# Replication & Availability

- Internet-scale services must be partition-tolerant
    - What if Amazon hangs if my client can not communicate with a server in India?
- Internet-scale services must be always available
    - What if Facebook hangs because busy propagating consist status updates?
- We must give up (strong) consistency for (weak) consistency
    - P + A + (s)C is impossible
    - P + A - (s)C is possible but useless
    - P + A + (w)C is possible
- References:
    - http://book.mixu.net/distsys
    - Bailis, Ghodsi, Eventual Consistency Today: Limitations, Extensions, and Beyond, ACM Queue, 2013
    - Shapiro, Saito, Optimistic Replication, ACM Computing Surveys, 2005. https://www.ece.cmu.edu/~ece845/sp06/docs/optimistic-data-rep.pdf

# Consistency

- A consistency model is a contract between programmer and system, wherein the system guarantees that if the programmer follows some specific rules, the results of operations on the data store will be predictable.

- Strong consistency models guarantee that the apparent order and visibility of updates is equivalent to a non-replicated system
    - Linearizable consistency
    - Sequential consistency
- Weak consistency models are not strong:
    - Client-centric consistency models
    - Causal consistency: strongest model available
    - Eventual consistency models

# Strong Consistency Models

- Linearizable consistency: Under linearizable consistency, all operations appear to have executed atomically in an order that is consistent with the global real-time ordering of operations. (Herlihy & Wing, 1991)

- Sequential consistency: Under sequential consistency, all operations appear to have executed atomically in some order that is consistent with the order seen at individual nodes and that is equal at all nodes. (Lamport, 1979)

# Weak Consistency Models

- Client-centric consistency models are consistency models that involve the notion of a client or session in some way.

- The eventual consistency model says that if you stop changing values, then after some undefined amount of time all replicas will agree on the same value.

- Since it is trivially satisfiable (liveness property only), it is useless without supplemental information.

# Giving up consistency, but…

- If I upload a document on my desktop computer, I want to download it sooner or later from my tablet

- If I upload a new version of the document on my desktop computer, and I downloaded it successfully on my tablet, I do not want to download the older version in subsequent downloads

- Sooner better than later!

# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives


x = 0, y = 0, z = 0

# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives


x = 0, y = 0, z = 0

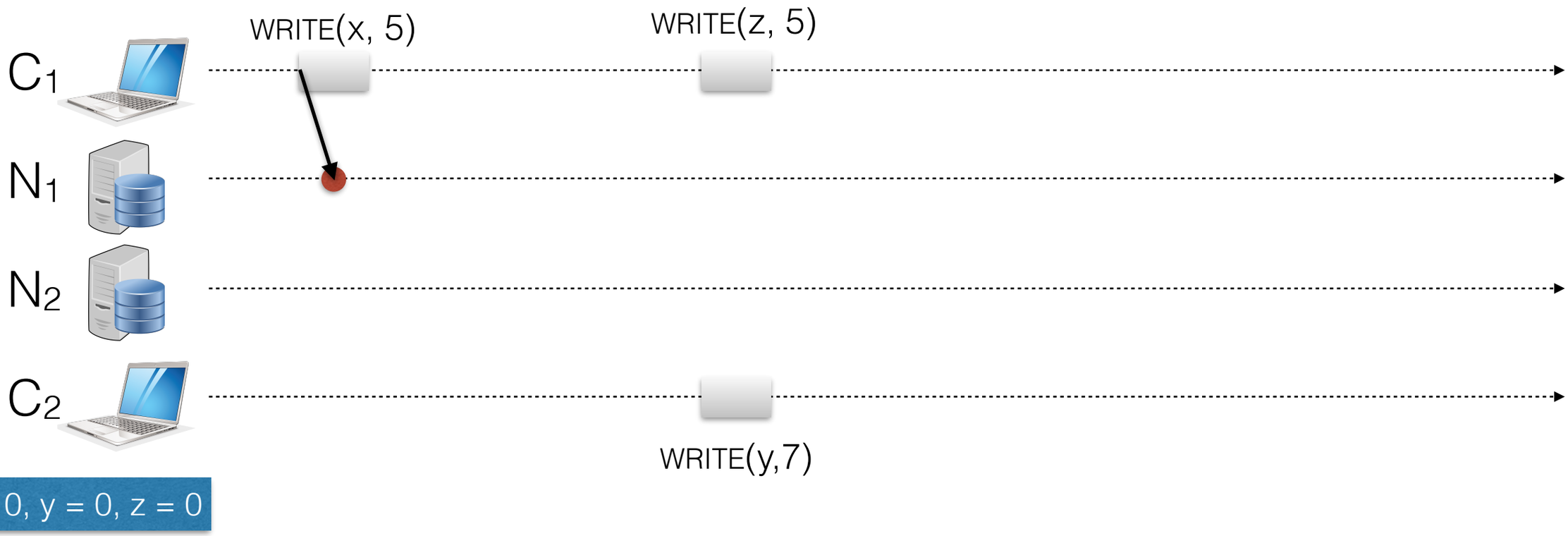
# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives


x = 0, y = 0, z = 0

# Weak Consistency Example

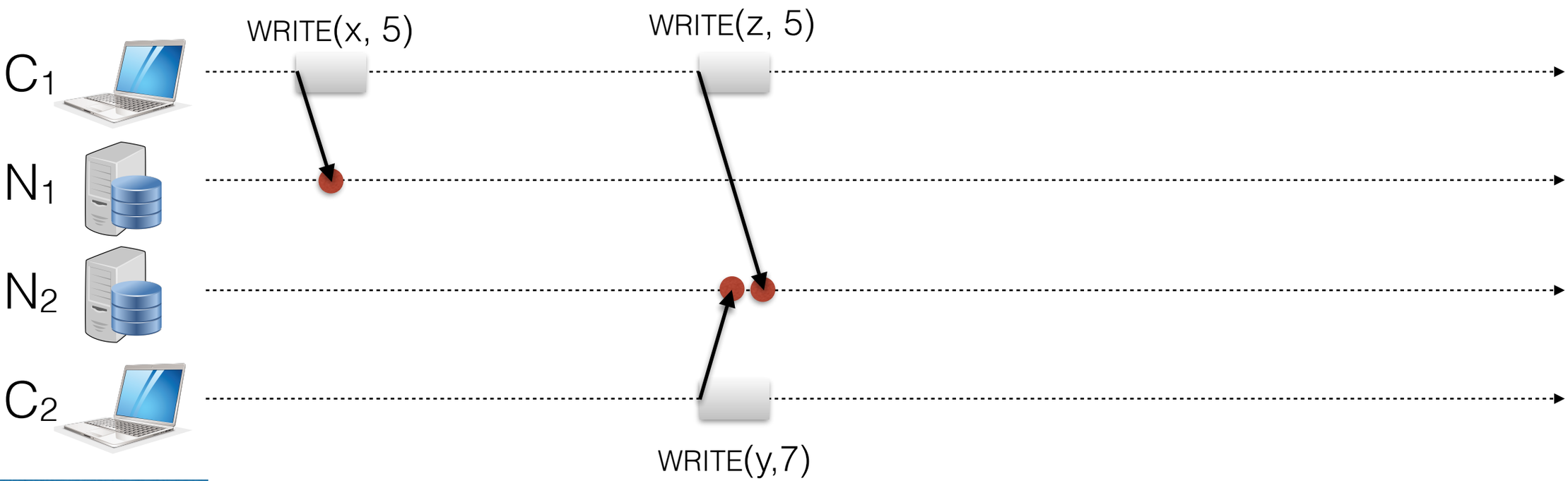
Each update is transmitted independently, and applied to each replica whenever it arrives




