DATA DISTRIBUTION

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)

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
99,999%	five nines	about 5 minutes
99.9999%	six nines	about 31 seconds

Examples

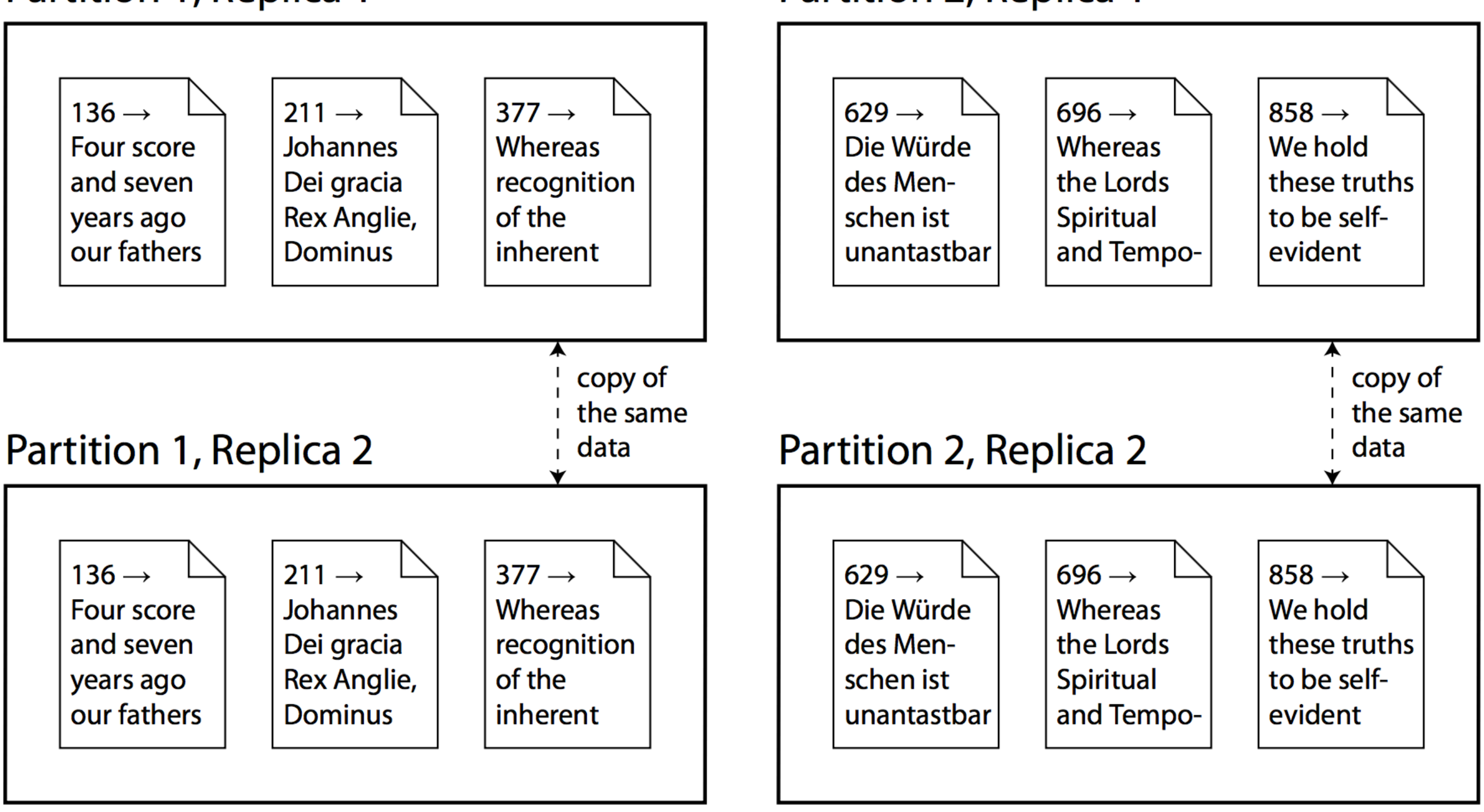
- One single server
- On a
  - Me
  - Mean Time To Restart (MTTR) 2 mins
- Two r
- 10 se
- MTB
- All se

