# 5. Consistency and Replication

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2024/2025 Q2

## Contents

## 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

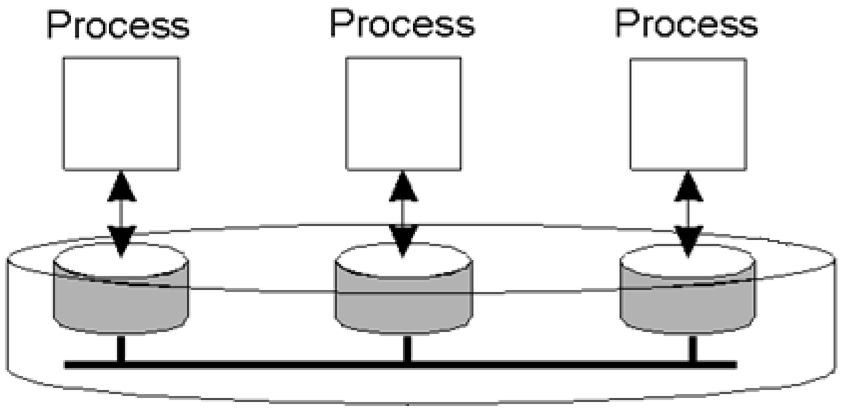
  ⇒ How do we transparently (and efficiently) keep all the replicas up-to-date and consistent?
- Dilemma: replication improves scalability, but incurs the overhead of synchronizing replicas
  - The solution often results in a relaxation of the consistency constraints

## 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'

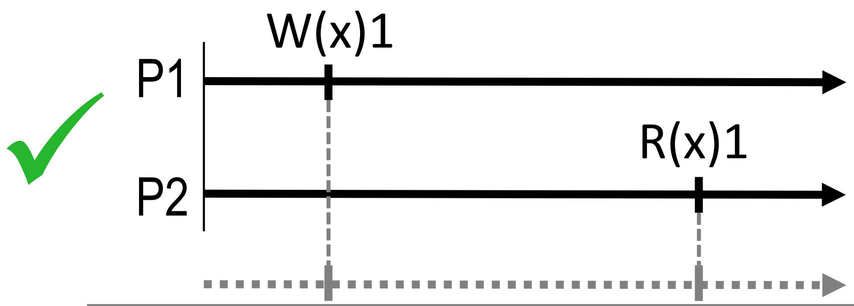
# Strict consistency

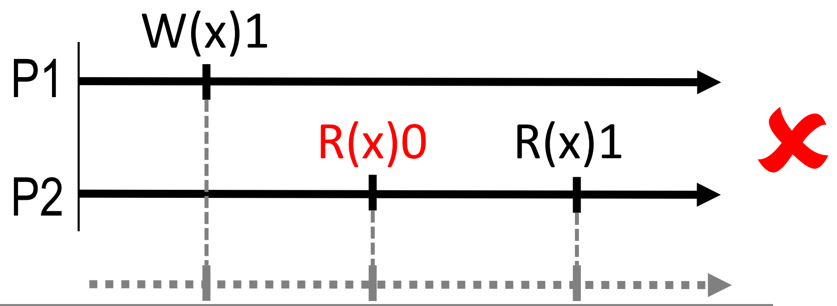
- Meet sequential specification of each item 'x'
  - Given 2 concurrent writes on 'x', one must go first
  - A read on 'x' returns the value of the most recent write on 'x', regardless of the used replica
  - Subsequent reads on 'x' return the same value or some later value, regardless of the used replica

EXCLUSIVE ORDERED WRITES

UP-TO-DATE MONOTONIC READS

- 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

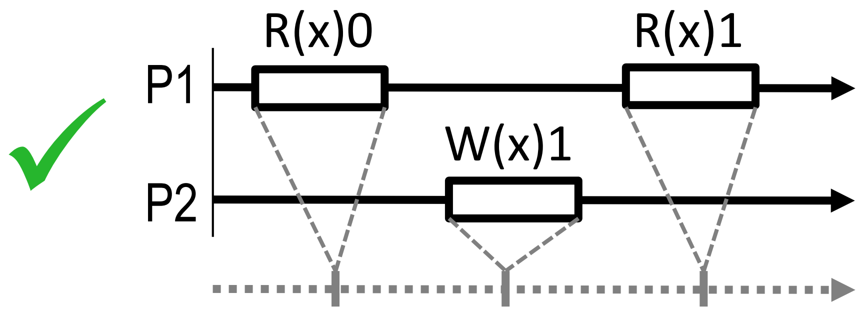
  $\Rightarrow$ Network latencies preclude this

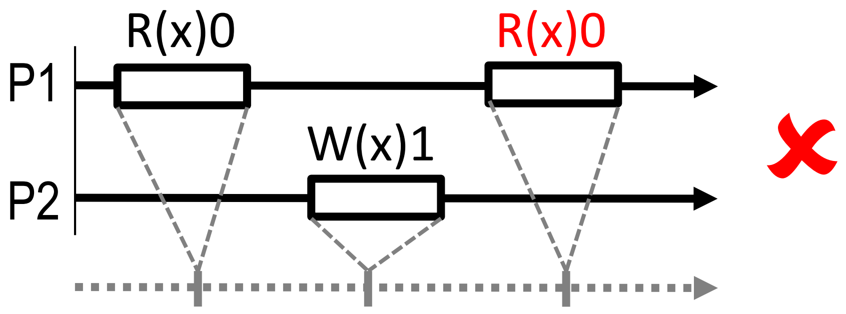
  2. Requires an absolute global time such that at most one operation occurs at a time

  $\Rightarrow$ Clocks can neither be perfectly synchronized nor have infinite precision

## Linearizability (a.k.a. strong consistency)

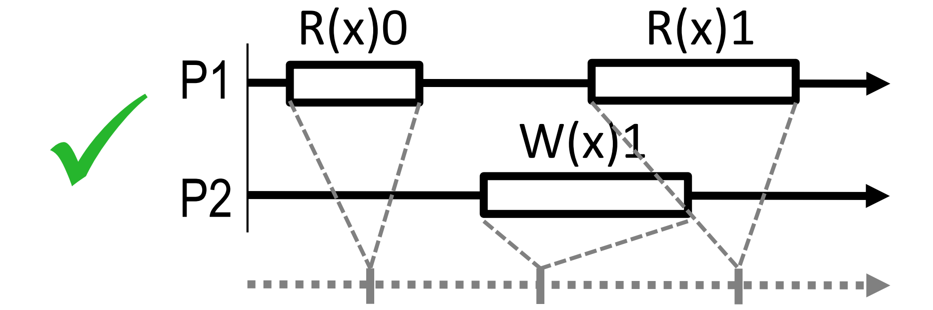
- Same goal as strict consistency (exclusive ordered writes + up-to-date monotonic reads) but recognizing that operations take time to complete and <u>can overlap</u>
- Operations appear to run in <u>some common sequence</u> that retains their <u>real-time</u> order <u>when they do not overlap</u>

