Consistency & Replication

Yao Liu

Outline

- Data-centric consistency models
- Approaches for implementing sequential consistency
  - primary-backup approaches
  - active replication using multicast communication
  - quorum-based approaches
- Eventual consistency
  - Client-centric consistency models

Replication - motivation

- Performance enhancement
- Enhanced availability/reliability
- Fault tolerance
- Scalability
  - tradeoff between benefits of replication and work required to keep replicas consistent

Replication - requirements

- Consistency
  - Depends upon applications
  - In many applications, we want that different clients making (read/write) requests to different replicas of the same logical data item should not obtain different results
- Replica transparency
  - desirable for most applications

Consistency models

- Consistency model is a contract between processes and a data store
  - if processes follow certain rules, then the data store will work “correctly”
- Needed for understanding how concurrent reads and writes behave w.r.t. shared data

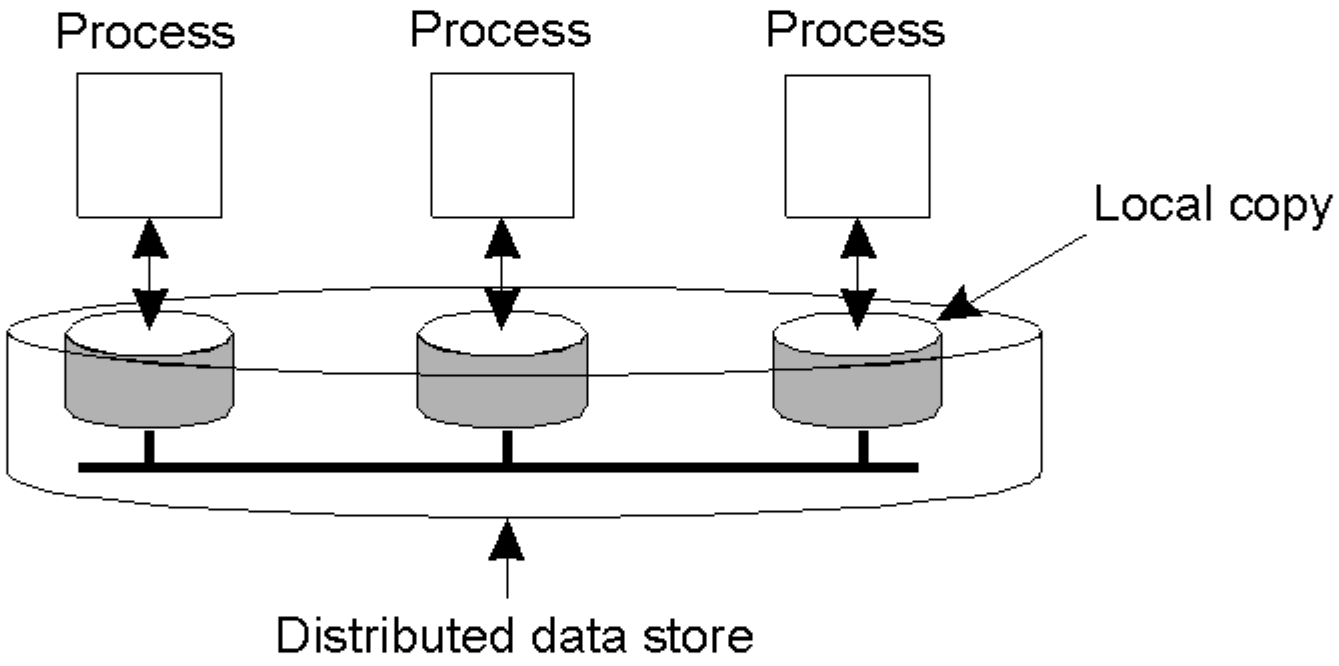
Consistency models

- **Data-centric** consistency model
  - Concerns read and write on shared data (e.g., shared memory, shared database, distributed file system, etc.)