mins

$$A = 10800/10802 = 0.\underline{999}8$$

$$p_f = 2/10802$$
$$A = (1-p_f)^{10} = 0.\underline{99}8$$

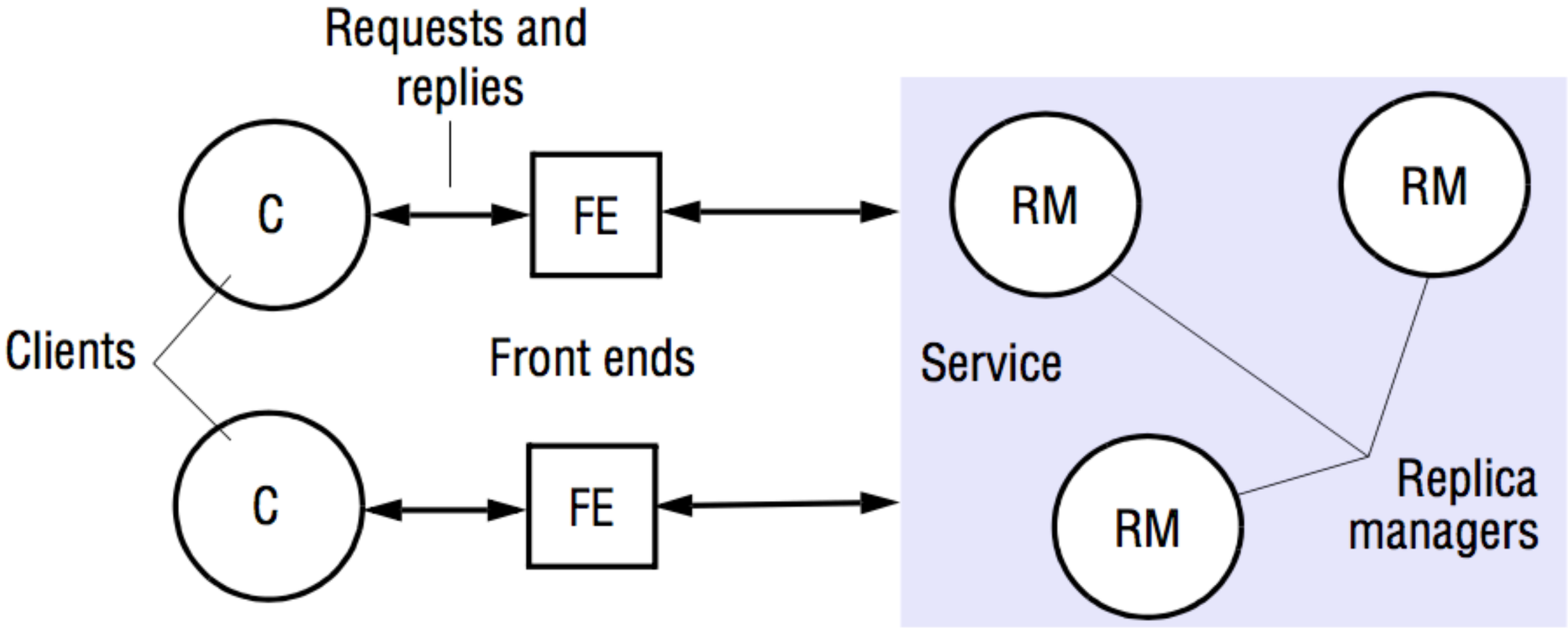
Replication & Partitioning



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
- Due to the nonzero latency of the messages, strict consistency is not implementable on distributed hardware.
- It is the model of uniprocessor systems!

