Consistency & Replication

Agenda

- Introduction
  - Partitioning versus replication
  - Quality drivers; replication, partitioning
  - Replication transparency
  - Architectural concerns
  - Basic architecture
- Consistency models
- Replica management
- Architectural case studies

Partitioning versus replication

Partitioning

- Splitting the data into smaller subsets (called partitions) that are distributed over multiple nodes
- Also called sharding

Replication

- Distributing multiple copies (replicas) of the same data over distinct nodes
- Redundancy

Often combined

- Multiple replicas per partition

Quality drivers for partitioning

Introducing partitioning into the architecture of a data store is mainly motivated by the following quality drivers

- Size of the data set (data base) simply too large for a single node
  - What will be partitioned? Tables, records, ...
- Scalability
  - Partitioning allows load balancing of data and its access
  - Works best for NO-SQL data stores, e.g., key-value data stores
    - each partition contains a range of records
- Performance
  - Throughput improvement by concurrent access of distinct partitions
    - requires proper allocation of records to partitions!
      - avoid hot spots that receive the bulk of the queries (skewed workload)
      - or combine with replication

Quality drivers for replication

Introducing replication into the architecture of a data store is mainly motivated by the following quality drivers

- Reliability
  - Replication removes a single point of failure and allows consensus protocols to deal with corrupted data (majority voting, etc.)
- Availability
  - If the probability that a single server becomes inaccessible is p, then the availability with n servers is $(1 - p^n)$ (assuming server failures are independent, which need not be the case in, e.g., datacenters)
- Performance
  - Throughput improvement by concurrent access of distinct replicas
  - Latency reduction by accessing a nearby replica
  - Bandwidth reduction due to increased data proximity
- Scalability
  - Replication (mostly of compute resources) allows load balancing

Replication transparency

- An architecture is *replication transparent* when the users of the system are unaware of the fact that several *replicas* (physical copies) of an object (resource) exist.
  - Refers primarily to data values; services that are implemented via multiple servers or multi-threaded servers are (by intention) distinguishable through increased performance.
  - Implies that clients identify only a single (logical) data object as the target of an operation and also expect only a single return value (as opposed to one for each replica on which the operation is performed).
  - Implies that the architecture is also *location transparent*, otherwise replicas could be identified by their location.
    - e.g., as revealed in their name.

Architectural concerns

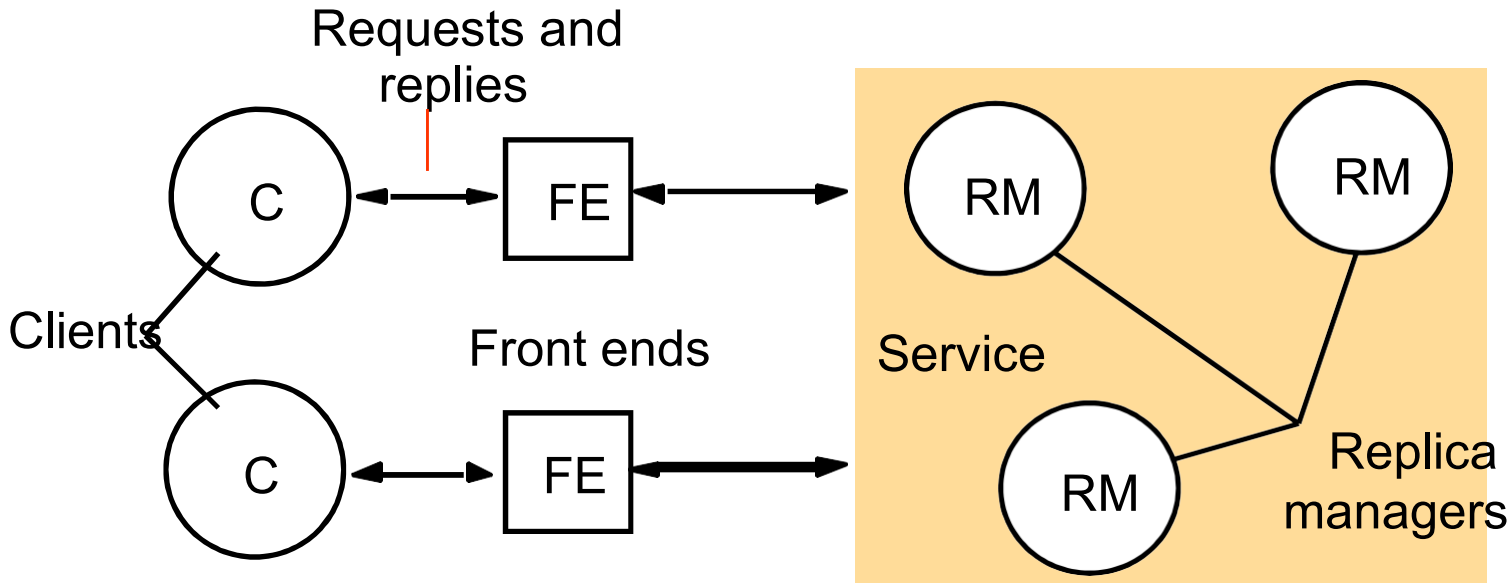
Replication (especially of data) is not for free: it raises additional concerns whose solutions incur costs:

1. What is the number of replicas and where should the replicas and replica managers be located?
   - Statically or dynamically resolved. In the latter case, also which party initiates the creation/destruction of replicas?
2. Whether and how to maintain consistency?
   - Ideally, all replicas have the same value (at least upon access)
   - Difficult because there is no notion of global time and state.
3. What architectural elements to use for storage and management?
   - Front ends, replica managers, caches, key-value stores, relational databases, load balancers, multicast infrastructure, logical clocks for time-stamping and versioning
4. What protocols to use for accessing (reading & writing) of replicas?
   - Push(server)-based or pull(client)-based, unicast versus multicast, group view, gossiping, degree of synchronization

Basic model (CDK5, Wiesmann et al.)

- The basic model is multiple-client-single-server or multiple-client-multiple-server.
  - Servers represent a distributed data store with one or more access points for their clients
  - Operations invoked on the objects in the store are classified in two broad categories
    - Updates (also referred to as *write operations*) that *potentially* modify the state of the store
    - Queries, (also referred to as *read operations*) that inspect part of the state of the store.
  - Each server has a special entity, the *replica manager (RM)*, that is responsible for managing the local part of the data store.
    - Depending on the details of the architecture, this entity also serves as access point for clients. Alternatively, this task is performed by a separate front end (FE).

Figure 15.1
A basic architectural model for the management of replicated data



- RMs manage replicas of multiple objects, and each object is managed by a subset of all RMs. The size of the subset determines the number of replicas of the object. Often, each RM manages every object.
- RMs apply operations to their replicas as indivisible actions, and all actions can be recovered.
- The state of an RM is completely determined by its initial configuration and the sequence of actions performed.

Basic architecture: behavioral view

1. Request phase
   - Requests accepted by FE are communicated to a fixed RM (passive)
   - Requests accepted by FE are multi-casted to all RMs (active)
2. Coordination phase
   - RMs determine whether, by which RMs and in which order (FIFO, causal, total) requests are performed; ordering can be enforced through delivery mechanism (usually total order)
3. Execution phase
   - RMs tentatively, i.e. undo is possible, execute the requested operation
4. Agreement phase
   - RMs reach consensus on the effect of the operation and commit
5. Response phase
   - Some RMs respond to the FE which in turn replies to the client

Beware: Phases need not be executed in this order

Agenda

- Introduction
- Consistency models
  - Single-server paradigm
  - Conflicting operations
  - Data-centric models
  - Client-centric models
- Replica management
- Architectural case studies

Consistency models

1. Are contracts between data store and clients
   - Consistency is a property of the data store as a whole; for individual data items we speak about coherence for which there can be separate models.
2. Define the unit of consistency (the conit)
3. Determine the outcome of a sequence of read/write operations performed by one or more clients
   - Results obtained by individual clients
   - Resulting state of the store
4. Can be classified in two broad categories:
   - Data-centric models
   - Client-centric models

Single-server paradigm

- The key idea behind consistency is that, for all parties involved, the operations appear *as if they were performed as indivisible actions by a single server*, i.e., in the *same order* and having the *same effect.*
  - *"the same effect"* requires that queries return the same value and updates leave the data store in the same state.
    - This state is a logical concept, because, in practice, it can occur that there is never a moment in time at which all replicas hold values in accordance with the state. However, if clients cease to access the store and no RM crashes, all replicas eventually must assume the same value.
  - *"the same order"* requires the existence of a global, system wide, notion of time, that is used to totally order the receipt of operation requests by the data store.
    - In practice, synchronization of local clocks can only approximate this ideal, which may or may not be sufficient (beware True Time in Google Spanner ☺)
- Since neither demand can be fulfilled, consistency models basically delineate how far the system may deviate from this ideal.

Conflicting operations

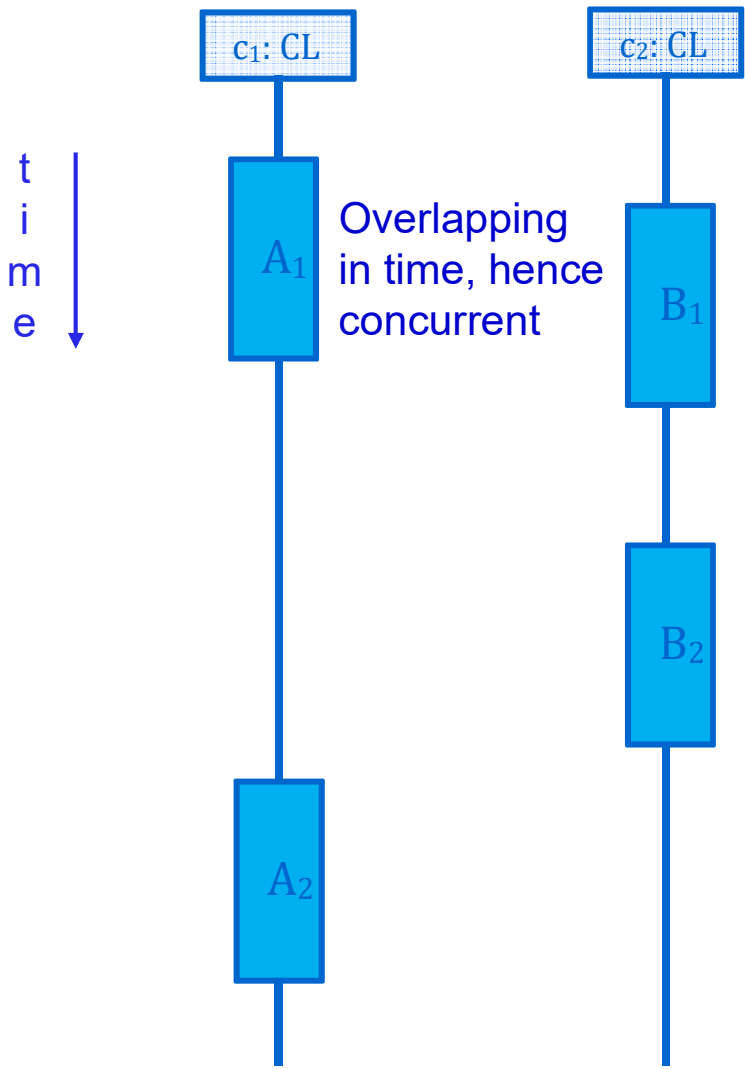
- Two operations are *conflicting* when the outcome of executing them as a sequence of two atomic actions may *potentially* differ for the two possible execution orderings.
- Conflicting operations come in two flavors:
  - read-write conflicts (different values returned)
  - write-write conflicts (store left in different states)
- The actual values involved in write operations can be such that no conflict arises.
  - Example: Client1::Write (X, 42) Client2::Write (X,42)
  - To limit the amount of state information, replication management protocols, in general, do not take values of operations into account, only their order.

Execution constraints

Consistency puts constraints on the interleaving of operations allowed to the single server, and therefore, by the single server paradigm, also on the behavior, i.e., order of operations, occurring at the RMs of the data store.

- R1: The interleaved sequence of operations meets the specification of a (single) correct copy of the objects
  - i.e., produces the same results (return values and internal state)
  - maintains system invariants
- R2: The order of operations in the interleaving originating from a single client is consistent with the order in which that client issued them (program order).
- R3: The order of operations in the interleaving is consistent with the global (real-time) ordering of the operations.

Concurrency and interleavings



Total number of interleavings: 24

Interleavings by R2

- $A_1B_1B_2A_2$
- $B_1A_1B_2A_2$
- $A_1A_2B_1B_2$
- $A_1B_1A_2B_2$
- $B_1B_2A_1A_2$
- $B_1A_1A_2B_2$

Interleavings by R3

- $A_1B_1B_2A_2$
- $B_1A_1B_2A_2$

Sequential consistency

Actual executions that satisfy R1 and R2 are *sequentially consistent.*

- A data store with replicated objects is *sequentially consistent* when *all* its executions are sequentially consistent.

- Sequential consistency allows swapping the order of pairs of subsequent operations originating from distinct clients to obtain a single server execution (even when they are conflicting)
  - Reflects possible transmission delays between clients and the data store and between data store AEs.
- This is the strongest form of consistency that can be enforced without loosing full benefits offered by concurrency.
  - Where concurrent events are those that have no causal dependency

Definition

A data store is sequentially consistent when

The result of any execution is the same as if the read and write operations by all processes on the data store where executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

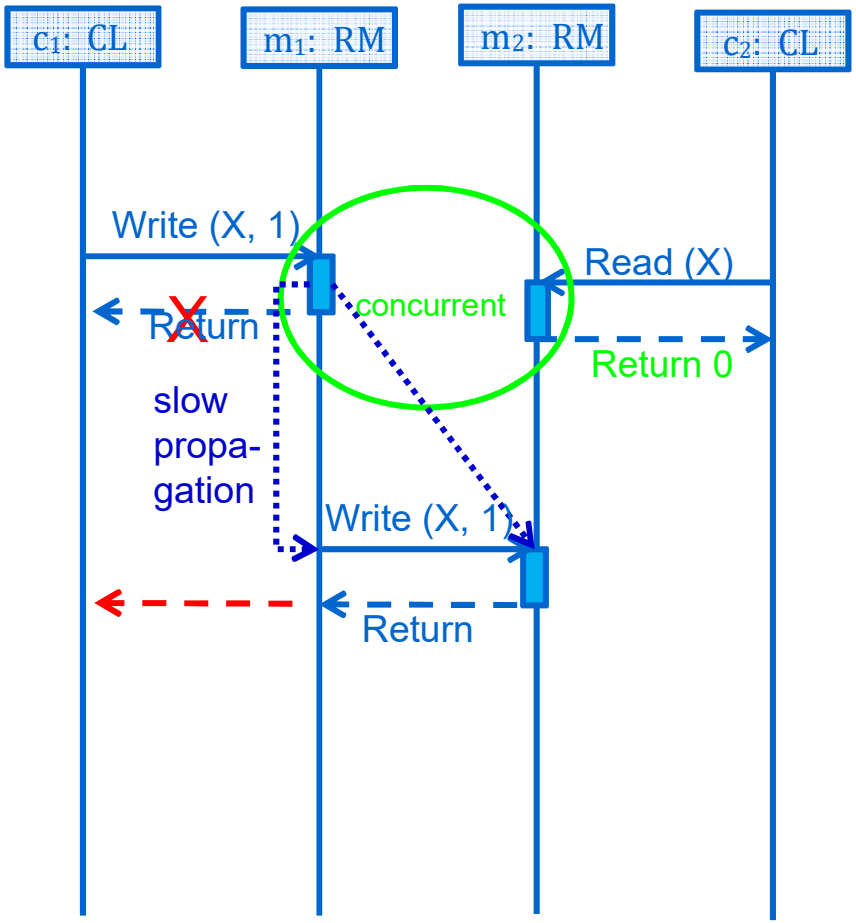
Linearizability

Actual executions that satisfy R1 and R3 are called *linearizable*

- A data store with replicated objects is *linearizable* when *all* its executions are linearizable.
- Linearizability allows swapping the order of pairs of subsequent operations originating from distinct clients provided they do not conflict, i.e., they refer to distinct objects, or they are both read operations.
- Swapping conflicting operations at a single server, in principle modifies the result (return value of reads or resulting state of logical object).

