Consistency Models

Lecture 9

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

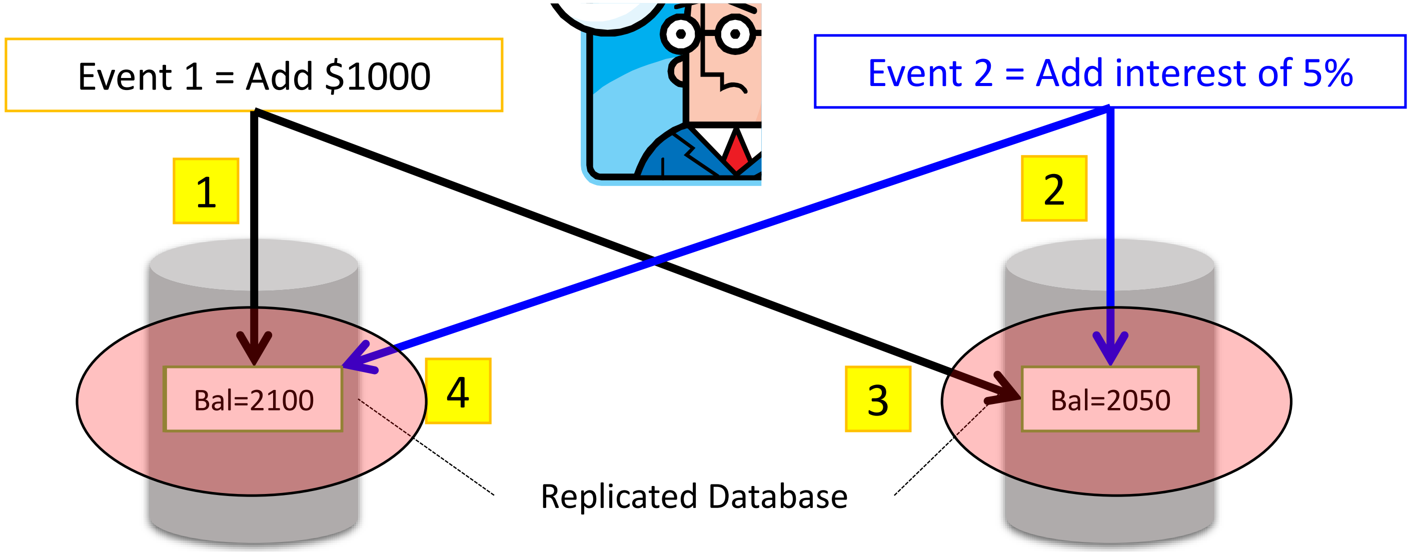
    2. Increasing the availability of services
        - Replication can mask failures such as server crashes and network disconnection

    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?

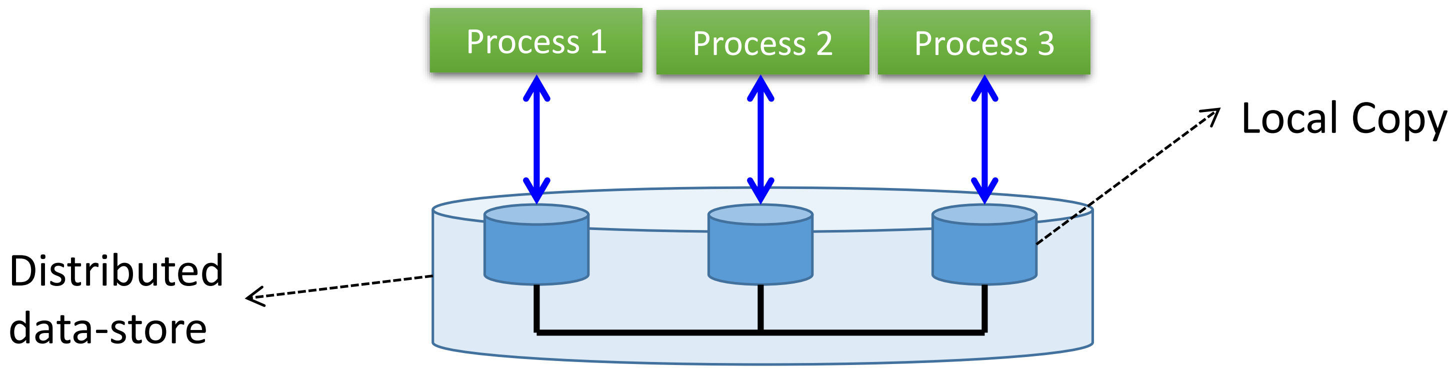
- In a distributed system with replicated data, one of the main problems is keeping the data consistent
- An example:
  - In an e-commerce application, the bank database has been replicated across two servers
  - Maintaining consistency of replicated data is a challenge



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



Consistency Models

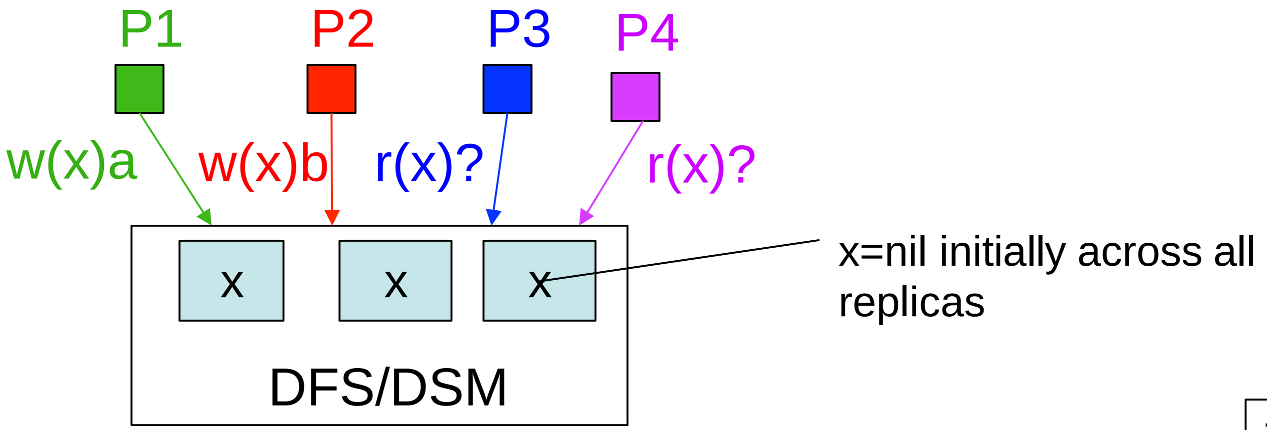
- What is a consistency model?
  - Contract between a distributed data system (e.g., DFS, DSM) and processes constituting its applications
  - E.g.: “If a process reads a certain piece of data, I (the DFS/DSM) pledge to return the value of the last write”

