DATA DISTRIBUTION

[DDIA, Ch 5-6, DSCD, Ch 18]

Why more machines?

- Scalability
    - Deal with more data, higher read/write loads
    - Scaling up vs scaling out
        - scaling up = vertical scaling (shared-memory or shared-disk architectures)
        - scaling out = horizontal scaling (shared-nothing architectures)
- Performance
    - If you have users around the world, you might want to have have servers at various locations worldwide, so that users can be served from a datacenter that is geographically close to them.
- Fault tolerance/high availability
    - Use multiple machines to give you redundancy: when one or more fail, another one can take over

Scalability

- The ability of a system, network, or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

- We can measure growth in almost any terms. But there are three particularly interesting things to look at:

    - Size scalability: adding more nodes should make the system linearly faster; growing the dataset should not increase latency

    - Geographic scalability: it should be possible to use multiple data centers to reduce the time it takes to respond to user queries, while dealing with cross-data center latency in some sensible manner.

    - Administrative scalability: adding more nodes should not increase the administrative costs of the system (e.g. the administrators-to-machines ratio).

- A scalable system is one that continues to meet the needs of its users as scale increases. There are two particularly relevant aspects - performance and availability - which can be measured in various ways.

Performance (and latency)

- Characterization of the amount of useful work accomplished by a computer system compared to the time and resources used.

- Depending on the context, this may involve achieving one or more of the following:
    - Short response time/low latency for a given piece of work
    - High throughput (rate of processing work)
    - Low utilization of computing resource(s)

- Latency: the state of being latent; delay, a period between the initiation of something and the occurrence.

- Latent: From Latin *latens, latentis*, present participle of *lateo* ("lie hidden"). Existing or present but concealed or inactive.

Availability (and fault tolerance)

- the proportion of time a system is in a functioning condition. If a user cannot access the system, it is said to be unavailable.
- In formula, availability = uptime / (uptime + downtime)
- from a technical perspective, availability is mostly about being fault tolerant.
- Fault tolerance is the ability of a system to behave in a well-defined manner once faults occur

Availability	Nickname	Downtime per year
90%	one nine	more than a month
99%	two nines	less than 4 days
99.9%	three nines	less than 9 hours
99.99%	four nines	less than 1 hour
99999%	five nines	about 5 minutes
99.9999%	six nines	about 31 seconds

Examples

- One single server
- On a
  - Me
    - Mean Time To Restart (MTTR) 2 mins
- Two r
- 10 se
- MTB
- All se

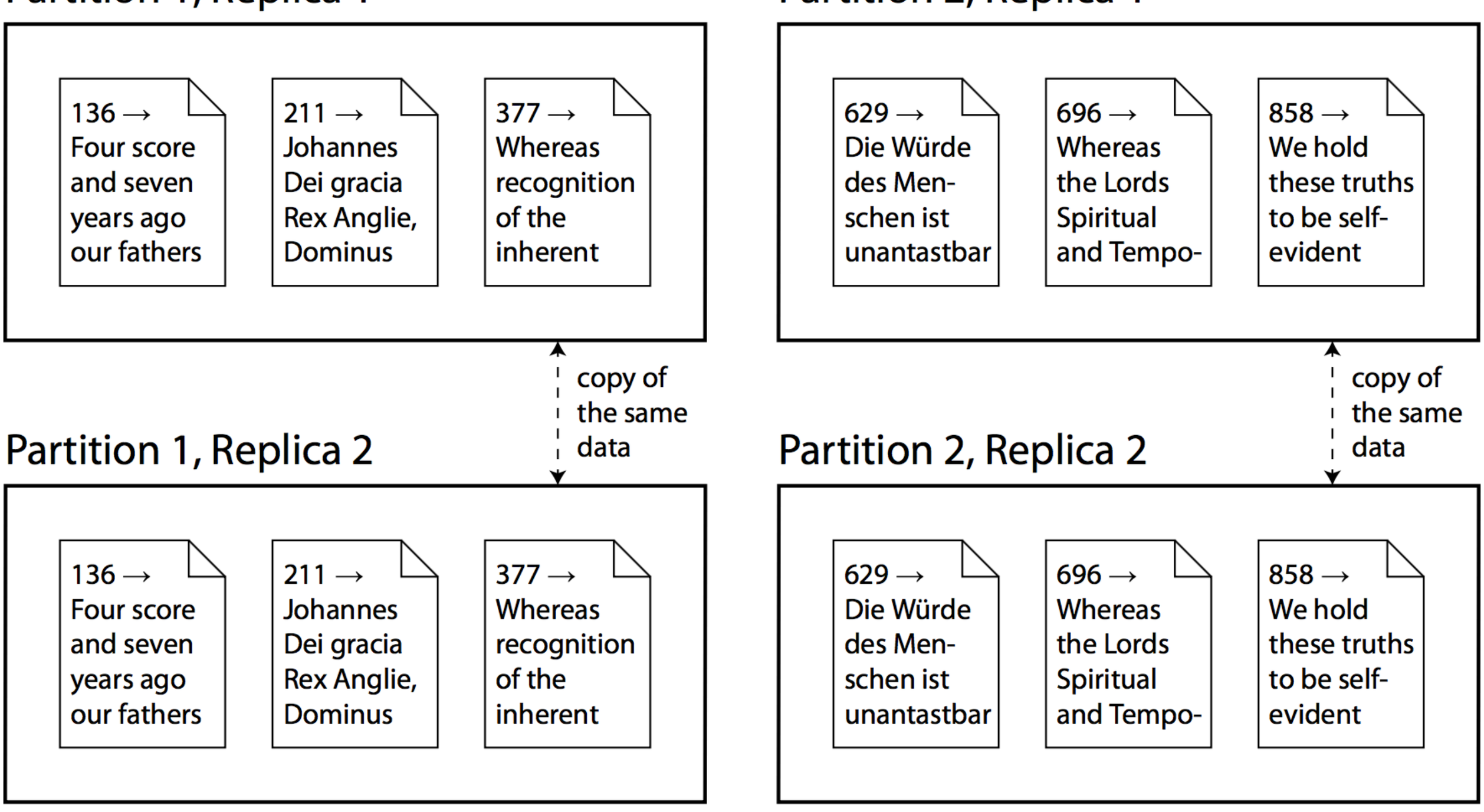
mins

$$A = 10080/10802 = 0.\underline{999}8$$

$$p_f = 2/10802$$

$$A = (1-p_f)^{10} = 0.\underline{99}8$$

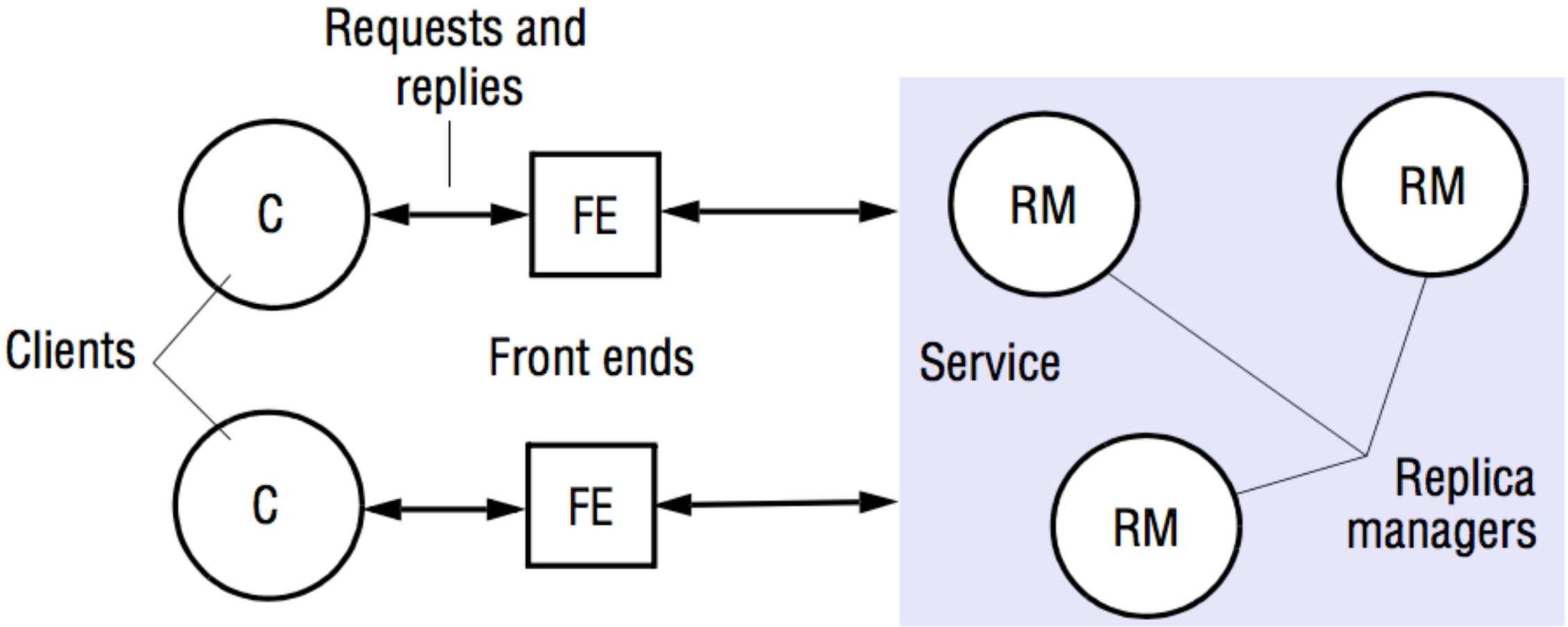
Replication & Partitioning



System Model

- The data in our system consist of a collection of items that we shall call objects.
    - An ‘object’ could be a file, say, or a Java object.
- Each logical object is implemented by a collection of physical copies called replicas.
- The replicas are physical objects, each stored at a single computer.
- The ‘replicas’ of a given object are not necessarily identical, at least not at any particular point in time. Some replicas may have received updates that others have not received.
- We assume an asynchronous system in which processes may fail only by crashing