From here on: front ends omitted!

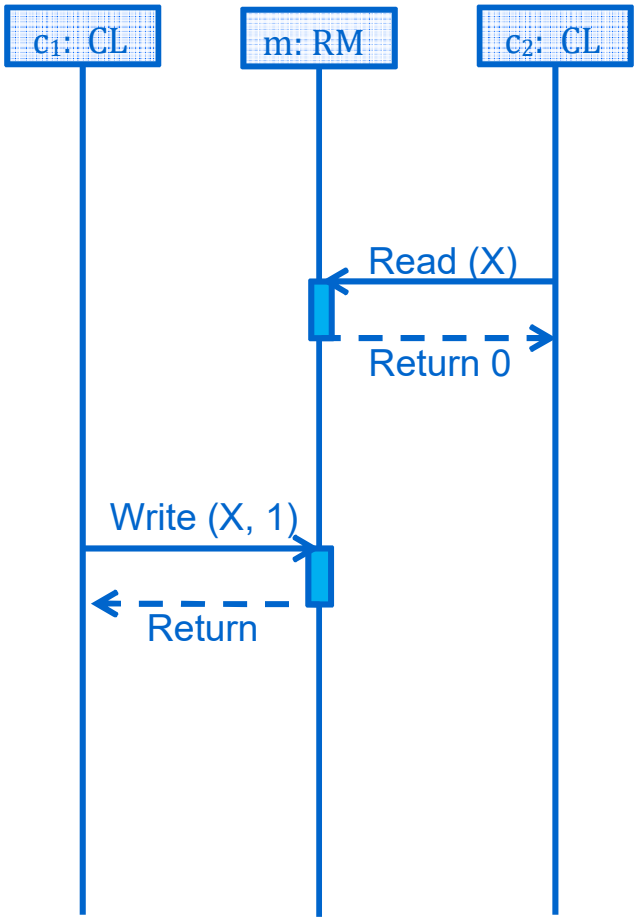
Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



Linearizable (execution, but not the store!!!)

but wrong implementation!

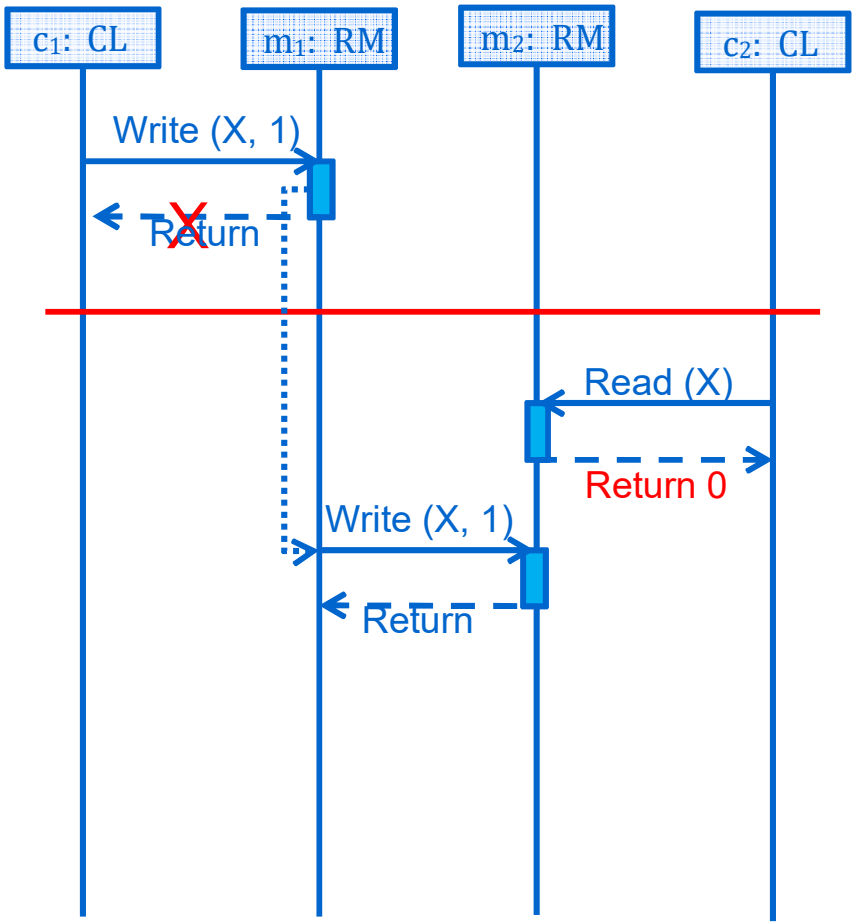
Single server execution
(1-out-of-2 allowed by R3)



R1 not violated

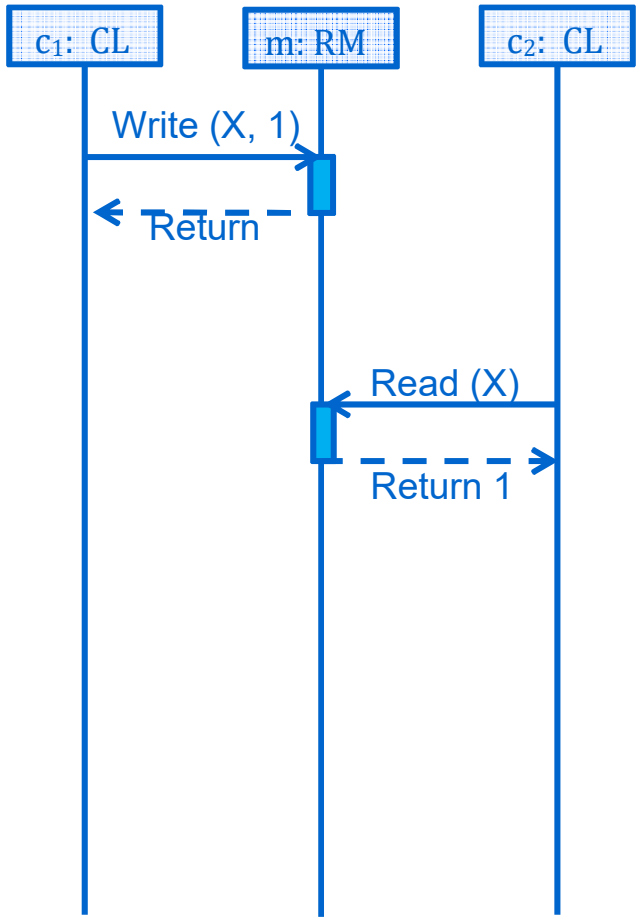
m_2 follows m

Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



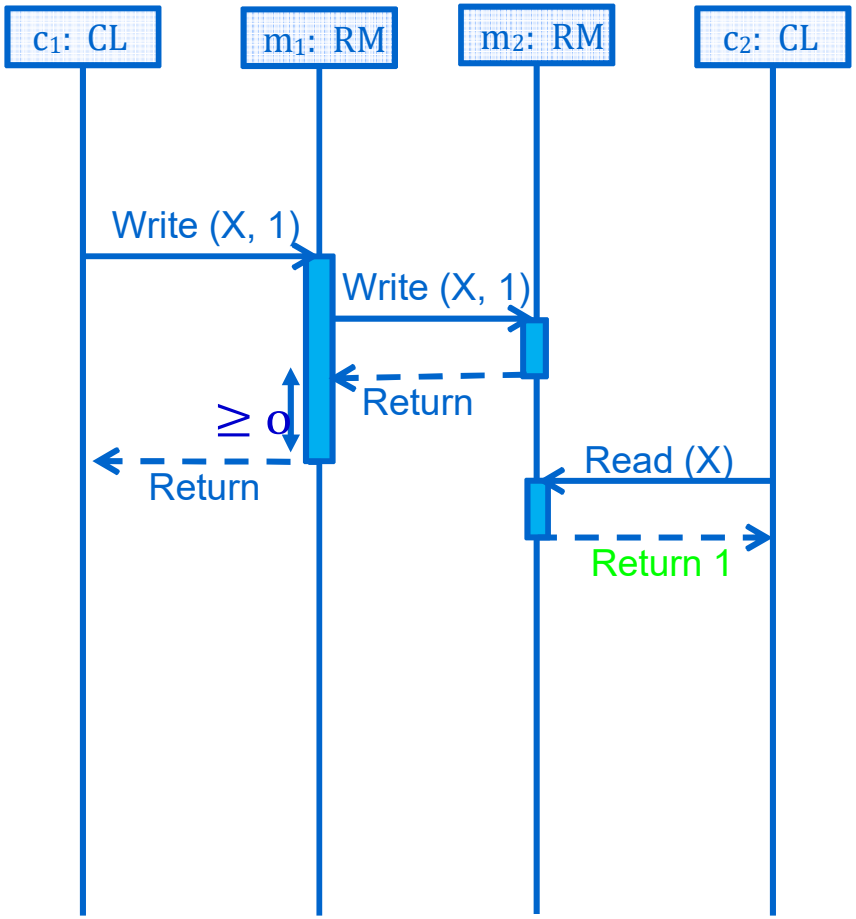
Not linearizable: the execution and hence also the store

Single server execution
(the only one allowed by R3)



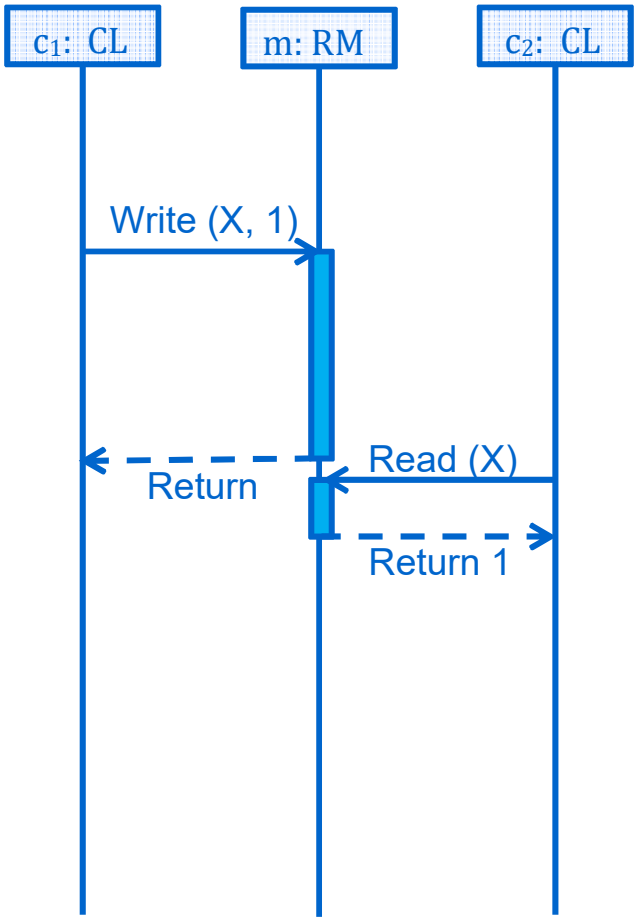
R1 violated

Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



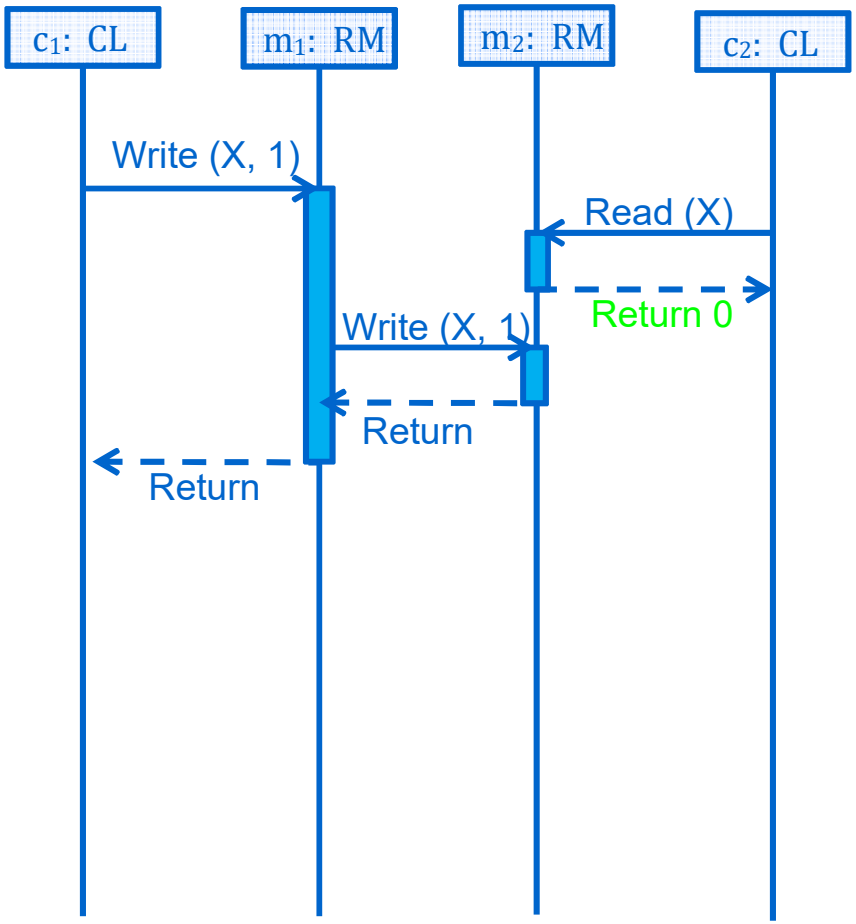
Linearizable (both execution and store)

Single server execution
(the only one allowed by R3)



