Consistency Models

Lecture 7

What Is Consistency?

- What is consistency?
  - What processes can expect when RD/WR shared data concurrently

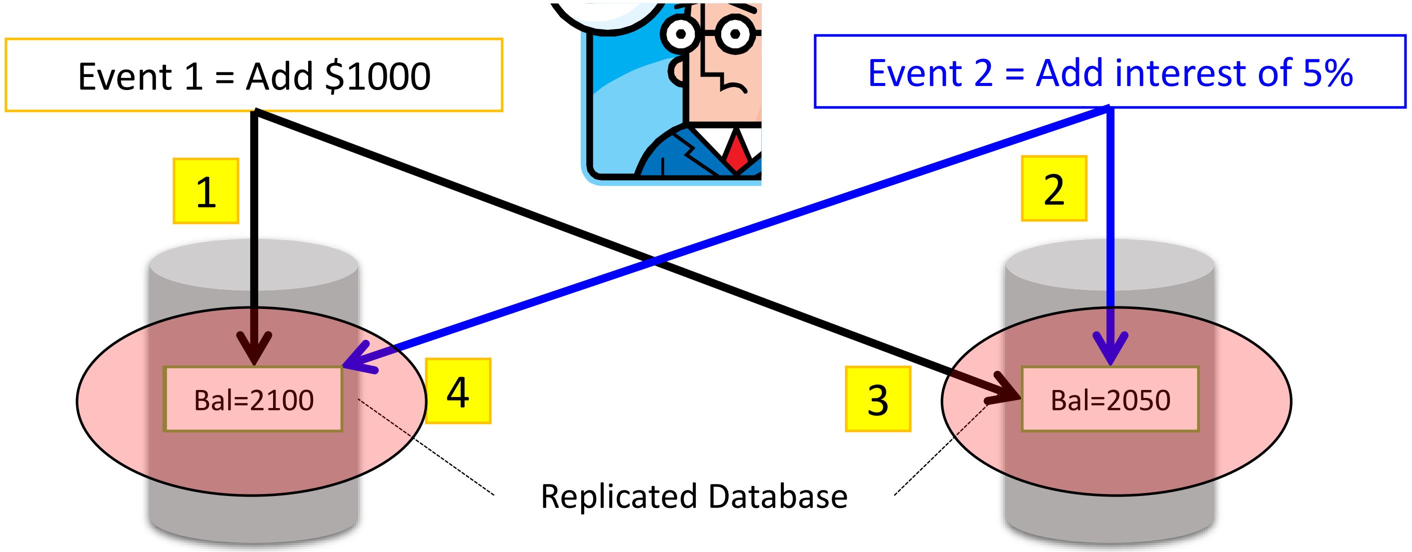
- When do consistency concerns arise?
  - With replication and caching

Why Replication?

- Replication is the process of maintaining the data at multiple computers
- Replication is necessary for:
  1. Improving performance
     - A client can access the replicated copy of the data that is near to its location
  2. Increasing the availability of services
     - Replication can mask failures such as server crashes and network disconnection
  3. Enhancing the scalability of the system
     - Requests to the data can be distributed to many servers which contain replicated copies of the data
  4. Securing against malicious attacks
     - Even if some replicas are malicious, secure data can be guaranteed to the client by relying on the replicated copies at the non-compromised servers

Why Consistency?

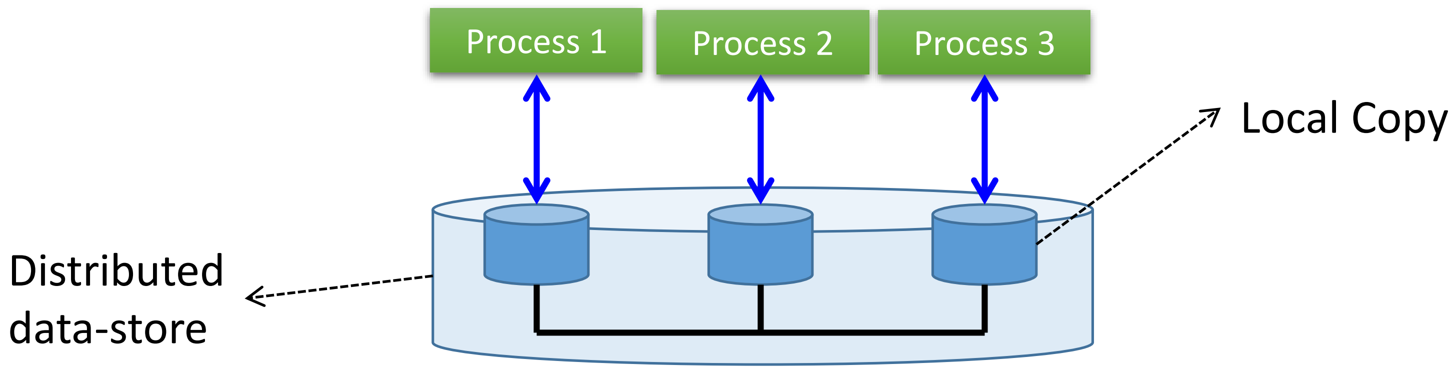
- In a distributed system with replicated data, one of the main problems is keeping the data consistent
- An example:
  - In an e-commerce application, the bank database has been replicated across two servers
  - Maintaining consistency of replicated data is a challenge



Introduction to Consistency and Replication

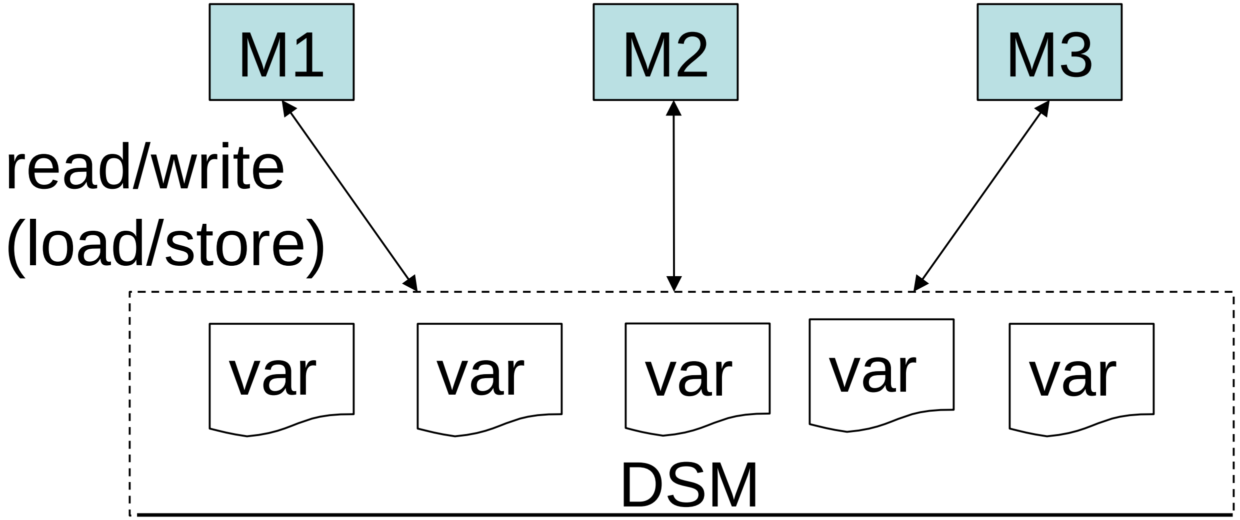
- In a distributed system, shared data is typically stored in distributed shared memory, distributed databases or distributed file systems.
  - The storage can be distributed across multiple computers
  - Simply, we refer to a series of such data storage units as *data-stores*

- Multiple processes can access shared data by accessing any replica on the data-store
  - Processes generally perform read and write operations on the replicas

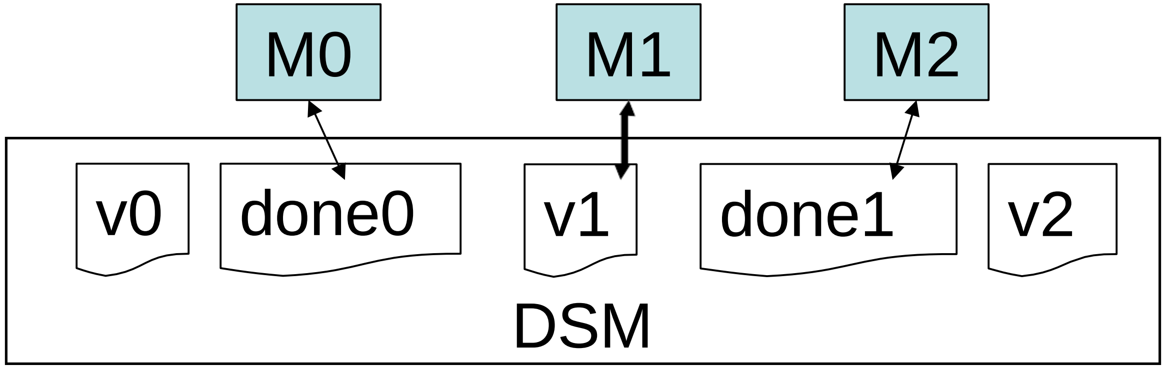


Distributed Shared Memory (DSM)

- Two models for communication in distributed systems:
  - message passing
  - shared memory
- Shared memory is often thought more intuitive to write parallel programs than message passing
  - Each machine can access a common address space



Example Application



M0:

```
v0 = f0();
done0 = 1;
```

M1:

```
while (done0 == 0)
    ;
v1 = f1(v0);
done1 = 1;
```

M2:

```
while (done1 == 0)
    ;
v2 = f2(v0, v1);
```

What's the intuitive intent?

