Architecture of Distributed Systems 2017-2018

Replication

R.H. Mak

Agenda

- Introduction
- Consistency models
- Replica management
- Architectural case studies

Agenda

- Introduction
  - Quality drivers
  - Replication transparency
  - Architectural concerns
  - Basic architecture
- Consistency models
- Replica management
- Architectural case studies

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
  - Latency reduction by accessing a nearby replica
  - Throughput improvement by concurrent access of distinct replicas
  - 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

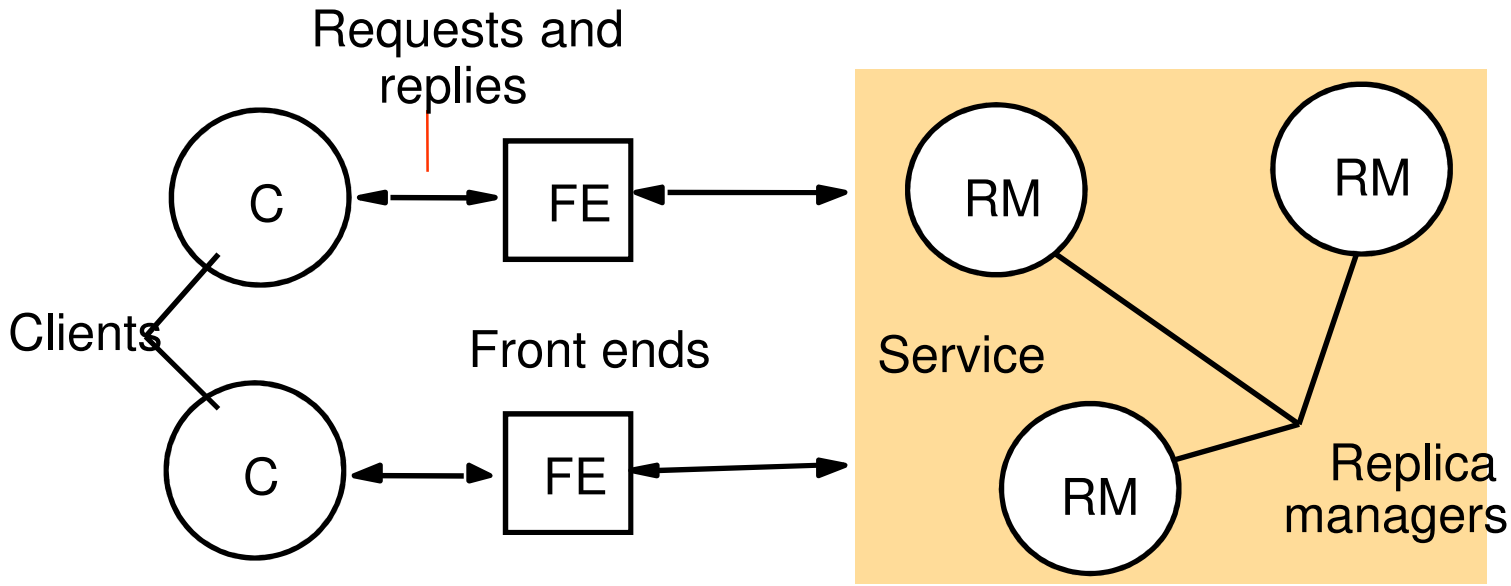
Replication (especially of data) is not for free: it raises additional concerns whose solutions incur costs:

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, 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 basic-ally delineate how far the system may deviate from this ideal.

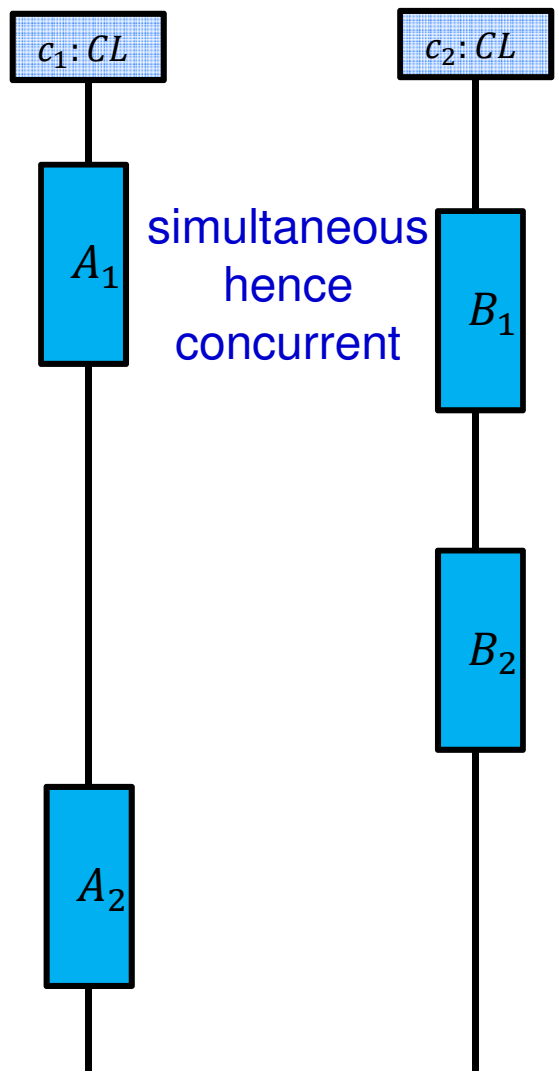
Conflicting operations

- 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
  - write-write conflicts
- 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.



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 from TvS

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.

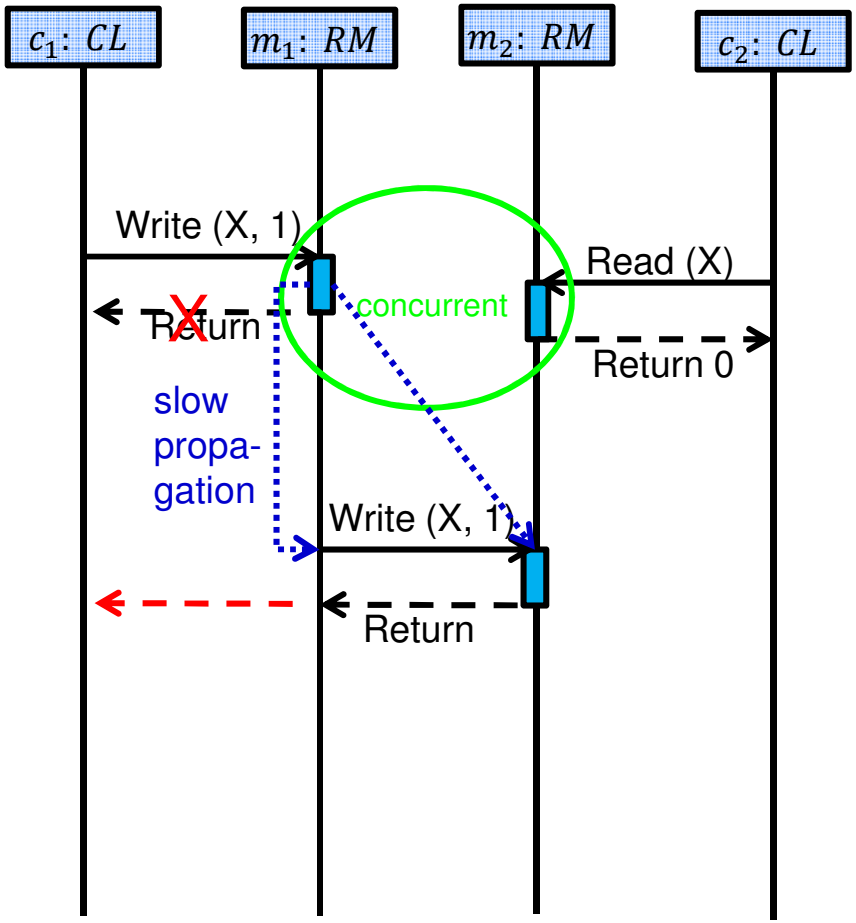
Linearizability

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!

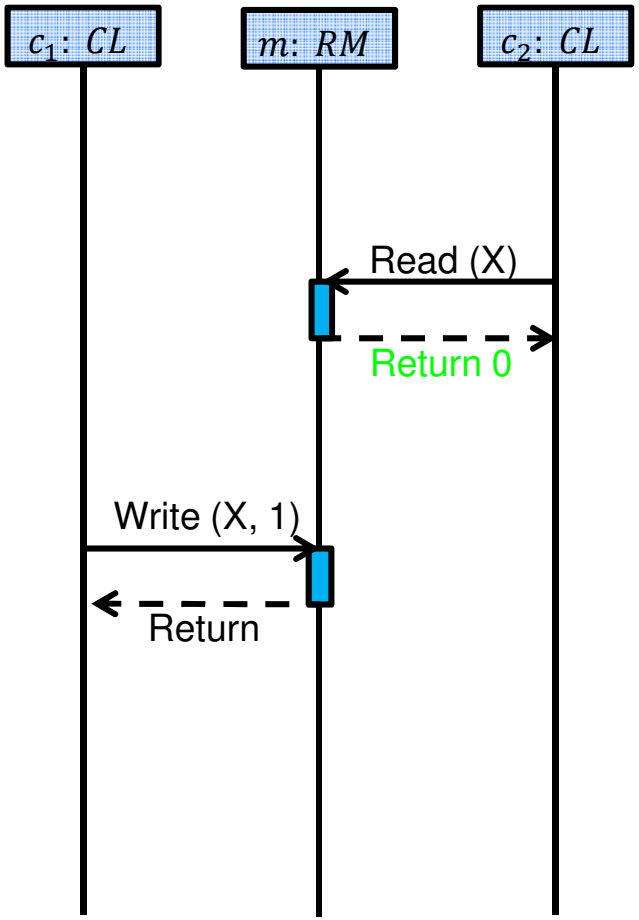
Initially: $X = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



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

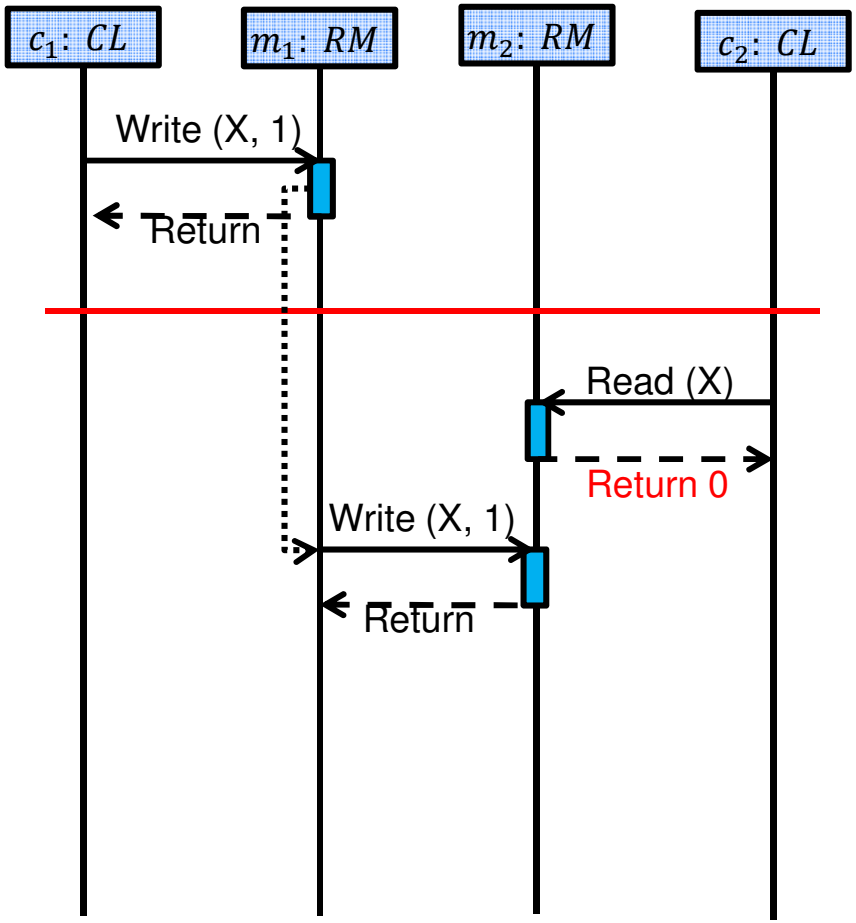
but wrong implementation!

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



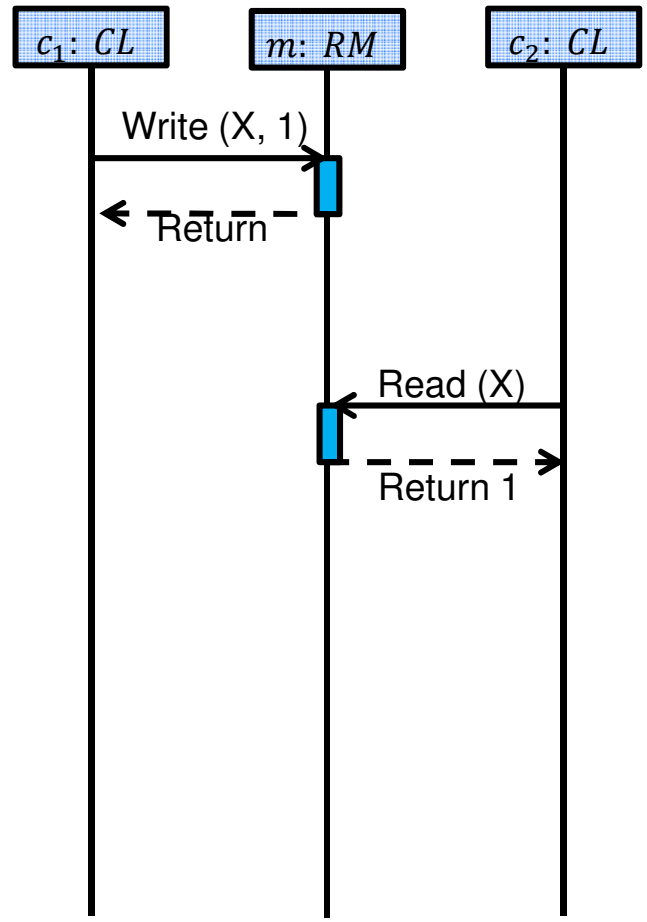
R1 not violated

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



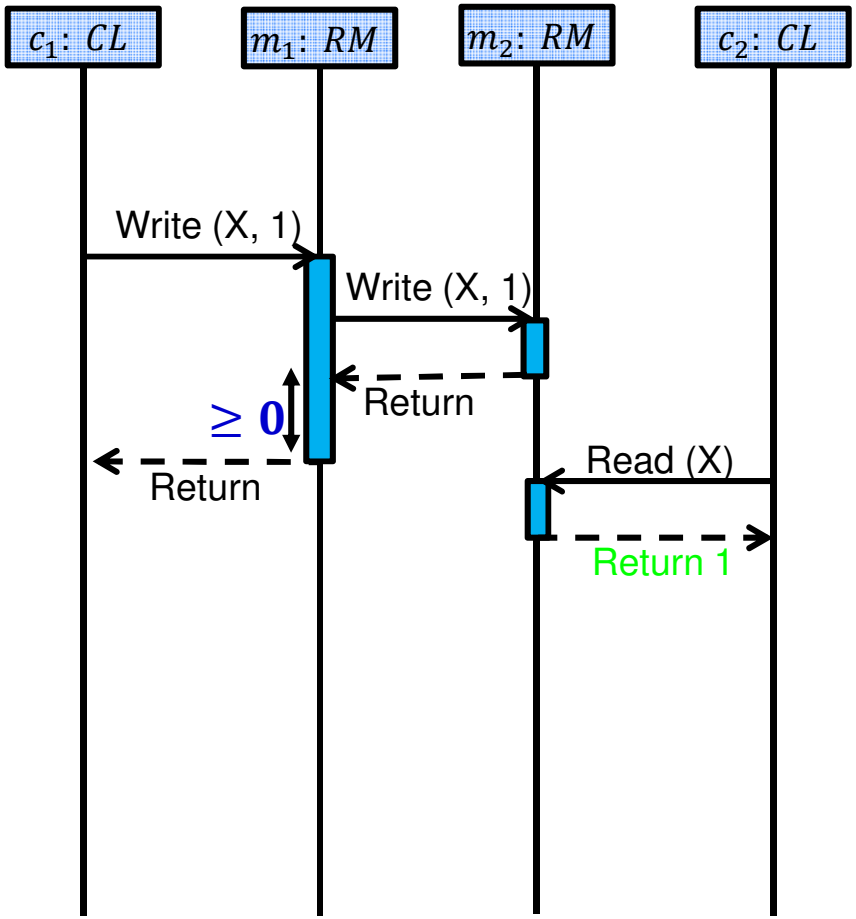
Not linearizable

Single server execution
(the only one allowed by R3)



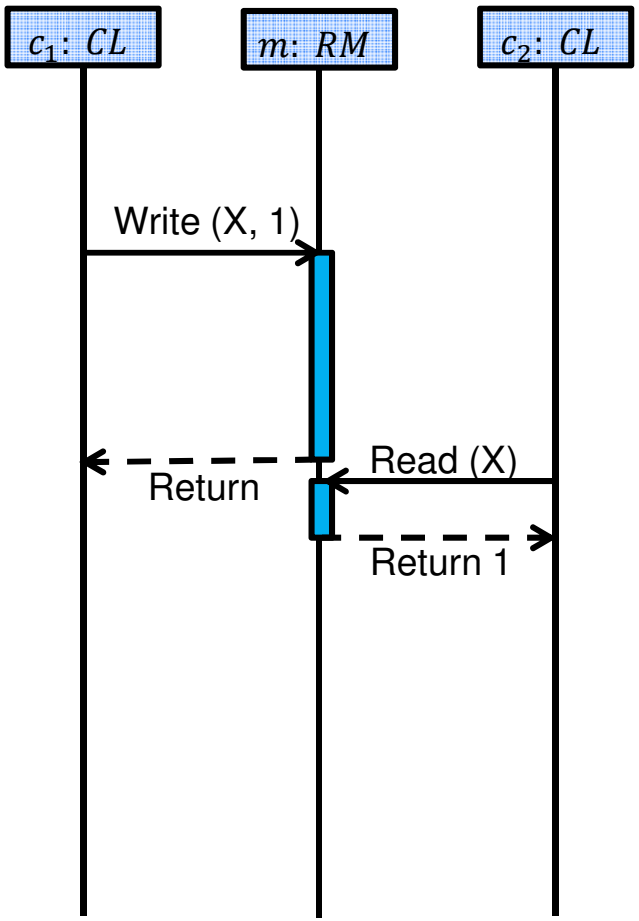
R1 violated

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



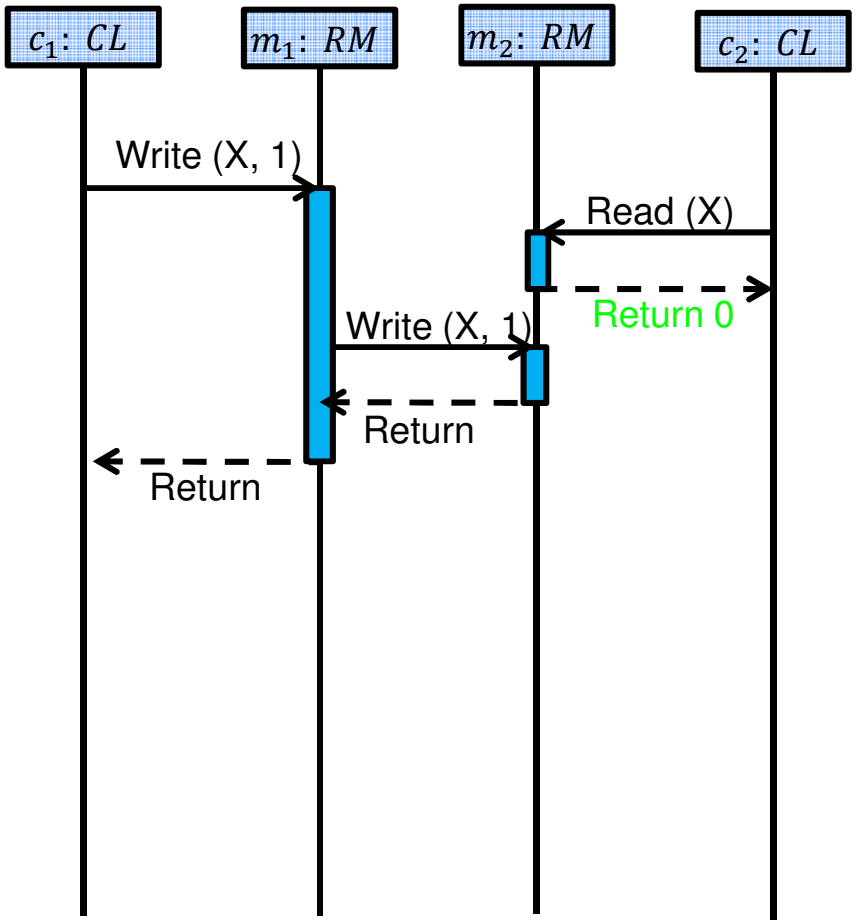
Linearizable (both execution and store)

Single server execution
(the only one allowed by R3)