x = 0, y = 0, z = 0

# Weak Consistency Example

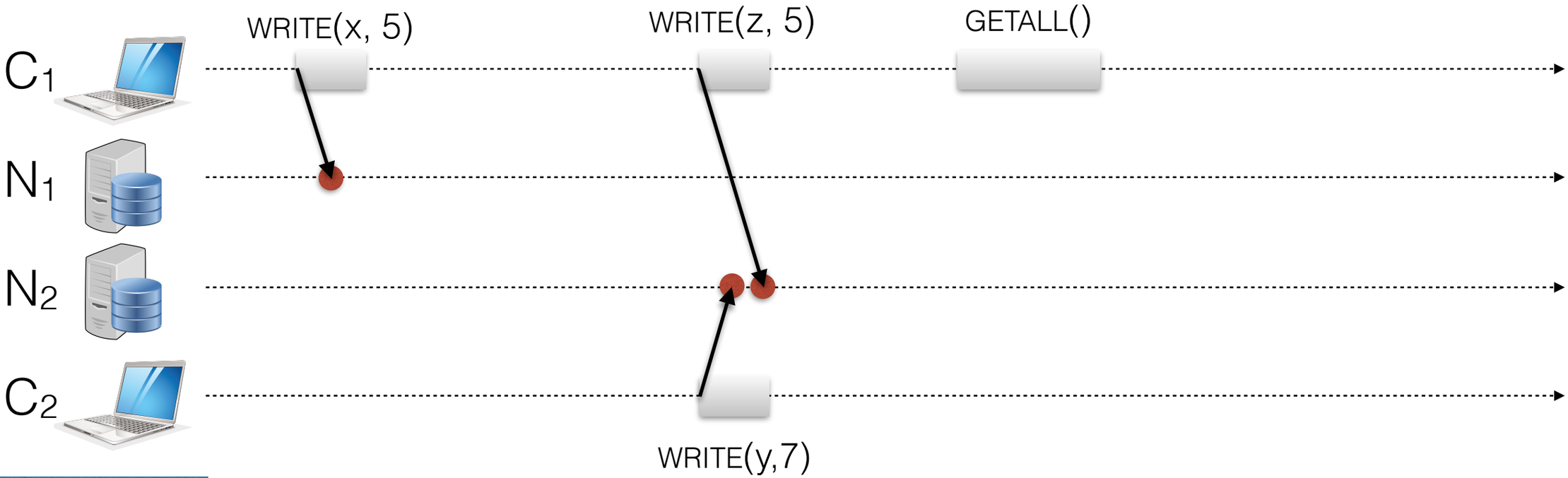
Each update is transmitted independently, and applied to each replica whenever it arrives




x = 0, y = 0, z = 0

# Weak Consistency Example

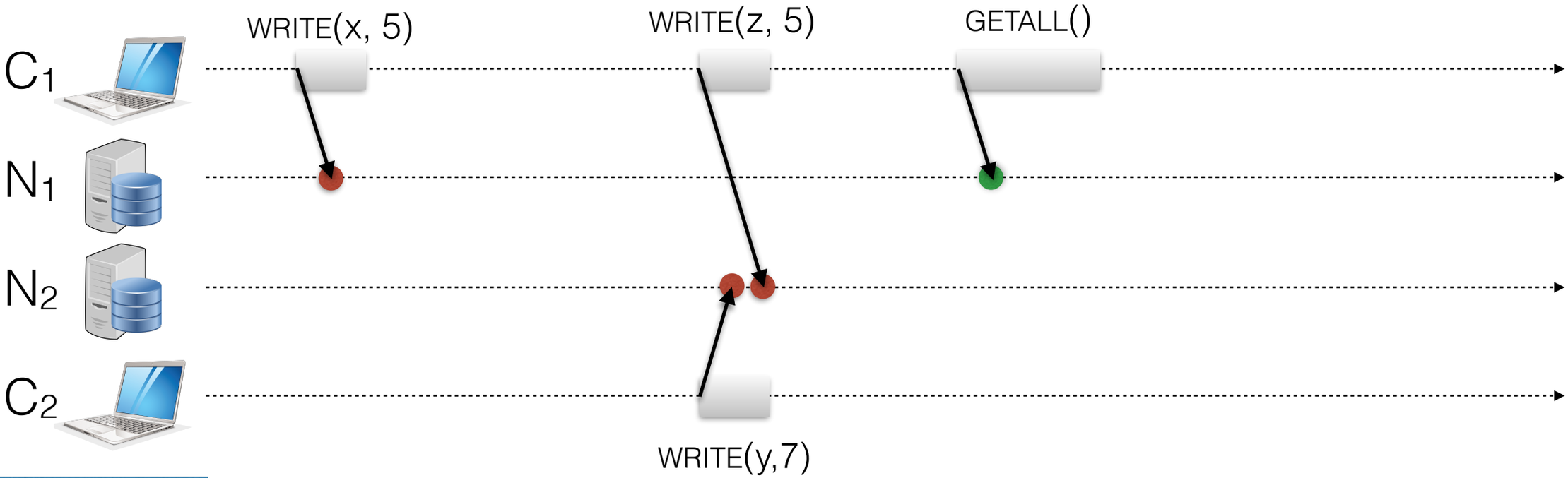
Each update is transmitted independently, and applied to each replica whenever it arrives




