
5. Consistency and Replication

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2021/2022 Q2

Contents

- **Introducing replication & consistency**
- Data-centric consistency models
- Client-centric consistency models
- Distribution protocols
- Consistency protocols

Introduction

- Reasons for replication
 - A. Increase **availability**: data is available despite server failures and network partitions
 - B. Enhance **reliability**: data is correct on the presence of faults (e.g. protection against data corruption, Byzantine failures, and stale data)
 - C. Improve **performance**: this directly supports the goal of enhanced **scalability**
 - Size: Replicate data and distribute work instead of having one single server
 - Geographical: Replicate data close to where it is used (e.g. data caching)

Introduction

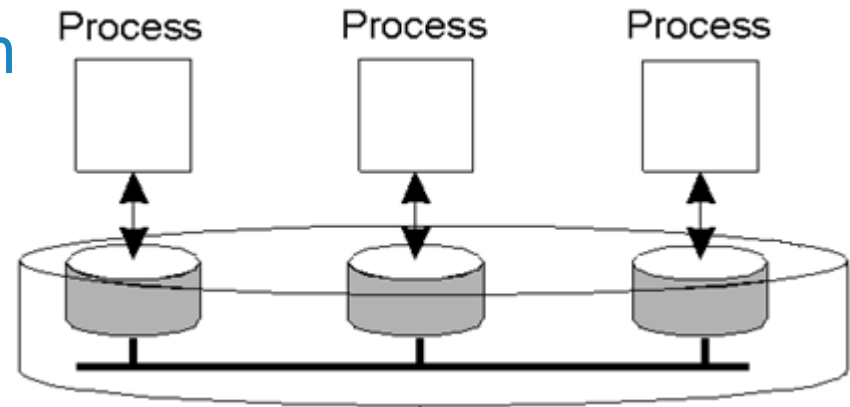
- What are the issues with replication?
 - Replication should be **transparent**
 - Clients are not aware that multiple copies of data exist
 - Replicated data should be **consistent**
 - Clients see the same data despite the replica they read
- ⇒ How do we transparently (and efficiently) keep all the replicas up-to-date and consistent?
- Dilemma: replication improves scalability, but incurs the overhead of synchronizing replicas
 - The solution often results in a relaxation of the consistency constraints

Contents

- Introducing replication & consistency
- **Data-centric consistency models**
- Client-centric consistency models
- Distribution protocols
- Consistency protocols

Consistency models

- Distributed data store
 - Distributed across multiple nodes in a replicated fashion
 - Each replica holds a copy of the entire store
 - Each process can access one or several replicas
 - Write operations to a replica are propagated to the others
- A consistency model specifies precisely what the results of concurrent reads/writes are



Consistency models

- Diagram notation
- Time axis drawn horizontally with time increasing from left to right in all diagrams
- **$W_i(x)a$** : a write by process 'i' to item 'x' with a value 'a'
 - i.e. 'x' is set to 'a'
 - Note: The process is often shown as ' P_i '
- **$R_i(x)b$** : a read by process 'i' from item 'x' returning the value 'b'
 - i.e. reading 'x' returns 'b'

Strict consistency

- Meet sequential specification of each item 'x'

EXCLUSIVE
ORDERED
WRITES

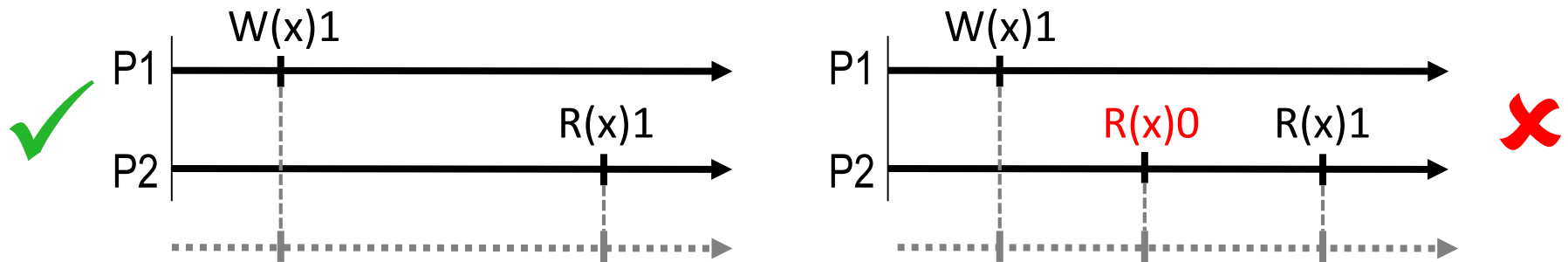
– Given 2 concurrent writes on 'x', one must go first

– A read on 'x' returns the value of the most recent write on 'x', regardless of the used replica

UP-TO-DATE
MONOTONIC
READS

– Subsequent reads on 'x' return the same value or some later value, regardless of the used replica

- Operations run in a unique legal sequence that matches their **real-time** occurrence

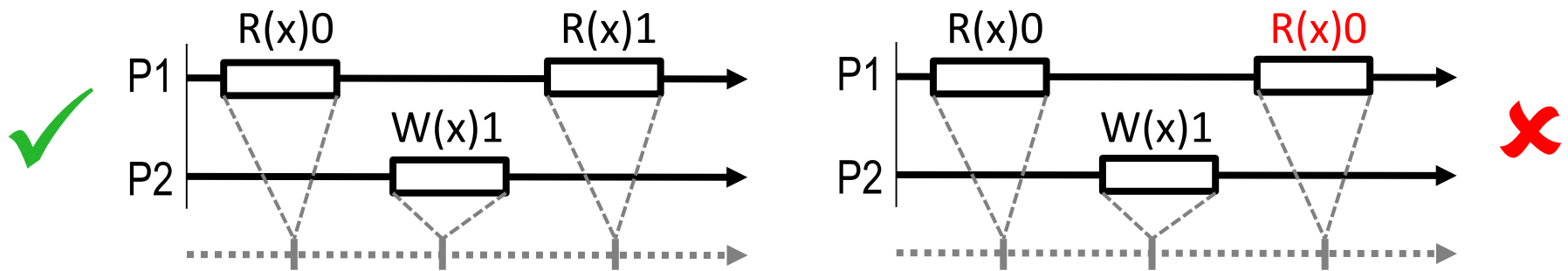


Strict consistency

- Theoretical model that corresponds to full replication transparency
 - Also the model of uniprocessor systems
- Impossible to achieve in a distributed system
 1. Writes must be instantly visible in all replicas
 - ⇒ Network latencies preclude this
 2. Requires an absolute global time such that at most one operation occurs at a time
 - ⇒ Clocks can neither be perfectly synchronized nor have infinite precision