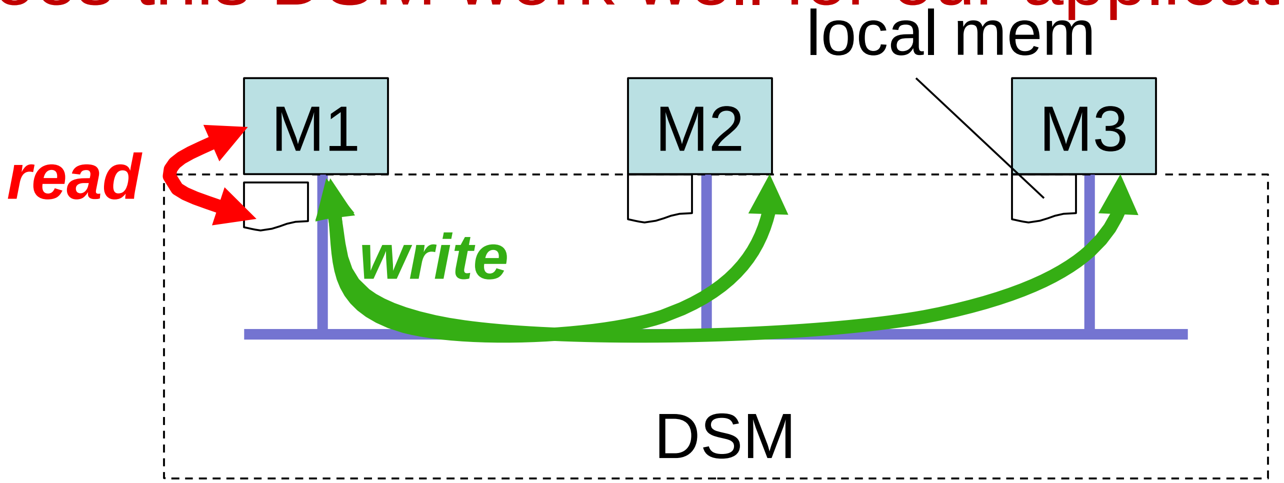
- M2 should execute f2() with results from M0 and M1
- waiting for M1 implies waiting for M0

Naïve DSM System

- Each machine has a local copy of all of memory

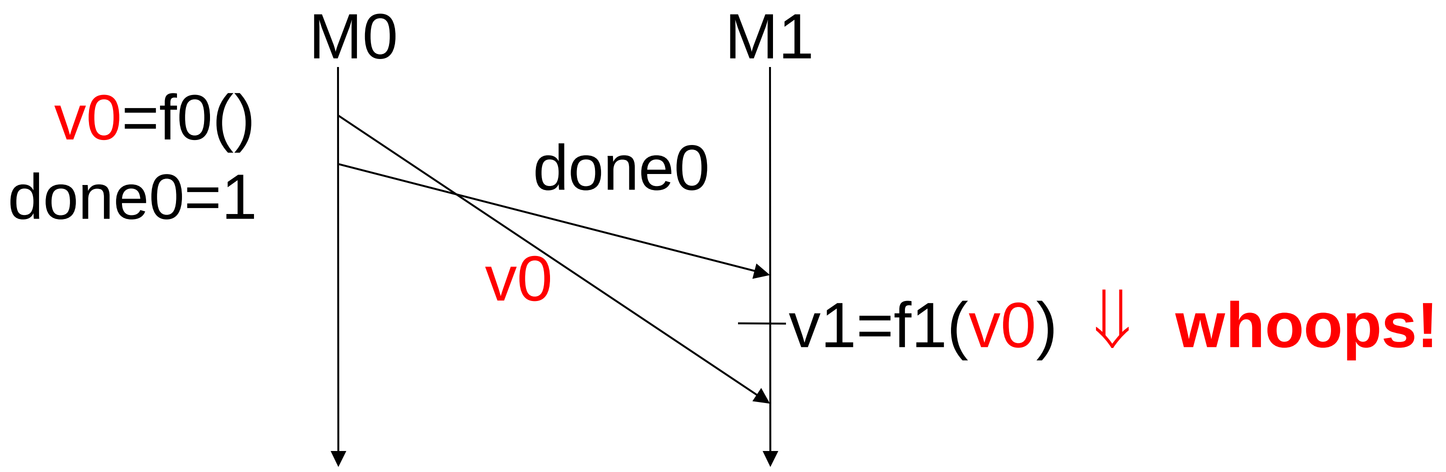
  Operations:

  - **Read**: from local memory
  - **Write**: send update msg to each host (but **don't wait**)
- Fast: never waits for communication
- Question: Does this DSM work well for our application?



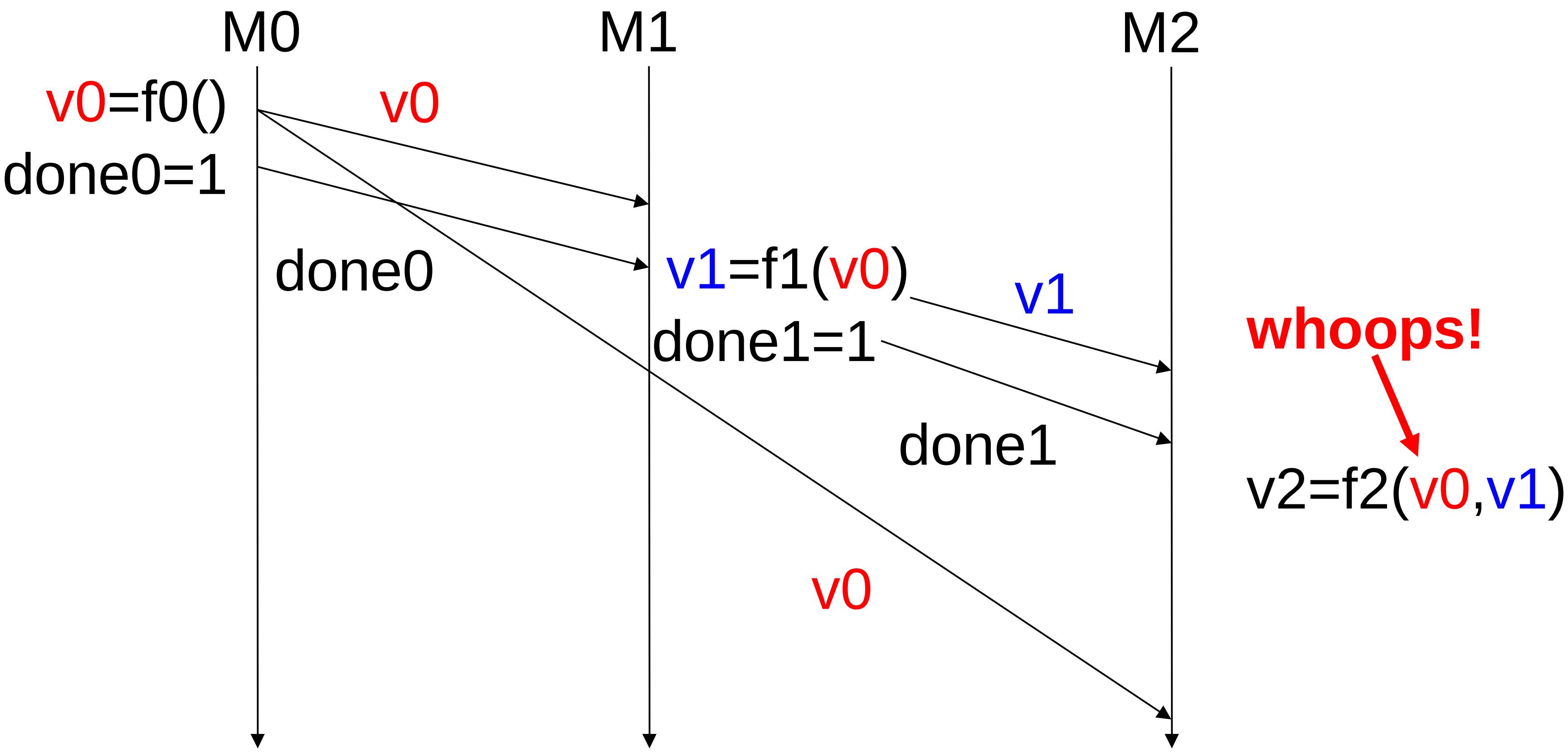
Problem 1 with Naïve DSM

- M0's v0=… and done0=… may be interchanged by network, leaving v0 unset but done0=1



Problem 2 with Naïve DSM

- M2 sees M1's writes before M0's writes
  - I.e. M2 and M1 disagree on order of M0 and M1 writes



Naive DSM is fast but has unexpected behavior

Consistency Models

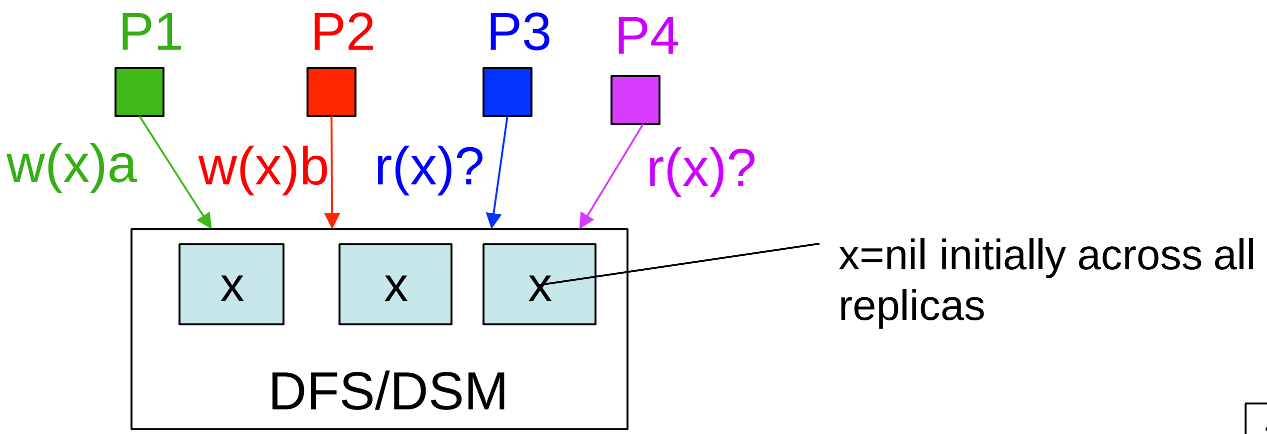
- What is a consistency model?
  - Contract between a distributed data system (e.g., DFS, DSM) and processes constituting its applications
  - E.g.: “If a process reads a certain piece of data, I (the DFS/DSM) pledge to return the value of the last write”

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

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

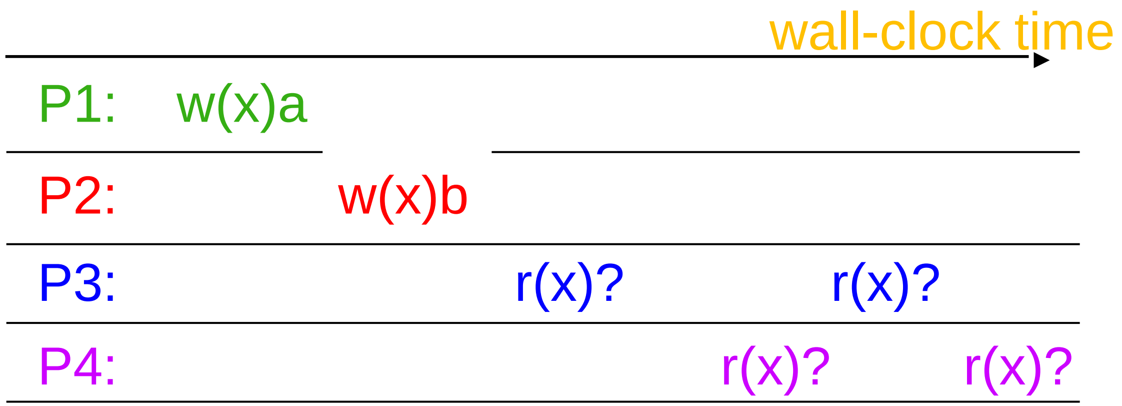
- Variations boil down to:
  - The allowable staleness of reads
  - The ordering of writes across all replicas

Example



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

Time at which client process issues op



May differ from the time at which the op request gets to relevant replica!