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

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

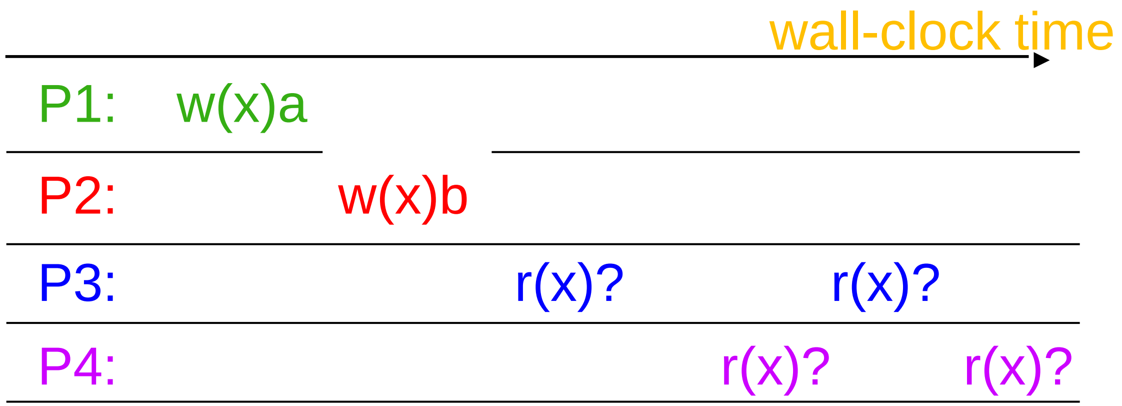
- Variations boil down to:
  - The allowable staleness of reads
  - The ordering of writes across all replicas

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?

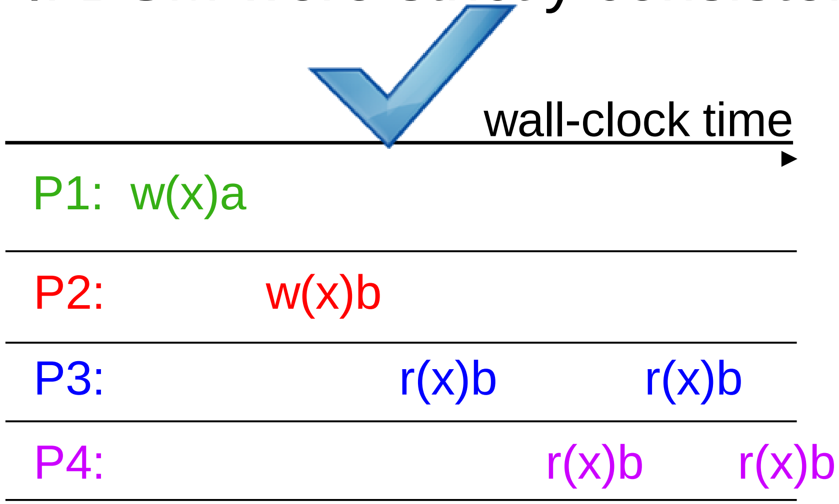
- Suppose we implement rules, can we still get problems?
  - Rule 1: Each read gets the latest written value
    - Reads are never stale
  - Rule 2: All operations at one CPU are executed in order of their timestamps
    - All replicas enforce wall-clock ordering for all writes
  - If DSM were strictly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

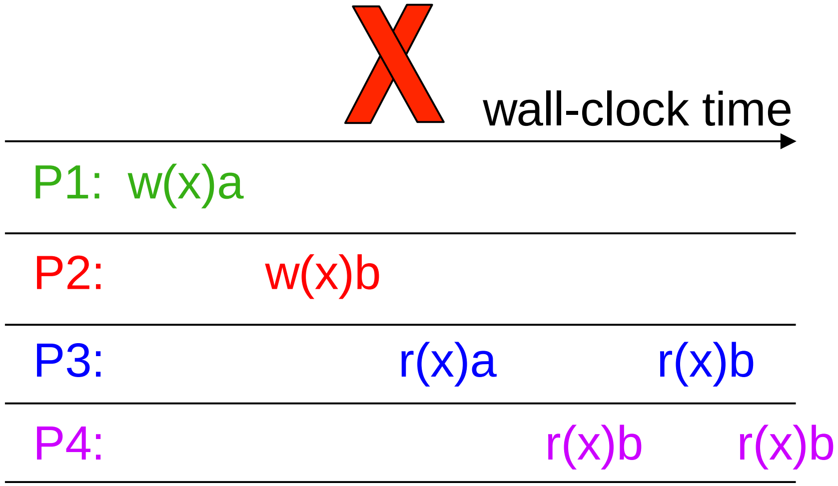
wall-clock time →

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

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

	1	2	3	4	*time*
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

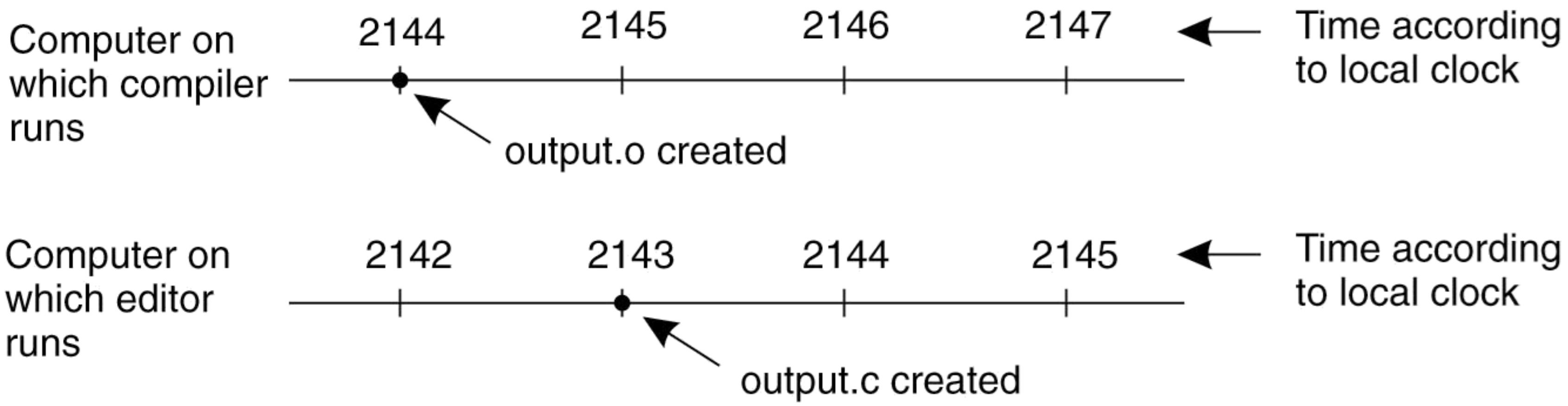
Clocks in Distributed System

- 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?