x = 0, y = 0, z = 0

# Weak Consistency Example

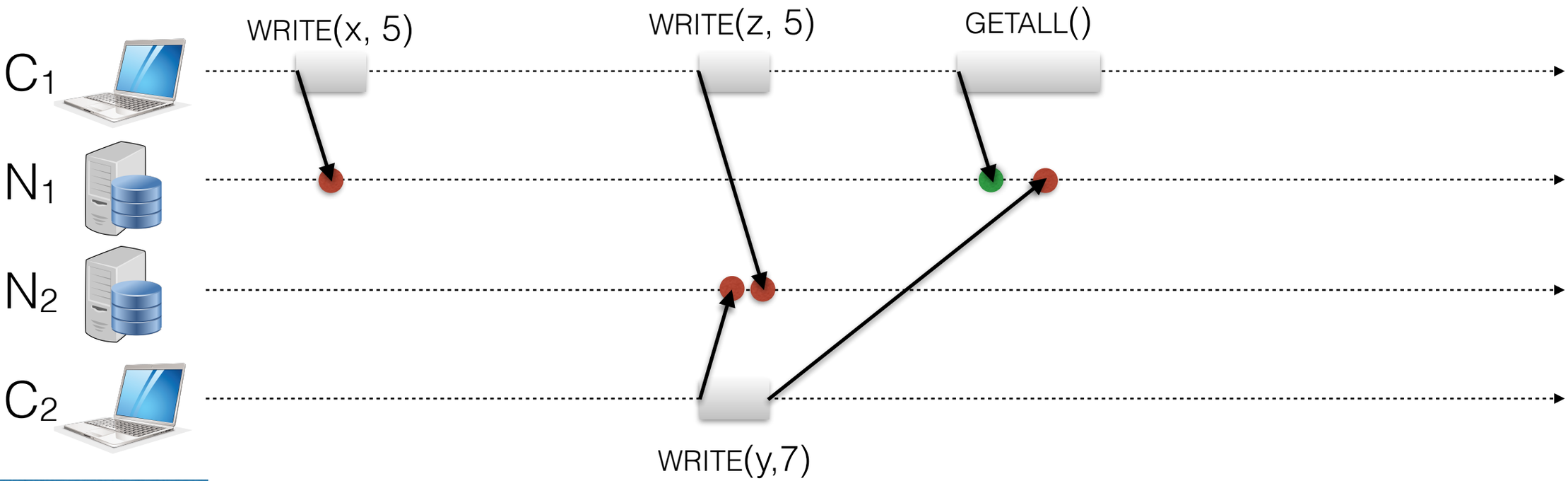
Each update is transmitted independently, and applied to each replica whenever it arrives




x = 0, y = 0, z = 0

# Weak Consistency Example

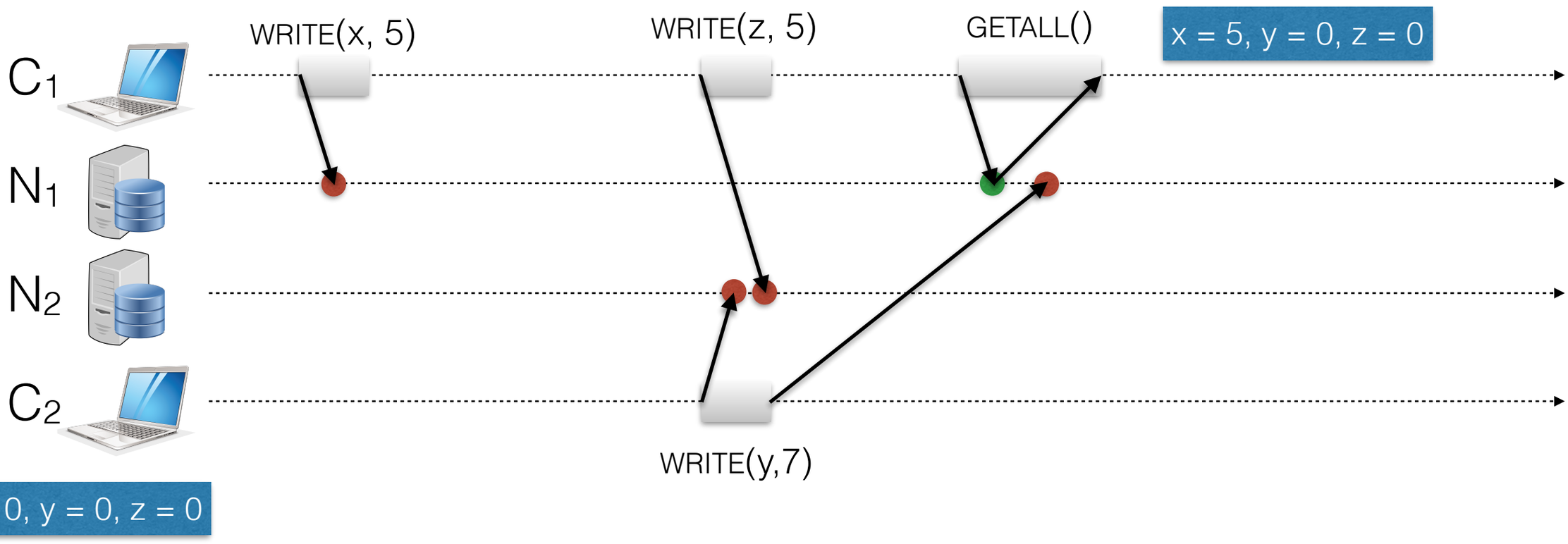
Each update is transmitted independently, and applied to each replica whenever it arrives




x = 0, y = 0, z = 0

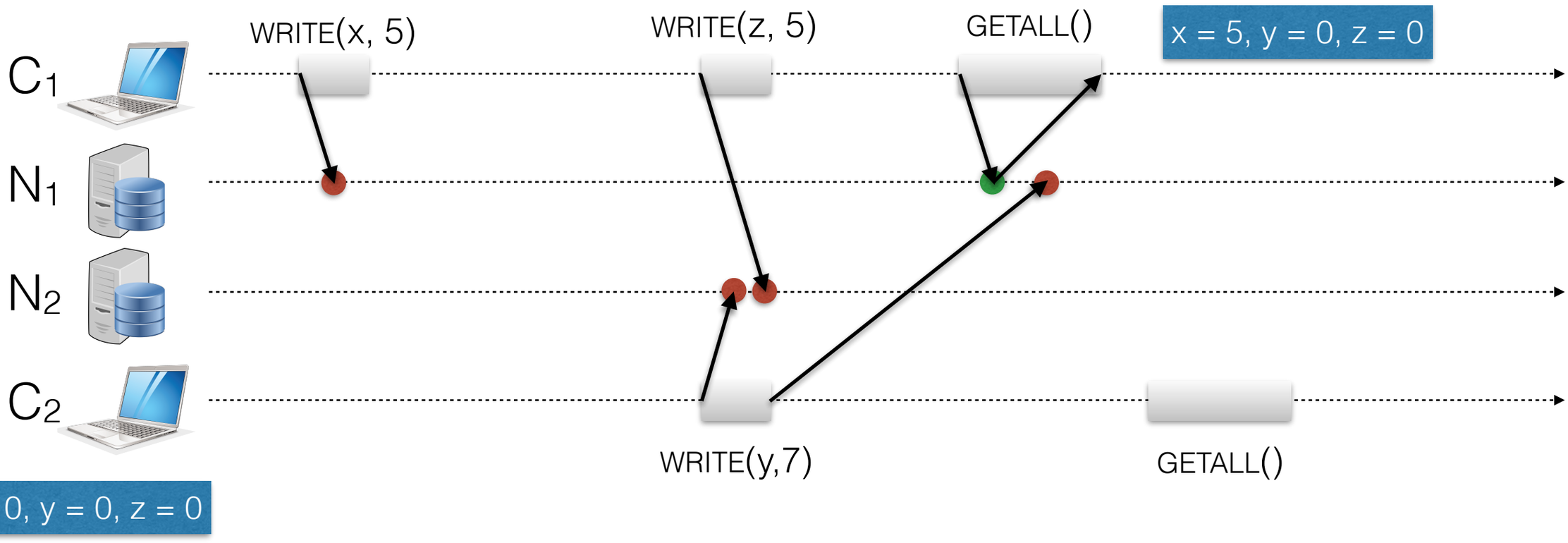
# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives



