Replication & Consistency

Slide set 5
Distributed Systems

Graduate Level

K. N. Toosi Institute of Technology
Dr. H. Khanmirza
h.khanmirza@kntu.ac.ir

Consistency

- In distributed systems we need replication (= repeat) of data for
  - Reliability
    - We discuss later
  - Performance
    - Is important for scaling in size or geographical span

Consistency

- Consistency
  - Keeping the same content in all replicas
  - When a replica is updated we must ensure this update is propagated to other replicas
  - A read operation performed at any copy will always return the same data
  - When and how determines the price of consistency problem

Consistency Problem Example

- User needs a web page from a far remote site
  - Far means: delay ~ multi-seconds
  - How access time can be improved?

- Approach 1:
  - Browser can keep a copy of that page in cache (client-side replication)
  - What if the content of the page is modified

  - Browser can always talk with server and prefetches the latest content → If read_count << modification_count the browser wastes the bandwidth!

  - Cache has a invalidation time, If read_period > validation_period caching is useless

Consistency Problem Example

- Approach 2:
  - Remote server keeps the track of caches and updates cache contents when they modified
  - Implies server processing load & state maintenance
  - Server bandwidth
    - If read_count << modification_count it is a clear waste of the bandwidth!

Consistency Problem

- Replication solves scalability problems
  - Keeping all replicas tightly-consistent needs global synchronization
  - Another costly scalability problem!

- Consistency problems cannot be solved efficiently
  - There is no best solution to replicating data
  - We have to relax the atomic operation condition to avoid global synchronization and find an efficient solution

- There are also no general rules for relaxing
  - Exactly what can be tolerated is highly dependent on applications

- We should define the access and update patterns of the replicated data

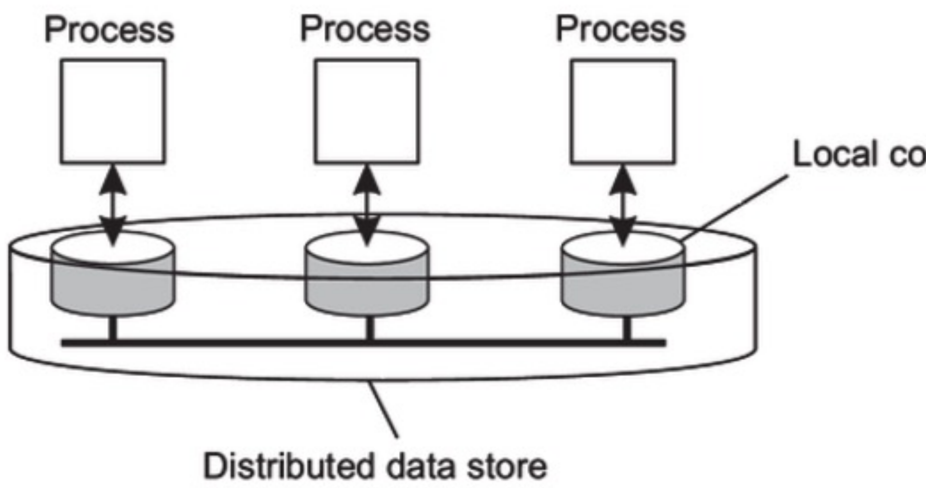
Consistency Problem

- Tight Consistency
  - Informally, the update should be propagated to all copies before a subsequent operation takes place
  - Note that this is an imprecise definition

  - The key idea is that an update is performed at all copies as a single atomic operation, or a transaction.

Consistency Model

- System Model
  - Data is physically distributed and replicated across multiple processes
    - Assume any shared data like shared memory, shared database, shared file system, ⋯
  - Each process has a local copy of data
  - Write Op: Every action on data that modifies it
    - Write operations are propagated to other copies
  - Read Op: Non-write operation



Consistency Model

- A contract between processes and the data store that says that if processes agree to obey certain rules, the store promises to work correctly

Data-Centric Consistency Models

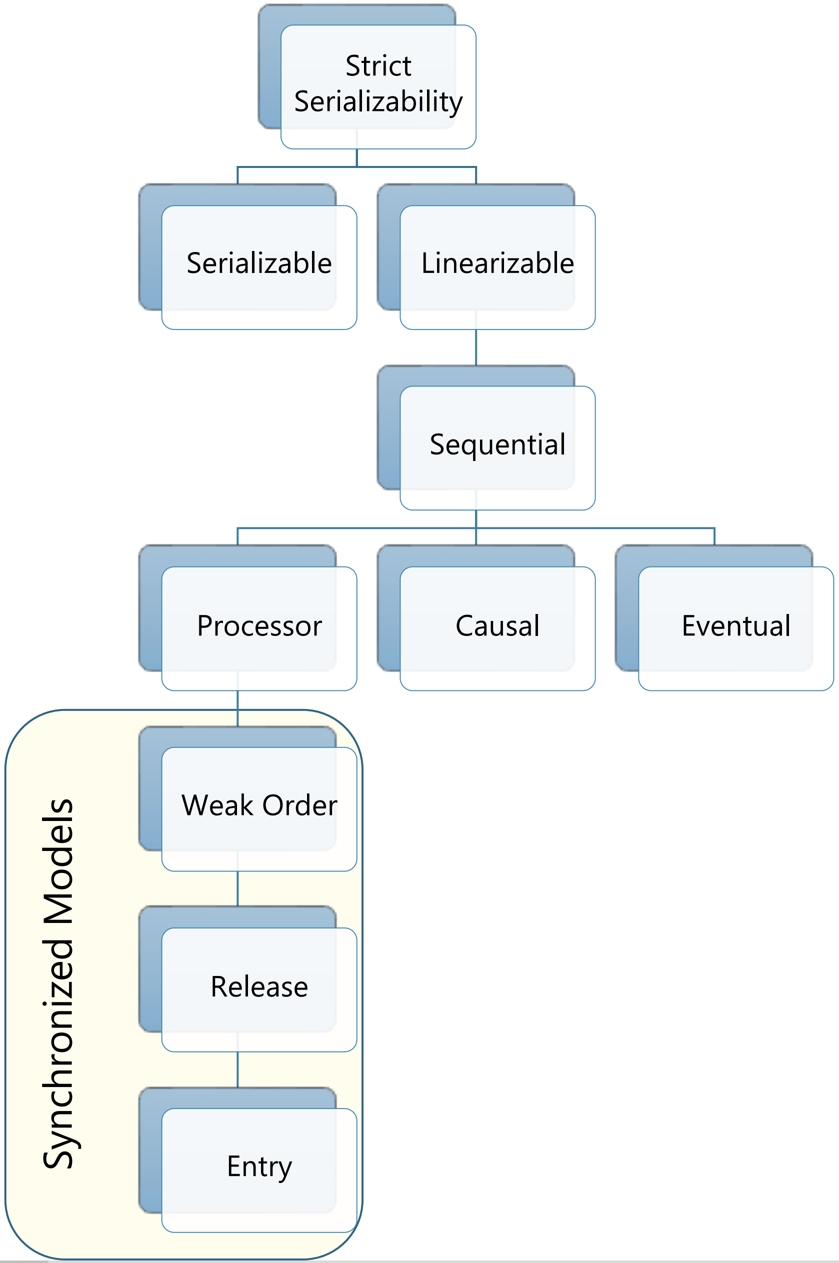
- An important class of models comes from the field of parallel programming

- In parallel and distributed computing multiple processes will need to share resources and access these resources simultaneously

- In such conditions, there is need for consistent ordering of operations

  - All replicas first need to reach agreement on when exactly an update is to be performed locally

Hierarchy of Consistency Models



Strict Serializability Consistency

- A write to a variable by any process needs to be seen instantaneously by all other processes

- Instantaneously: implies having a global time and only one update operation is executed in a predefined time period