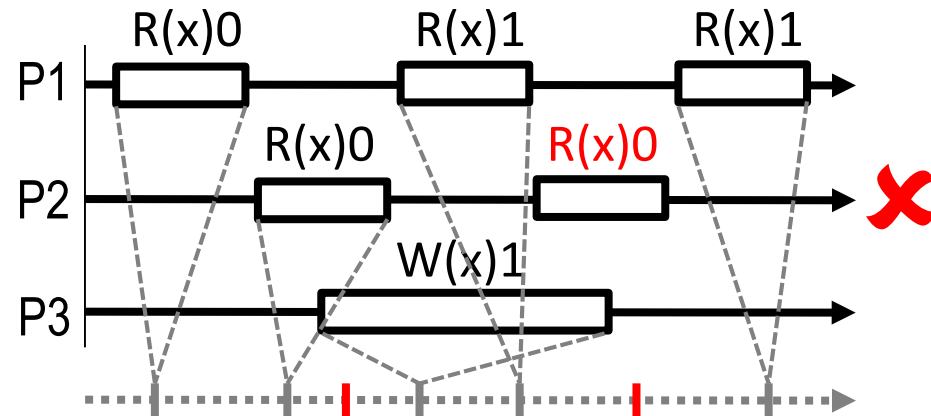
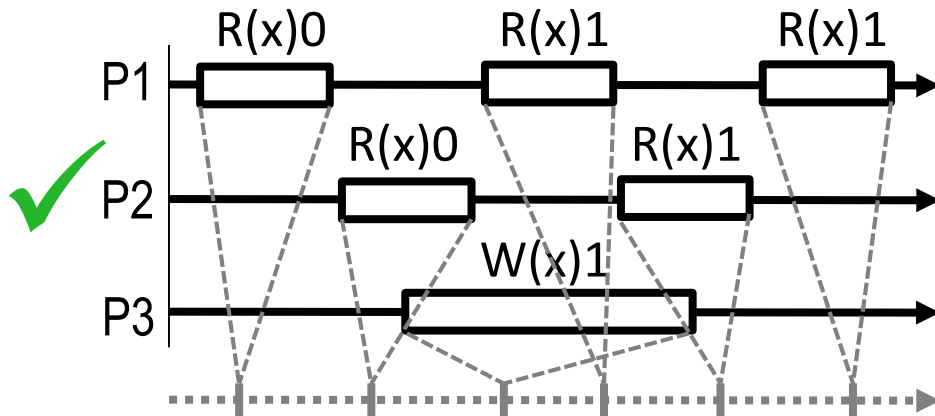
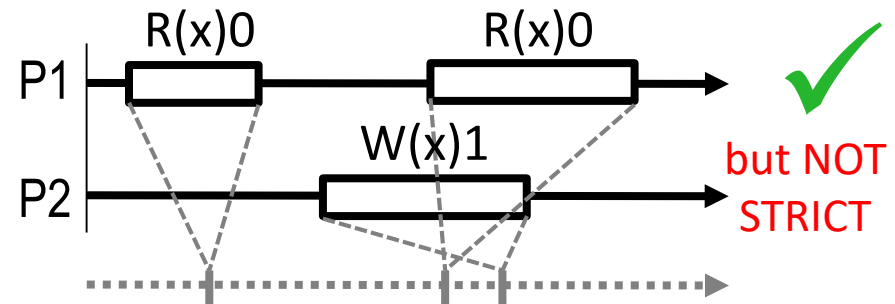
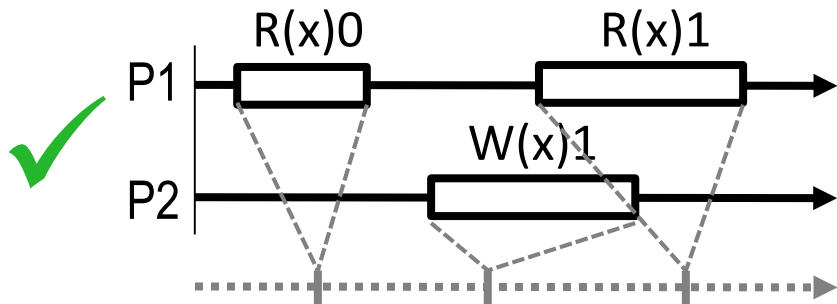
Linearizability (a.k.a. strong consistency)

- Same goal as strict consistency (exclusive ordered writes + up-to-date monotonic reads) but recognizing that operations take time to complete and can overlap
- Operations appear to run in some common sequence that retains their real-time order when they do not overlap



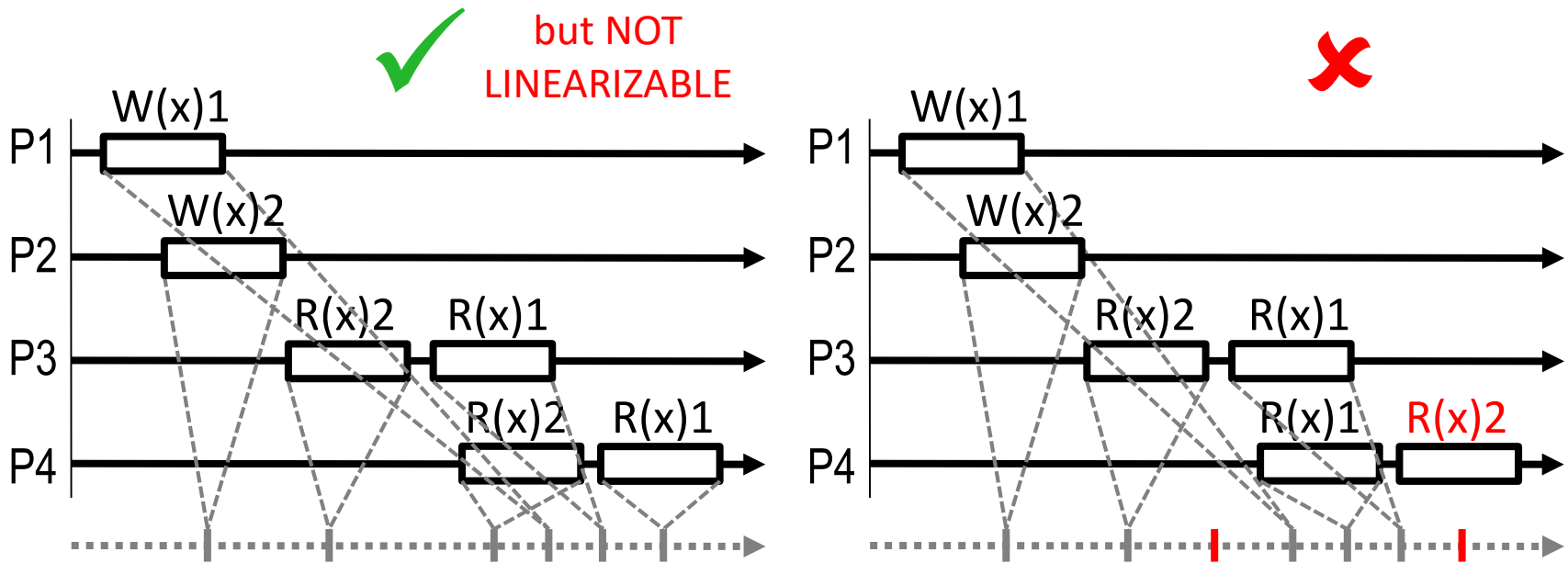
Linearizability (a.k.a. strong consistency)

- Overlapping ops. can take effect in any order
 - But their effect must become visible to everyone before they return (appearing to be atomic)



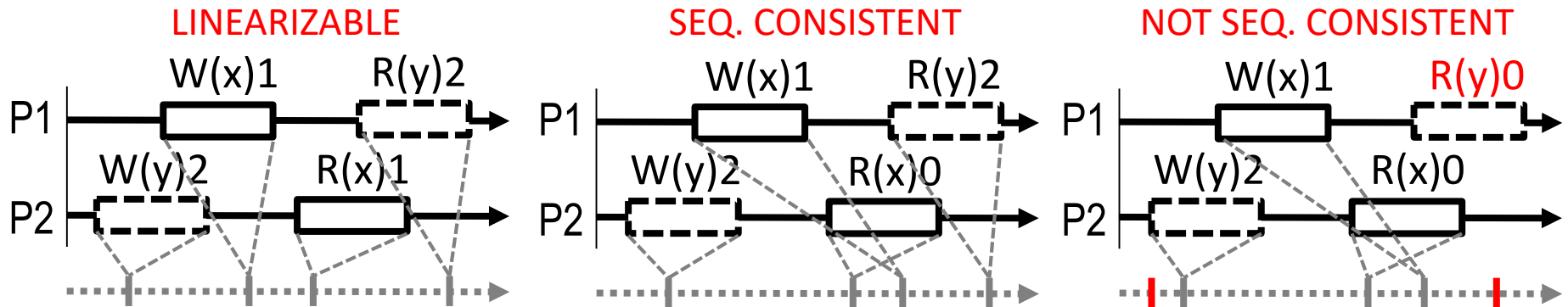
Sequential consistency

- Operations appear to run in some common sequence that retains the program order for the operations of each process
 - Real time is not taken into consideration