Strict Consistency

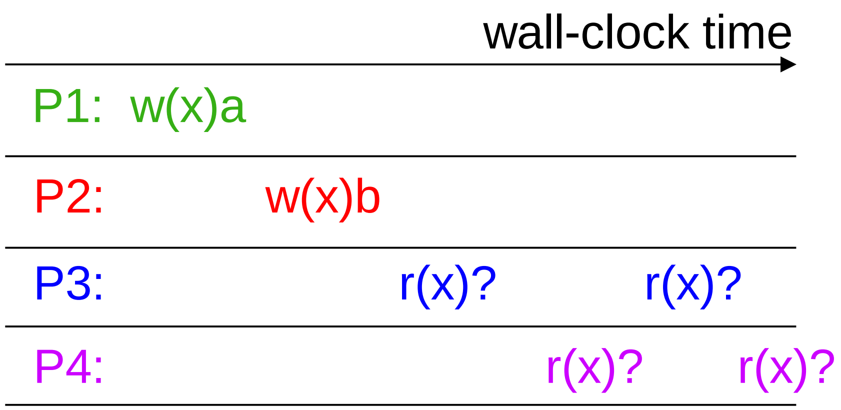
- Each operation is stamped with a global wall-clock time

- Any execution is the same as if all read/write ops were executed in order of wall-clock time at which they were issued

- Rules:
  - Rule 1: Each read gets the latest written value
  - Rule 2: All operations at one CPU are executed in order of their timestamps

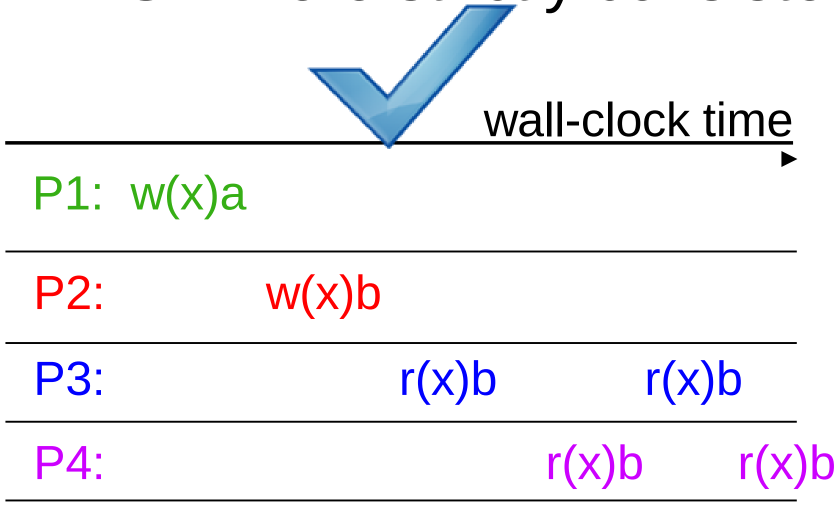
Does Strict Consistency Avoid Problems?

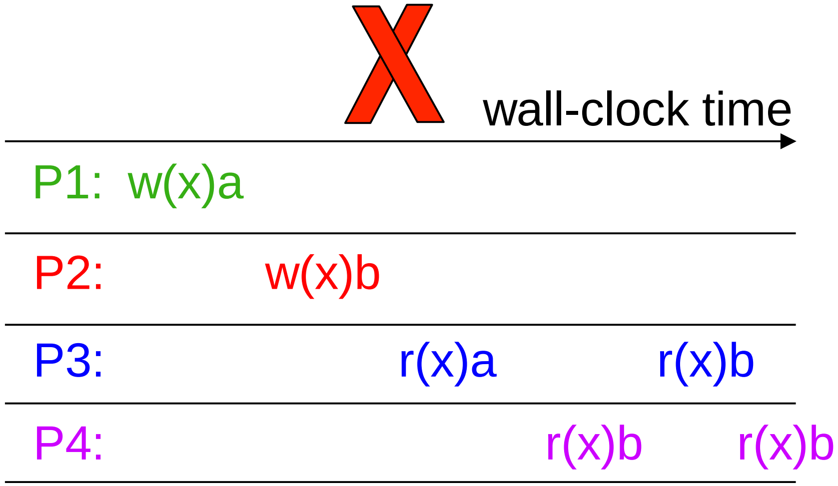
- Suppose we implement rules, can we still get problems?
  - Rule 1: Each read gets the latest written value
    - Reads are never stale
  - Rule 2: All operations at one CPU are executed in order of their timestamps
    - All replicas enforce wall-clock ordering for all writes
  - If DSM were strictly consistent, what can these reads return?



Does Strict Consistency Avoid Problems?

- Suppose we implement rules, can we still get problems?
  - Rule 1: Each read gets the latest written value
    - Reads are never stale
  - Rule 2: All operations at one CPU are executed in order of their timestamps
    - All replicas enforce wall-clock ordering for all writes
  - If DSM were strictly consistent, what can these reads return?





Does Strict Consistency Avoid Problems?

- Suppose we implement rules, can we still get problems?
  - Rule 1: Each read gets the latest written value
    - Reads are never stale
  - Rule 2: All operations at one CPU are executed in order of their timestamps
    - All replicas enforce wall-clock ordering for all writes

- So, strict consistency has very intuitive behavior
  - Essentially, the same semantic as on a uniprocessor!
- But how to implement it efficiently?
  - Without reducing distributed system to a uniprocessor…

Implementing Strict Consistency

					time
	1	2	3		
M0:	WR			WR	
M1:		RD	RD		

- To achieve, one would need to ensure:
  - Each read must be aware of, and wait for, each write
    - RD@2 aware of WR@1; WR@4 must know how long to wait…
  - Real-time clocks are strictly synchronized…
- Unfortunately:
  - Time between instructions << speed-of-light…
  - Real-clock synchronization is tough

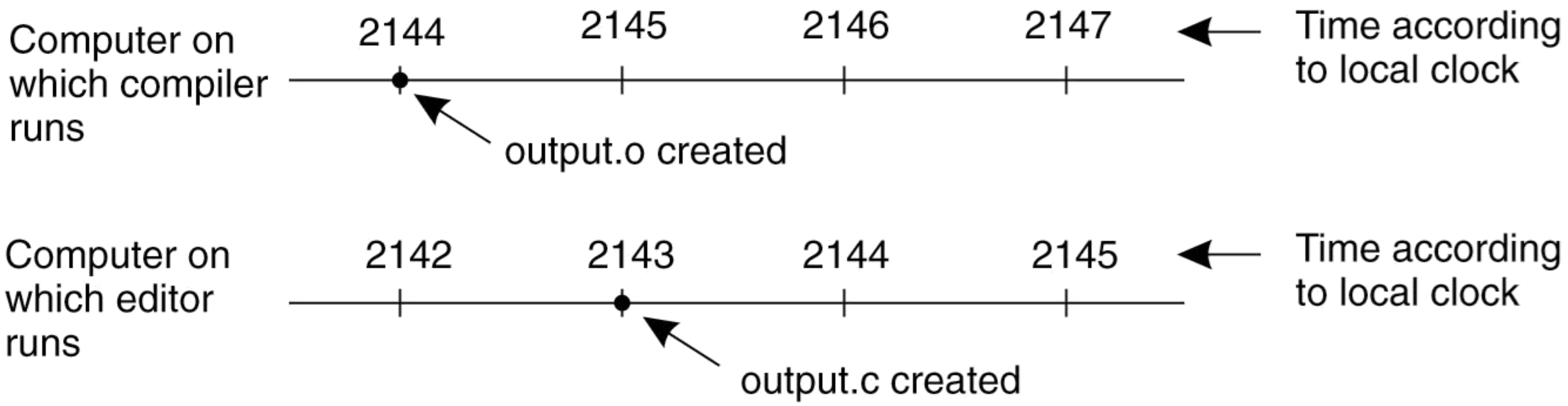
Clocks in Distributed System

- Computer clocks are not generally in perfect agreement
  - **Skew**: the difference between the times on two clocks (at any instant)

- Computer clocks are subject to clock drift (they count time at different rates)
  - **Clock drift rate**: the difference per unit of time from some ideal reference clock
  - Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).
  - High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

Impact of Clock Synchronization

- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time



- Need globally consistent time standard
  - Who got last seat on airplane?
  - Who submitted final auction bid before deadline?

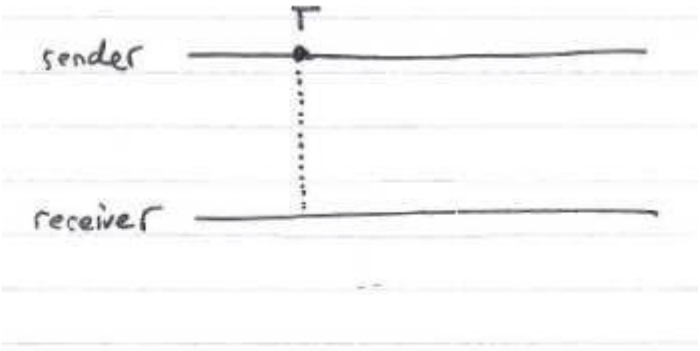
Coordinated Universal Time (UTC)

- Is broadcast from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1-10 millisecond
- Signals from GPS are accurate to about 1 microsecond

Clock synchronization: The problems

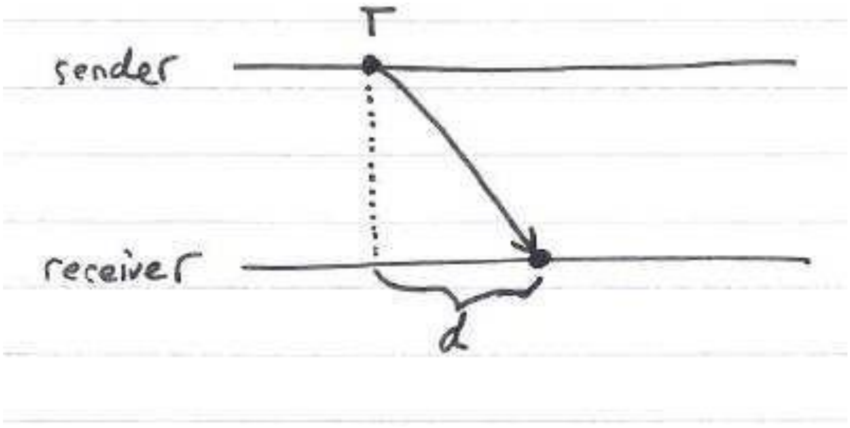
- Suppose I want to synchronize the clocks on two machines (M1 and M2)

- One solution:
  - M1 (sender) sends its own time T in message to M2
  - M2 (receiver) sets its time according to the message
  - But what time should M2 set?



