# Causal Consistency

Distributed Systems

Lecture 14

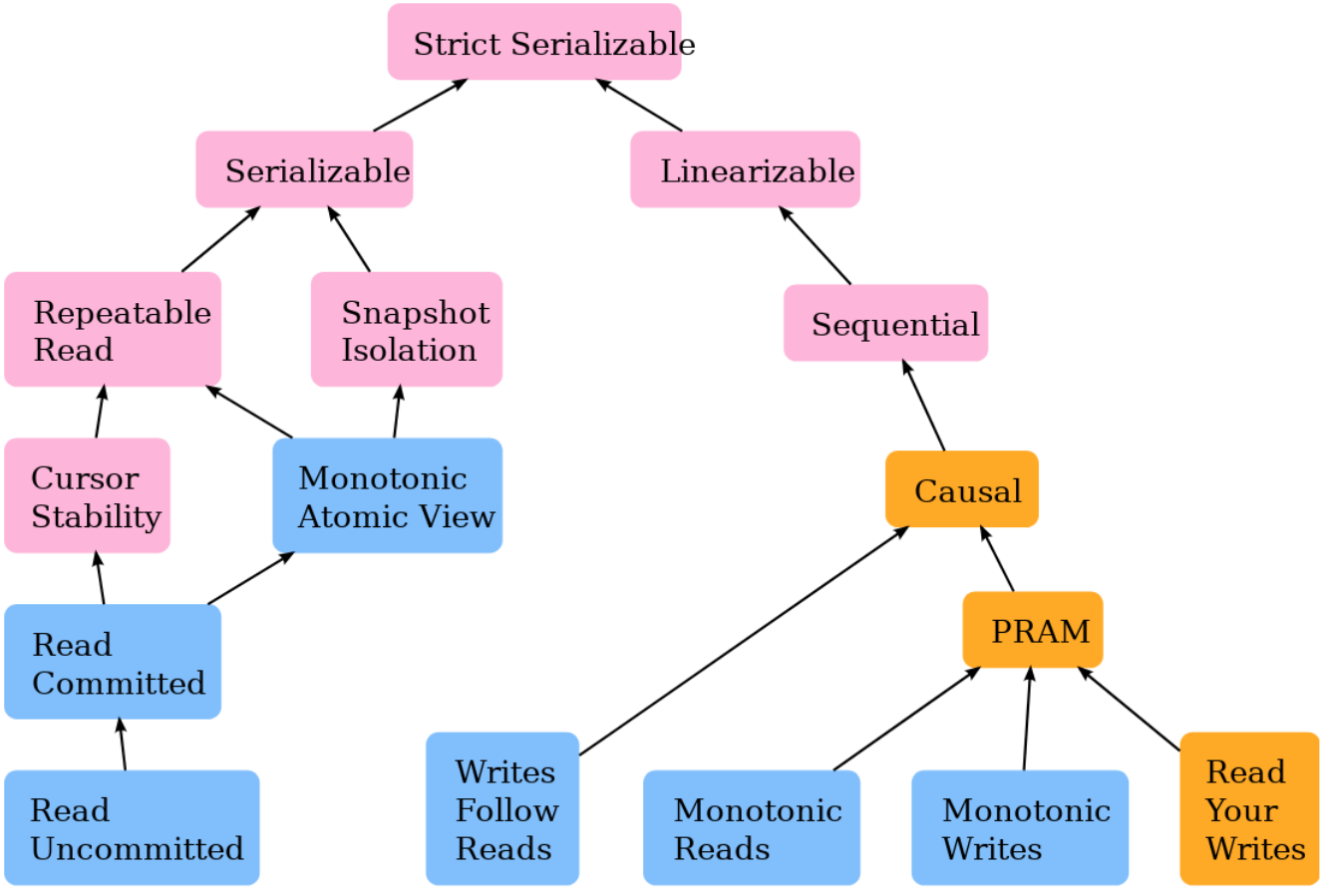
# Sequential consistency

## Definition

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.

- All writes are seen in the same order by all processes.
- Implemented using write-primaries or total-order-broadcast
- Writes are **blocking**: Slow performance
- Replicas can be geographically distributed

# Consistency Models




https://jepsen.io/consistency

# Causal consistency

## Definition

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.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| P1: | W(x)a | | | | | | W(x)c | | |
| P2: | | | R(x)a | | W(x)b | | | | |
| P3: | | | | R(x)a | | | | R(x)c | R(x)b |
| P4: | | R(x)a | | | | | | R(x)b | R(x)c |

- Processs "communicate" via the datastore and not with each other
- Reading a variable value potentially causes subsequent write on the same process
- Happens before dependency graph composed of Write-Read-Write edges: W(x) $<$ R(x)a $<$ W(x)b
- P2:W(x)b and P1:W(x)c are concurrent. P3 and P4 can thus see different orderings.