Linearizability vs. sequential consistency

- Only linearizability is compositional
 - Separated linearizable histories of objects can be composed in a linearizable history



- Both can be provided in distributed systems
 - Linearizability is stronger and more expensive
 - $\text{Strict} \subset \text{Linearizability} \subset \text{Sequential}$
 - See 'Consistency protocols' lesson for details

Other consistency models

- Causal consistency
 - Writes that are potentially causally related must be seen by all processes in the same order
- FIFO consistency
 - Writes done by a single process are seen by all the others in the order in which they were issued
- Weak consistency
 - Enforce consistency on groups of operations (i.e. critical section) instead of individual reads/writes
 - Operate on **synchronization variables** (acquire, release) to synchronize all the replicas

Contents

- Introducing replication & consistency
- Data-centric consistency models
- **Client-centric consistency models**
- Distribution protocols
- Consistency protocols

Eventual consistency

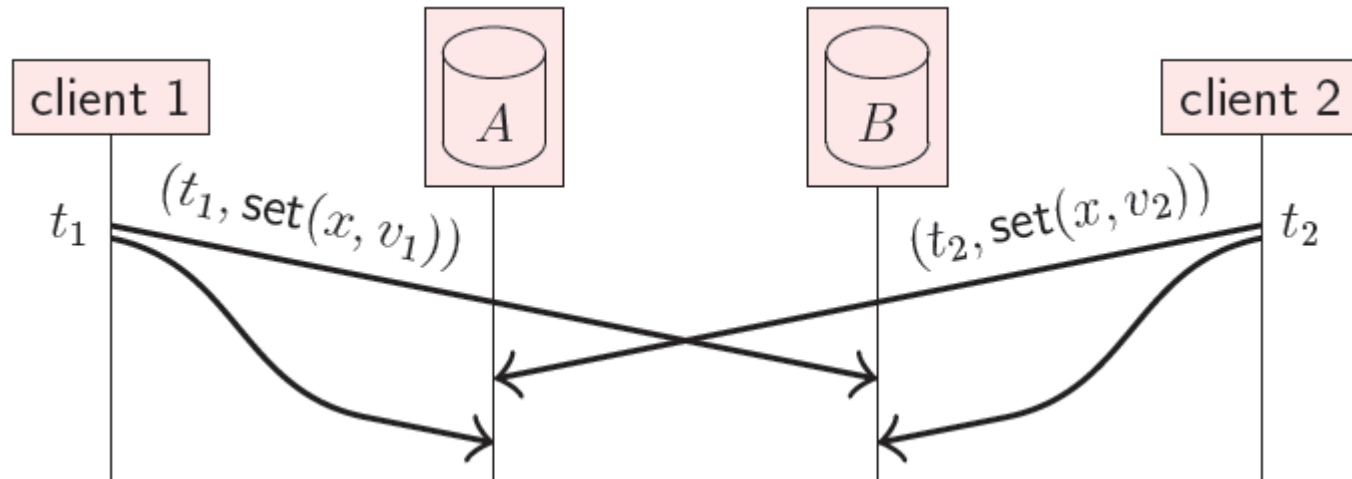
- Data-centric models aim to provide a system-wide consistent view on data stores with concurrent write operations
 - These models require highly available connections
- **Eventual consistency** targets stores that:
 - Execute mainly read operations and have none or few write-write conflicts
 - Have users that often accept reading stale data
 - Deal with disconnected users, network partitions, and users preferring availability vs. consistency
 - e.g. DNS, Amazon Dynamo, Dropbox

Eventual consistency

- Def: In absence of new updates, eventually all replicas will converge to identical values
- Each client operates on a replica and updates propagate to other replicas asynchronously, when connections are available
- Writes might arrive in different orders to the replicas: potential write-write conflicts
 - Replicas must apply the updates in the same order to ensure eventual convergence \Rightarrow this requires consistent conflict resolution for concurrent writes

Conflict resolution: Last writer wins

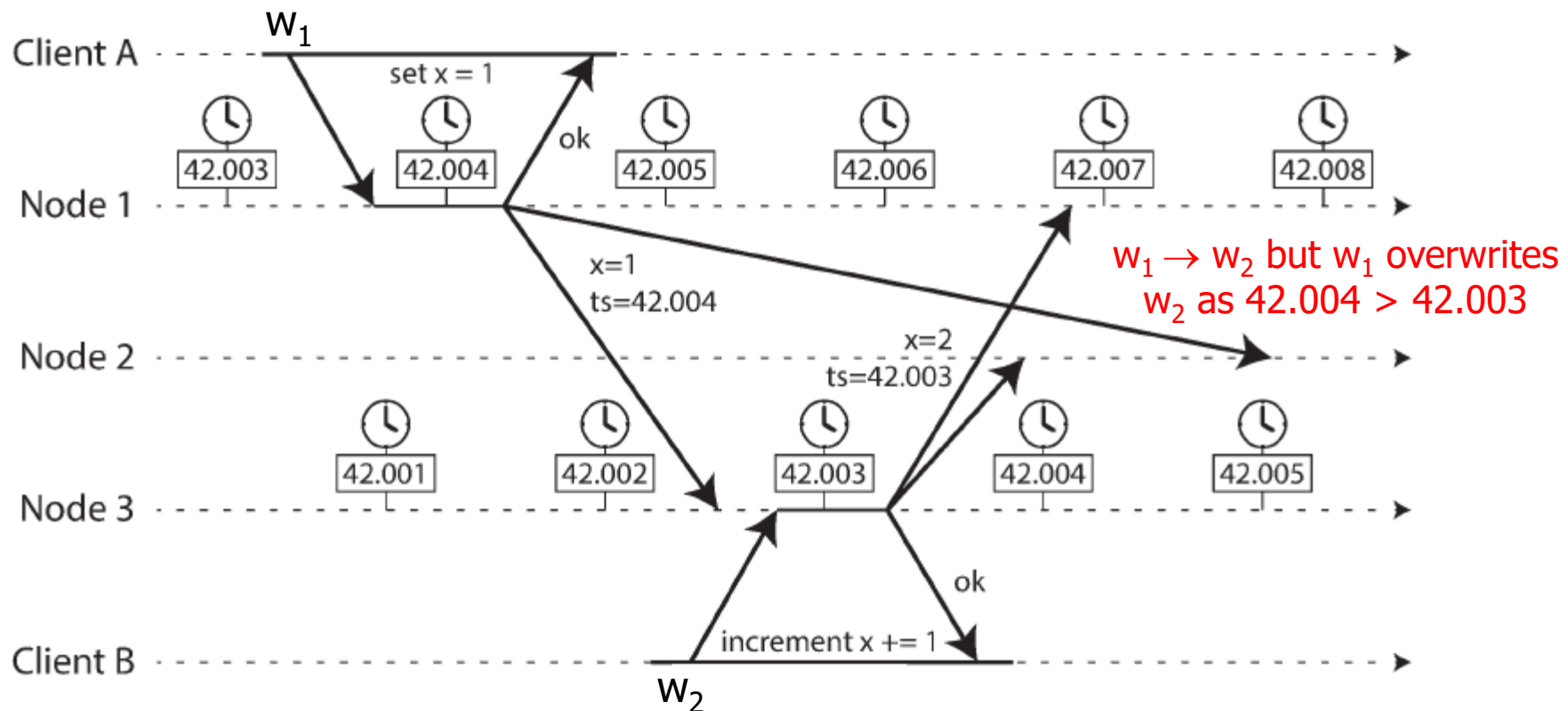
- Order writes according to clock timestamp
 - Overwrite value only if timestamp of incoming write is higher than the current one
- ↓ Some concurrent writes are silently dropped



if $t_2 > t_1$ then A and B will keep v_2 and discard v_1

Conflict resolution: Last writer wins

↓ Later writes can be overwritten by happened-before writes (not concurrent) due to clock skew



