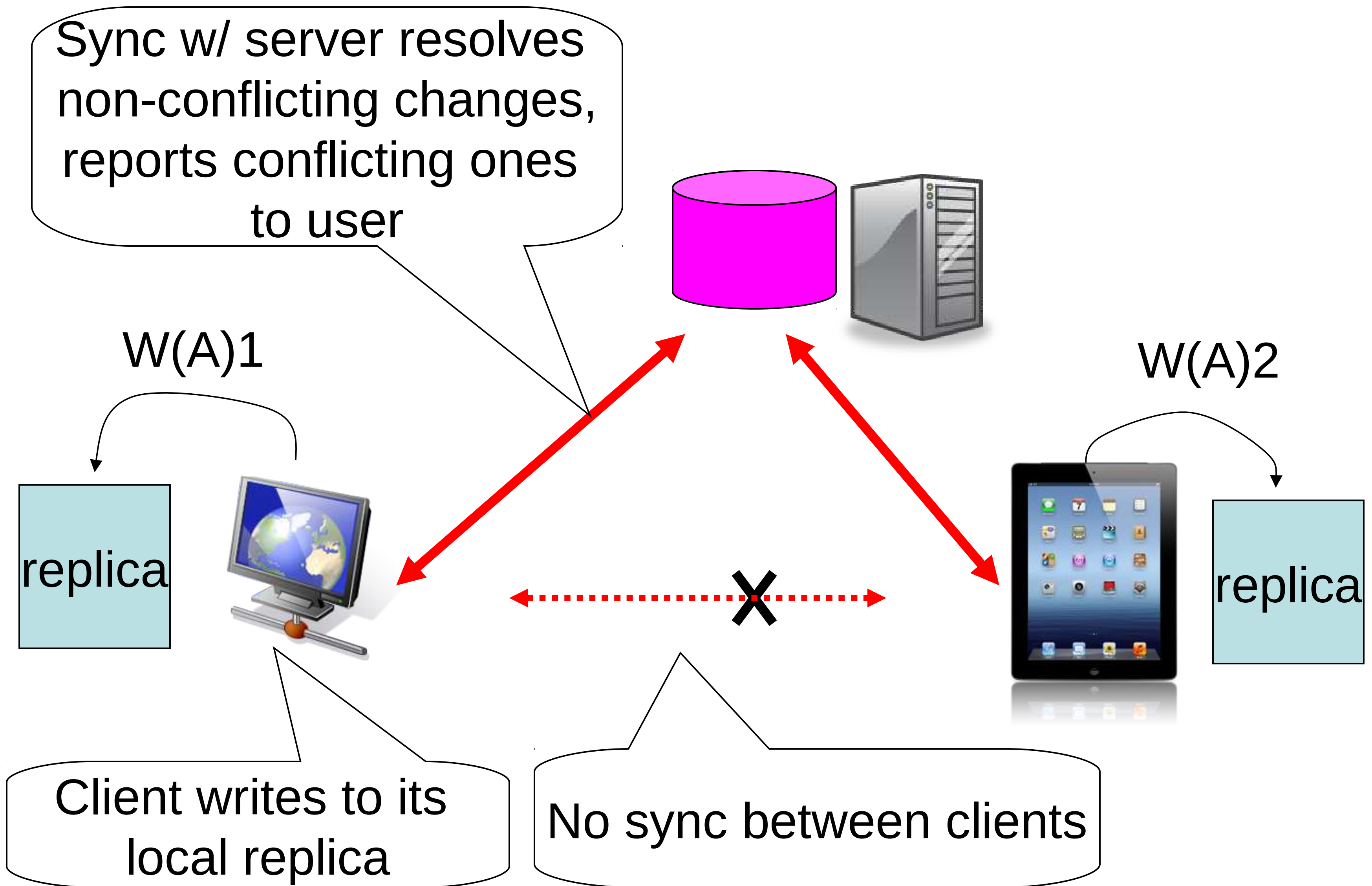
- Eventual: optimistic concurrency handling
  - Let updates happen, worry about deciding their order later
  - May raise conflicts
    - Think about when you code offline for a while – you may need to resolve conflicts with other teammembers when you commit
    - Resolving conflicts is not that difficult with code, but it’s really hard in general (e.g., think about resolving conflicts when you’ve updated an image)