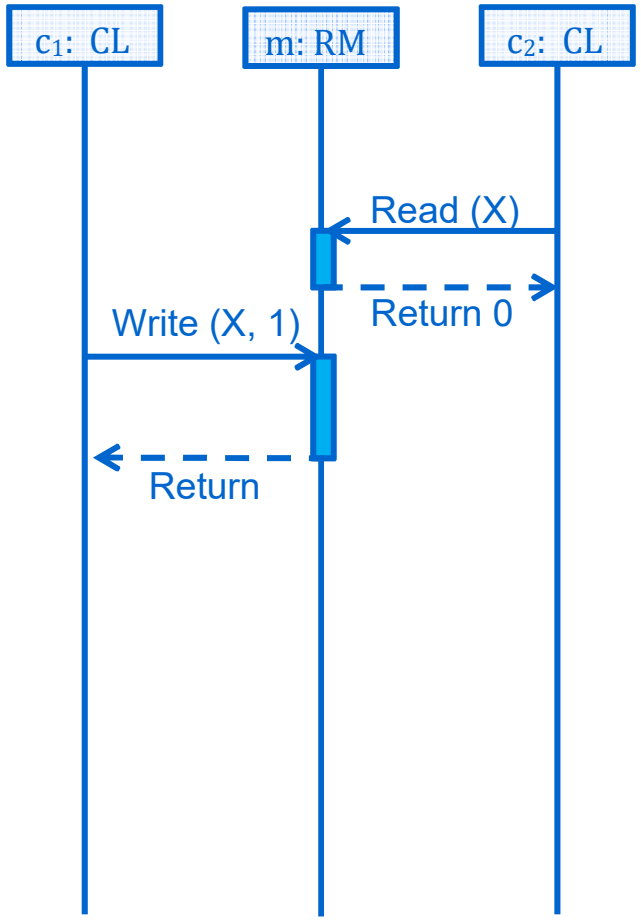
R1 not violated

Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



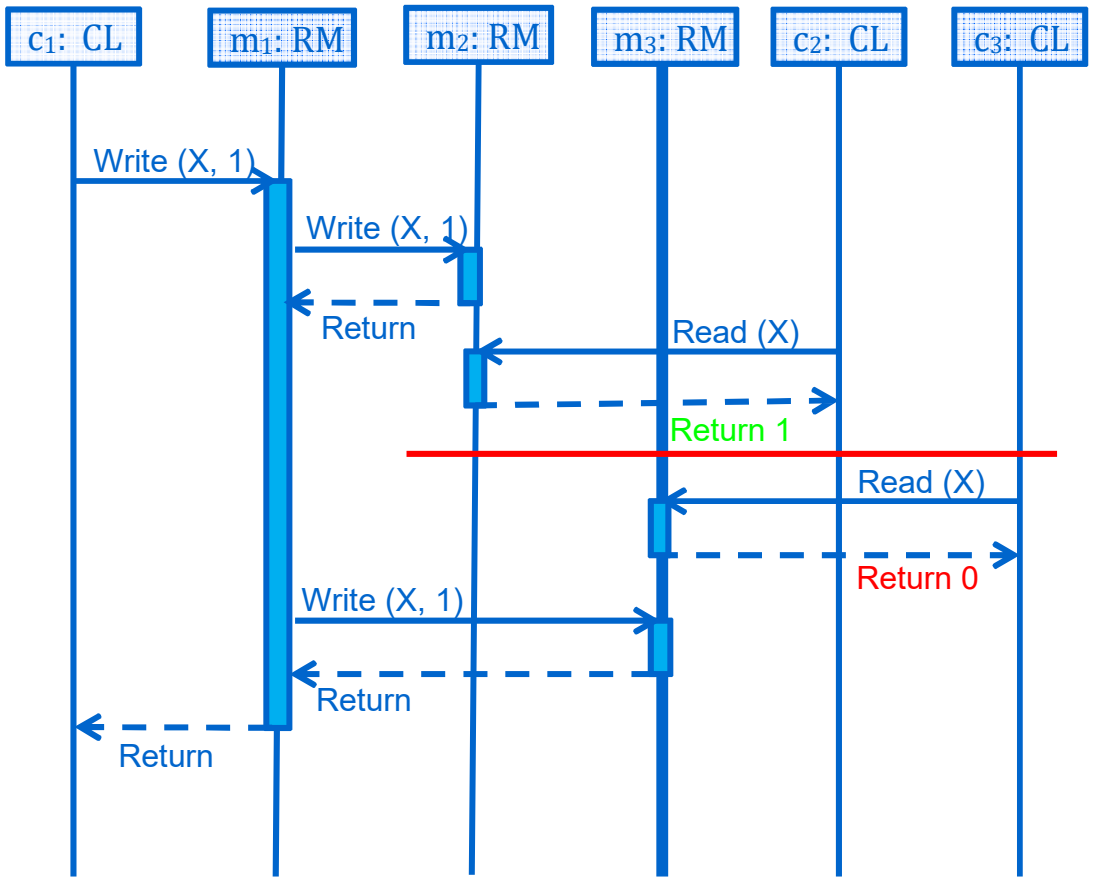
Linearizable

Single server execution
(1 out-of 2 allowed by R3)



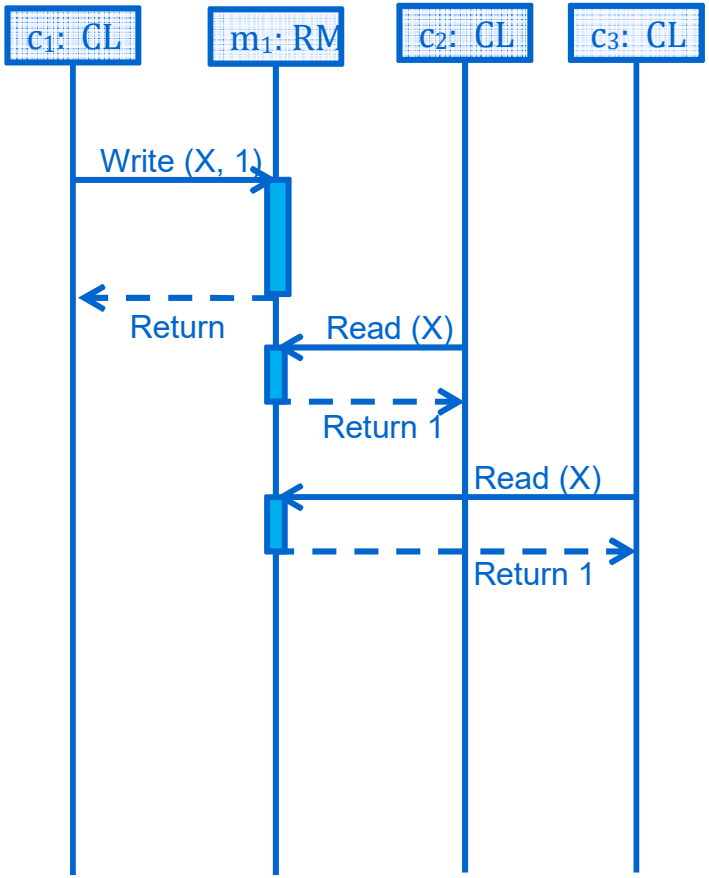
R1 not violated

Initially: $x = \langle 0,0,0 \rangle$



Not linearizable

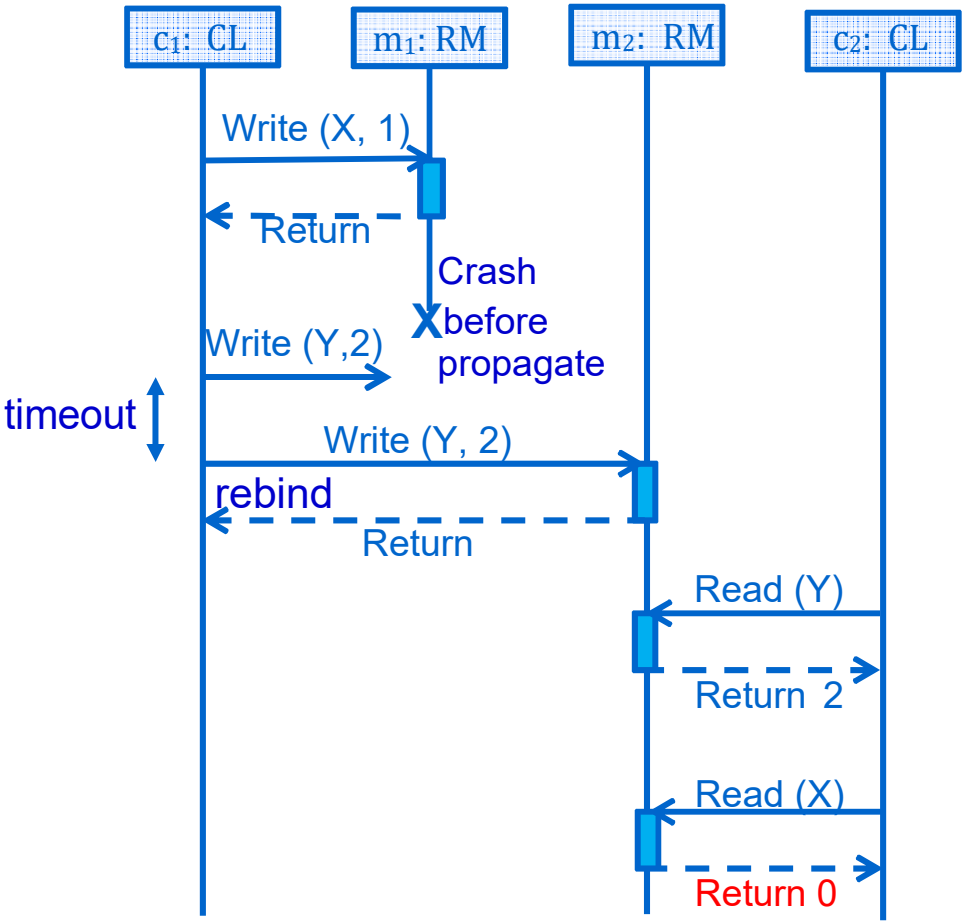
Single server execution (1 out-of 3) allowed by R3. None allows reading 1 twice.



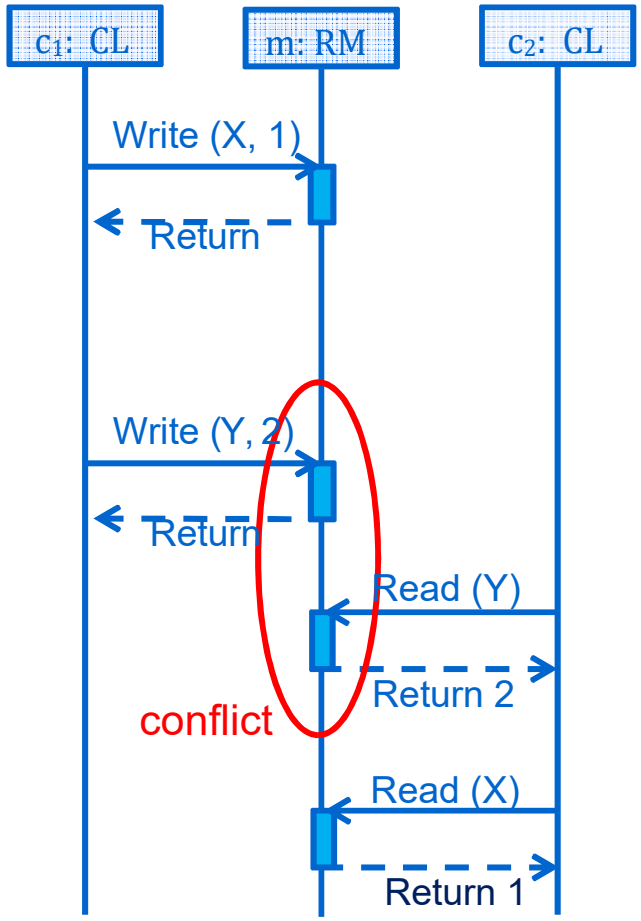
R1 violated

Logical variables x and Y

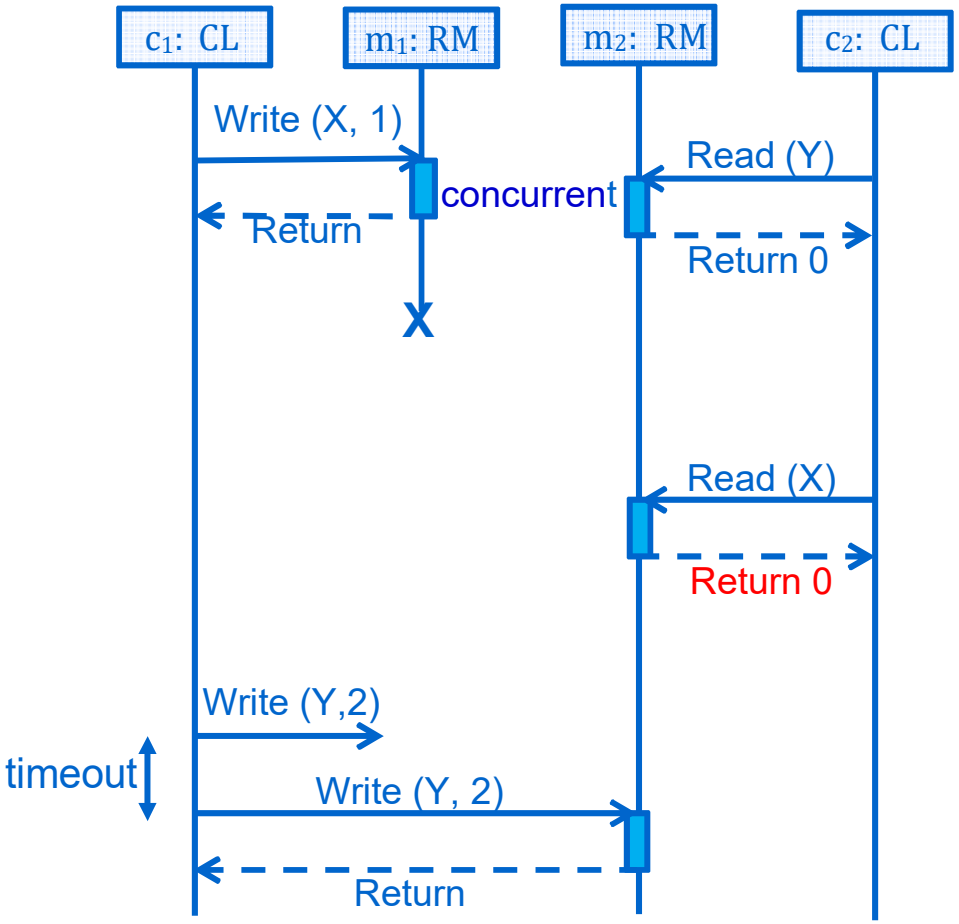
Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



Single server execution
(the only one satisfying R3)