- **Client-centric** consistency model
  - Concerns consistency experienced by any one client when accessing a distributed data store.

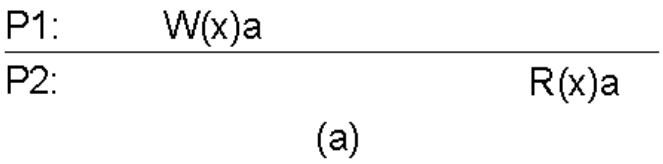
Data-centric consistency models

- The general organization of a logical data store, physically distributed and replicated across multiple processes.



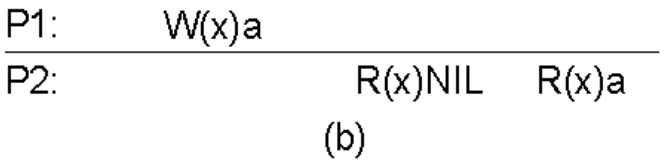
Strict consistency

- Any read on a data item x returns a value corresponding to the result of the most recent write on x.



(a)

A strictly consistent store



(b)

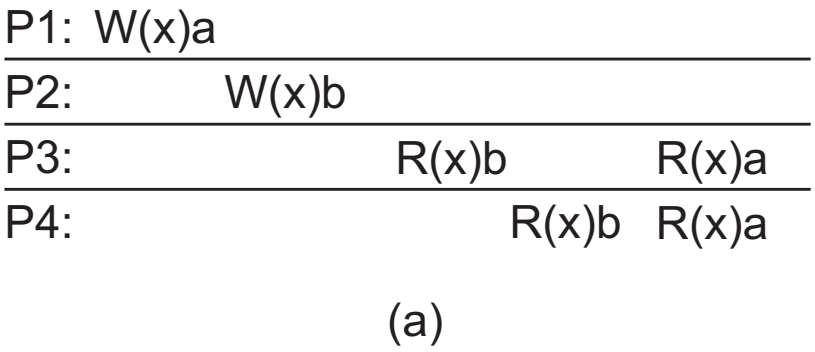
A store that is not strictly consistent.

Behavior of two processes, operating on the same data item.

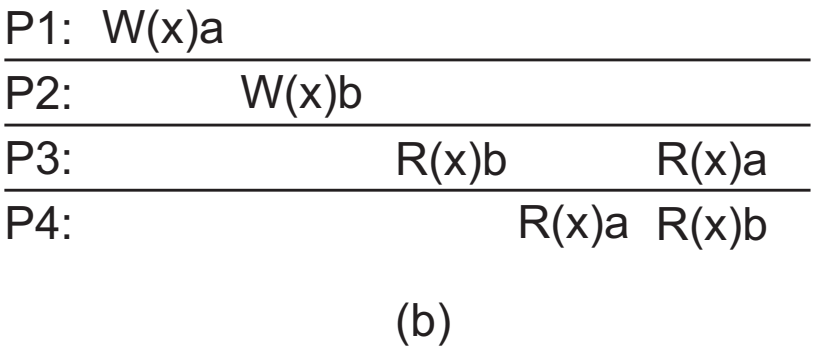
- The problem with strict consistency is that it relies on absolute global time

Sequential consistency

1. The result of any execution is the same as if the operations by all processes were executed *in some sequential order*, and
2. The operations of each individual process appear in this sequence in the order specified by its program



(a)



(b)