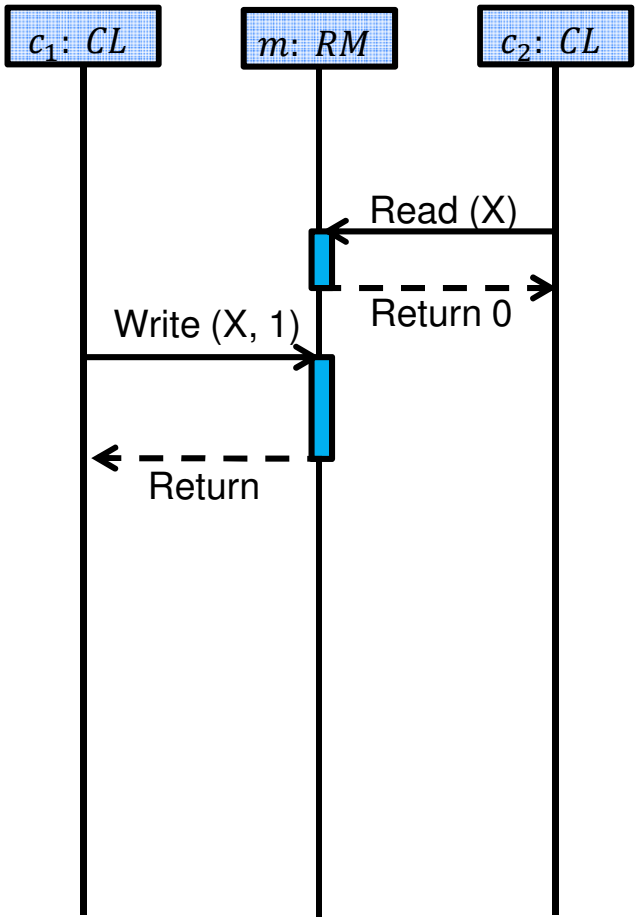
R1 not violated

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



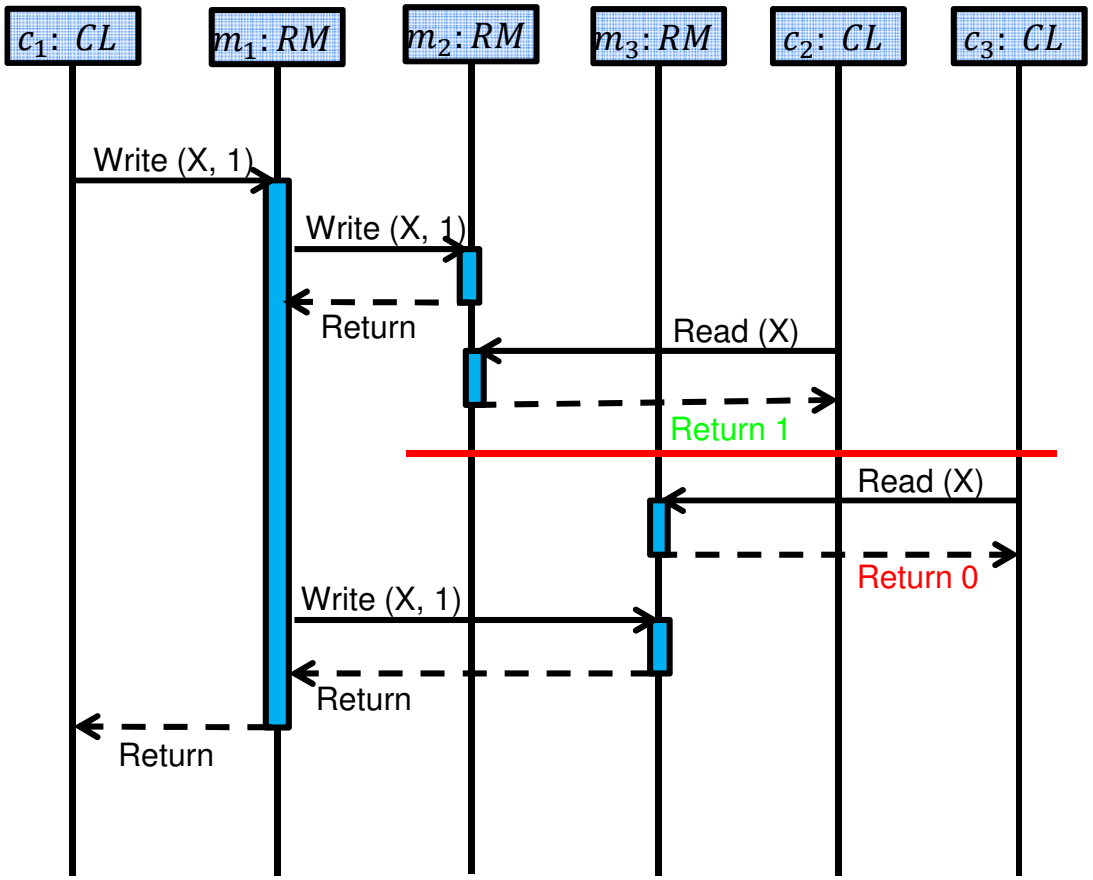
Linearizable

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



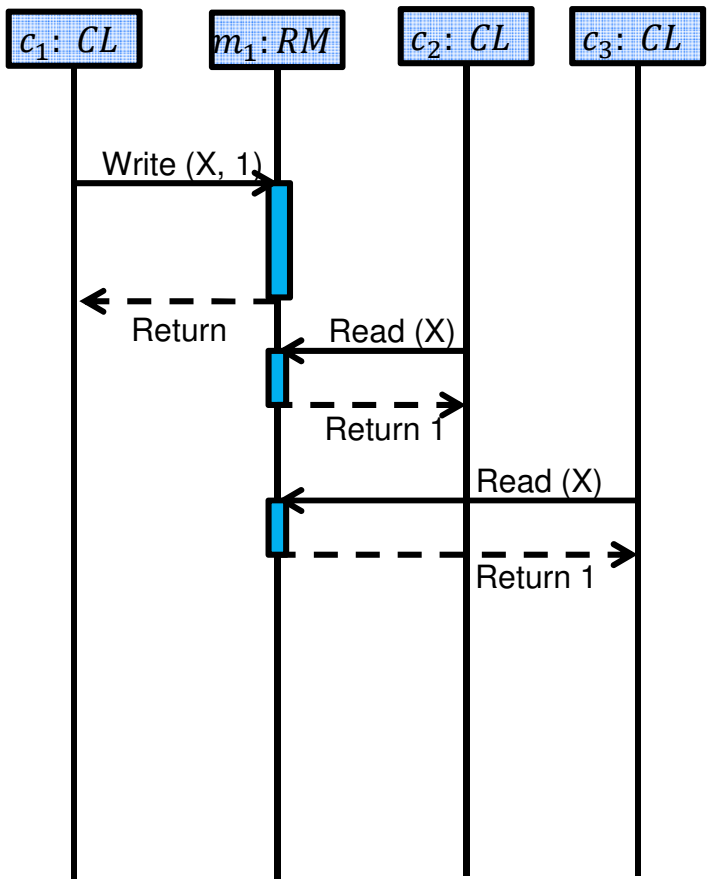
R1 not violated

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



Not linearizable

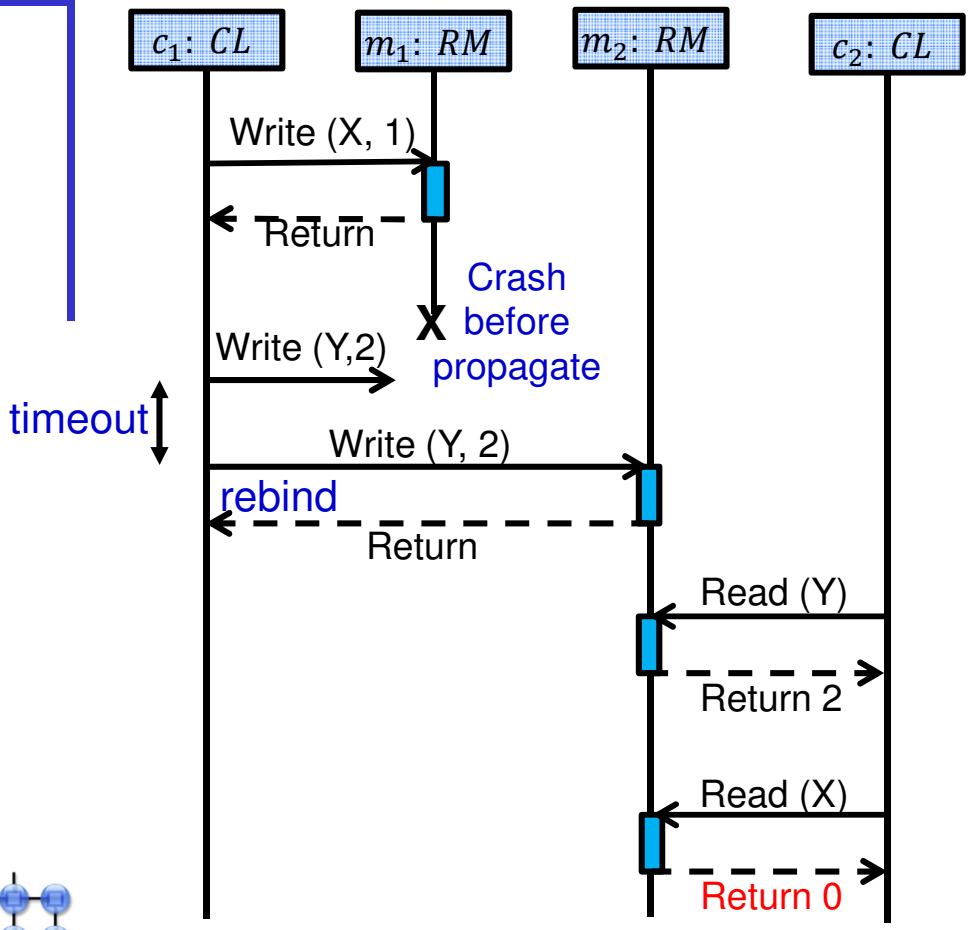
Single server execution (1 out-of 3) allowed by R3. None allows reading 1 twice.



R1 violated

Logical variables X and Y

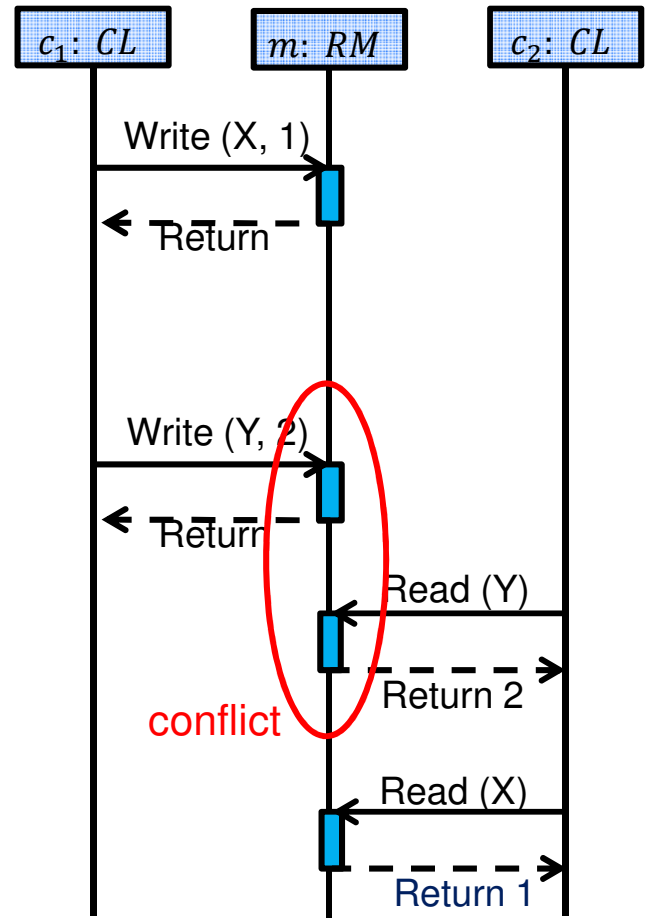
Initially: $X = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



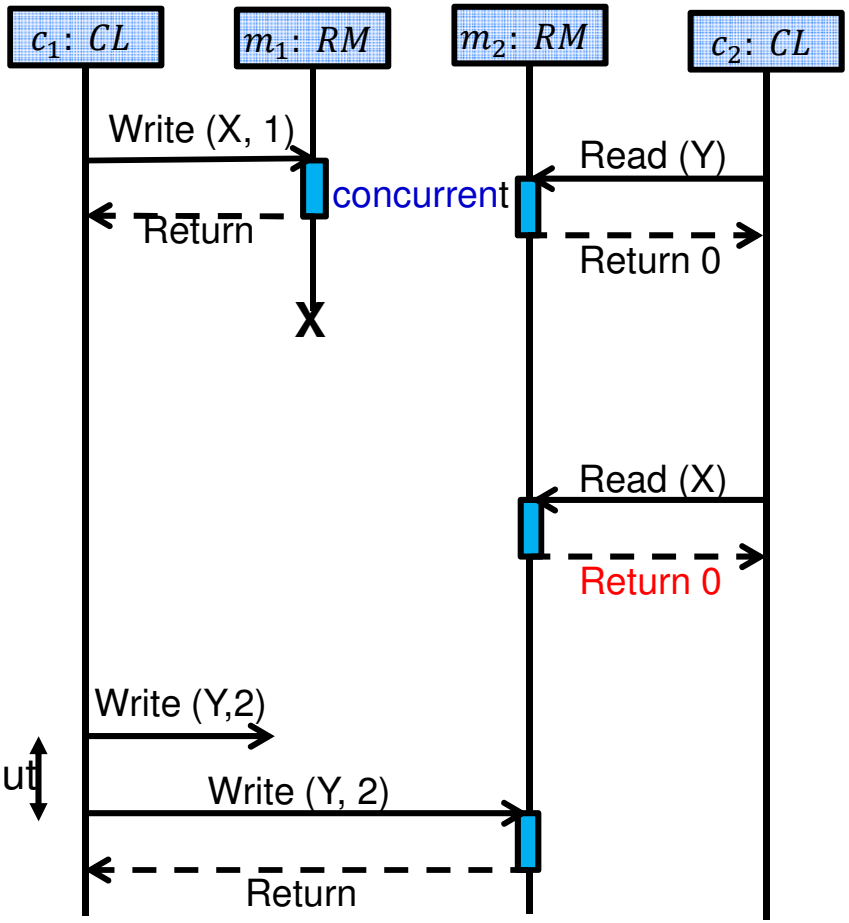
Not linearizable

see also slide 29

Single server execution
(the only one satisfying R3)