Basic Replication Architecture



General Request Phases

- **Request**: The front end issues the request to one or more replica managers:
    - either the front end communicates with a single replica manager, which in turn communicates with other replica managers;
    - or the front end multicasts the request to the replica managers.
- **Coordination**: The replica managers coordinate in preparation for executing the request consistently. They agree, if necessary at this stage, on whether the request is to be applied (it might not be applied at all if failures occur at this stage). They also decide on the ordering of this request relative to others (mostly FIFO).
- **Execution**: The replica managers execute the request
    - Perhaps **tentatively**: that is, in such a way that they can undo its effects later.
- **Agreement**: The replica managers reach consensus on the effect of the request – if any – that will be committed.
- **Response**: One or more replica managers responds to the front end.

Correctness criteria

- *Intuitively*, a service based on replication is correct
  - if it keeps responding despite failures and
  - if clients cannot tell the difference between the service they obtain from an implementation with replicated data and one provided by a *single* correct replica manager
- A single server managing a single copy of the objects would serialize the operations of the clients.
- We need some consistency criteria capturing the requirements concerning the ordering (virtual interleaving) in which the requests are processed by replica managers.
- There are several consistency models

Message from Amazon

"Whether or not inconsistencies are acceptable depends on the client application. In all cases **must be** consiste are provided by the storage systems and must be taken into account when developing applications."

Se non lo sapevi, sallo!!!!

Amazon vice-president and Chief Scientific Officer

W. Vogels. *Eventual consistent*. **Comm. of the ACM**, 52(1):40–44, 2009

Consistency

- **Consistency model** – A contract between a distributed data store and a set of processes, which specifies what the results of read/write operations are in the presence of concurrency
- **Strong consistency** models
  - Strict consistency
  - Linearizability
  - Sequential consistency
- **Weak consistency** models
  - Eventual consistency
  - Client-centric consistency models
    - Read-after-read (*monotonic read*)
    - Read-after-write (*read your writes*)
  - Causal consistency

Strict Consistency

Definition

- A read operation must return the result of the latest write operation which occurred on the data item

Implementation:

- Only possible with a global, perfectly synchronized clock
- Only possible if all writes instantaneously visible to all
- It is the model of uniprocessor systems!

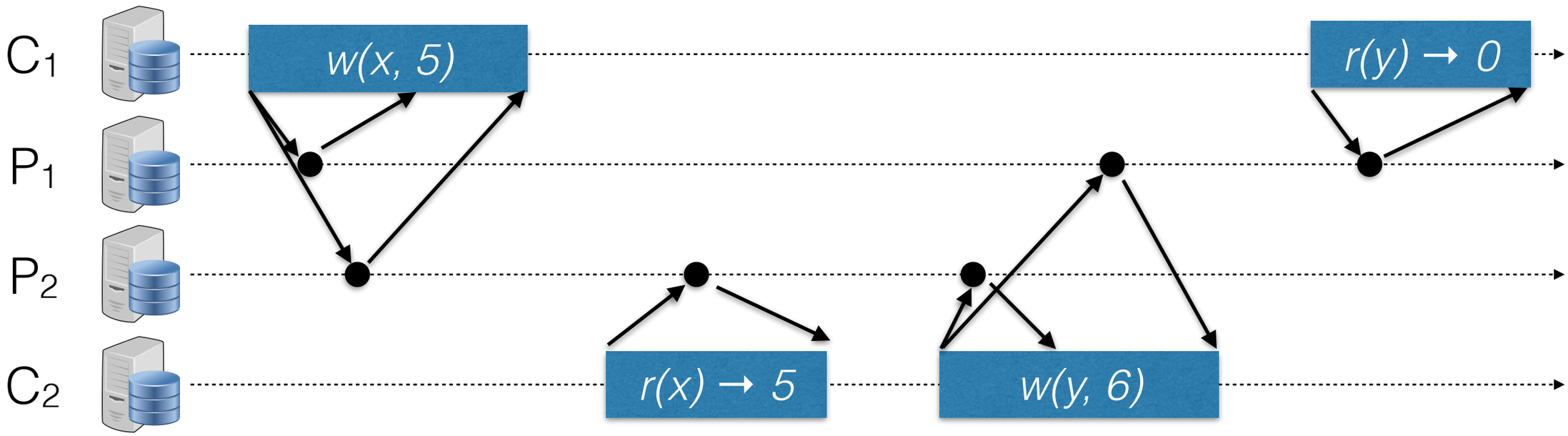
Linearizability

Definition

An execution E is **linearizable** provided that there exists a sequence (*linearization*) H such that:

- H *contains exactly* the same operations that occur in E, each paired with the return value received in E
- H is a *legal history* of the sequential data type that is replicated
- the total order of operations in H is *compatible with* the **real-time partial order** <
  - $o_1 < o_2$ means that the duration of operation o_1 (from invocation till it returns) occurs entirely before the duration of operation o_2

Example



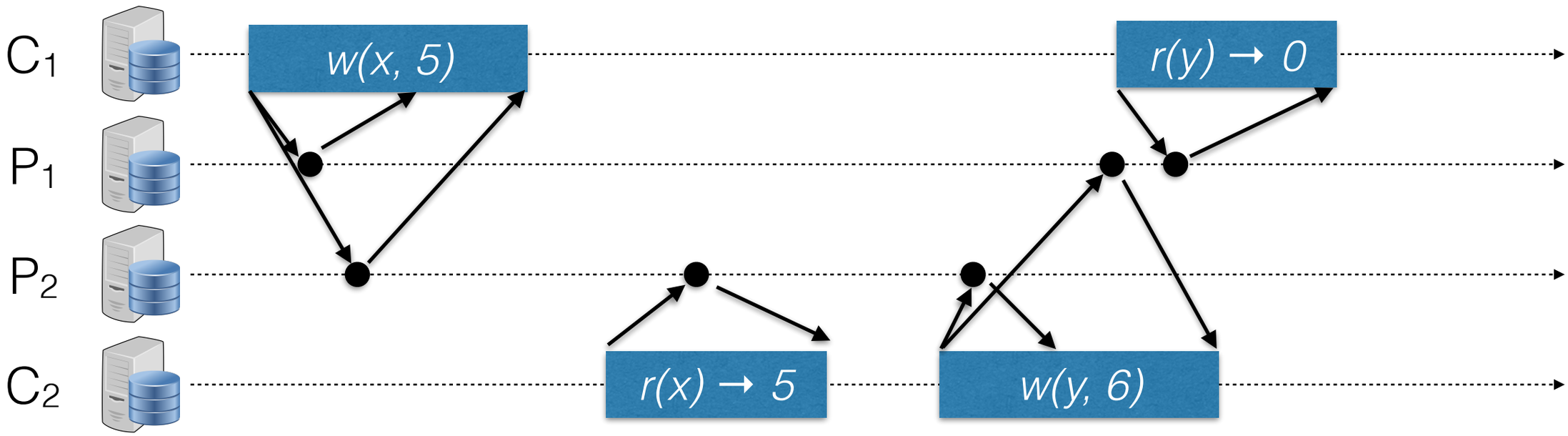
Are the following sequences possible linearizations?