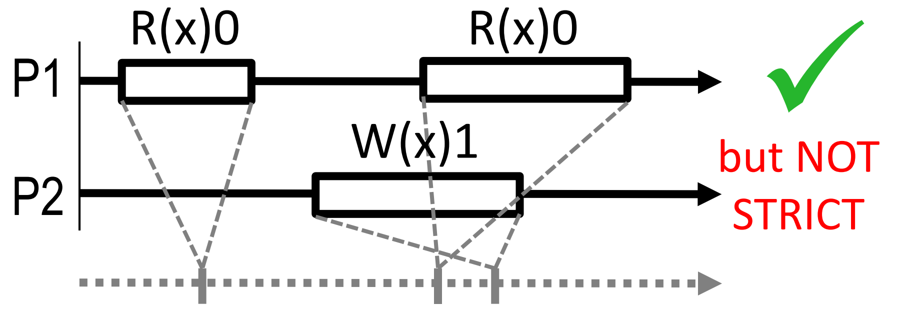


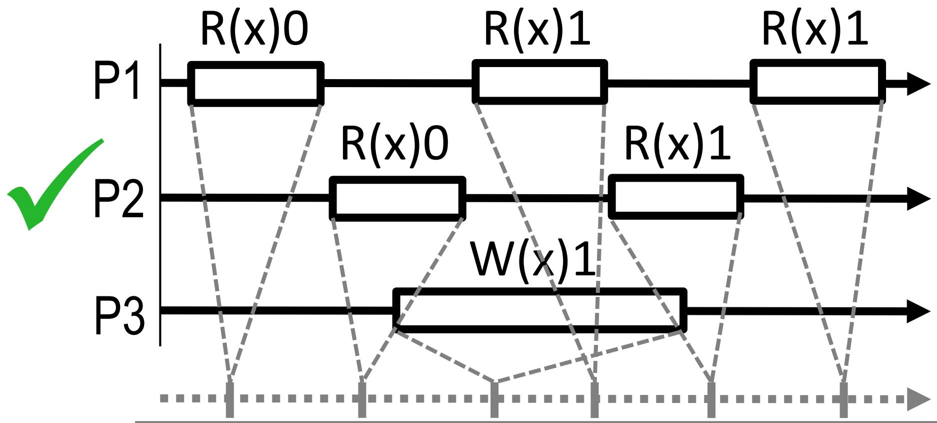


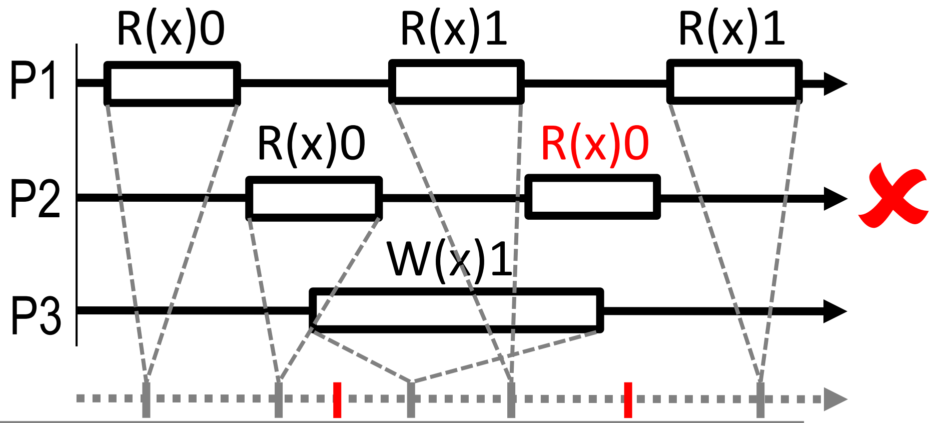
## Linearizability (a.k.a. strong consistency)