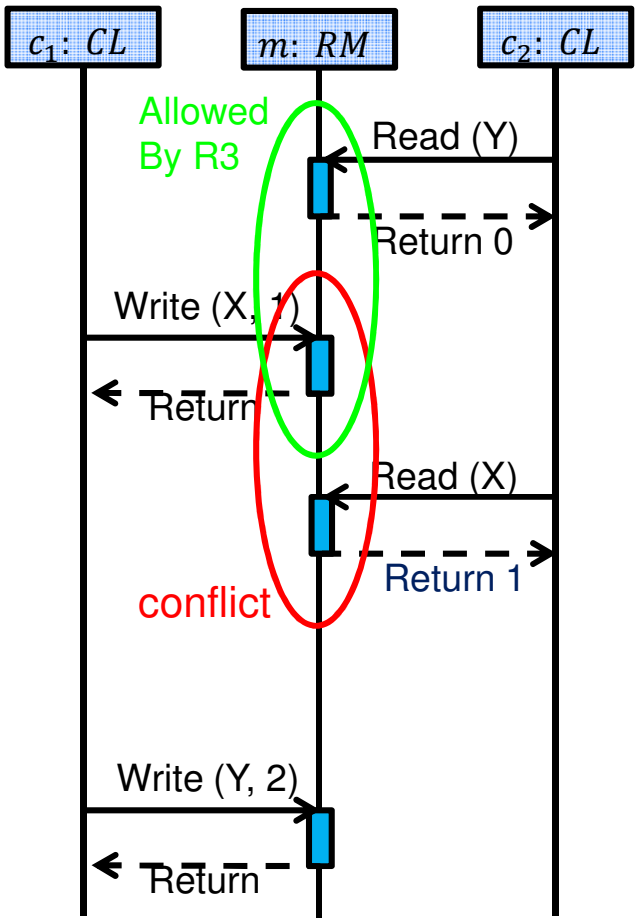


Initially: $X = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$

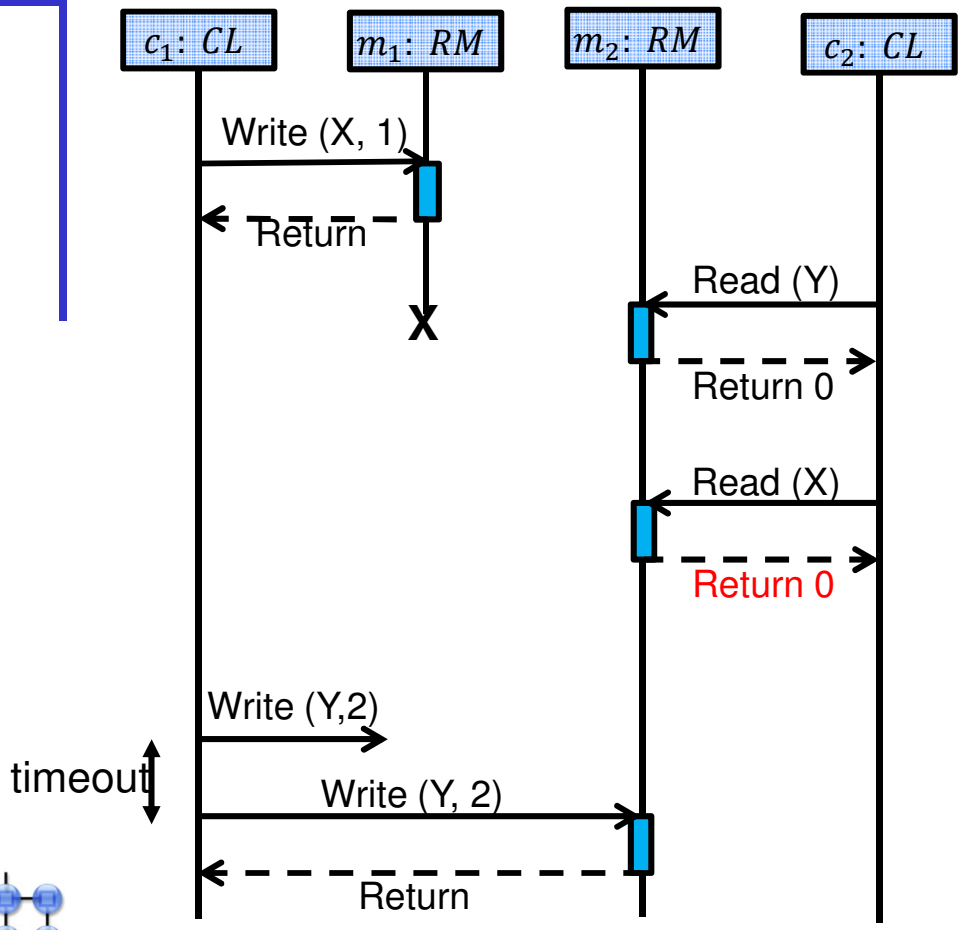


Not linearizable

Single server execution
1 out-of 2 allowed by R3

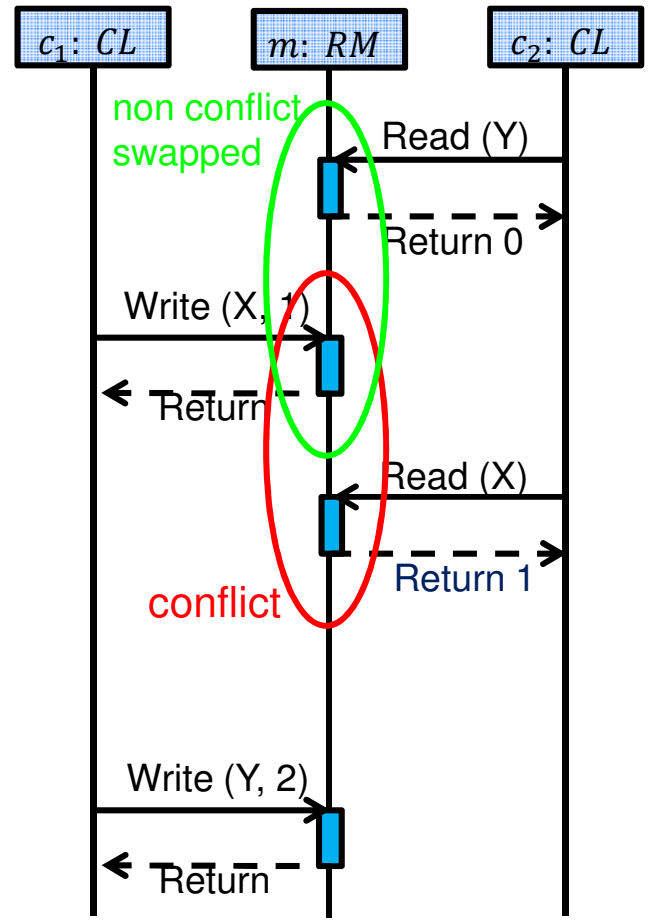


Initially: $X = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



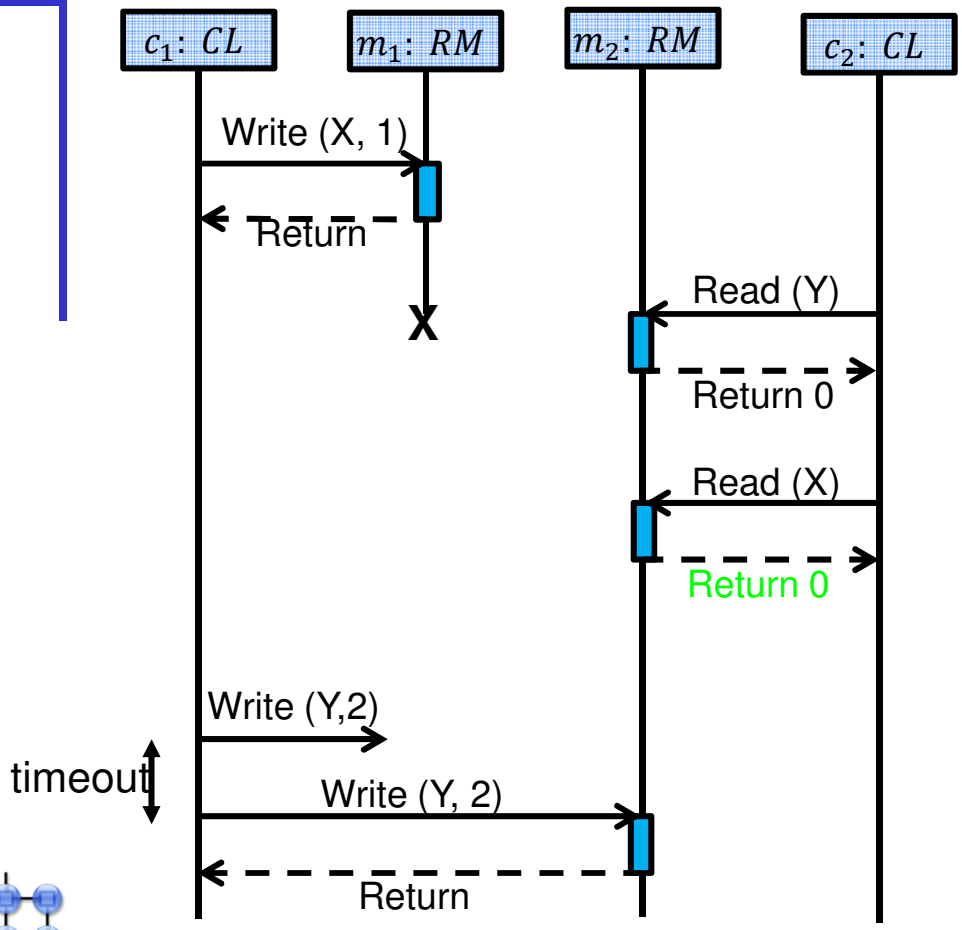
Not linearizable

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



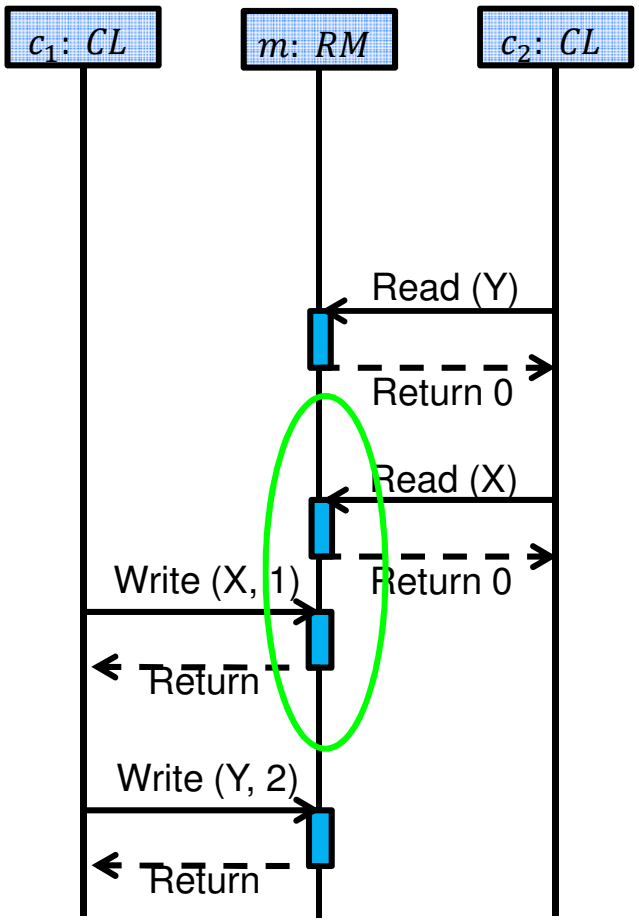
not linearizable

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



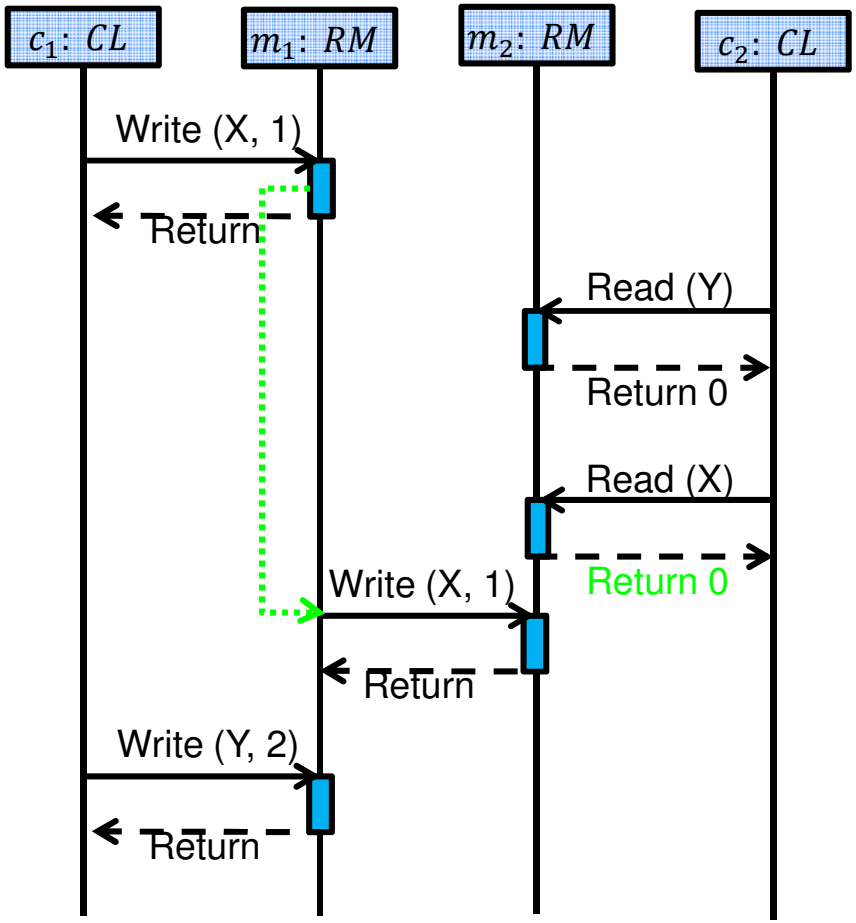
sequential consistent
early return for Write(X,1) is OK in this execution

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



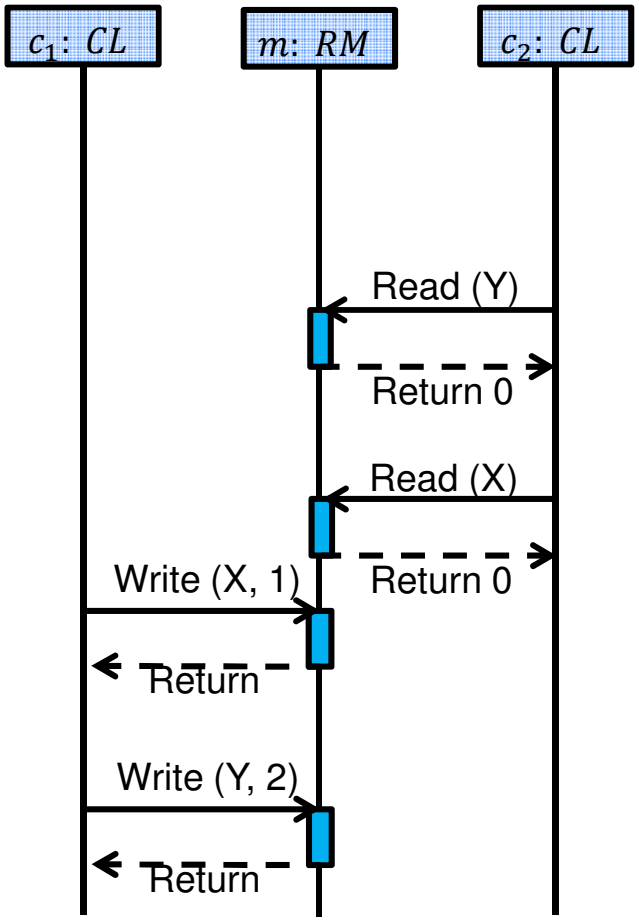
conflicting operations swapped (allowed)

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



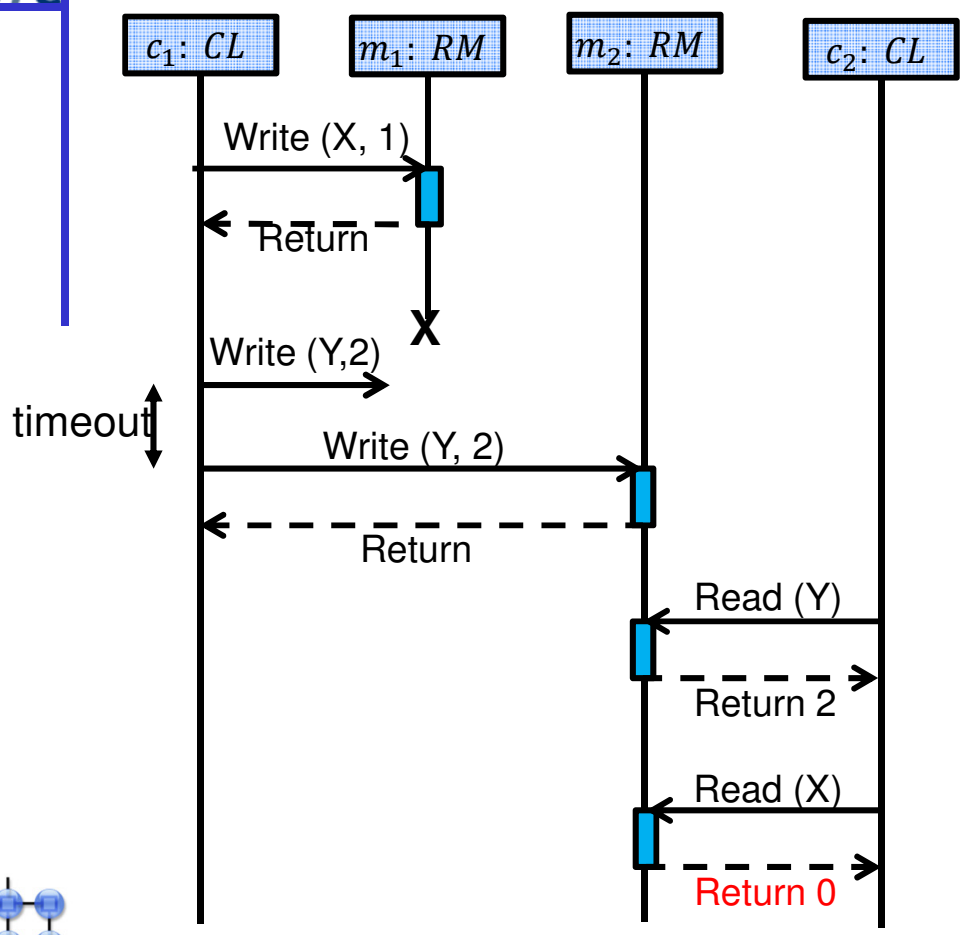
Can also happen without failure
Just slow propagation

Single server execution



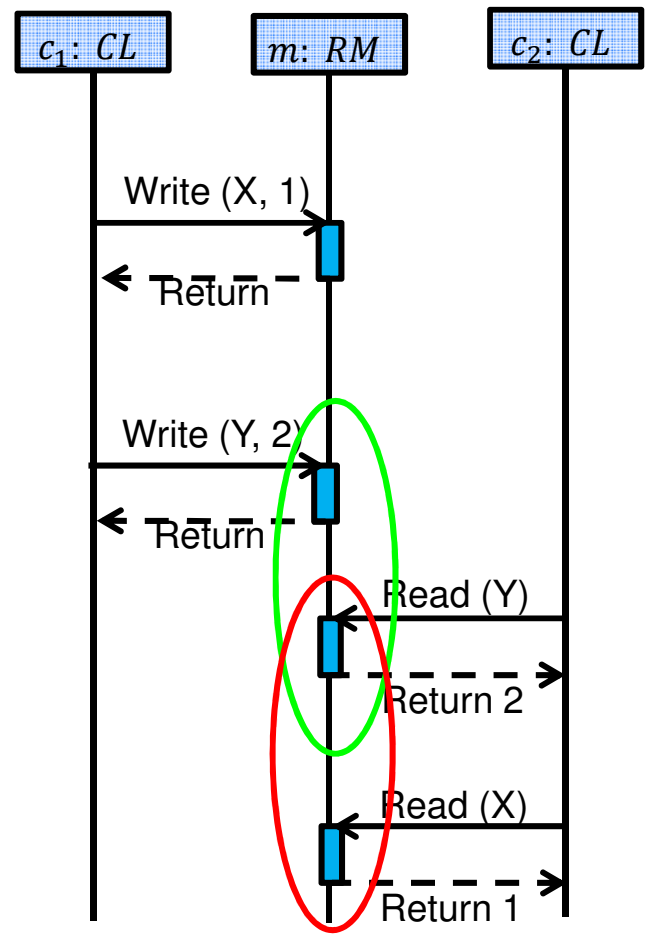
Reordering reflects arrival time at replica manager m_2

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

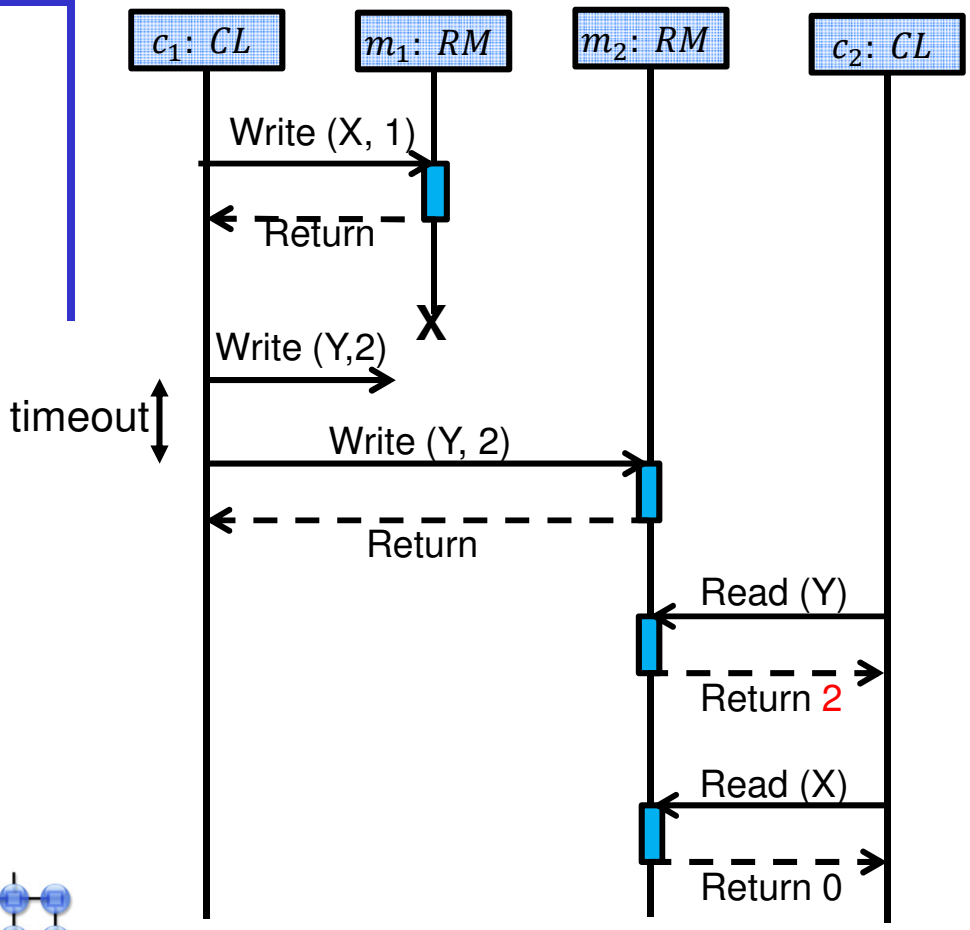


Not sequential consistent

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

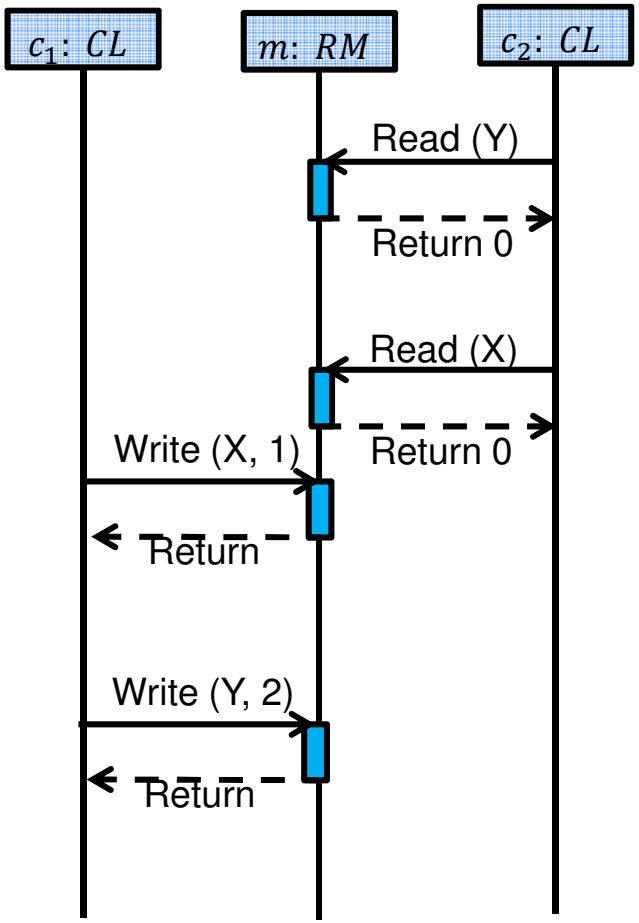


Initially: $X = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



Not sequential consistent

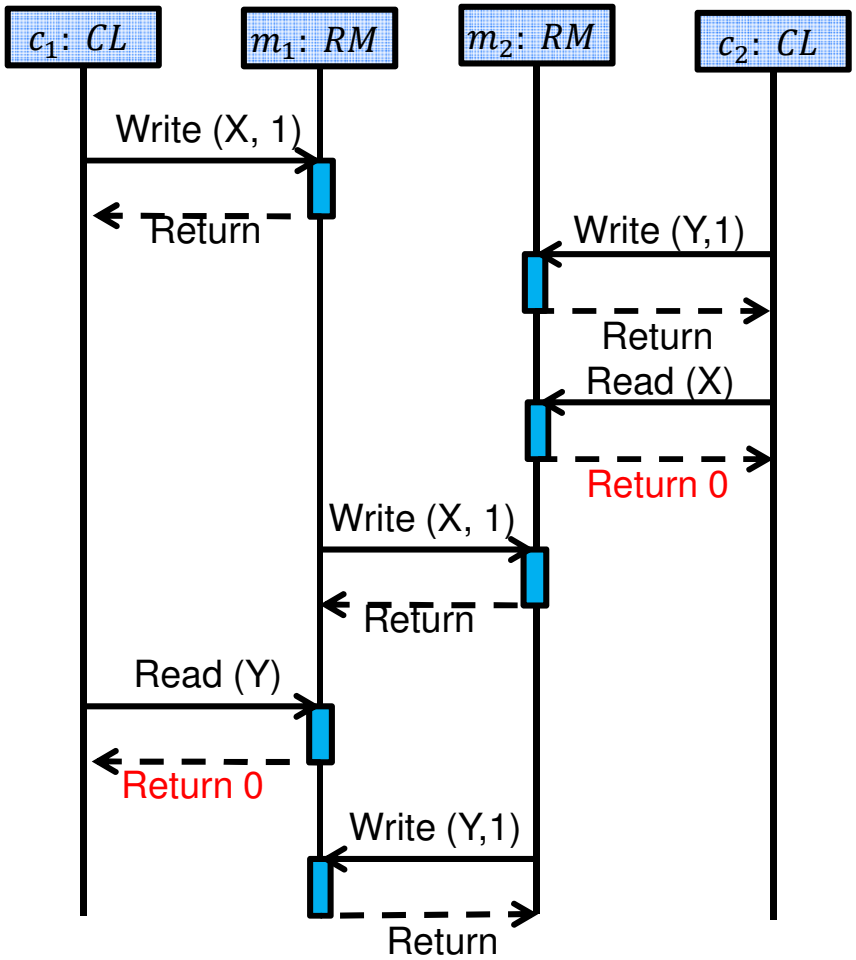
Single server execution
1-out-of 6
Reordering does not help



X-value ok, but
now Y-value wrong

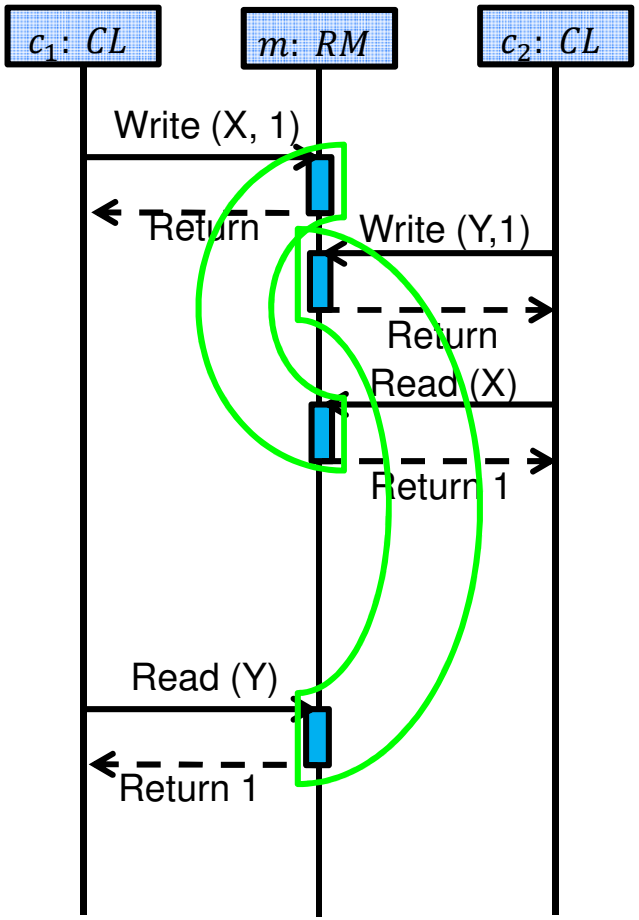
Updates from distinct clients

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

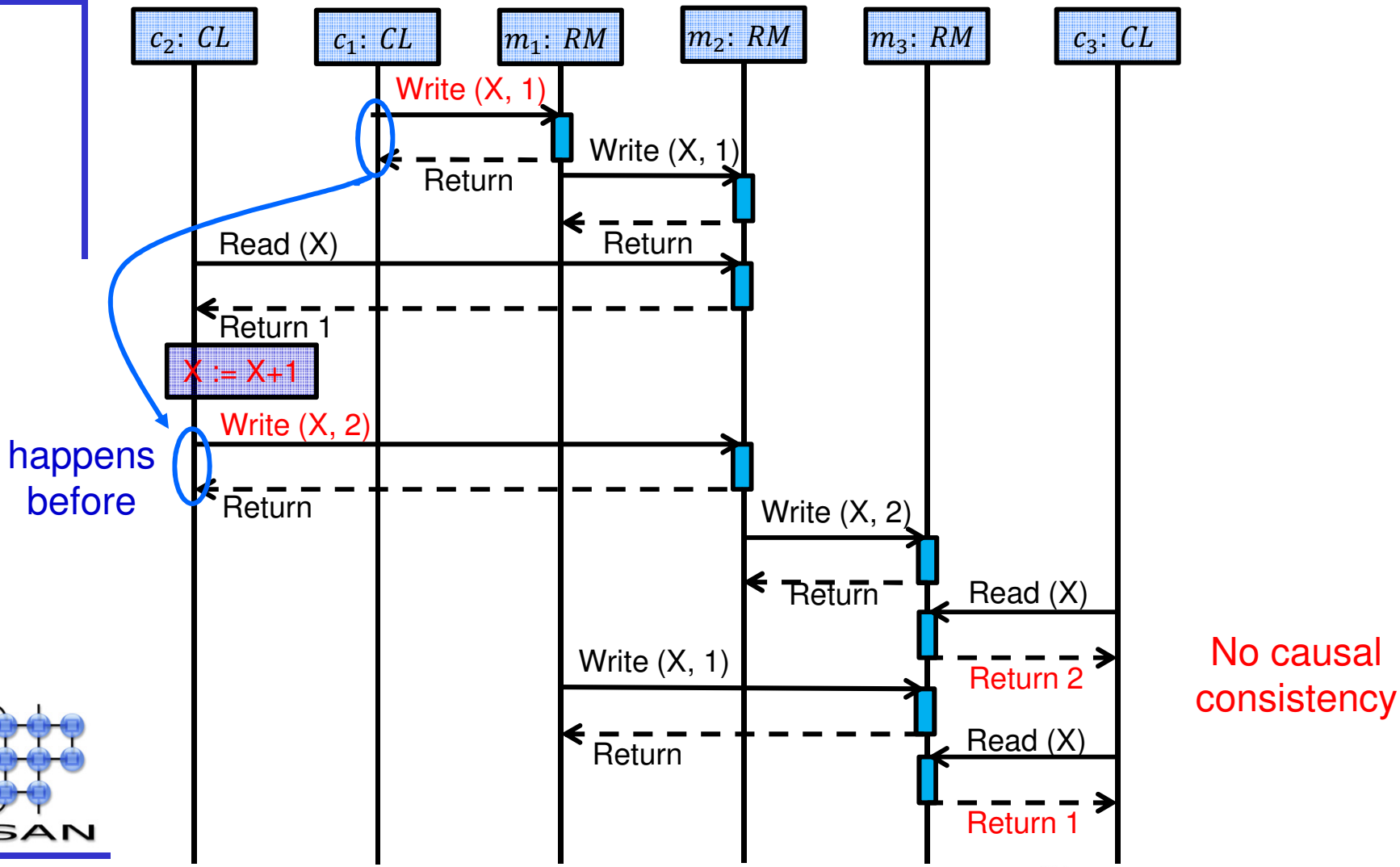
Other forms of consistency

- 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.