w(x, 5) *r(x) → 5* *w(y, 6)* *r(y) → 0* NO

w(x, 5) *r(x) → 5* *r(y) → 0* *w(y, 6)* NO

Is the above execution linearizable? NO

Example



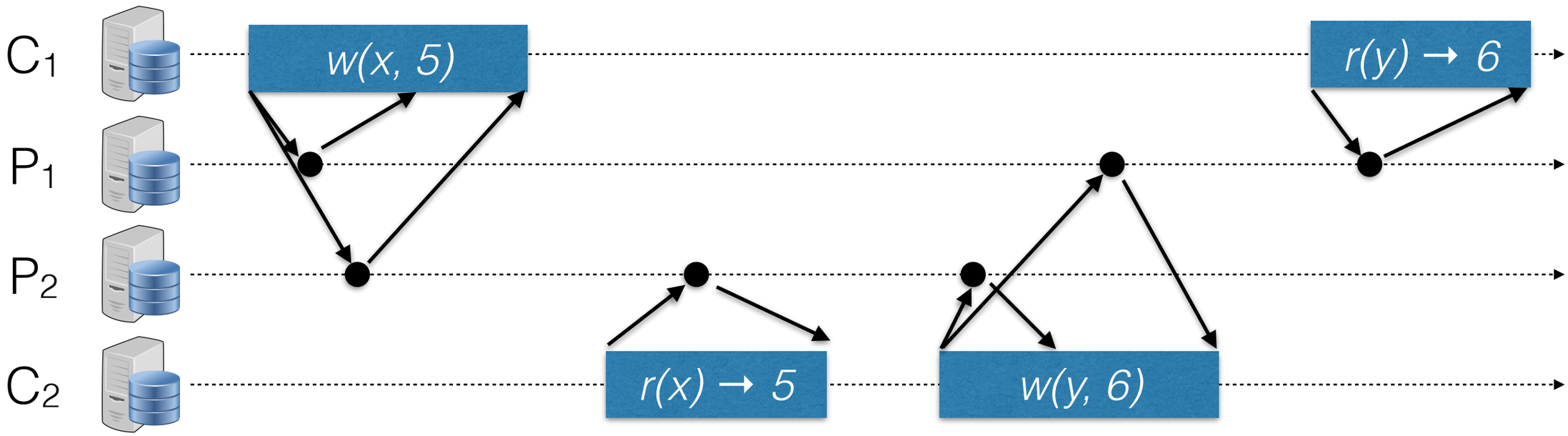
Are the following sequences possible linearizations?

w(x, 5) *r(x) → 5* *w(y, 6)* *r(y) → 0* NO

w(x, 5) *r(x) → 5* *r(y) → 0* *w(y, 6)* YES

Is the above execution linearizable? YES

Example



Are the following sequences possible linearizations?

w(x, 5) *r(x) → 5* *w(y, 6)* *r(y) → 6* YES

w(x, 5) *r(x) → 5* *r(y) → 6* *w(y, 6)* NO

Is the above execution linearizable? YES

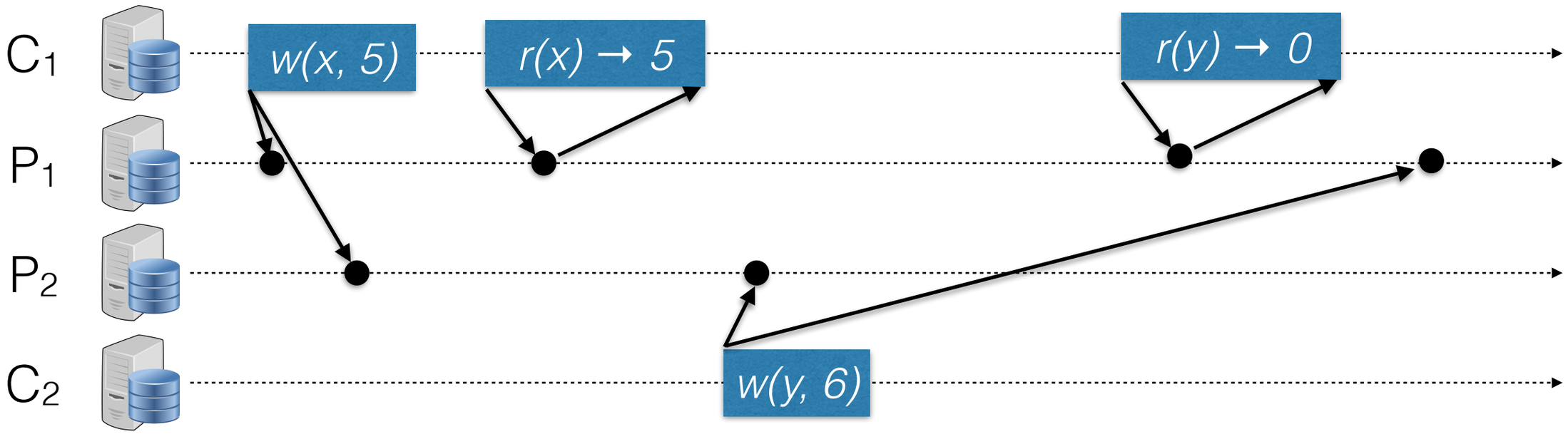
Sequential Consistency

Definition

An execution E is **sequential consistent** provided that there exists a sequence H such that

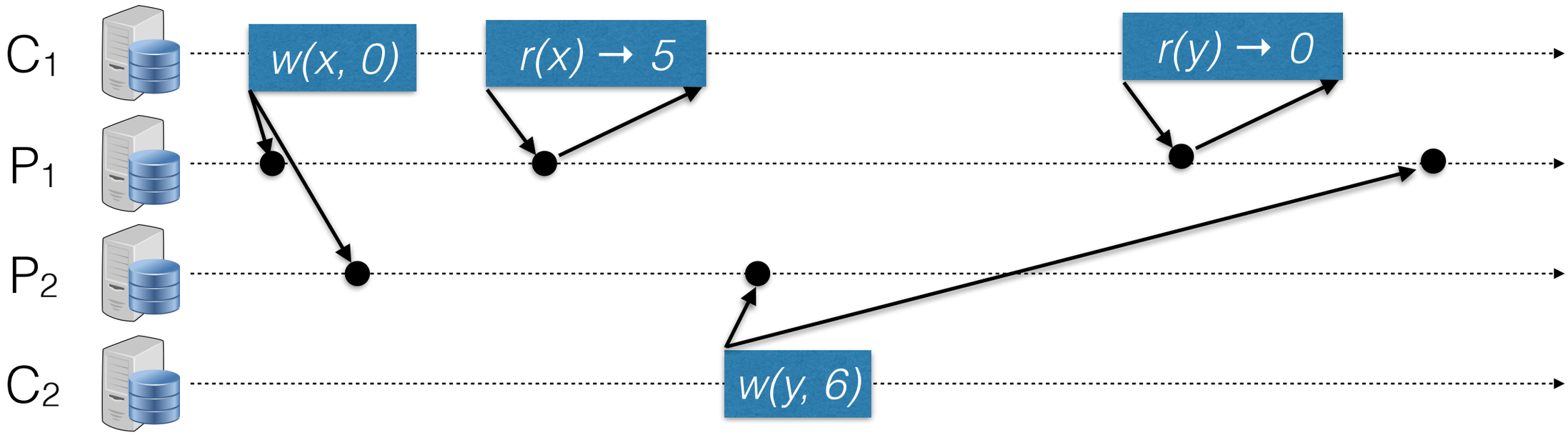
- H *contains exactly* the same operations that occur in E, each paired with the return value received in E
- H is a *legal history* of the sequential data type that is replicated
- The total order of operations in H is *compatible with* the **client partial order** <
  - $o_1 < o_2$ means that the o_1 and o_2 occur at the same client and that o_1 returns before o_2 is invoked

Example



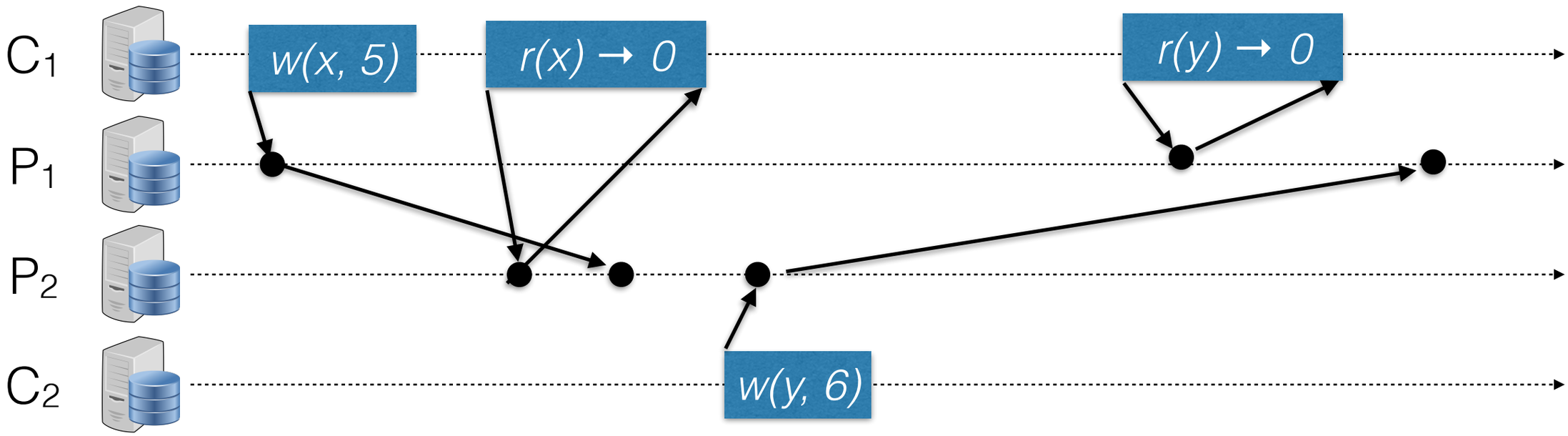
Is the execution above sequentially consistent? YES