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



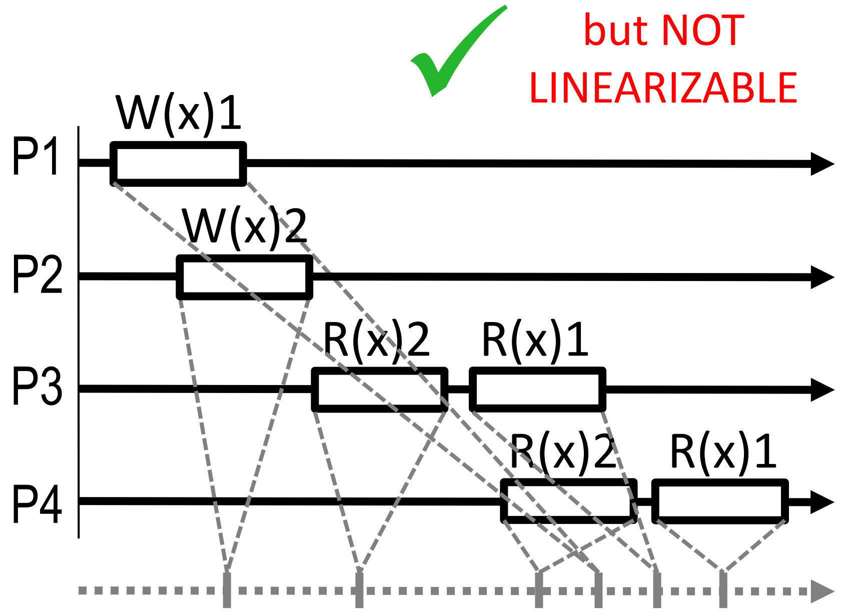


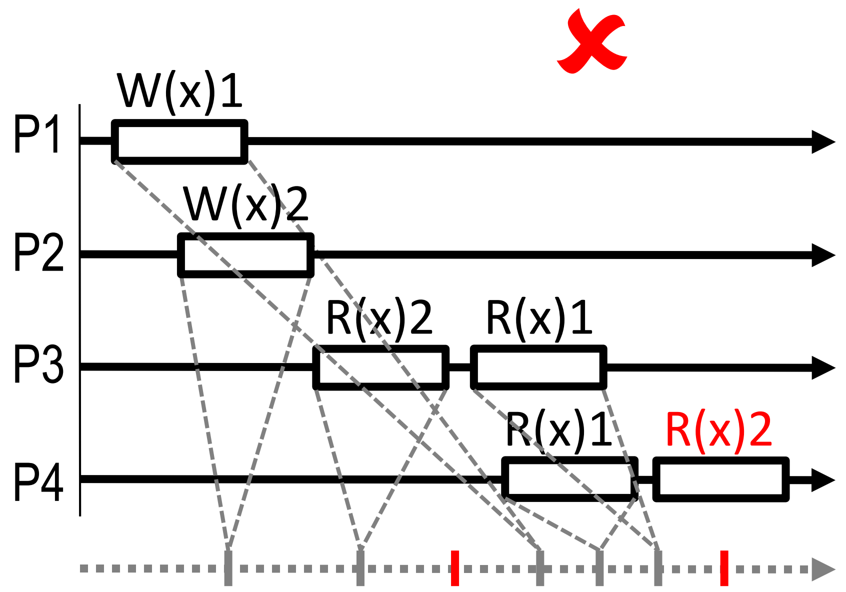




# Sequential consistency

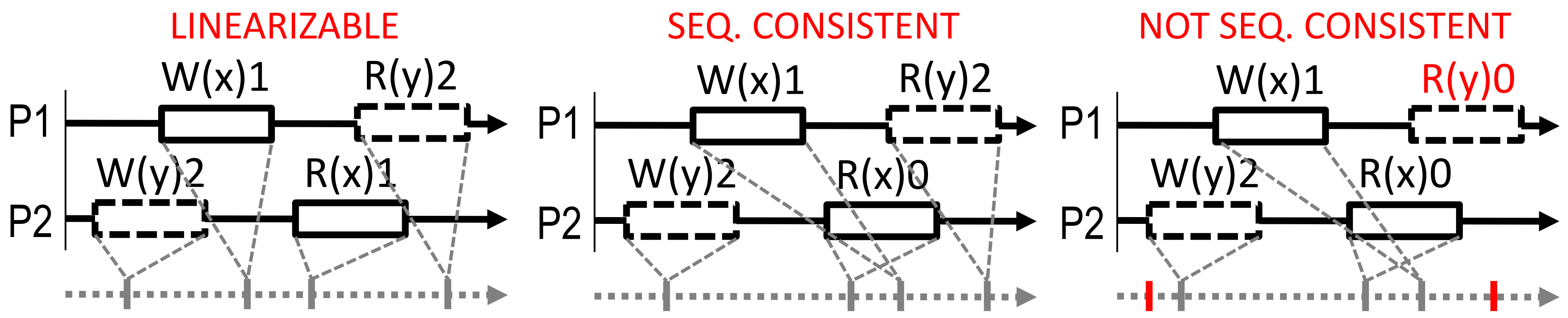
- Operations appear to run in <u>some common sequence</u> that retains the <u>program order</u> for the operations of each process
  - Real time is not taken into consideration





## 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
    - Strict ⊂ Linearizability ⊂ Sequential
  - See ‘Consistency protocols’ lesson for details

## Contents

## 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

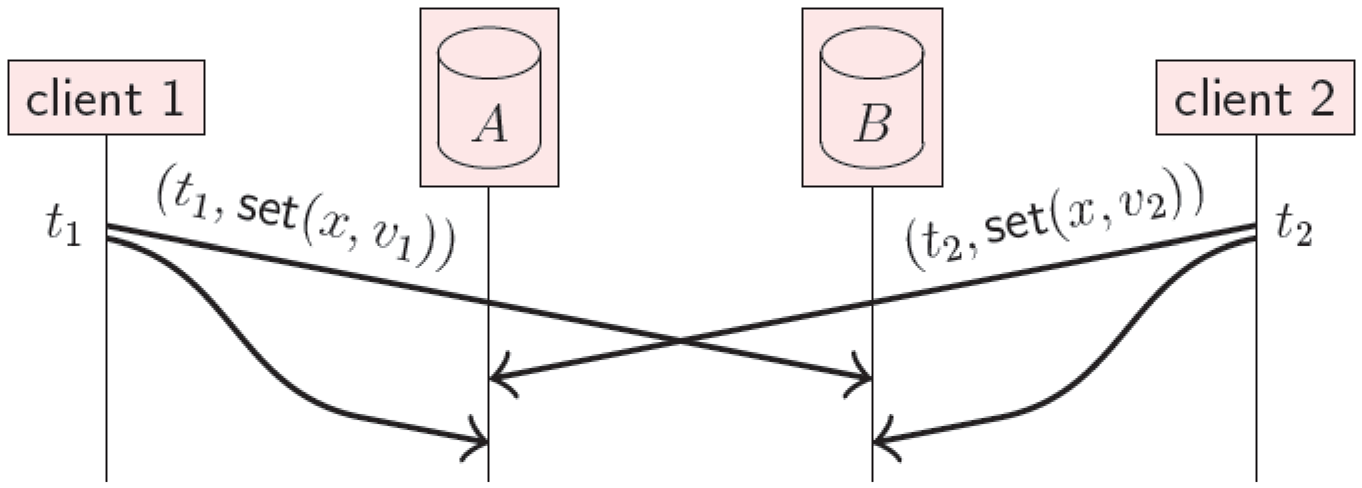
## Eventual consistency

- 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

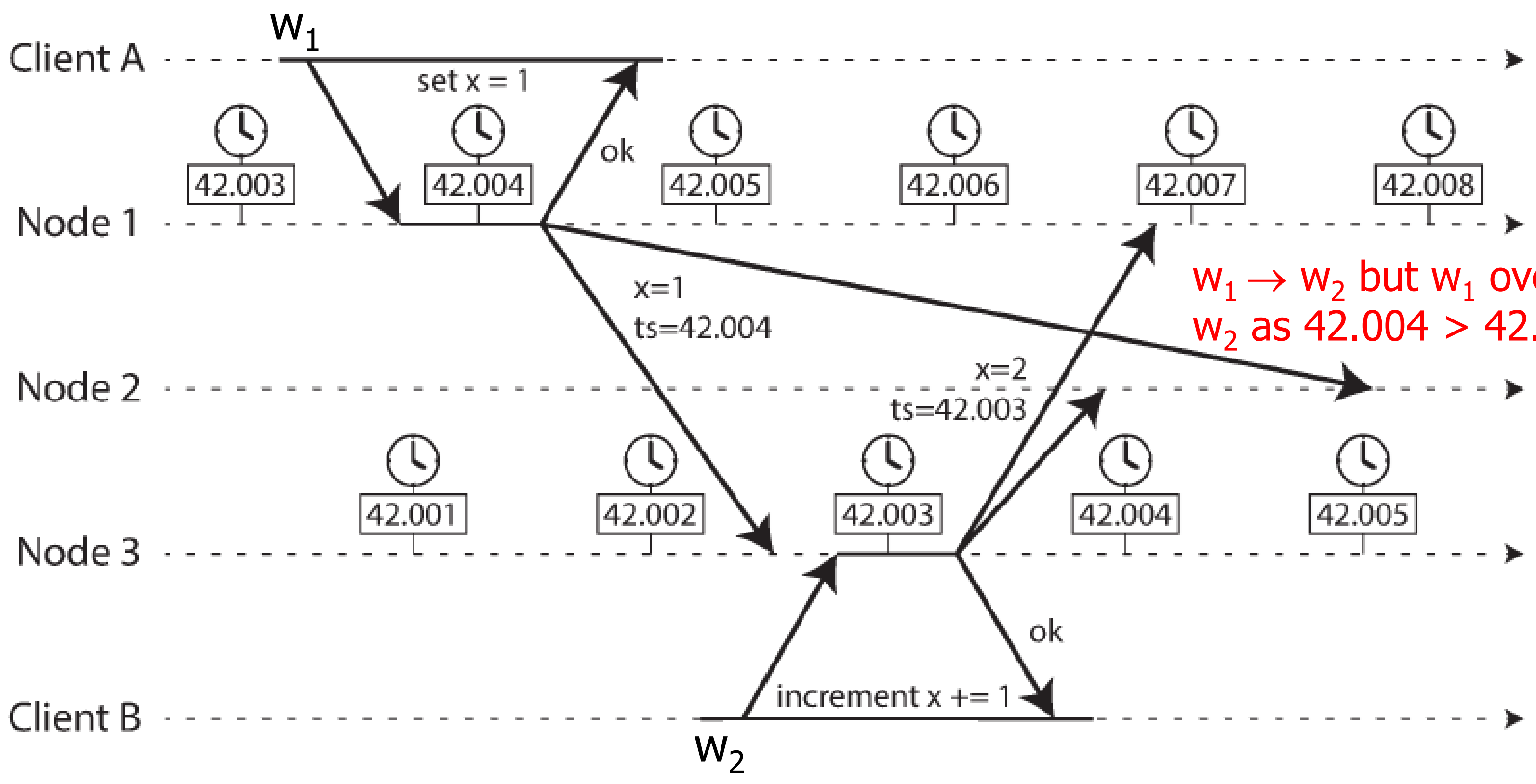
↓ Some concurrent writes are silently dropped