Conflict resolution: Version vectors

- Each replica keeps a vector for each item with its version number per replica
 - Increment its own version number on every write
- Each incoming write includes the vector from a prior read (a.k.a. causal context)
- Replica compares both vectors to distinguish causally-related and concurrent writes
 - Overwrite values from happened-before writes
 - Keep values from concurrent writes as siblings
 - Siblings can be merged together by prompting the user or automatically (CRDTs, Operational Transformation)

Conflict resolution: Version vectors

- Dotted version vectors implement this idea
 - Each replica k keeps a version vector $VV_k = \{(r_1, i_1, j_1), \dots, (r_n, i_n, j_n)\}$ for each item
 - (replica r_k id, range of events $\{1, i_k\}$ [, last event j_k])
 - e.g. $\{(a, 2), (b, 1), (c, 2, 4)\} \equiv \{a1, a2, b1, c1, c2, c4\}$
 - Each incoming write includes the vector VV_{inc}

if $VV_{inc} \geq VV_k$ **then**

increment VV_{inc}

get last event of VV_{inc} as dot

write incoming as sole value

otherwise

$VV_{new} = \text{merge } VV_{inc} \ \& \ VV_k$ (pairwise maximum)

increment VV_{new}

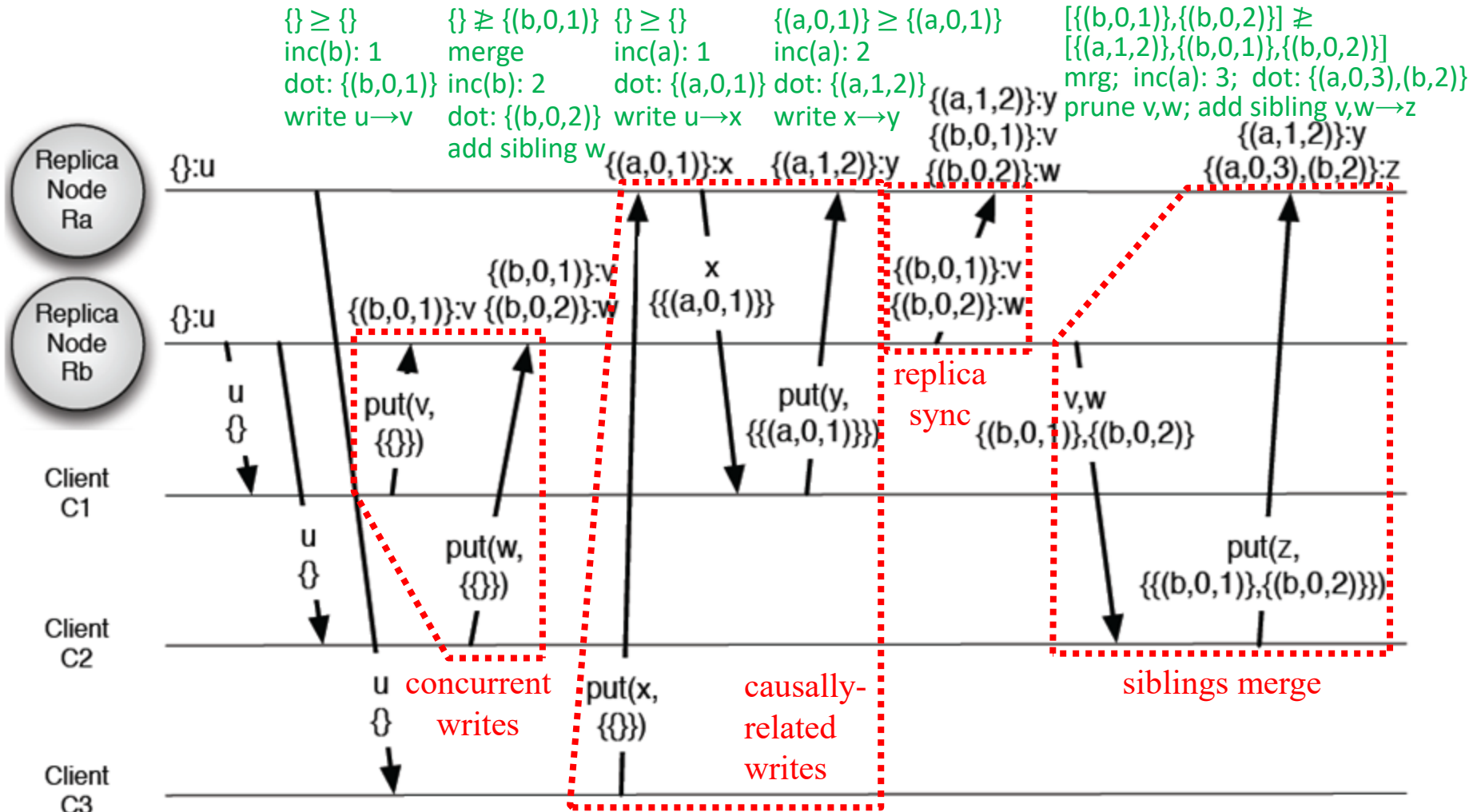
get last event of VV_{new} as dot

(use causal context from VV_{inc})

add new value as a **sibling**

prune all siblings s which $VV_{inc} \geq VV_k(s)$

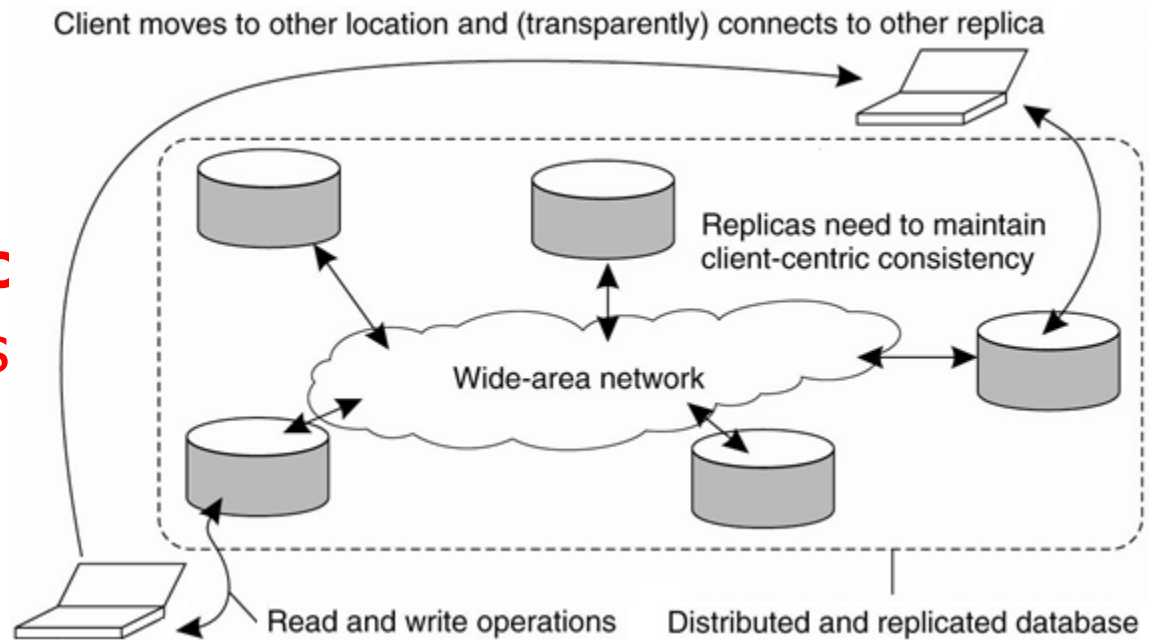
Conflict resolution: Version vectors



Eventual consistency

- Eventual consistency works well as long as every client always updates the same replica
- What if clients are mobile (operate on different replicas)?