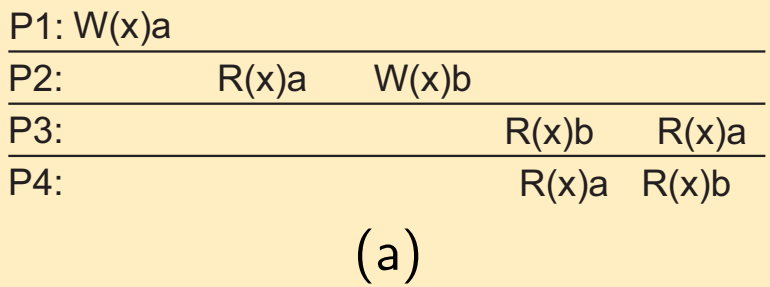
# Why Causal Consistency?

1. Sally cannot find her son Billy. She posts update S to her friends: "I think Billy is missing!"
2. Momentarily after Sally posts S, Billy calls his mother to let her know that he is at a friend's house. Sally edits S, resulting in S2: "False alarm! Billy went out to play."
3. 3 . Sally's friend James observes S2 and posts status J in response: "What a relief!"

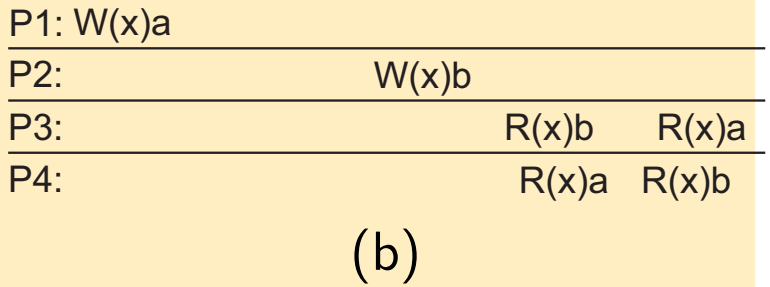
If causality is not respected, a third user, Henry, could perceive effects before their causes; if Henry observes S and J but not S2, he might think James is pleased to hear of Billy's would-be disappear- ance! If the site had respected causality, then Henry could not have observed J without S. (From "Bolt-on Causal Consistency". Baillis et.al. SIGMOD '13)

# Causal Consistency Examples

## (a) A violation of a causally-consistent store. (b) A correct sequence of events in a causally-consistent store




(a)




(b)

1. P1:W(x)a $\rightarrow$ P2:R(x)a $\rightarrow$ Wx(b). Thus the two writes are causally related and must take effect in same order.
2. Writes are not causally related (no interleaved read), and thus can be seen in any order.

# More Causal Consistency Examples

| | | | | | |
|---|---|---|---|---|---|
| P1: W(x)a | | | | | |
| P2: | R(x)a | W(y)b | | | |
| P3: | | | R(y)b | R(x)? | |
| P4: | | | R(x)a | R(y)? | |

- What should the reads return?
- P3 R(x): a
- Acceptable for P4 to return NULL

# Causal Consistency Caveats

- Similar in principle to causal-order broadcast discussed earlier
- Dont want to see a reply before original post
- In causal-order broadcast, a process waits if it receives a message from the "future", based on its vector clock timestamp.
- Same principle can be used to implement causal consistency
- Reads are causally related to writes.
- Out of band causality is not captured
  - Cant "phone" a friend and coordinate
  - Fails to capture causality of writes
  - "When you see x=1, write y=1"

# Implementing Causal Consistency

Causal consistency the strongest we can have in presence of partitions

| | | | | |
|---|---|---|---|---|
| P1: W(x)a | | | | |
| P2: | R(x)a | W(y)b | | |
| P3: | | | R(y)b | R(x)? |
| P4: | | | R(x)a | R(y)? |

- Need to keep track of causal histories
- P3 needs to know about $W(x)a \rightarrow W(y)b$
- Need to keep a dependency graph of operations
- Similar to causal order broadcast
- When reading from a replica, "wait" until replica has applied all causally preceeding writes
- For performance, want to *lazily* propagate writes
- Note: Local-read sequential consistency algorithm has *eager* propagation.

# Vector-clock based Algorithm

1. Local reads. Writes are non-blocking (async broadcast).
2. Causal Memory: Definitions, Implementation, and Programming
3. `https://smartech.gatech.edu/bitstream/handle/1853/6781/GIT-CC-93-55.pdf?sequence=1&isAllowed=y`
4. Each replica maintains outgoing and incoming queue of write requests.
5. Write requests timestamped with vector clocks $t$ ($t[i]$: number of writes known by $i$)
6. Incoming queue sorted by timestamp $s$.
7. Apply write only if all caught up ($s[j] == t[j] + 1$), where j is source of broadcast.