if t2 > t1 then A and B will keep v2 and discard v1

# Conflict resolution: Last writer wins

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



$w_1 \rightarrow w_2$ but $w_1$ overwrites $w_2$ as 42.004 > 42.003

## Conflict resolution: Version vectors

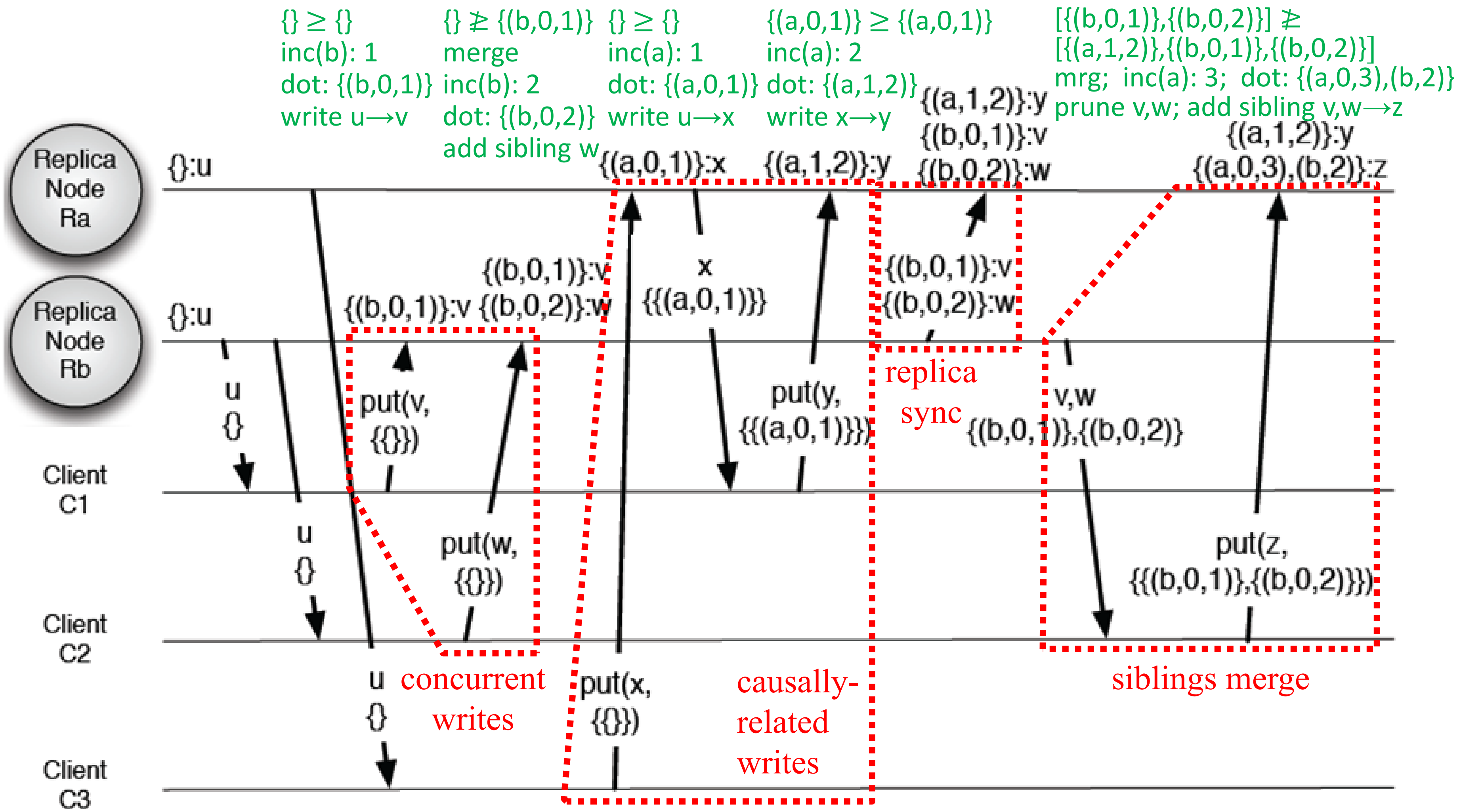
- Each replica keeps a vector for each item with its version number per replica
  - Increment its own version number on every write
- Each incoming write includes the vector from a prior read (a.k.a. causal context)
- Replica compares both vectors to distinguish causally-related and concurrent writes
  - Overwrite values from happened-before writes
  - Keep values from concurrent writes as <u>siblings</u>
    - 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 $VV_k=\{(r_1, i_1, j_1), \ldots, (r_n, i_n, j_n)\}$ for each item
    - (replica $r_k$ id, range of events $\{1, i_k\}$ $\overbrace{[, \text{last event } j_k]}^{\textbf{dot}}$)
    - 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 ≥ VV_k then
    increment VV_inc
    get last event of VV_inc as dot
    write incoming as sole value
otherwise
    VV_new = 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 ≥ VV_k(s)
```

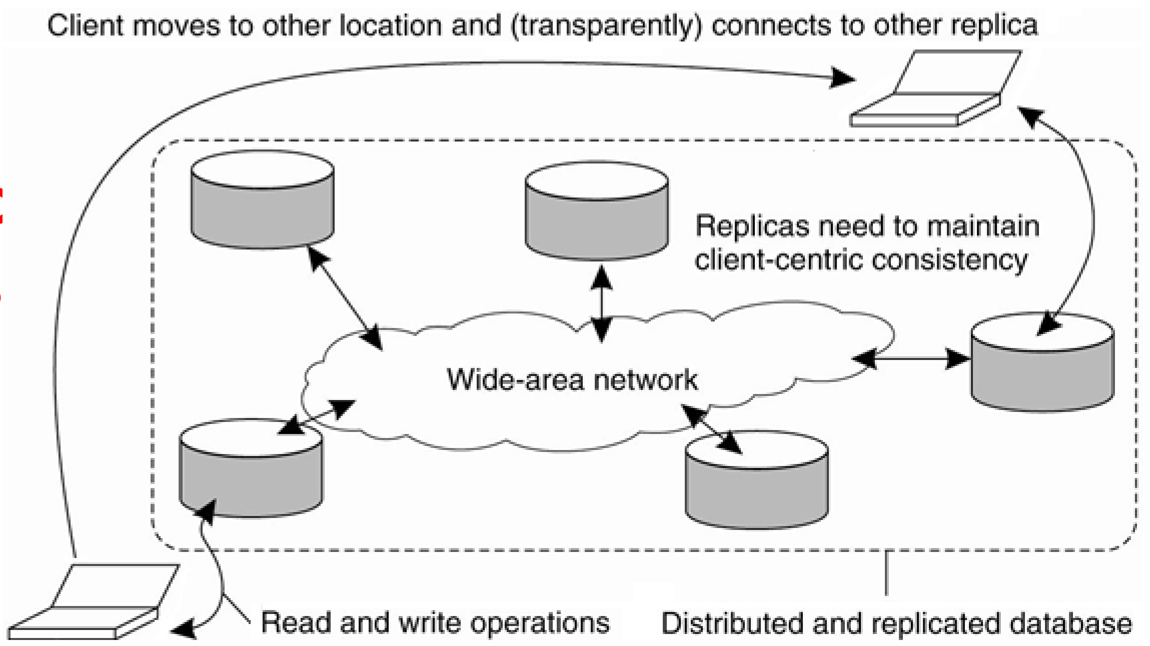
# Conflict resolution: Version vectors



## 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 (a.k.a. session guarantees)


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Read and write operations
Distributed and replicated database

# 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
- 1) Read Your Writes, 2) Monotonic Reads, 3) Monotonic Writes, 4) Writes Follow Reads
  - Implemented by keeping track of version vectors representing two sets of writes for each client
    - **Read set** ($RS_i$): writes relevant (whose effects were visible) for the read operations performed by client $\mathrm{i}$
    - **Write set** ($WS_i$): writes performed by client $\mathrm{i}$

# Client-centric consistency models

**CLIENT i**: $C_i$