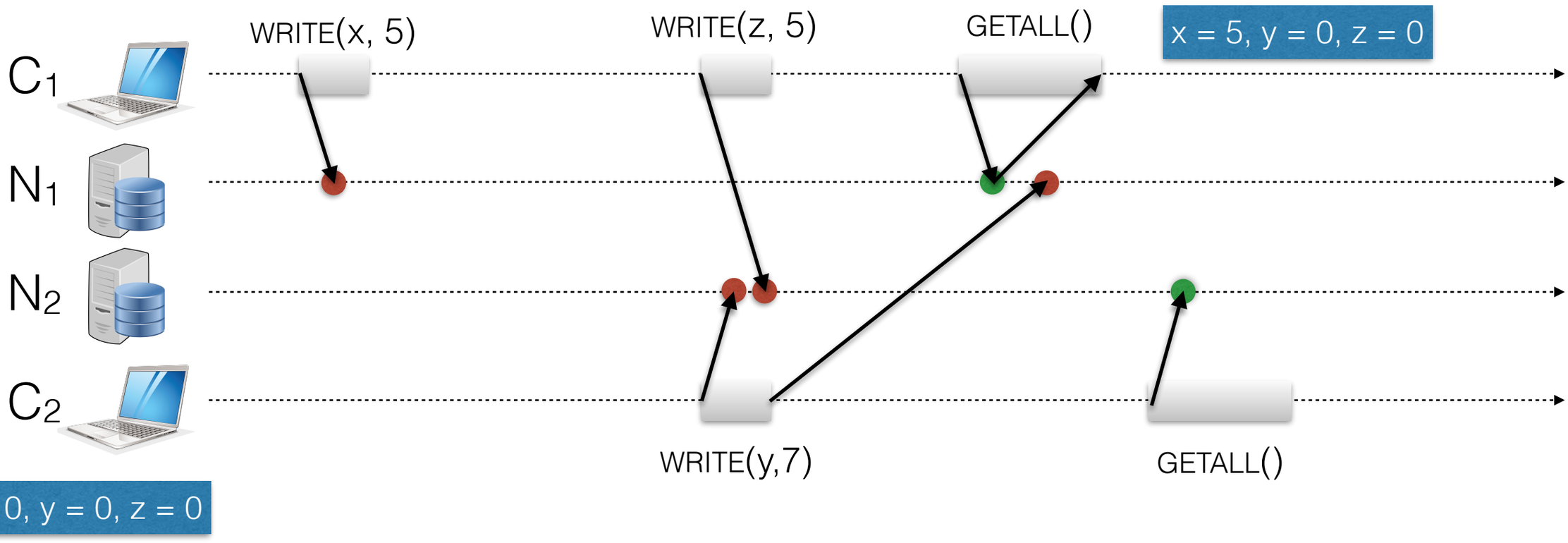
# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives



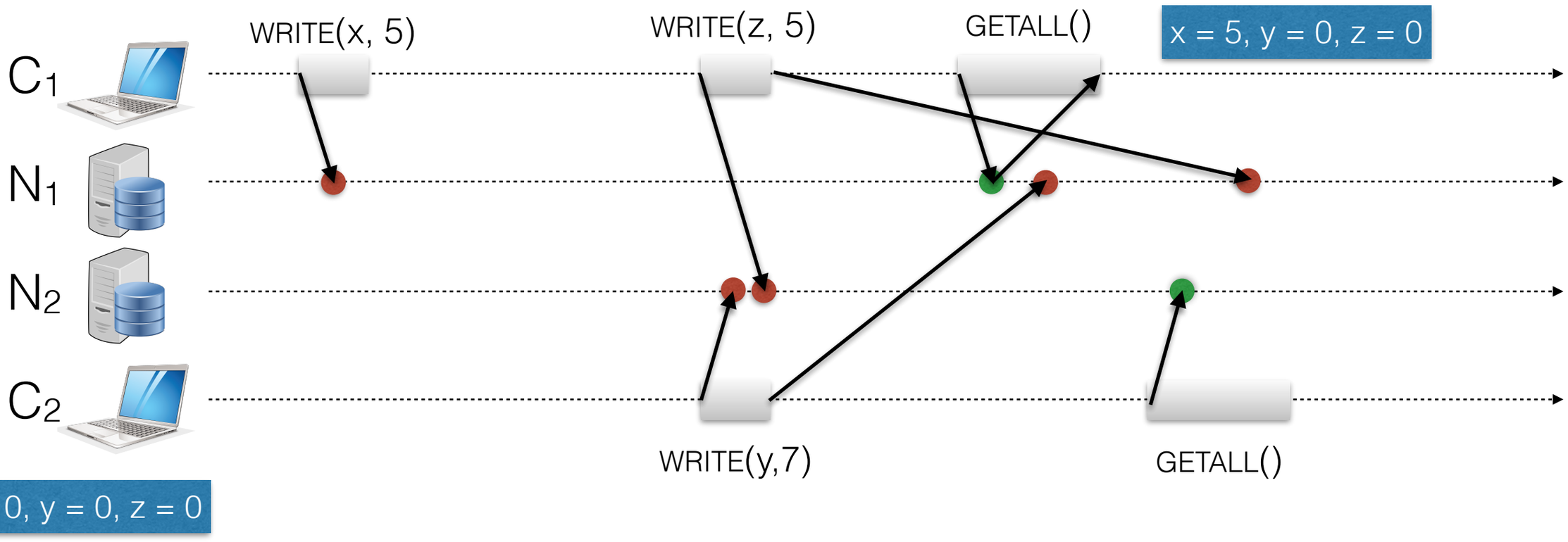
# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives



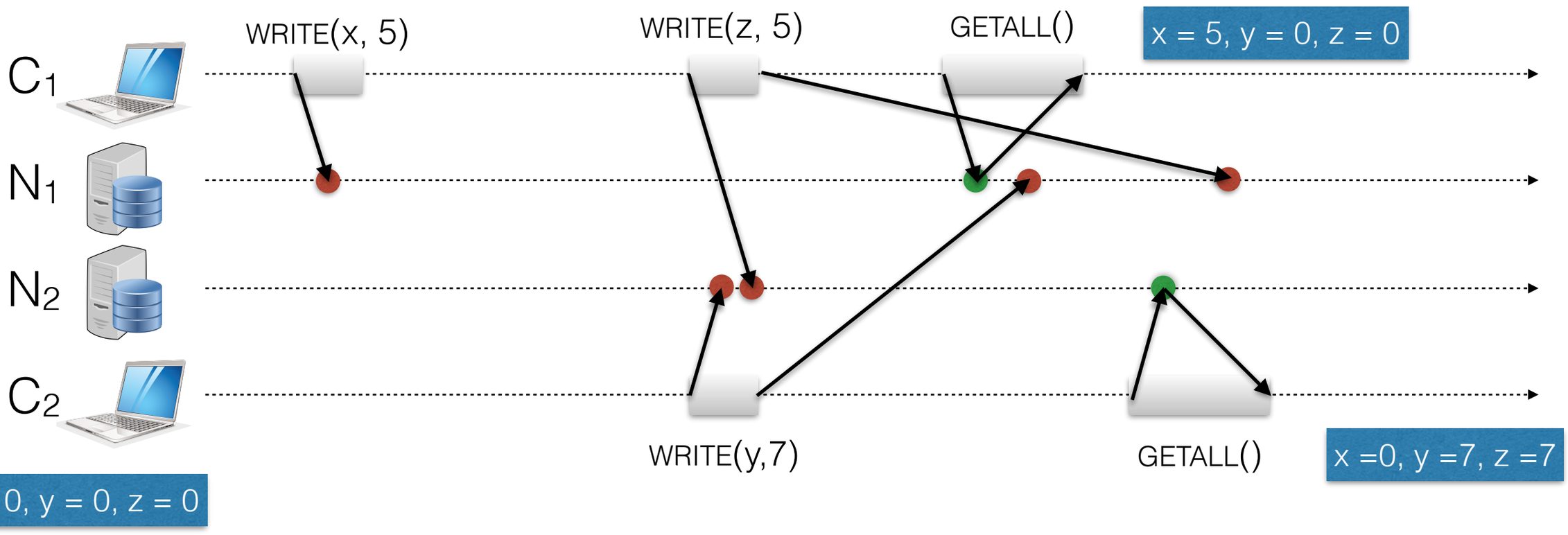
# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives