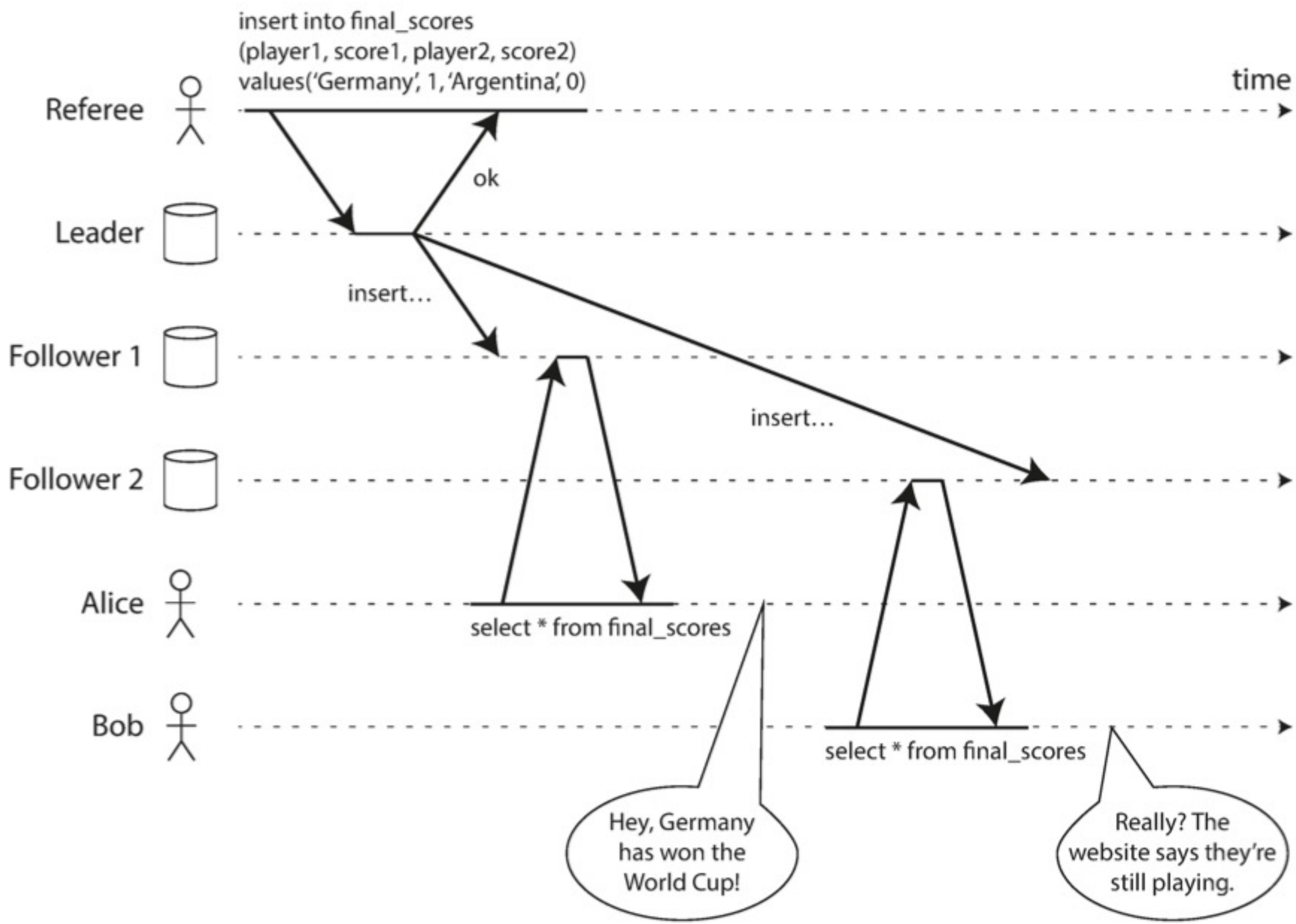
Serializability Consistency

- Is a transactional model where each operation takes place atomically
- Transactions have total order
- Mostly discussed in database field
- Database guarantees that transactions have the same effect as if they ran serially
- Read Committed, Read Uncommitted, Repeatable Reads

Linearizable Consistency

- Also known as Strong Consistency, Atomic Consistency, Immediate Consistency
- Make a system appear as if there were only one copy of the data, and all operations are atomic
- This is recency guarantee and has not notion of transactions

Linearizable Consistency



Linearizable Consistency

- There must be some point in time (between the start and end of the write operation) at which the value of x atomically flips from old to new.
- After that point all clients must see the new value, reading from any data store

Sequential Consistency

- Defined by Lamport in the context of shared memory for multi-processor systems

- A data store is sequentially consistent if
  - When processes run concurrently on (possibly) different machines, any valid interleaving of read and write operations is acceptable behavior
  - However, all processes must see the same interleaving (order) of operations

Sequential Consistency

- The following notation is used to demonstrate behavior of two processes operating on a shared data item
  - The horizontal axis is time which increases from left to right
  - Process P1 Writes value *a* to variable *x*

P1:	W(x)a		
P2:		R(x)NIL	R(x)a

  - Process P2 Reads *NIL* from *x* first and then *a*

Sequential Consistency

- P1 writes *a* to variable *x*

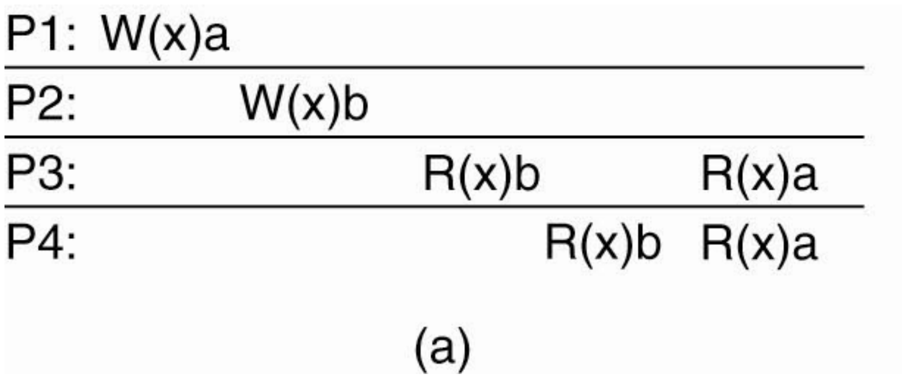
- P2 reads data but value *a* is not propagated to the second replica (process)

- P2 after some time reads the written data

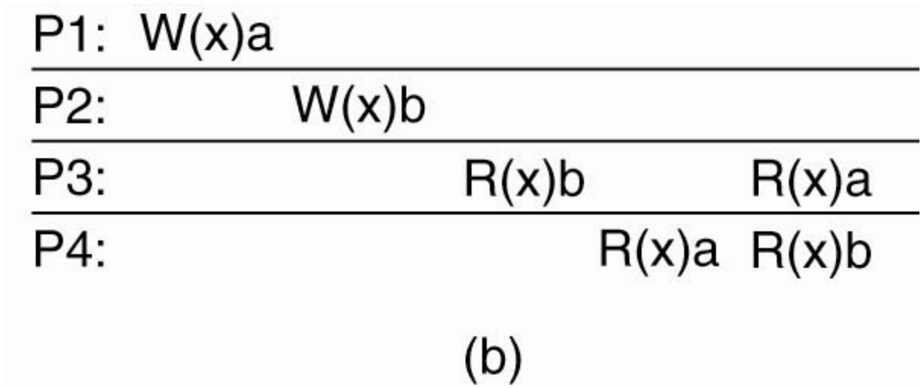
P1:	W(x)a		
P2:		R(x)NIL	R(x)a

- According to sequential consistency this behavior is acceptable

Sequential Consistency



(a) A sequentially consistent data store.



(b) A data store that is *not* sequentially consistent.

Sequential Consistency

- Example: Three concurrently-executing processes.

Process P1	Process P2	Process P3
x ← 1;	y ← 1;	z ← 1;
print(y, z);	print(x, z);	print(x, y);

- Assuming each line is indivisible, statements can be executed in 720 (= 6!) different orderings
  - Some of them are not correct: print(y,z) can not be executed before x ← 1
  - Totally 90 correct mutations exists

Sequential Consistency

- Example
  - The vertical axis is time → 64 unique answers is produced and based on the consistency model all of them are correct.

x ← 1;	x ← 1;	y ← 1;	y ← 1;
print(y, z);	y ← 1;	z ← 1;	x ← 1;
y ← 1;	print(x, z);	print(x, y);	z ← 1;
print(x, z);	print(y, z);	print(x, z);	print(x, z);
z ← 1;	z ← 1;	x ← 1;	print(y, z);
print(x, y);	print(x, y);	print(y, z);	print(x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
Signature: 001011	Signature: 101011	Signature: 110101	Signature: 111111
(a)	(b)	(c)	(d)

Causal Consistency

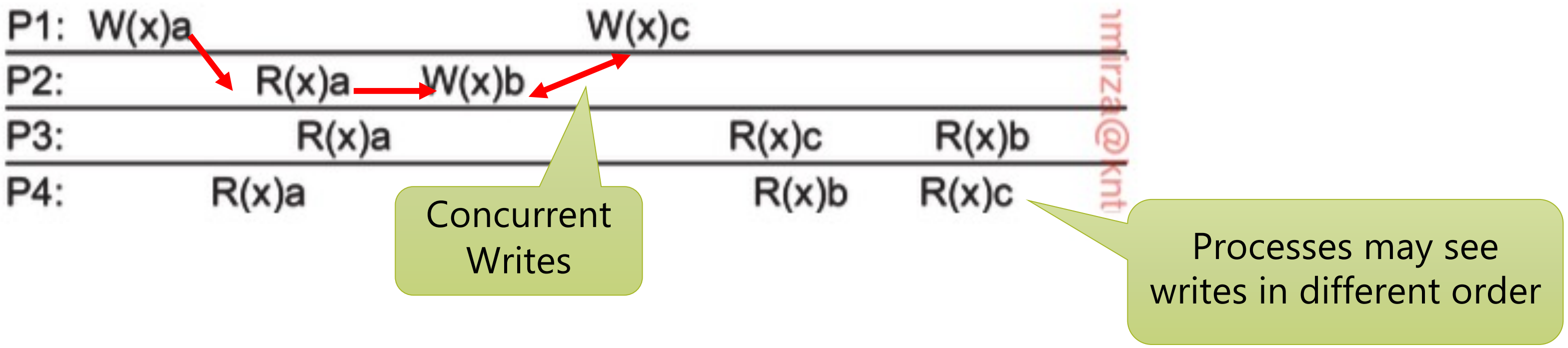
- Makes a distinction between events that are potentially causally related and those that are not
- If event *b* is caused or influenced by an earlier event *a*, causality requires that everyone else first see *a*, then see *b*.
- Operations not causally related are concurrent

Causal Consistency

- Casually related writes must be seen by all processes in the same order

- Concurrent writes, may be seen in a different order on different machines

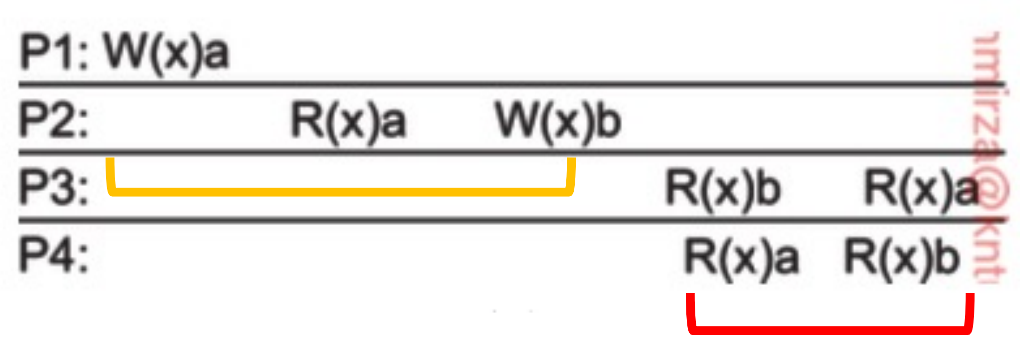
Causal Consistency



- $W_2(x)b \leftarrow R_2(x)a \leftarrow W_1(x)a$: causal dependency → all processes must see them in the same order.