```
val = read()
    send(rd, $RS_i$, $WS_i$) to $R_k$
    …
    recv(val, $VV_k$) from $R_k$
    $RS_i = \max(RS_i, VV_k)$
    return val

write(val)
    send(wr, val, $RS_i$, $WS_i$) to $R_k$
    …
    recv($VV_k$) from $R_k$
    $WS_i = \max(WS_i, VV_k)$
```

max: pairwise maximum

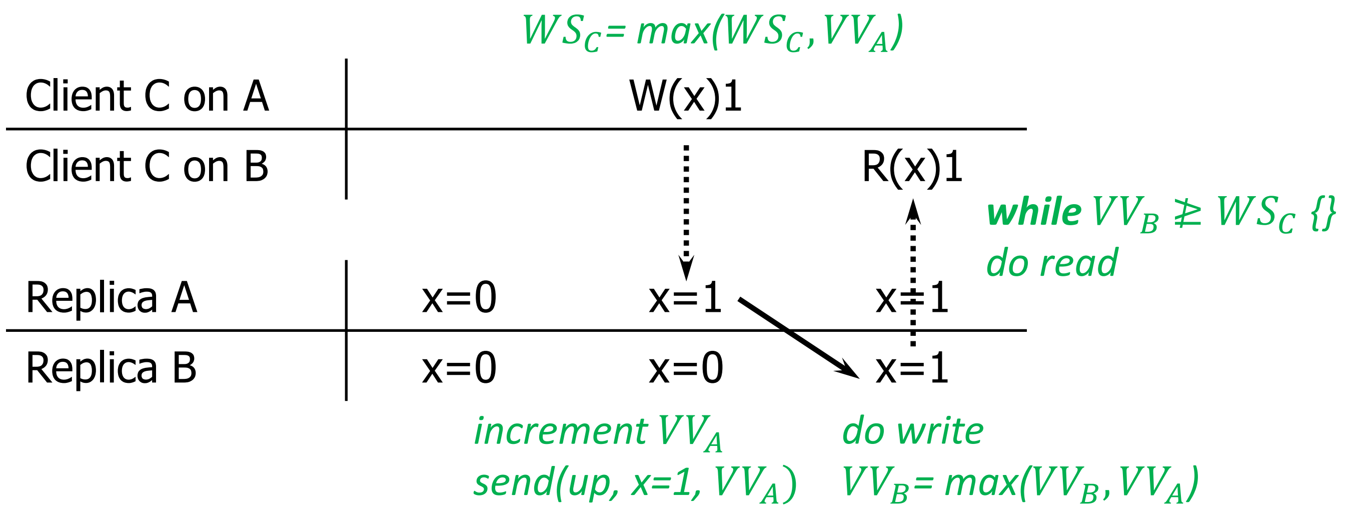
**REPLICA k**: $R_k$