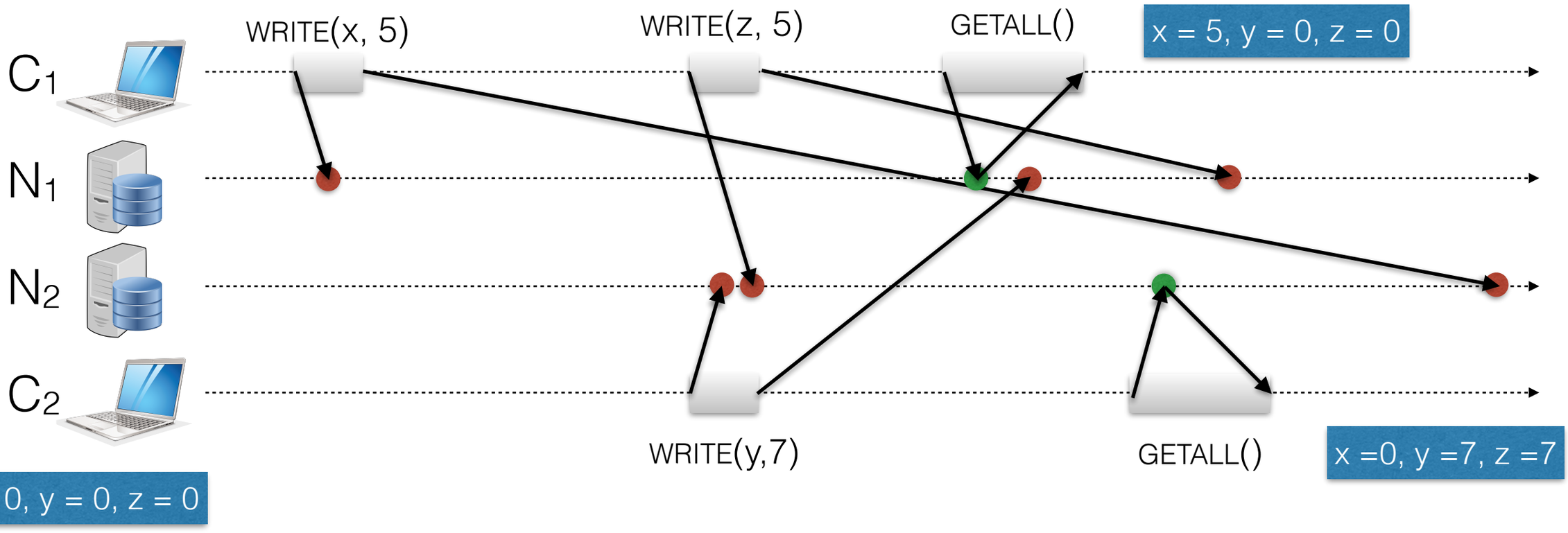
# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives



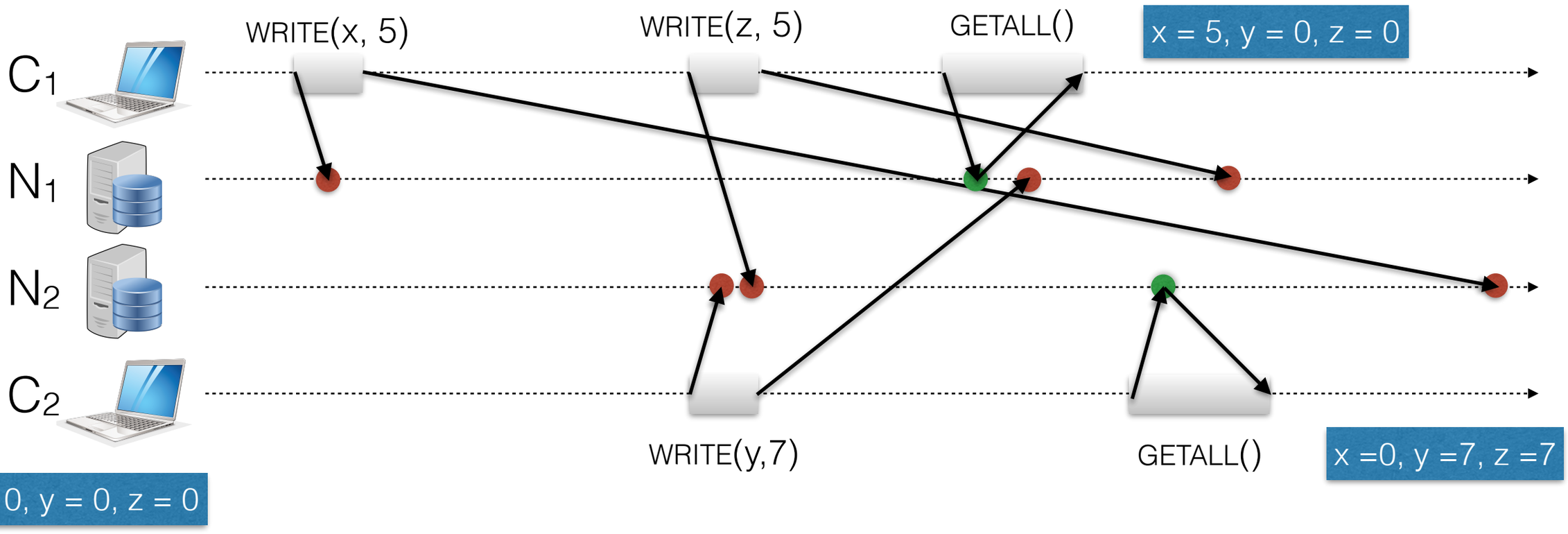
# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives



# Weak Consistency Example

Each update is transmitted independently, and applied to each replica whenever it arrives




This execution is not sequentially consistent

# Switching Perspective

- Problem: Sharing data in mobile computing scenario
  - A client connects with different replicas over time
  - Differences between replicas should be made transparent
    - No particular problems of simultaneous updates, here
- Client-centric consistency models
  - In essence, they ensure that whenever a client connects to a new replica, that replica is up to date according to the previous accesses of the client to the same data in the other replicas on different sites

# New Scenario

- Large, distributed data store with almost no update conflicts
- Typically, a single authority updating, with many processes simply reading
- The only conflict is read-write conflict where one process wants to update a data item while another concurrently attempts to read the same data
- Examples: DNS changes, Web content

Issues

- Non-updated data may be provided to readers
- In most cases, such an inconsistency might be acceptable to readers
- Typically, if no update takes place for a while, gradually all replicas will become consistent
- This sort of consistency is called eventual consistency

# Eventual Consistency

If no new updates are made to the data object, **eventually** all accesses will return the **last updated** value.