Example



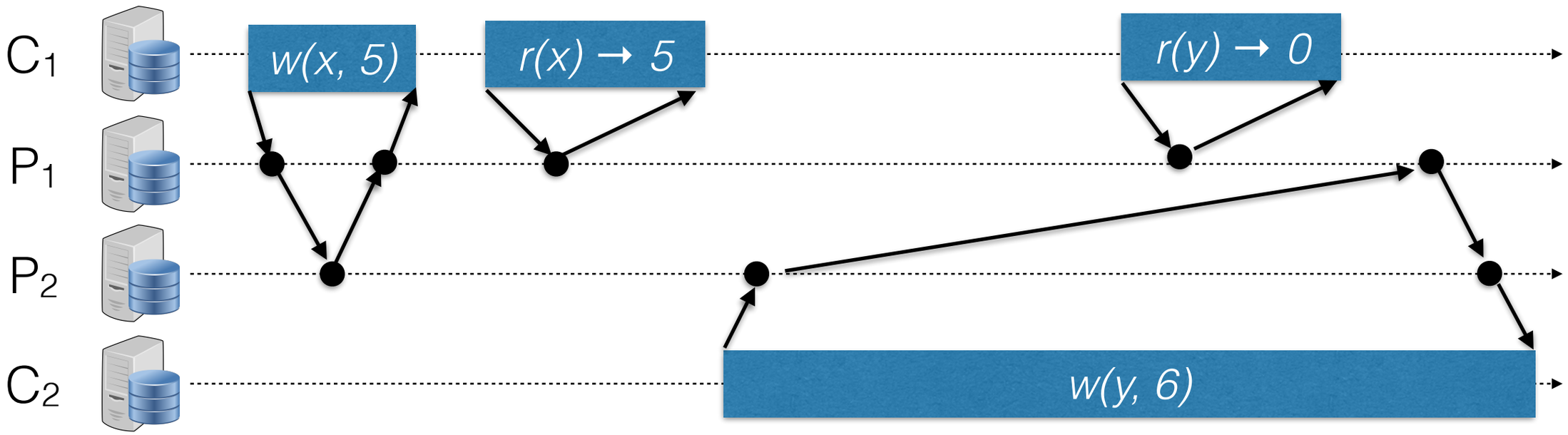
Is the execution above sequentially consistent? NO

Example



Is the execution above sequentially consistent? NO

Example



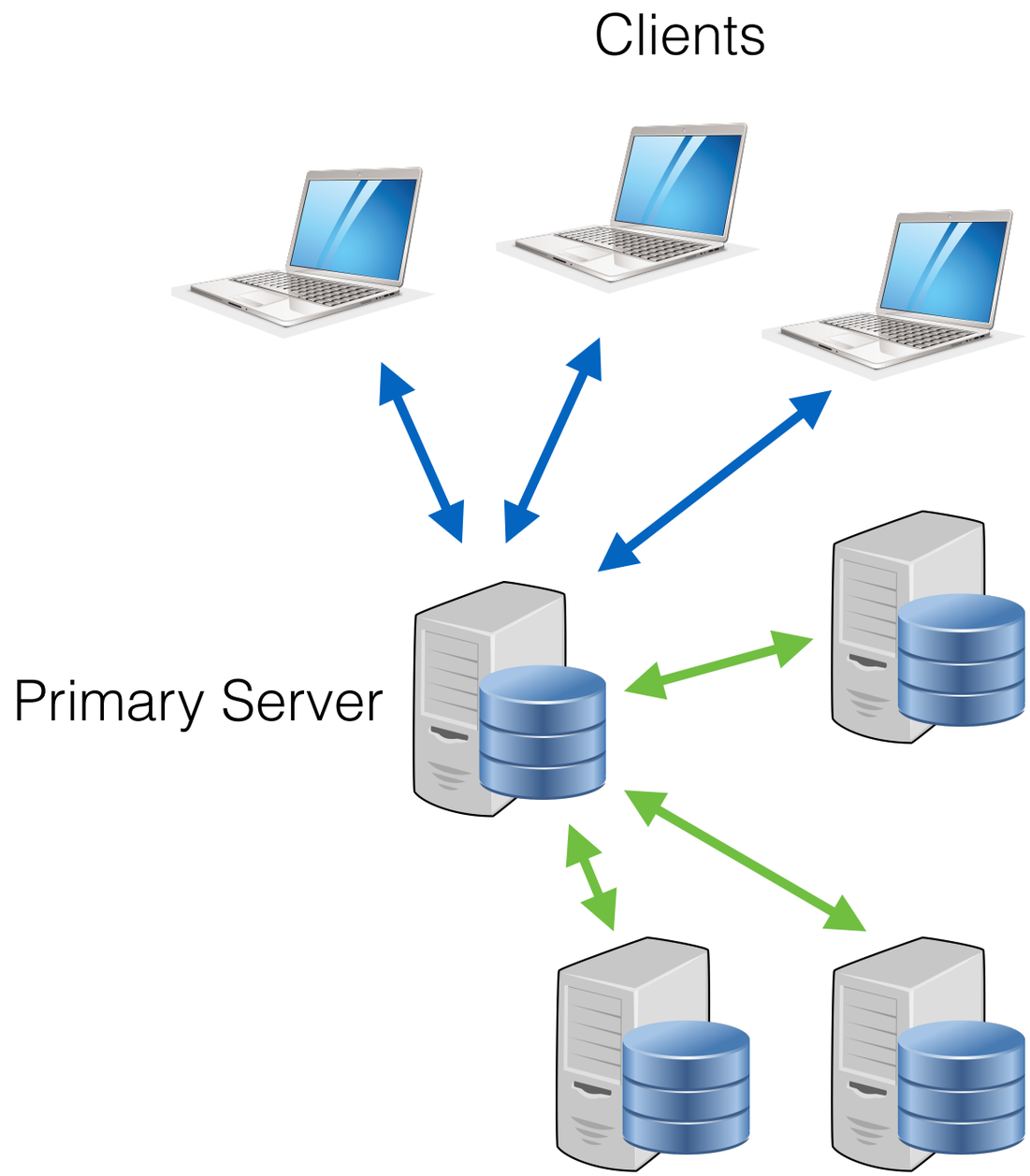
Is the execution above sequentially consistent? YES