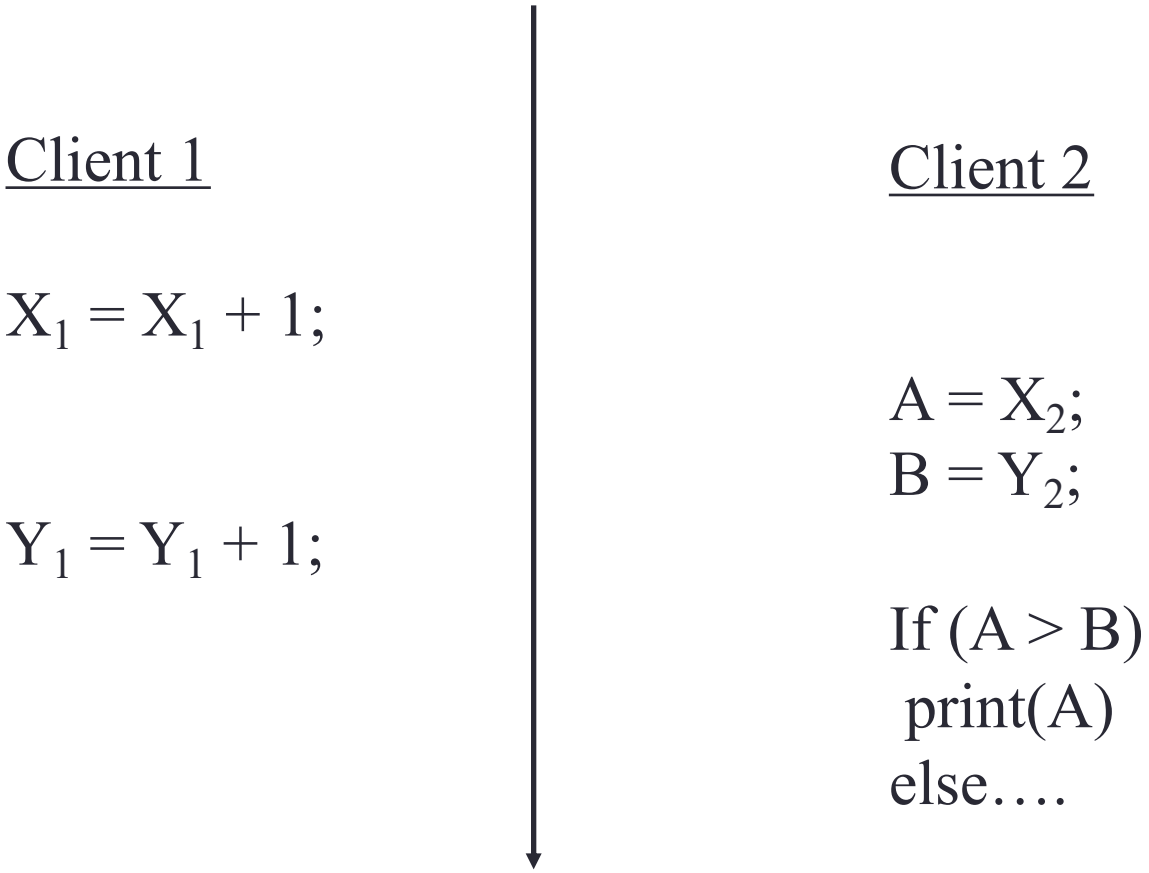
a) A sequentially consistent data store.
b) A data store that is not sequentially consistent.