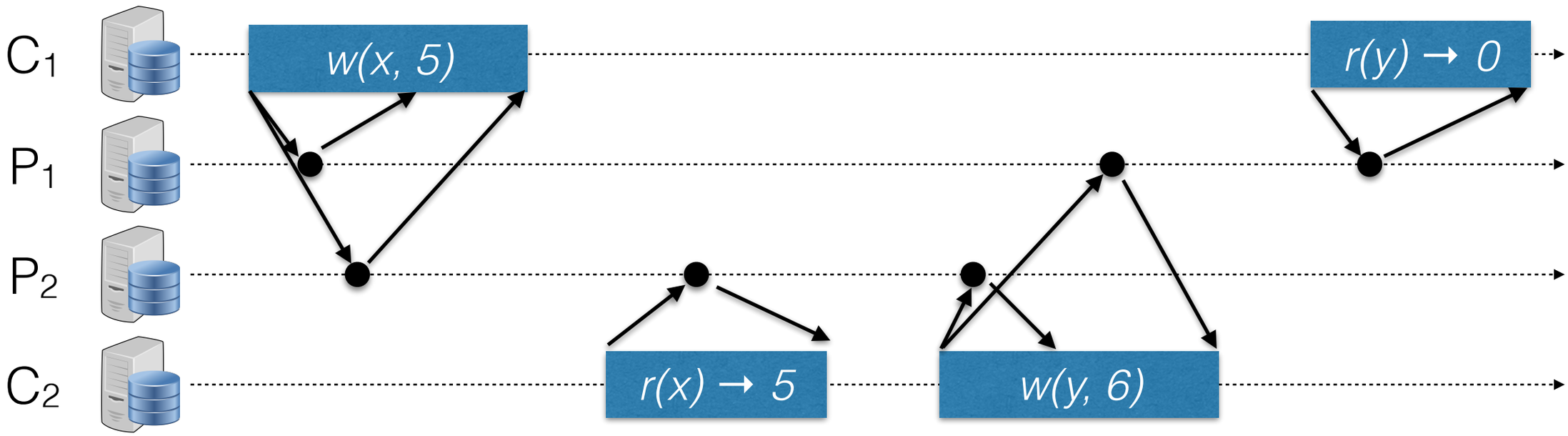
Linearizability

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



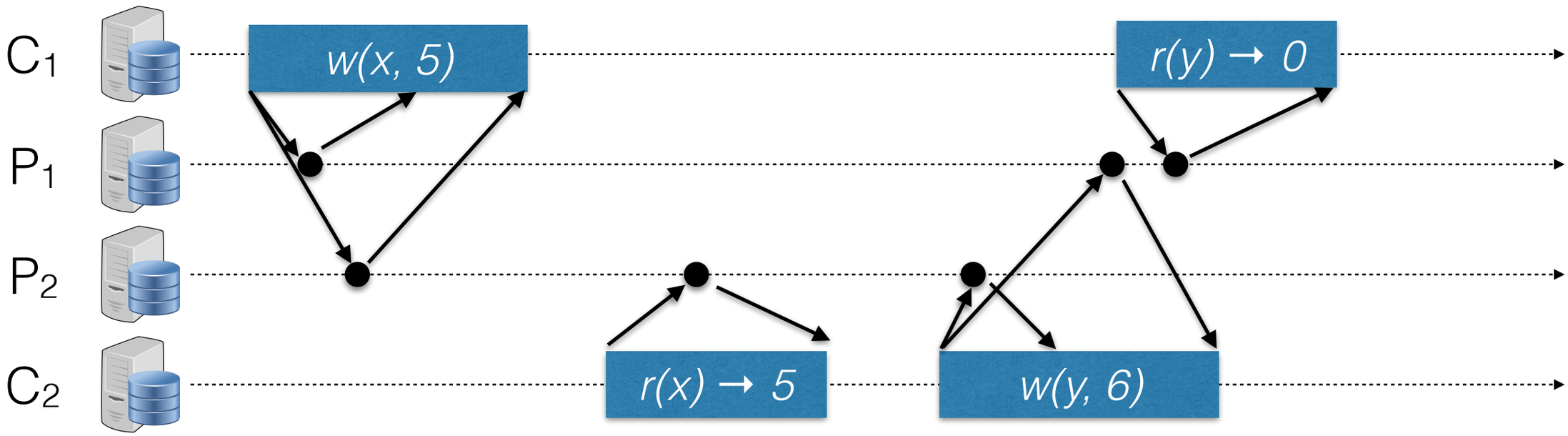
Are the following sequences possible linearizations?