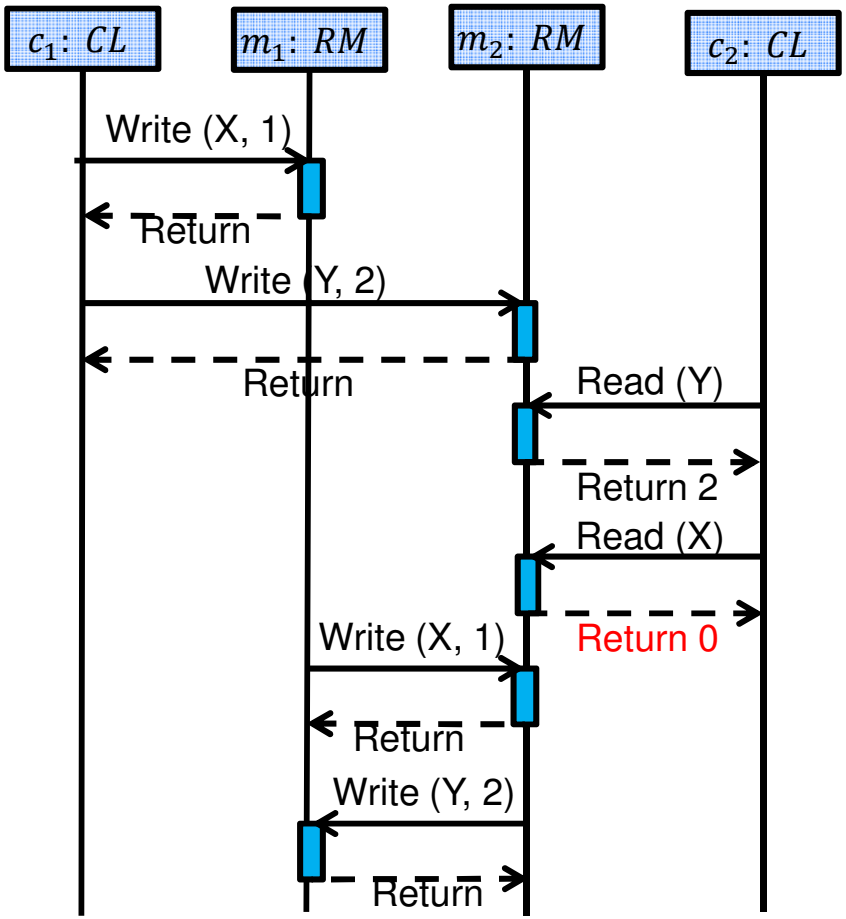
Beware: RM m_2 is superfluous. Only present to show *chain* of causal dependence

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



No causal consistency

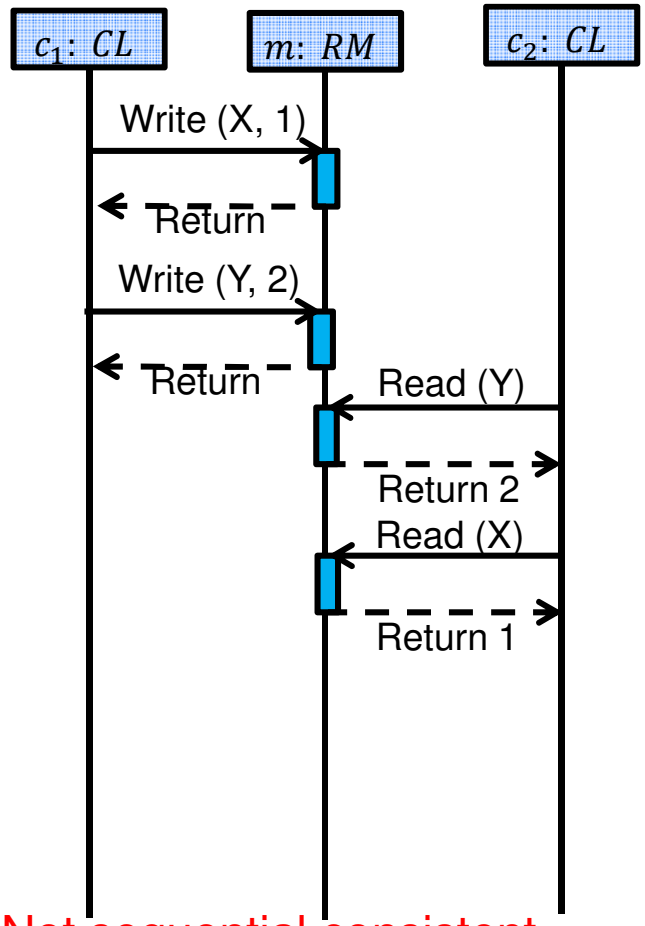
Initially: $X = \langle 0,0 \rangle \wedge Y = \langle 0,0 \rangle$



Eventually consistent

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

Single server linear execution

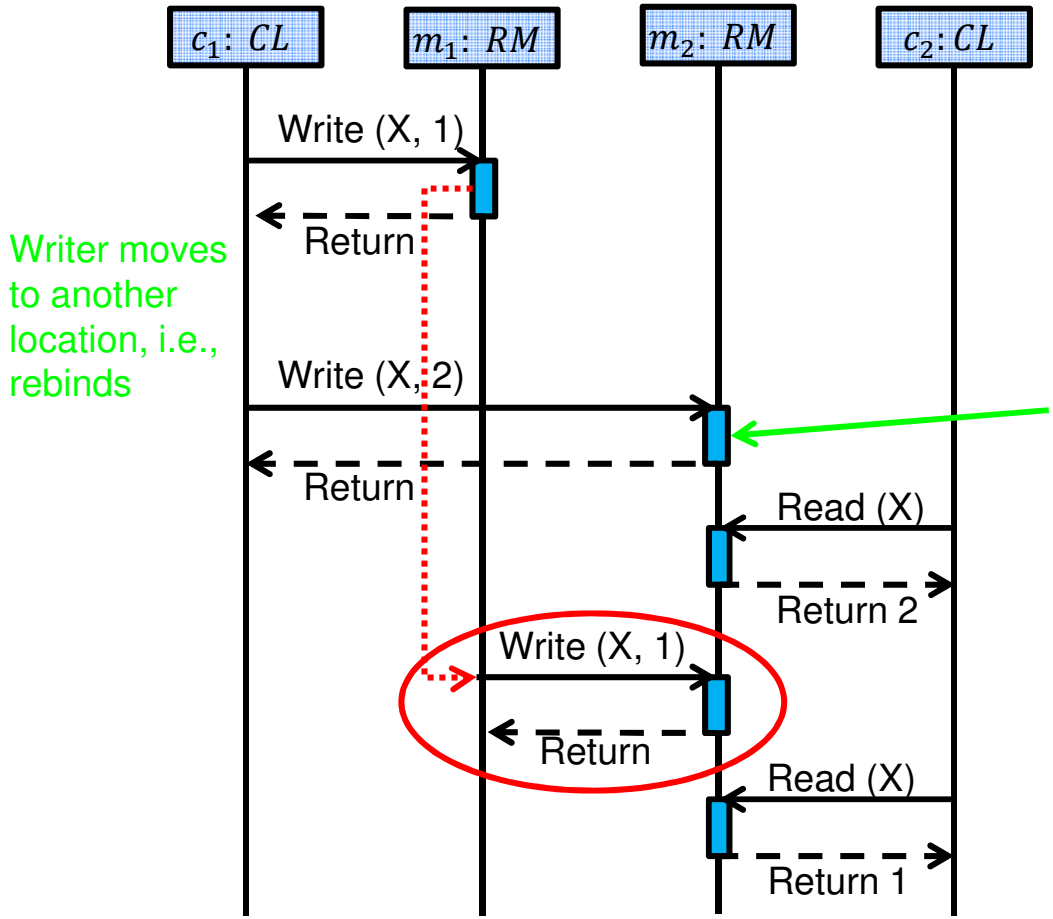


Not sequential consistent.
Swapping to get X right,
makes Y wrong

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.

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



Writer moves to another location, i.e., rebinds

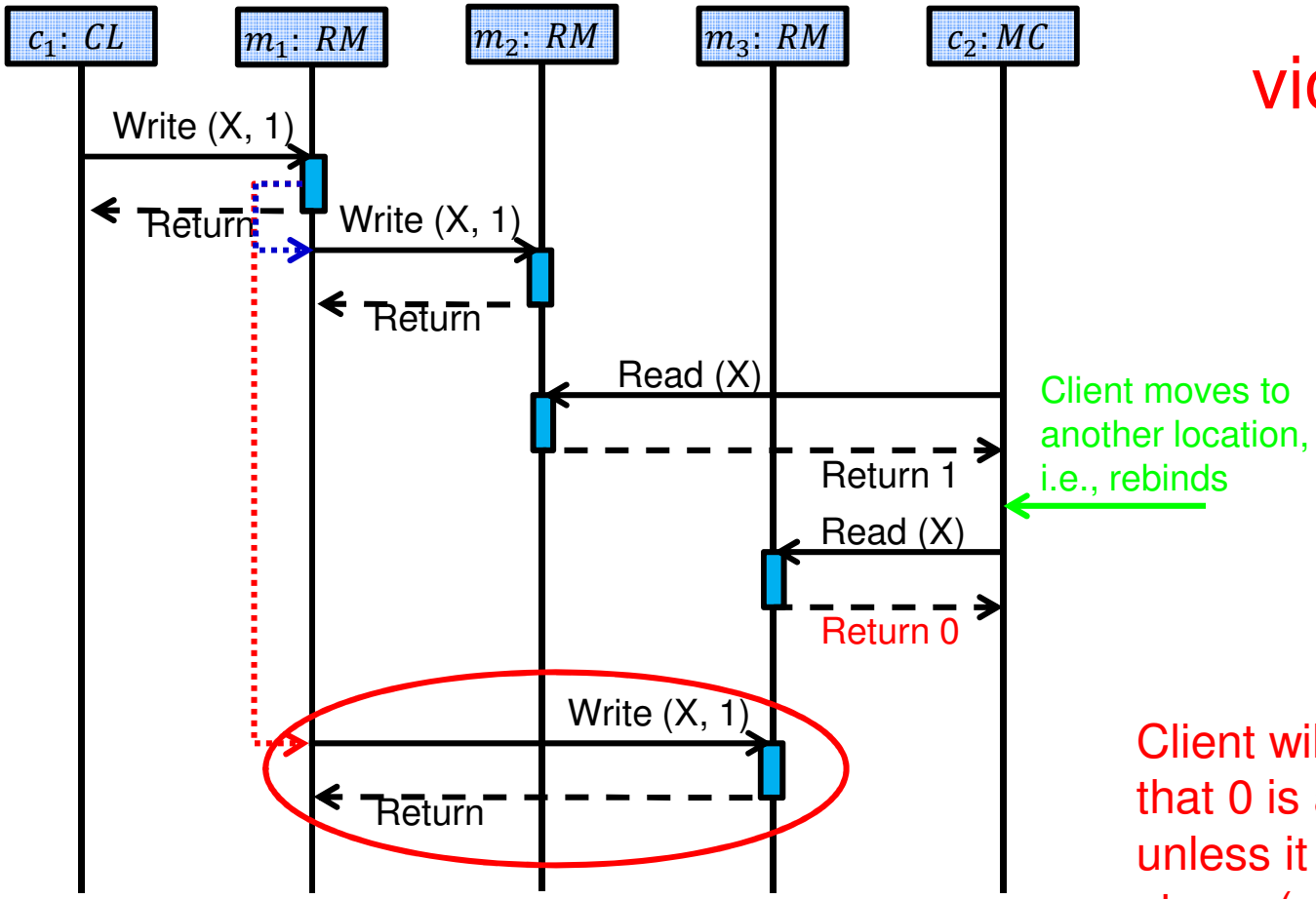
or written by another client, but then global time stamps are needed to determine whether it is more recent

Monotonic read violation at c_2

TOO LATE delayed propagation

Monotonic read violation

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

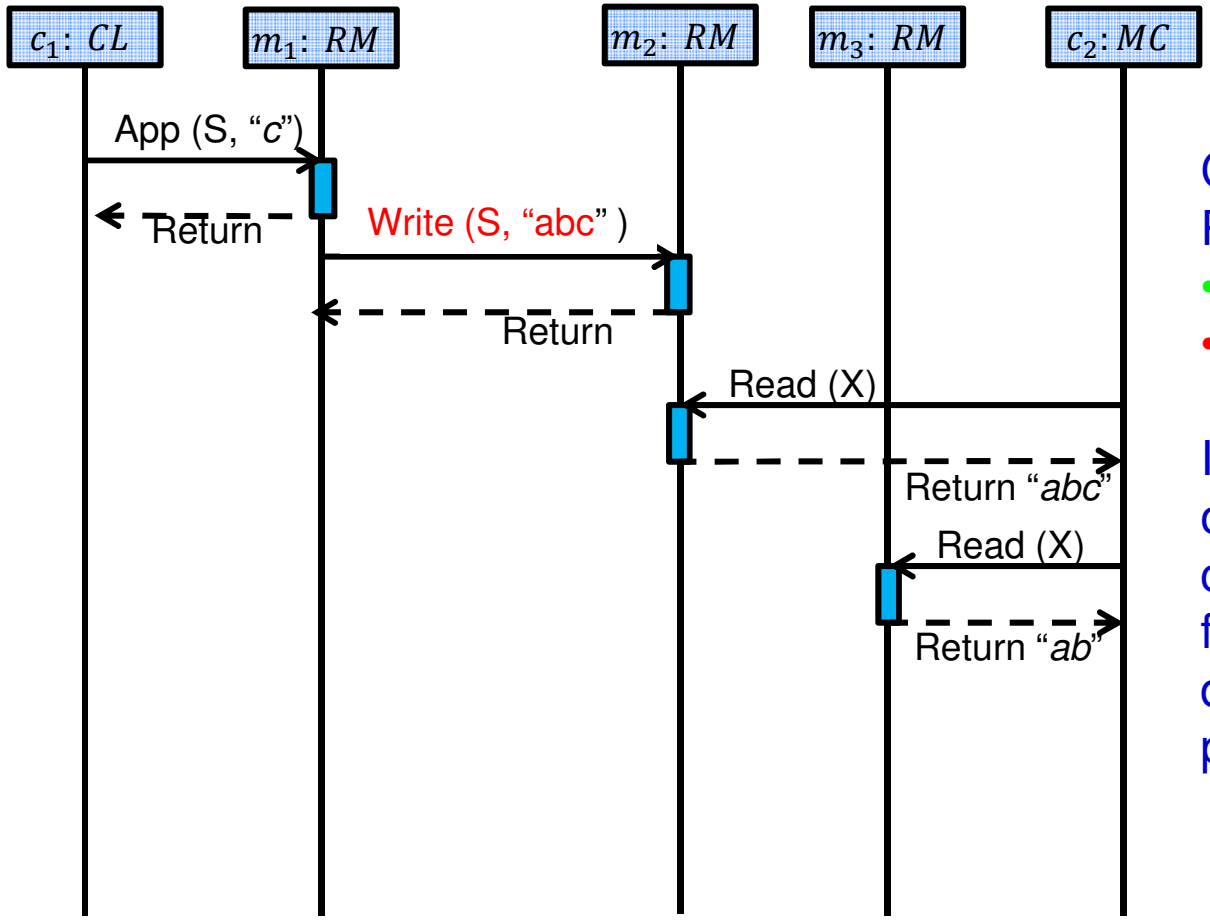


Client moves to another location, i.e., rebinds

TOO LATE
lazy propagation

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

Initially: $S = \langle "ab", "ab", "ab" \rangle$



Clients can only append, RMs have two options

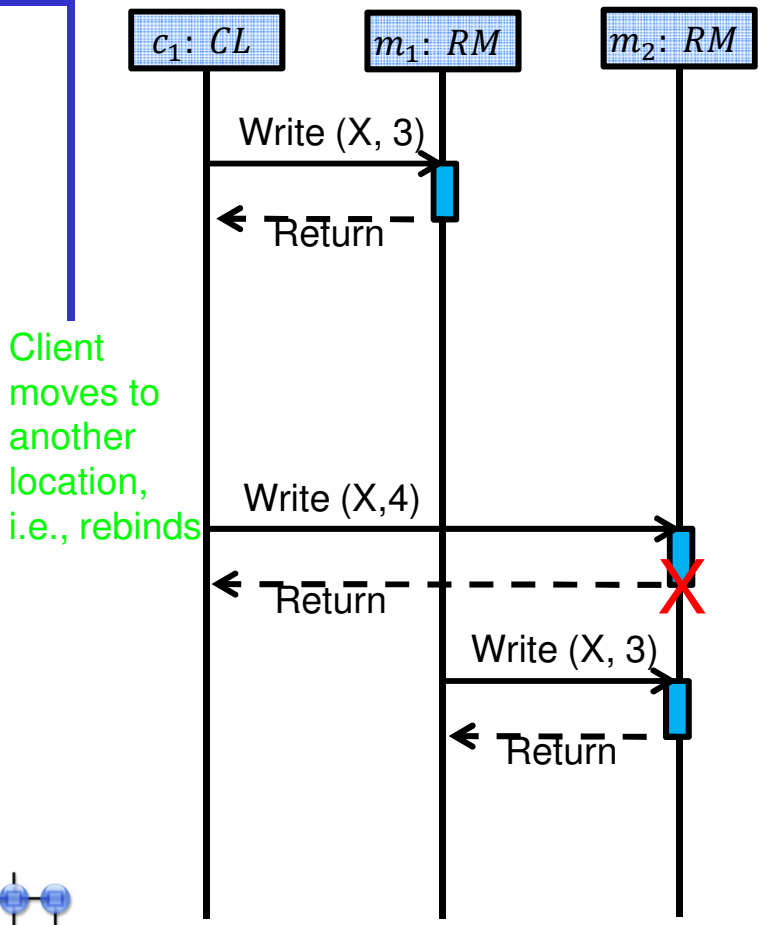
- forward the operation
- forward its result

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

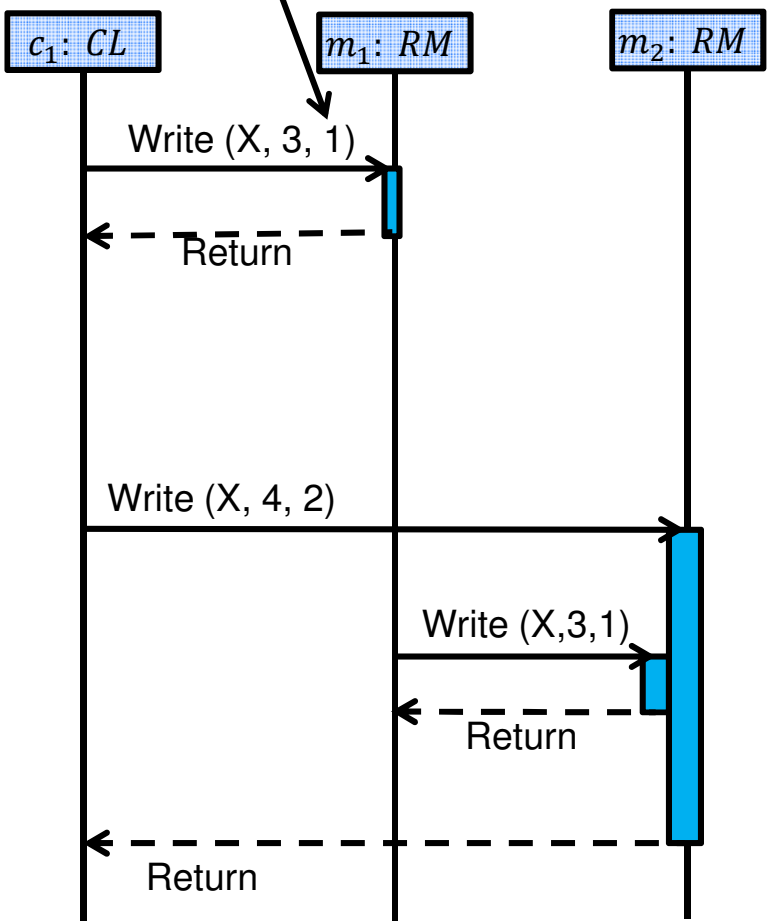
- Monotonic read consistency
  - If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more *recent* one.
- Monotonic write consistency
  - A write operation on a data item x is *completed* before any successive write 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: $X = \langle 0, 0 \rangle$