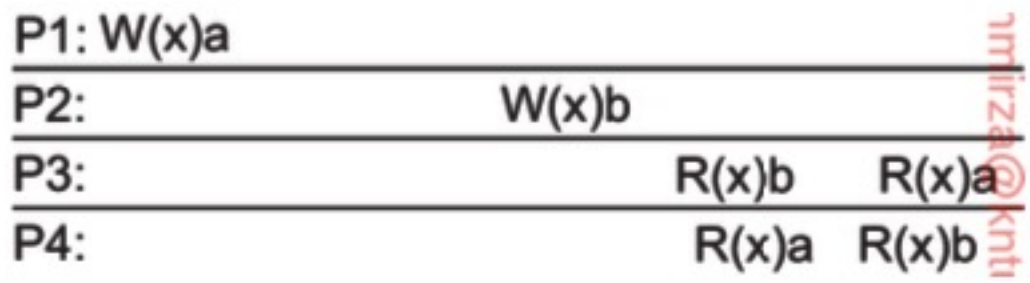
- $W(x)c$ *and* $W(x)b$ are concurrent, it is not required that all processes see them in the same order

Causal Consistency



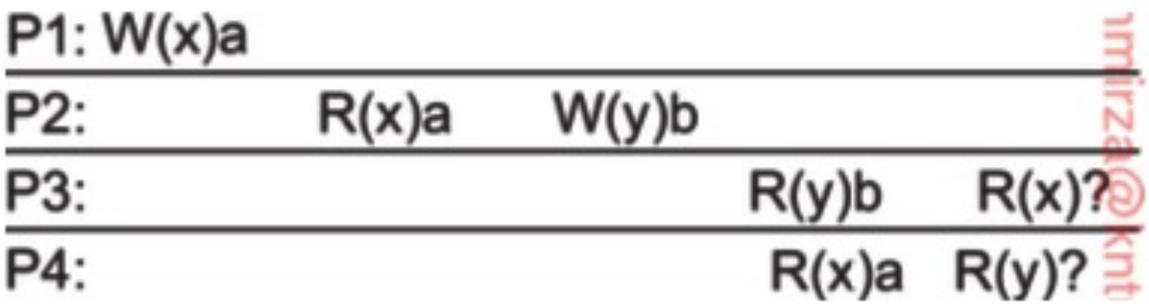
- Writing b depends on reading value of a, then they are casually dependent
- Violation has been occurred P3 and P4 must see equal value for X

Causal Consistency



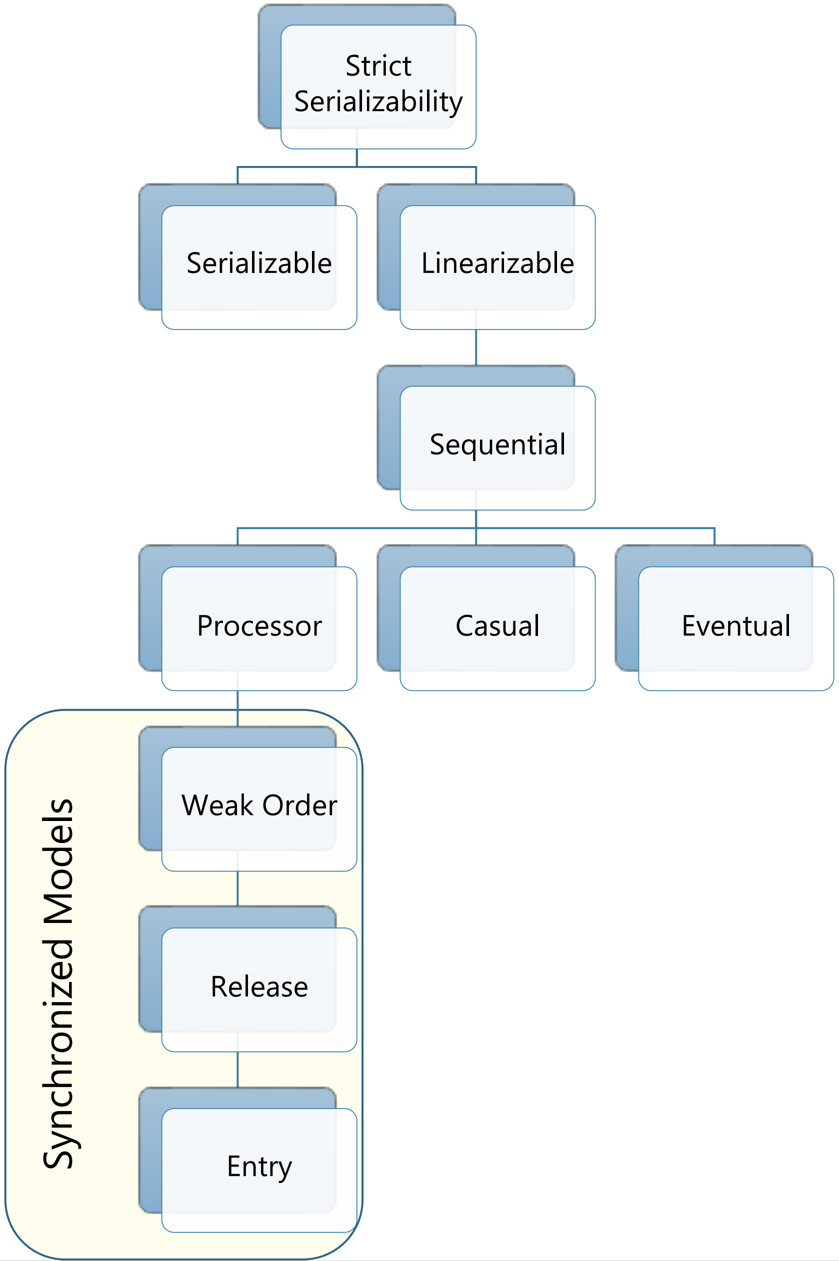
- Store is causally consistent, because W(x)a and W(x)b are concurrent
- Note that the store is not sequentially consistent

Causal Consistency



- $R_3(x)$?
  - W(x)a *happened before* W(y)b (W(x)a → R(x)a → W(y)b) ➔ $R_3(x)a$
- $R_4(y)$?
  - Trivially $R_4(y)b$ is correct
  - But $R_4(y)NIL$ is also correct!

Hierarchy of Consistency Models



Processor Consistency

- All writes to the same memory location must be seen in the same sequential order by all other processes.

Weak Order Consistency

- Write operations before critical section must be globally performed
- All operations in all processors need to be visible before critical section
- Write operations inside the critical section performed only after the critical section completes
- All other operations can be reordered

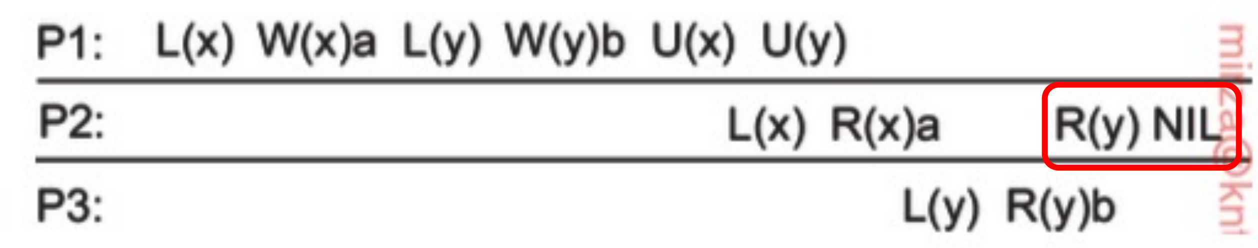
Release Consistency

- During the entry to a critical section, all operations with respect to the local memory variables need to be completed.