Issues

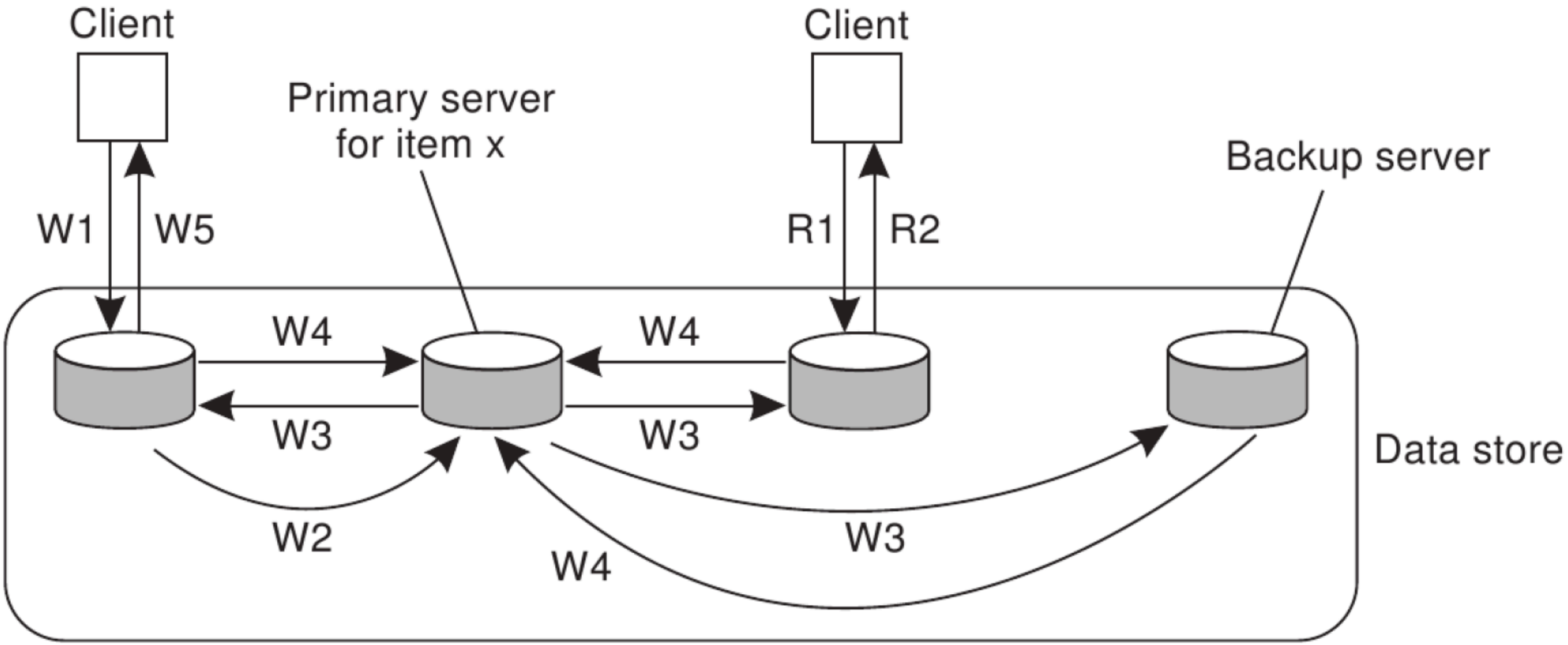
- It is easy to provide **strong consistency** through appropriate hardware and/or software mechanisms
  - But these are typically found to incur considerable **penalties**, in latency, availability after faults, etc.
- **Strong consistency** often implies that *message should arrive in the same order*
  - Can be implemented through a **sequencer replica**
    - Latency: the sequencer replica becomes a bottleneck
    - Availability: a new sequencer must be elected after a failure
- **Weak consistency** relaxes the *precise details* of *which reorderings are allowed*
  - Within the activity of a client
  - By whether there are any constraints at all on the information provided to different clients

Passive Replication

- Clients communicate with primary server
- WRITES are atomically forwarded from primary server to backup servers
- READS are replied by the primary server
- Also known as Primary Copy (or Backup) Replication
- Specifications:
  - At most one replica can be the primary server at any time.
  - Each client maintains a variable L (leader) that specifies the replica to which it will send requests. Requests are queued at the primary server.
  - Backup servers ignore client requests.



Passive Replication Protocol



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

Request Phases

- **Request**: The front end issues the request, containing a unique identifier, to the primary replica manager.

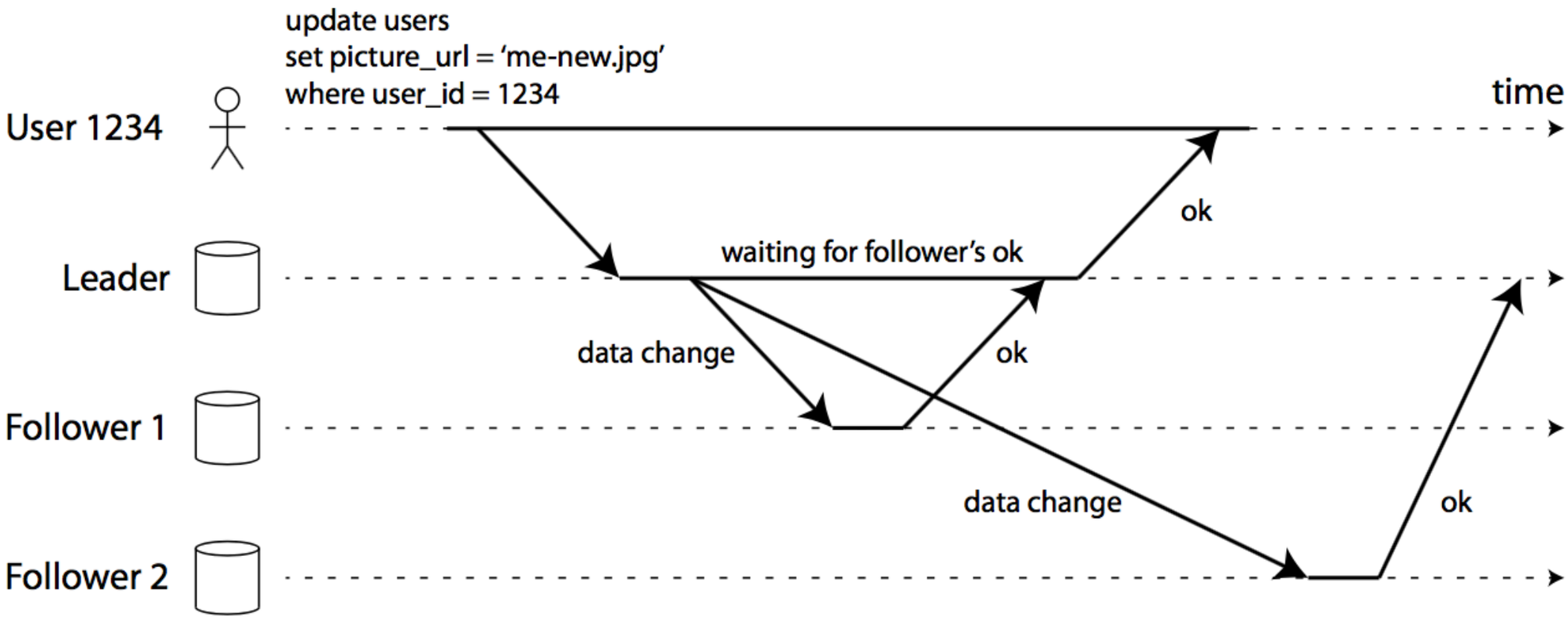
- **Coordination**: The primary takes each request atomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request, and if so it simply resends the response.

- **Execution**: The primary executes the request and stores the response.

- **Agreement**: If the request is an update, then the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.

- **Response**: The primary responds to the front end, which hands the response back to the client.

Synchronous vs Asynchronous



- The **advantage** of synchronous replication is that the follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader.
    - If the leader suddenly fails, we can be sure that the data is still available on the follower.
- The **disadvantage** is that if the synchronous follower doesn't respond (because it has crashed, or there is a network fault, or for any other reason), the write cannot be processed.
    - The leader must block all writes and wait until the synchronous replica is available again.

Primary Failure

- When the primary replica fails, a failover procedure is required (can be manual or automatic)
    1. Determining that the leader has failed.
    2. Choosing a new leader.
    3. Reconfiguring the system to use the new leader.
- Issues:
    - If asynchronous replication is used, the new leader may not have received all writes from the old leader before it failed.
    - Discarding writes is especially dangerous if other storage systems outside of the database need to be coordinated with the database contents.
    - It could happen that two nodes both believe that they are the leader

Replication Log

- The primary replica stores all changes locally, in a replication log
- Applying the replication allows us to perform the correct update on any object (given the correct log sequence number)
- This is used so set up new backup replicas, given a snapshot of the objects, the corresponding log sequence number, and the primary's replication log.
- The same holds for catch-up recovery of failing backup replicas.

Implementing Replication Log

- Statement Log: the primary logs every write request (*statement*) that it executes, and sends that statement log to its followers.
    - *Potential issues*: non-deterministic values (rand()), concurrency issues, side effects on other components
- Write-Ahead Log: similarly to LSM trees, the primary append every write requests in the log, and sends the whole sequence of write requests
    - *Potential issues*: the log describes the data on a very low level: a WAL contains details of which bytes were changed in which disk block. What if we update something?