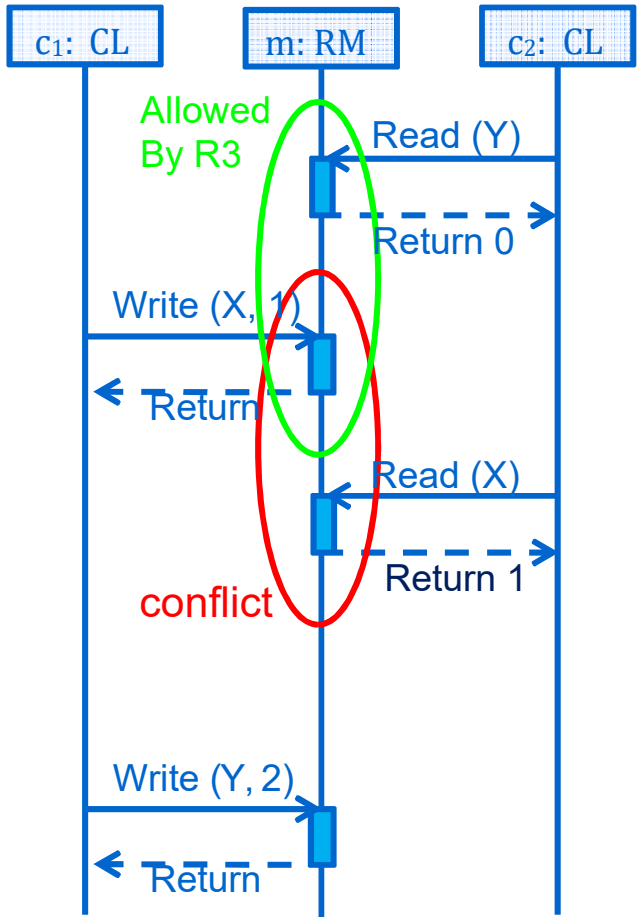


Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$

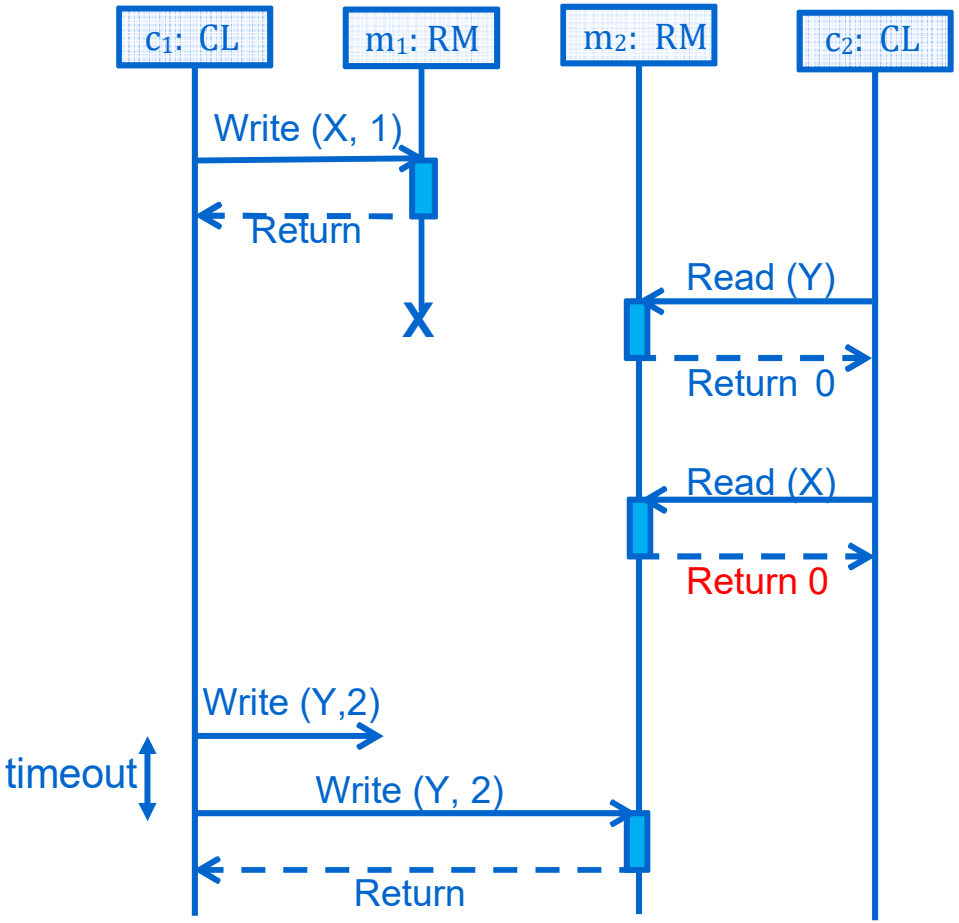


Not linearizable

Single server execution
1 out-of 2 allowed by R3

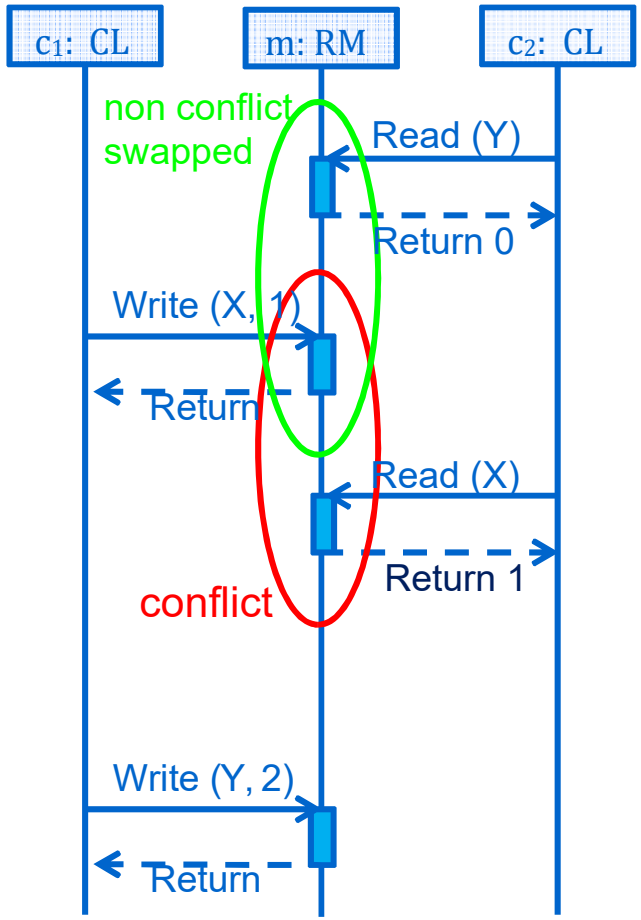


Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



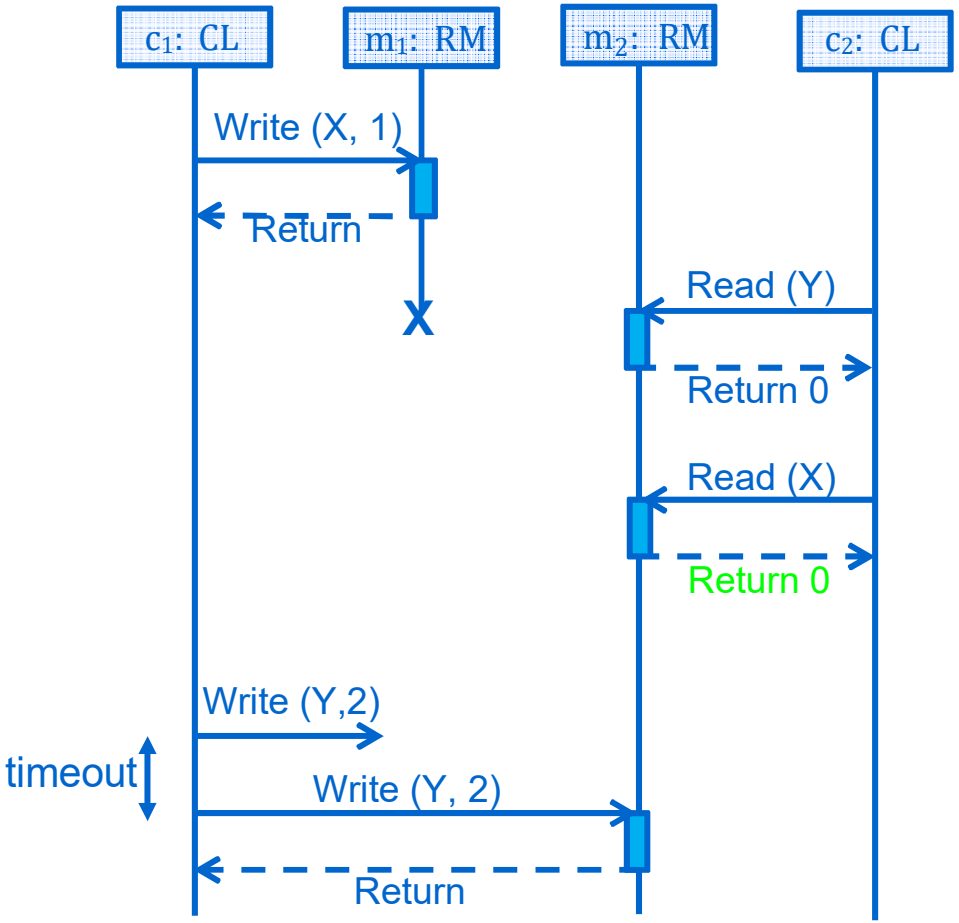
Not linearizable

Single server execution does not satisfy R3, shows swapping non-conflicts does not alter linearizability



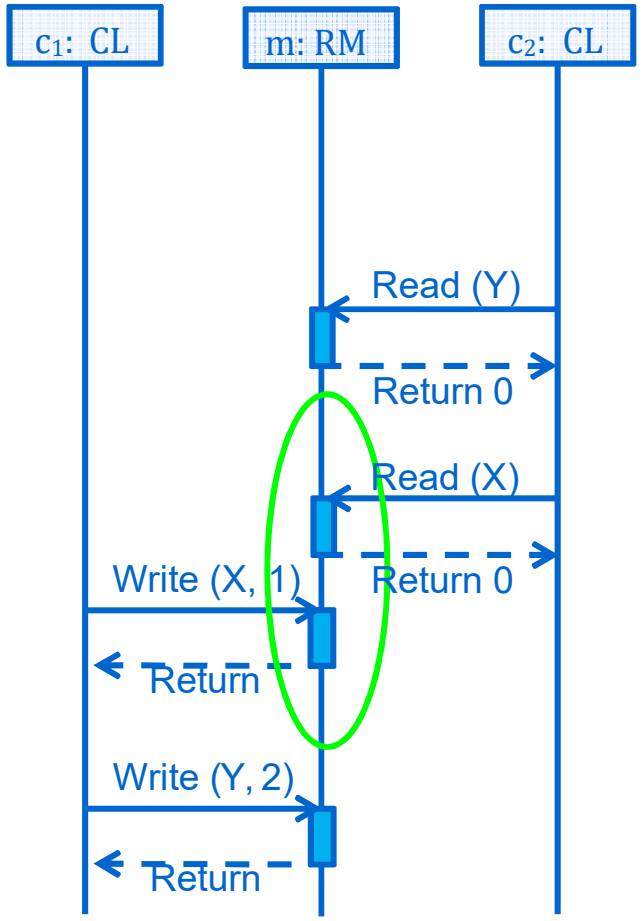
not linearizable

Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



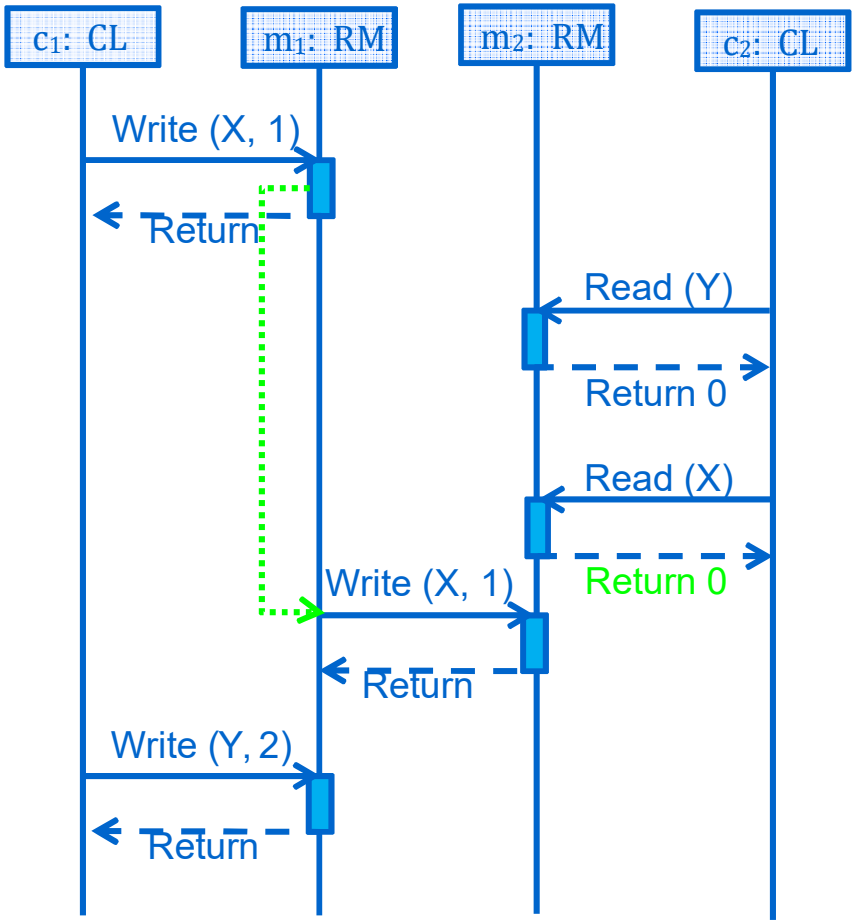
sequential consistent
early return for Write(X,1) is OK in this execution

Single server execution:
does not satisfy R3,
but does satisfy R2.