⇒ Use **client-centric** consistency models

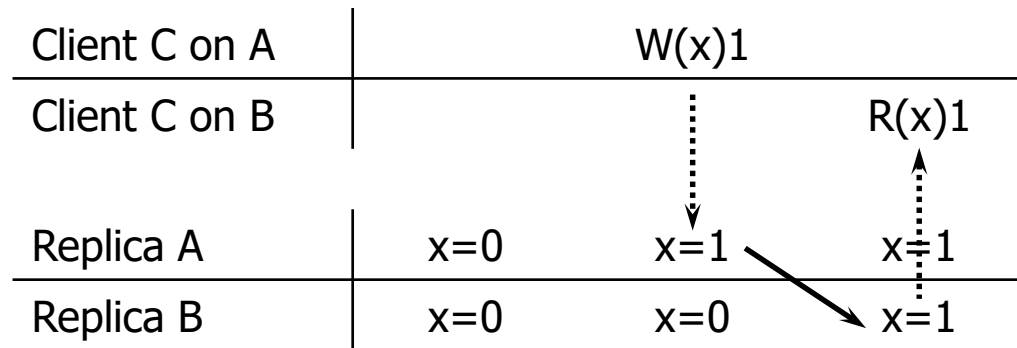


Client-centric consistency models

- Guarantee that **a single client** sees its accesses to the data store in a consistent way
 - No guarantees are given concerning concurrent accesses by multiple clients
- i) Read Your Writes, ii) Monotonic Reads, iii) Monotonic Writes, iv) Writes Follow Reads
 - For their implementation, we keep track of two sets of writes for each client
 - **Read set**: writes relevant (whose effects were visible) for the read operations performed by that client
 - **Write set**: writes performed by that client

Read Your Writes

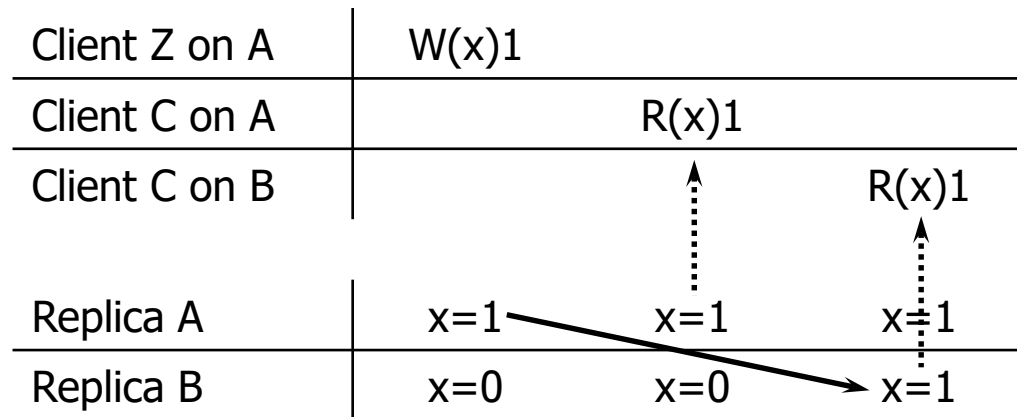
- If client C writes a data item 'x' on replica A, any successive read on 'x' by C on replica B will return the written value or a more recent one
 - e.g. updating your web page



- Replica A updates the client's write set WS; Replica B checks WS before reading to ensure that all writes in the set have taken place locally

Monotonic Reads

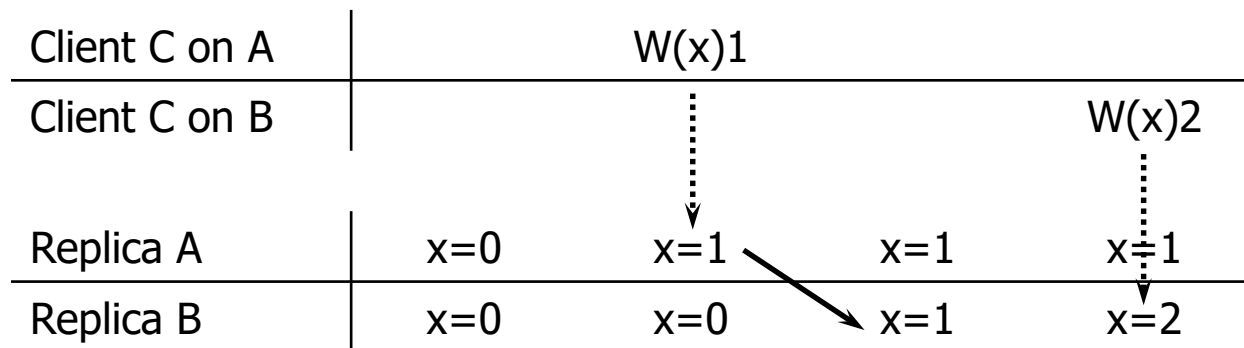
- If client C reads a data item 'x' on replica A, any successive read on 'x' by C on replica B will return the same value or a more recent one
 - e.g. reading email while you are on the move



- Replica A updates the client's read set RS; Replica B checks RS before reading to ensure that all writes in the set have taken place locally

Monotonic Writes

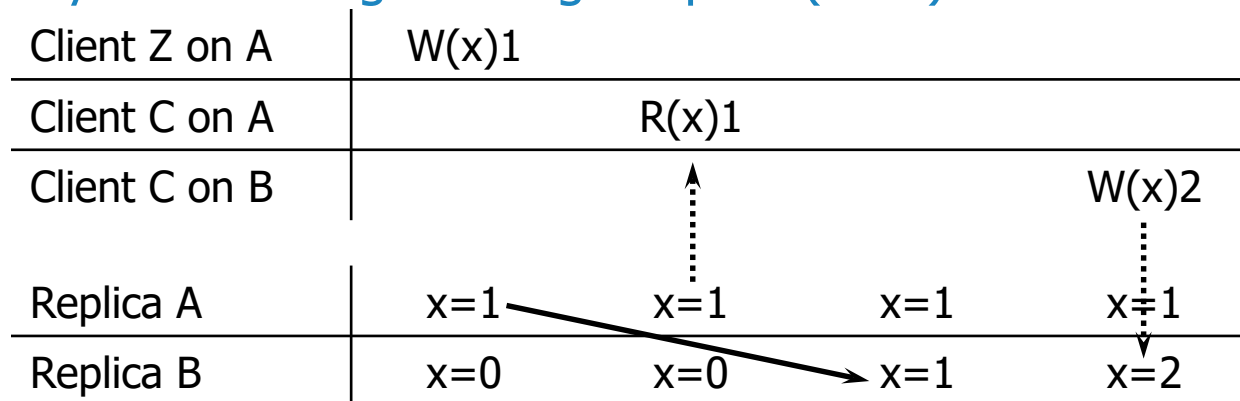
- A write by client C on a data item 'x' on replica A is completed also in replica B before C performs any successive write on 'x' on replica B
 - e.g. versioning a software library



- Replica A updates the client's write set WS; Replica B checks WS before writing to ensure that all writes in the set have taken place locally

Writes Follow Reads

- If C reads a data item 'x' on replica A, relevant writes for that read are completed also in replica B before C performs any successive write on 'x' on replica B
 - e.g. distributed newsgroup. A reaction (write) can be posted only after seeing the original post (read)



- Replica A updates the client's read set RS; Replica B checks RS before writing to ensure that all writes in the set have taken place locally

Consistency models on real systems

- Real systems supporting strong models:
 - NewSQL data stores: Google Spanner, Microsoft Yesquel
 - NoSQL data stores: Hyperdex, Apache HBase, MongoDB (*), COPS & Eiger (causal) (*) extended to support also eventual consistency
 - Distributed shared memory: IVY
- Real systems supporting eventual models:
 - NoSQL data stores: Apache CouchDB, Amazon Dynamo (*), Apache Cassandra (*), Riak (*), LinkedIn Voldemort (*) extended to support also strong consistency
 - File synchronizers: Dropbox

READING REPORT

[Terry13] Terry, D., *Replicated Data Consistency Explained Through Baseball*, Communications of the ACM, Vol. 56, No. 12, pp. 82-89, December 2013

Contents

- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- **Distribution protocols**
- Consistency protocols