Linearizability

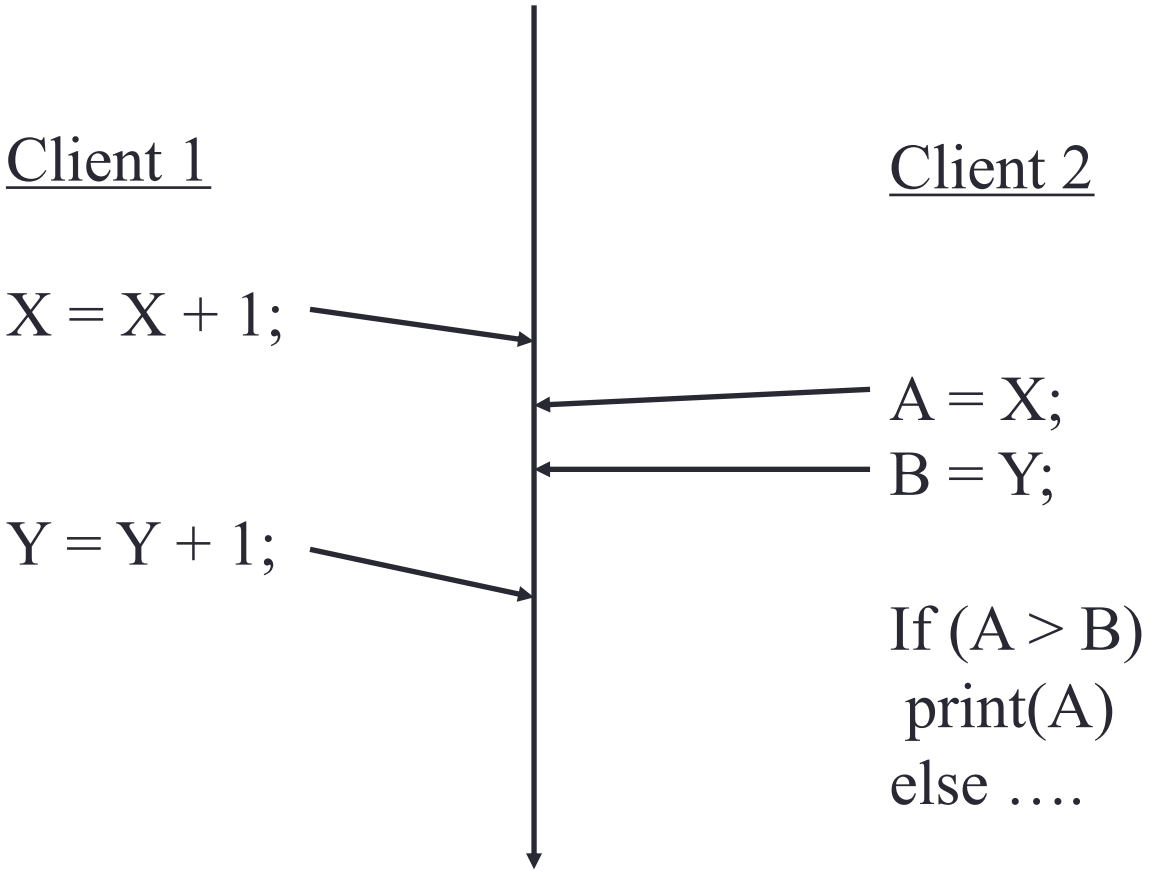
- Definition of sequential consistency says nothing about time
  - there is no reference to the “most recent” write operation
- Linearizability
  - weaker than strict consistency, stronger than sequential consistency
  - operations are assumed to receive a timestamp with a global available clock that is *loosely synchronized*

1. The result of any execution is the same as if the operations by all processes on the data store were executed in some sequential order, and
2. The operations of each individual process appear in this sequence in the order specified by its program.
3. In addition, if $ts_{op1}(x) < ts_{op2}(y)$, then OP1(x) should precede OP2(y) in this sequence