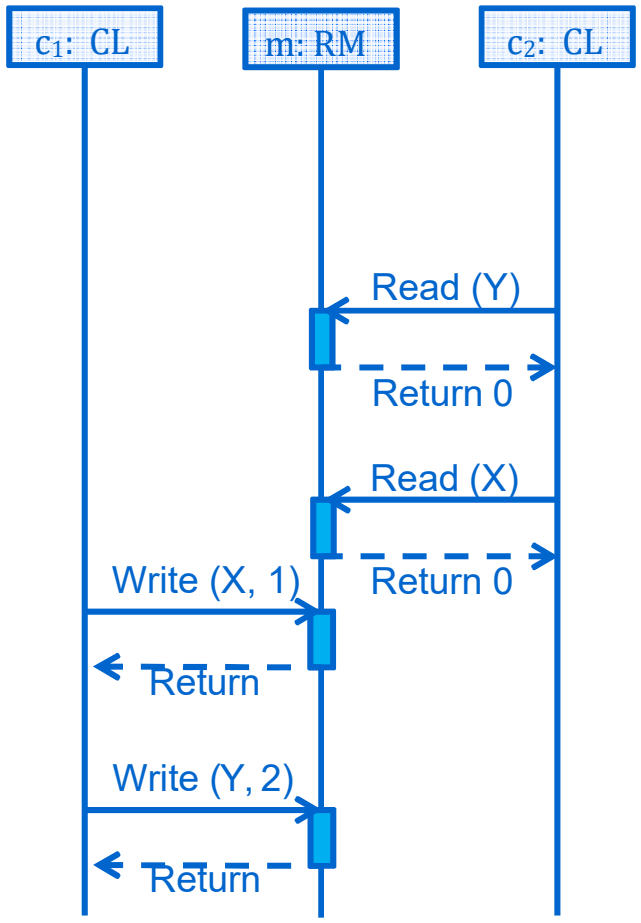
conflicting operations swapped (allowed)

Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



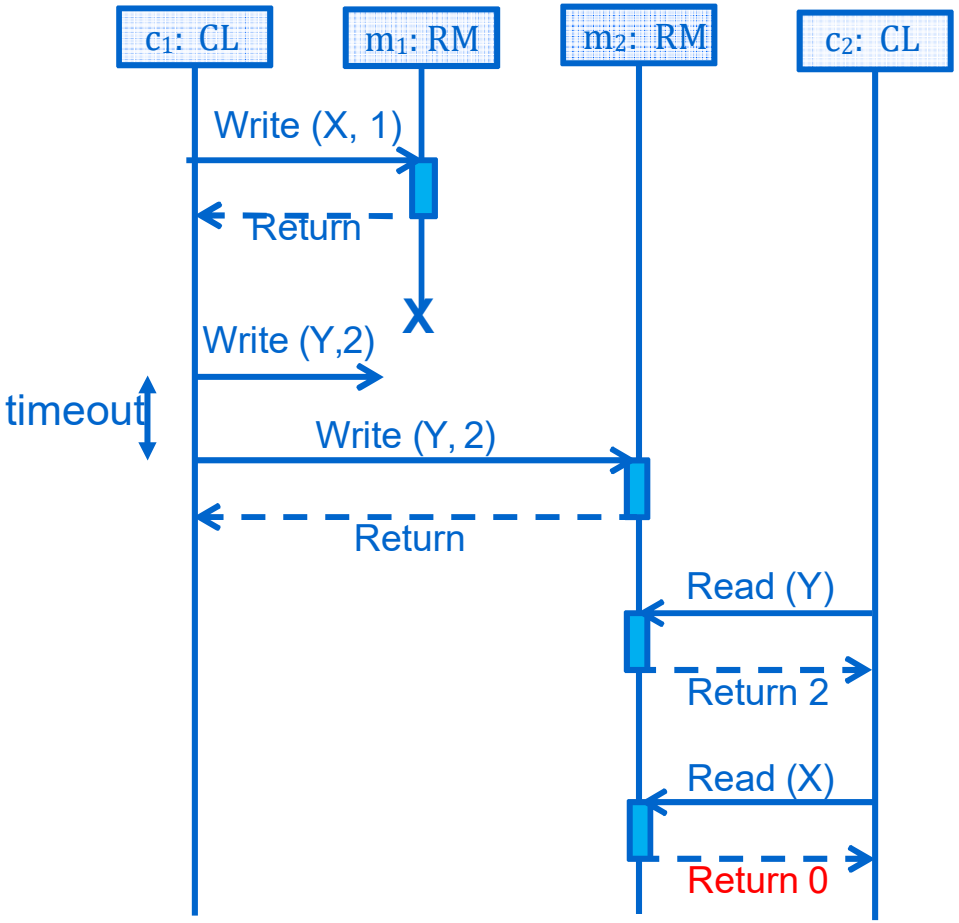
Can also happen without failure
Just slow propagation

Single server execution



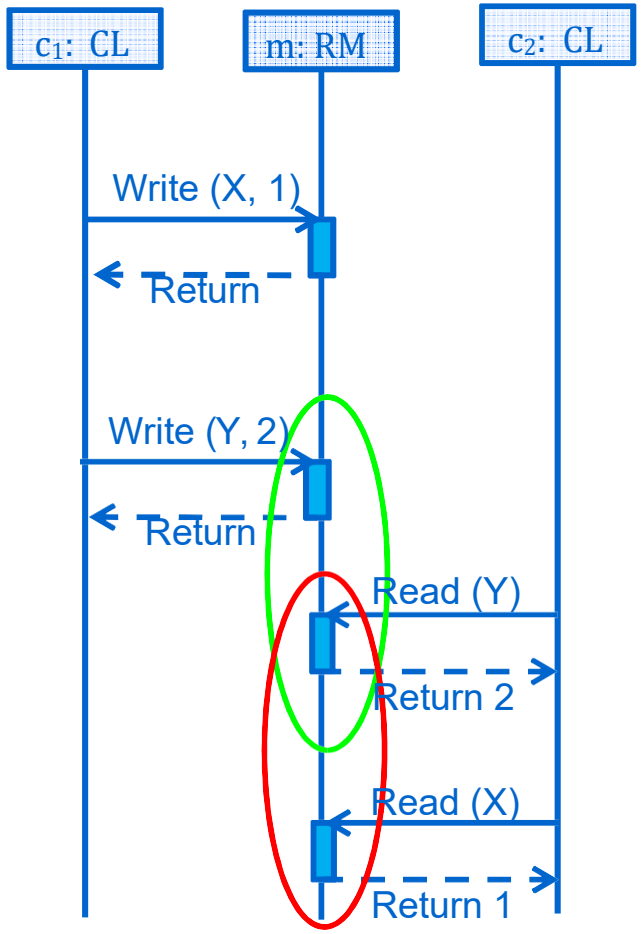
Reordering reflects arrival time at replica manager m_2

Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$

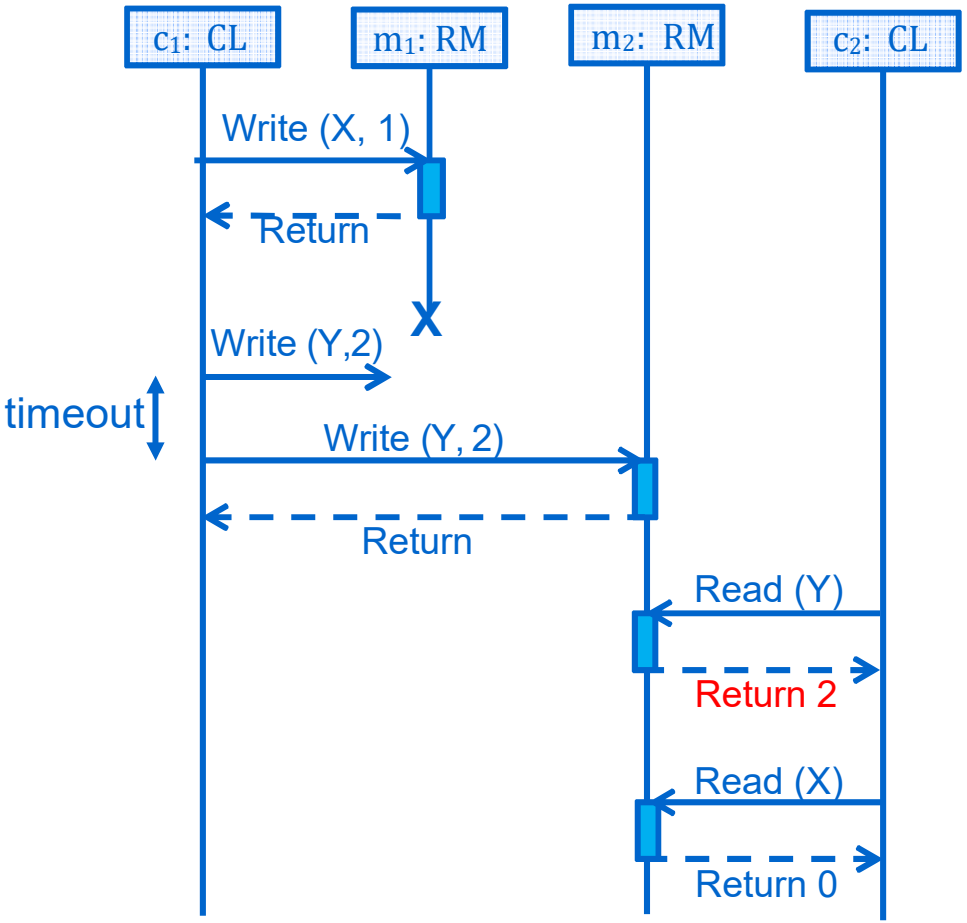


Not sequential consistent

Single server execution
1-out-of 6
Reordering does not help

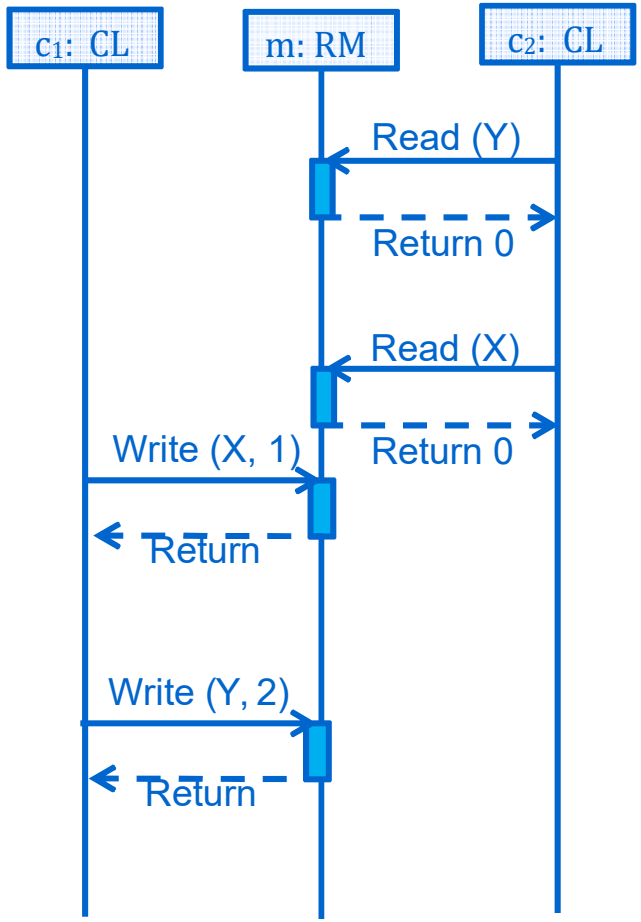


Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



Not sequential consistent

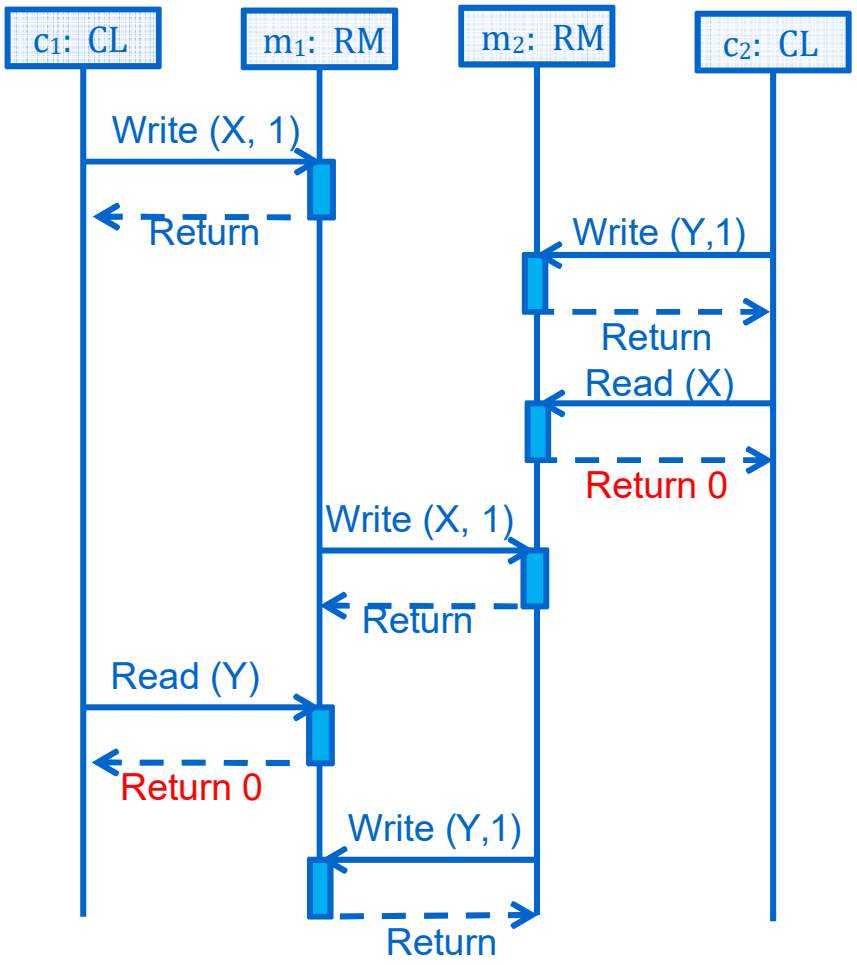
Single server execution
1-out-of 6
Reordering does not help