Entry Consistency

- Every shared variable is assigned a synchronization variable specific to it.
- Before critical section all operations related to x need to be completed with respect to that process

Entry Consistency



- It is associated with lock/unlock operations
- $L(x) \rightarrow Lock(x)$
- $U(x) \rightarrow Unlock(x)$
- Each process has its own copy of variables
- When they read variables as usual, they may read their own copy
- Acquiring locks means, underlying distributed system must synchronize the copies of the variable

Eventual Consistency

- Observed in practice
  - Only few processes do update operation
  - Chance of write-write conflict is very rare
  - Most of the operations is read
- Examples
  - DNS record: only the authority updates, write-write conflict never occurs!
  - Web pages: only the admin updates a page, write-write conflict never occurs!
- An important issue in these scenarios is when (how fast) the update is propagated into other replicas or local caches
  - When browser caches, or local DNS servers get the updated content

Eventual Consistency

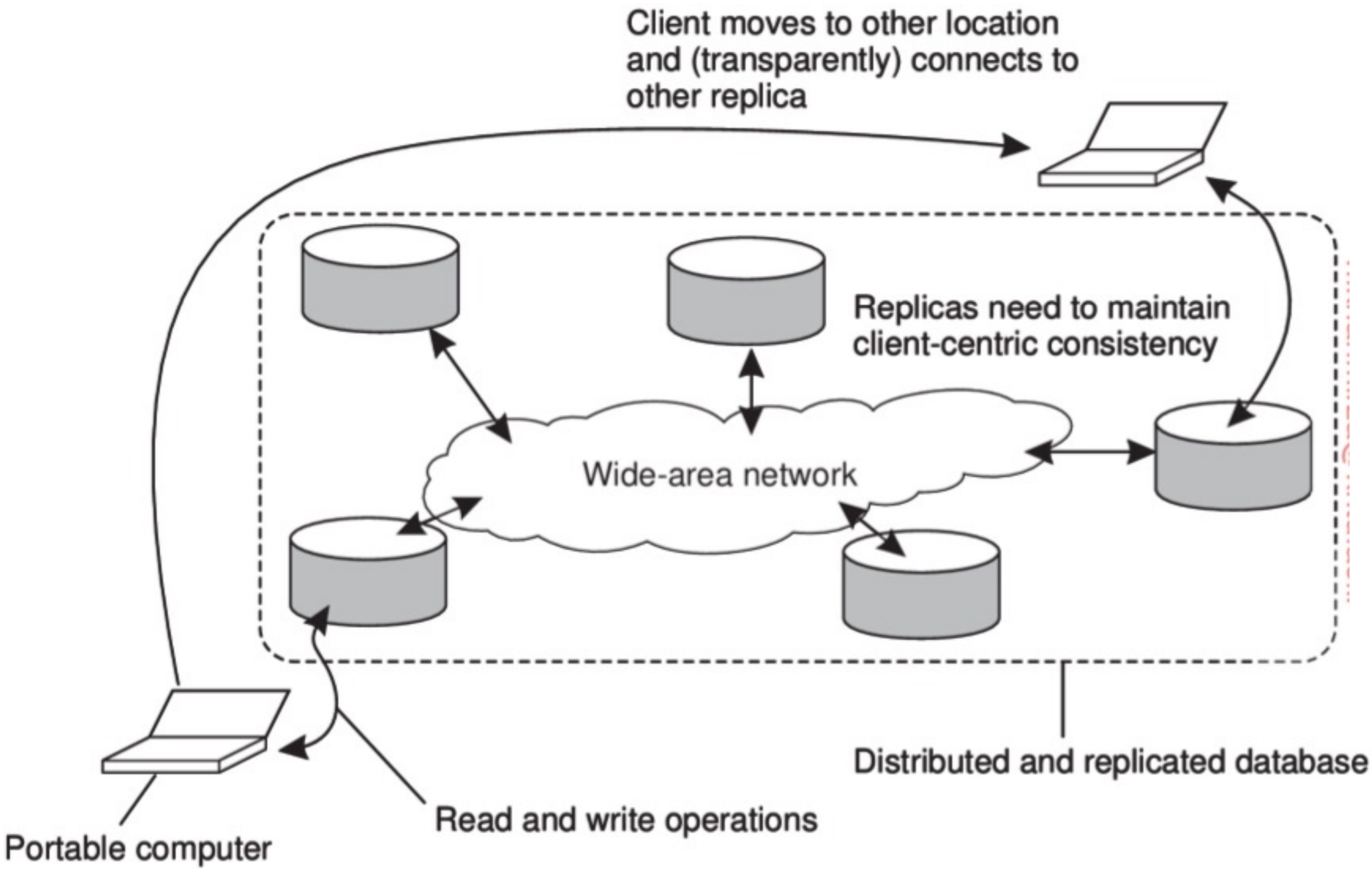
- There are large-scale distributed and replicated systems that tolerate a relatively high degree of inconsistency
- Reading stale data for a period of time is acceptable, updates can be lazily propagated
- If no updates take place for a long time, all replicas will gradually become consistent, sometime in future
- Eventual consistency essentially requires only updates are guaranteed to propagate to all replicas
  - Eventual consistency relaxes the consistency, with in write-write conflicts
  - It is used in iPhone sync, Dropbox, git, Amazon Dynamo, Cassandra, ONOS, ..

Client-Centric Consistency

- Special class of distributed data stores
  - Mostly read, updated by one admin
  - No shared data

- Provides guarantees for a single client concerning the consistency of accesses to a data store by that client

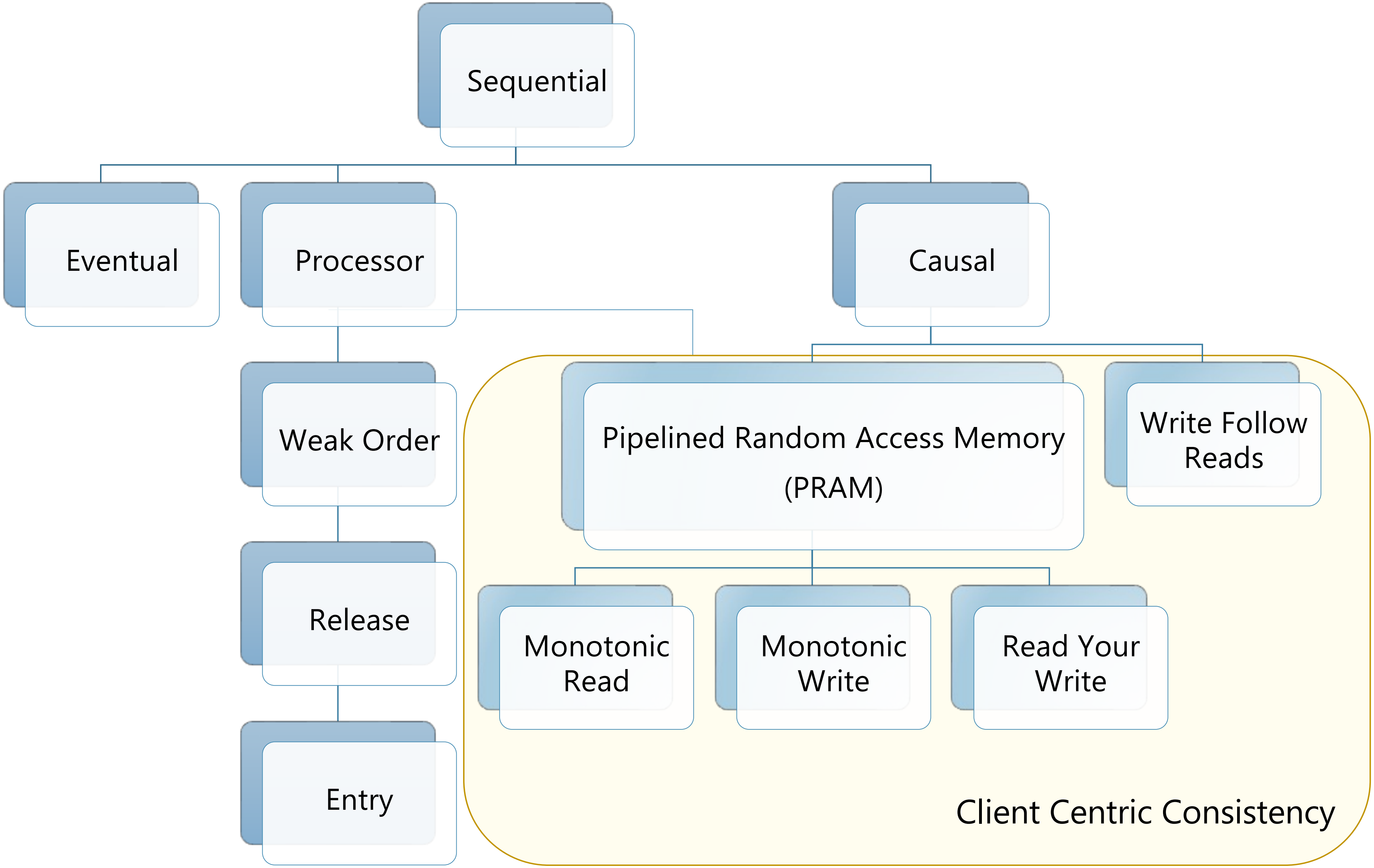
Client-Centric Consistency



Client-Centric Consistency

- Eventual consistency works fine as long as user accesses one replica
- If a user is mobile and accesses several replicas in a short time, eventual consistency is no longer held
- Consider a mobile user that modifies data in an store then disconnects and moves, after a while connects to another store and modifies some data, which creates write-write conflict!

Hierarchy of Consistency Models



PRAM Consistency

- Pipelined Random Access Memory
- Also known as FIFO consistency

- Writes executed by a single process are observed by other processes in the order the process executed them as if they were in the pipeline.