Example Usage: File Synchronizer

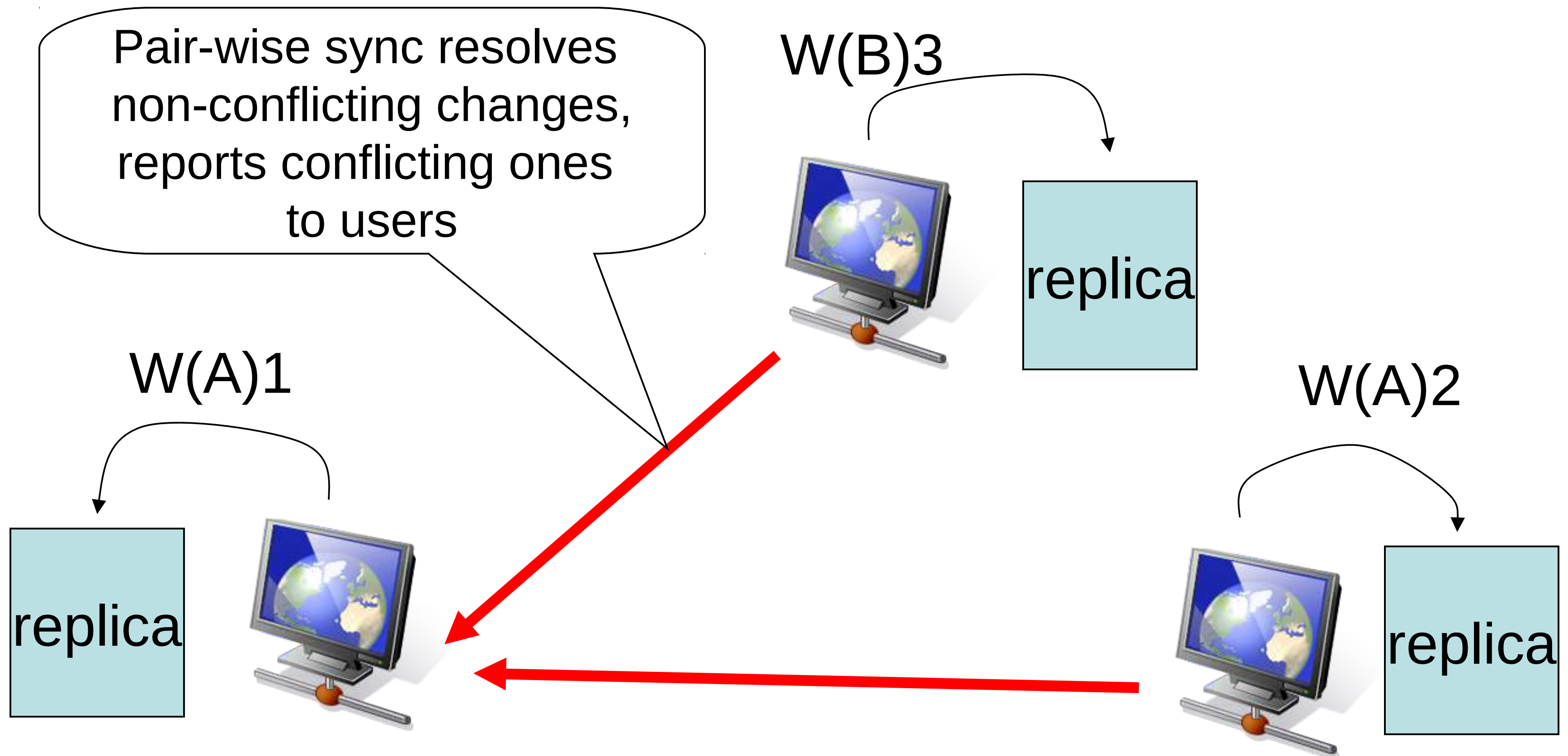
- One user, many gadgets, common files (e.g., contacts)