- Special form of weak consistency
- “Eventual” network partitions tolerated
- Requires “conflict resolution” mechanisms to disseminate write
- The message exchange is a.k.a. anti-entropy

# Monotonic Read

- A data store is said to provide monotonic-read consistency if the following condition holds

  If a process reads the value of a data item *x*, any successive read operation on *x* by the process will always return that same value or a more recent value

- Example: distributed email database

# Monotonic Write

- A data store is said to provide monotonic-write consistency if the following condition holds

  A write operation by a process on a data item *x* is completed before any successive operation on *x* by the same process

- The order of updates is maintained over distributed replicas

- Example: concurrent software development systems

# Read Your Writes

- A data store is said to provide read-your-writes consistency if the following condition holds

  The effect of a write operation by a process on data item *x* will always be seen by a successive read operation on *x* by the same process

- Avoid the “web page failed update” effect

- Example: password updates

# Writes Follow Reads

- A data store is said to provide writes-follow-reads consistency if the following condition holds

  A write operation by a process on data item *x* following a previous read operation on *x* by the same process is guaranteed to take place on the same or a more recent value of *x* that was read

- Writes affect only up-to-date data items

- Example: comments to posts on Facebook

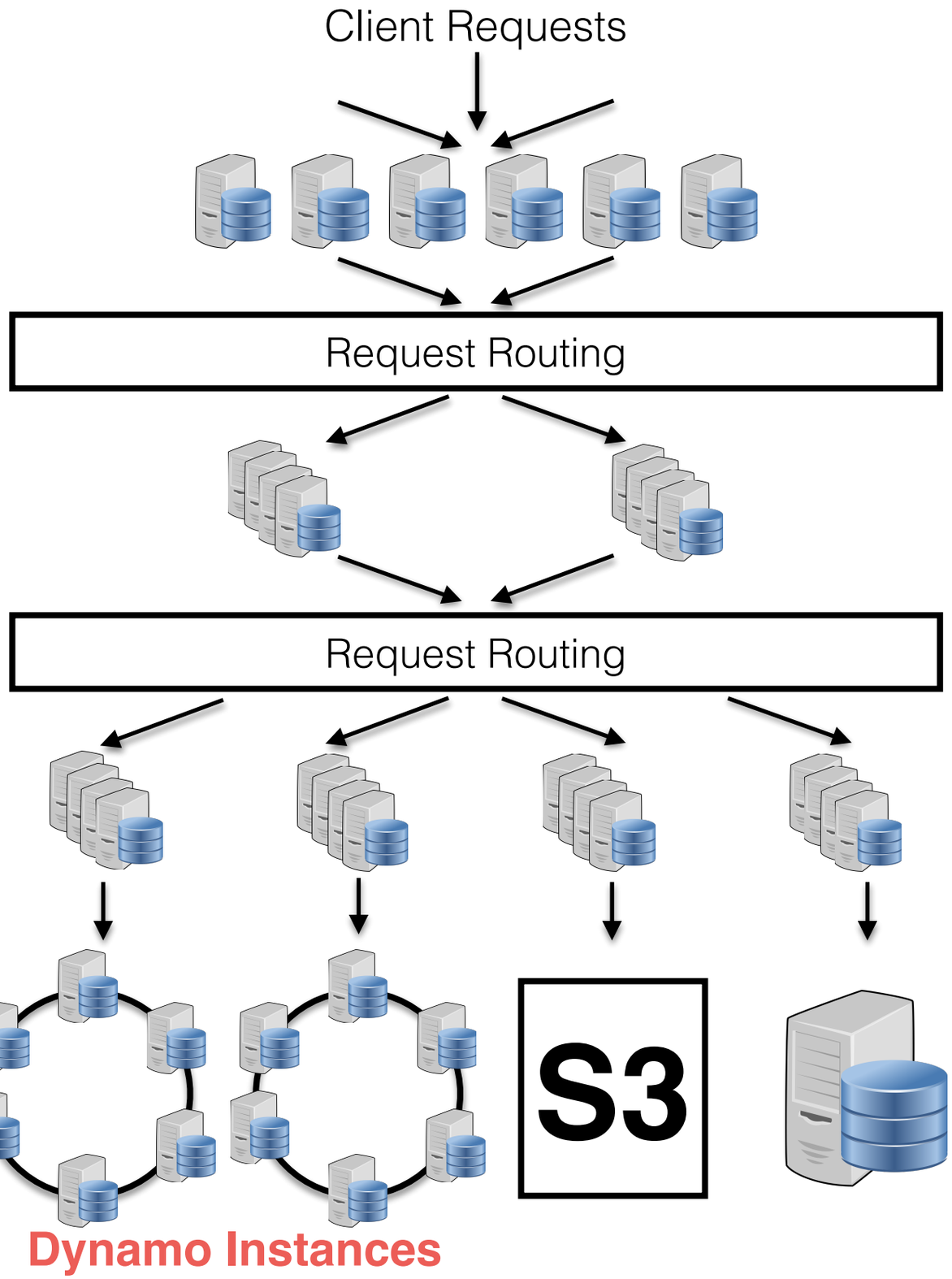
# Dynamo

- Service providing persistent data storage to other services running on Amazon's distributed computing infrastructure.

- Dynamo: Amazon's Highly Available Key-value Store, SOSP 2007

- First paper discussing implementation of weak consistent, highly available data replication service

# Amazon's Platform Architecture




**Page Rendering Services**

**Stateless Aggregator Services**

**Stateful Services**

**Datastores**

# Design Issues

- Replication for high availability and durability
  - replication technique: synchronous or asynchronous?
  - conflict resolution: when and who?
- Dynamo's goal is to be "always writable"
  - rejecting writes may result in poor Amazon customer experience
  - data store needs to be highly available for writes, e.g., accept writes during failures and allow write conversations without prior context
- Design choices
  - optimistic (asynchronous) replication for non-blocking writes
  - conflict resolution on read operation for high write throughput
  - conflict resolution by client for user-perceived consistency