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



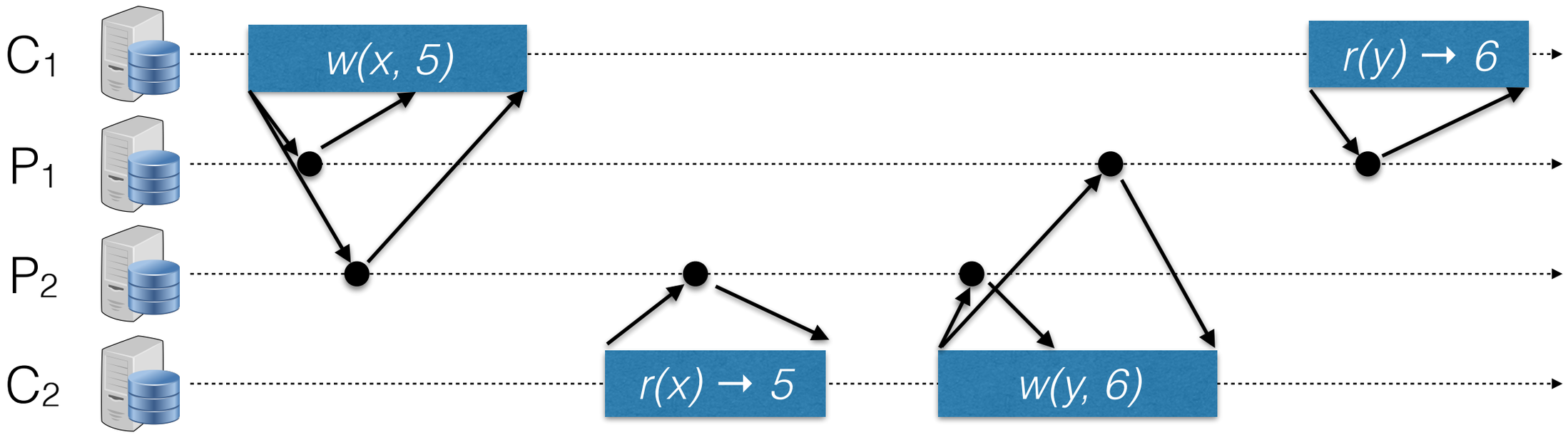
Are the following sequences possible linearizations?