Perfect Networks

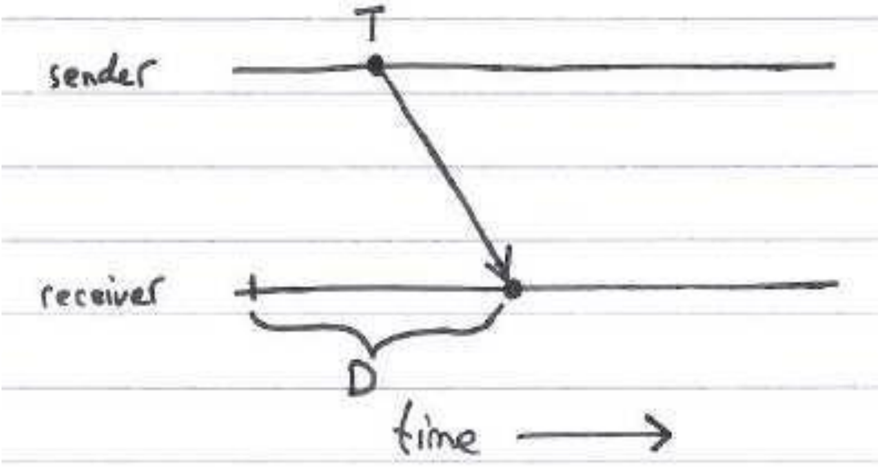
- Messages always arrive, with propagation delay exactly *d*



- Sender sends time *T* in a message Receiver sets clock to *T+d*
  - Synchronization is exact

Synchronous networks

- Messages always arrive, with propagation delay *at most D*



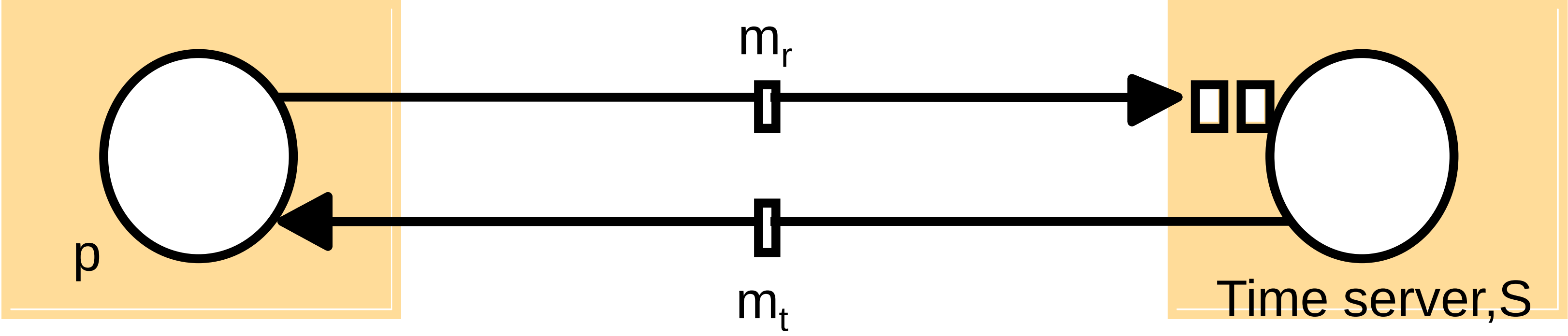
- Sender sends time *T* in a message
- Receiver sets clock to *T + D/2*
  - Synchronization error is at most *D/2*

Synchronization in the Real World

- Real networks are asynchronous
  - Propagation delays are arbitrary
- Real networks are unreliable
  - Messages don’t always arrive

Cristian's Time Sync

- A time server S receives signals from a UTC source
  - Process p requests time in m_r and receives t in m_t from S
  - p sets its clock to $t + T_{round}/2$
  - Accuracy $\pm (T_{round}/2 - min)$:
    - because the earliest time S puts t in message m_t is min after p sent m_r.
    - the latest time is min before m_t arrived at p
    - the time by S's clock when m_t arrives is in the range $[t+min, t + T_{round} - min]$



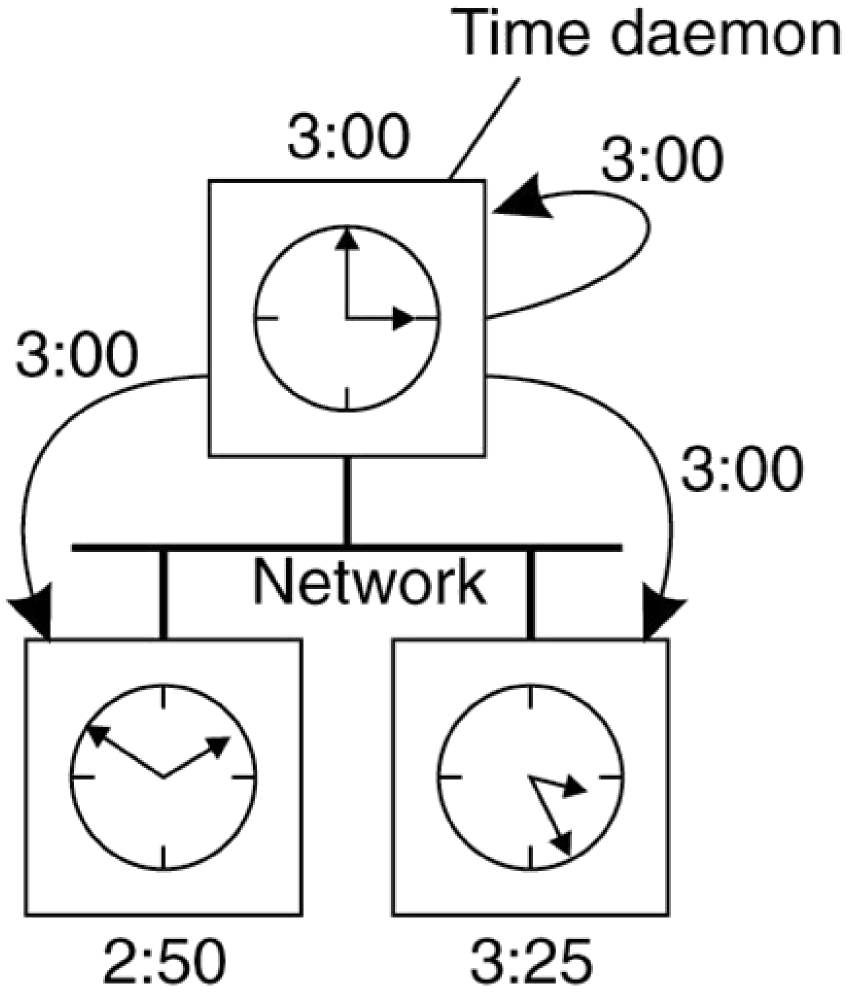
T_{round} is the round trip time recorded by p
min is an estimated minimum one-way transmit time

The Berkeley Algorithm

- An algorithm for internal synchronization of a group of computers
- A *master* polls to collect clock values from the others (*slaves*)
- The master uses round trip times to estimate the slaves' clock values
- It takes an average (eliminating any above average round trip time or with faulty clocks)
- It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)

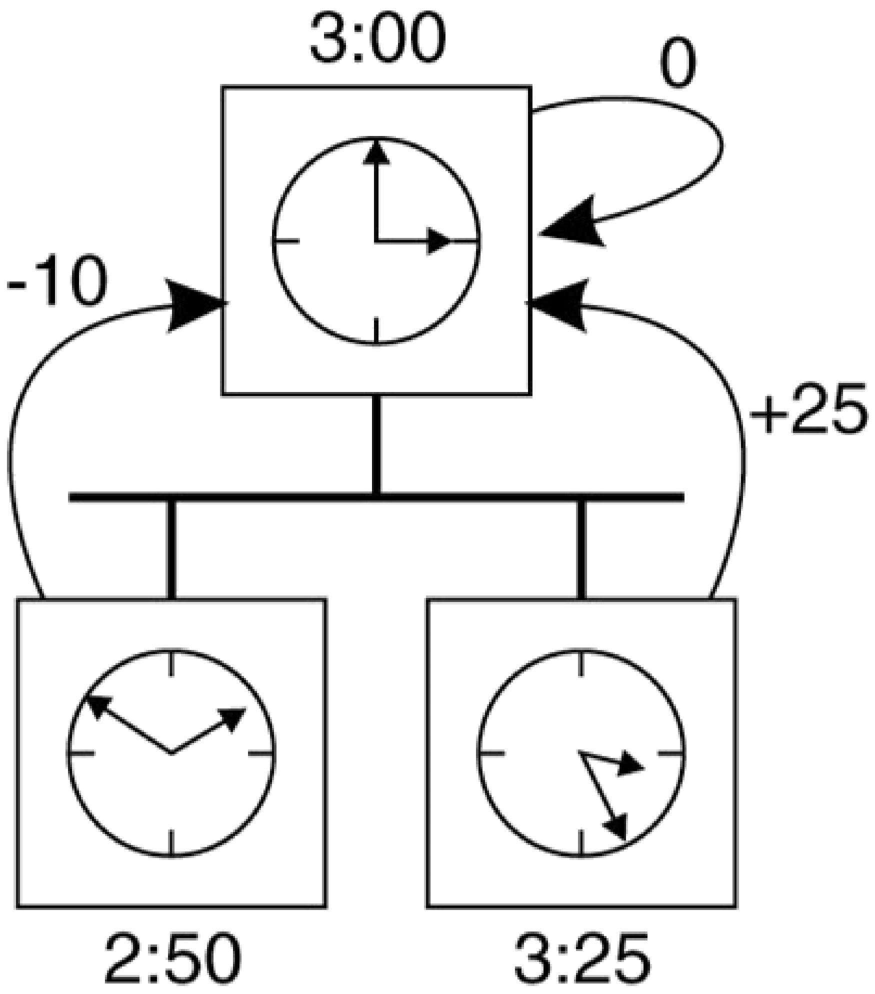
The Berkeley Algorithm (1)

- The time daemon asks all the other machines for their clock values.



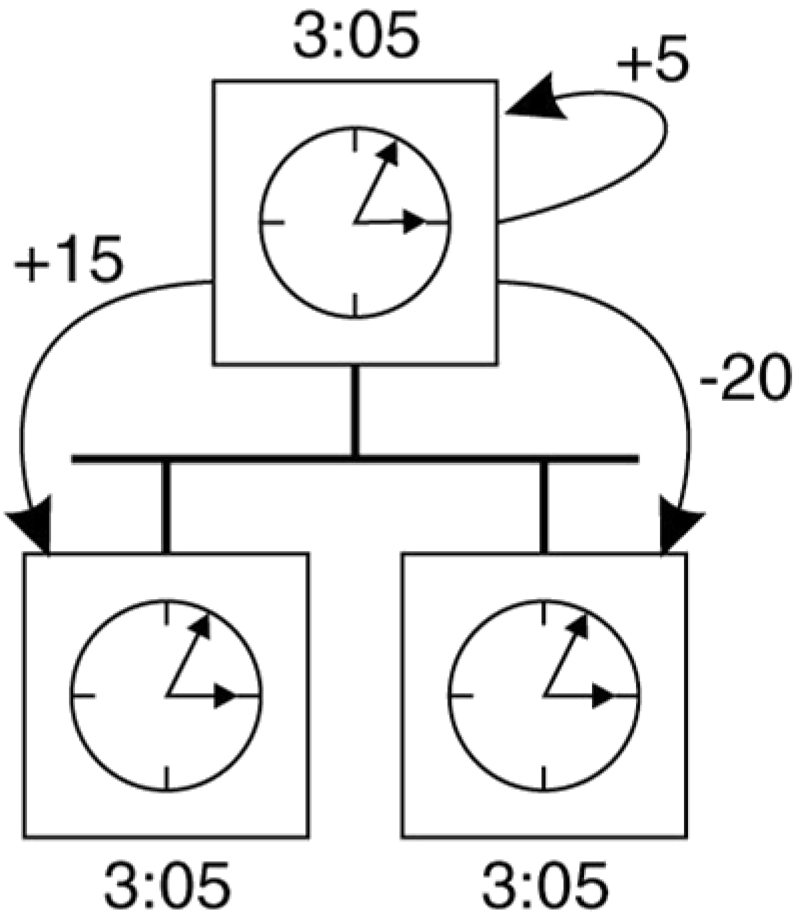
The Berkeley Algorithm (2)

- The machines answer.