- Writes from different processes may be seen in a different order by different processes

- PRAM is a combination of the next three consistencies

PRAM Consistency

- Implementation:
- Force a process always write to one particular data store
- or
- Before each write ensure the previous write is propagated to all other stores

Client-Centric Consistency Notations

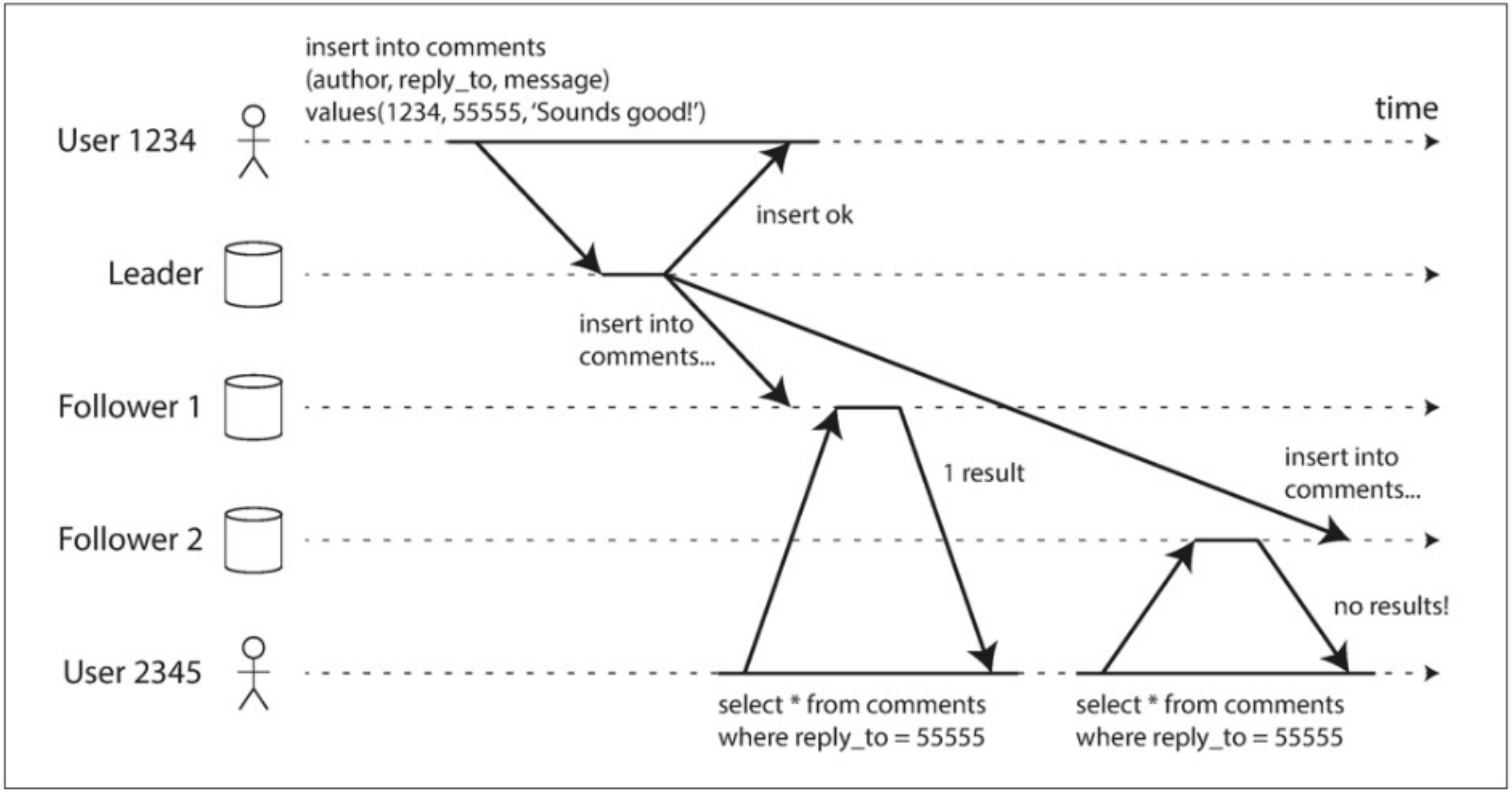
- Notations
  - X: data item
  - X_i: i^{th} version of x
  - $WS(x_i)$: A series of writes has leaded to x_i (i^{th} version of x)
  - $WS(x_i; x_j)$: By appending series of writes on x_i, version x_j is obtained
  - $WS(x_i|x_j)$: We don't know if x_j follows from x_i
  - $W_1(x_1)a$*:* process P1 wrote value *a* to *x* and produces version 1 of x
  - $R_1(x_2)$ simply means that P_1 reads version x_2
  - L_i: i^{th} data store

Monotonic-Read Consistency

- If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value

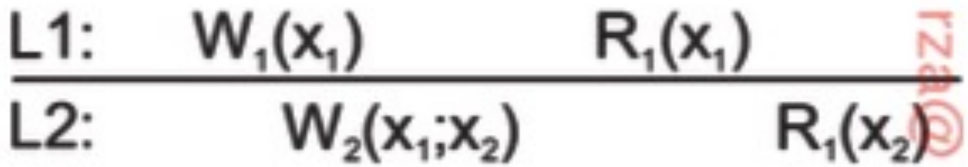
- This guarantees once a process has seen a value of x, it will never see an older version of x

Monotonic-Read Consistency



[4]

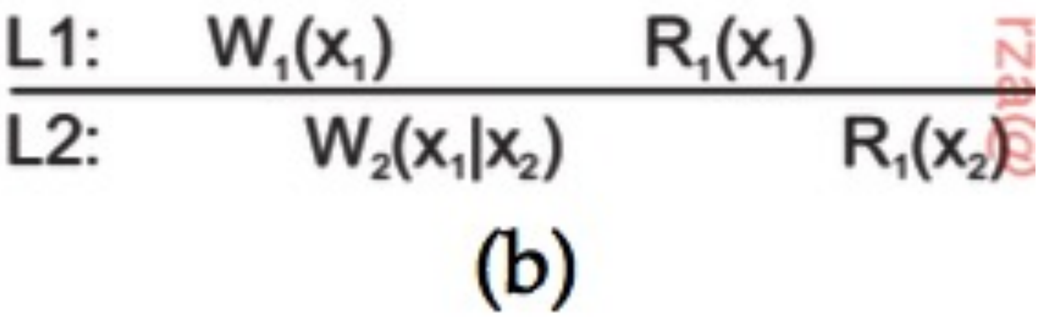
Monotonic-Read Consistency



(a)

- You open the mailbox you see some unread emails
- From then you should always see at least the same unread messages from every where
- You may see newer emails or not

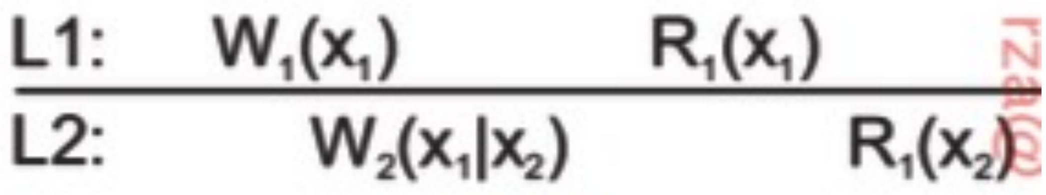
Monotonic-Read Consistency



- You’ve registered in a multi-branch sport club
- First you enroll for swimming
- Later you decide to enroll for body-building in an another branch
- In that branch, they say you’ve not enrolled for swimming!

- The process (you) reads the most recent data, does it implies monotonic read?

Monotonic-Read Consistency



(b)

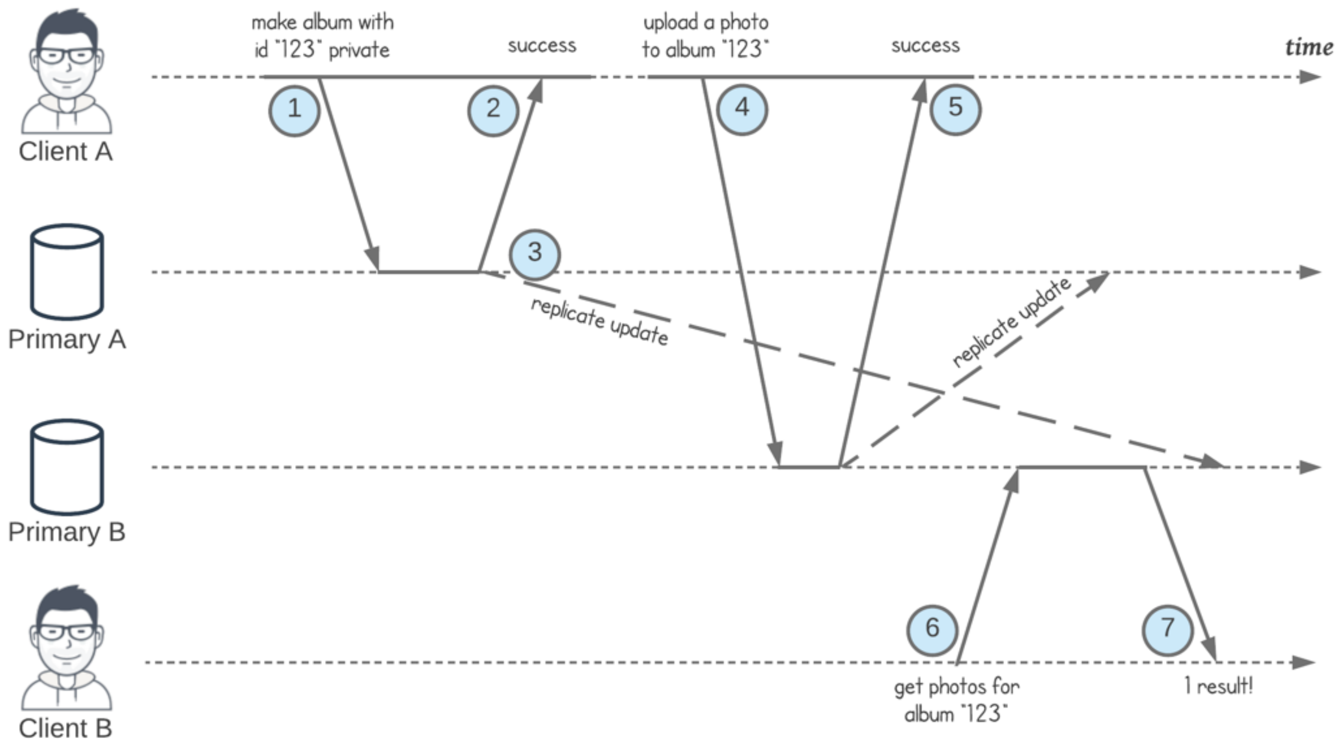
- Remember monotonic read is descended from Causal