Distribution protocols

- Implementation issues regardless of which consistency model is supported
 1. Replica placement
 - Decide where, when, and by whom copies of the data store are to be placed
 2. Update propagation
 - Decide which mechanisms to use for keeping the replicas consistent
 - What to propagate
 - How to propagate updates

Replica placement

1. Permanent replicas

- Initial set of replicas of the data store
- Small in number, organized as clusters or mirror sites

2. Server-initiated replicas

- Used to enhance performance
- Created at the initiative of the owner of the data store
- e.g. web server locates replicas close to where they are needed most (a.k.a. 'push caches')

3. Client-initiated replicas (a.k.a. client caches)

- Used to improve access times to data
- Created as a result of client requests
- e.g. browser caches

Update propagation

- What is actually propagated when a client initiates an update to a data store?
 - A. Propagate a notification to other replicas
 - Recipients **invalidate** their copy
 - Good when read to write ratio is low
 - B. Transfer the data from one replica to another
 - Recipients **replace** their copy
 - Good when read to write ratio is high
 - C. Propagate the update operation to other replicas
 - This is **active replication**
 - Each replica must be capable of 'actively' keeping its data up-to-date by performing operations

Update propagation

A. Push-based/Server-based approach

- Update sent 'automatically' by server
- Useful when a high degree of consistency is needed and the read to write ratio is high
- Often used between permanent and server-initiated replicas

B. Pull-based/Client-based approach

- Updates are requested by the client
 - No request \Rightarrow no update!
- Useful when the read to write ratio is low
- Used by client caches (e.g. browsers)

Push vs. pull protocols

- Comparison (* when using invalidations)

Issue	Push-based	Pull-based
State on server	List of client replicas	None
Messages sent	Update (and fetch-update later*)	Poll and update
Response time at client	Immediate (or fetch-update time*)	Fetch-update time

- Hybrid schemes are possible (e.g. **Leases**)
 - Server promises to push updates to client for a period of time. Once lease expires, client reverts to a pull approach (until another lease is issued)
- Adaptive leases
 - Lease expiration time depends on system behavior

Adaptive leases

A. Age-based leases

- Lease depends on the last time the item was modified
- An item that has not changed for a long time, will not change in a near future, so provide a long-lasting lease

B. Renewal-frequency-based leases

- Lease depends on how often a specific client requests a cached item to be refreshed
- The more often, the longer the lease

C. State-based leases

- Lease depends on state-space overhead at the server
- The more loaded a server is, the shorter the expiration times become (server must keep track of fewer clients)

Contents

- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- Distribution protocols
- **Consistency protocols**

Consistency protocols

- Consistency protocols describe:
 - A. Implementations of specific consistency models
 - We focus on sequential consistency and linearizability
 - B. Architectures of replicated processes for fault tolerance
 - Hierarchical and flat process groups
- Consistency protocols discussed:
 1. Primary-based protocols
 - Writes go to a single replica
 2. Replicated-write protocols
 - Writes can go to any replica

Contents

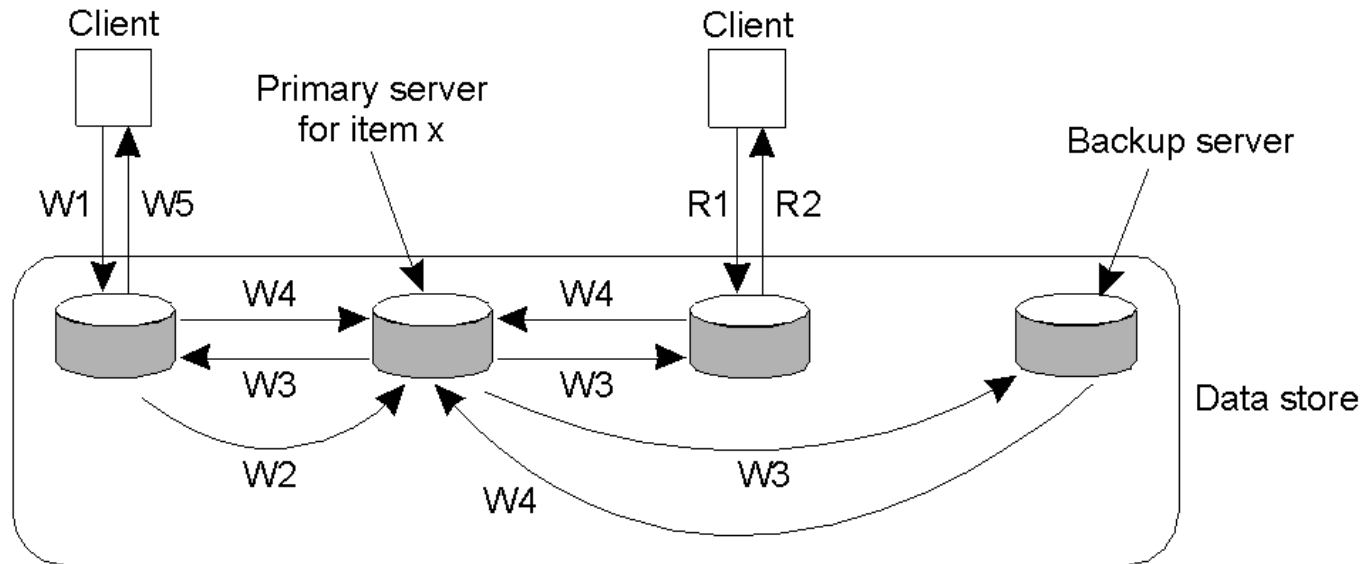
- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- Distribution protocols
- **Consistency protocols**
 - **Primary-based protocols**
 - Replicated-write protocols

Primary-based protocols

- Each data item is associated with a **primary replica** which is in charge of coordinating write operations to the data item
 - If the primary fails, one of the replicas is elected to act as the primary
- Also called passive replication protocols
- Two types:
 - A. Primary-backup remote-write protocols
 - B. Primary-backup local-write protocols

Primary-backup remote-write protocols

- All writes are done at a fixed single replica
- Reads can be carried out locally



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-backup remote-write protocols

↓ Bad write performance

- Writes can take a long time because a blocking write protocol is used

• Alternative: Use a non-blocking write protocol

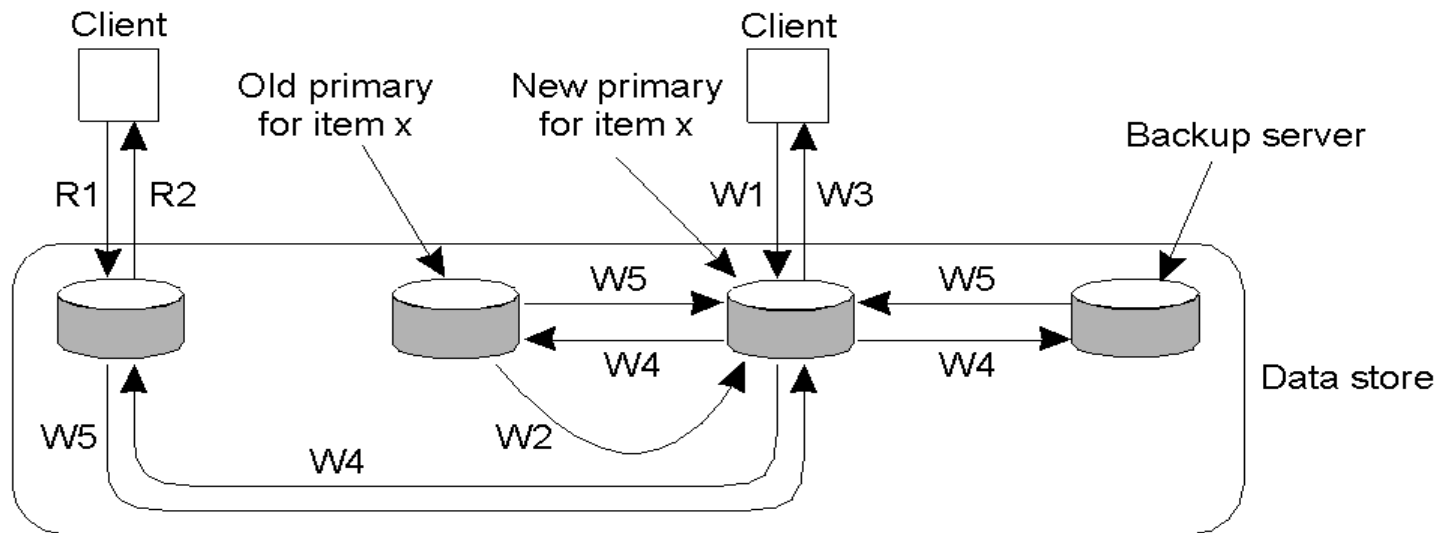
- Primary acknowledges the client's replica just after updating its local copy
- If some replica failed during the update, read-your-writes consistency is not guaranteed

