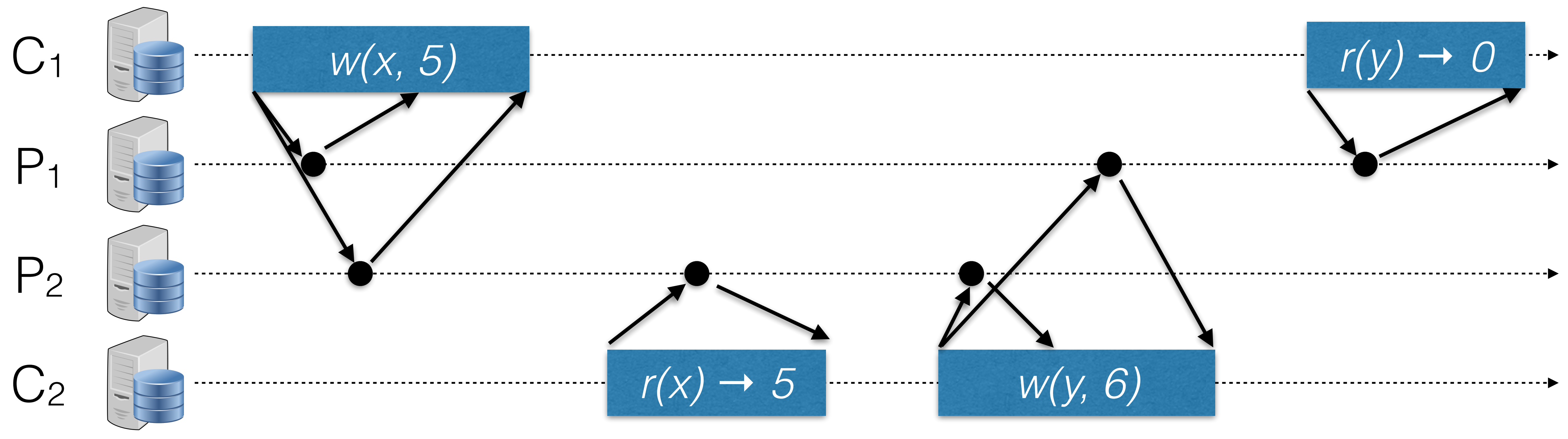
Linearizability

Definition

An execution E is **linearizable** provided that there exists a sequence of operations (*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, i.e., every read returns the last written value
- 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



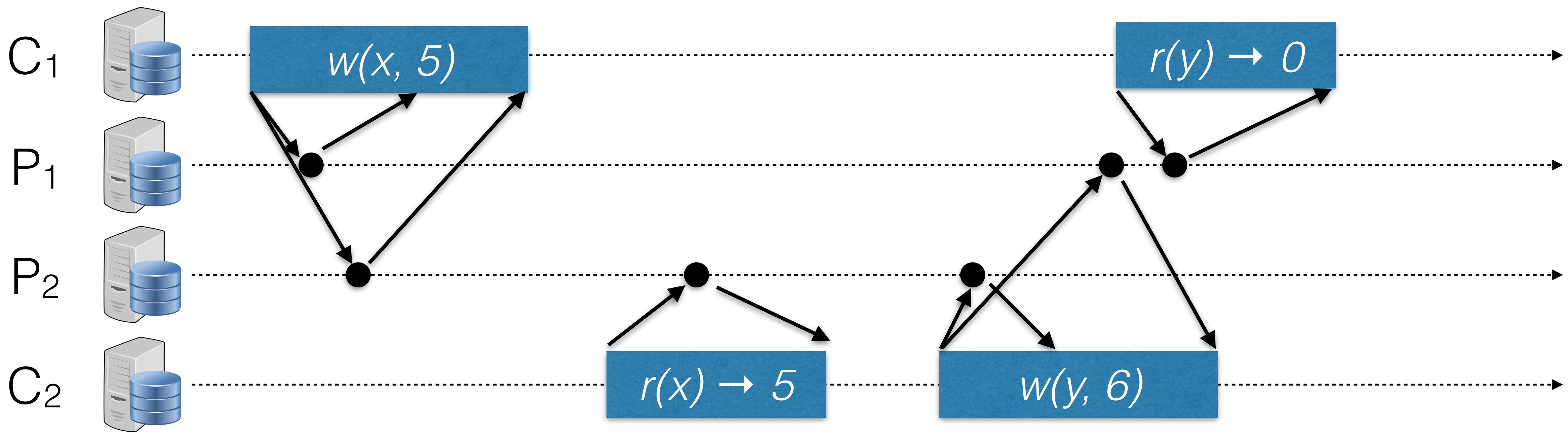
Are the following sequences possible linearizations?