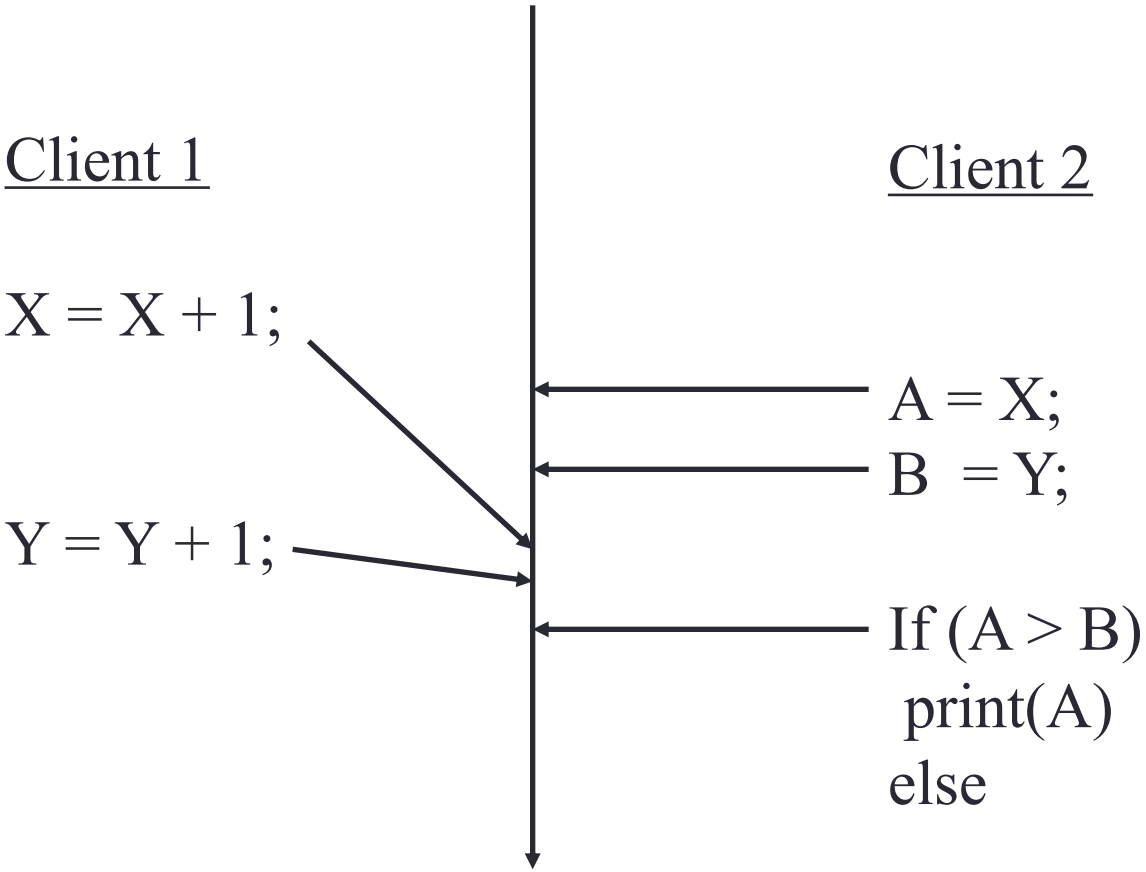
Example



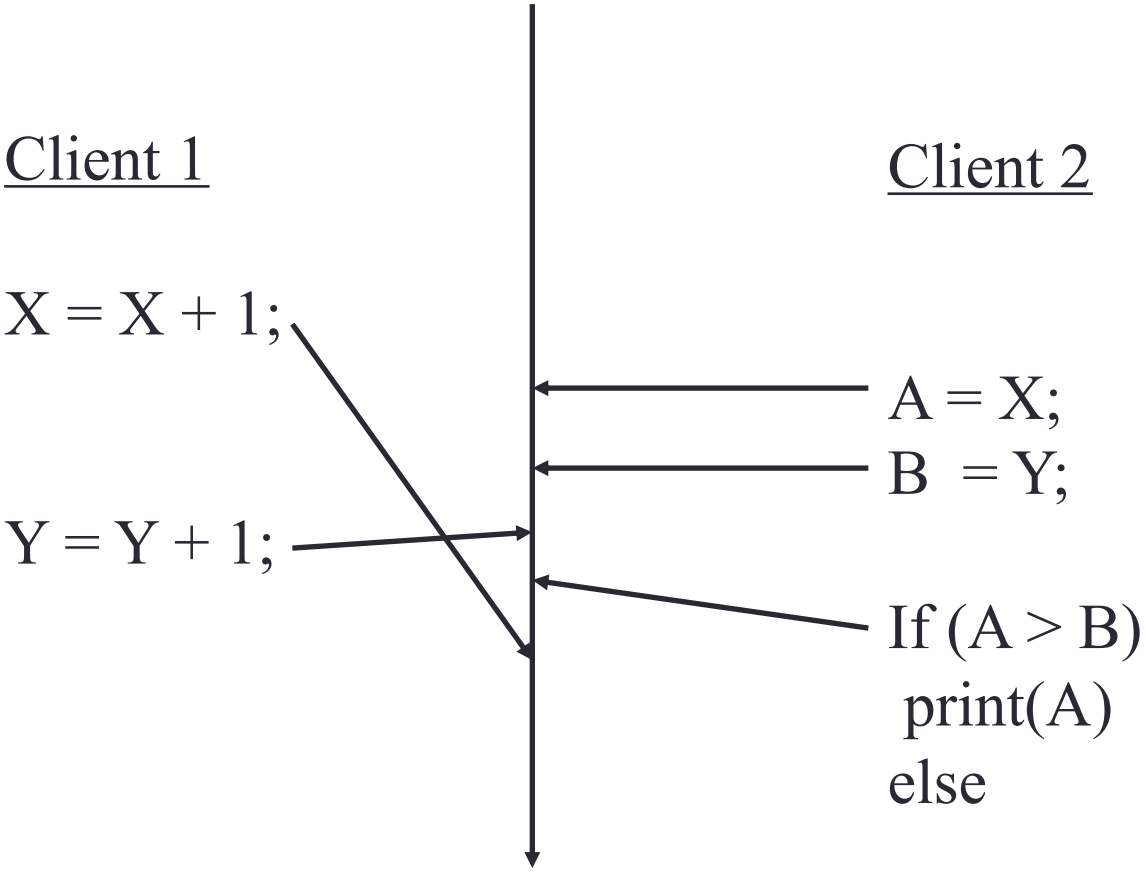
Linearizable



Not linearizable but sequentially consistent



Neither linearizable nor sequentially consistent



Causal consistency

- Necessary condition: writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

P1:	W(x)a			W(x)c	
P2:		R(x)a	W(x)b		
P3:		R(x)a		R(x)c	R(x)b
P4:		R(x)a		R(x)b	R(x)c

- This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

Causal consistency (2)

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

P1: W(x)a				
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(b)

a) A violation of a causally-consistent store.
b) A correct sequence of events in a causally-consistent store.

FIFO consistency

- Necessary Condition: Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

P1: W(x)a						
P2:	R(x)a	W(x)b	W(x)c			
P3:				R(x)b	R(x)a	R(x)c
P4:				R(x)a	R(x)b	R(x)c

A valid sequence of events of FIFO consistency

Summary of consistency models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (non-unique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were issued. Writes from different processes may not always be seen in that order

Weak consistency models

- In distributed systems, we typically refer to weaker consistency models than sequential consistency as weak consistency models.
  - e.g., causal consistency
  - optimistic approaches that use application-specific operations to achieve *eventual consistency*