Primary-backup remote-write protocols

- ↑ Easy to implement sequential consistency
 - Primary sends the write operations to each replica via FIFO-ordered view-synchronous atomic multicast (see 'Multicast' lesson for details)
 - Having a single primary and FIFO-order ensure that all replicas will see the writes in the same order
 - Virtual synchrony allows the system to take over exactly where it left off upon primary failure
- Implements linearizability if read requests are also forwarded to the primary

Primary-backup local-write protocols

- Primary migrates to the replica that is writing, successive writes are carried out locally, and then the replicas are updated using a non-blocking protocol

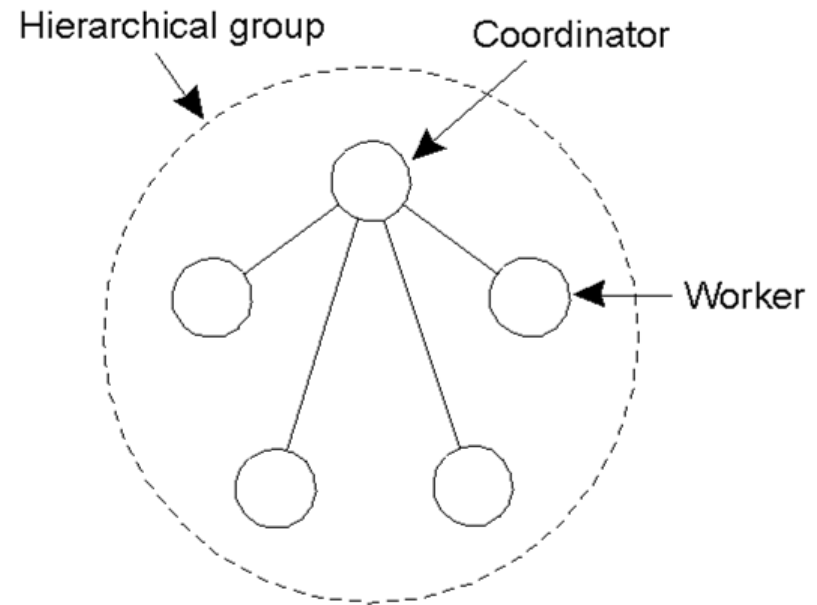


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Hierarchical process groups

- Use process replication to build a hierarchical process group that tolerates process failures
 - Processes are organized in a hierarchical fashion
- Implemented through a primary-based protocol
 - The coordinator acts as the primary
 - The workers act as backups



Hierarchical process groups

1. Clients send requests to the coordinator
 - Clients might submit read requests to the workers
 2. Coordinator provides the requested service and updates the workers using FIFO-ordered view-synchronous atomic multicast
- If the coordinator fails, one of the workers is promoted to act as new coordinator
 - K-fault tolerance on coordinator failure
 - **K+1** processes can tolerate K crash/omission failures
 - Cannot tolerate Byzantine process failures

SEMINAR PREPARATION – Groupy

[Guerraoui97] Guerraoui, R., Schiper, A.,
Software-based Replication for Fault Tolerance,
Computer, Vol. 30, No. 4, pp. 68-74, April 1997

Contents

- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- Distribution protocols
- **Consistency protocols**
 - Primary-based protocols
 - **Replicated-write protocols**

Replicated-write protocols

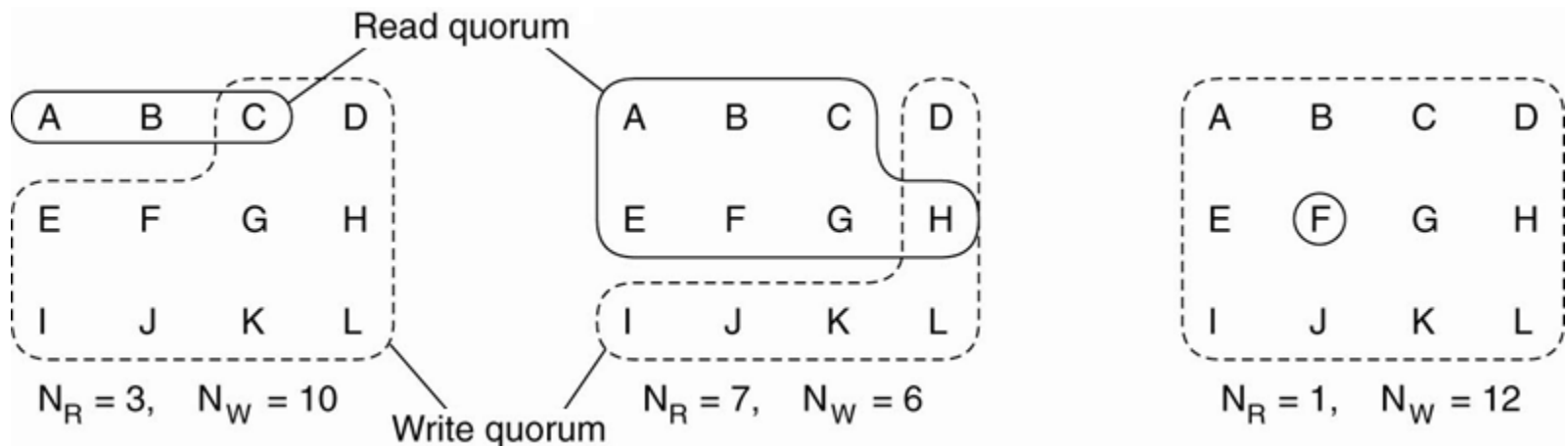
- Writes can be carried out at **multiple replicas** instead of only one, as occurs in primary-based protocols
- Also called distributed-write protocols
- Two types:
 1. Active replication
 2. Quorum-based protocols (majority voting)

Active replication

- Clients broadcast each operation to all the replicas and they are carried out in the same order everywhere
 - Use **total+FIFO-ordered multicast** to send the operations to the replicas
 - See 'Multicast' lesson for details
 - Sender waits until the multicast message is delivered back and then it returns to the client
- Multicasting reads+writes → linearizability
- Multicasting only writes → sequential cons.
 - Reads return the current value of the local copy