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



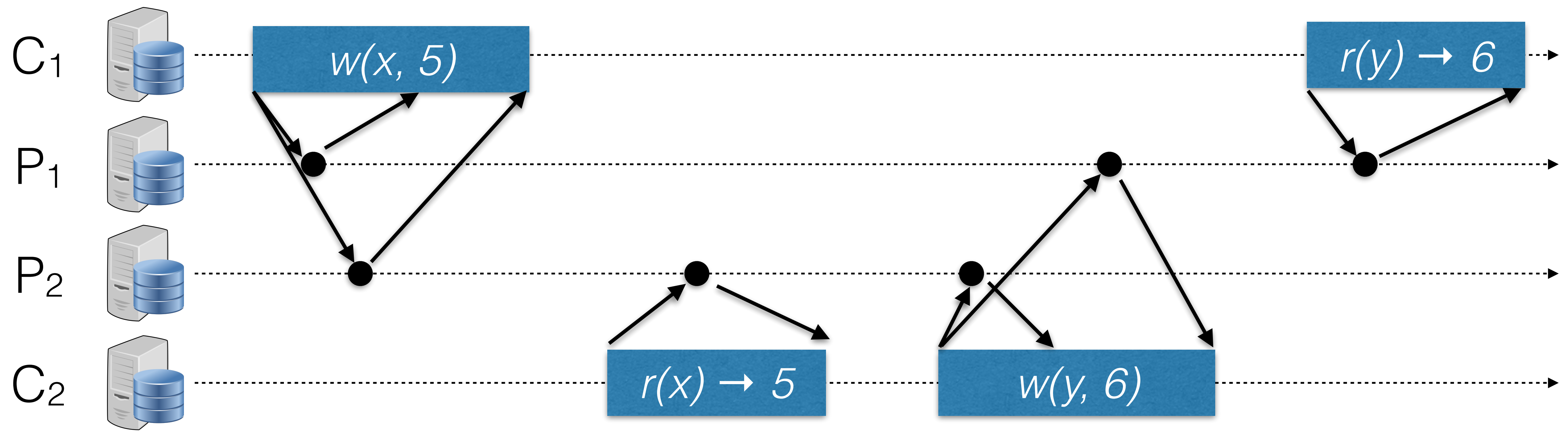
Are the following sequences possible linearizations?