- Goal of file synchronization
  1. All replica contents eventually become identical
  2. No lost updates
     - Do not replace new version with old ones

Operating w/o Total Connectivity



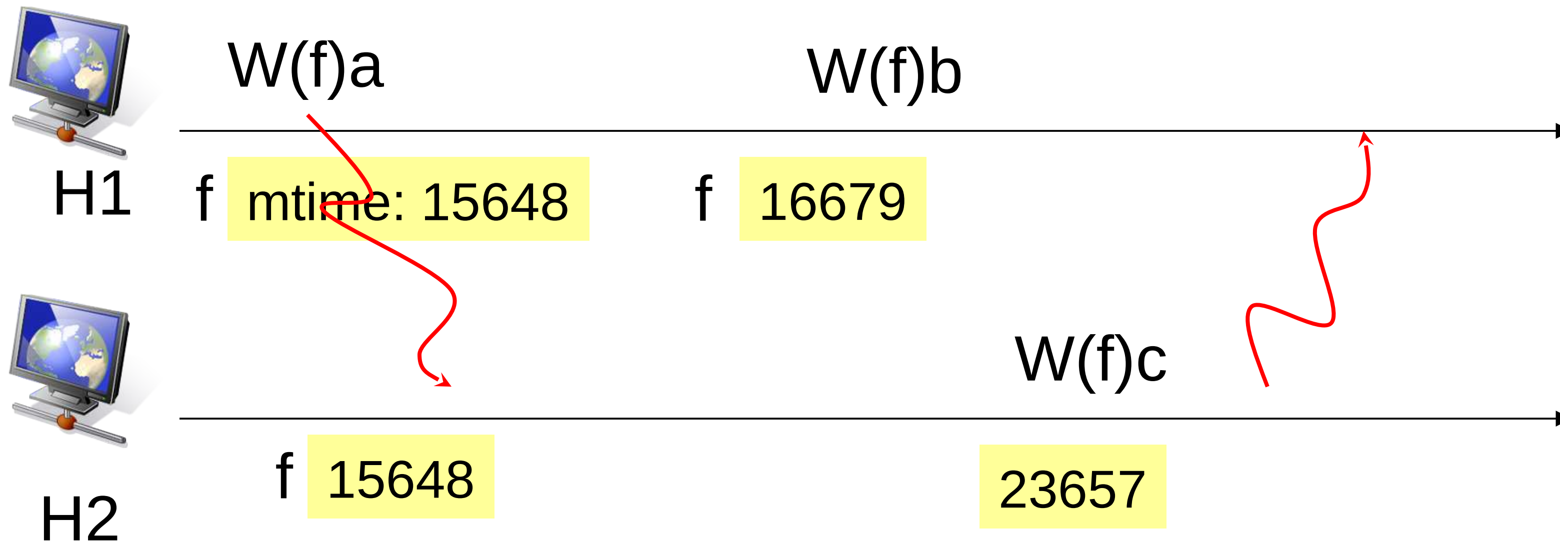