Simple Protocol

- System model:
  - point-to-point communication
  - no communication failures → no network partitions
  - upper bound on message delivery time → synchronous communications
  - FIFO channels
  - at most one server crashes
- Two servers:
  - The primary p_1
  - The backup p_2
- Variables:
  - At *server* p_i, primary = true if p_i acts as the current primary
  - At *clients*, primary is equal to the identifier of the current primary

Simple Protocol

Protocol executed by the primary p_1

upon initialization **do**
  $primary \leftarrow$ **true**

upon receive $\langle \text{REQ}, r \rangle$ **from** c **do**
  $state \leftarrow \mathsf{update}(state, r)$ % Update local state
  send $\langle \text{STATE}, state \rangle$ **to** p_2 % Send update to backup
  send $\langle \text{REP}, \mathsf{reply}(r) \rangle$ **to** c % Reply to client

repeat every τ seconds
  send $\langle \text{HB} \rangle$ **to** p_2 % Heartbeat message

upon recovery after a failure **do**
  { start behaving like a backup }

Simple Protocol

Protocol executed by the backup p_2

upon initialization **do**
  $primary \leftarrow$ **false**

upon receive $\langle \text{STATE}, s \rangle$ **do**
  $state \leftarrow s$ % Update local state

upon not receiving a heartbeat for $\tau + \delta$ seconds **do**
  $primary \leftarrow$ **true** % Becomes new primary
  send $\langle \text{NEWP} \rangle$ **to** c % Inform the client of new primary
  { start behaving like a primary }

Simple Protocol

Protocol executed by client c

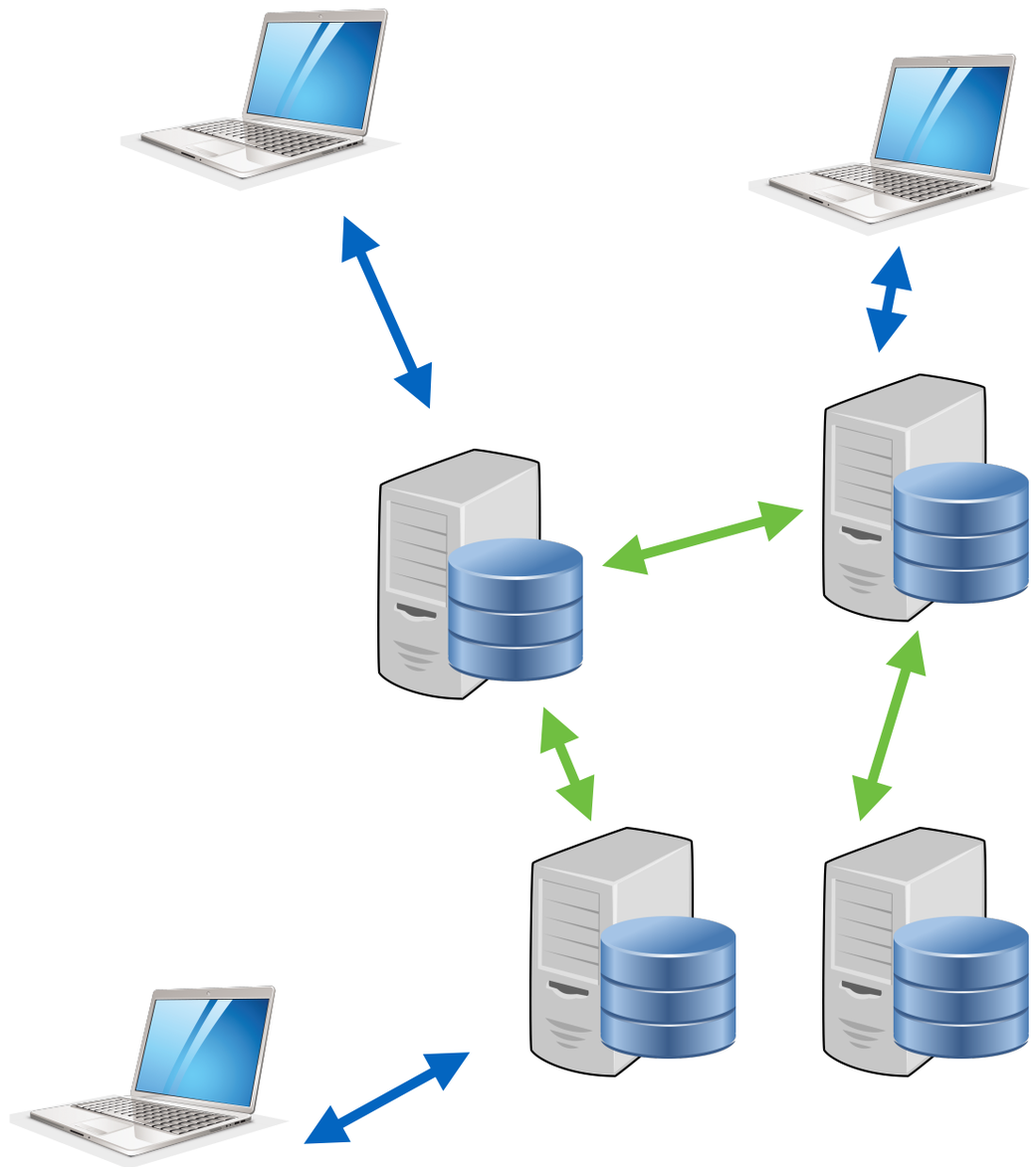
upon initialization **do**
    $primary \leftarrow p_1$ % Initial primary

upon receive $\langle \text{NEWP} \rangle$ **from** p_2 **do**
    $primary \leftarrow p_2$ % Backup

upon operation(r) **do**
    while not received a reply **do**
        send $\langle \text{REQ}, r \rangle$ **to** $primary$
        wait receive $\langle \text{REP}, v \rangle$ **or receive** $\langle \text{NEWP} \rangle$
    return v

Active Replication

- a.k.a. multi-leader replication
- Clients communicate with several/all servers
- Every server handles any operation and sends the response
- WRITES must be applied in the same order (**total order broadcast**)
- One way to implement totally-ordered multicast is to use logical clocks



Use Cases

- When does it make sense to use active replication?
    - multi-datacenter operations (increased performance and fault tolerance w.r.t. passive replication)
- Clients with offline operations
    - Each client device is a 'datacenter'
- Collaborative editing
    - Each copy is a 'datacenter'

Request Phases

- **Request**: The front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive.

- **Coordination**: The group communication system delivers the request to every correct replica manager in the same (total) order.

- **Execution**: Every replica manager executes the request. Since they are state machines and since requests are delivered in the same total order, correct replica managers all process the request identically. The response contains the client's unique request identifier.

- **Agreement**: No agreement phase is needed, because of the multicast delivery semantics.

- **Response**: Each replica manager sends its response to the front end. The number of replies that the front end collects depends upon the failure assumptions and the multicast algorithm.