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

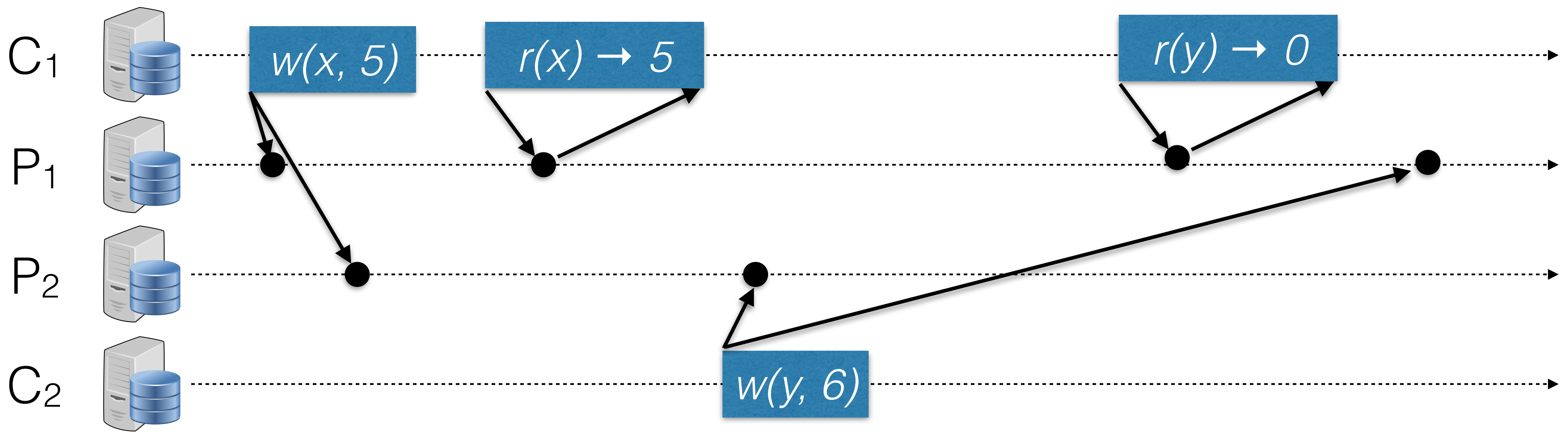
Sequential Consistency

Definition

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

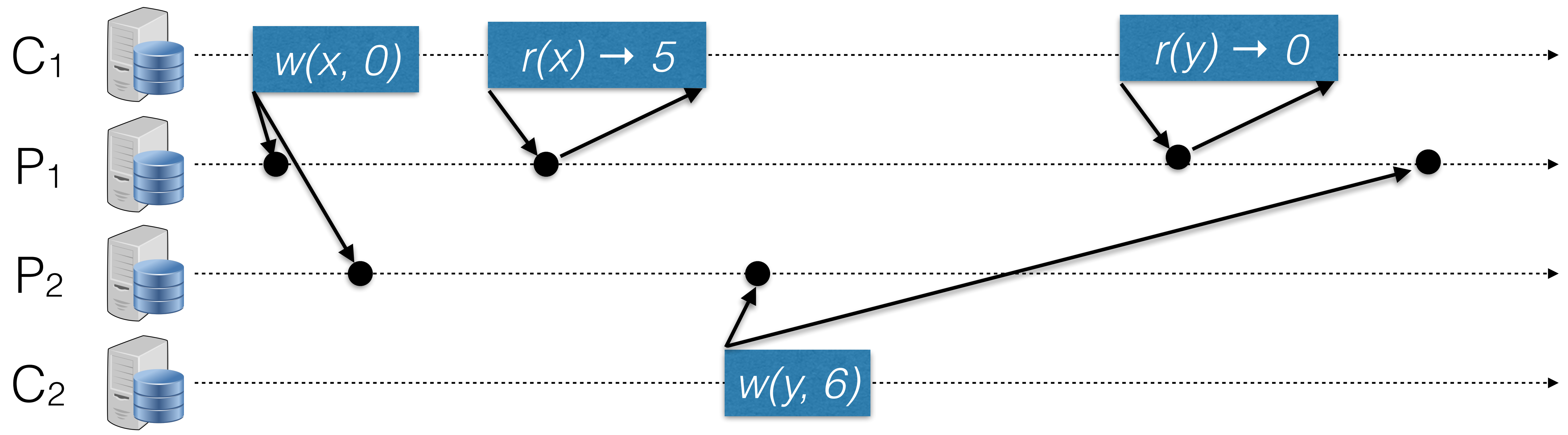
- H *contains exactly* the same operations that occur in E, each paired with the return value received in E
- H is a *legal history* of the sequential data type that is replicated
- The total order of operations in H is *compatible with* the **client partial order** <
  - $o_1 < o_2$ means that the o_1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