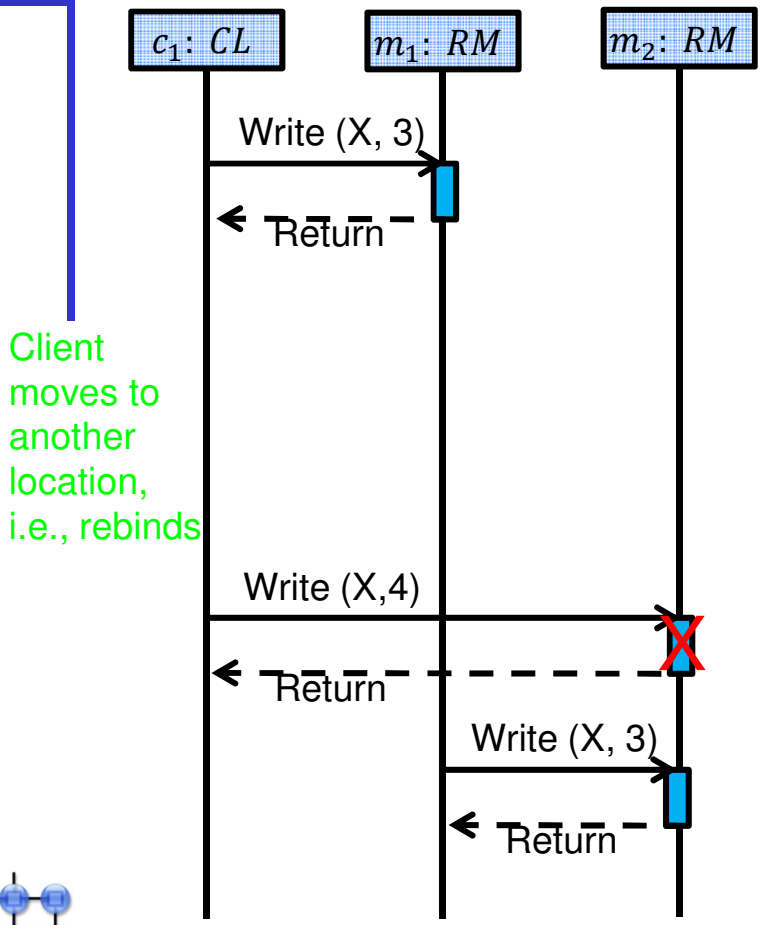
Monotonic-write violation

version number

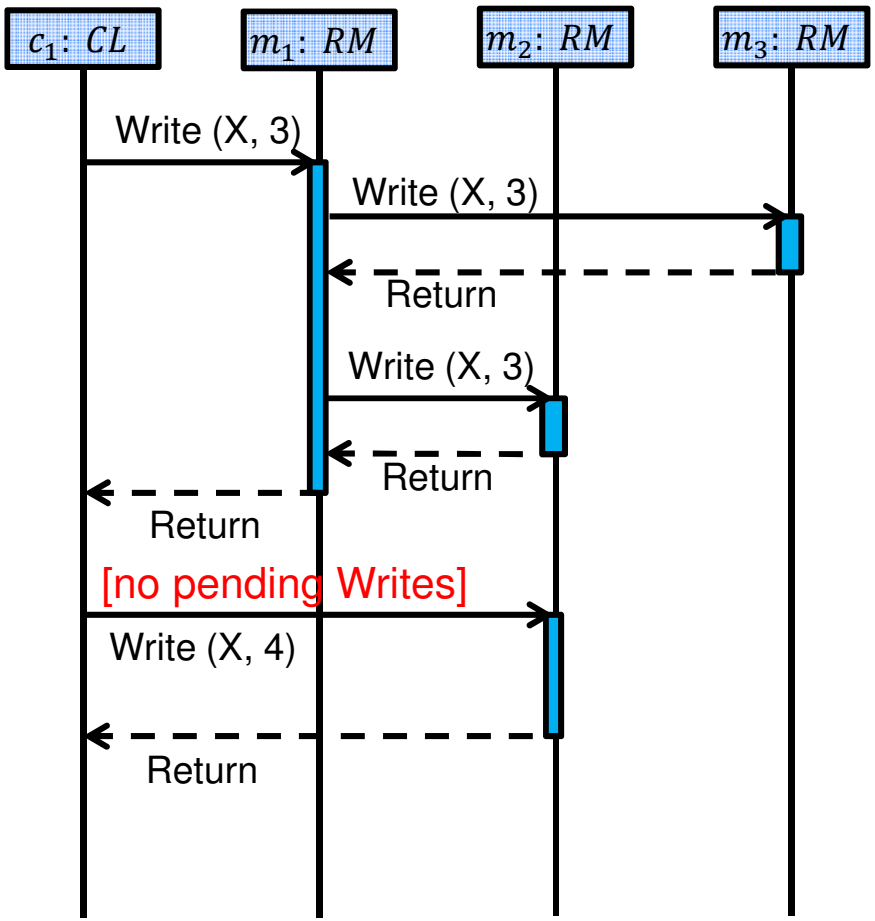


Can be prevented by versioning updates

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



Monotonic-write violation



Can be prevented by making writes atomic

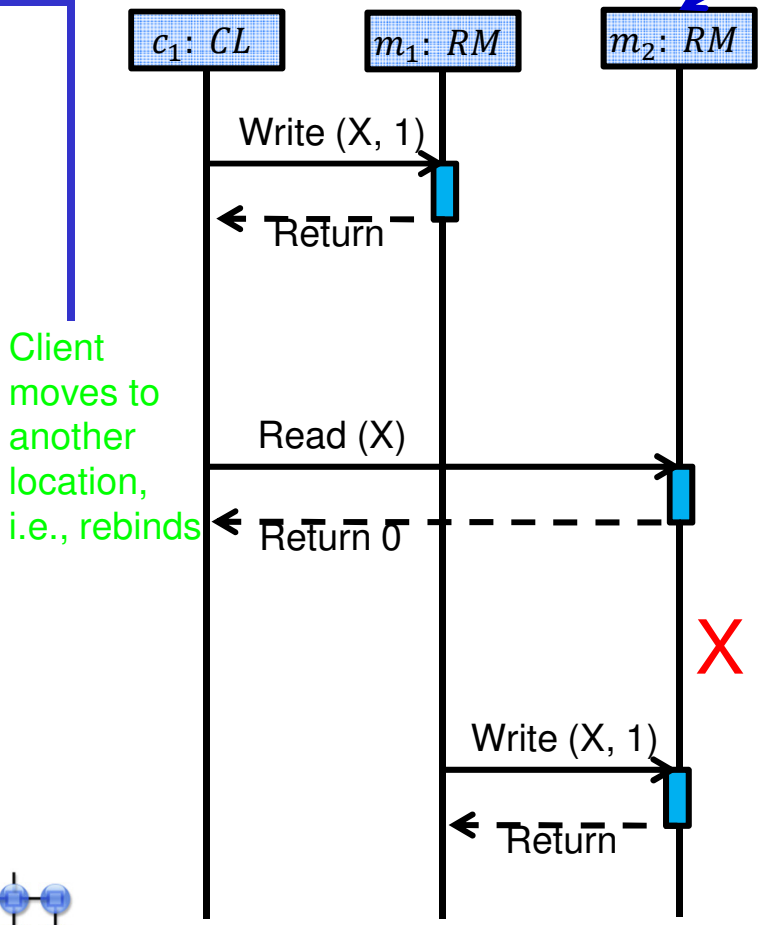
Client-centric models

- 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

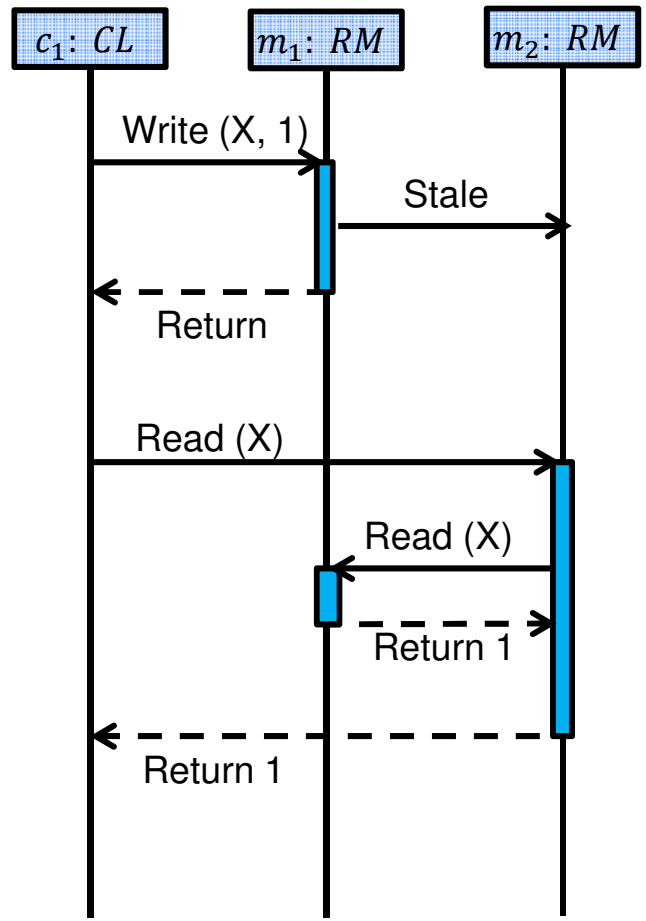
Initially $\langle 0, 0 \rangle$

m_2 can be, e.g, a stale cache

X a private variable of c_1



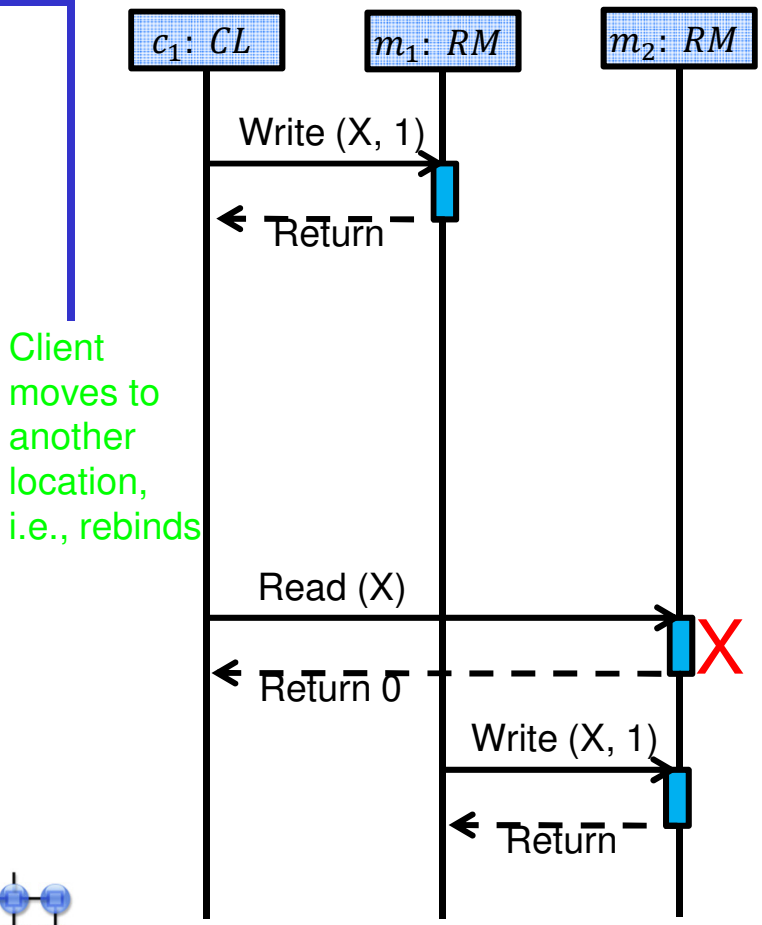
Read-your-writes violation



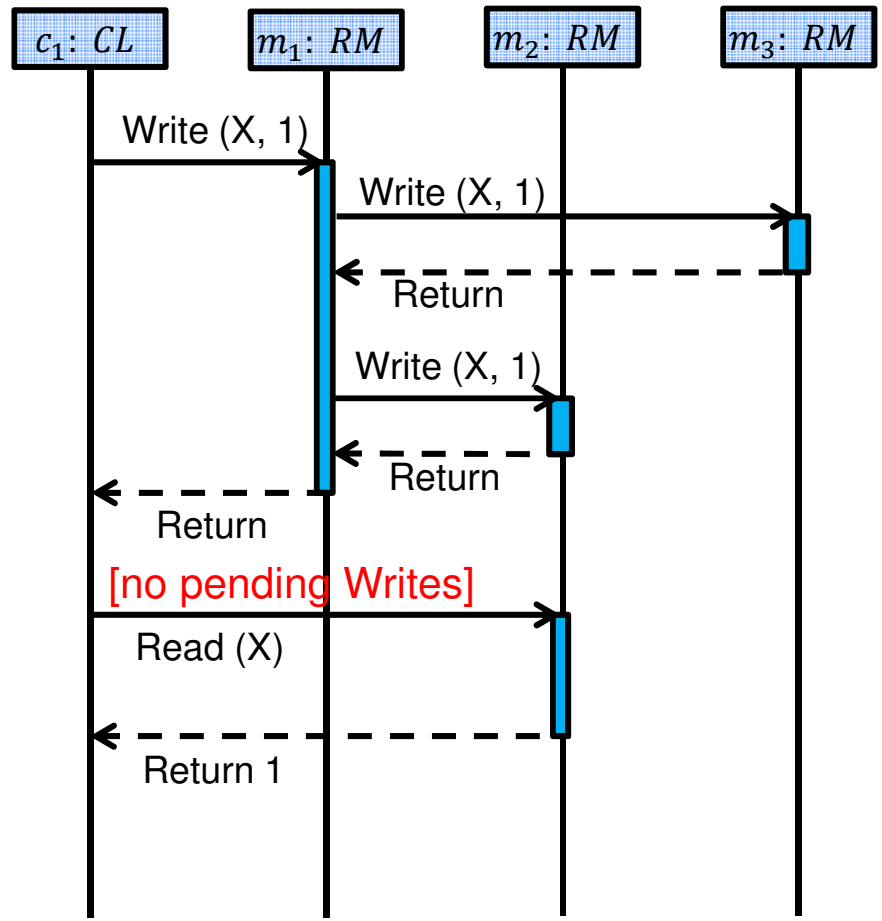
Can be prevented by stale notification
Can also be prevented by making writes atomic

Initially $\langle 0, 0 \rangle$

X a private variable of c_1



Read-your-writes violation

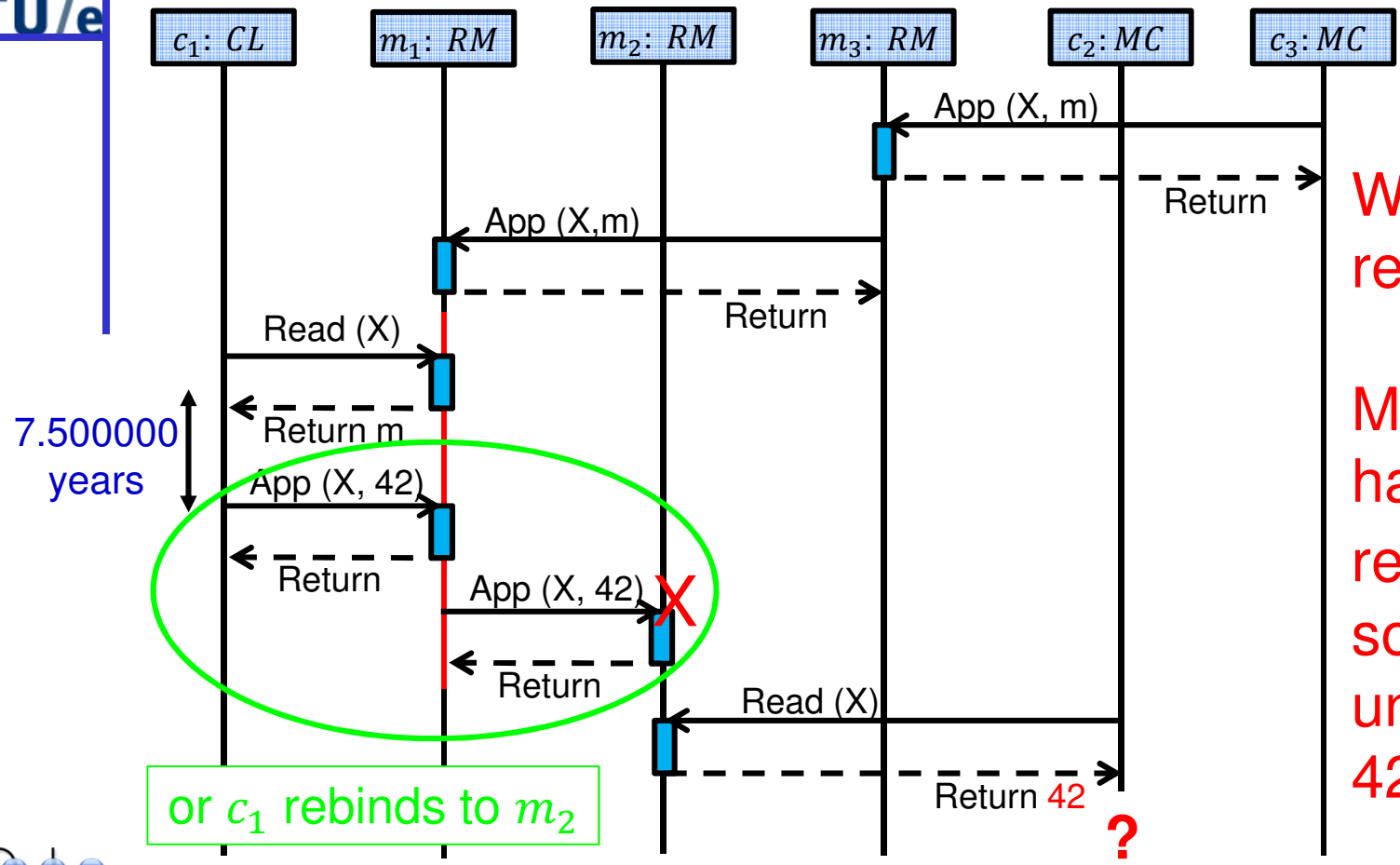


Can be prevented by making writes atomic

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: $X = \langle "", "", "" \rangle$



Write follows read violation

Message m has not been read by c_2, so it does not understand 42

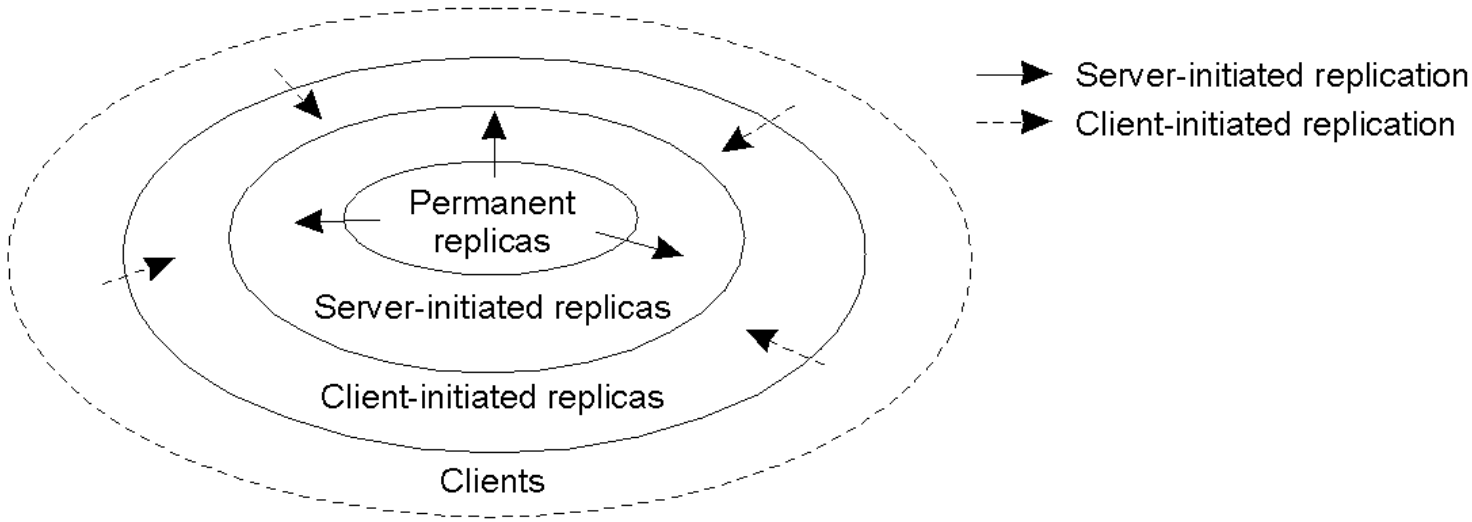
Of course m is the "Ultimate Question of Life, the Universe and Everything"!!!
See: Douglas Adams, The Hitchhikers Guide to the Universe

Agenda

- Introduction
- Consistency models
- Replica management
  - Placement strategies
  - Update protocols
  - Caching
  - Fault tolerance and recovery strategies
- Architectural case studies

Placement

- Server placement (the replica managers)
  - Based on geographical positions or internet topology
  - Important for data-owners, service providers
  - Organized in sites (data centers)
- Content placement (the replicas)
- Static versus dynamic



Replica placement

- Permanent replicas
  - Limited, fixed, statically determined number of replicas
  - Presumes a stable and a-priori known load distribution
  - Increased reliability and performance
- Server-initiated replicas
  - Dynamically changing number of replicas
  - To cope with server peak loads; increased availability
- Client-initiated replicas (aka client caches)
  - To improve response time by improving data proximity; to reduce network load
  - No persistent storage; data has limited life-time
  - Client is responsible for consistency; although the server may assist

Protocol classification

- What information is disseminated upon an update operation
  - The operation (active), the resulting state (passive), a notification
- Which party takes the initiative for dissemination upon an update operation
  - Push-based (aka server-based), pull-based (aka client-based)
- What communication primitives are used to disseminate an update operation
  - Unicast, multicast, gossiping
- When is completion of an operation reported to the client (i.e., the data store commits the operation)
  - Immediate on receipt, after atomic execution, quorum-based

Together these determine the supported consistency model

Pull versus Push Protocols

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

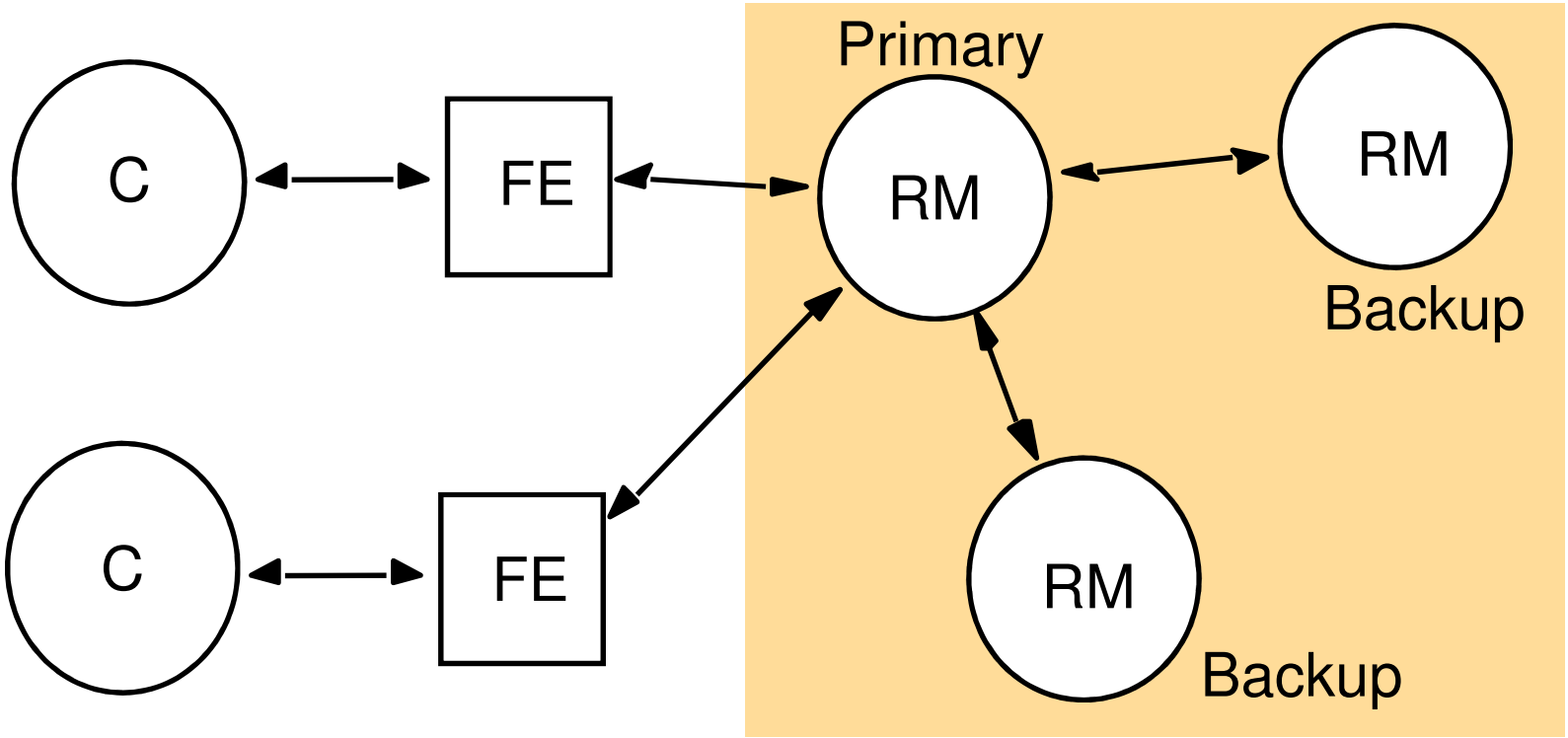
Protocol classes

- Passive replication protocols
- Active replication protocols
- Gossip-based protocols

Passive replication protocols

- Primary-read, primary-write
  - seldom-used
  - supports linearizability; clients perceive atomic reads and writes.
  - blocking reads and writes, no concurrency, limited availability
- Local-read, primary-write (TvS remote-write)
  - supports sequential consistency
  - blocking writes, non-blocking concurrent reads
  - bounded staleness, at most one write missed
- Local-read, local-write (TvS local-write)
  - supports eventual consistency
  - no blocking operations, 100% availability (w.r.t. protocol not failures)
  - incomplete description in TvS
- All protocols are pushed-based w.r.t. to update values
  - local-write protocol pulls the “primary status”.