The Berkeley Algorithm (3)

- The time daemon tells everyone how to adjust their clock.



The Network Time Protocol (NTP)

- Uses a hierarchy of time servers
  - Class 1 servers have accurate (and expensive) clocks
    - connected directly to atomic clocks or GPS receivers
  - Class 2 servers get time from Class 1 and Class 2 servers
  - Class 3 servers get time from any server
  - Client machines (e.g., your smartphones, laptops, desktops, or server machines) synchronize w/ time servers
- Synchronization similar to Cristian's alg.
- Accuracy: Local ~1ms, Global ~10ms

Important Lessons

- Clocks on different systems will always behave differently
  - Skew and drift between clocks
- Time disagreement between machines can result in undesirable behavior
- Clock synchronization
  - Rely on a time-stamped network messages
  - Estimate delay for message transmission
  - Can synchronize to UTC or to local source
  - Clocks never exactly synchronized

Back to Strict Consistency

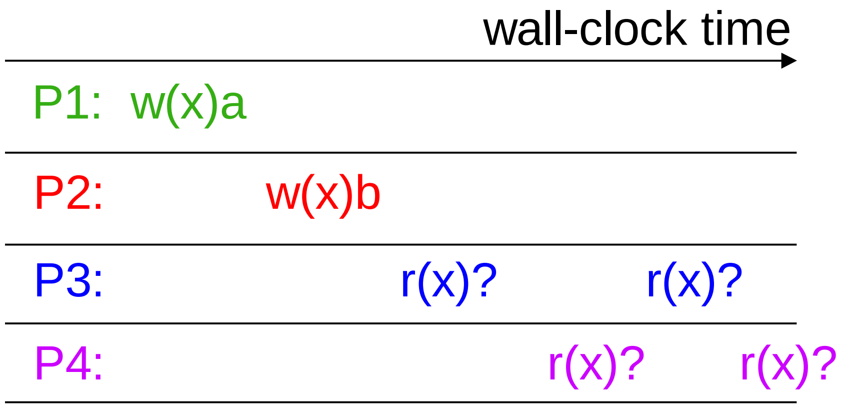
- To achieve strict consistency
  - Each read must be aware of, and wait for, each write
  - Real-time clocks must be strictly synchronized
- Unfortunately:
  - Clocks are never *exactly* synchronized. Often inadequate for distributed systems.
  - Might need totally-ordered events
  - Might need millionth-of-a-second precision
- So, strict consistency is tough to implement efficiently

Model 2: Sequential Consistency

- Slightly weaker model than strict consistency
  - Most important difference: doesn't assume real time
- Rules: There exists a total ordering of ops such that
  - – Rule 1: Each machine's own ops appear in order
  - – Rule 2: All machines see results according to total order
- We say that any runtime ordering of operations can be "explained" by a sequential ordering of operations that follows the above two rules

Sequential Consistency

- 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

- 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

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

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

wall-clock ordering

This was also strictly consistent

wall-clock time

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

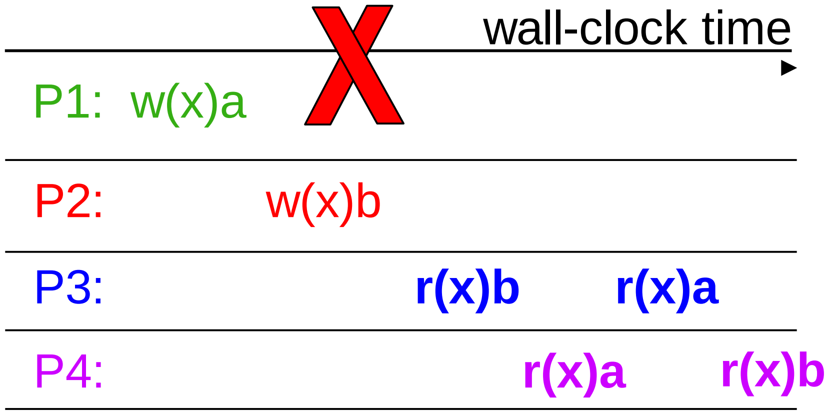
What's a global sequential order that can explain these results?

w(x)a, r(x)a, w(x)b, r(x)b, …

This wasn't strictly consistent

Sequential Consistency

- 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

Sequential Consistency: Implementation

- Easier to implement than strict consistency
  - No notion of real time
  - System can interleaves different machines' ops (not forced to order by op start time, as in strict consistency)

- Each processor issues requests in the order specified by the program
  - Do not issue the next request unless the previous one has finished

- Requests to an individual memory location (storage object) are served from a single FIFO queue.
  - Writes occur in a single order
  - Once a read observes the effect of a write, it’s ordered behind that write

Sequential Consistency: Efficiency

- Performance is still not great
  - Once a machine's write completes, other machines' reads must see new data
  - Thus communication cannot be omitted or much delayed
  - Thus either reads or writes (or both) will be expensive

Time vs ordering

- What usually matters is not that all processes agree on exactly what time it is, but that they agree on the order in which events occur. Requires a notion of ordering.

- Idea: Capture just the “happens before” relationship between events without worrying about actual time
  - Corresponds roughly to causality

Happens-before: Formally defined

- Definition (→): We define e → e' using the following rules:
  - Local ordering: $e \rightarrow e'$ if $e \rightarrow_i e'$ for any process i
  - Messages: send(m) → receive(m) for any message m
  - Transitivity: $e \rightarrow e''$ if $e \rightarrow e'$ and $e' \rightarrow e''$
- We say *e* "happens before" *e'* if $e \rightarrow e'$
- → is only a partial-order
  - Some events are unrelated
- Definition (concurrency): We say *e* is concurrent with *e'* (written $e \| e'$) if neither $e \rightarrow e'$ nor $e' \rightarrow e$

A Baseball example

- Four locations: pitcher's mound (P), home plate, first base, and third base
- Ten events:

e_1: pitcher (P) throws ball toward home

e_2: ball arrives at home

e_3: batter (B) hits ball toward pitcher

e_4: batter runs toward first base

e_5: runner runs toward home

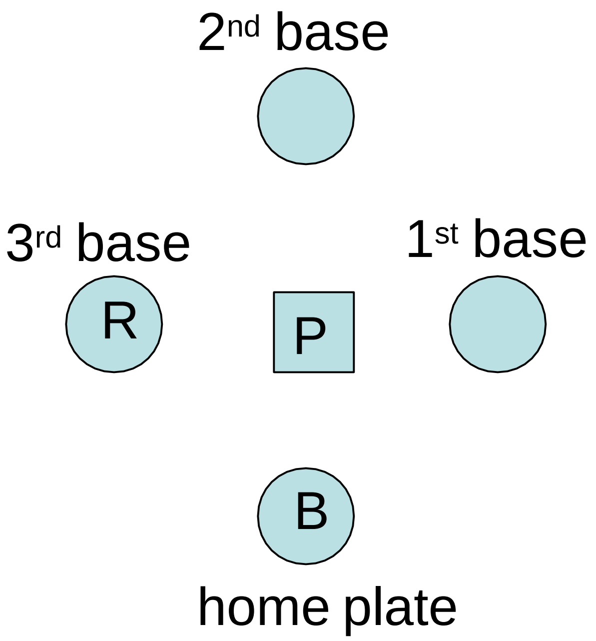
e_6: ball arrives at pitcher

e_7: pitcher throws ball toward first base

e_8: runner arrives at home

e_9: ball arrives at first base

e_{10}: batter arrives at first base



A Baseball example

- $e_1 \rightarrow e_2$
  - by the message rule
- $e_1 \rightarrow e_{10}$, because
  - $e_1 \rightarrow e_2$, by the message rule
  - $e_2 \rightarrow e_4$, by local ordering at home plate
  - $e_4 \rightarrow e_{10}$, by the message rule
  - Repeated transitivity of the above relations
- $e_8 \| e_9$, because
  - No application of the $\rightarrow$ rules yields either $e_8 \rightarrow e_9$ or $e_9 \rightarrow e_8$

e_1: pitcher (P) throws ball toward home
e_2: ball arrives at home
e_3: batter (B) hits ball toward pitcher
e_4: batter runs toward first base
e_5: runner runs toward home
e_6: ball arrives at pitcher
e_7: pitcher throws ball toward first base
e_8: runner arrives at home
e_9: ball arrives at first base
e_{10}: batter arrives at first base

Lamport Logical Clocks

- How do we build a logical clock based on happens-before relationships?
- Attach a timestamp $C(e)$ to each event e, satisfying the following properties:
  - **P1** If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.
  - **P2** If a corresponds to sending message m, and b to the receipt of that message, then also $C(a) < C(b)$.
- Problem
  - How to attach a timestamp to an event when there's no global clock?
  - Idea: Maintain a consistent set of logical clocks, one per process.

Lamport's Algorithm

- Each process P_i maintains a local counter C_i and adjusts it
  1. For each new event that takes place within P_i, C_i is incremented by 1.
  2. Each time a message *m* is sent by process P_i, the message receives a timestamp $ts(m) = C_i$.
  3. Whenever a message *m* is received by a process P_j, P_j adjusts its local counter C_j to $max\ \{ C_j; ts(m) \};$ then executes step 1 before passing *m* to the application.

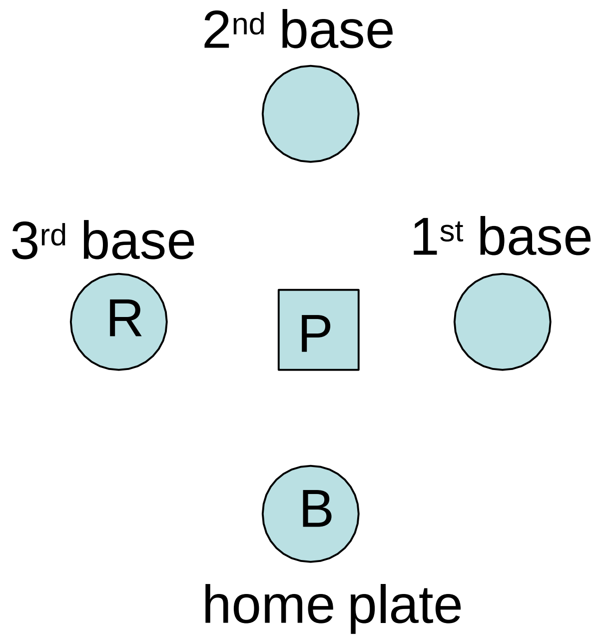
- Note:
  - Property P1 is satisfied by (1)
  - Property P2 by (2) and (3).