Universal Coordinated Time (UTC)

- Based on number of transitions of Cesium 133 atom
  - UTC = Average of ~50 cesium clocks around world
- Is broadcast from radio stations on land and satellite (e.g. GPS)
  - Signals from land-based stations are accurate to about 0.1-10 millisecond
  - Signals from GPS are accurate to about 1 microsecond
- Computers with receivers can synchronize their clocks with these timing signals
- How do we keep other machines synchronized with “time server” equipped with UTC receiver?
  - Subject of clock synchronization algorithms to keep clocks **precise** among internal servers in a system, and **accurate** with external sources
  - Ex: Cristian’s algorithm, NTP, Berkeley algorithm, RBS (reference broadcast synch) for wireless
- But in most distributed systems, clocks are never exactly synchronized

Back to Strict Consistency

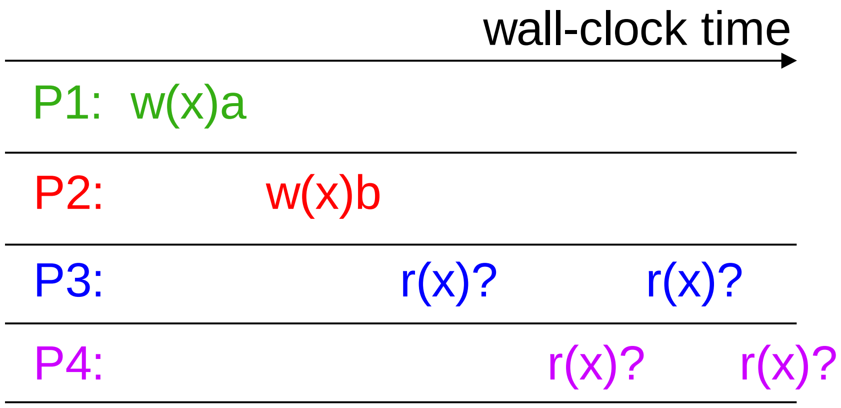
- To achieve strict consistency
  - Each read must be aware of, and wait for, each write
  - Real-time clocks must be strictly synchronized
- Unfortunately
  - Clocks are never *exactly* synchronized. inadequate for distributed systems.
    - Might need totally-ordered events
    - Might need millionth-of-a-second precision
- Strict consistency is theoretical
  - Globally wall clock time => concurrency is not accounted for!
  - Impossible 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

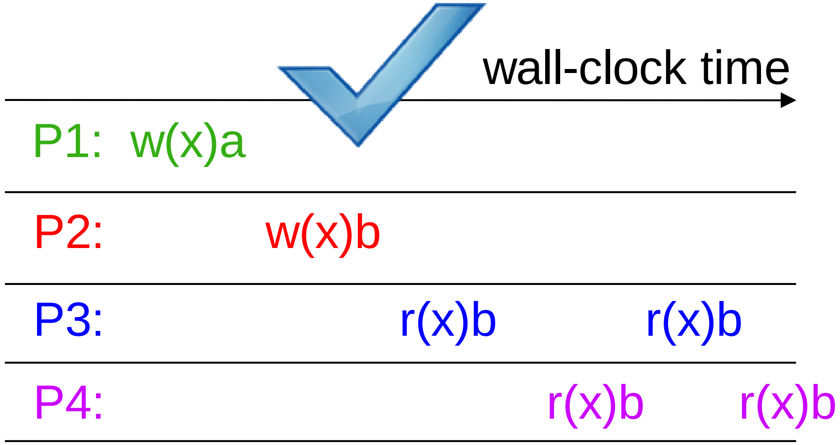
Sequential Consistency

- 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

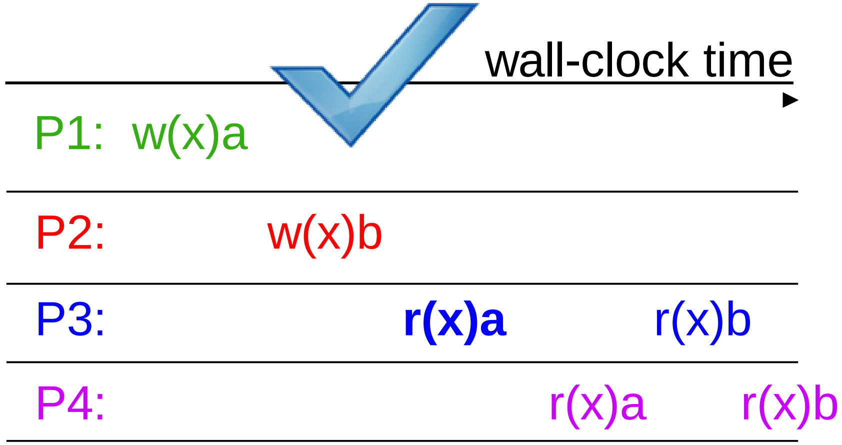
- Any execution is the same as if all read/write ops were executed in some global ordering, and the ops of each client process appear in the order specified by its program



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

wall-clock ordering

This was also strictly consistent



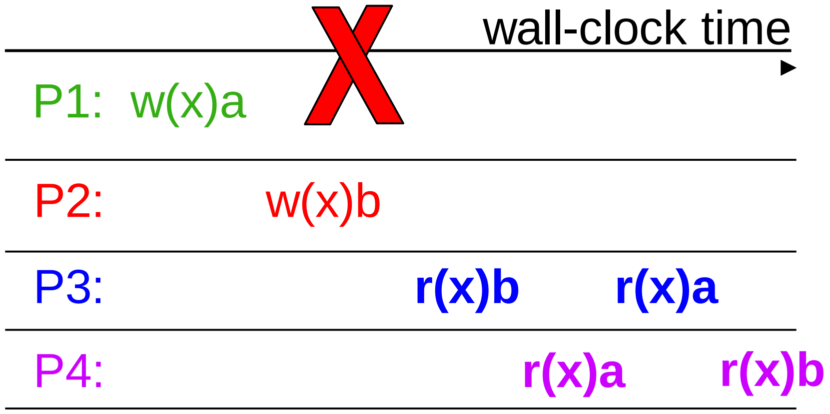
What's a global sequential order that can explain these results?