```
read($RS_i$, $WS_i$)
    if MR then while $VV_k \ngeq RS_i$ {}
    if RYW then while $VV_k \ngeq WS_i$ {}
    val ← do read ; send(val, $VV_k$) to $C_i$
write(val, $RS_i$, $WS_i$)
    if MW then while $VV_k \ngeq WS_i$ {}
    if WFR then while $VV_k \ngeq RS_i$ {}
    do write or add sibling†
    increment $VV_k$ ; send($VV_k$) to $C_i$
    foreach j≠k do send(up, val, $VV_k$) to $R_j$
update(val, $VV_j$)
    do write or add sibling†
    $VV_k = \max(VV_j, VV_k)$
```

† see 19-21 for details

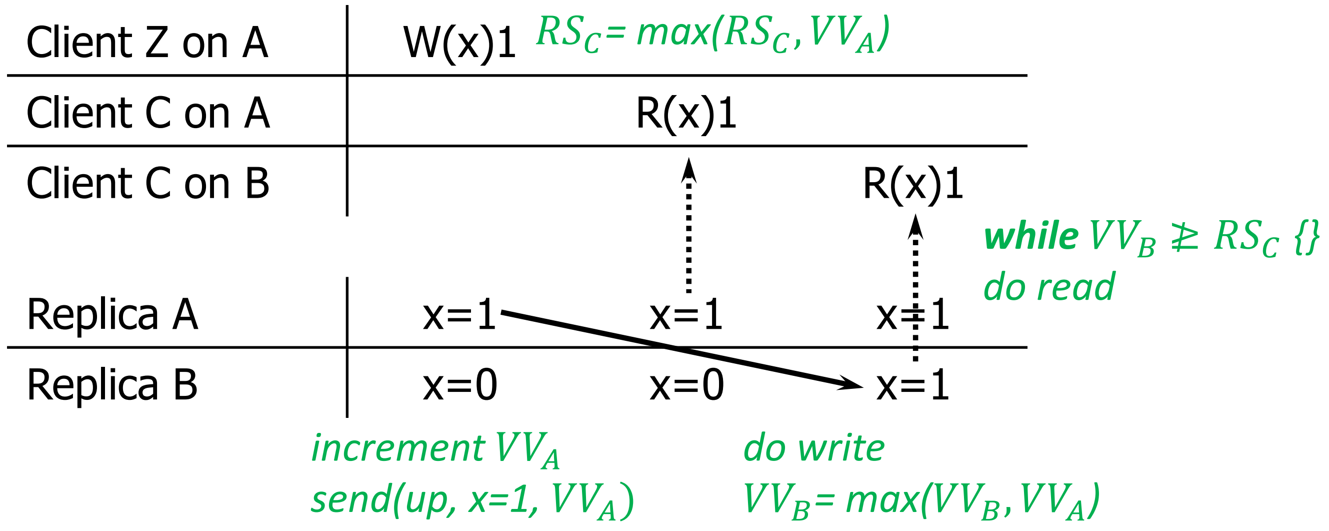
## Read Your Writes (RYW)

- 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



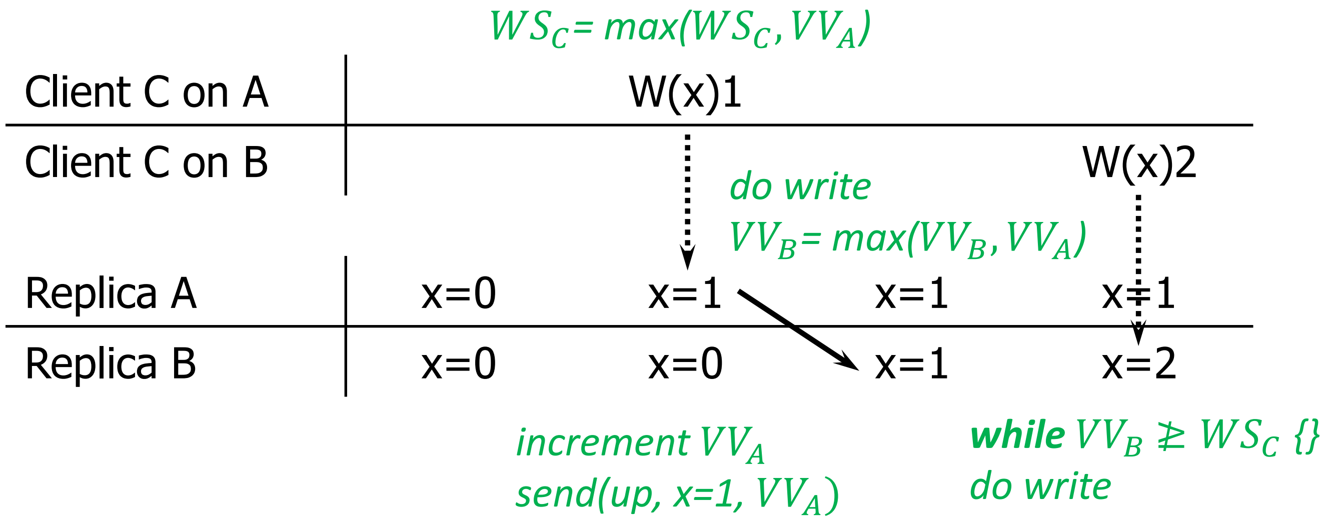
## Monotonic Reads (MR)

- 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



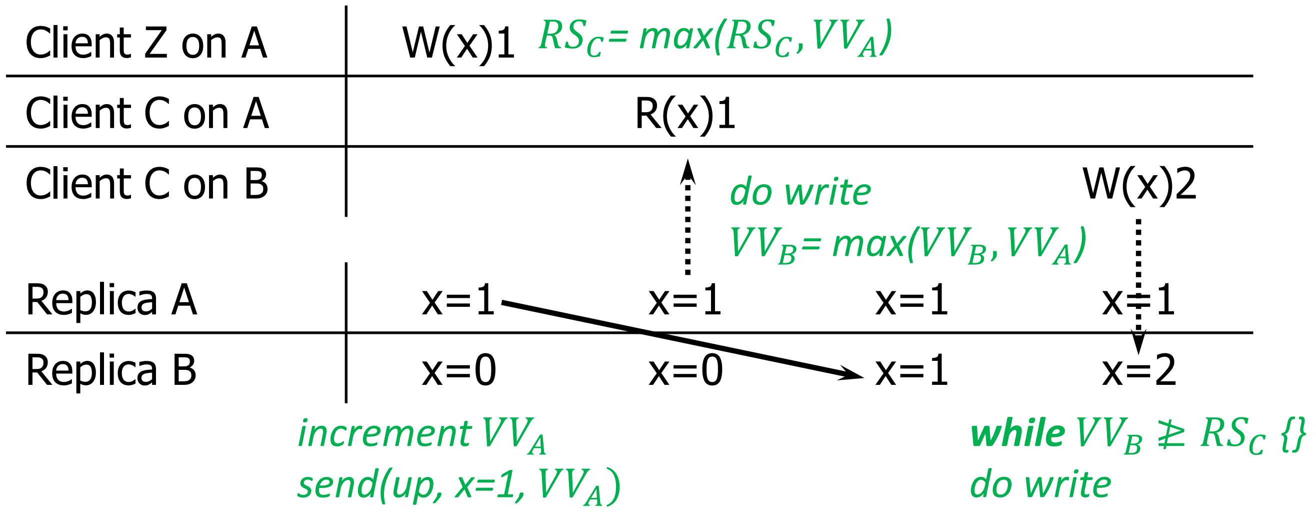
# Monotonic Writes (MW)

- 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



# Writes Follow Reads (WFR)

- 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)



## 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)
  - Distributed shared memory: IVY

(*) extended to support also eventual consistency

- Real systems supporting eventual models:
  - NoSQL data stores: Apache CoughDB, Amazon Dynamo (*), Apache Cassandra (*), Riak (*), LinkedIn Voldemort
  - File synchronizers: Dropbox

(*) extended to support also strong consistency

## 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

## 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. push content on-demand closer to clients via CDN
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 by 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**