- With logical clocks, we have a way to track and order events in a distributed system

Lamport on the baseball example

- Initializing each local clock to 0, we get

  $C(e_1) = 1$ (pitcher throws ball to home)
  $C(e_2) = 2$ (ball arrives at home)
  $C(e_3) = 3$ (batter hits ball to pitcher)
  $C(e_4) = 4$ (batter runs to 1st base)
  $C(e_5) = 1$ (runner runs to home from 3rd base)
  $C(e_6) = 4$ (ball arrives at pitcher)
  $C(e_7) = 5$ (pitcher throws ball to 1st base)
  $C(e_8) = 5$ (runner arrives at home)
  $C(e_9) = 6$ (ball arrives at 1st base)
  $C(e_{10}) = 7$ (batter arrives at 1st base)



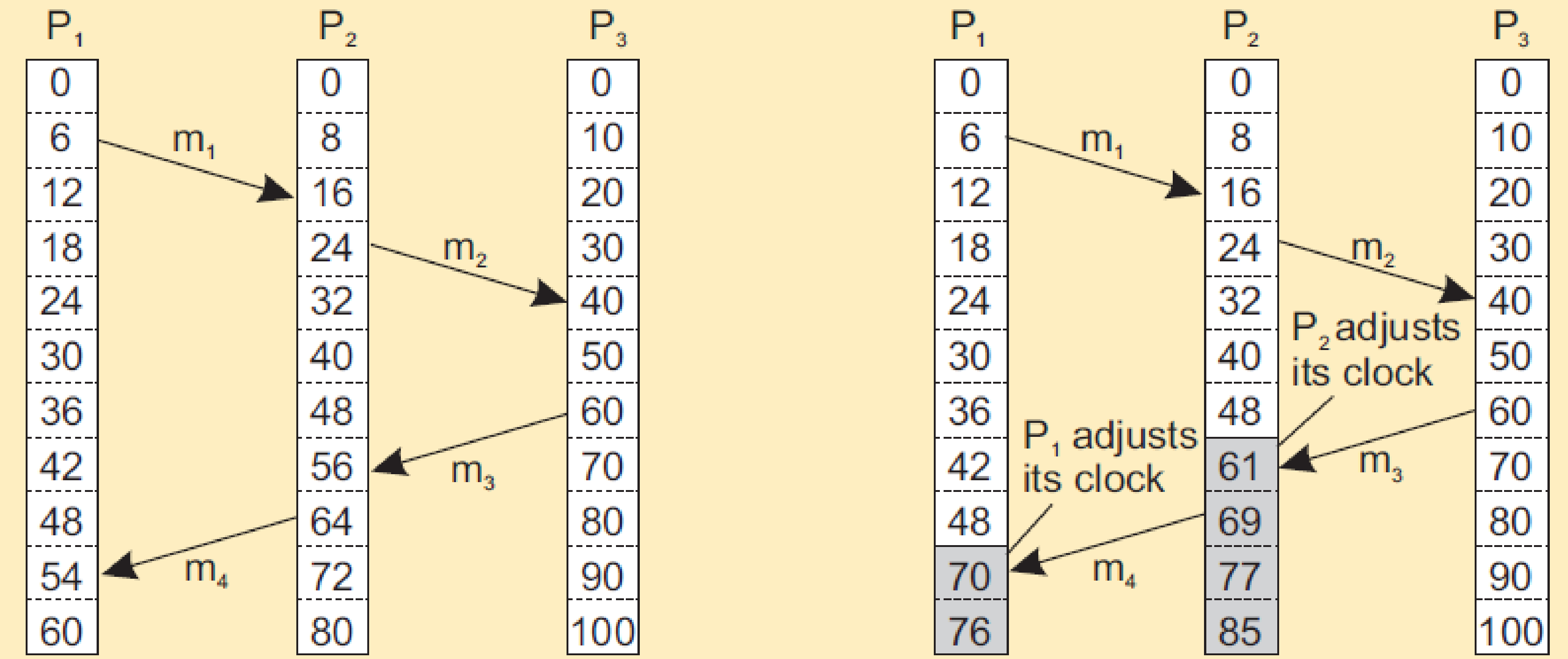
- For our example, Lamport's algorithm says that the run scores!

Lamport Logical Clocks: Note

- Lamport clock C assigns logical timestamps to events consistent with "happens before" ordering
  - If $e \rightarrow e'$, then $C(e) < C(e')$
- But not the converse
  - $C(e) < C(e')$ does not imply $e \rightarrow e'$
- Similar rules for concurrency
  - $C(e) = C(e')$ implies $e \| e'$ (for distinct e, e')
  - $e \| e'$ does not imply $C(e) = C(e')$
- i.e., Lamport clocks arbitrarily order some concurrent events

Lamport Clocks: Another example

Consider three processes with event counters operating at different rates

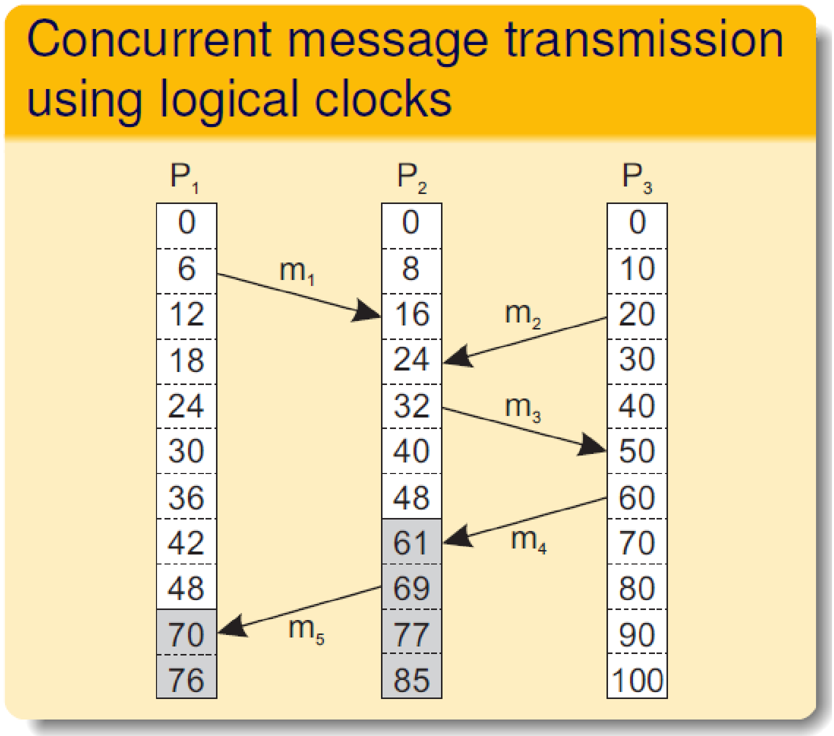


Vector Clocks

- Observation
  - Lamport's clocks do not guarantee that if C(a) < C(b) that *a* causally preceded *b*.

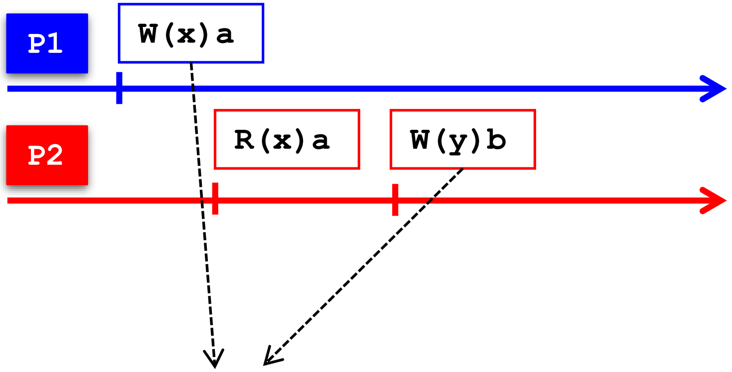
- Observation
  - Event *a*: m1 is received at T = 16;
  - Event *b*: m2 is sent at T = 20.

- We cannot conclude that *a* causally precedes *b*



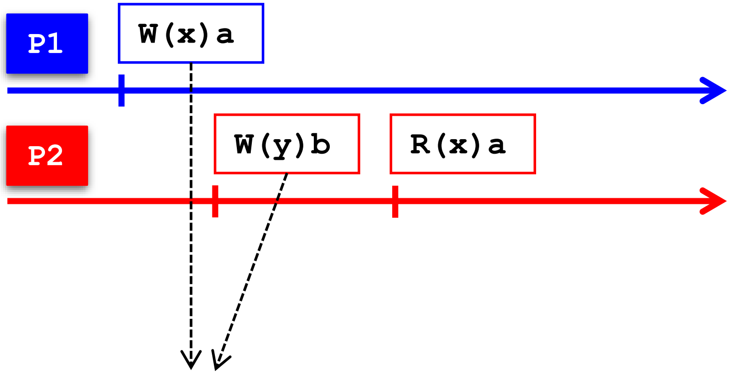
Causal relationship

- Consider an interaction between processes $\mathtt{P_1}$ and $\mathtt{P_2}$ operating on replicated data $\mathtt{x}$ and $\mathtt{y}$



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$_1$**

Capturing causality with vector clocks

- Each P_i maintains a vector VC_i
  - $VC_i\ [i]$ is the local logical clock at process P_i.
  - If $VC_i\ [j] = k$ then P_i knows that k events have occurred at P_j.

- Maintaining vector clocks
  1. Before executing an event P_i executes $VC_i = VC_i\ [i]+1$.
  2. When process P_i sends a message m to P_j, it sets m's (vector) timestamp $ts(m)$ equal to VC_i after having executed step 1.
  3. Upon the receipt of a message m, process P_j sets $VC_j\ [k] = max\ \{ VC_j\ [k]; ts(m)[k] \}$ for each k, after which it executes step 1 and then delivers the message to the application

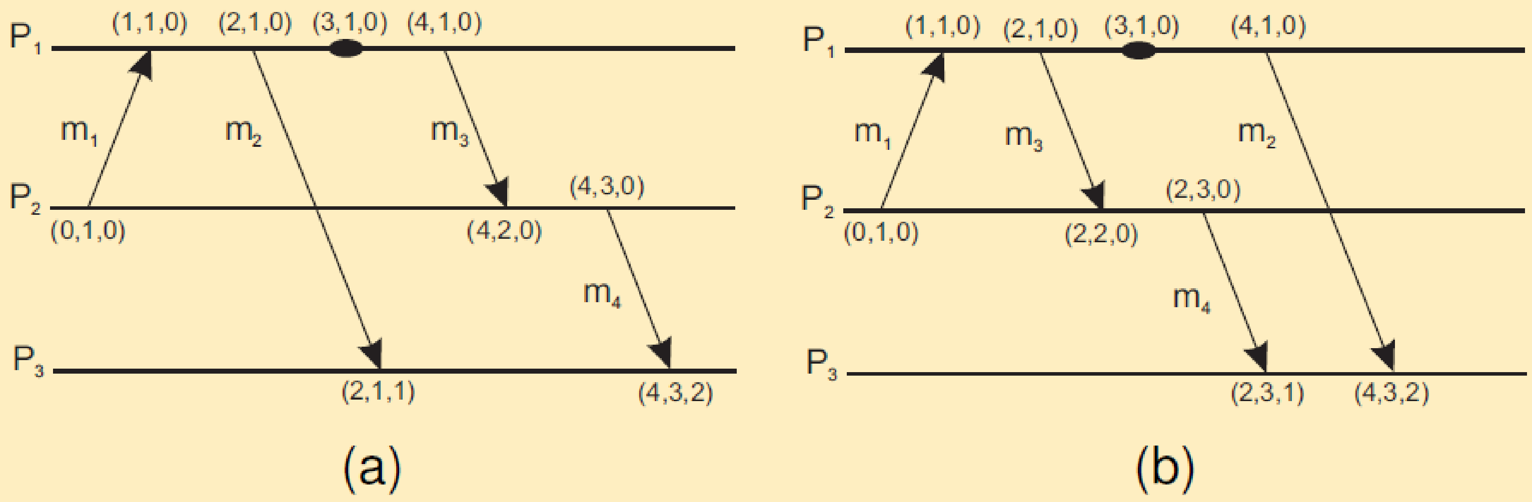
Causal Precedence

- We say that *b* may causally depend on *a* if *ts(a)* < *ts(b)*, with:
  - for all *k*, *ts(a)[k]* <= *ts(b)[k]* and
  - there exists at least one index *k'* for which *ts(a)[k']* < *ts(b)[k']*

- Precedence vs. dependency
  - We say that *a* causally precedes *b*.
  - *b* may causally depend on *a*, as there may be information from *a* that is propagated into *b*.