- Reading a recent value must include all of the writes led to this value

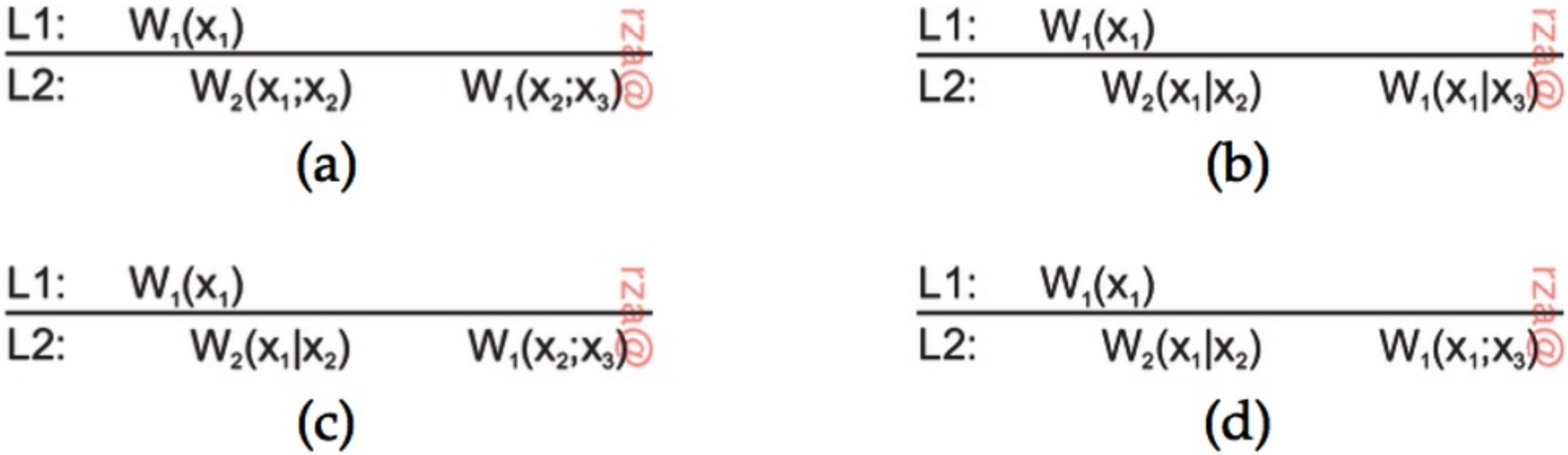
Monotonic-Write Consistency

- A write operation by a process on a data item x is completed before any successive write operation on x by the same process
- Write operation on a copy of item x is performed only if that copy has been brought up to date.
- if a process performs write w_1, then w_2, then all processes observe w_1 before w_2.
- FIFO ordering of writes

Monotonic-Write Consistency



Monotonic-Write Consistency



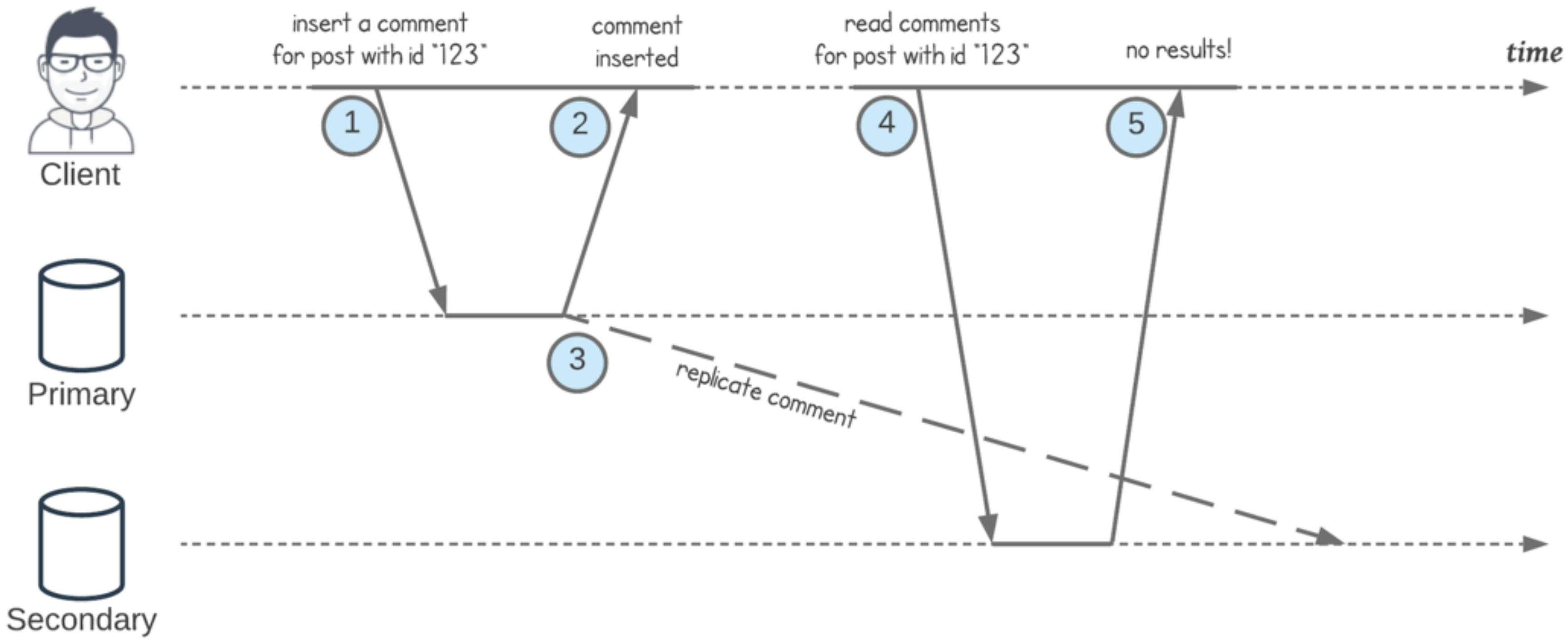
Read Your Write Consistency

- Also read-my-writes

- The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process

- A write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place

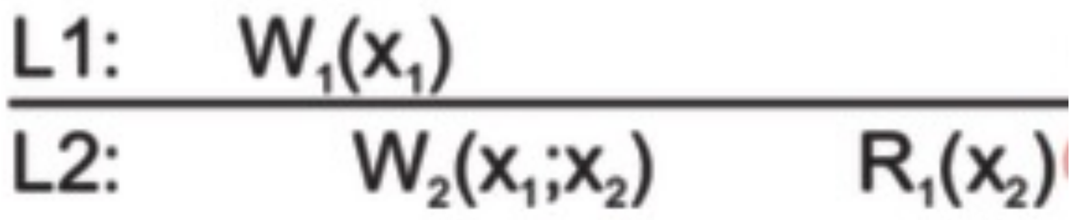
Read Your Write Consistency



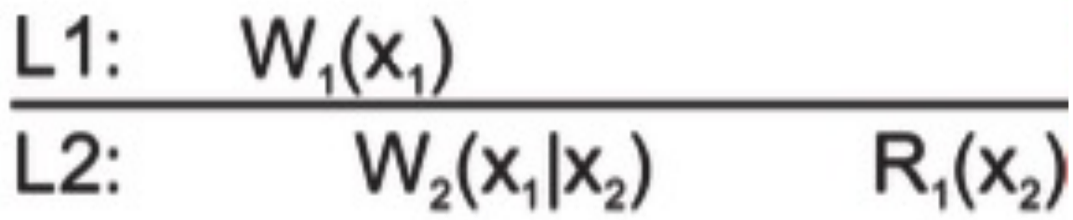
Read Your Write Consistency

- Example
- You update your personal web-page
- You refresh the page but the most recent version is not shown
- Previous page is cached in browser
- With this consistency, all cached versions must be invalidated

Read Your Write Consistency



(a)



(b)