X-value ok, but
now Y-value wrong

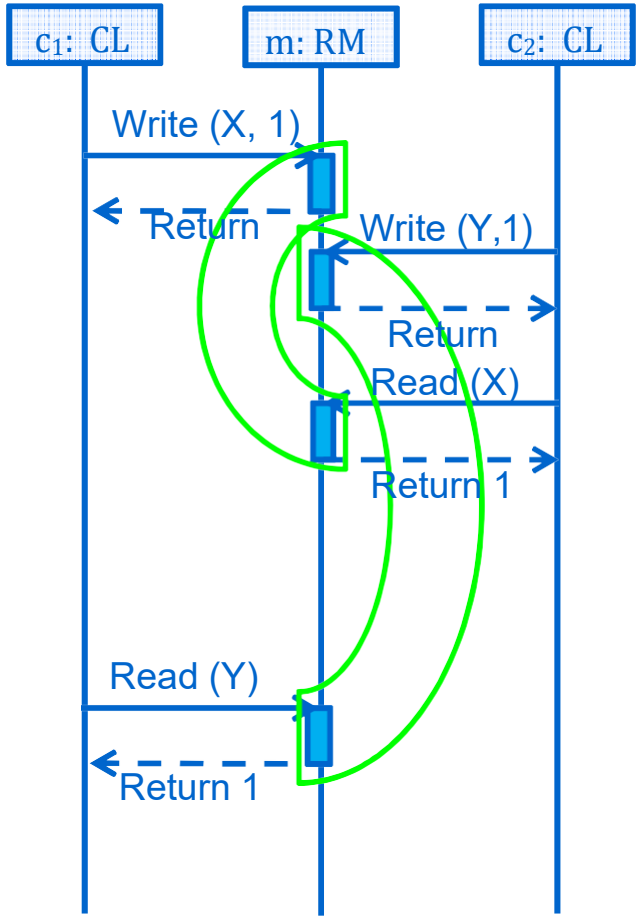
Updates from distinct clients

Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



Not sequential consistent

Single server execution



Reordering conflicts can only modify 1 read

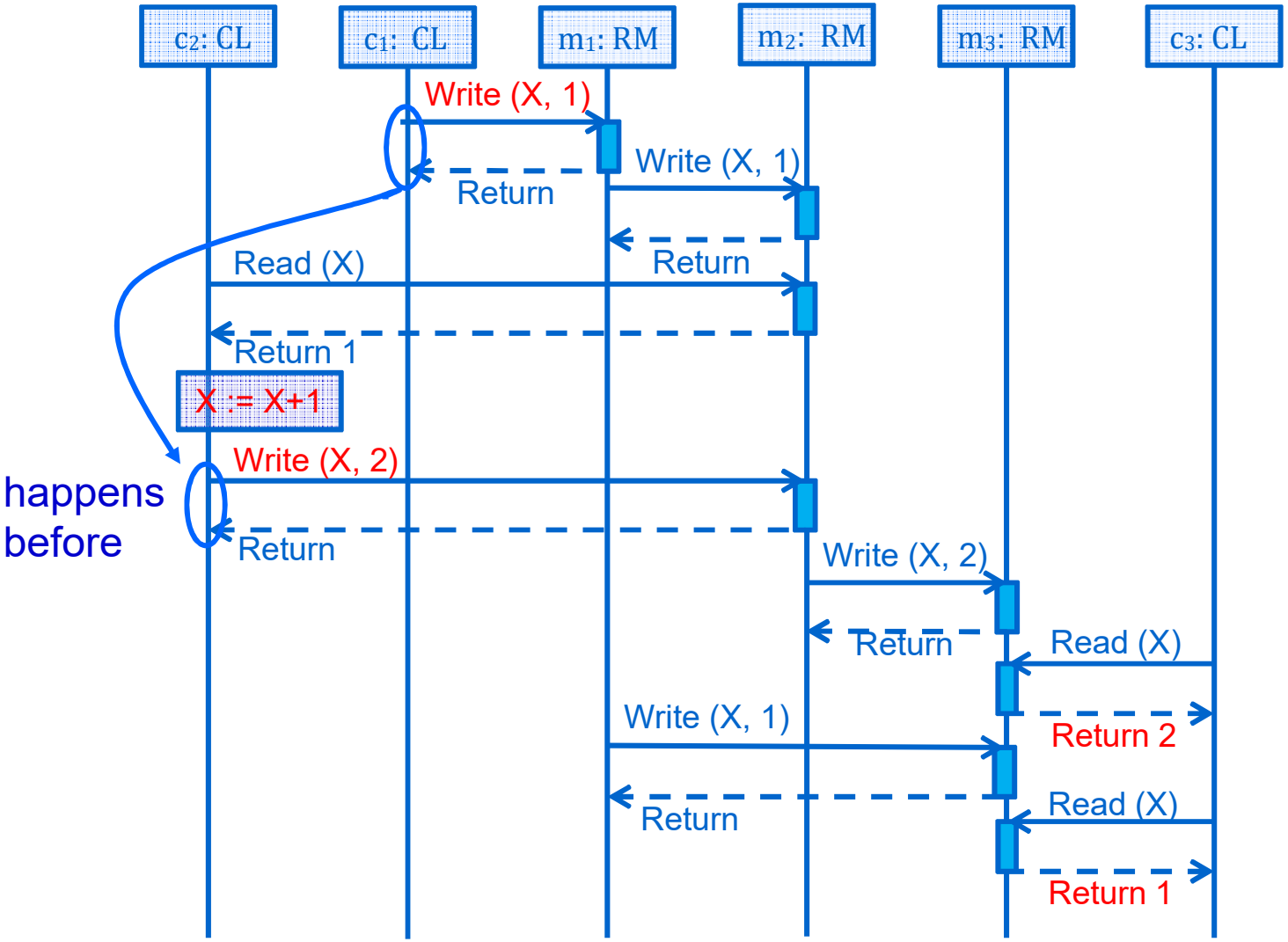
Other forms of consistency

- Causal consistency
  - Operations from different clients that are causally related (as defined by Lamport's happens before relation) may not be swapped.
    - are seen by all clients in the causal order.
  - Determined by means of time stamps and vector clocks.
  - Enforced by the delivery mechanism in the coordination phase
  - Weaker than sequential consistency.

- Eventual consistency
  - In the absence of further updates and system failures all replicas eventually have the same state.
  - Weaker than causal consistency.

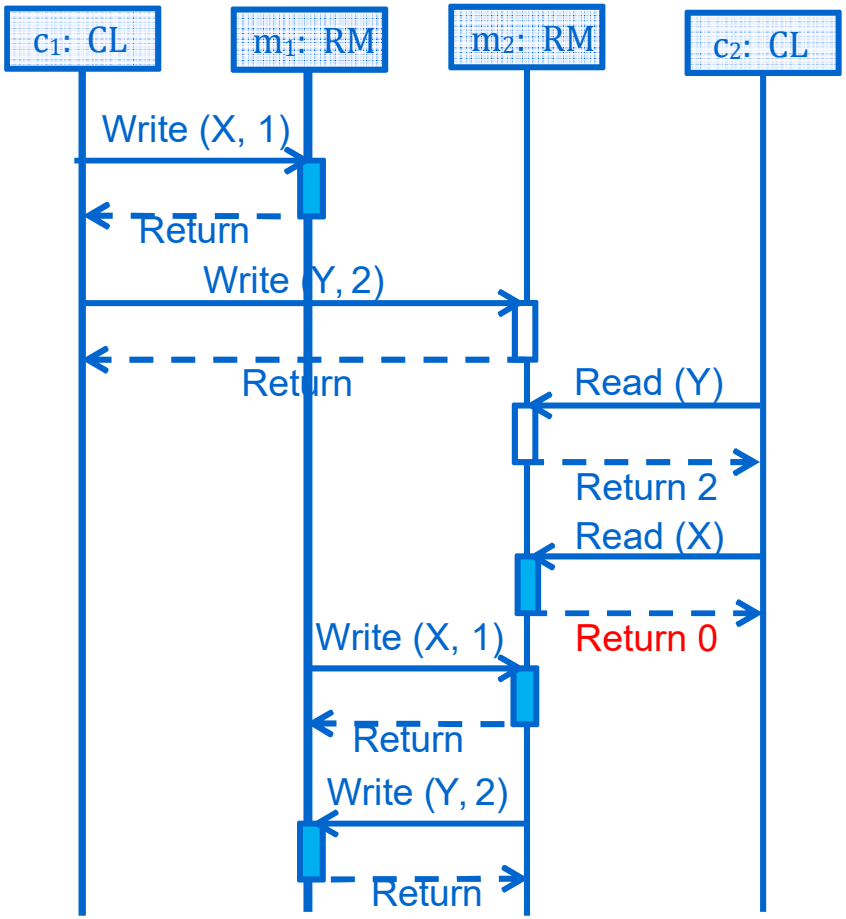
Beware: RM m_2 is superfluous. Only present to show *chain* of causal dependence

Initially: $x = \langle 0,0,0 \rangle$



No causal consistency

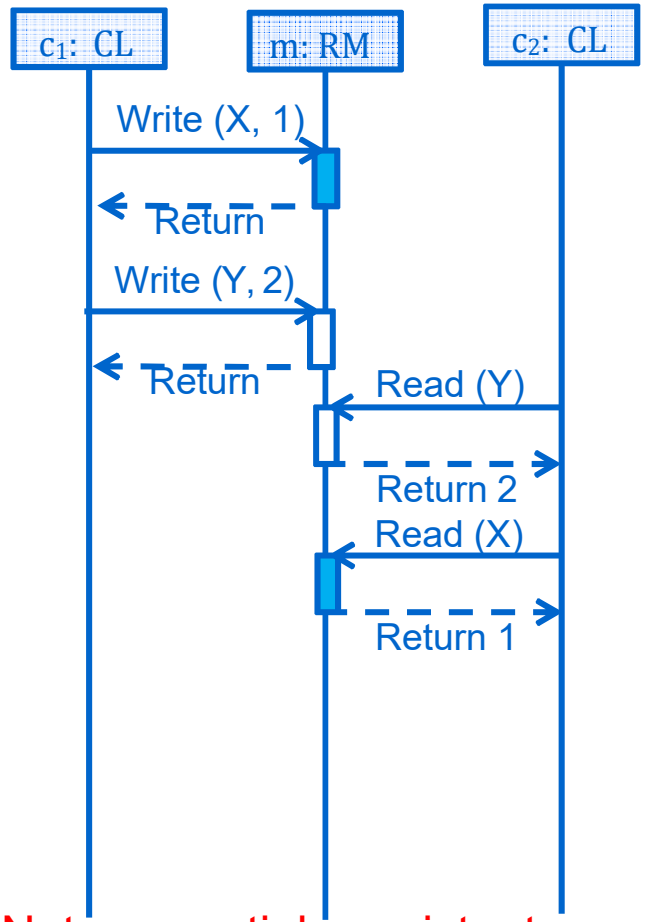
Initially: $x = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