Example



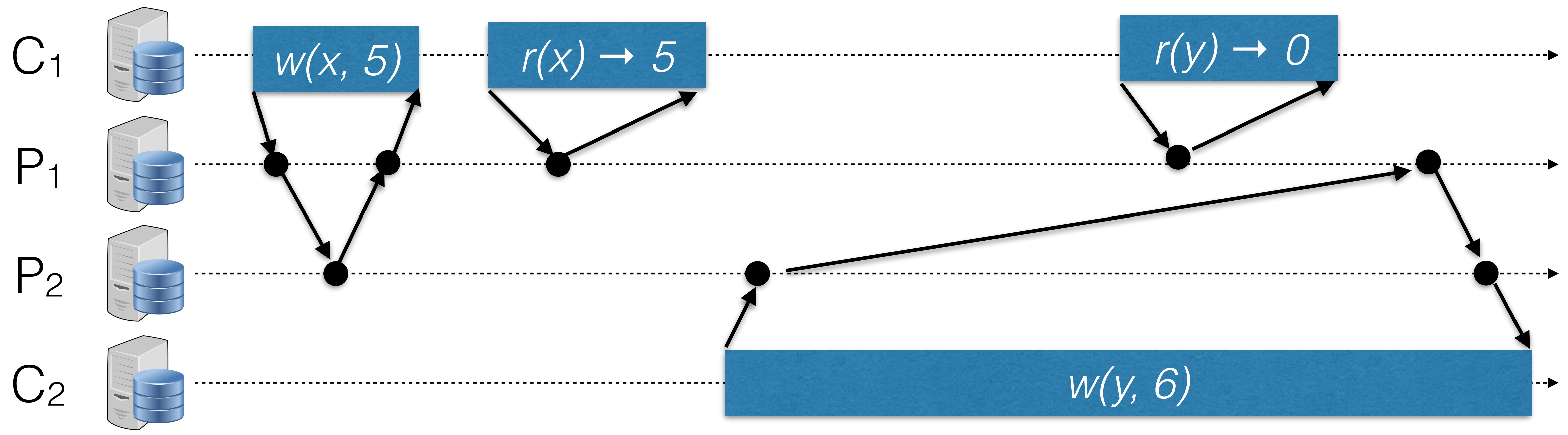
Is the execution above sequentially consistent? YES

Example



Is the execution above sequentially consistent? NO

Example



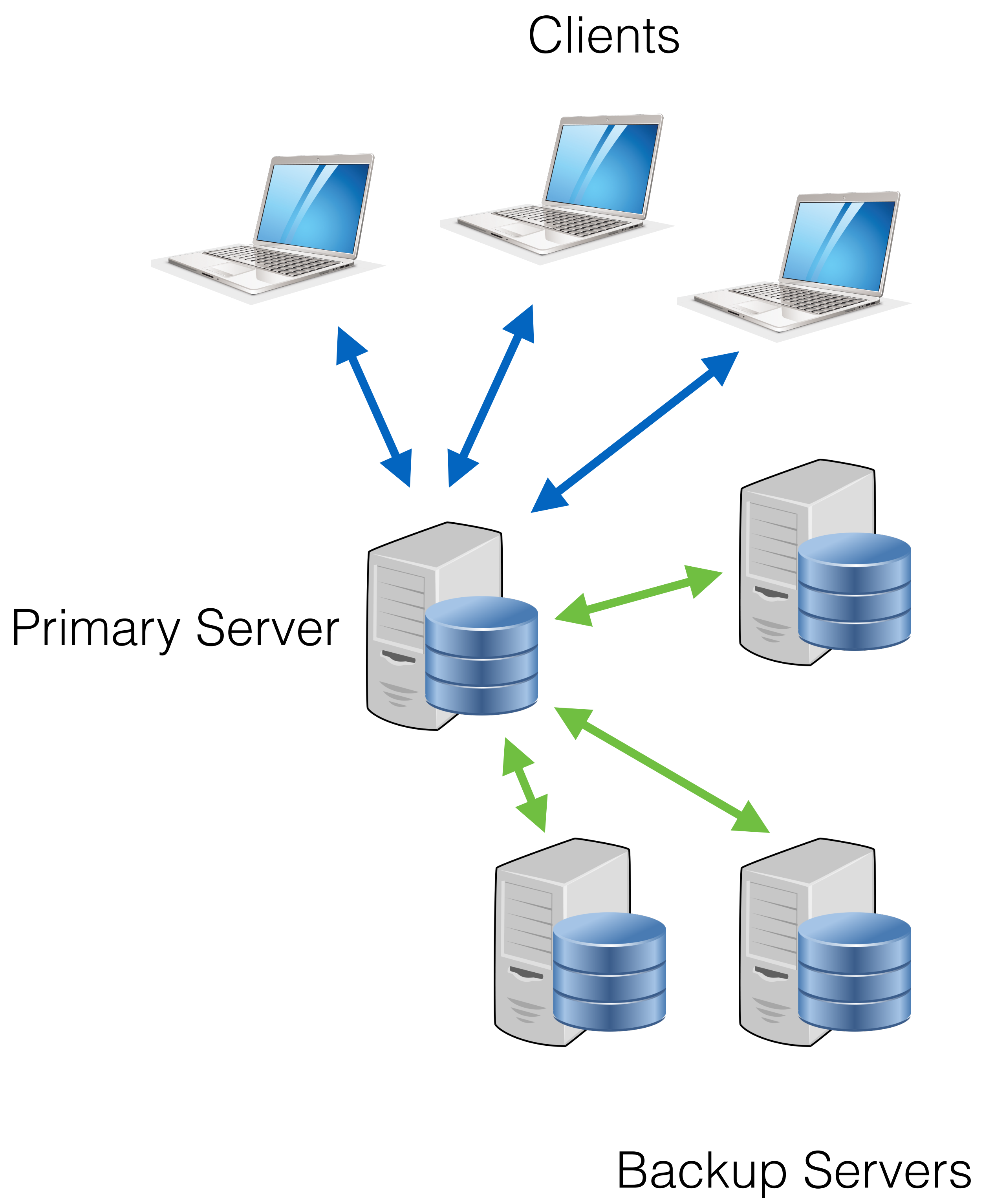
Is the execution above sequentially consistent? YES