w(x)a, r(x)a, w(x)b, r(x)b, …

This wasn't strictly consistent

Sequential Consistency

- 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

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

Happens-before: Formally defined

- Definition ($\rightarrow$): We define e $\rightarrow$ e' using the following rules:
  - Local ordering: $e \rightarrow e'$ if $e \rightarrow_i e'$ for any process i
  - Messages: send(m) $\rightarrow$ 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 $e \rightarrow e'$

- $\rightarrow$ 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

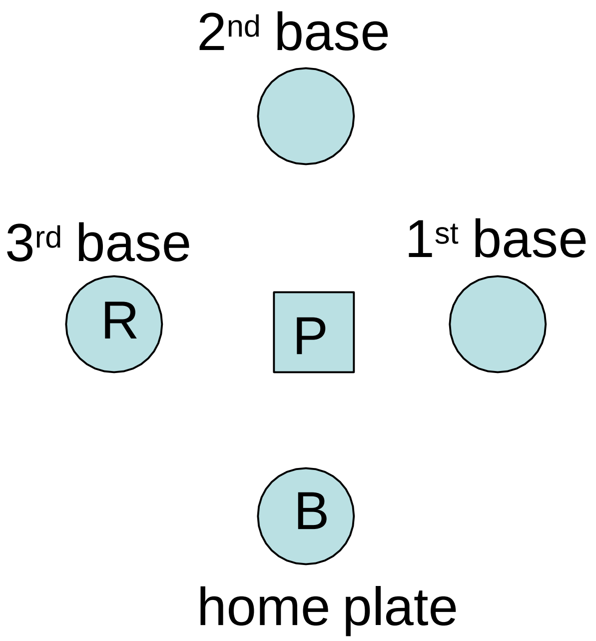
e_6: ball arrives at pitcher

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.

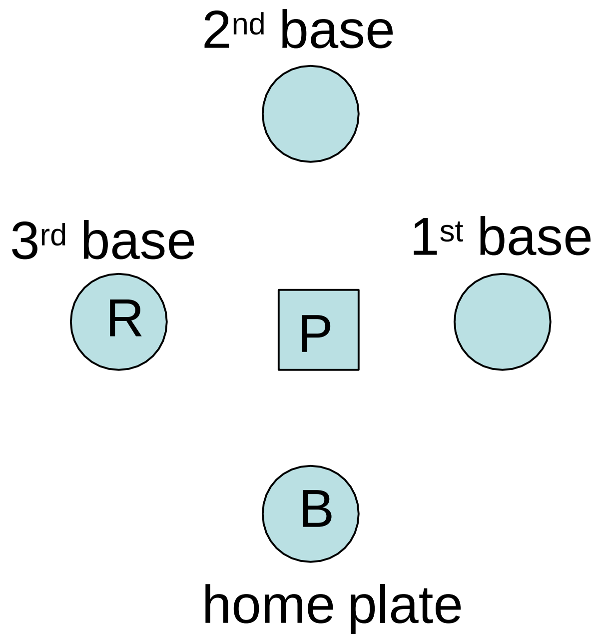
- Note:
  - Property P1 is satisfied by (1)
  - Property P2 by (2) and (3).

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

Lamport on the baseball example

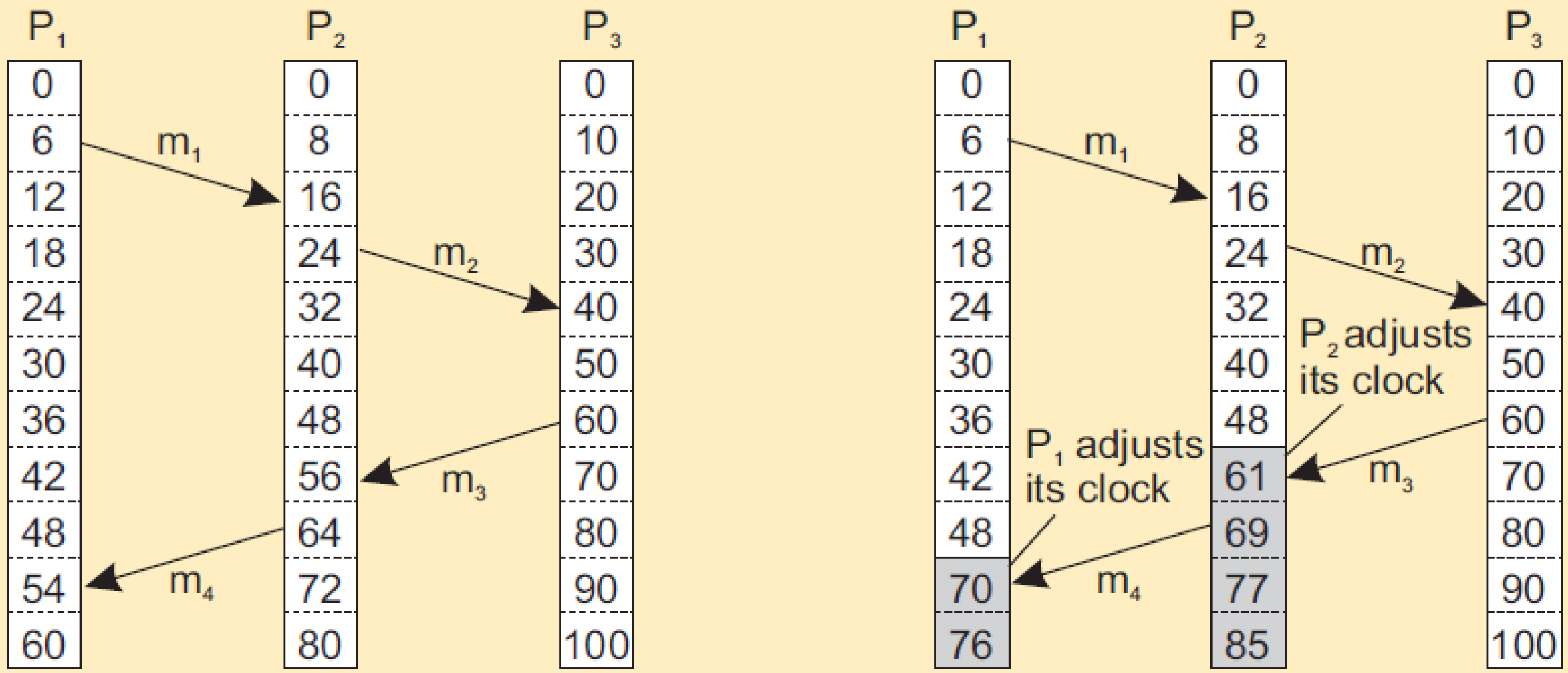
- Initializing each local clock to 0, we get