Writes Follow Reads Consistency

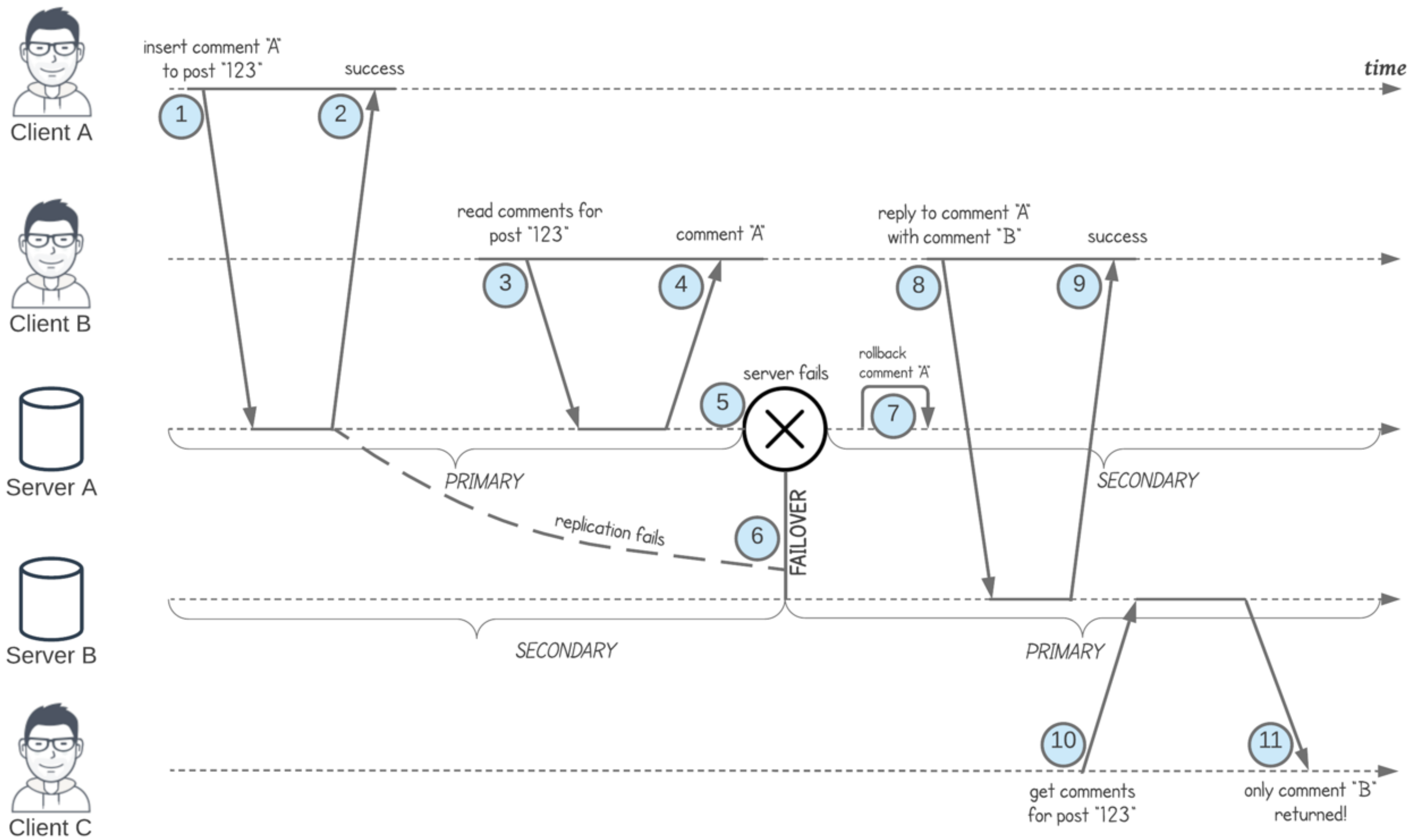
- Also known as session causality

- If a process reads a value v, caused by write w_1, and later performs write w_2
- then w_2 must be visible after w_1.
- Once you've read something, you can't change that read's past.

Writes Follow Reads Consistency

- Example
- Assume a user first reads an article A.
- Then, reacts by posting a response B.
- By requiring writes-follow-reads consistency, B will be written to any copy of the newsgroup only after A has been written as well
- Guarantees users of a group see a posting of a reaction to an article only after they have seen the original article

Writes Follow Reads Consistency



Writes Follow Reads Consistency

$$
\begin{array}{llll}
\text{L1:} & W_1(x_1) & & R_2(x_1) \\
\hline
\text{L2:} & & W_3(x_1;x_2) & W_2(x_2;x_3)
\end{array}
$$

(a)

$$
\begin{array}{llll}
\text{L1:} & W_1(x_1) & & R_2(x_1) \\
\hline
\text{L2:} & & W_3(x_1|x_2) & W_2(x_1|x_3)
\end{array}
$$

(b)

Writes Follow Reads Consistency

- Causal Consistency only for one process and W-R-W sequence
- Re-ordering of actions of other processes is possible

Replica Server Location

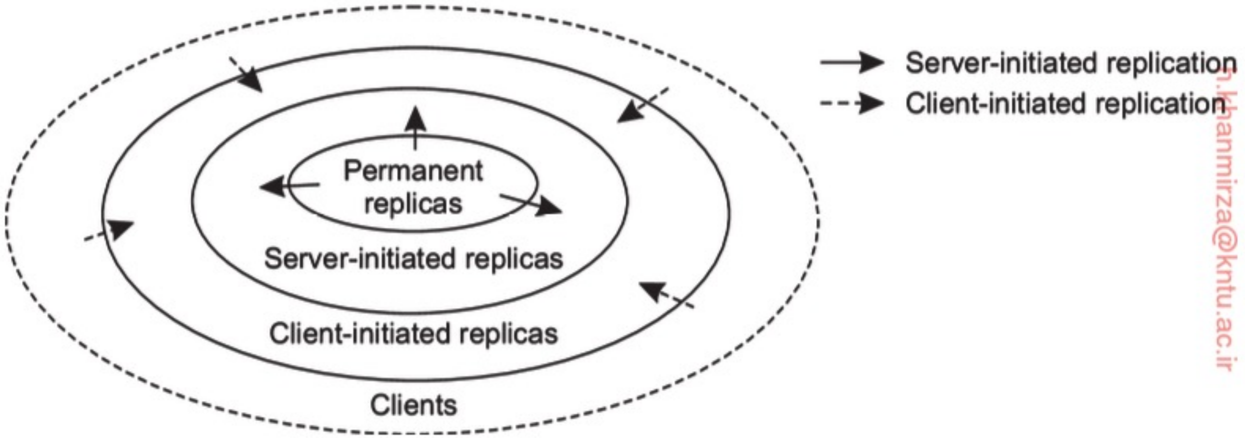
- With the advent of the many large-scale data centers located across the Internet and constant improvement of connectivity, precisely locating servers is less critical.

- It is more of a management and commercial issue than a scientific problem

- This issue maybe a real concern in Wireless or Sensor Networks
  - The problem become similar to choosing cluster head problems

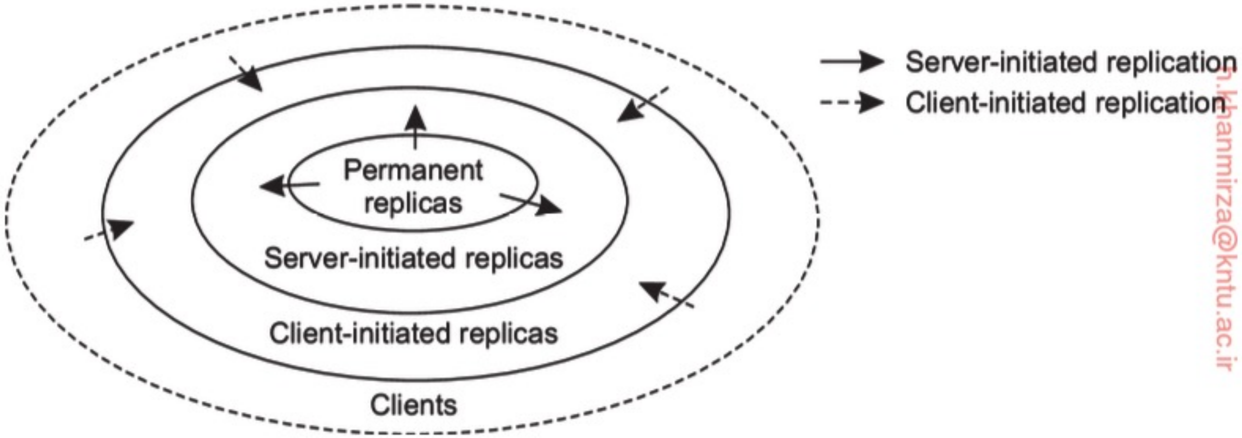
Content Replication & Placement

- Permanent Replicas
  - Several servers in one location (cluster)
  - Several servers in different locations (Site Mirroring)



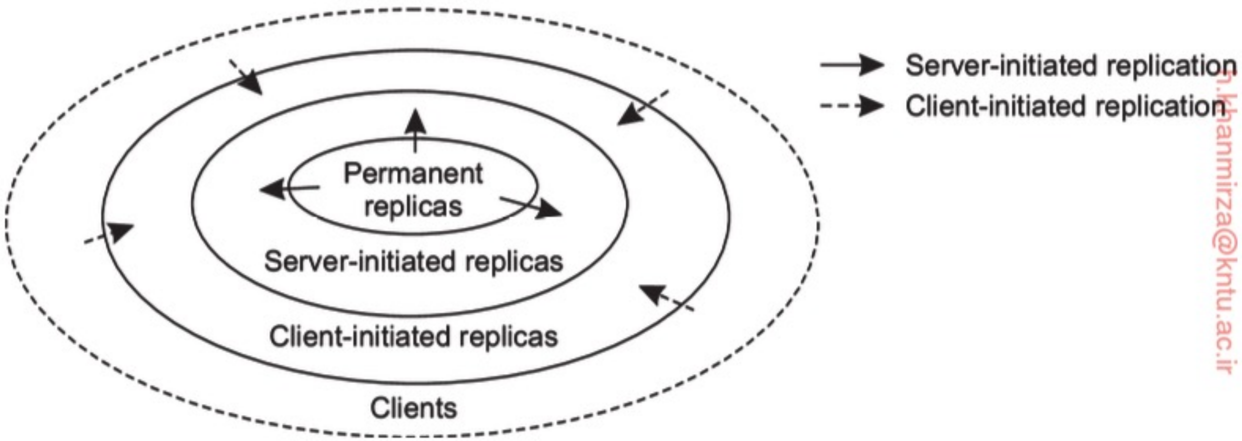
Content Replication & Placement

- Server-initiated Replicas
  - Server-initiated replicas are copies of a data store that exist to enhance performance, and created at the initiative of the owner of the data store



Content Replication & Placement

- Client-initiated Replicas
  - Local caches in client
  - Local caches for a site (cache servers in a LAN)
  - Best for mostly-read, static data
  - Because of network connectivity improvements, nowadays, is less attractive



Content Distribution

- When an update is performed by a client? what should be propagated?
- State vs. operation
- Propagate only a notification of an update
  - Known as invalidation protocols
  - Just notify some part of data is updated
  - Use little bandwidth
  - Useful when write_count >> read_count
  - Otherwise, large updates are replicated throughout the network without being read

Content Distribution

- Transfer data from one copy to another
  - Transfer the new data to other replicas
  - Useful when write_count << read_count
  - It is possible to send logs of changes instead of the data itself,
    - increases chance of aggregating logs of several updates into one packet

- Propagate the update operation to other copies
  - Send parameter values and the operation other replicas must do