Pair-wise Synchronization



Prevent lost updates

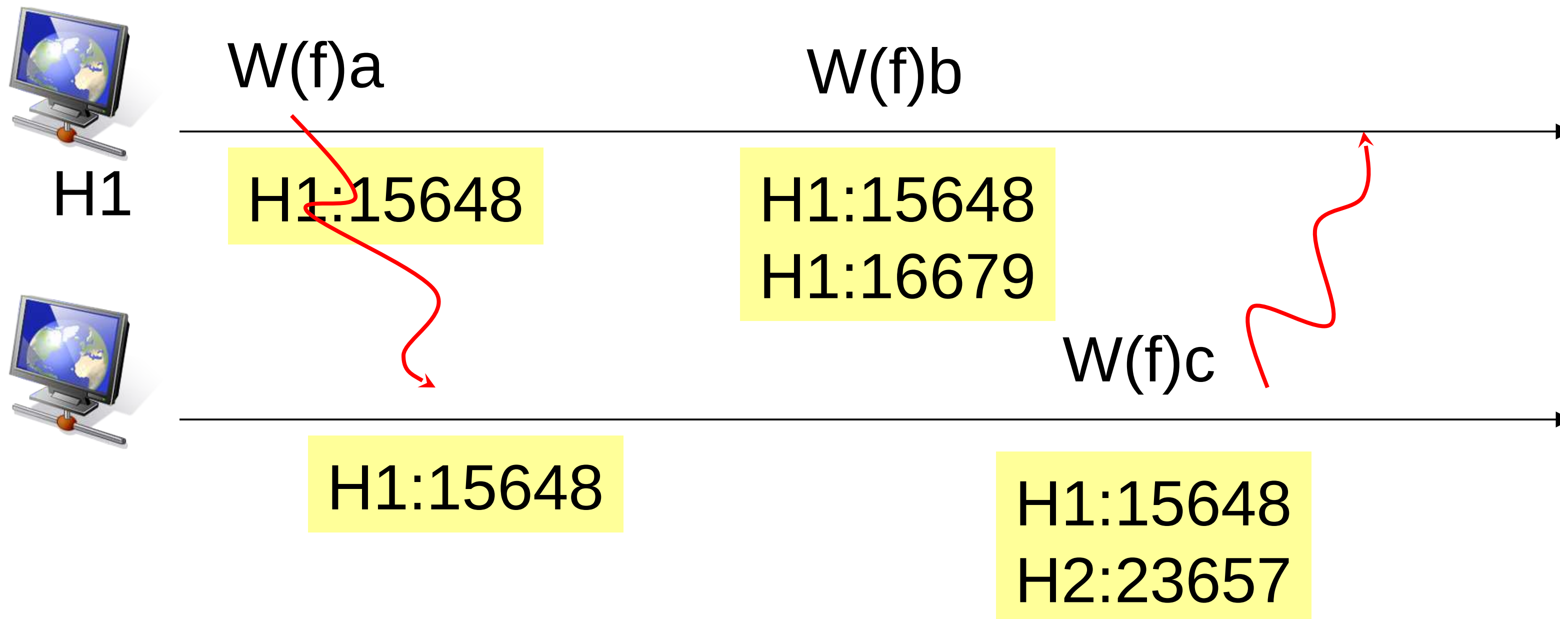
- Detect if updates were sequential
  - If so, replace old version with new one
  - If not, detect conflict

How to Prevent Lost Updates?