Vector Clock: Example

Capturing potential causality when exchanging messages



Analysis

Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	**Conclusion**
(a)	$(2,1,0)$	$(4,3,0)$	Yes	No	m_2 may causally precede m_4
(b)	$(4,1,0)$	$(2,3,0)$	No	No	m_2 and m_4 may conflict

Model 3: 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
  - Causally-related writes are ordered by all replicas in the same way
  - Concurrent writes may be committed in different orders by different replicas, and hence read in different orders by different applications
  - Reads are fresh only w.r.t. writes that they are causally dependent on

Causal Consistency: 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

wall-clock time

P1: w(x)a

P2: w(x)b

P3: **r(x)b** **r(x)a**

P4: **r(x)a** **r(x)b**

w(x)a || w(x)b, hence they can be seen in ≠ orders by ≠ processes

This wasn't sequentially consistent.

wall-clock time

P1: w(x)a

P2: r(x)a w(x)b

P3: r(x)b r(x)a

P4:

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
  - Parallel 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

Relaxing consistency further

- More concurrency opportunities than strict, sequential, or causal consistency

- Strong consistency may be unsuitable in certain cases:
  - 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

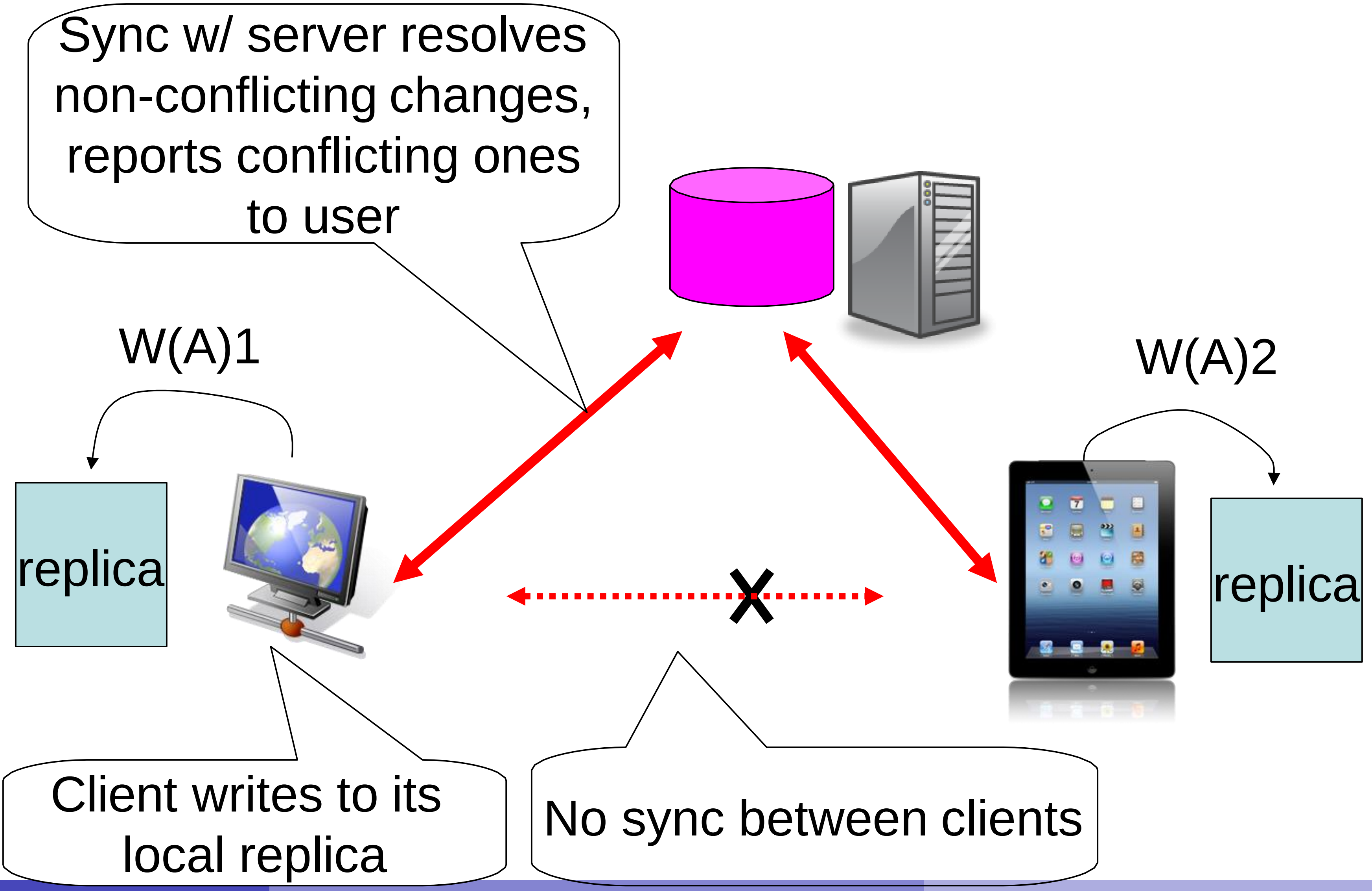
Eventual Consistency

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

- Example: 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
  3. Do not replace new version with old ones

Operating without Total Connectivity

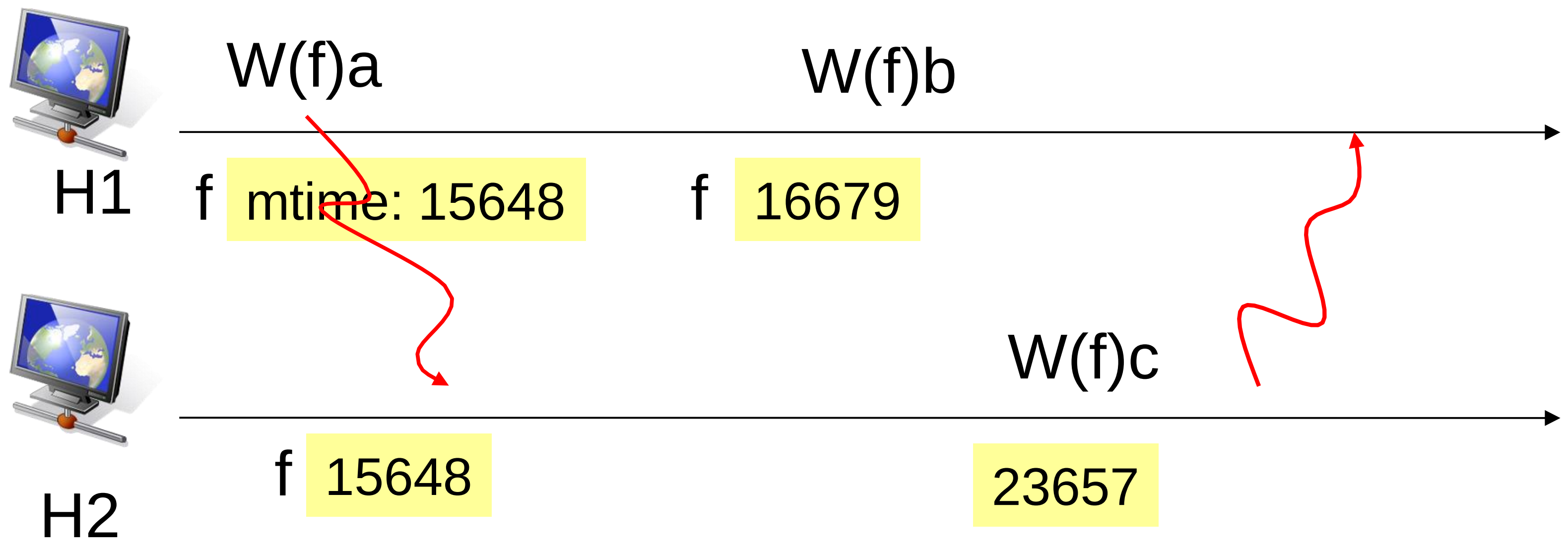


Prevent Lost Updates

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

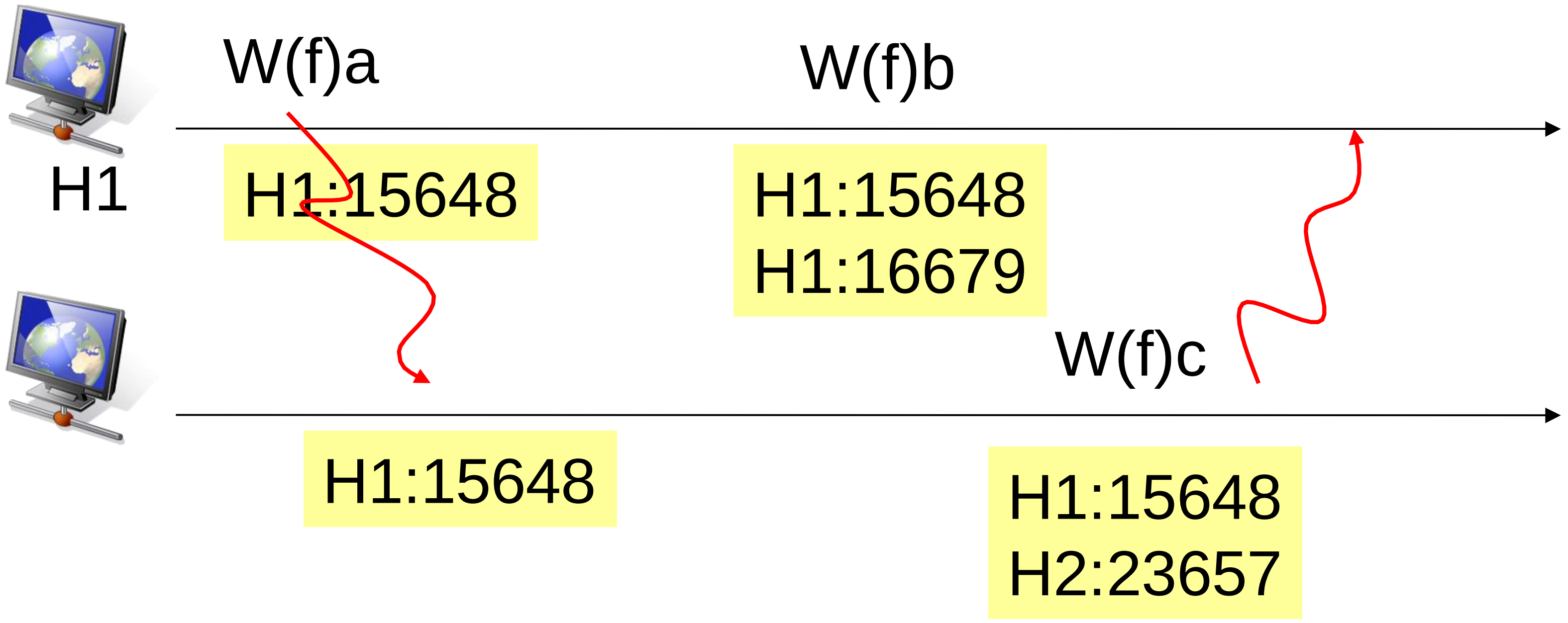
- “Optimistic” vs. “Pessimistic”
  - Eventual Consistency: Let updates happen, worry about whether they can be serialized later
  - Sequential Consistency: Updates cannot take effect unless they are serialized first

How to Prevent Lost Updates?