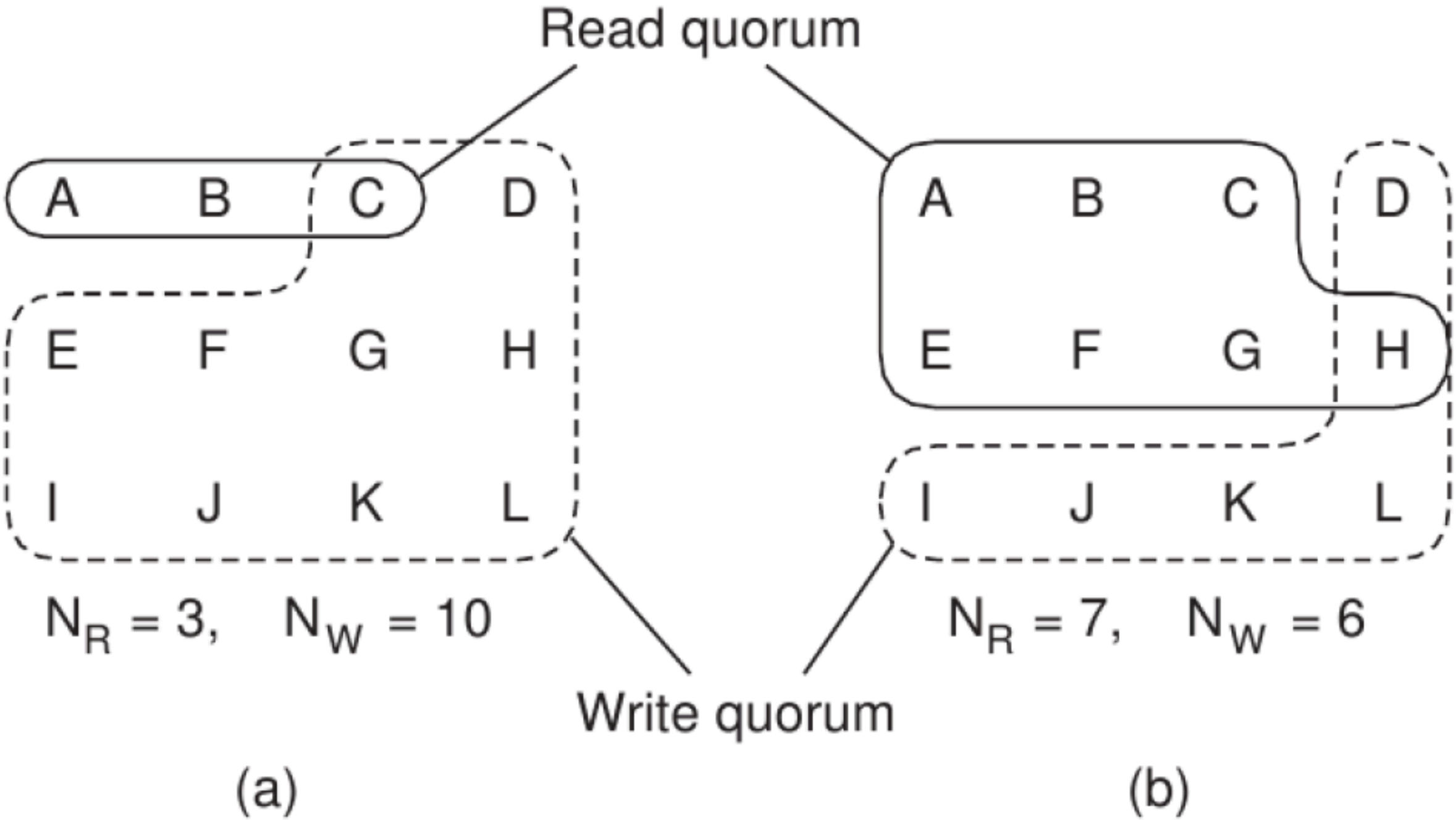
Quorum Protocols

- Proposed by Gifford in 1979
- **Quorum-based protocols** guarantee that each operation is carried out in such a way that a *majority vote* (a quorum) is established.
  - *Write quorum W*: the number of replicas that need to acknowledge the receipt of the update to complete the update
  - *Read quorum R*: the number of replicas that are contacted when a data object is accessed through a read operation

Quorum Systems

- Formally, a **quorum system** S = {S_1, …, S_N} is a collection of **quorum sets** $S_i \subseteq U$ such that two quorum sets have at least an element in common
- For replication, we consider two quorum sets, a **read quorum** R and a **write quorum** W
- **Rules**:
  1. Any read quorum must overlap with any write quorum
  2. Any two write quorums must overlap
- U is the set of replicas, i.e., |U| = N

Quorum Examples



Quorum Examples

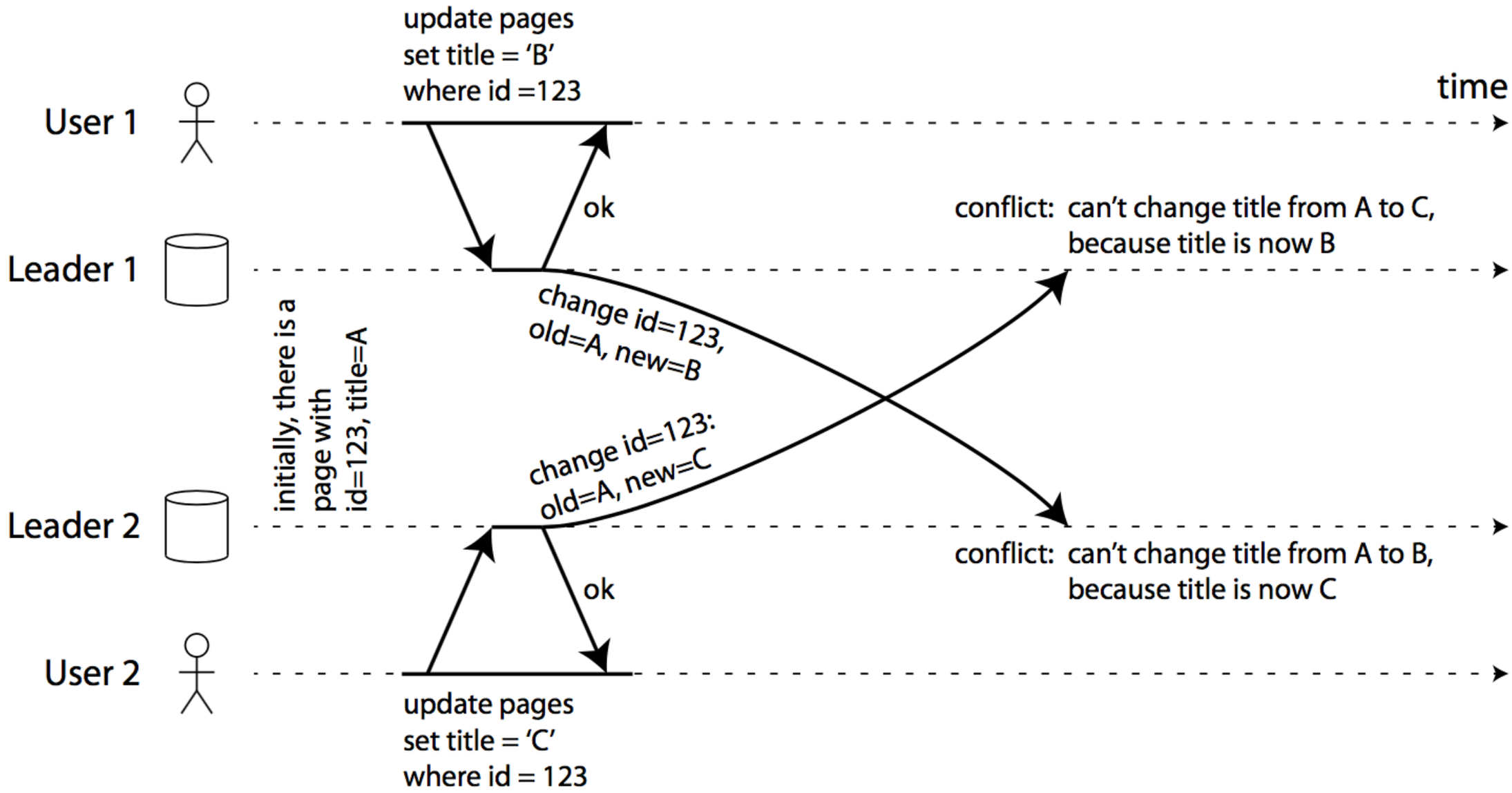
- Read rule: |R| + |W| > N ⇒ read and write quorums overlap
- Write rule: 2 |W| > N ⇒ two write quorums overlap
- The quorum sizes determine the costs for read and write operations
- Minimum quorum sizes for are

$$\min |W| = \left\lfloor \frac{N}{2} \right\rfloor + 1 \qquad \min |R| = \left\lceil \frac{N}{2} \right\rceil$$

  - Write quorums requires majority
  - Read quorum requires at least half of the nodes
- ROWA (R,W,N) = (N = N, R = 1, W = N)
- Amazon’s Dynamo (N = 3, R = 2, W = 2)
- Linkedin's Voldemort (N = 2 or 3, R = 1, W = 1 default)
- Apache's Cassandra (N = 3, R = 1, W = 1 default)

Write Conflicts

The biggest problem with active replication is that write conflicts can occur, which means that conflict resolution is required.



Handling Write Conflicts

- Avoid them by 'normally' using a single leader, and change leader for exceptional conditions only.

- Converge towards a consistent state

    - Give each write a unique ID, pick the write with the highest ID as the winner, and throw away the other writes. If a timestamp is used, this technique is known as last write wins (LWW). Although this technique is popular, it is dangerously prone to data loss [30].

    - Give each replica a unique ID, and let writes that originated at a higher-numbered replica always take precedence over writes that originated at a lower-numbered replica. This also implies data loss.

    - Record the conflict in an explicit data structure that preserves all information, and write application code which resolves the conflict at some later time (perhaps by prompting the user).