  $C(e_1) = 1$ (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)

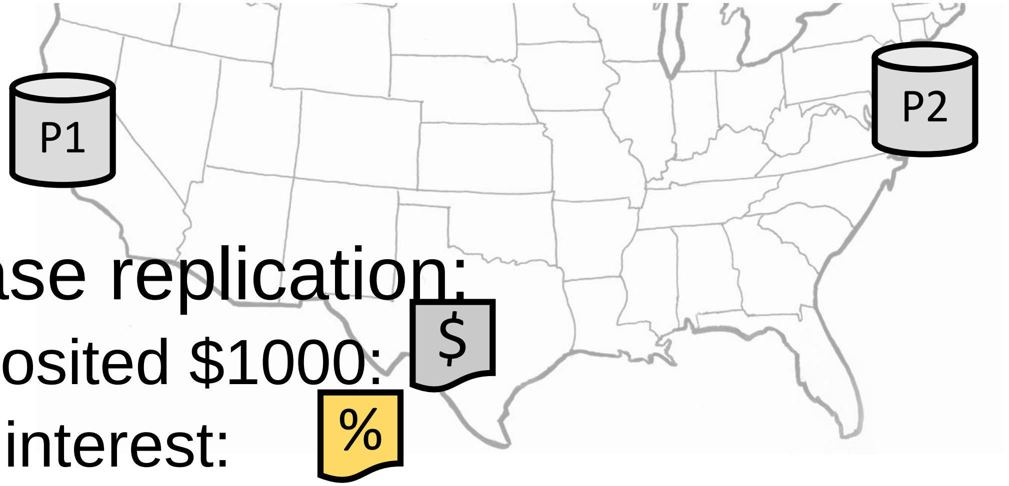


Lamport Clocks: Another example

Consider three processes with event counters operating at different rates



Lamport Clock Application: Making concurrent updates consistent



- Recall multi-site database replication:
  - San Francisco (**P1**) deposited $1000:
  - New York (**P2**) paid 5% interest:

We reached an inconsistent state

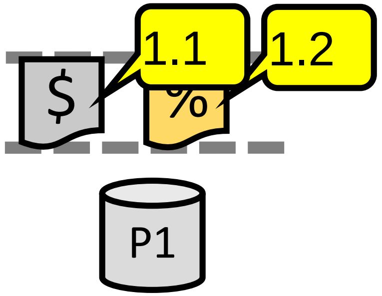
Could we design a system that uses Lamport Clock total order to make multi-site updates consistent?

Totally-Ordered Multicast

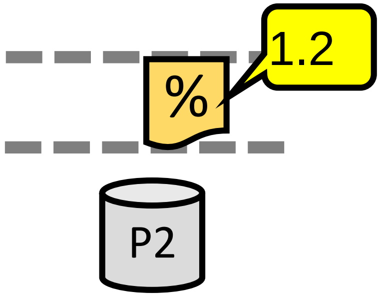
- Client sends update to **one replica → Lamport timestamp C(*x*)**

- **Key idea:** Place events into a **local queue**
  - **Sorted** by increasing C(x)

P1's local queue:



P2's local queue:



Goal: All sites apply the updates in (the same) Lamport clock order

Totally-Ordered Multicast (Almost correct)

1. On **receiving** an event from **client**, broadcast to others (including yourself)

2. On **receiving** an **event from replica:**
   a) Add it to your local queue
   b) Broadcast an ***acknowledgement message*** to every process (including yourself)

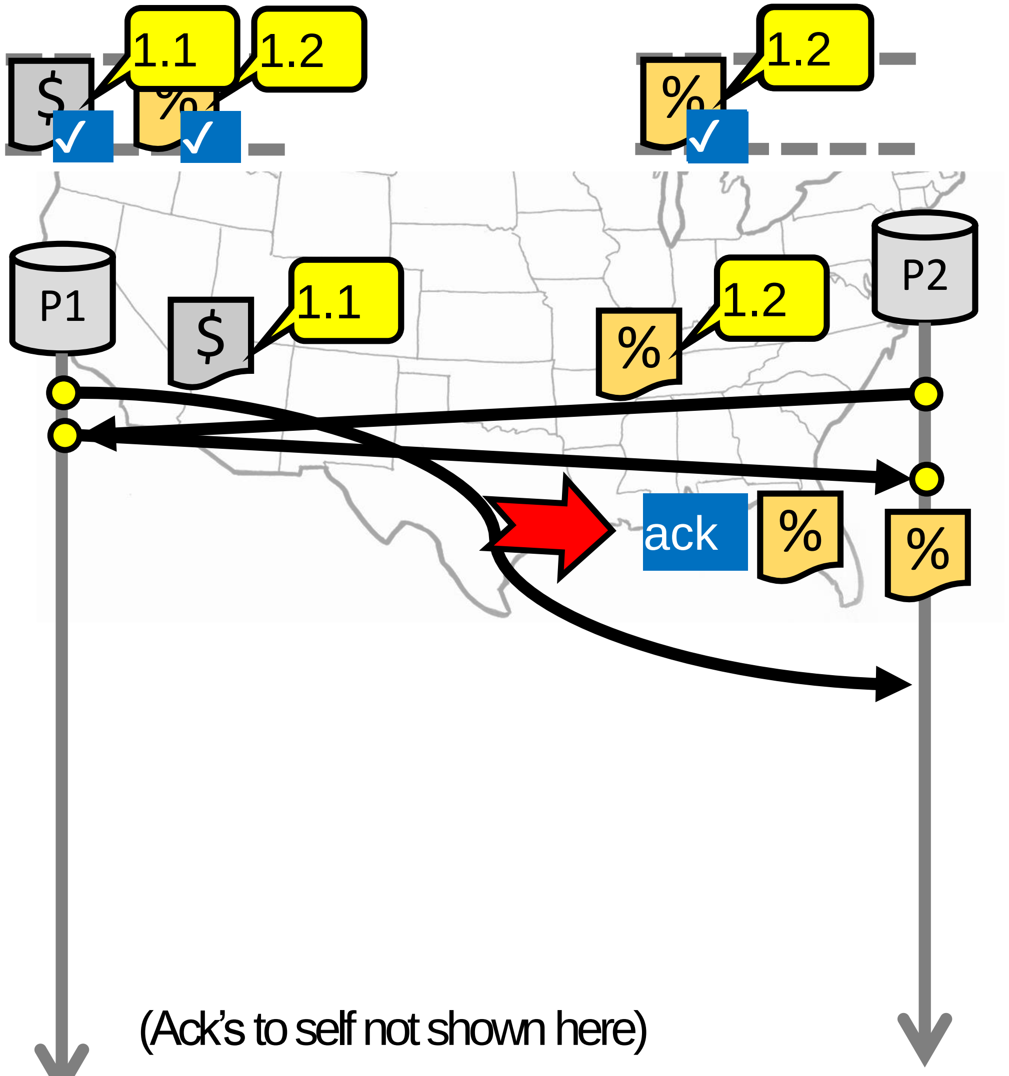
3. On **receiving** an **acknowledgement:**
   - Mark corresponding event ***acknowledged*** in your queue