Figure 15.4
The passive (primary-backup) model for fault tolerance

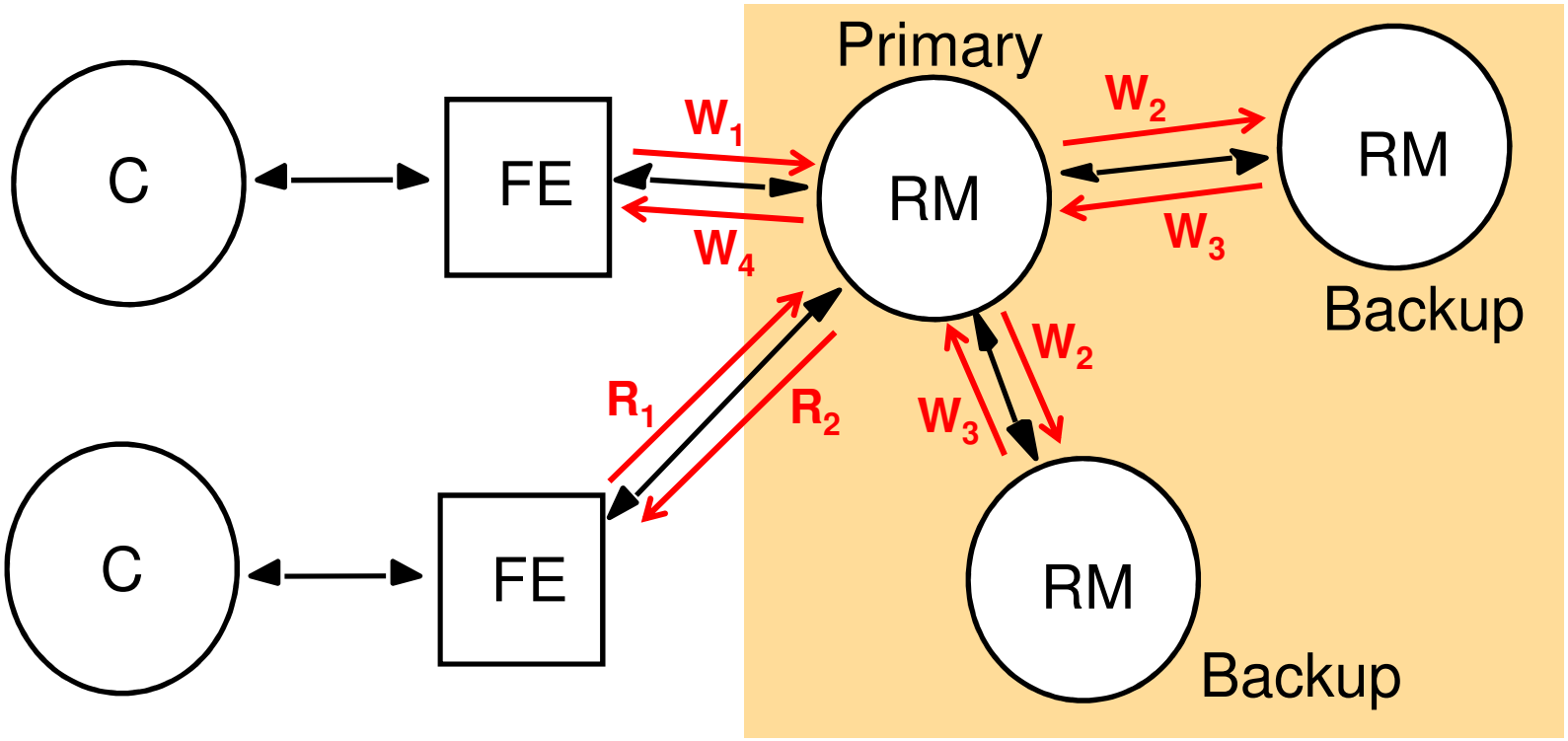


Passive replication

1. Request phase
   - The FE attaches a UID to the request and forwards it to the primary.
2. Coordination phase
   - The primary handles the request atomically and in order of receipt, using the UID to prevent executing an update more than once.
3. Execution phase
   - The primary executes the request and stores the response.
4. Agreement phase
   - In case of an update the primary sends the updated state, the UID and the response to the backups and waits for acknowledgements.
5. Response phase
   - The primary responds to the client's FE, which hands the response to the client.

CDK Figure 15.4 : including protocol

Primary-read, Primary-write version



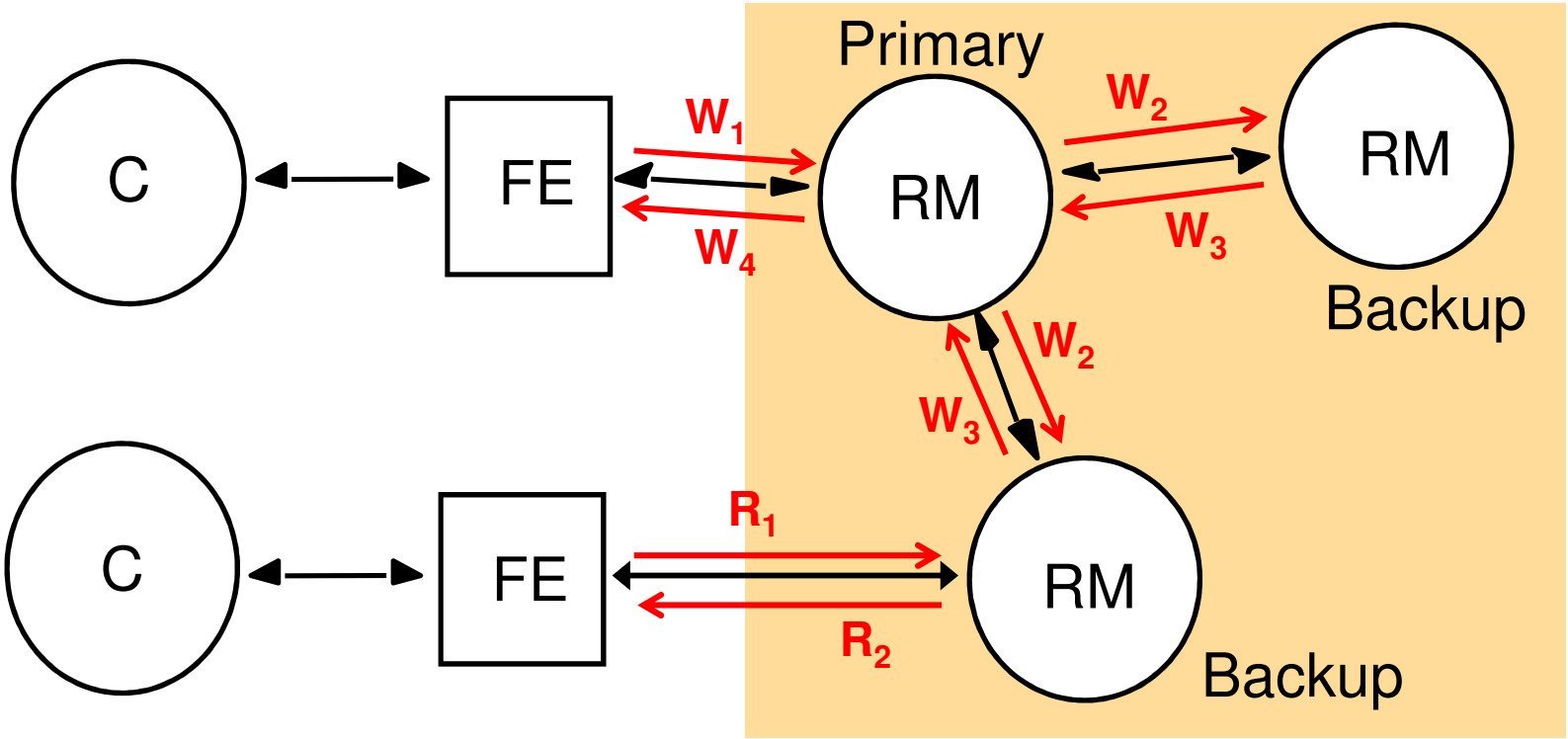
Recovery procedures

Upon failure of the primary replica manager the system can retain linearizability when

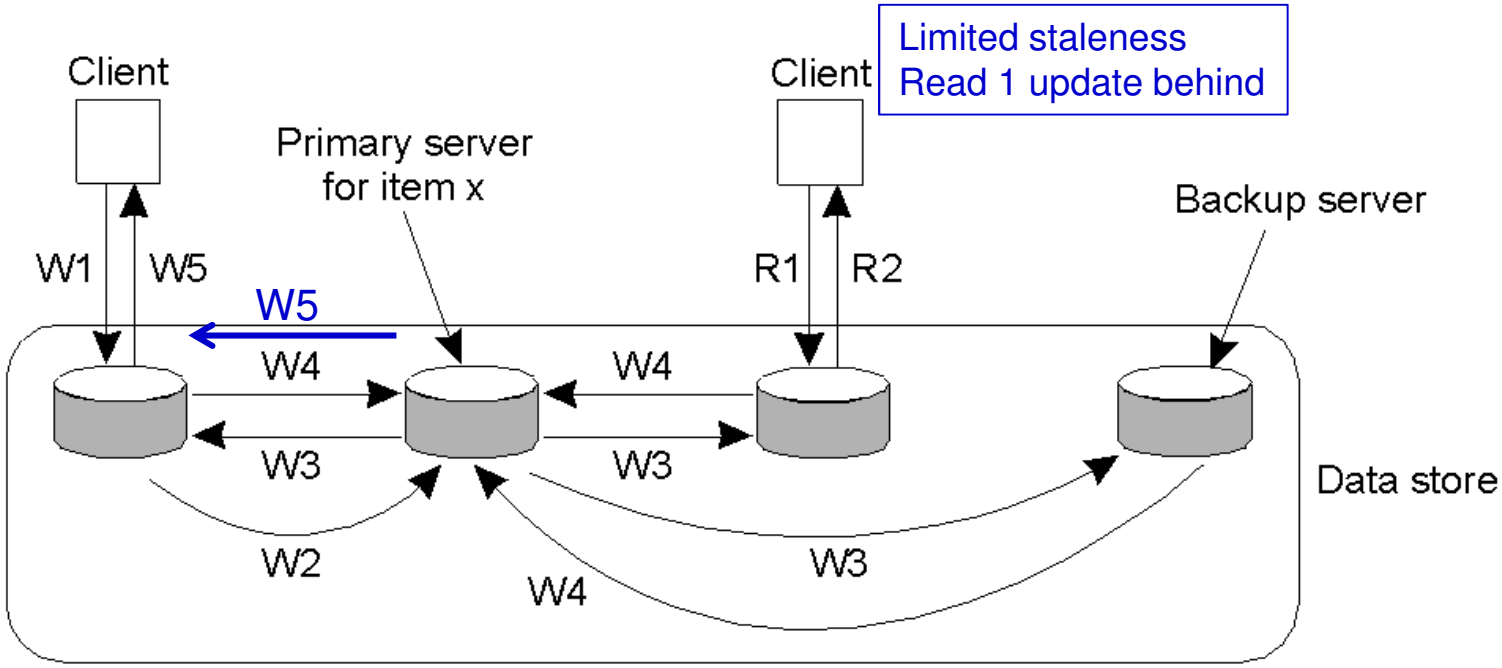
- The primary is replaced by a unique backup
  - Determined by leader election algorithm or new group view
- All backups agree on the set of operations that have been performed at the moment when the replacement backup takes over.
  - This requires that writes are sent to the backups using atomic multicast, i.e., all forwarded writes are delivered in the same order to all backups.
  - FEs may use a discovery service to locate the primary

CDK Figure 15.4 : including protocol

Local-read, Primary-write version



Local-read, primary-write
aka remote-write

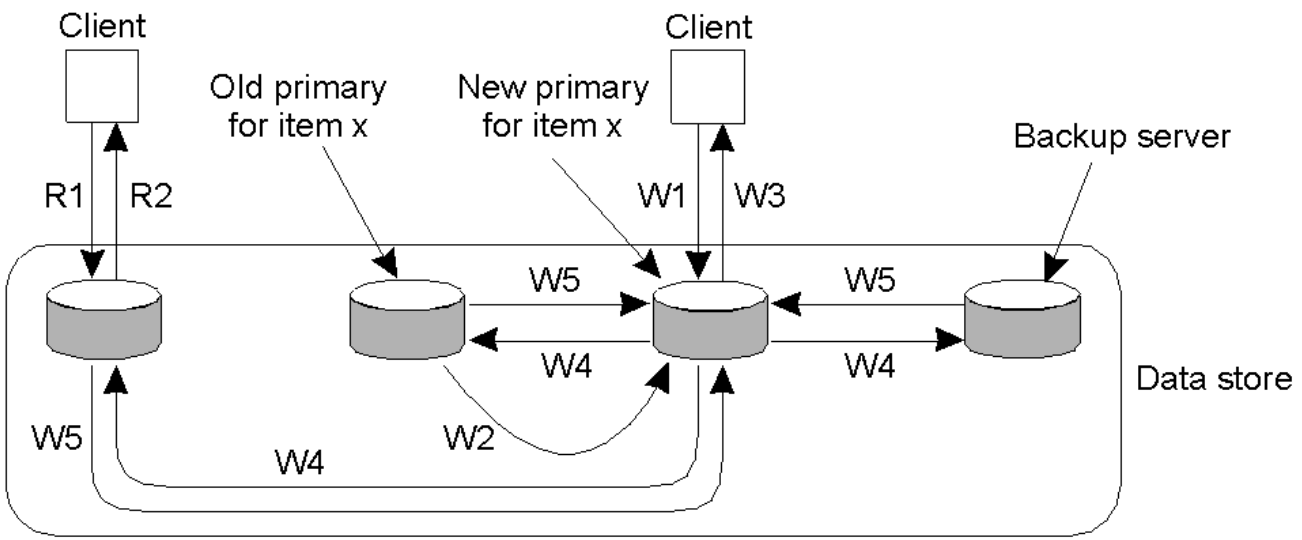


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Local-read, local-write aka local-write

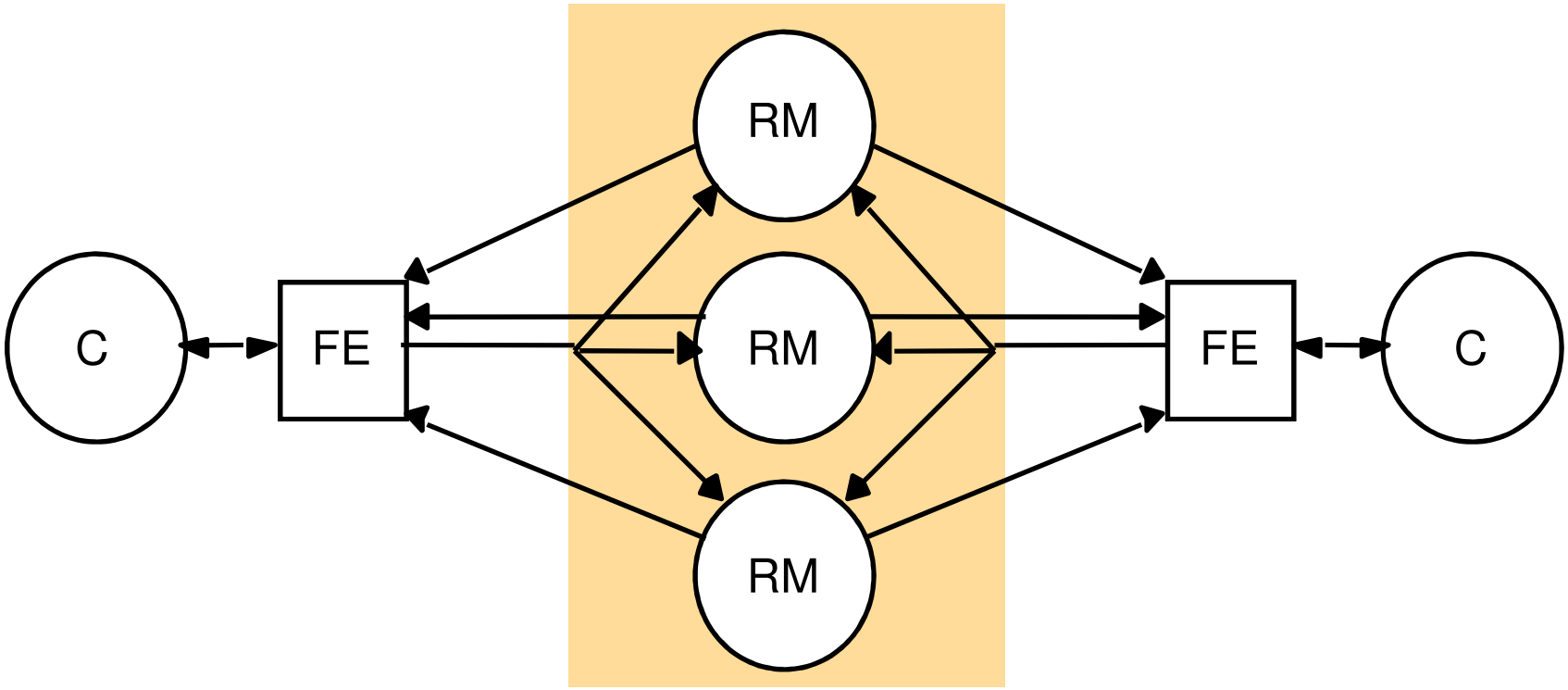
What happens to a write request that arrives at a replica manager that has lost its primary status?



W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

CDK Figure 15.5
Active replication



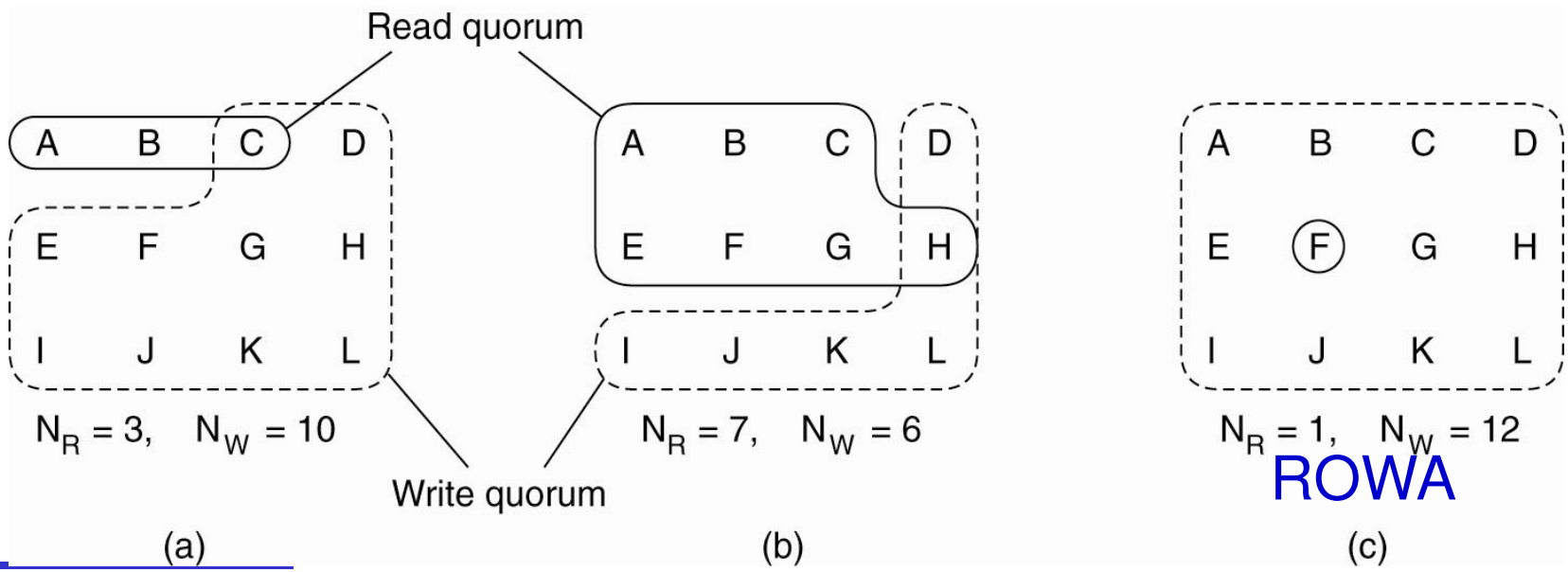
- FE multicasts request to all RMs, which all behave the same (symmetry)
- Achieves sequential consistency, due to total order delivery of updates

Active replication

1. Request phase
   - FE attaches a UID to client operations and forwards it to all RMs using totally ordered reliable multicast.
2. Coordination phase
   - Group communication delivers the request to every *correct* RM in the same total order.
3. Execution phase
   - Every RM executes the operation on its replica, resulting in the same local state and response.
4. Agreement phase absent
5. Response phase
   - RMs send their responses labeled with the request identifier to the FE which forwards it to the client after a certain number (depending on the desired fault resilience) responses have been received.