- Use custom logic

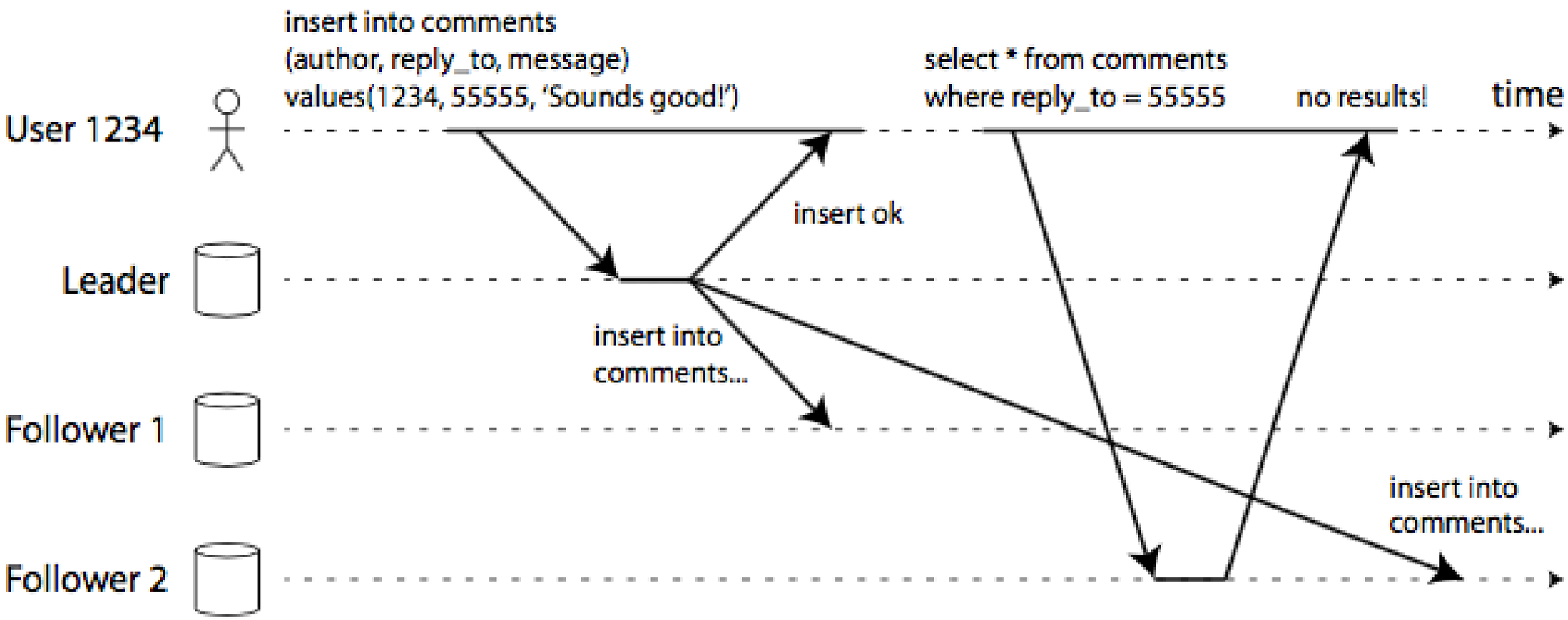
- Use automatic logic (e.g., conflict-free replicated data types, CRDTs)

Replication Lag

Unfortunately, if an application reads from asynchronous followers, it may see outdated information if the follower has fallen behind. This leads to apparent inconsistencies in the database: if you run the same query on the leader and a follower at the same time, you may get different results, because not all writes have been reflected in the follower. This inconsistency is just a temporary state — if you stop writing to the database and wait a while, the followers will eventually catch up and become consistent with the leader. For that reason, this effect is known as eventual consistency.

Read Your Writes Consistency

if the user views the data shortly after making a write, the new data may have not yet reached the replica.



Client-centric Consistency Models

- Each WRITE operation is assigned a unique identifier
  - Done by the server where the operation is requested
- For each client c, we keep track of:
  - Read set WS_R : contains write operations relevant to the read operations performed by c
  - Write set WS_W : contains write operations relevant to the write operations performed by c
- For each server, we keep track of:
  - Write set WS : contains the write operations executed so far

Read-Your-Writes Implementation

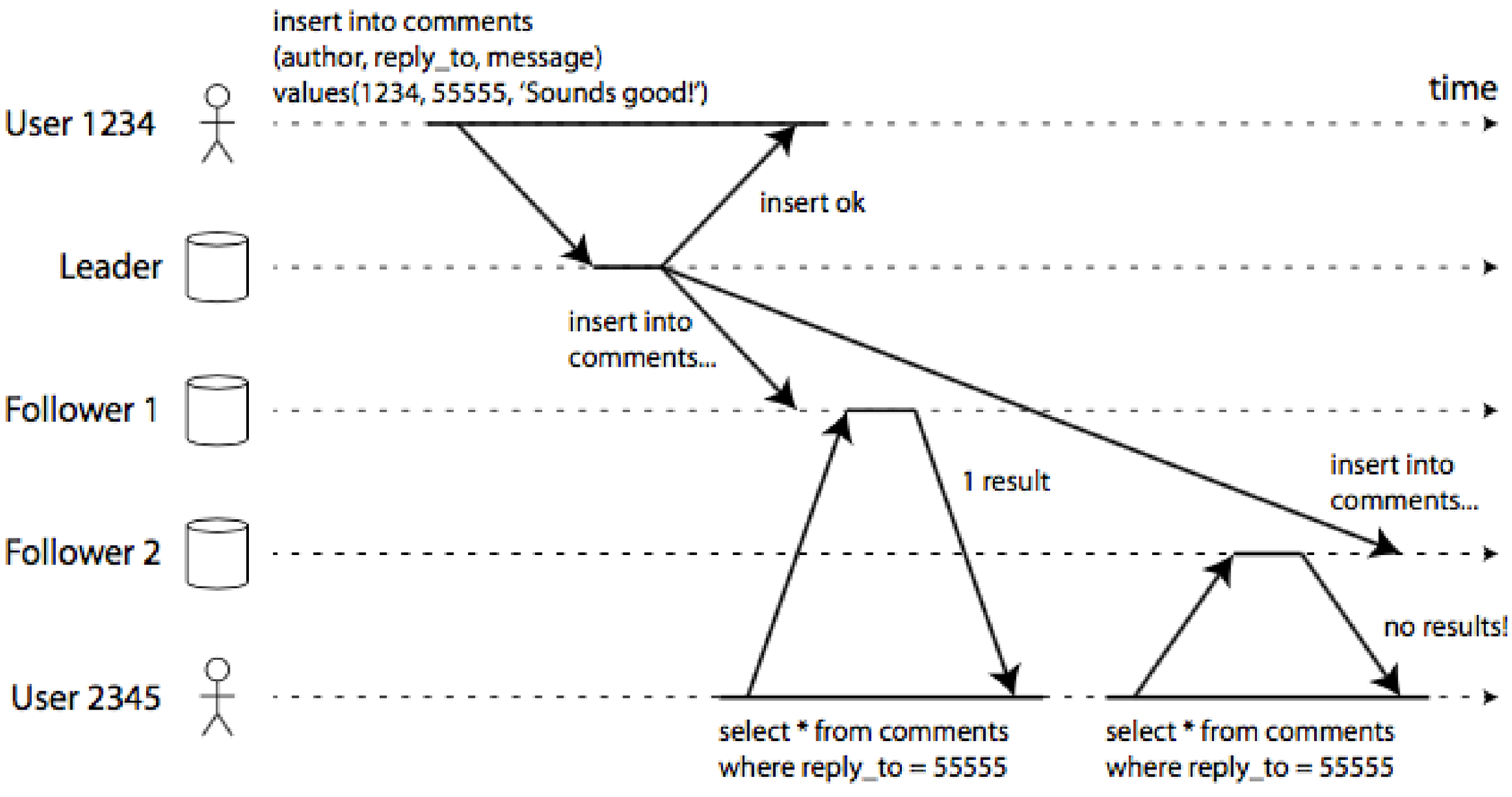
- To perform a READ:
  - A client
    - sends READ and its WS_W to a server S.
  - The server S:
    - Checks if the $WS_W \subseteq WS$, i.e., all the WRITES seen from the client have been applied by the server
    - If not, asks the other servers the missing WRITES
    - Applies the missing WRITES locally and update its WS
    - Return the requested value to the client

Read-Your-Write Implementation

- To perform a WRITE:
  - A client
    - sends WRITE and adds it to its WS_W
- The server S:
  - Perform the WRITE
  - adds it to its WS

Monotonic Reads Consistency

if a user makes several reads from different replicas, it's possible for a user to see things moving backwards in time.



Monotonic-Read Implementation

- To perform a READ:
  - A client
    - sends READ and its WS_R to a server S.
  - The server S:
    - Checks if the $WS_R \subseteq WS$, i.e., all the WRITES seen from the client have been applied by the server
    - If not, asks the other servers the missing WRITES
    - Applies the missing WRITES locally and update its WS
    - Return the requested value and WS to the client
- The client
  - adds WS to its WS_R

Monotonic-Read Implementation

- To perform a WRITE:
  - A client
    - sends WRITE
  - The server S:
    - Perform the WRITE
    - adds it to its WS

Writes-Follow-Reads & Monotonic-Writes

- Two additional constraints on the server:
  - When a server S accepts a new WRITE W_2 at time t, it ensures that WriteOrder(W_1,W_2) is true for any WRITE W_1 already in DB(S,t).
  - Anti-entropy is performed such that if WRITE W_2 is propagated from server S_1 to server S_2 at time t then any W_1 in DB(S_1,t) such that WriteOrder(W_1,W_2) is also propagated to S_2.

Additional References

- D. Terry et al., *Session Guarantees for Weakly Consistent Replicated Data*, https://www.cis.upenn.edu/~bcpierce/courses/dd/papers/SessionGuaranteesPDIS.ps

- D. Terry, *Replicated Data Consistency Explained Through Baseball*, http://research.microsoft.com/pubs/157411/ConsistencyAndBaseballReport.pdf