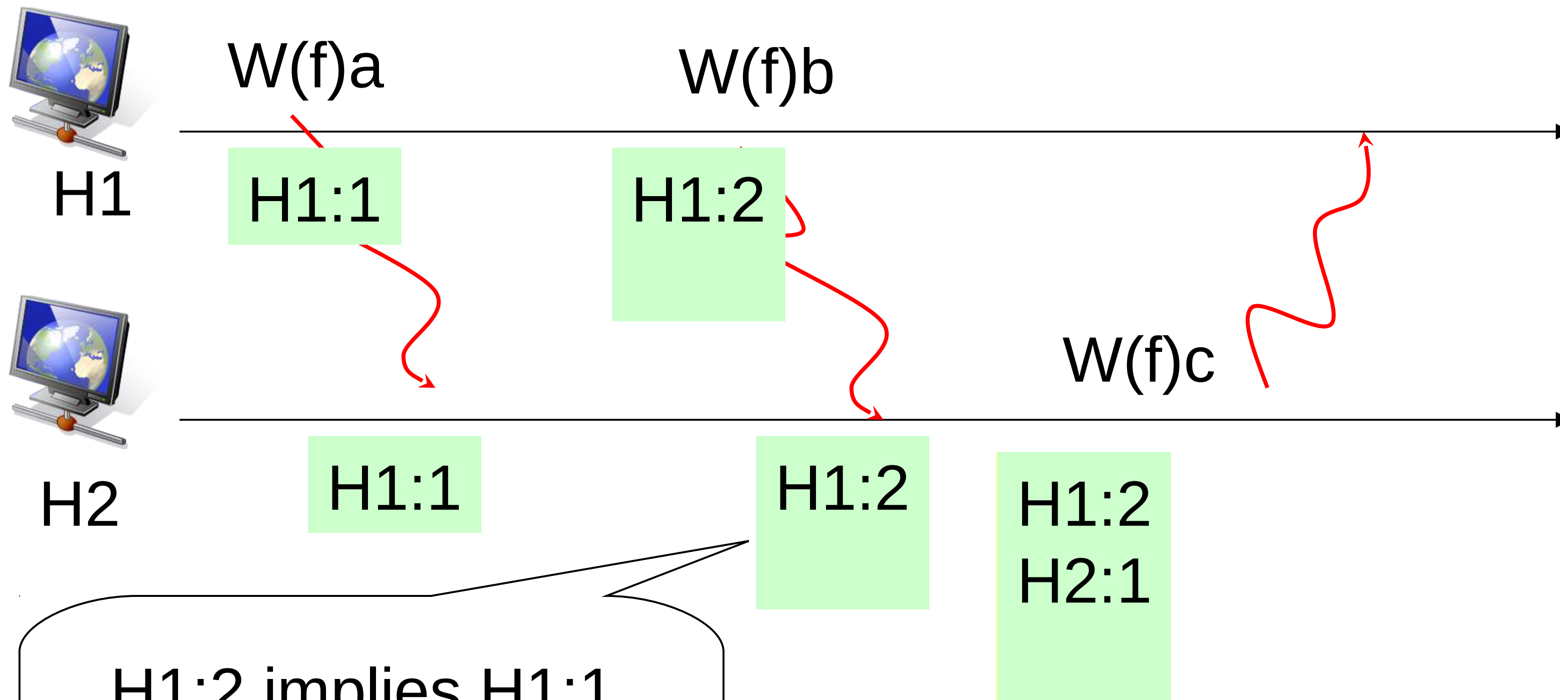
- Strawman: use mtime to decide which version should replace the other
- Problems?
  1. If clocks are unsynchronized: new data might have older timestamp than old data
  2. Does not detect conflicts => may lose some contacts…

Strawman Fix



- Carry the entire modification history (a log)
- If history X is a prefix of Y, Y is newer
- If it’s not, then detect and potentially solve conflicts

Compress Version History



How to Deal w/ Conflicts?

- Easy: mailboxes w/ two different set of messages
- Medium: changes to different lines of a C source file
- Hard: changes to same line of a C source file

- After conflict resolution, add a new item to the history?

So, What's Used Where?

- Strict consistency
  - Google's just presented Spanner at last week's OSDI '12 conference, which looks similar to strict consistency (they call it "external consistency")
  - EXCITING: thus far thought of as impossible
- Sequential consistency
  - A number of both academic and industrial systems provide (at least) sequential consistency (some a bit stronger – linearizability)
  - Examples: Yale's IVY DSM, Microsoft's Niobe DFS, Cornell's chain replication, …
- Causal consistency – dunno
- Eventual consistency
  - Very popular for a while both in industry and in academia
  - Examples: file synchronizers, Amazon's Dynamo, Bayou

Many Other Consistency Models Exist

- Other standard consistency models
  - Linearizability
  - Serializability
  - Monotonic reads
  - Monotonic writes
  - … read Tanenbaum 7.3 if interested (these are not required for exam)

- In-house consistency models:
  - AFS’s close-to-open
  - GFS’s atomic at-most-once appends