Finally: $x = \langle 1,1 \rangle \wedge Y = \langle 2,2 \rangle$

So, eventually consistent

Single server linear execution

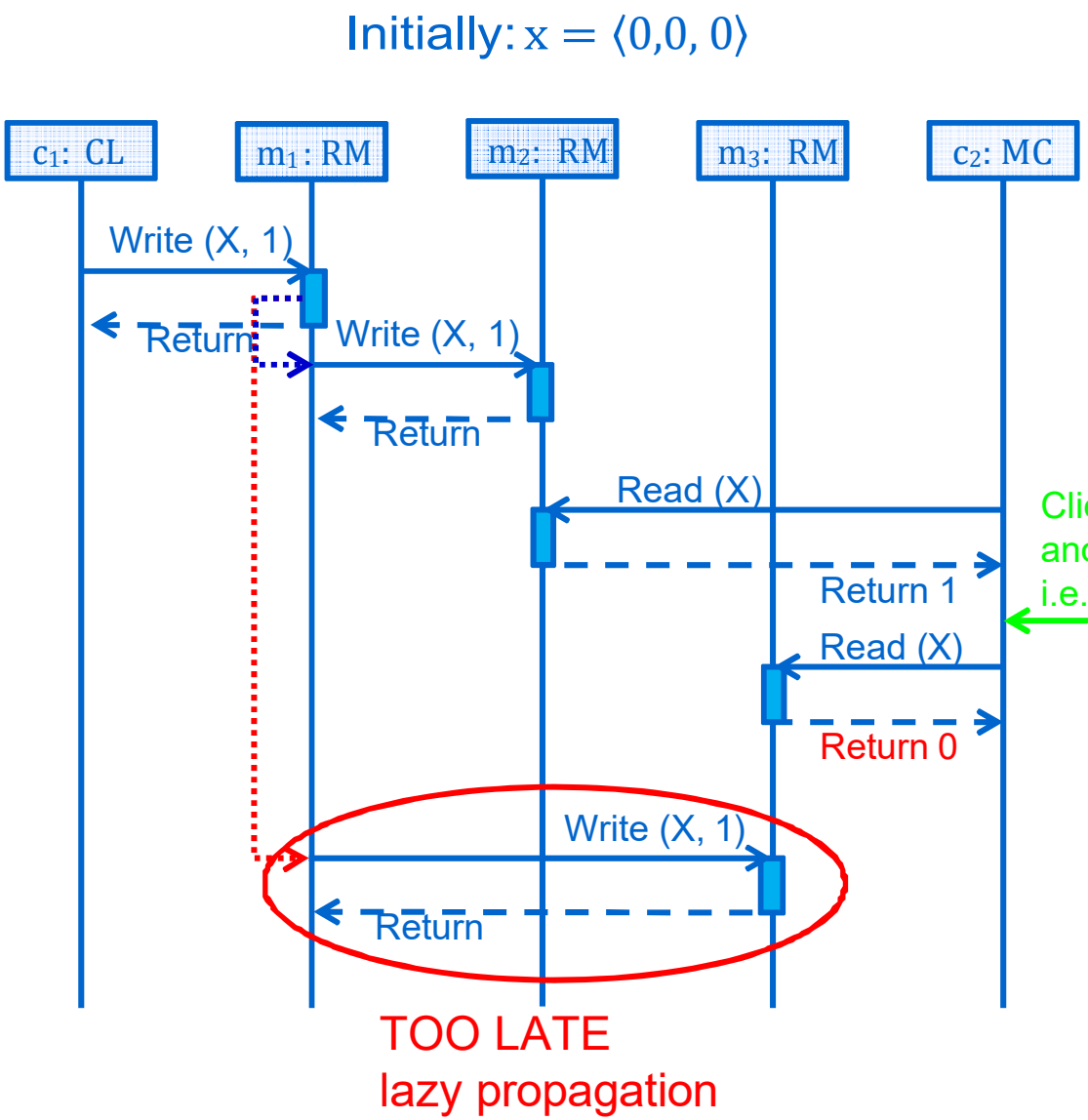


Not sequential consistent.
Note that swapping to get X right, makes Y wrong

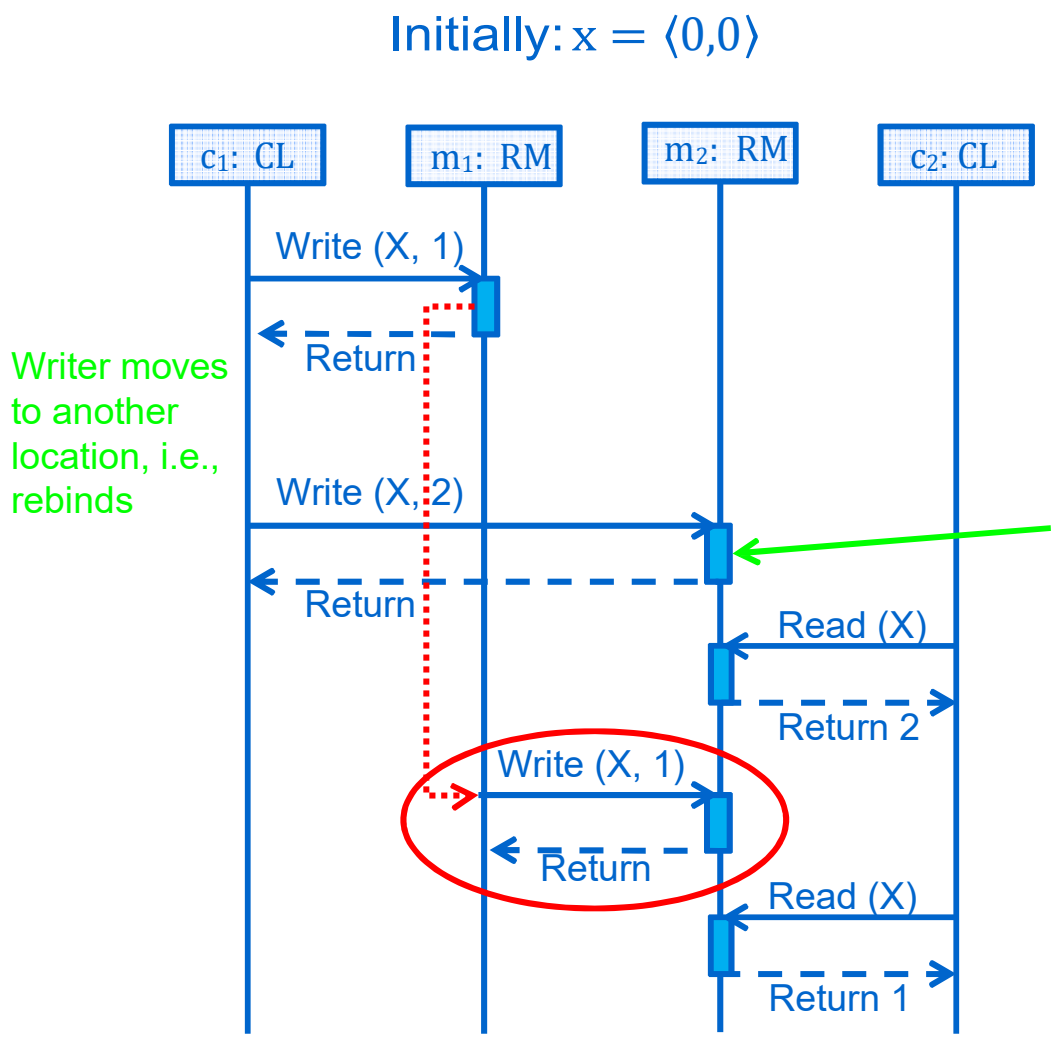
Client-centric models

- Monotonic read consistency
  - If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more *recent* one.
- Monotonic write consistency
  - A write operation on a data item x is *completed* before any successive write operation on x by the same process.
- Read your writes
  - 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.
- Writes follow reads
  - A write operation by a process on a 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 than was read.

Monotonic read violation



Client will not notice that 0 is an old value unless it has a time stamp (or version number)

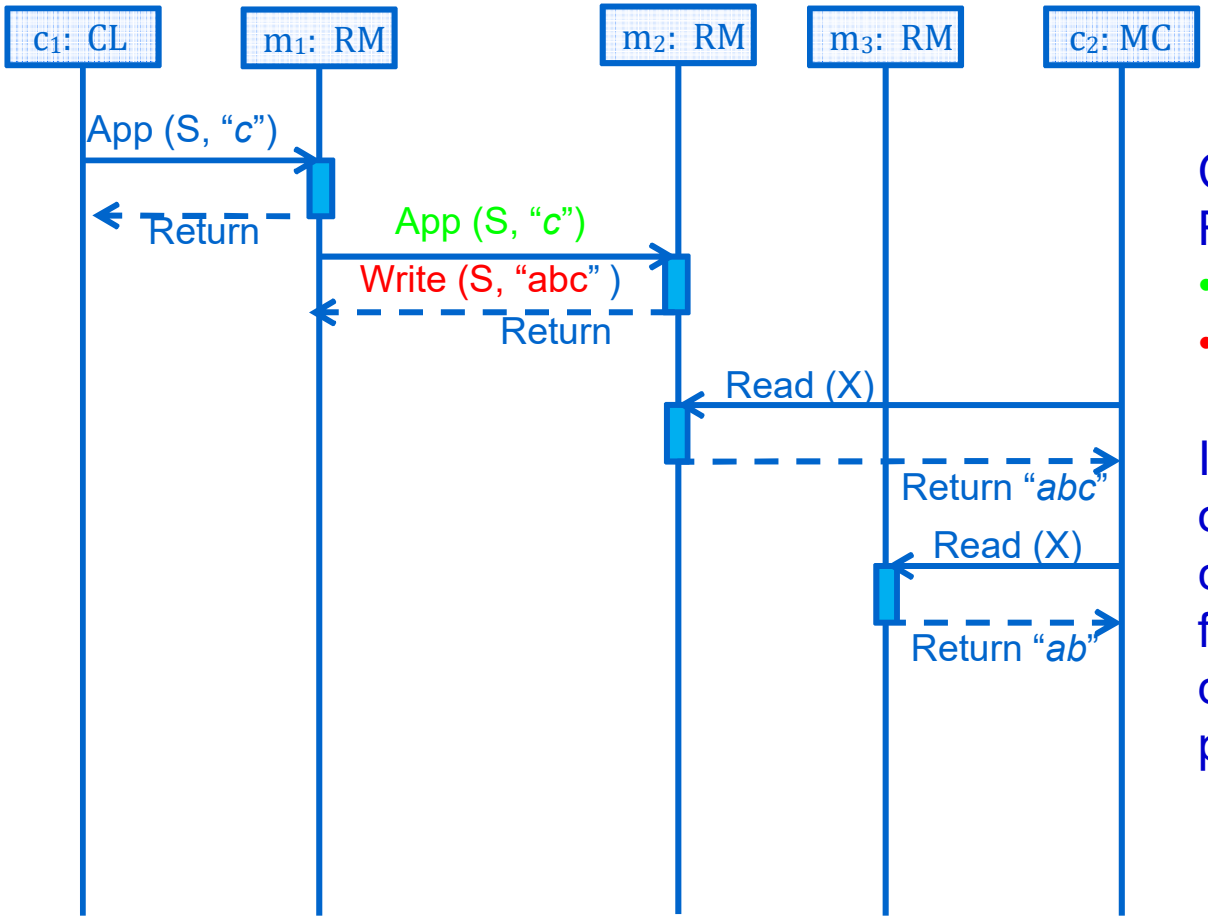


Monotonic read violation at c_2

or written by another client, but then global time stamps are needed to determine whether it is more recent

TOO LATE
delayed propagation

Initially: $S = \langle \text{"ab"}, \text{"ab"}, \text{"ab"} \rangle$



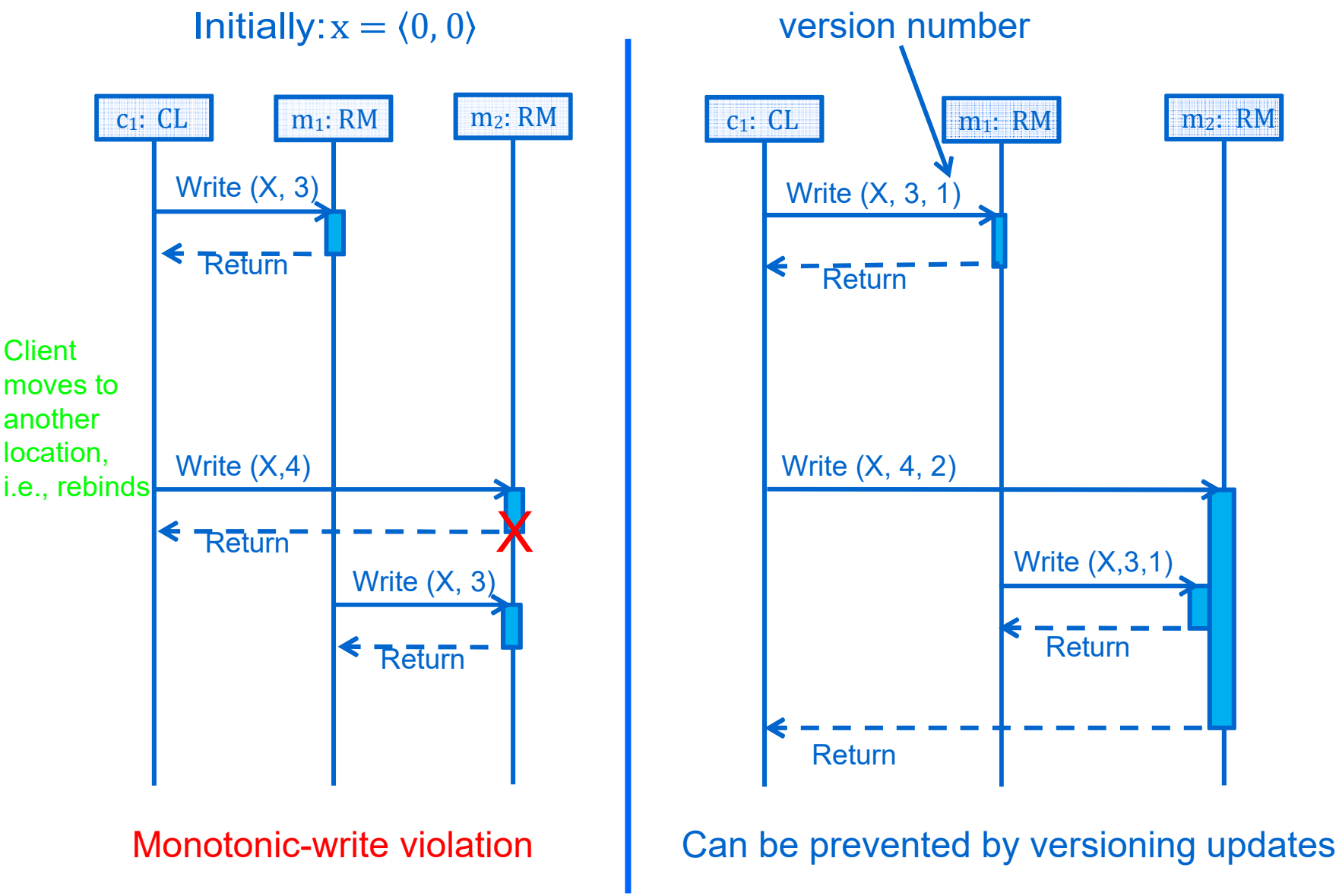
Clients can only append, RMs have two options

- forward the operation
- forward its result

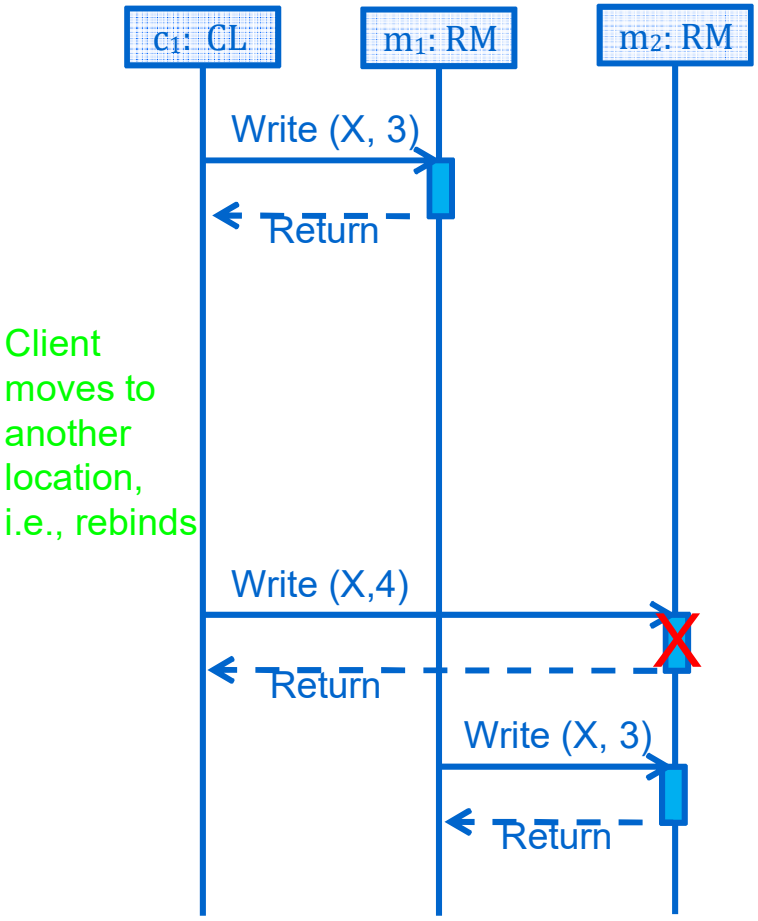
In either case, for this operation the mobile client can observe that forwarding has not been done, because it sees a prefix on the second read

Client-centric models

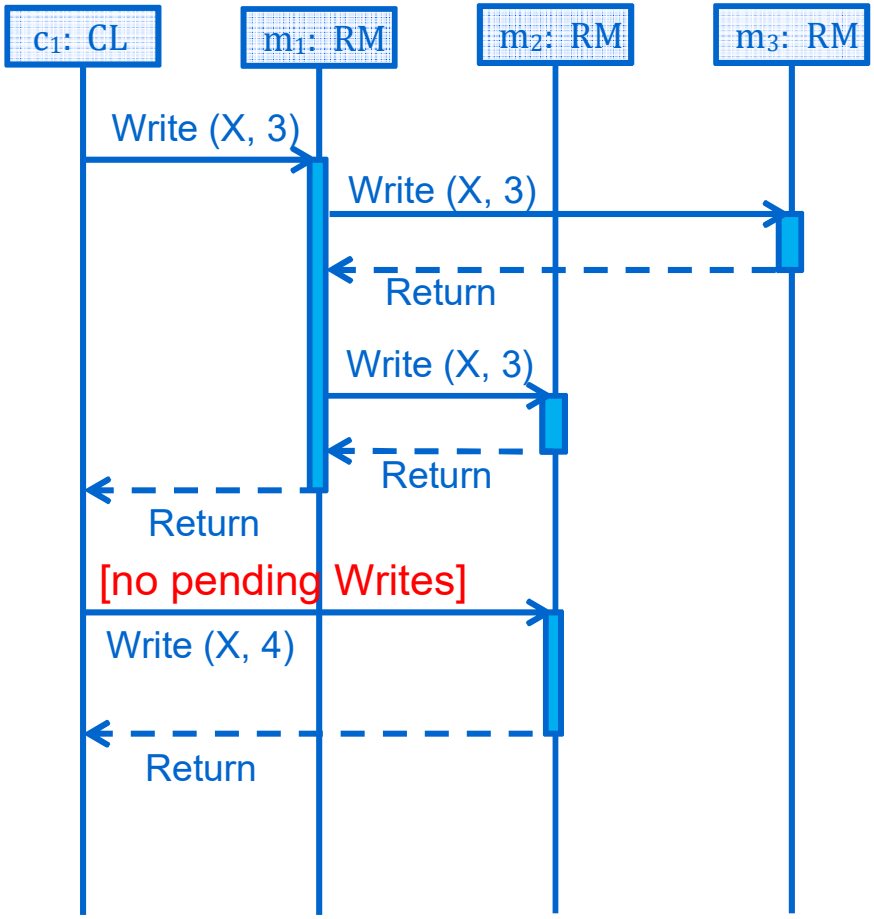
- Monotonic read consistency
  - If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more *recent* one.
- Monotonic write consistency
  - A write operation on a data item x is *completed* before any successive write operation on x by the same process.
- Read your writes
  - 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.
- Writes follow reads
  - A write operation by a process on a 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 than was read.