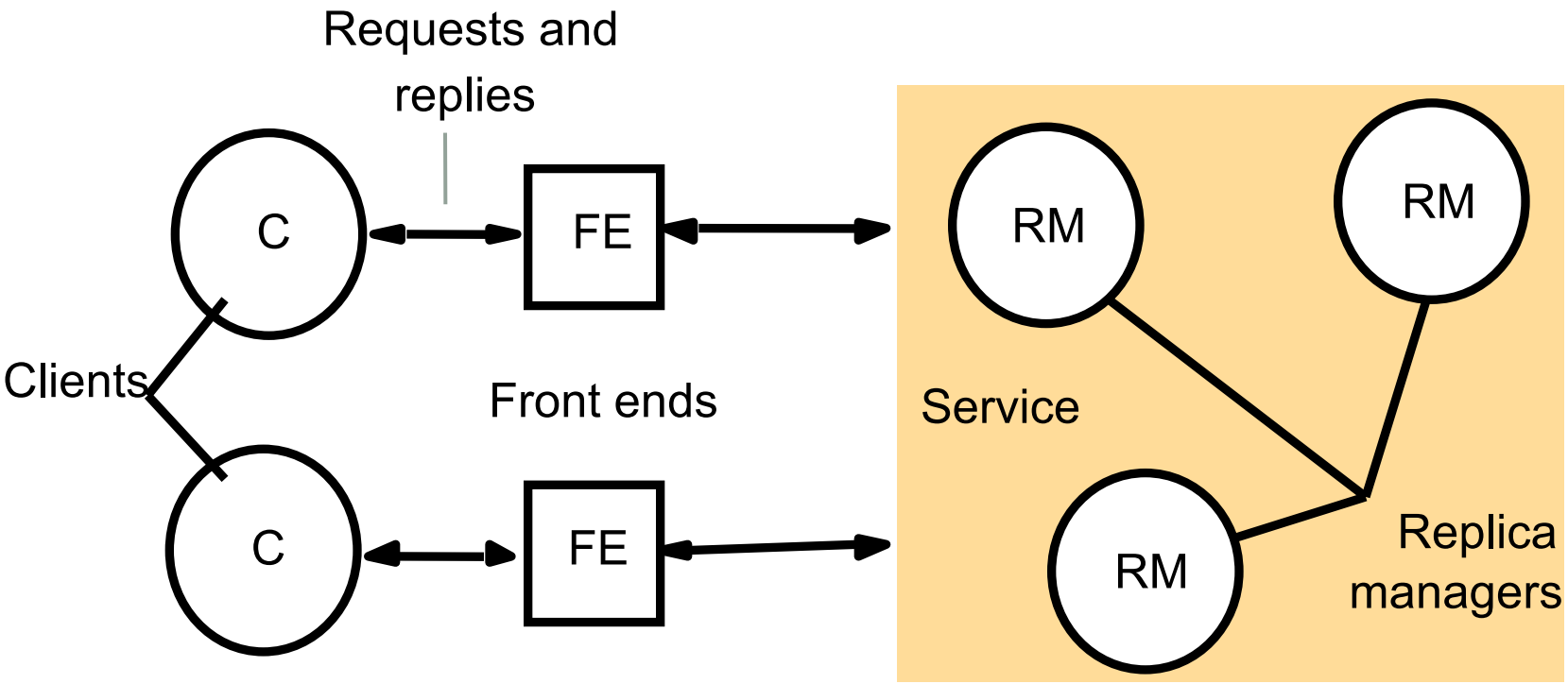
Implementing sequential consistency

- Good compromise between utility and practicality
  - We can do it
  - We can use it
- Stricter: too hard
- Less strict: replicas can disagree forever

System model

- Assume replica manager applies operations to its replicas *recoverably*
- Set of replica managers may be static or dynamic
- Requests are reads or writes (updates)