Content Distribution

- Push or Pull updates?
  - Push-based (server-based protocols)
    - Updates are propagated to other replicas without their asking
    - Used between permanent and server-initiated replicas
    - Need for strong consistency
    - Efficient for high read-to-write ratio
  - Pull-based (client-based protocols):
    - A server or client requests another server to send it all updates up to now
    - Mostly, used for client caches
    - Efficient for low read-to-update conditions
  - Hybrid protocols: Lease-based model
    - Server pushes updates for a specific period of time
    - When lease expires, client must poll the server

Consistency Protocols

- A consistency protocol describes an implementation of a specific consistency model

- Based on experience, simpler methods succeed even if the complex methods have better performance

- Categories
  - Primary-based Protocols
  - Replicated-Write Protocols

Primary-based Protocols

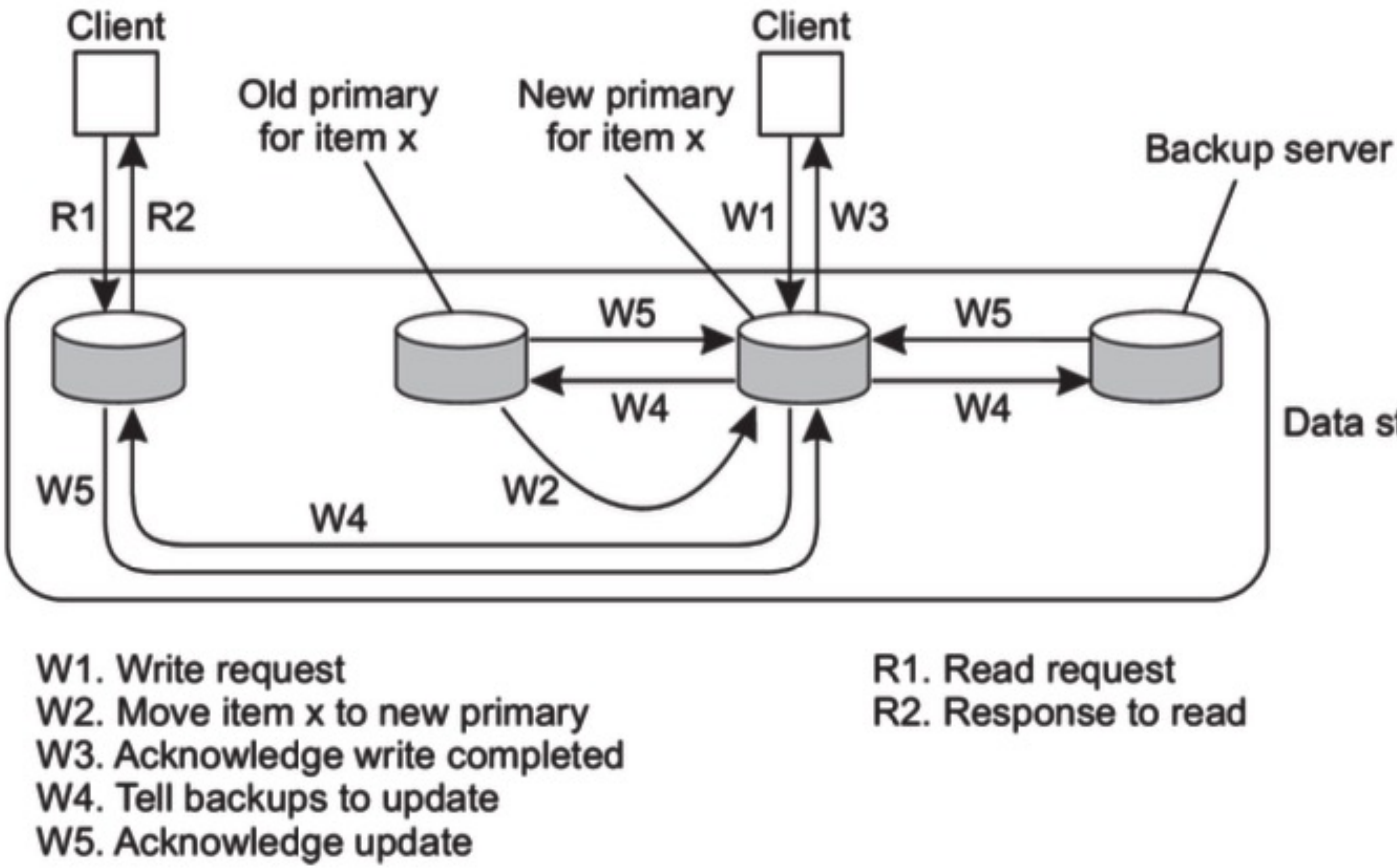
- Each data item in the data store has an associated primary, which is responsible for coordinating write operations

Primary-based Protocols

- Remote-Write or Primary-Backup Protocol
  - Updates forwarded to one server which is responsible for that data item
  - When update is performed, it forwards the update to all backups
    - Then, backups acknowledge the server, their reception
  - All reads are done locally
  - A straightforward implementation of sequential consistency
    - As the primary can order all incoming writes in a globally unique time order.
  - If update is implemented as blocking, processes will see the effects of the most recent write.

Primary-based Protocols

- Local-Write Protocols



Primary-based Protocols

- Local-Write Protocols
  - When a process wants to update a data item, it locates the primary copy of data, and moves it to its own location
  - Advantage: multiple, successive write operations can be carried out locally, while reading processes can still access their local copy
  - It can be used for disconnected operations like mobile clients
    - Before disconnecting a mobile system become primary
    - Others can only read the data store
    - After connecting, the system updates other backups

Primary-based Protocols

- Primary-backup protocols have poor response time

- Why we don’t write updates to several copies? → Replicated write protocols

Replicated-write Protocols

- Active Replication
  - Write operation is sent to all replicas (not the updates)
  - This scheme needs global ordering
    - Totally-ordered multicast
    - Practical implementations
      - Updates are sent to a central sequencer, which assigns order and sends update to all replicas
      - For scalability, we can use several sequencers using Lamport's total-ordering mechanism, a group of processes work with a sequencer

Replicated-write Protocols

- Quorum-based protocols
  - Replicated writes with voting!
  - Clients must send their request and acquire the permission of multiple servers before reading or writing a replicated data item
  - To write a data, agreement of at least $\frac{N}{2}+1$ replicas should be achieved
    - After update a new version number is assigned with the data
  - To read a data, client contacts at least $\frac{N}{2}+1$ replicas and asks for the version number
    - If all the version numbers are the same, this must be the most recent version

Replicated-write Protocols

- General Quorum-based protocols

  - For reading, a client must assemble a collection of N_R replicas: read quorum

  - For writing, a client must assemble a collection of N_W replicas: write quorum

  - The following conditions must be satisfied:
    - $N_R + N_W > N$ ➔ prevents read-write conflicts
    - $N_W > \frac{N}{2}$ ➔ prevents write-write conflicts

Replicated-write Protocols

- Quorum-based protocols

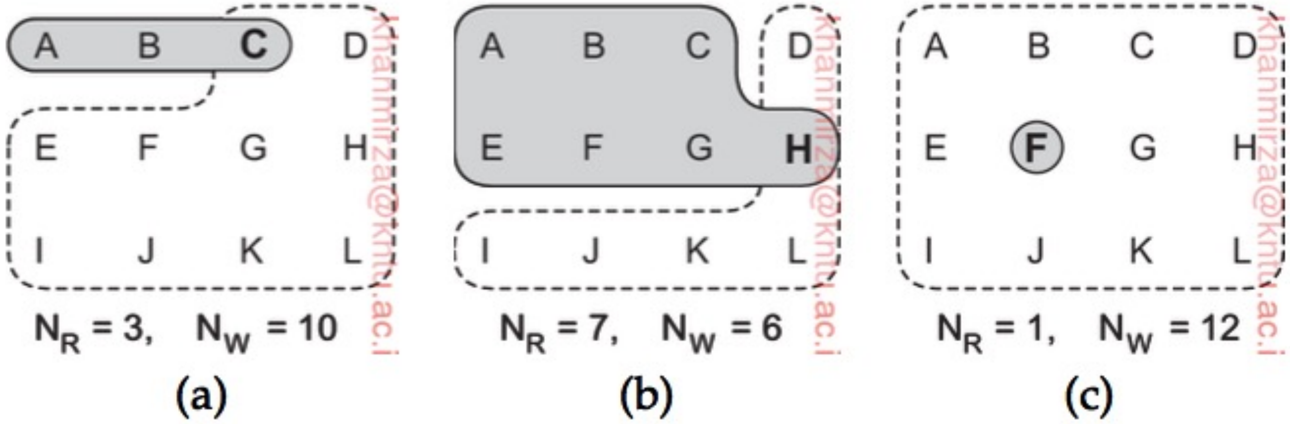


Figure 7.29: Three examples of the voting algorithm. The gray areas denote a read quorum; the white ones a write quorum. Servers in the intersection are denoted in boldface. (a) A correct choice of read and write set. (b) A choice that may lead to write-write conflicts. (c) A correct choice, known as ROWA (read one, write all).

Coherence

- Consistency is concerned with a set of data items

- The copies of a data item are coherent when the various copies conform to the rules as defined by its associated consistency model

- Deals with only a single data item
  - Mostly studied in caches of shared memory multi-processor/chip-multi-processors context
  - They have hardware support

Other References

1. https://jepsen.io/consistency
2. https://en.wikipedia.org/wiki/Consistency_model
3. Viotti, Paolo, and Marko Vukolić. "Consistency in non-transactional distributed storage systems." *ACM Computing Surveys (CSUR)* 49.1 (2016): 1-34.
4. Kleppmann, Martin. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.
5. https://vkontech.com/causal-consistency-guarantees-case-studies/

The End!