Initially: x = ⟨0, 0⟩
c1: CL
m1: RM
m2: RM
Write (X, 3)
Return
Client moves to another location, i.e., rebinds
Write (X,4)
Return
Write (X, 3)
Return
Monotonic-write violation
version number
c1: CL
m1: RM
m2: RM
Write (X, 3, 1)
Return
Write (X, 4, 2)
Write (X,3,1)
Return
Return
Can be prevented by versioning updates

Initially: x = ⟨0, 0⟩



Monotonic-write violation



Can be prevented by making writes atomic

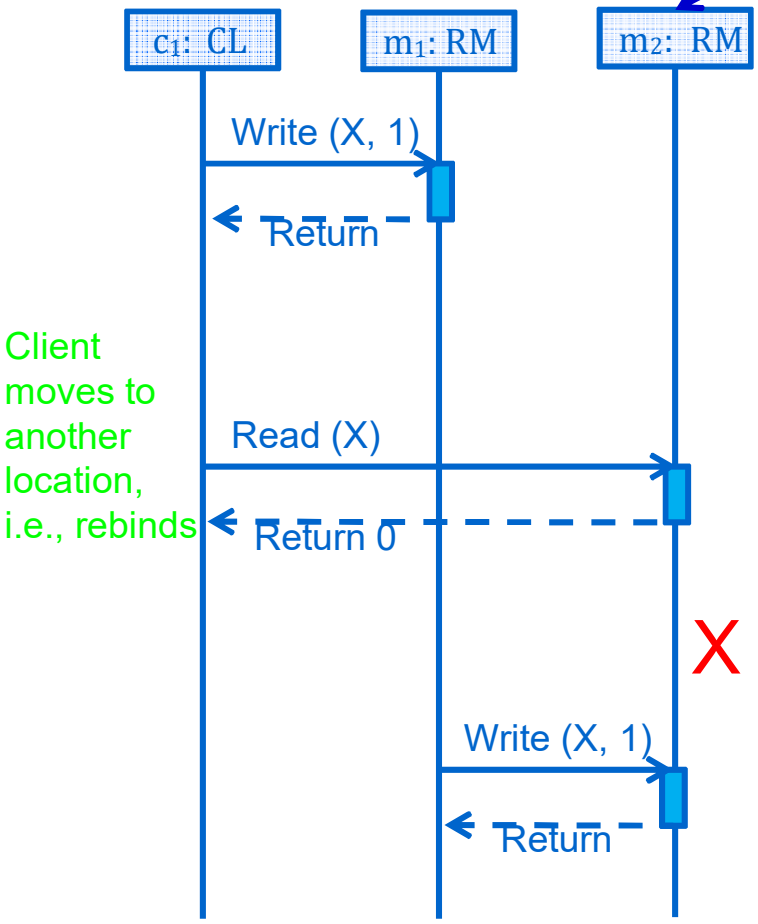
Client-centric models

- Monotonic read consistency
  - If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more *recent* one.
- Monotonic write consistency
  - A write operation on a data item x is *completed* before any successive write operation on x by the same process.
- Read your writes
  - 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.
- Writes follow reads
  - A write operation by a process on a 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 than was read

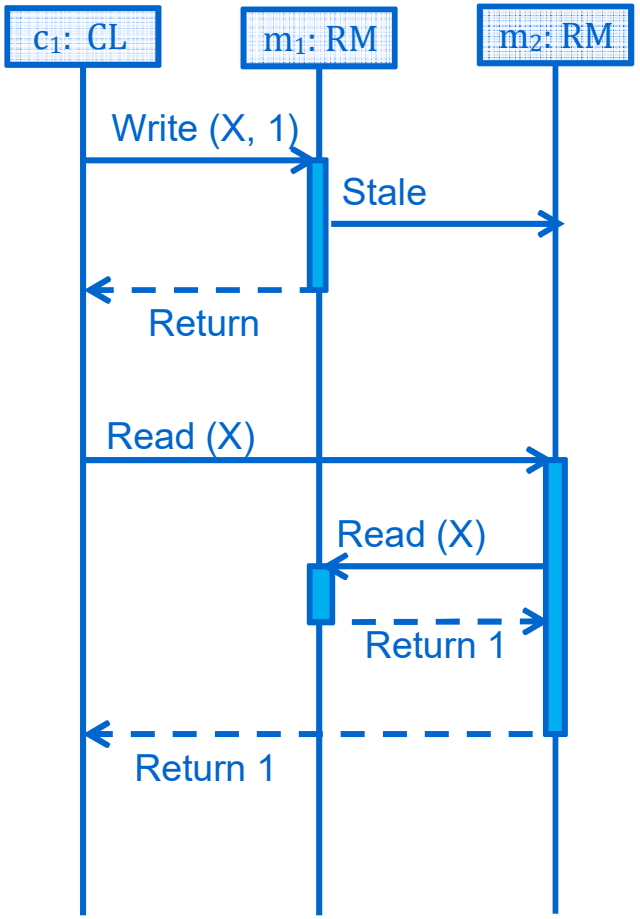
Initially $\langle 0, 0 \rangle$

m_2 can be, e.g, a stale cache

X a private variable of c_1



Read-your-writes violation

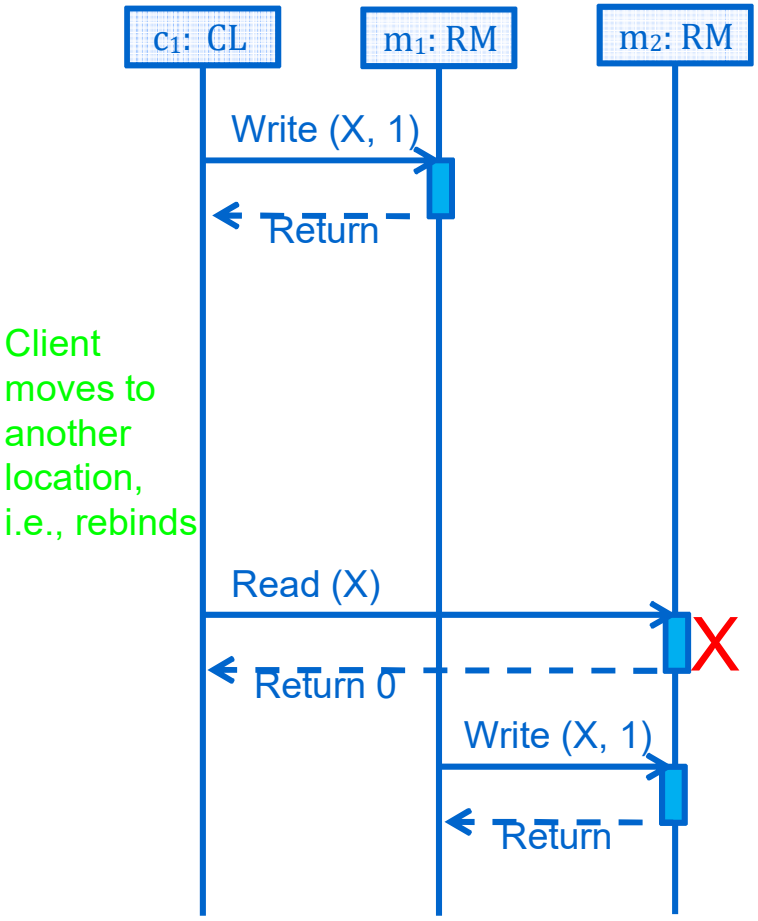


Can be prevented by stale notification

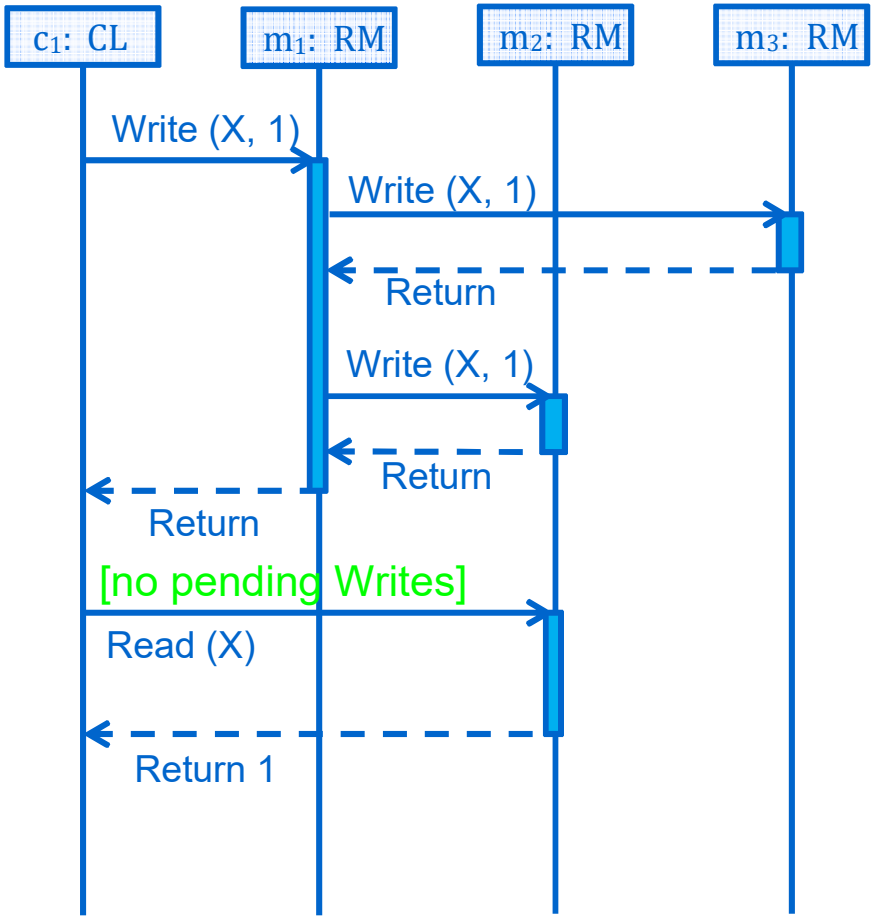
Can also be prevented by making writes atomic

Initially $\langle 0, 0 \rangle$

X a private variable of c_1



Read-your-writes violation

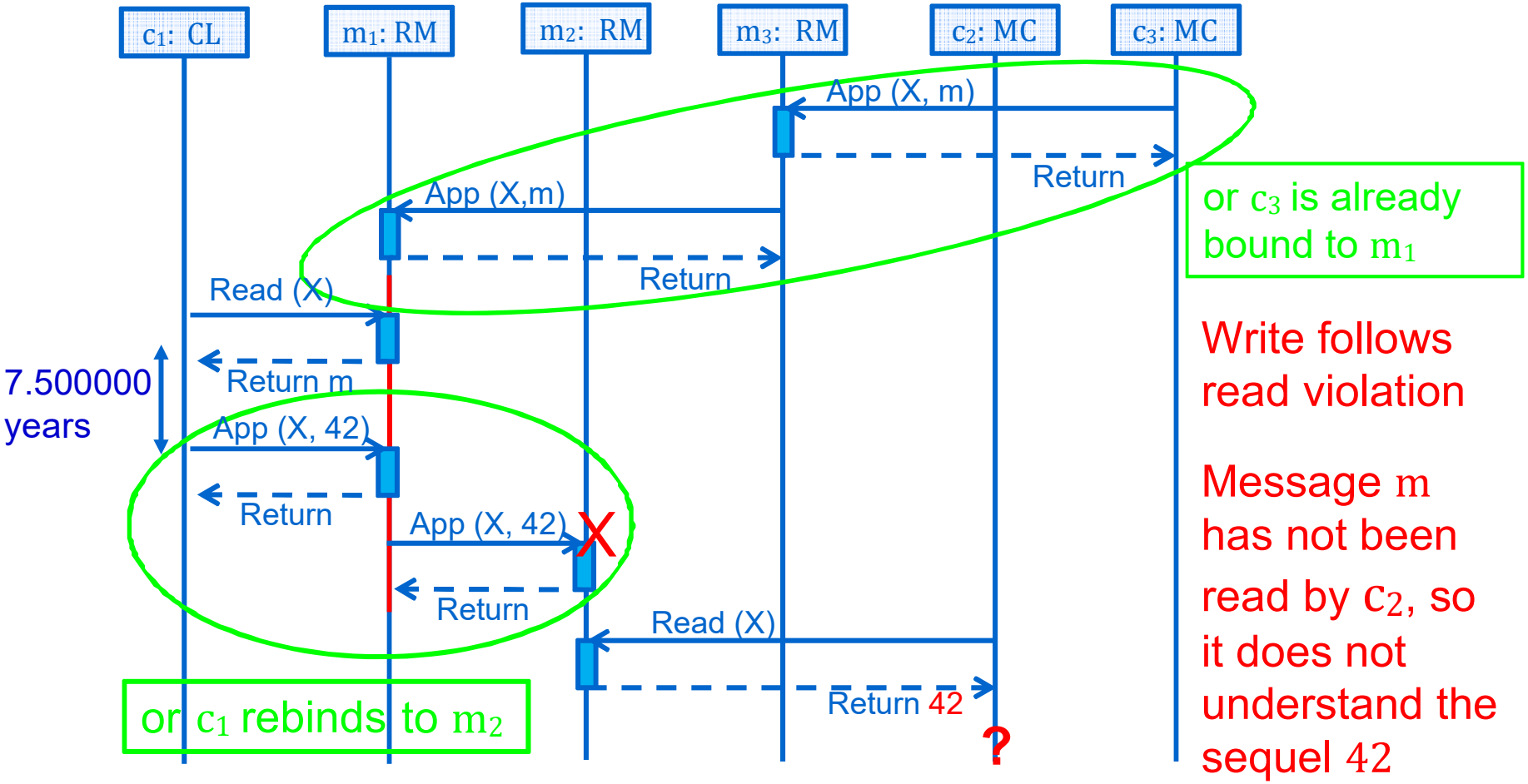


Can be prevented by making writes atomic

Client-centric models

- Monotonic read consistency
  - If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more *recent* one.
- Monotonic write consistency
  - A write operation on a data item x is *completed* before any successive write operation on x by the same process.
- Read your writes
  - 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.
- Writes follow reads
  - A write operation by a process on a 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 than was read.

Initially: x = ⟨"", "", ""⟩



Of course m is the "Ultimate Question of Life, the Universe and Everything"!!!
See: Douglas Adams, The Hitchhikers Guide to the Universe

Lecture Slides Source(Reference)

Course: Architecture of Distributed Systems (2IMN10)

Lecturer: Dr. R.H. Mak

TU/e Technische Universiteit Eindhoven University of Technology