Quorum-based protocols

- Clients must get replies from a quorum of the N replicas before reading (N_R) or writing (N_W)
 - $N_R + N_W > N$ to avoid read-write conflicts
 - $N_W > N/2$ to avoid write-write conflicts



correct choice of
read and write set

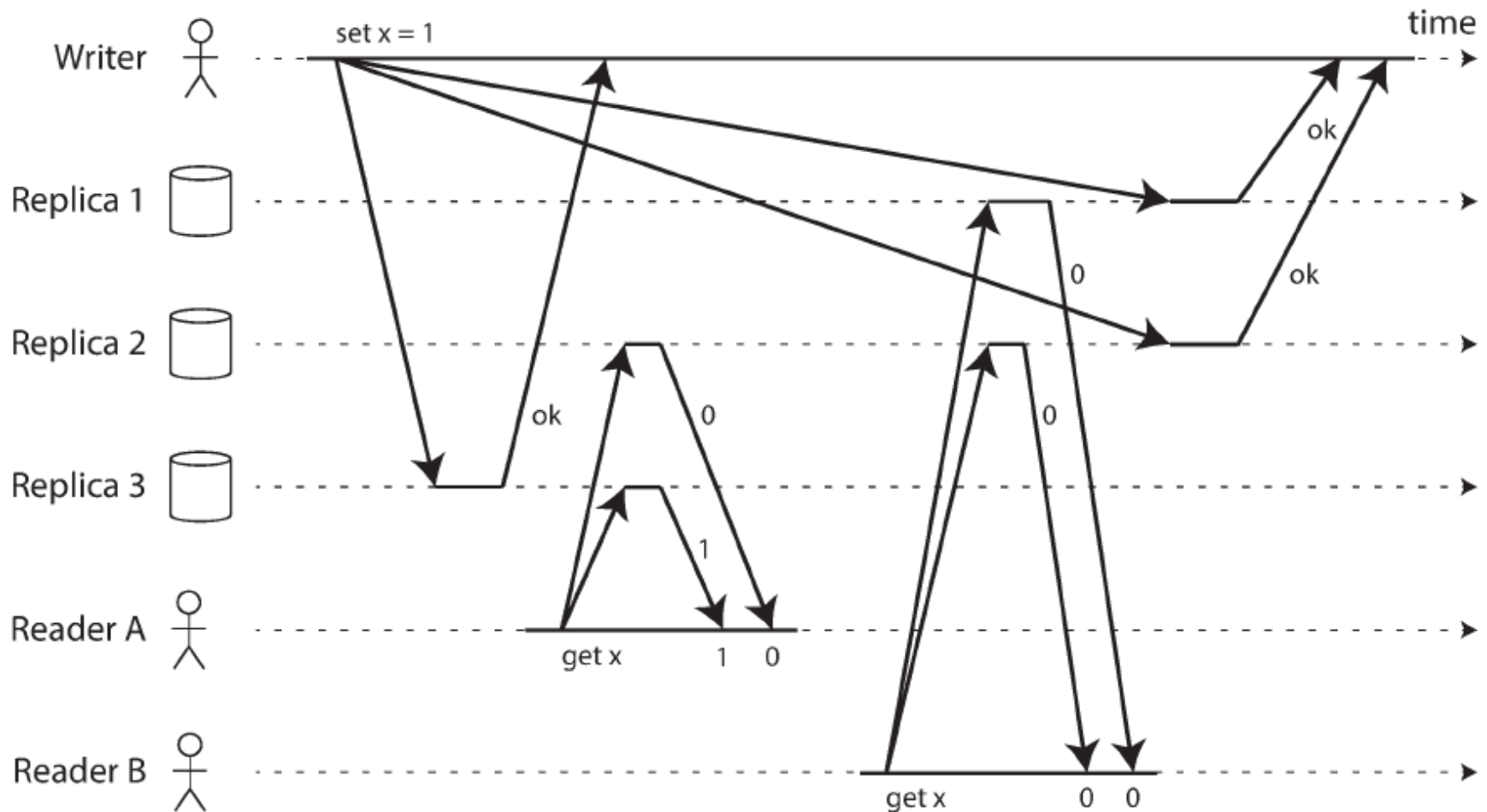
choice that may lead
to WR-WR conflicts

correct choice: ROWA
(read-one, write-all)

Quorum-based protocols

- These protocols can implement linearizability, but this requires additional steps in addition to reading/writing from a quorum
 1. Use version vectors to achieve ordered writes according to their real-time order
 - Write operations must read first to learn the highest version and the potential siblings
 - See 'Conflict resolution' lesson for details
 2. Perform 'read repair' to achieve up-to-date monotonic reads
 - Read operations must write to update stale replicas

Why we need 'read repair'



Quorum write

1. Read phase

- Send read requests to all (or N_R) replicas
- Await replies from N_R replicas
- Learn the highest version among the replicas
 - Merge siblings if needed

2. Write phase

- Send write requests (containing the version learnt) to all (or N_W) replicas (if N_W , remaining replicas are updated as a background task: anti-entropy)
- Await ACKs from N_W replicas
- Write is complete and can return to the user code

Quorum read

1. Read phase

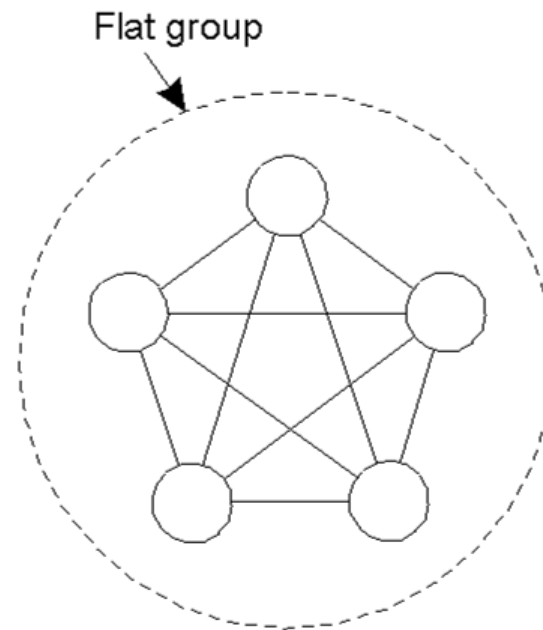
- Send read requests to all (or N_R) replicas
- Await replies from N_R replicas
- If all version numbers are the same, return the value to the user code

2. Write phase ('read repair')

- Otherwise, send write requests (containing the largest version number and the associated up-to-date value) to all (or only outdated) replicas
- Await ACKs from N_W (or only updated) replicas
- Return the value to the user code

Flat process groups

- Use process replication to build a flat process group that tolerates process failures
 - Processes are identical and the group response is defined through **voting**
- Implemented through a replicated-write protocol
 - Active replication
 - Quorum-based



Flat groups: active replication

- Clients send requests to all workers using total+FIFO-ordered atomic multicast
- Await one reply (crash) or $K+1$ identical replies from workers (Byzantine)
- K -fault tolerance on worker failure
 - **$K+1$** processes can tolerate K crash/omission failures
 - **$2K+1$** processes can tolerate K Byzantine failures
 - K failing processes could generate the same wrong reply, thus we need $K+1$ correct processes

Flat groups: quorum-based

- Clients send requests to all workers and await replies from N_R (or N_W) of them
- K-fault tolerance on worker failure
 - **$2K+1$** and $2K$ processes can tolerate K crash or omission failures for writes and reads, respectively
 - $N_W = \lfloor N/2 \rfloor + 1$; $N = K + N_W \Rightarrow N > 2K$
 - $N_R + N_W = N + 1$; $N = K + N_R \Rightarrow N \geq 2K$
 - **$3K+1$** and $3K$ processes can tolerate K Byzantine failures for writes and reads, respectively
 - $N_W = \lfloor (N + K)/2 \rfloor + 1$; $N = K + N_W \Rightarrow N > 3K$
 - $N_R + N_W = K + N + 1$; $N = K + N_R \Rightarrow N \geq 3K$

Summary

- Reasons for replication
 - Improved availability
 - Enhanced reliability
 - Improved performance & scalability
- But replication can lead to inconsistencies ...
- How can we propagate updates so that these inconsistencies are not noticed without severely degrading performance?
- Proposed solutions apply for the relaxation of any existing consistency constraints

Summary

- Consistency models
 - Data-centric models
 - Strict, Linearizability, Sequential, Causal, FIFO concern themselves with individual reads/writes to data items
 - Weaker models, which introduce the notion of synchronization variables, concern themselves with a group of reads/writes
 - Client-centric models
 - Concerned with maintaining consistency for a single client access to the distributed data store
 - Models based on Eventual consistency
 - Read your writes, Monotonic reads, Monotonic writes, Writes follow reads

Summary

- We looked at 'distribution protocols' designed to facilitate the propagation of updates
- We looked at 'consistency protocols' as a way to implement consistency models
 - The most common schemes are those that support sequential consistency or linearizability, but eventual consistency is gaining popularity
- Further details:
 - [Tanenbaum]: chapter 7
 - [Coulouris]: chapter 18