Quorum-based protocols

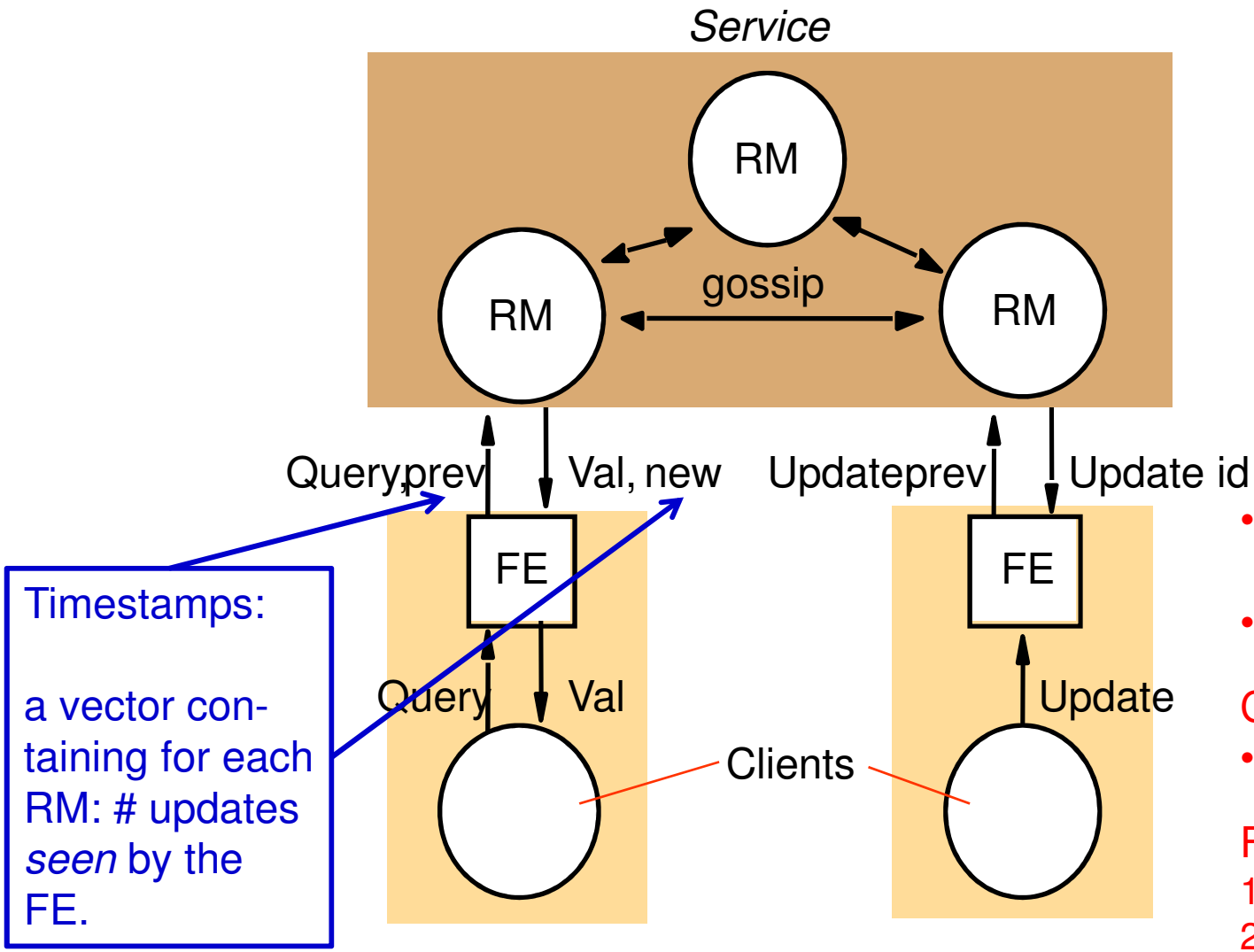
- Read operations need to inspect N_R replicas (the read quorum)
- Write operations need to update N_W replicas (the write quorum)
- With a total of N replicas
  - $N_R + N_W > N \ \wedge \ N_W > N/2$
- Replicas need a version number in order for the read to determine which replica contains the most recent value



Gossip architecture

- Aims at high availability
- Provides monotonic reads
- Customizable update delivery ordering
  - Usually causal ordering, because sequential consistency defies the goal of achieving high availability
- RMs exchange gossip messages on a periodic basis to disseminate update operations.
  - Per exchange a gossip partner chosen at random (anti-entropy)
  - Either push or pull or push-pull
- Resilient against network partitioning
  - Only one healthy RM in a partition is sufficient
- Scalability is an issue
  - # gossip messages, size of vector clocks

Figure 18.5
Query and update operations in a gossip service



- Updates should not read the state!
- Handled by FE

Client issues:

- X := X+1

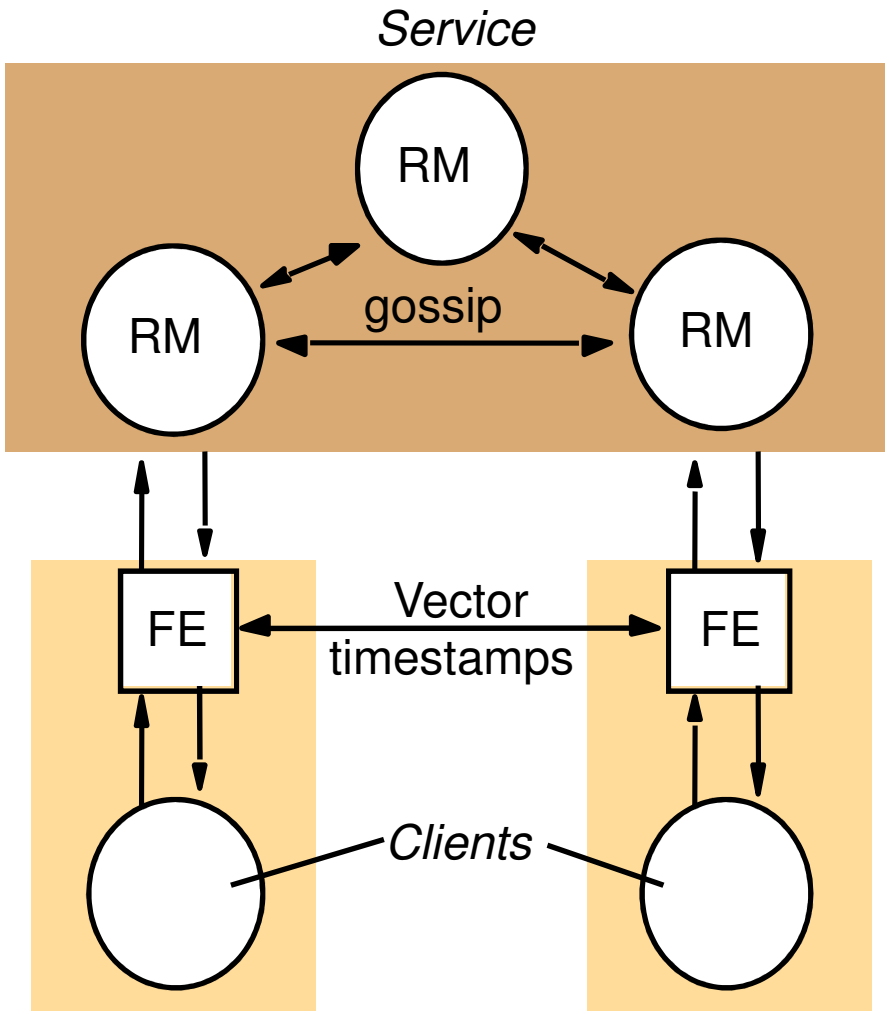
FE issues

1. tmp := Read (X)
2. Write (X, tmp+1)

Gossip architecture

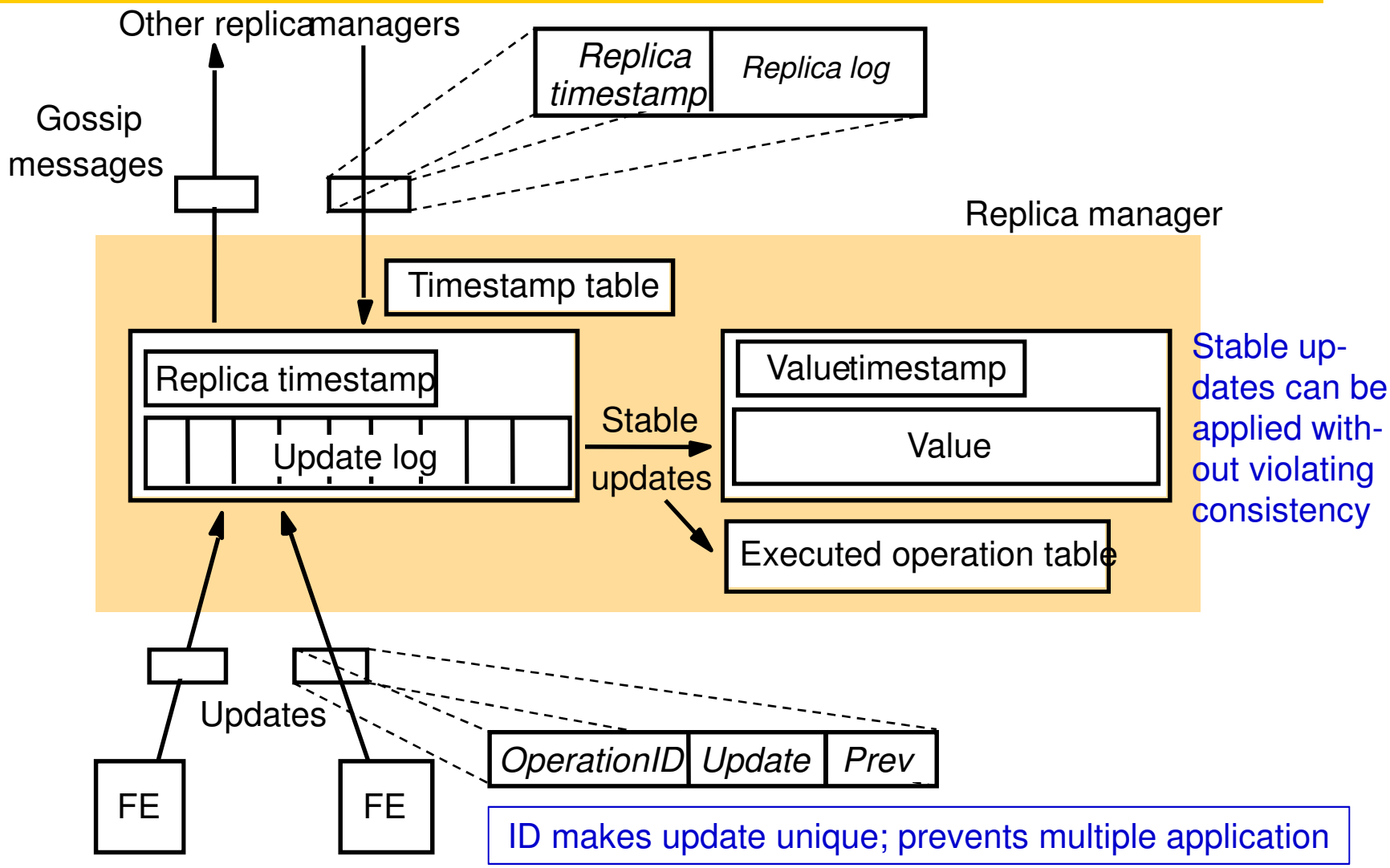
1. Request phase
   - FE sends request to a single RM (variable per request)
   - FE blocks client on query, not on update (although it may)
2. Coordination phase
   - RM does not process a request until allowed by ordering constraints
   - Implies waiting for gossip messages from other RMs
3. Execution phase
   - Every RM executes the operation when due (has become stable)
   - Query results are returned to the FE
4. Agreement
   - Exchange of gossip messages between RMs
   - Lazy, after some updates or when knowing a peer that has the update
5. Response phase
   - FE returns outcome query to client

Figure 15.7
Front ends propagate their timestamps whenever clients communicate directly



ALL client-to-client messages also run via the FEs to ensure proper timestamps for causal ordering.

Figure 15.8
A gossip replica manager, showing its main state components



Caching

Caching is a specific form of data replication. Its distinguishing features are:

- Initiated and managed by clients
  - The server-side data store may assist by notifying the cache that its data has become stale
  - A single cache may be shared by several clients
- Introduced with the objective to decrease access time, i.e., reduce latency of client requests
  - although the system as a whole may also benefit by reduction of network traffic
- Applies to transient data
- Consistency is referred to as cache coherence

Coherence enforcement strategies

1. No caching of shared data
   - Provides only limited performance improvement
2. Server-initiated enforcement
   - Server sends invalidation notification
     - Next query pulls the fresh value from the master replica
   - Server sends update
3. Client-initiated enforcement
   - Write-through
     - Write operations to the cache are immediately forwarded to the master replica
     - Client needs to have exclusive write permission to guarantee sequential consistency
   - Write-back
     - Write operations are delayed until the cache entry holding the replica is selected for eviction because the cache is full

Fault-tolerance

In most large systems (especially those build from commodity hardware) failure is the norm not the exception!

- Failure models for nodes
  - Fail stop (detectable by neighbors)
  - Crash failures (not detectable by neighbors)
- Failure models for connections
  - Corruption failures (handled by error-correcting codes)
  - Omission failures
  - Duplication failures
  - Arbitrary aka Byzantine failures

Network may become partitioned due to failures: of either a separator node or a bridge connection.

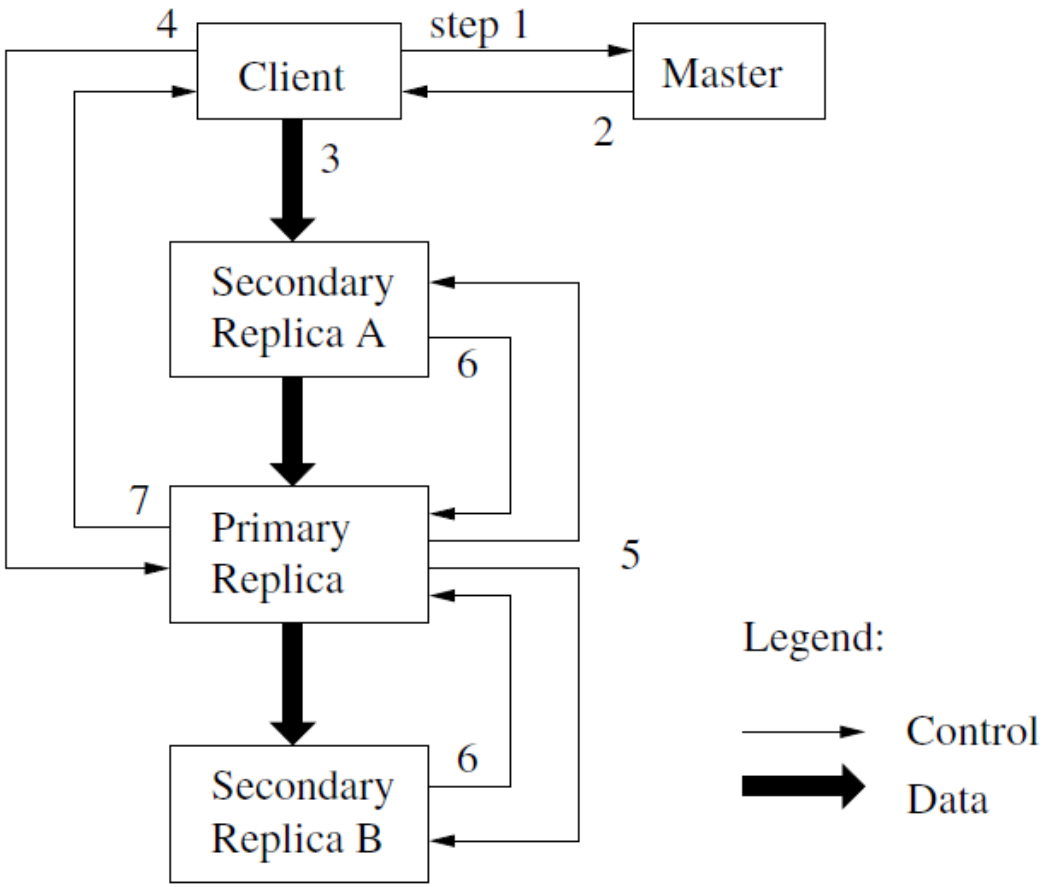
Agenda

- Introduction
- Consistency models
- Replica management
- Architectural case studies
  - Google File System
  - Coda File System
  - Amazon Dynamo
  - Squirrel Web cache

Google File system

Conit	File chunk of 64KB
# Replicas	User defined (default 3)
API	Read, Append, Write (optimized for appends, not writes) Large operations are divided into series of chunk sized operations that are not considered transactions.
Placement	Chunk replicas are spread over different racks for optimal reliability and availability.
Read access	To the nearest replica (using central location service)
Update policy	Essentially primary-based. Data is pushed by clients to replicas using pipelining. Writes are committed by leased-based primary. Reads from any replica.
Consistency	Relaxed consistency. After successful updates all clients will see the same data. For concurrent updates, although all replicas are the same there may not be a sequential execution that would have produced the outcome. (undefined file regions are detected using chunk version numbers)
Fault tolerance	Crash resilient.

GFS write operation



Determination primary

- by lease (60 sec)
- renewal requests are piggy backed on keep alive messages

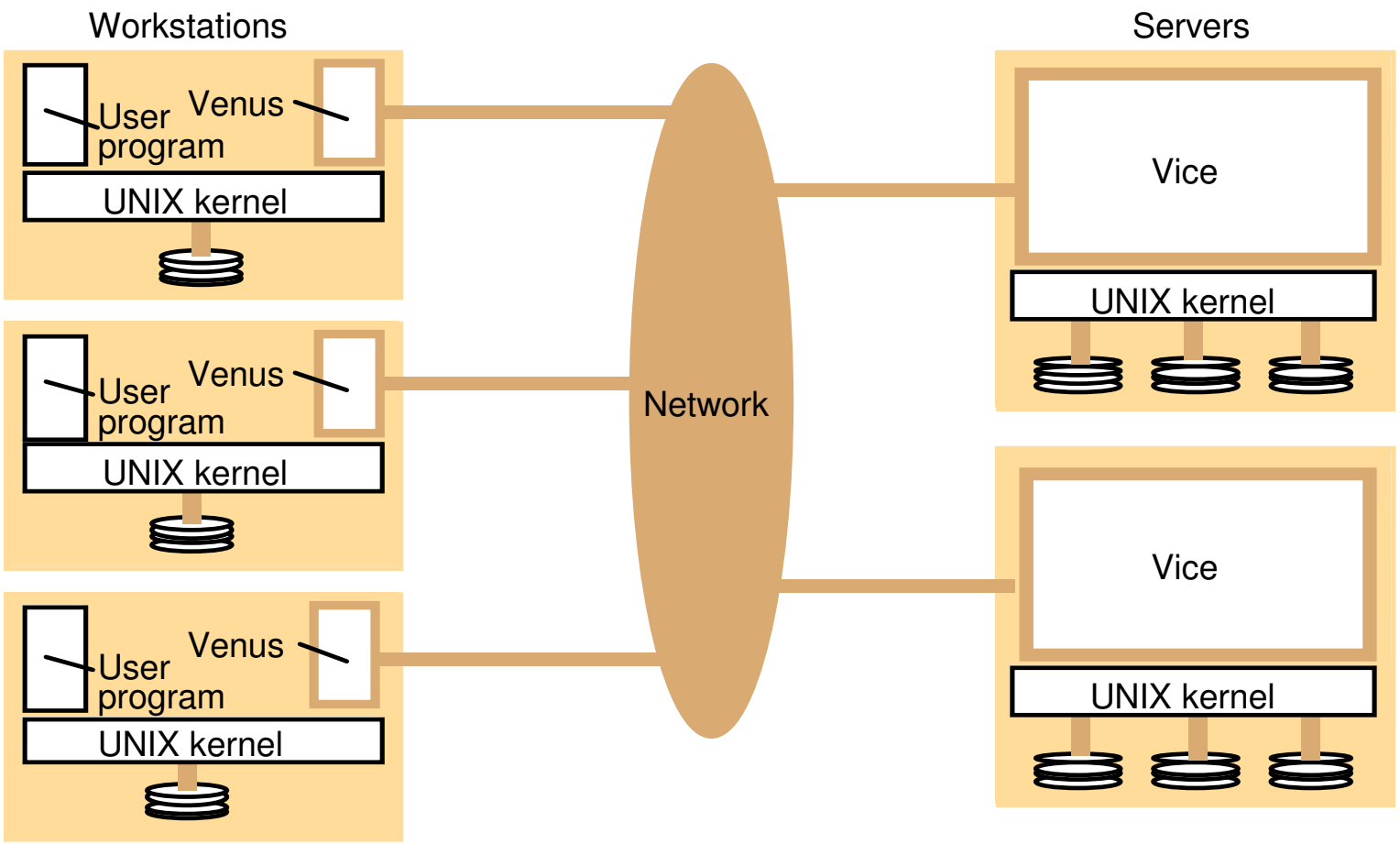
Coda

- Location transparent distributed file system
  - Descendant of the Andrew File System (AFS)
  - Caches volumes at the client side
  - Keeps more than one replica at the server side
- Key driver: constant availability
  - also increased resilience against failures
  - also improved scalability
- Mechanism
  - Server replication
  - Disconnected operation (using local cache)
  - Complete files are cached
- No distinction between voluntary disconnection and network or server failure