4. **Remove and process** events **everyone** has ack'ed from **head** of queue

Totally-Ordered Multicast (Almost correct)

- **P1** queues $, **P2** queues %
- **P1** queues and **ack's** %
  - **P1** marks % fully **ack'ed**
- **P2** marks % fully **ack'ed**

P2 processes %



(Ack's to self not shown here)

Totally-Ordered Multicast (Correct version)

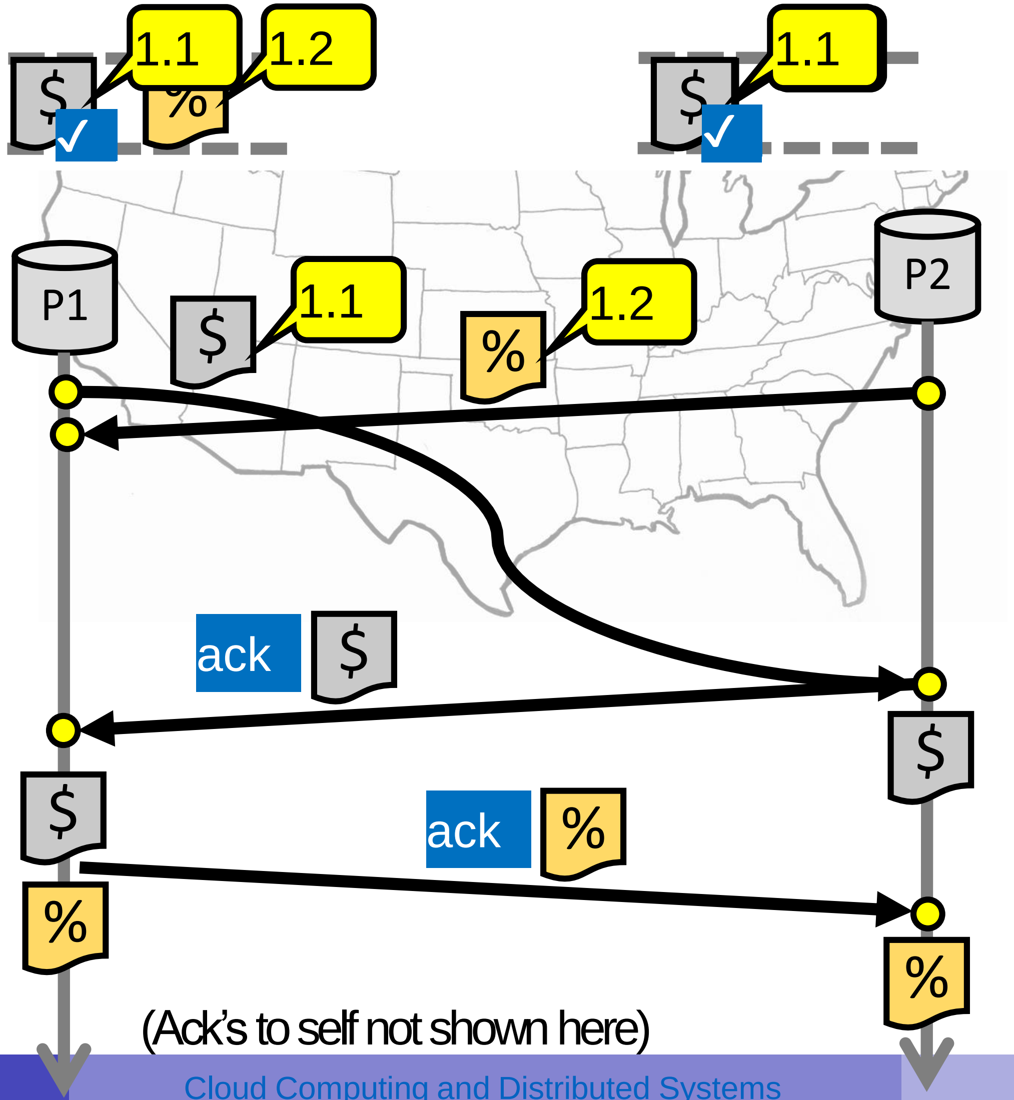
1. On **receiving** an event from **client**, broadcast to others (including yourself)

2. On **receiving** **or processing** an **event:**
   a) Add it to your local queue
   b) Broadcast an ***acknowledgement message*** to every process (including yourself) **only from head of queue**

3. When you **receive** an **acknowledgement:**
   - Mark corresponding event ***acknowledged*** in your queue

4. **Remove and process** events **everyone** has ack'ed from **head** of queue

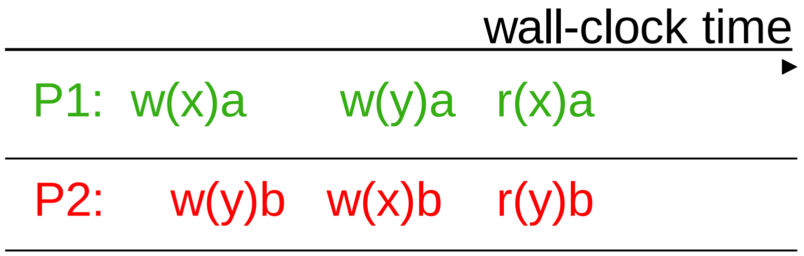
Totally-Ordered Multicast (Correct version)



(Ack's to self not shown here)

Back to Sequential Consistency

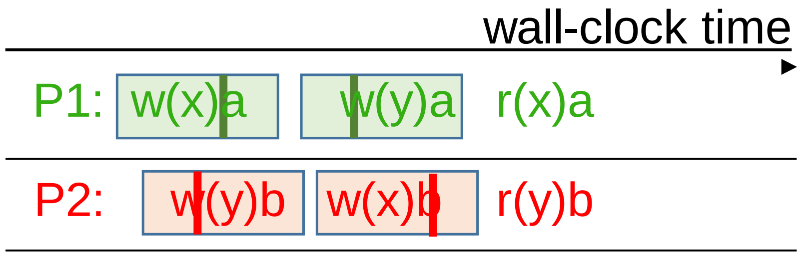
- Easier to implement than strict consistency
  - No notion of real time => System can interleave different machines' ops
- But sequential consistency is not compositional
  - Example below, operations on x and y considered separately are sequentially consistent
    - Fact that P1 reads x and gets a is OK, P2 reads y and gets b is ok
  - But taken together, this is inconsistent
    - No sequentially consistent ordering can produce r(x)a and r(y)b while keeping program order