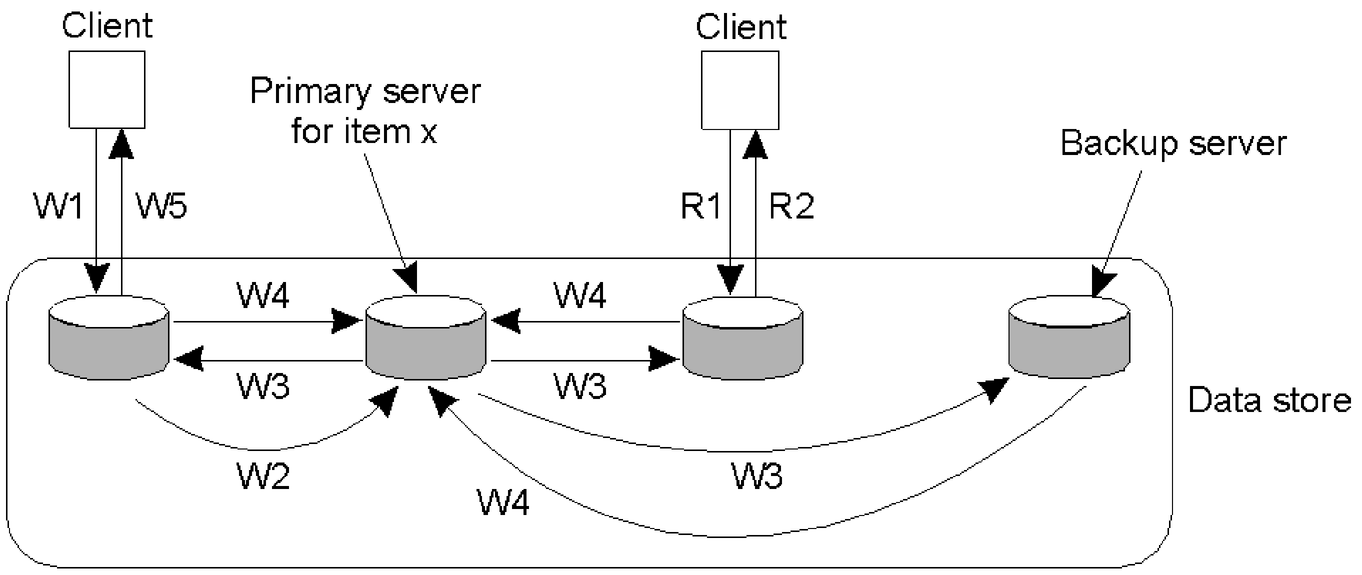
## Consistency protocols

- Consistency protocols describe:
  - A. Implementations of specific consistency models
    - We focus on sequential consistency and linearizability
  - B. Resilient architectures of replicated processes
    - Hierarchical and flat process groups
- Consistency protocols discussed:
  1. Primary-based protocols
     - Writes are managed by a single **primary replica**
       - If it fails, another replica is elected to act as primary
  2. Replicated-write protocols
     - Writes are managed by **multiple replicas**

## Contents

## Primary-backup remote-write protocols

- All writes are forwarded to the primary 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 <u>non-blocking write protocol</u>
  – 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

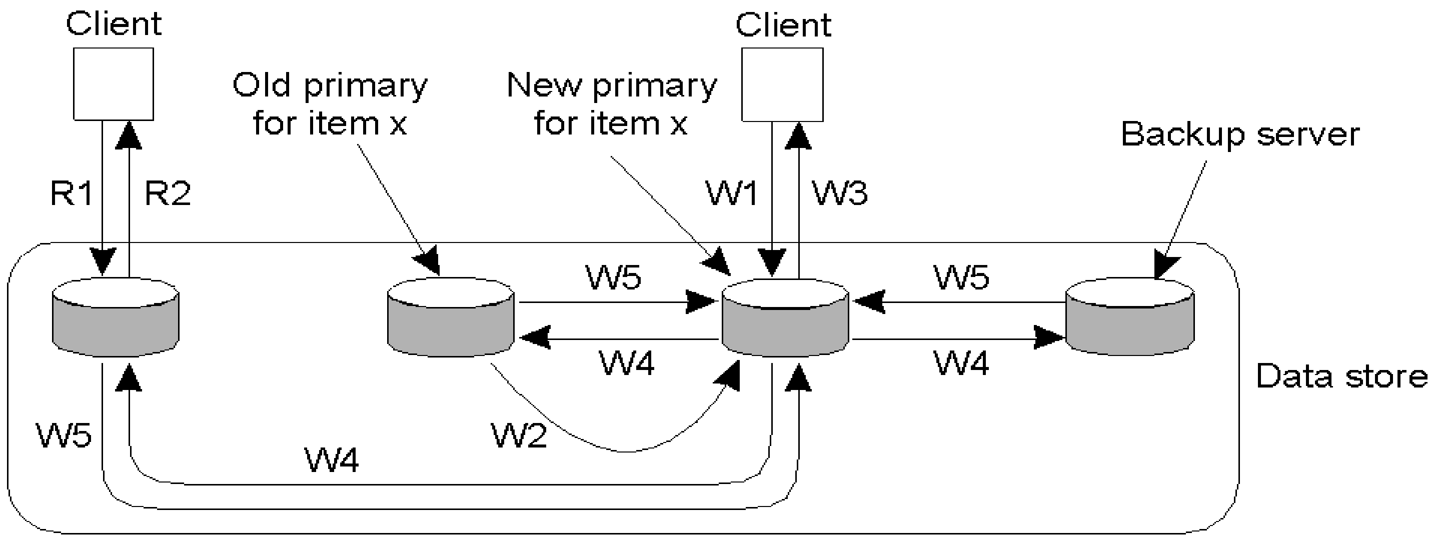
## Primary-backup remote-write protocols

↑ 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

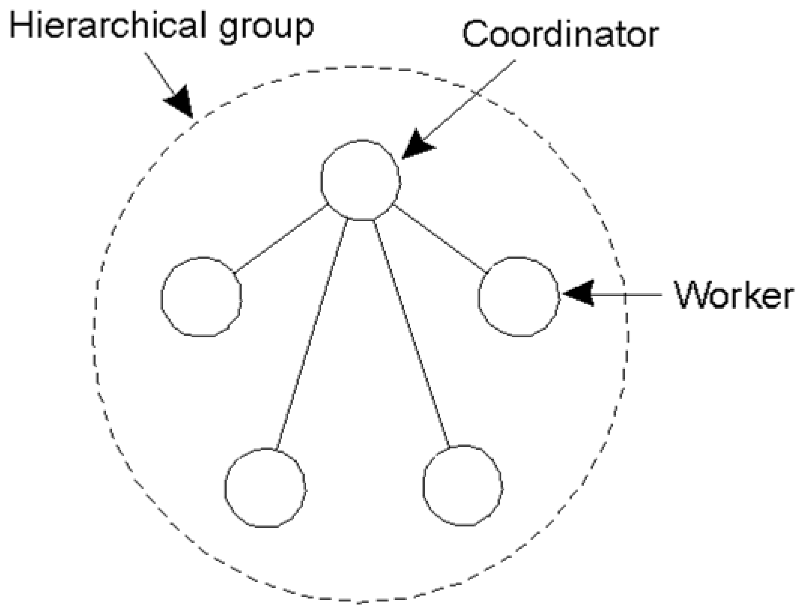


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

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

- 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

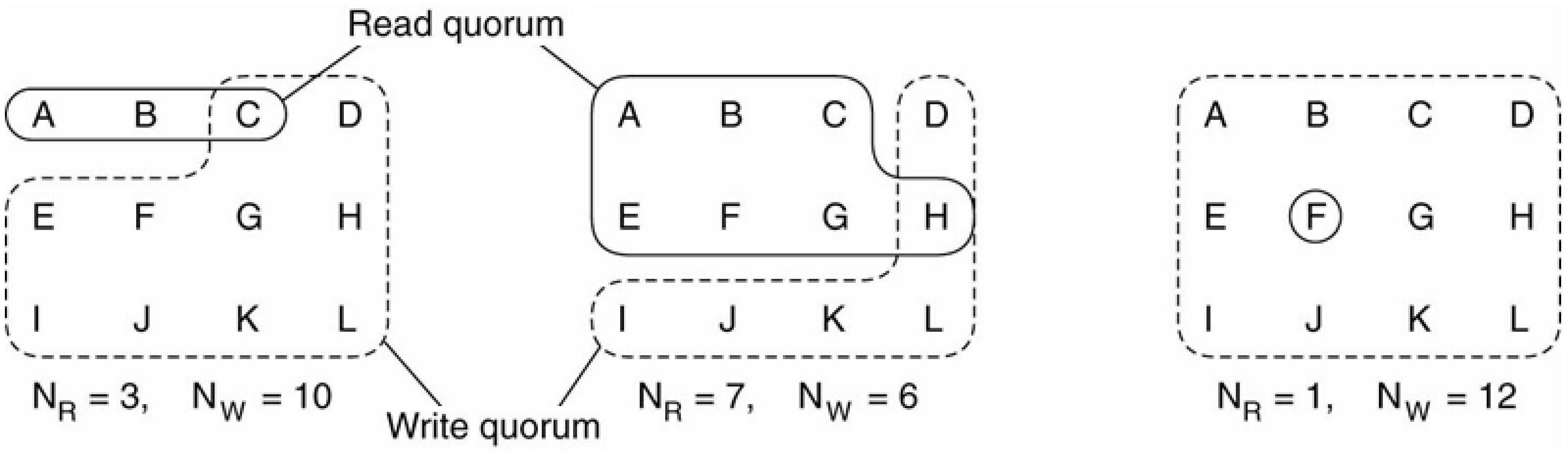
## Contents

## Active replication protocols

- 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 $\rightarrow$ linearizability
- Multicasting only writes $\rightarrow$ 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

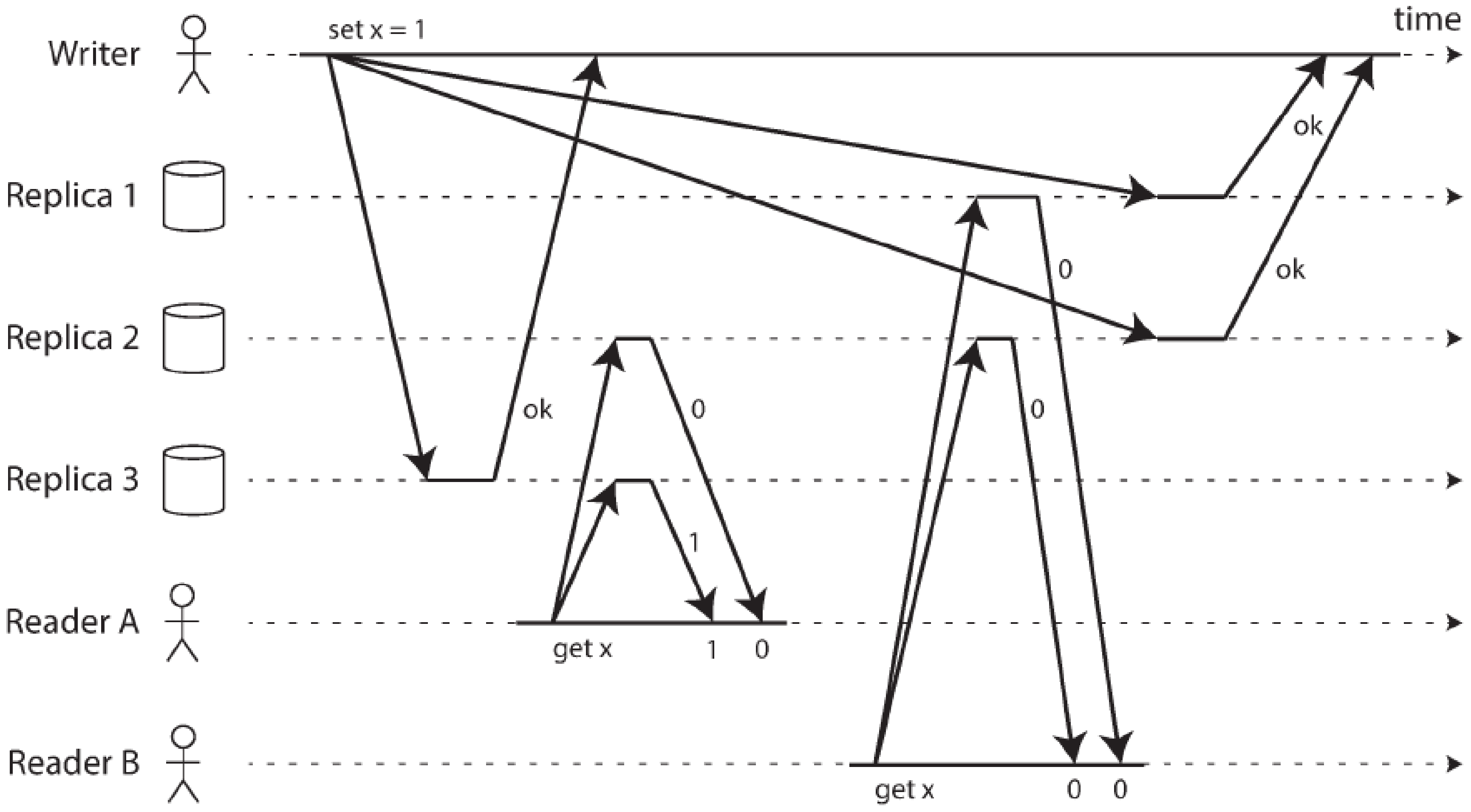
choice that may lead to WR-WR conflicts

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 majority quorums for reading and writing
     - ($N_R > N/2$) and ($N_W > N/2$)