Issues

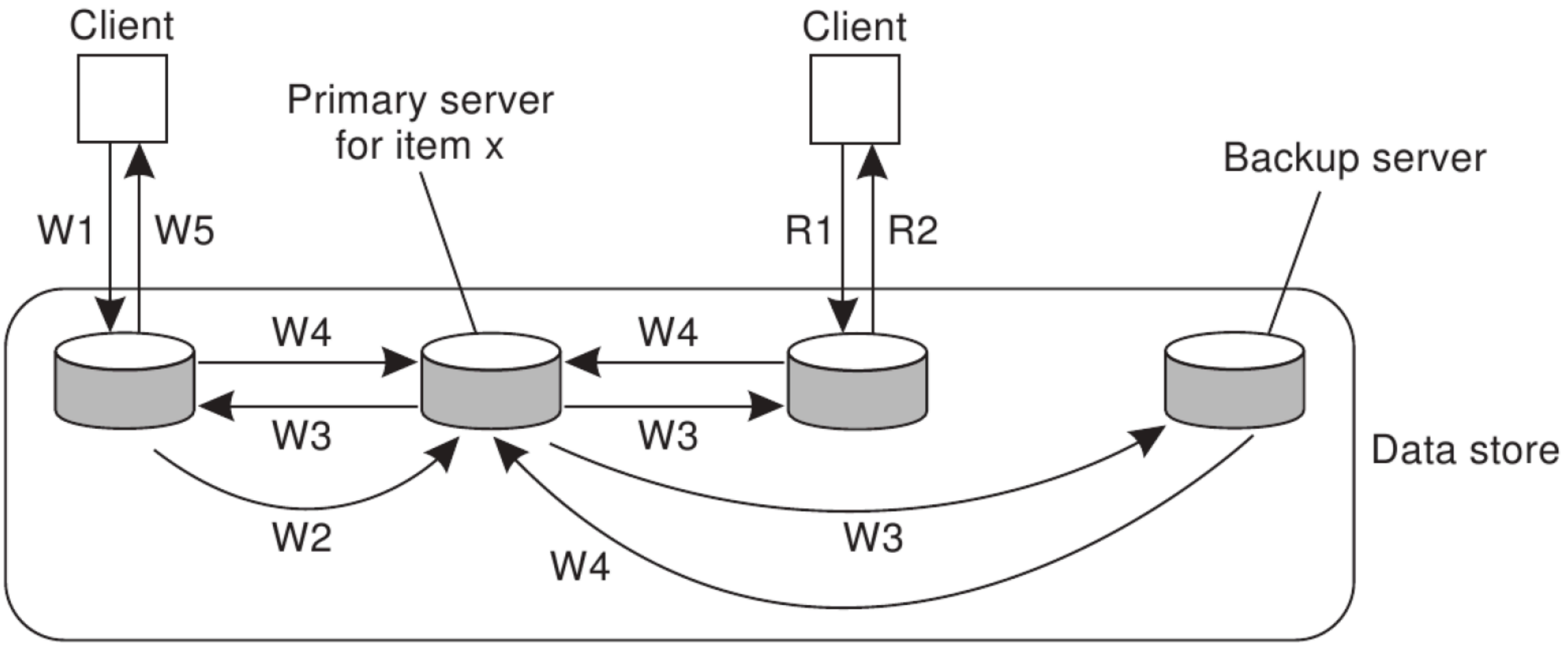
- It is easy to provide **strong consistency** through appropriate hardware and/or software mechanisms
  - But these are typically found to incur considerable **penalties**, in latency, availability after faults, etc.
- **Strong consistency** often implies that *message should arrive in the same order*
  - 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