Ordering of operations	Result	
$W_1(x)a; W_1(y)a; W_2(y)b; W_2(x)b$	$R_1(x)b$	$R_2(y)b$
$W_1(x)a; W_2(y)b; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a; W_2(y)b; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_2(x)b; W_1(x)a; W_1(y)a$	$R_1(x)a$	$R_2(y)a$

Linearizability (a.k.a strong consistency)

- Linearizability = sequential consistency + read-time guarantee
  1. All servers execute all ops in *some* identical sequential order
  2. Global ordering preserves each client's own local ordering
  3. Global ordering preserves real-time guarantee
     - Each operation should appear to take effect instantaneously at some moment between start and completion
     - Take effect => result of writes propagated and visible in other stores
- In example below, shaded area shows start/end of each op, line shows instant at which it took effect
  - w(x)a would preceed w(x)b. similarly, w(y)b would precede w(y)a.
  - These orderings must be preserved => x can only be *b*, y can only be *a*



Ordering of operations	Result	
$W_1(x)a; W_2(y)b; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a; W_2(y)b; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$

Sequential Consistency & Linearizability

- 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
  - …

- Could we relax consistency further to improve performance?

Take-away points: Lamport clocks

- Can **totally-order** events in a distributed system: that's useful!

- **But:** while by construction, **a** → **b** implies *C*(**a**) < *C*(**b**),
  - The converse is not necessarily true:
    - *C*(**a**) < *C*(**b**) does not imply **a** → **b** (possibly, **a** || **b**)

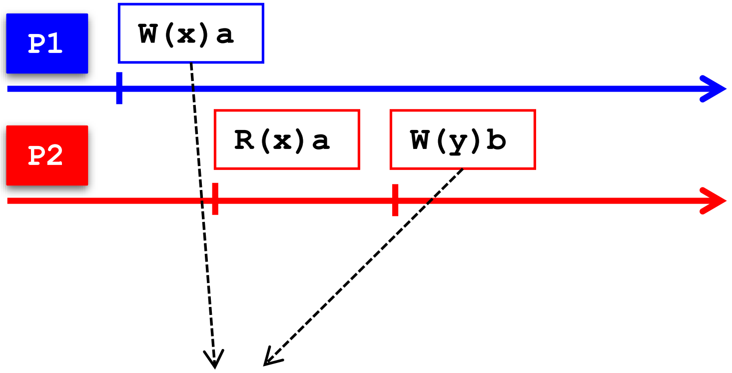
- Similar rules for concurrency
  - *C(e) = C(e')* implies *e* ǁ *e'* (for distinct *e,e'*)
  - *e* ǁ *e'* does not imply *C(e) = C(e')*

- i.e., Lamport clocks arbitrarily order some concurrent events that can actually be executed in parallel

Can't use Lamport clock timestamps to infer causal relationships between events

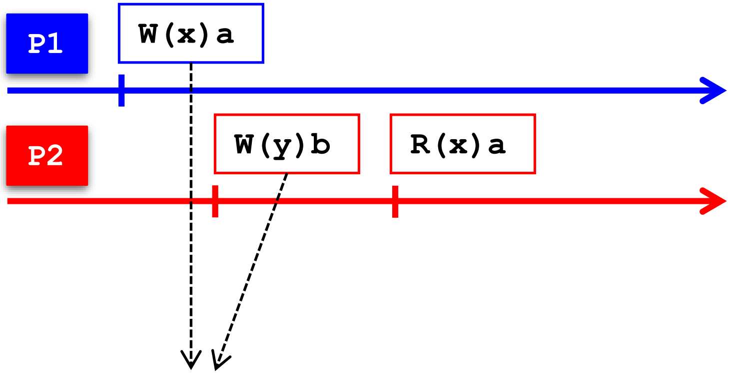
Causal relationship

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



Events are causally related
Events are not concurrent

- Computation of **y** at $\mathbf{P_2}$ may have depended on value of **x** written by $\mathbf{P_1}$



Events are not causally related
Events are concurrent

- Computation of **y** at $\mathbf{P_2}$ does not depend on value of **x** written by $\mathbf{P_1}$

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:		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.

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.

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

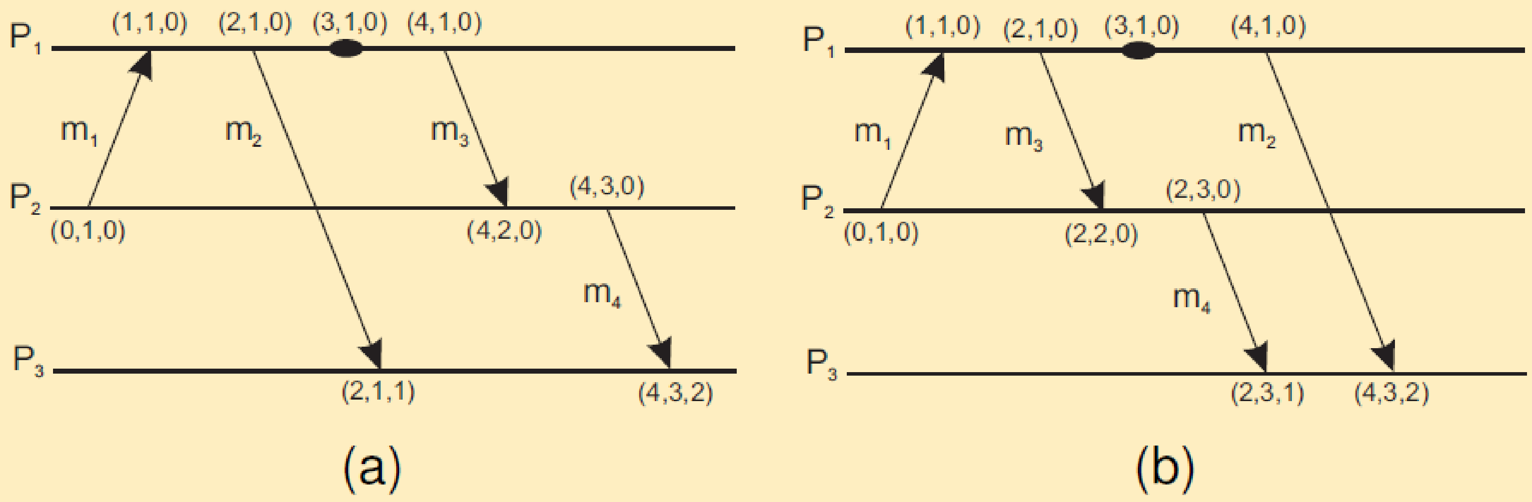
Causal Precedence

- 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

Vector Clock vs Lamport Clocks: Properties

- Both vector clocks (VC) and lamport clocks (LC) capture causality
  - If **a** → **b** implies LC(**a**) < LC(**b**), VC(**a**) < VC(**b**)