# Design Principles

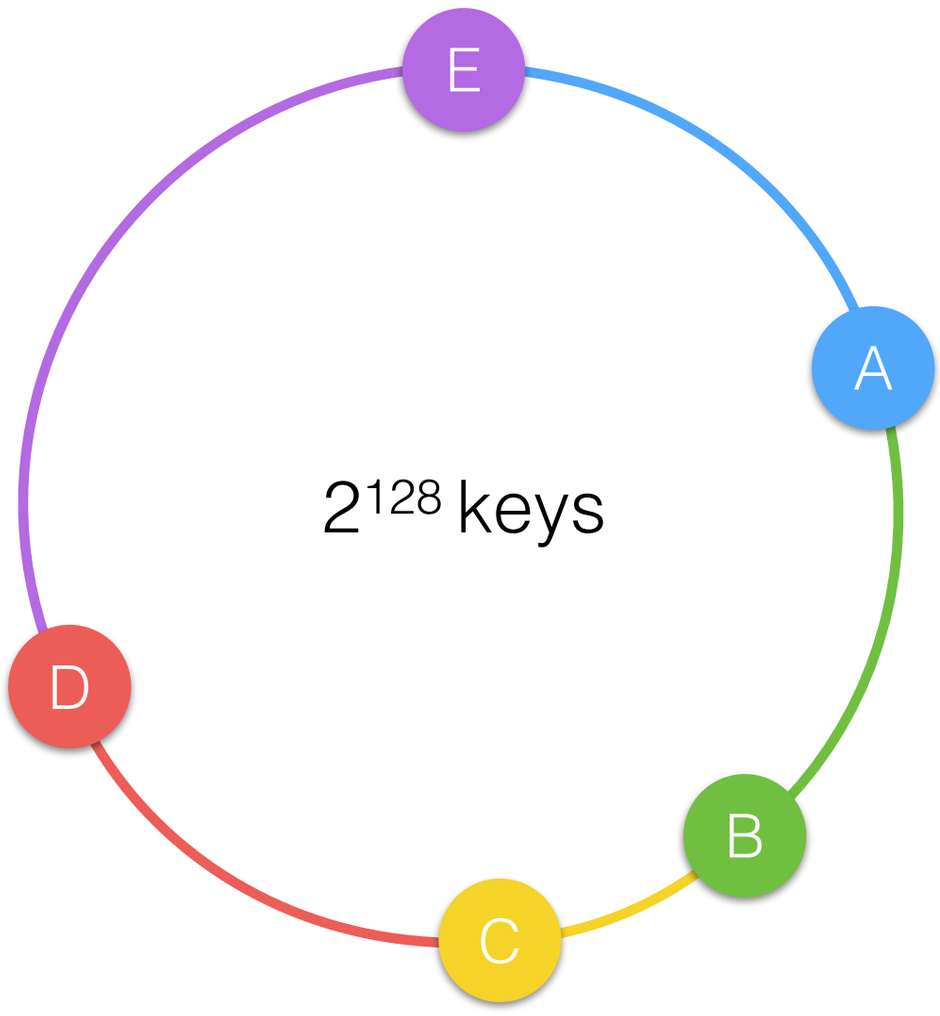
- Incremental Scalability
  - scale out (and in) one node at a time
  - minimal impact on both operators of the systems and the system itself
- Symmetry
  - every node should have the same set of responsibilities
  - no distinguished nodes or nodes that take on a special role
  - symmetry simplifies system provisioning and maintenance
- Decentralization
  - favor decentralized peer-to-peer techniques
  - achieve a simpler, more scalable, and more available system
- Heterogeneity
  - exploit heterogeneity in the underlying infrastructure
  - load distribution proportional to capabilities of individual servers
  - adding new nodes with higher capacity without upgrading all nodes

# Interfaces

- Data objects (blobs) are identified through an hash key
- Two single object operations, no group operations
- Read operation: GET(K) → [OBJECT(S), CONTEXT]
  - Locate object replicas associated with key k
  - Return single object or multiple conflicting version of the same object with context
- Write operation: PUT(K, OBJECT, CONTEXT)
  - Determine where the replicas should be placed based on key k
  - Write the replicas to disks
- Context
  - System metadata, opaque to the caller
  - Include information on object, such as versioning

# Partitioning

- Consisten Hashing
  - Load distribution
  - Arrival and departures have neighbor impact



# Replication

- Every data item is replicated at *N* hosts
  - *N* is configured “per-instance” of Dynamo
  - each key *k* is assigned a *coordinator host* that handles write requests for *k*
  - coordinator is also in charge of replication of data items within its range
  - Coordinator stores data item with key *k* locally
  - Coordinator stores data item at *N-1* clockwise successors nodes
  - Every node is responsible for the region of the ring between itself and its predecessor
  - *Preference list* enumerates nodes that are responsible for storing a key *k* – contains more than *N* nodes to account for node failures

# Data Versioning

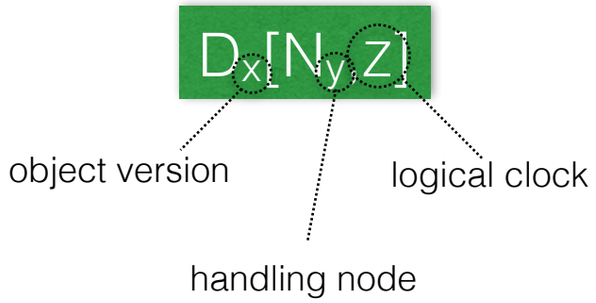
- Replication performed after a response is sent to a client
  - This is called asynchronous replication
  - May result in inconsistencies under network partitions
- But operations should not be lost
  - “add to cart” should not be rejected but also not forgotten
    - If “add to cart” is performed when latest version is not available it is performed on an older version
    - We may have different versions of a key/value pair
- Once a partition heals versions are merged
  - The goal is not to lose any “add to cart”
- Data versioning technique: Vector clocks
  - Capture causality between different versions of an object