  2. Use version vectors to achieve ordered writes according to their real-time order
     - Writes must read first to learn the highest version and the potential siblings $\Rightarrow$ see 'Conflict resolution' lesson
  3. 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

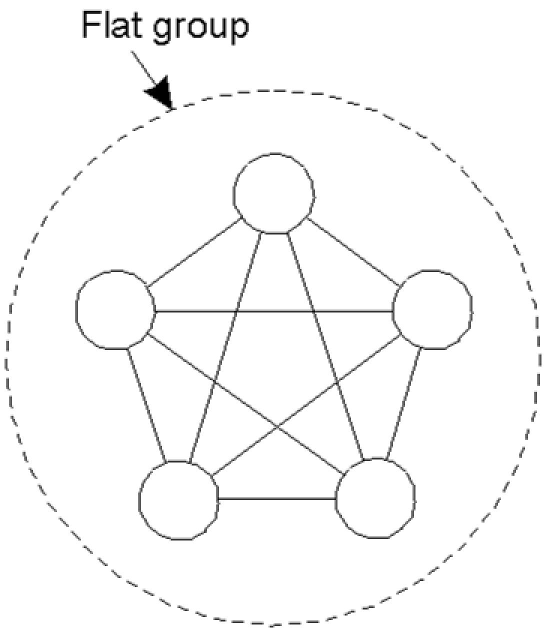
1. Read phase
   - Send read requests to all (or $N_R$) replicas
   - Await replies from $N_R$ replicas
   - Learn the highest version among the replicas
     - Merge siblings if needed
2. Write phase
   - Send write requests (containing the version learnt) to all (or $N_W$) replicas (if $N_W$, remaining replicas are updated as a background task: anti-entropy)
   - Await ACKs from $N_W$ replicas
   - Write is complete and can return to the user

## 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
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

# 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 and 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†
    - $N_W = \lfloor (N + K)/2 \rfloor + 1$ ; $N = K + N_W \Rightarrow N > 3K$
    - $N_R + N_W = K + N + 1$ ; $N = K + N_R \Rightarrow N \geq 3K$

† Assuming self-verifying data (with public/private keys) so that Byzantine values can be discarded. Otherwise, 4K+1 and 4K processes are needed

## 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 provide a system-wide consistent view on data stores with <u>concurrent</u> individual reads/writes to data items
  - Client-centric models
    - Concerned with maintaining consistency for the access of a single client 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