/* Initialization: */

**foreach** $x \in \mathcal{M}$ **do**
    $M[x] := \perp$
**for** $j := 1$ **to** $n$ **do**
    $t[j] := 0$
$OutQueue := \langle\rangle$
$InQueue := \langle\rangle$

/* Read action: to read from $x$ */

**return**$(M[x])$ /* Add $r_i(x)*$ to $L_i$ and $S_i$ */

/* Write action: to write $v$ to $x$ */

$t[i] := t[i] + 1$
$M[x] := v$ /* Add $w_i(x)v$ to $L_i$ and $S_i$ */
enqueue $\langle i, x, v, t\rangle$ to $OutQueue$

/* Send action: executed infinitely often */

**if** $OutQueue \neq \langle\rangle$ **then**
    let $A$ be some nonempty prefix of $OutQueue$
    remove $A$ from $OutQueue$
    **send** $A$ **to** all others

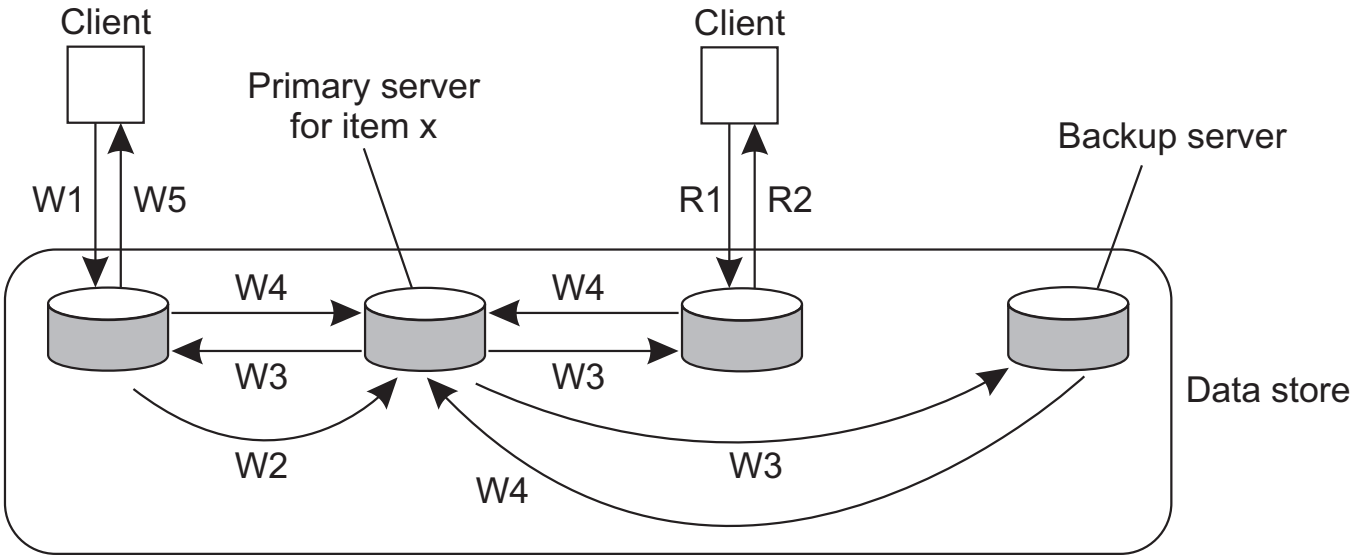
/* Receive action: upon receipt of $A$ from $p_j$ */

**foreach** $\langle j, x, v, s\rangle \in A$
    enqueue $\langle j, x, v, s\rangle$ to $InQueue$

/* Apply action: executed infinitely often */

**if** $InQueue \neq \langle\rangle$ **then**
    let $\langle j, x, v, s\rangle$ be head of $InQueue$
    **if** $s[k] \leq t[k]$ for all $k \neq j$ **and** $s[j] = t[j] + 1$ **then**
        remove $\langle j, x, v, s\rangle$ from $InQueue$
        $t[j] := s[j]$
        $M[x] := v$ /* Add $w_j(x)v$ to $S_i$ */

# Recall Primary-based Replication




W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

# Primary-based Implementation

- All writes go through a primary. (For simplicity, assume 1 primary for all objects)
- Writes are thus naturally causally ordered

## Central Problem

If a client issues reads to two different replicas, how to ensure that the reads are causally ordered?

All read and write operations are logical-clock timestamped:

1. Write operations assigned monotonically increasing timestamps by primary
2. Before a read, compute minimum acceptable timestamp.
   - Max ts across reads over all keys, and writes for that key
3. Each replica maintains $M_r$: max timestamp of all writes received by that replica
4. Read returns from replica only when $M_r >$ read timestamp

Doug Terry et.al. "Consistency-Based Service Level Agreements for Cloud Storage"

# Eventual Consistency

- Concurrent updates are rare
  - Mostly: read-write conflicts
  - Examples: Web-caches, CDN's, DNS

## Eventual Consistency

If no updates take place for a long time, all replicas *eventually* become consistent (have the same data stored)

- Easy to implement
- In practice, write-write conflicts handled through some form of leader election
- Inconsistency windows often small ($<$500 ms)

# 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 $A$

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

## The principle of a mobile user accessing different replicas of a distributed database