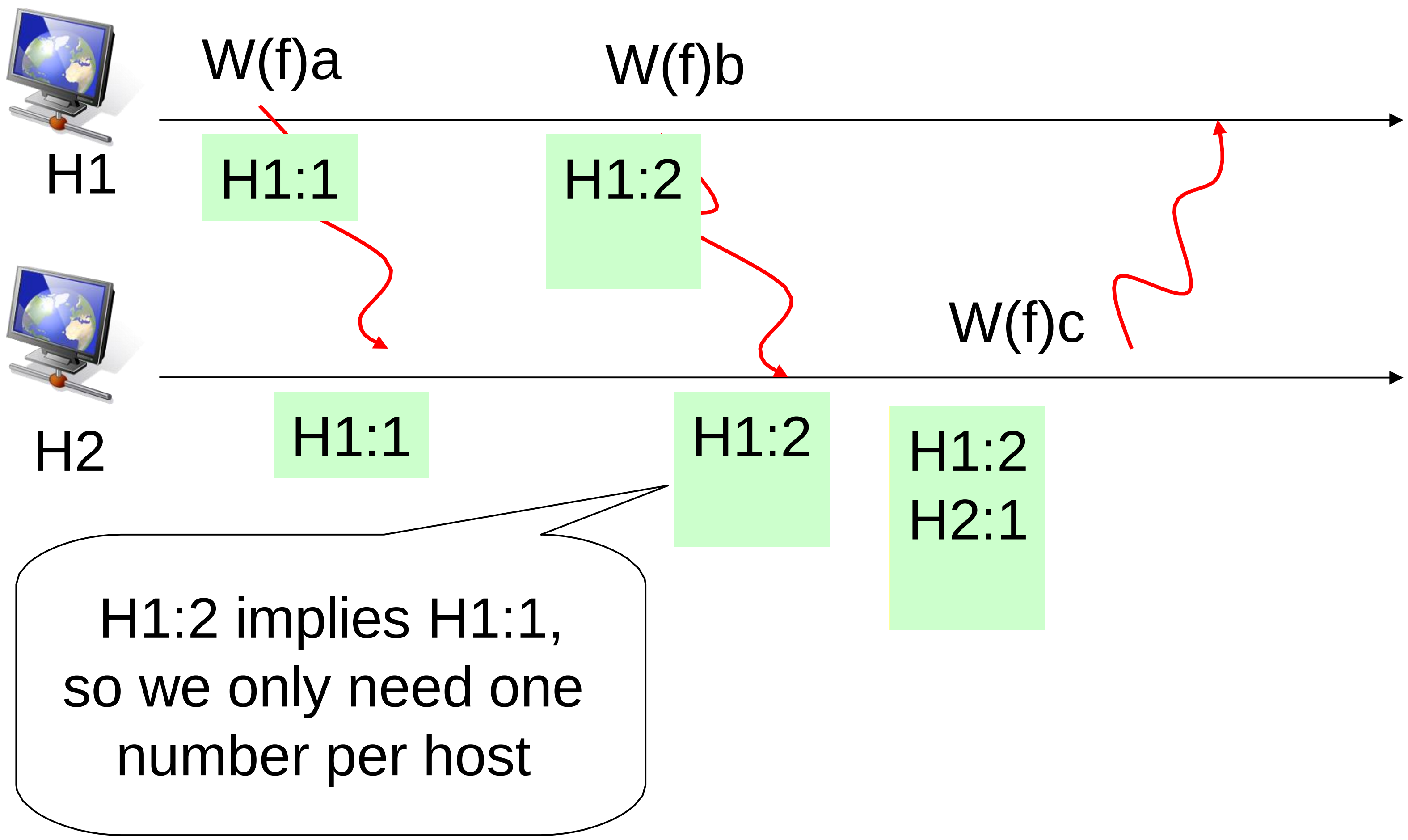
- Strawman: use modification time to decide which version should replace the other
- Problems?
    1. Unsynchronized clocks: new data might have older timestamp than old data
    2. Does not detect conflicts: may lose some contacts…

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



Can now use vector timestamps to compare versions

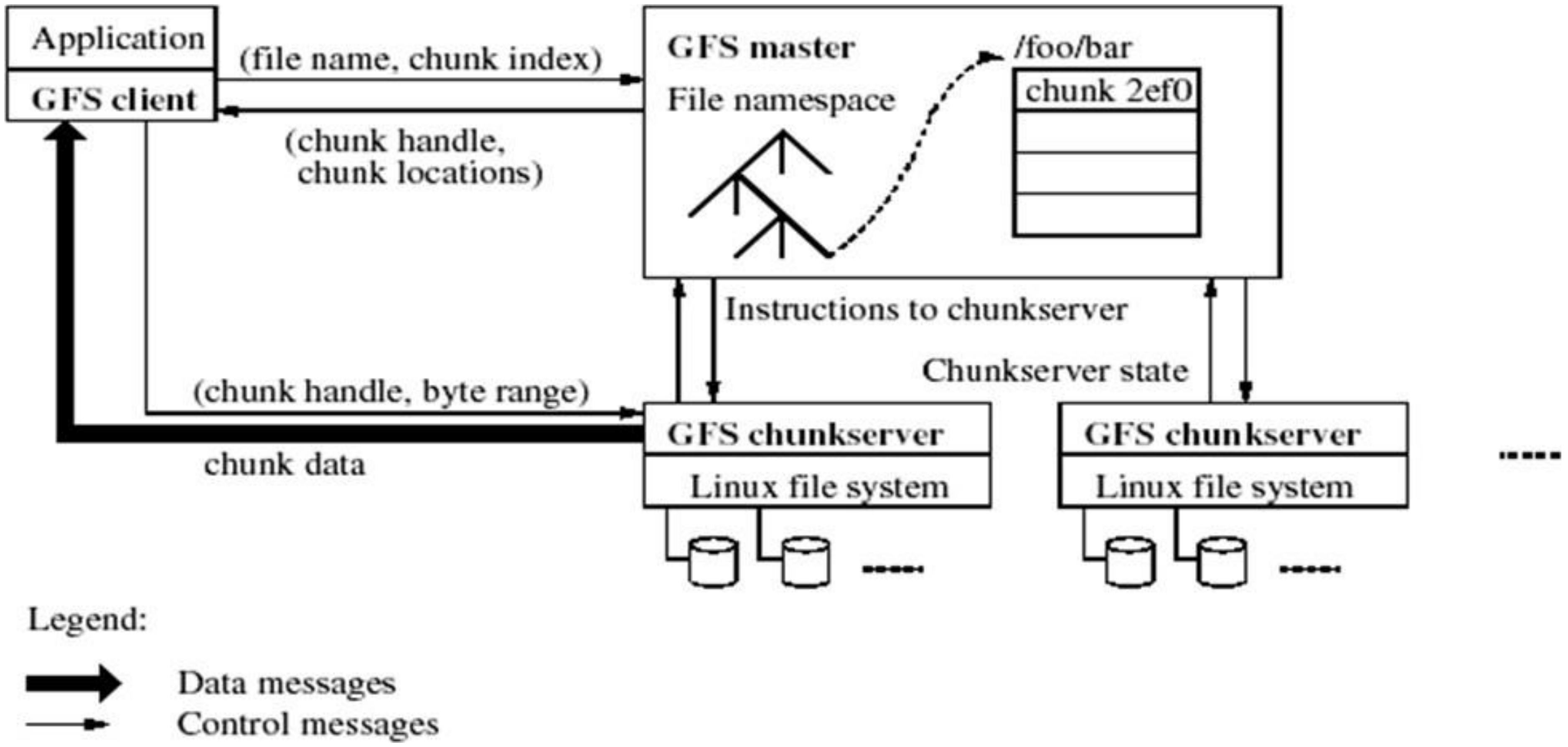
Eventual Consistency: Design tradeoffs

- In eventually consistent data-stores,
  - *Write-write conflicts* are rare
    - Two processes that write the same value are rare
    - Generally, one client updates the data value
      - e.g., One DNS server updates the name to IP mapping
    - Such rare conflicts can be handled through simple mechanisms, such as mutual exclusion
  - R*ead-write conflict* are more frequent
    - Conflicts where one process is reading a value, while another process is writing a value to the same variable
    - Eventual Consistency Design has to focus on efficiently resolving such conflicts
- Eventual Consistency is not good-enough when the client process accesses data from different replicas
  - We need consistency guarantees for a single client while accessing the data-store

Many Other Consistency Models Exist

- Other standard consistency models
  - Linearizability
  - Serializability
  - Monotonic reads
  - Monotonic writes
  - … read Tanenbaum 7.3 if interested
- In-house consistency models
  - AFS: close-to-open semantics
  - NFS: periodic refreshes, close-to-open semantic
  - GFS: atomic at-most-once appends

GFS Architecture: Recap



GFS Consistency

- GFS provides:
  - Hardly any guarantees for normal writes
  - At-least-once atomic appends

- Record Appends: Client only specifies data, not the file offset
  - If record fits in chunk, primary chooses the offset and communicates it to all replicas: *offset is arbitrary*
  - If record doesn't fit in chunk, the chunk is padded: *file may have blank spaces*
  - If a record append fails at any replica, the client retries the operation: *file may contain record duplicates*

Implications for Applications

- GFS-style consistency is not completely intuitive or generally applicable

- Applications must adapt to its weak semantics – how?
  - Rely on appends rather on overwrites
  - Write self-validating records
    - Checksums to detect and remove *padding*
  - Write self-identifying records
    - Unique Identifiers to identify and discard *duplicates*
  - Shifts the burden to the programmer!

- Key takeaway: Maintaining consistency should balance between the strictness of consistency versus efficiency
  - How much consistency is “good-enough” depends on the application

How does all of this relate to serializability?

- Serializability belongs to "Isolation" (I of ACID)
  - Remember C (consistency) in DBMS actually represents application-defined correctness via integrity constraints
- Serializability is one of several isolation levels
  - Just like relaxing consistency, lower isolation levels expose applications to various anomalies
- The isolation semantics are specific to transactional API
  - We saw DSM and FS have other APIs
- Isolation specifies the guarantees that the DBMS gives with respect to how **multi-operation transactions** are allowed to interact under concurrency
  - Consistency models today focus on single-operation consistency of replicated data