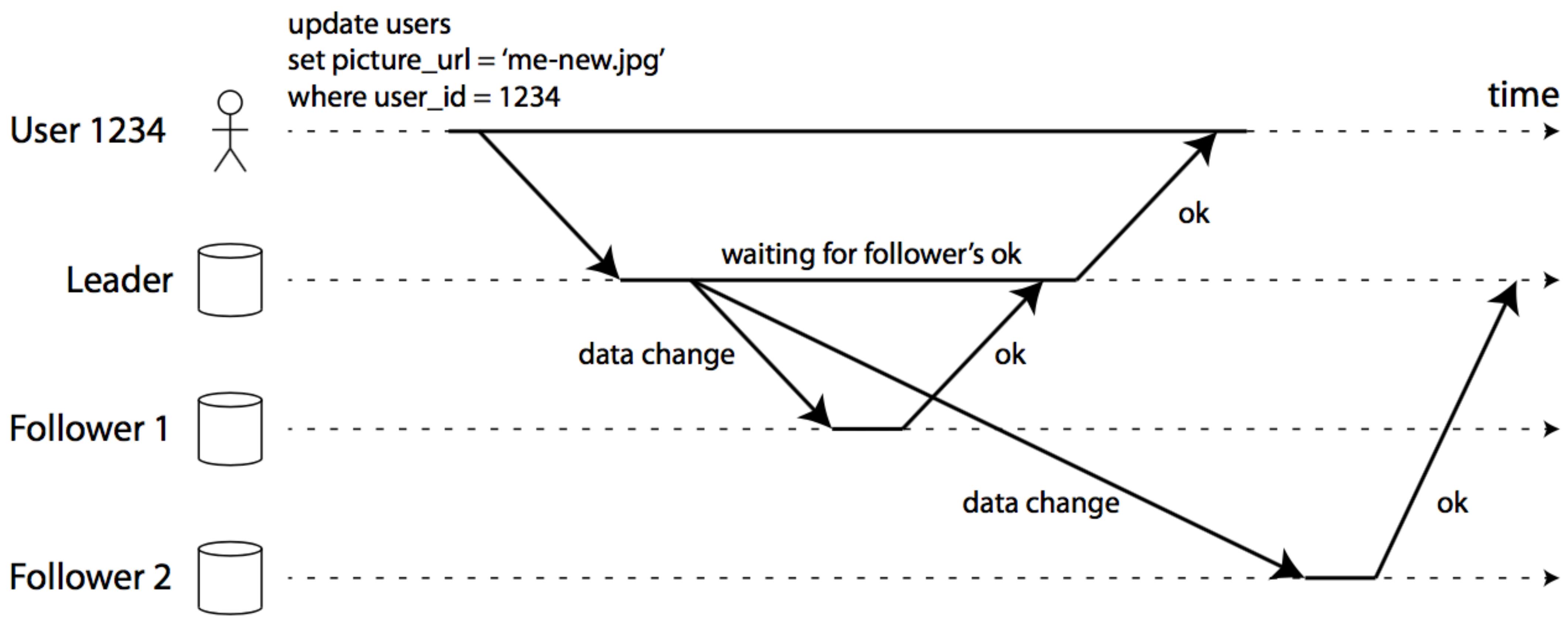
W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

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
    $primary \leftarrow \textbf{true}$