- But vector clocks allow us to causally order events based on timestamp
  - With lamport clocks, LC(**a**) < LC(**b**) does not imply **a** → **b**
  - But if VC(**a**) < VC(**b**) then **a** → **b**
    - Remember VC(**a**) < VC(**b**) means 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']
  - Similarly if VC(**a**) = VC(**b**) then **a** =
    - VC(**a**) = VC(**b**) means for all *k*, *ts(a)[k] == ts(b)[k]*
  - Similarly if VC(**a**) || VC(**b**) then **a** || **b**
    - VC(**a**) || VC(**b**) means VC(**a**) is not < VC(**b**) and VC(**b**) is not < VC(**a**)

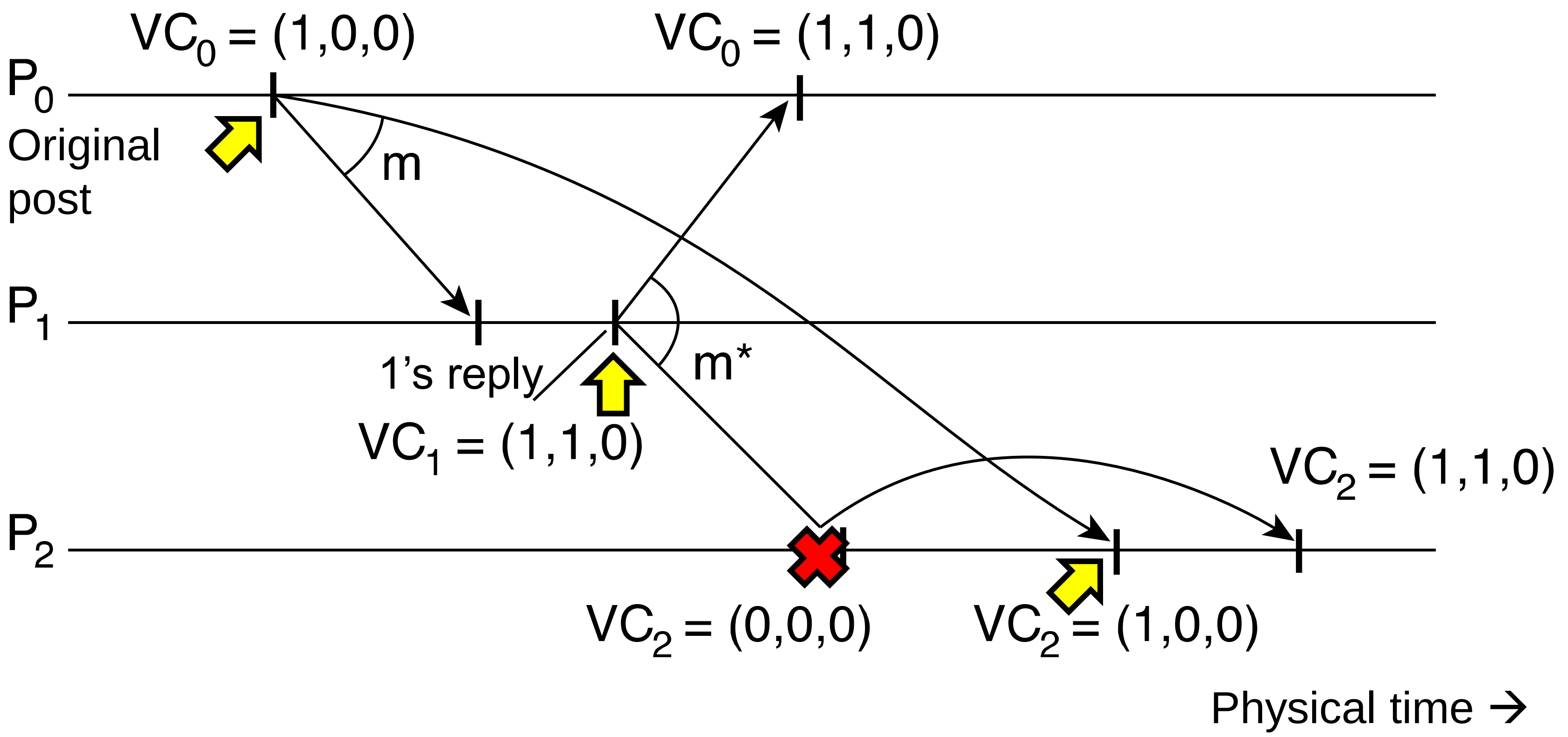
Vector Clock Application:
Causally-ordered bulletin board system

- Distributed bulletin board application
  - Each post → multicast of the post to all other users

- **Want:** No user to see a reply before the corresponding original message post

- Deliver message only **after** all messages that **causally precede** it have been delivered
  - Otherwise, the user would see a reply to a message they **could not find**

Vector Clock Application: Causally-ordered bulletin board system



- User 0 posts, user 1 replies to 0's post; user 2 observes

Casually-ordered Multicast with Vector Clocks

- Observation
  - We can ensure messages are delivered only if all causally preceding messages have been delivered
- Adjustment to achieve causal ordered multicast
  - Clocks are adjusted when sending message and when delivering message to application, **not during receiving a message**
- **Sending**
  - P_i increments $VC_i[i] = VC_i[i] + 1$
- **Receving** message m with ts(m) P_j checks following
  - $ts(m)[i] = VC_j[i] + 1$
    - Pj expects m to be the next message from Pi
  - $ts(m)[k] <= VC_j[k]$ for all $k != i$
    - P_j has already delivered all messages that have also been delivered by Pi when it sent message m
- If conditions met, deliver message m to application and update VC
  - $VC_j[k] = max\{VC_j[k], ts(m)[k]\}$ for each k

Back to 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 still need to enforce a global ordering of all operations
  - Hence, one can get better performance than sequential consistency

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 eventually reflect previously written values, potentially even after very long times
  - More on this later

Many Other Consistency Models Exist

- Other standard consistency models
  - 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 appends
- 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