Coda file organization

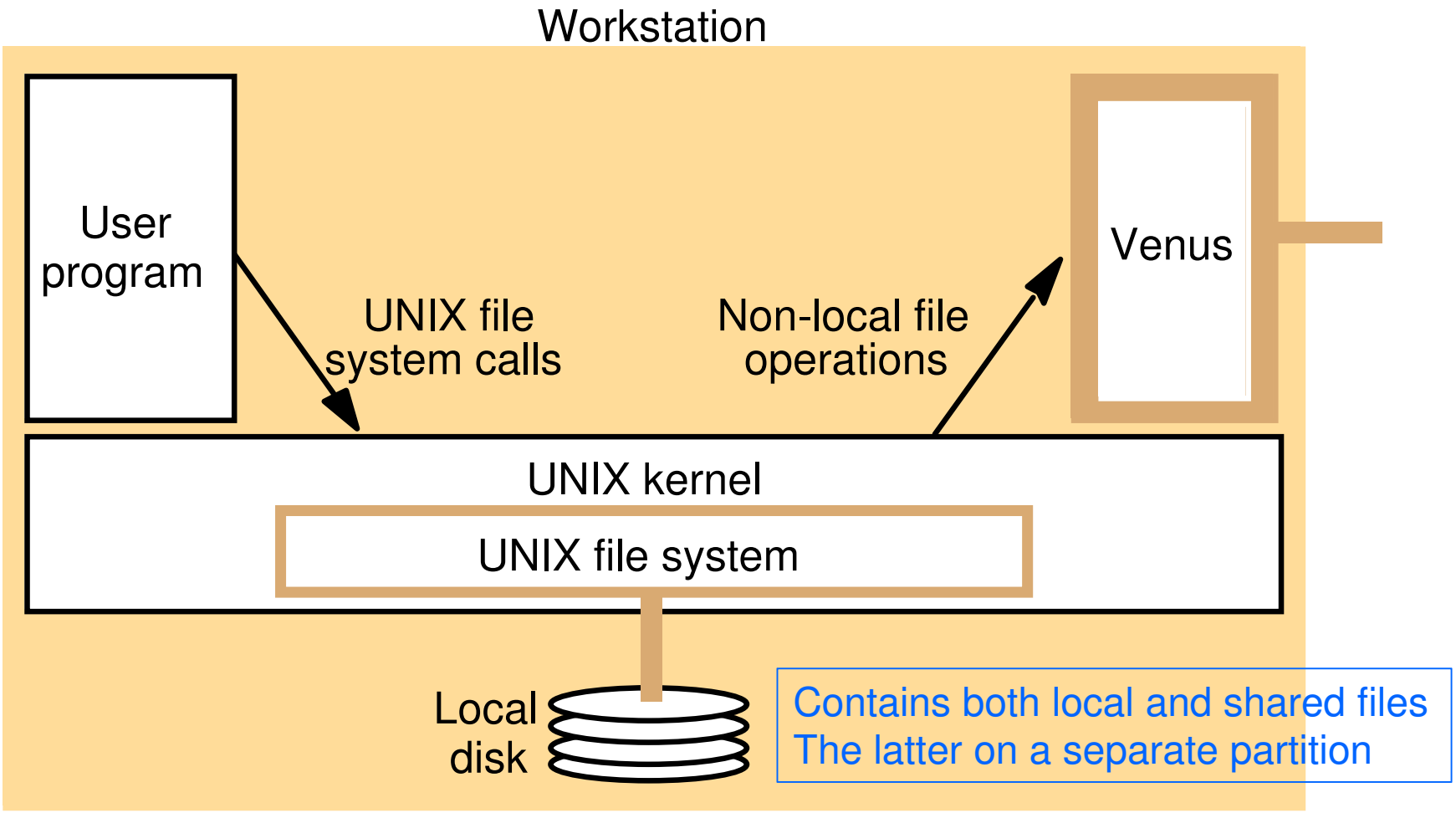
- Each file and directory in shared file space has a UFID
  - All replicas have the same UFID
- Unit of replication is a volume (set of files)
  - UFIDs include volume identification
- Volume Storage Group (VSG) of a volume
  - The set of servers that have a replica of the volume
  - Each file replica has a Coda Version Vector (CVV)
- Accessible volume storage group (AVSG)
  - Subset of the corresponding VSG
  - Maintained by clients for each volume from which it has cached files
  - Each client has a preferred server per AVSG
  - When a workstation is disconnected, AVSG is empty.

Figure 12.11
Distribution of processes in the Andrew File System



AFS: 1 server (custodian) per shared file;
CODA: multiple servers (VSG) per shared file

Figure 12.13
System call interception in AFS

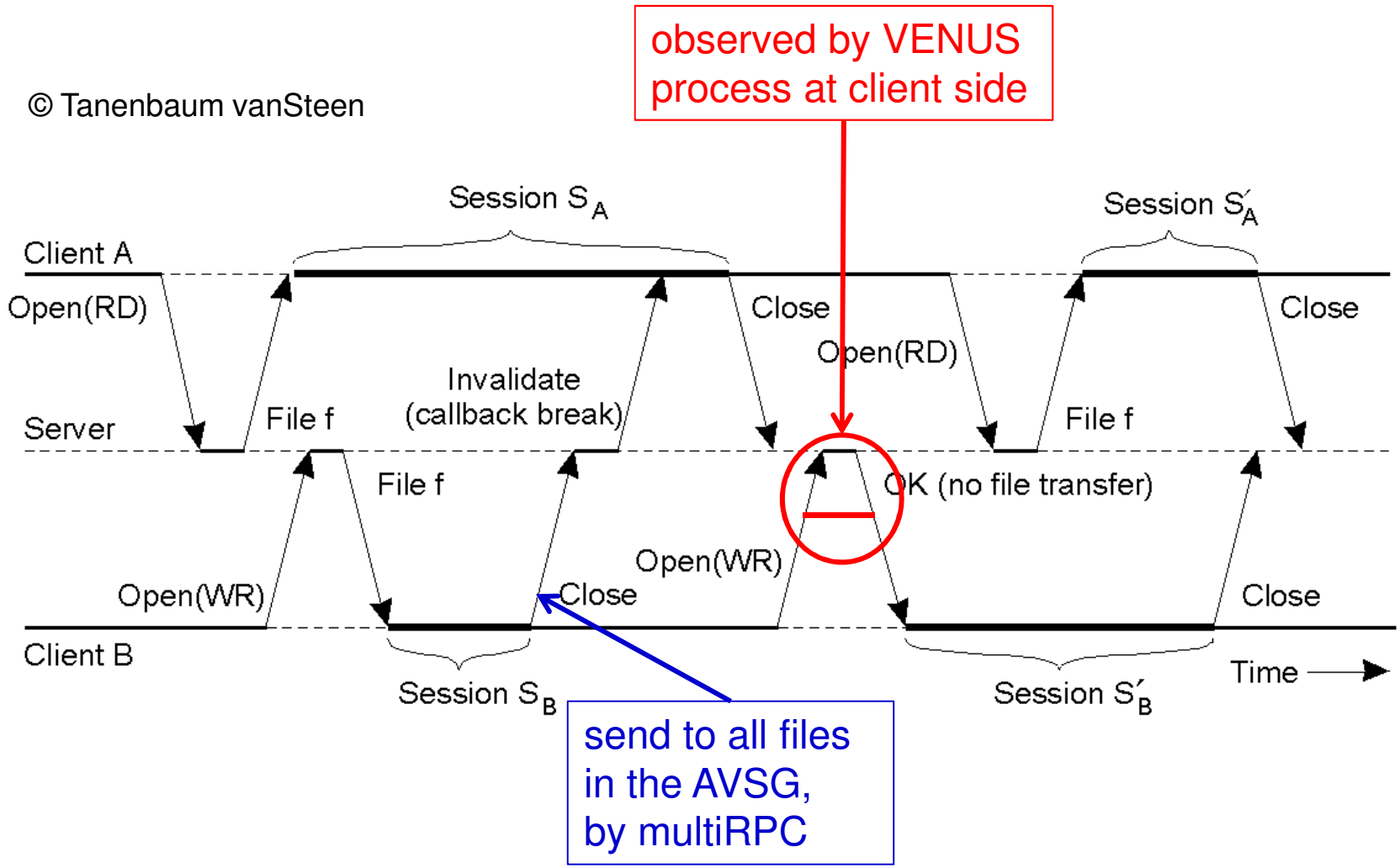


Callback mechanism

When a file f is copied from Vice to Venus

- A callback is registered at Vice servers (state-full server!)
- A valid callback promise for f is set at Venus client
  - Promises are either valid or cancelled (cf. staleness bit)
- When a Vice server becomes aware that another client has modified f it sends an invalidation message (*break*) to all clients that hold a promise changing the state to cancelled
  - So servers maintain state for each client
- Callback promises are checked when a file is opened
  - Opening a file for the first time or opening a file for which a callback break has been received results in downloading a fresh copy of the file
- Breaks are sent as a result of files being closed
- Callbacks are leases; valid for a period of time after which they have to be renewed.

Sharing Files in Coda



Consistency

- Variant of ROWA
- On “open file” and cache miss or stale copy
  - the server holding the most recent copy is made preferred server, callback is established and a copy is fetched
- On “close file” after modification the new file is transferred in parallel to AVSG.
- Venus regularly polls status AVSG both for loss of members or for newly available members
  - On loss of preferred server callback promise is dropped
- Inconsistencies, arising from partitioning, are resolved through Coda Version Vectors (vector clocks) or user intervention
- Further details, see Coda paper and TvS (Chp 11)

Amazon Dynamo

- Dynamo is a key-value store
- Availability is the key driver (achieved at the cost of reduced consistency)
- Number of replicas N
- Placement strategy:
  - On the first N healthy nodes of a preference list , containing > N physical nodes, spread across various data centers.
  - Physical nodes host virtual nodes (tokens) organized as a DHT. In principle, the replicas are on N consecutive virtual nodes starting at the one indicated by the hash of the object key using consistent hashing.
  - Zero-hop DHT.
    - i.e. enough routing info to reach the destination in 1 step

Amazon Dynamo (cntd)

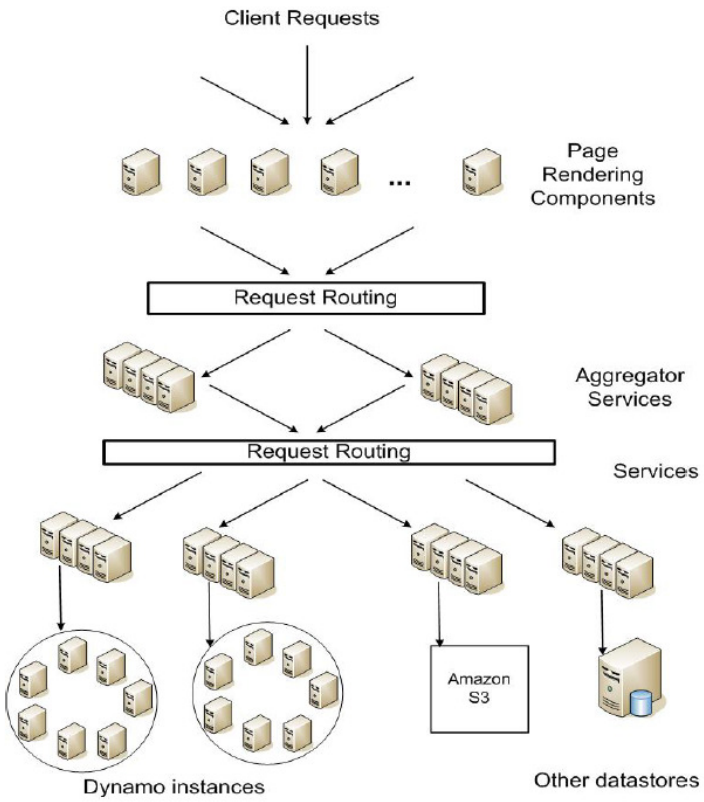


Figure 1: Service-oriented architecture of Amazon's platform

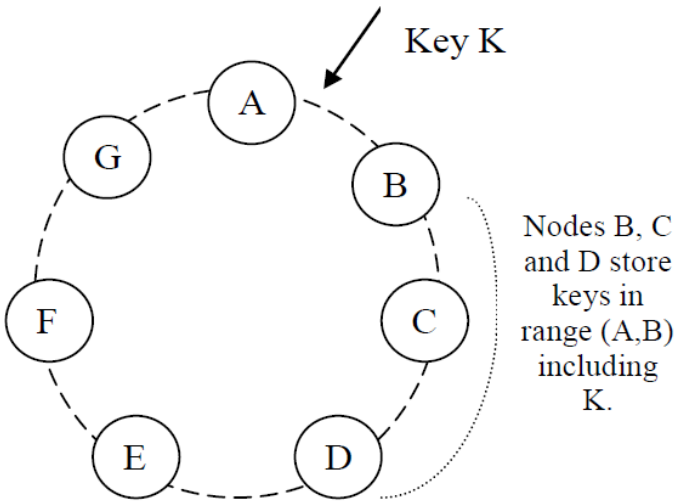


Figure 2: Partitioning and replication of keys in Dynamo ring.

Taken from: DeCandia et al, Dynamo: Amazon's Highly Available Key-value Store, SOSP 07, ACM, 2007

Amazon Dynamo (cntd)

- Very simple API
  - Get (key)
    - returns list of (object, context) pairs (usually 1 pair)
  - Put (key, context, object)
    - Writes an immutable new version. Older version are subsumed.
    - Due to concurrency and failures conflicting versions may result
- Access
  - Via coordinator node, usually the one determined by hashing the key. Using R+W quorums on the first N healthy nodes
    - Typically $(N, W, R) = (3,2,2)$
  - Coordinators generate vector clocks for versioning
  - Requests are blocked until the quorum is reached
    - Taking W=1 avoids blocking writes

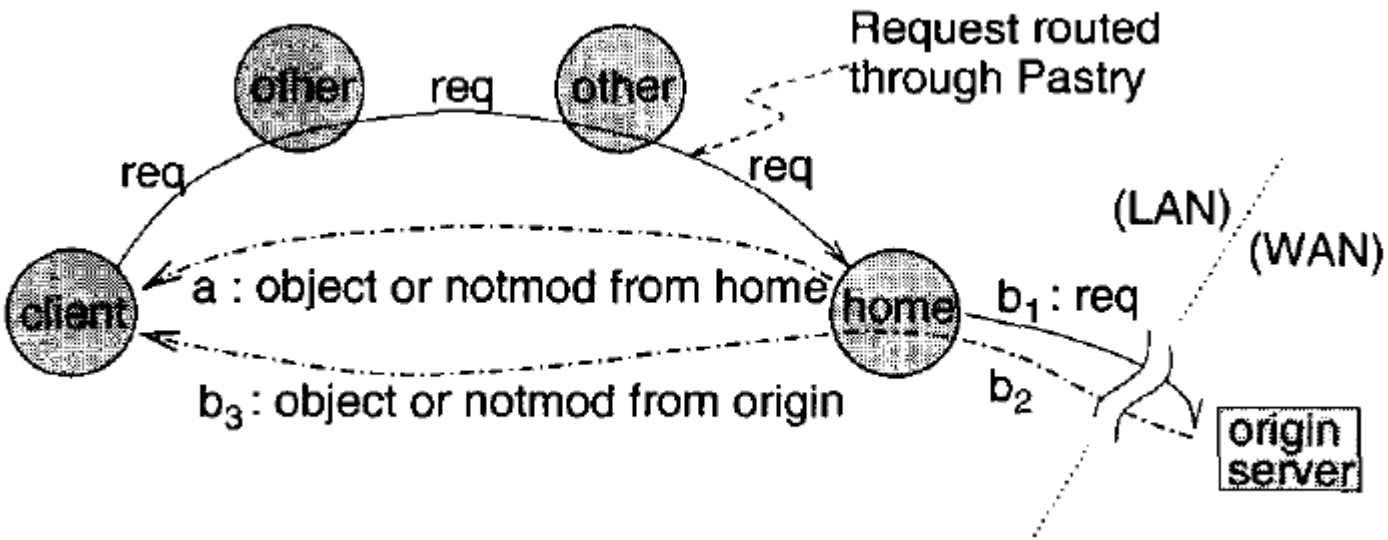
Amazon Dynamo

- Eventual consistency, sloppy quorum + some add-ons
  - Inconsistent values are stored side-by-side
    - Updates are time-stamped using a vector clock; updates arriving at a replica that are concurrent to the vector clock are both stored
    - Vector clocks allow detection of causal order
    - Clients need to resolve remaining inconsistency; applications are such that customer's are willing to do this, e.g., cleaning up their shopping cart that contains too many items, because deletes have not been seen.
    - Only 0.06% inconsistency
  - Committed values cannot be lost (durability), for this a (sloppy) quorum-based approach is used.
    - Nodes other than the first N of the preference list may assist to reach a quorum
- Fault tolerance (details see paper)
  - Transient node failures are covered by hinted handoff.
    - Values temporarily stored elsewhere return to home nodes, upon their recovery.
  - Permanent failures are covered by replica synchronization

Squirrel web cache

- Local caches in web browsers of clients in a single LAN are used to implement a distributed LAN-based web-server proxy.
  - Can handle corporate networks from 100-100,000 nodes or more.
  - Alternative for (a cluster of) web-proxy servers
- Uses a peer-to-peer architecture (overlay) on the LAN
  - Built on top of Pastry, but any P2P routing overlay will do.
  - Inserts a Squirrel proxy between the browser and the local web cache.
- Scales horizontally and automatically
  - As the number of browsers increases so does the cache capacity, whereas cache response time stays almost constant.
  - No administrative effort required.
- Makes use of HTTP headers to implement a conditional Get
  - If-Modified-Since <timestamp>
  - If-None-Match <ETag>

Home-store scheme



Items are stored in caches both at the client and at the home node
Each Web-resource has a home node on the LAN

- determined by a hash of its URL

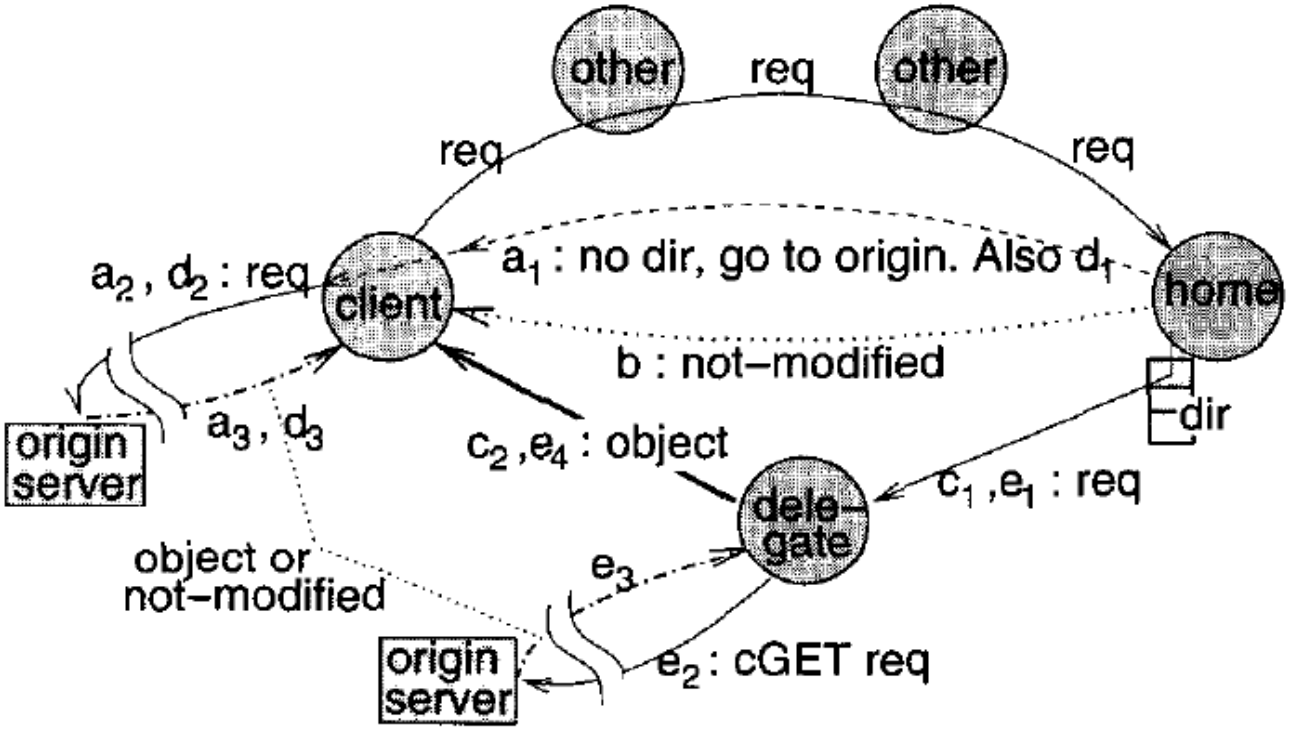
Seeking access to a cacheable object into its home node may result in

1. Miss (issue a GET and return object from origin))
2. Hit and fresh (return from home)
3. Hit and stale (issue a cGET and get an object or nonmod in return)

Directory scheme

The home node contains a (small) directory of delegates that

- are local nodes that have recently accessed the object (resource)
- are expected to have a replica of the object in their cache
- share the same version (ETag) of the object (so, all fresh, or all stale)



Literature

For the preparation of this slide set we have used material from various sources.

- Tanenbaum, van Steen: *Distributed Systems: Principles and Paradigms,*
  - Chapter 7, Sec 11.5.3, 11.6.1, 11.6.2.
- Coulouris, Dollimore, Kindberg, *Distributed Systems: Concepts and Design,*
  - Chapter 18 (basic model + gossip architecture)
- M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso, *Understanding Replication in Databases and Distributed Systems*,
  - basic model
- R. Ladin, B Liskov, L. Shrira, and S. Ghemawat
  Providing high availability using lazy replication
  - gossiping

Literature

- Sitaram Iyer, Antony Rowstron, and Peter Druschel, *Squirrel: A decentralized peer-to-peer web cache*, *PODC 2002.*

- Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere, *Coda: A Highly Available File System for a Distributed Workstation Environment* , IEEE Trans on Comp , Vol 39, Issue 4,1990.

- Giuseppe DeCandia et. al., *Dynamo: Amazon's Highly Available Key-value Store*, *SOSP 2007.*

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, *The Google File System*, *SOSP 2003*