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

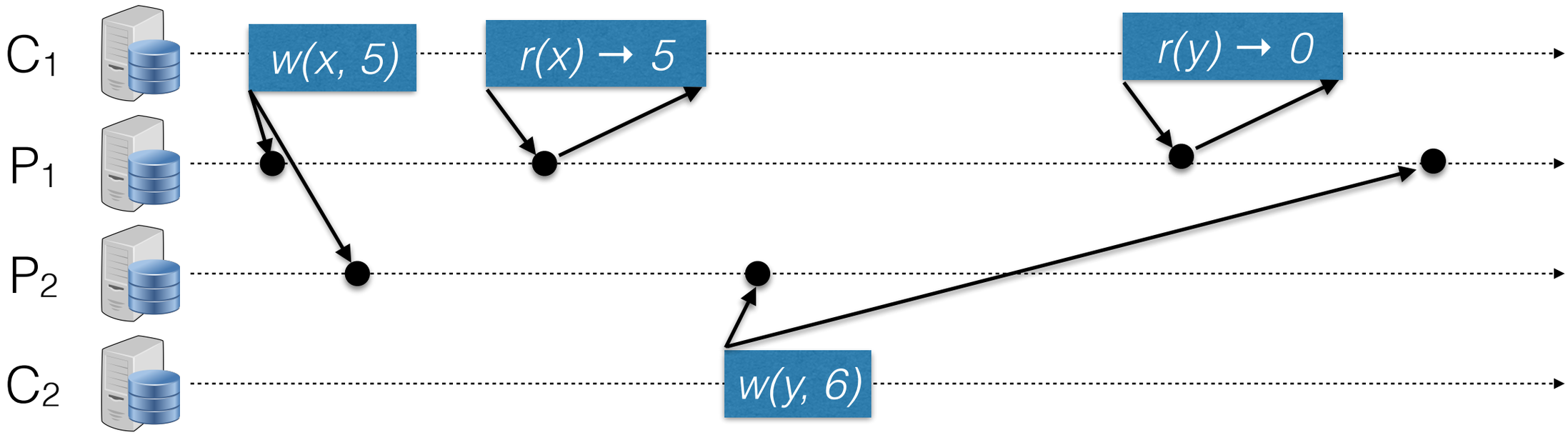
Sequential Consistency

Definition

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

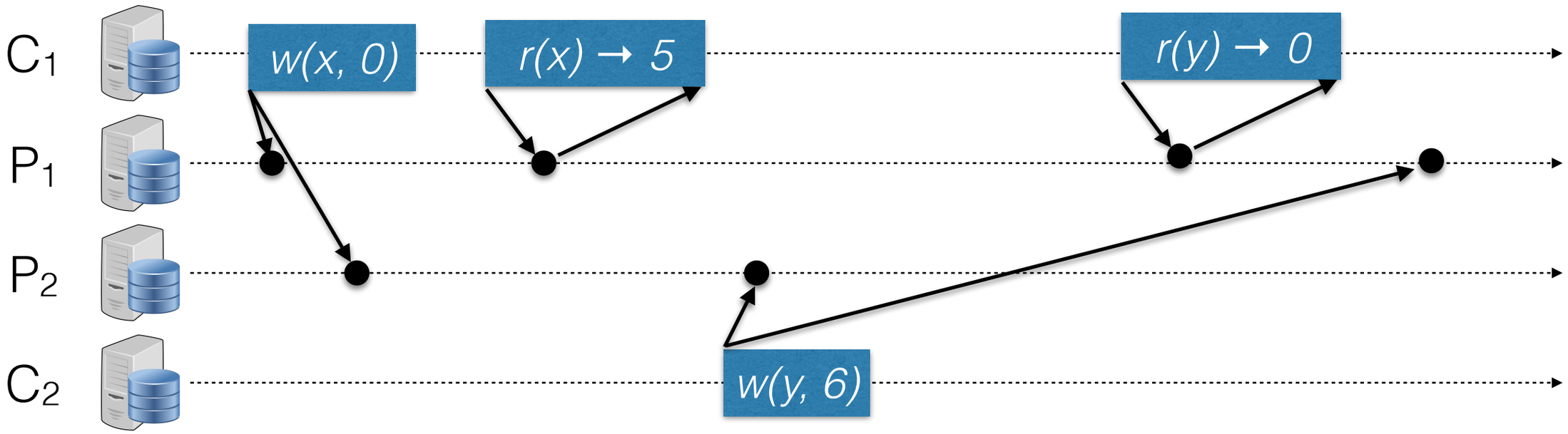
- H *contains exactly* the same operations that occur in E, each paired with the return value received in E
- H is a *legal history* of the sequential data type that is replicated
- The total order of operations in H is *compatible with* the **client partial order** <
  - $o_1 < o_2$ means that the o_1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

Example



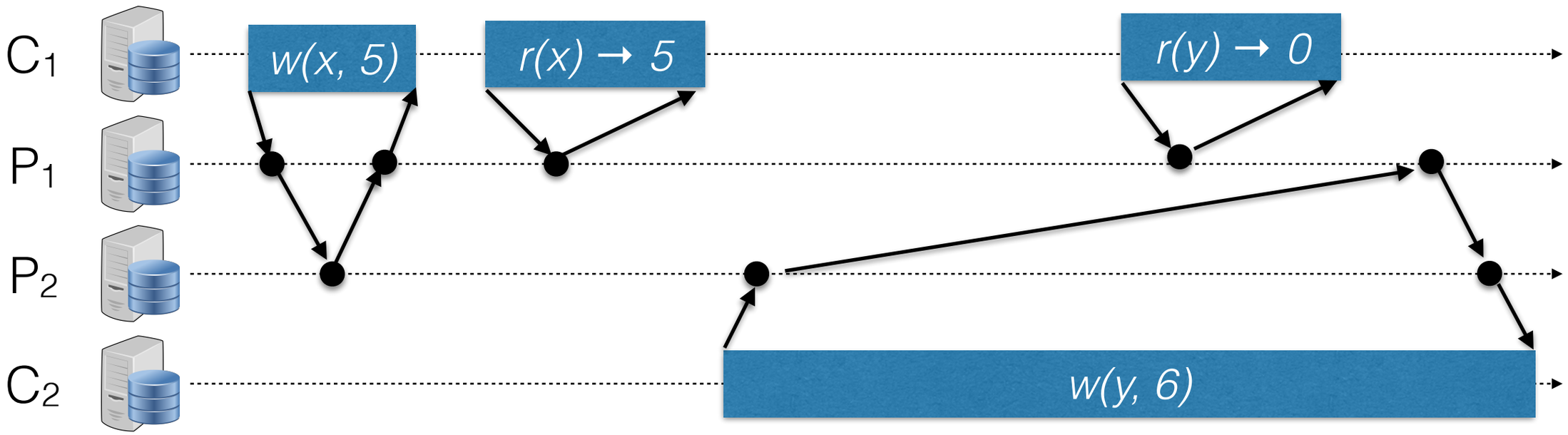
Is the execution above sequentially consistent? YES

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