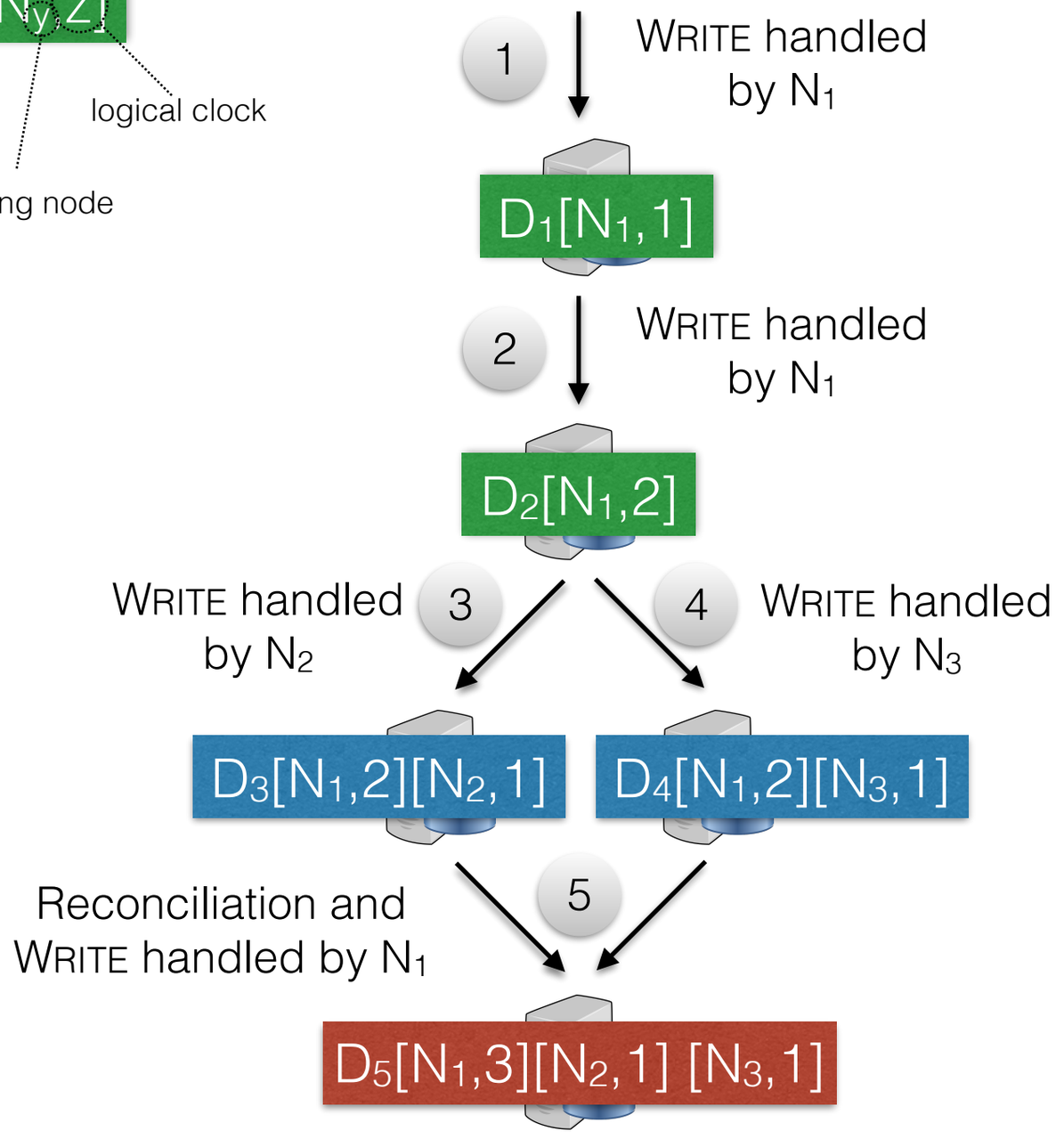
# Vector Clocks

- Each write to a key *k* is associated with a vector clock VC(k)
- VC(k) is an array (map) of integers
  - In theory: one entry VC(k)[i] for each node i
- When node i handles a write of key k it increments VC(k)[i]
  - VCs are included in the context of the put call
- In practice:
  - VC(k) will not have many entries (only nodes from the preference list should normally have entries), and
  - Dynamo truncates entries if more than a threshold (say 10)

# Data Versioning

1. Client A
   - writes new object D
   - node $N_1$ writes version $D_1$
2. Client A
   - updates object D
   - node $N_1$ writes version $D_2$
3. Client A
   - updates object D
   - node $N_2$ writes version $D_3$
4. Client B
   - reads and updates object D
   - node $N_3$ writes version $D_4$
5. Client C
   - reads object D (i.e., $D_3$ and $D_4$)
   - client performs reconciliation
   - node $N_1$ write version $D_5$



# Handling PUT/GET

- Any node can receive GET/PUT request for any key.
- This node is selected by
  - Generic load balancer
  - By a client library that immediately goes to coordinator nodes in a preference list
- If the request comes from the load balancer
  - Node serves the request only if in preference list
  - Otherwise, the node routes the request to the first node in preference list
- Each node has routing info to all other nodes
- Extended preference list: N nodes from preference list + some additional nodes (following the circle) to account for failures
  - Failure-free case: nodes from preference list are involved in get/put
  - Failures case: first N alive nodes from extended preference list are involved

# R and W

- R = number of nodes that need to participate in a GET
- W = number of nodes that need to participate in a PUT
- R + W > N (a quorum system)
- Handling PUT (by coordinator) // rough sketch
    1. Generate new VC, write new version locally
    2. Send value and VC to N selected nodes from preference list
    3. Wait for W -1
- Handling GET (by coordinator) // rough sketch
    1. Send read to N selected nodes from preference list
    2. Wait for R
    3. Select highest versions per VC, return all such versions
    4. Reconcile/merge different versions
    5. Write back reconciled version

# 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 Rules

- 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) = (1,N,N)
- Typical Dynamo (R,W,N) = (2,2,3)

# 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)

# Handling Failures

- N selected nodes are the first N healthy nodes
  - Might change from request to request
  - Hence these quorums are "Sloppy" quorums
- Sloppy vs. strict quorums
  - sloppy allow availability under a much wider range of
  - partitions (failures) but sacrifice consistency
- Also, important to handle failures of an entire data center
  - Power outages, cooling failures, network failures, disasters
  - Preference list accounts for this (nodes spread across data centers)
- Hinted Handoff
  - if a node is unreachable, a hinted replica is sent to next healthy node
  - nodes receive hinted replicas keep them in a *separate database*
  - hinted replica is delivered to original node when it recovers

# Replica Synchronization

- Using vector clocks, concurrent and out of date updates can be detected.
  - Performing read repair then becomes possible
  - In some cases (concurrent changes) we need to ask the client to pick a value.
- What about node rejoining?
  - Nodes rejoining the cluster after being partitioned
  - When a failed node is replaced or partially recovered
- Replica synchronization is used:
  - to bring nodes up to date after a failure
  - for periodically synchronizing replicas with each other

# Replica Reconciliation

- A node holding data received via “hinted handoff” may crash before it can pass data to unavailable node in preference list
- Need another way to ensure each (k, v) pair replicated N times
- Nodes nearby on ring periodically compare the (k, v) pairs they hold, and copy any they are missing that are held by the other

# Merkle Trees

- Idea: hierarchically summarize the (k, v) pairs a node holds by ranges of keys
- Leaf node: hash of one (k, v) pair
- Internal node: hash of concatenation of children
- Compare roots; if match, done
- If don’t match, compare children; recur…
- Benefits
  - minimize the amount of data to be transferred for synch
  - reduce the number of required disk reads

# Anti Entropy in Dynamo

- Dynamo uses Merkle trees for anti-entropy as follows
  - each node maintains a separate Merkle tree for each key range it hosts
  - two nodes exchange the root of the Merkle tree of the key ranges that they host in common
  - using the synch mechanism described before, the nodes determine if they have any differences and perform the appropriate synch

# Techniques used in Dynamo

| Problem | Technique | Advantage |
|---|---|---|
| Partitioning | Consistent hashing | Incremental scalability |
| High availability for writes | Vector clocks with reconciliation during reads | Version size is decoupled from update rates |
| Handling temporary failures | Sloppy quorum and hinted handoff | Provides high availability and durability guarantee when some of the replicas are not available |
| Recovering from permanent failures | Antil-entropy using Merkle trees | Synchronizes divergent replicas in the background |
| Membership and failure deteclon | Gossip-based membership protocol and failure deteclon | Preserves symmetry and avoids having a centralized registry for storing membership and node liveliness informalon |