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 $\tau$ 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 $\langle$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 $\langle$NEWP$\rangle$ to $c$                     % Inform the client of new primary
    { start behaving like a primary }
```

Simple Protocol

Protocol executed by client c

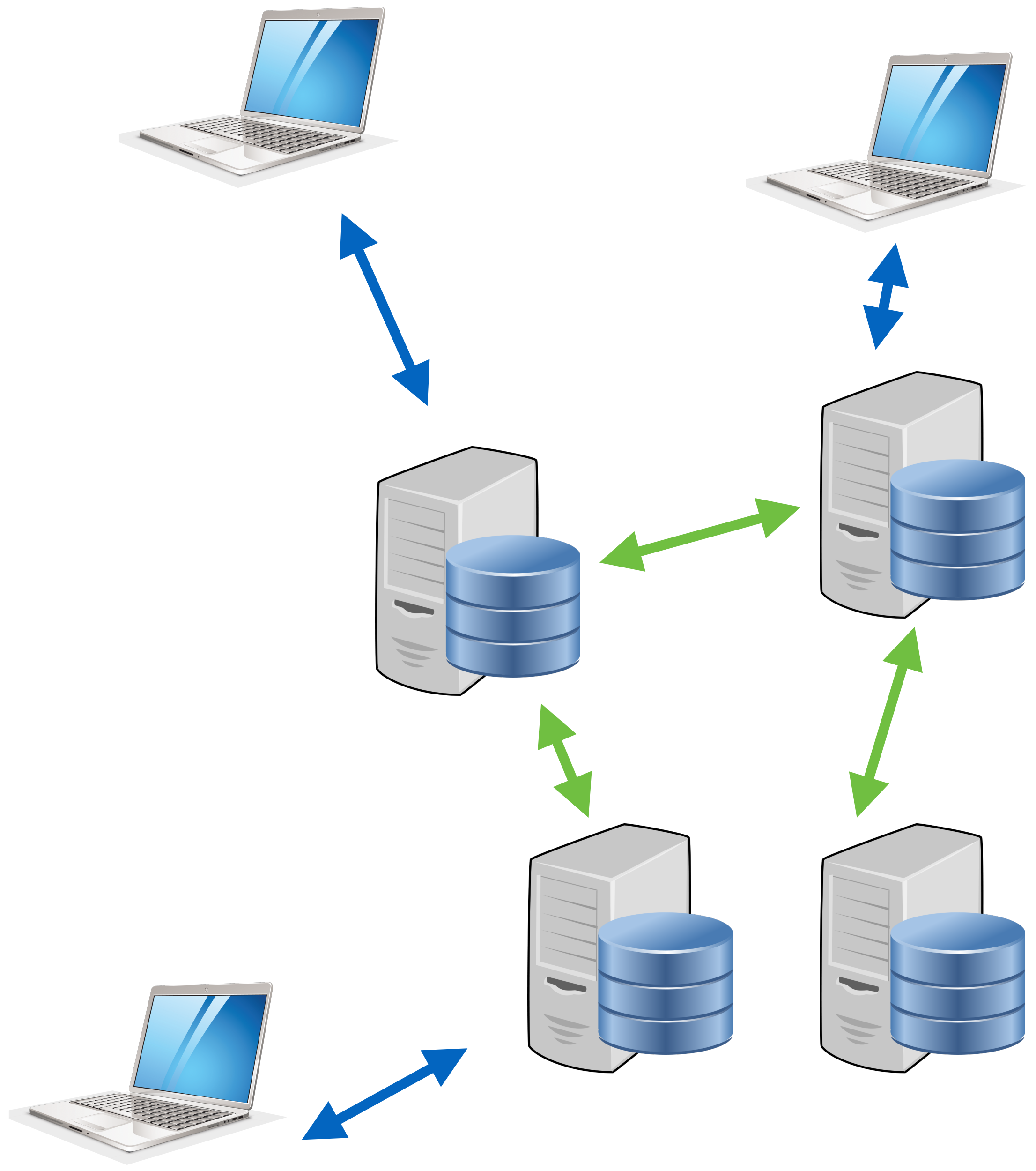
```
upon initialization do
    $primary \leftarrow p_1$                                   % Initial primary

upon receive ⟨NEWP⟩ from $p_2$ do
    $primary \leftarrow p_2$                                   % Backup

upon operation($r$) do
    while not received a reply do
        send ⟨REQ, $r$⟩ to $primary$
        wait receive ⟨REP, $v$⟩ or receive ⟨NEWP⟩
    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.