A basic architectural model for the management of replicated data



System model

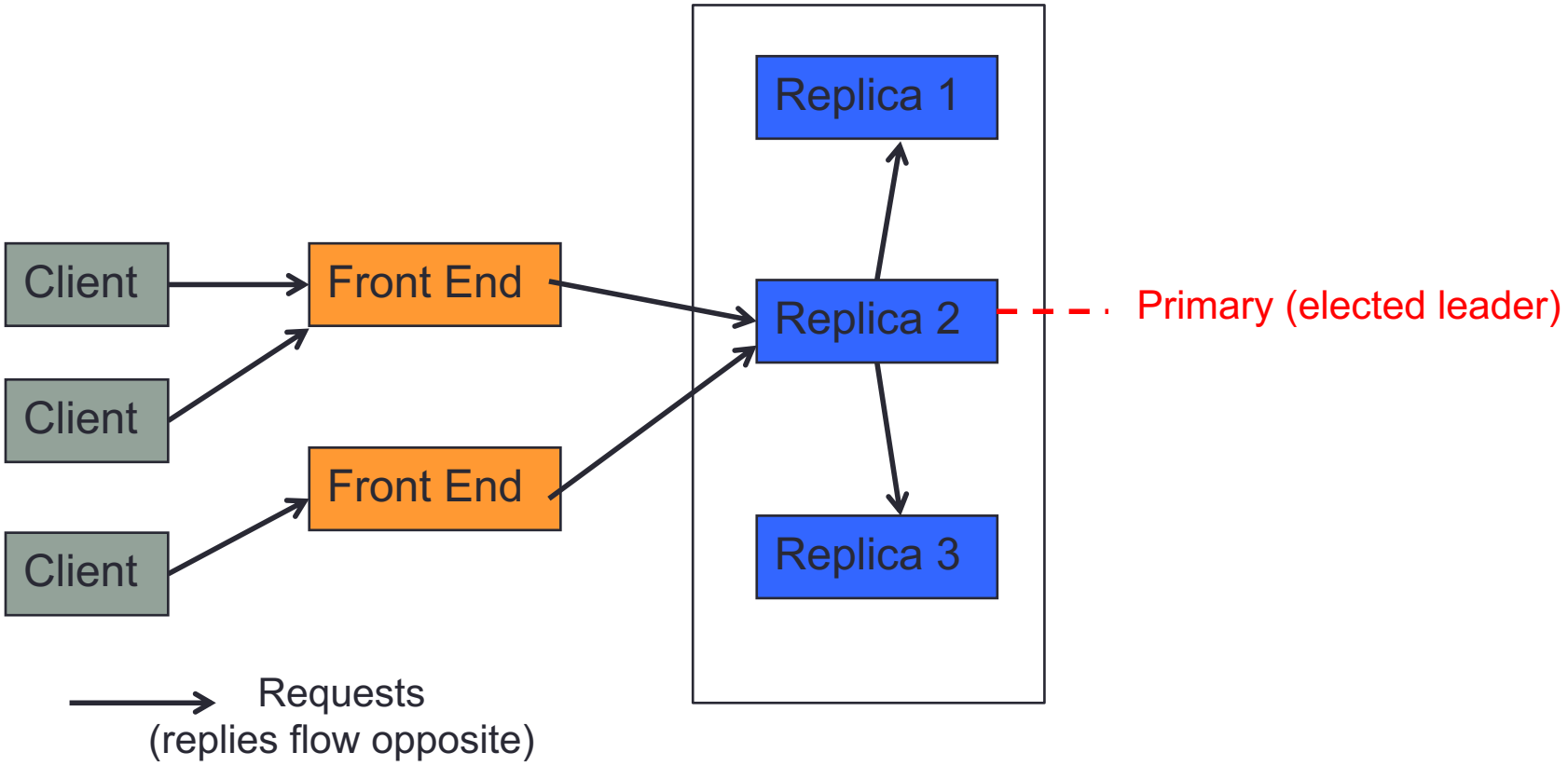
Five phases in performing a request

- Front end issues the request
  - Either sent to a single replica or ***multicast*** to all replica mgrs.
- Coordination
  - Replica managers coordinate in preparation for the execution of the request, i.e., agree if a request is to be performed and the ordering of the request relative to others
    - FIFO ordering, Causal ordering, Total ordering
- Execution
  - Perhaps tentative
- Agreement
  - Reach consensus on effect of the request, e.g., agree to commit or abort in a transactional system
- Response

Mechanisms for sequential consistency

- Primary-based replication protocols
- Replicated-write protocols
  - Active replication using multicast communication
  - Quorum-based protocols

The passive (primary-backup) model



Front ends only communicate with primary

Passive (primary-backup) replication

- **Request**: FE issues a request containing a unique identifier to the primary replica manager
- **Coordination**: The primary takes each request in the order in which it receives it
- **Execution**: The primary executes the request and stores the response
- **Agreement**: If the request is an update, the primary sends the updated state, the response, and the unique id to all backups. The backups send an acknowledgement
- **Response**: The primary responds to the front end, which hands the response back to the client

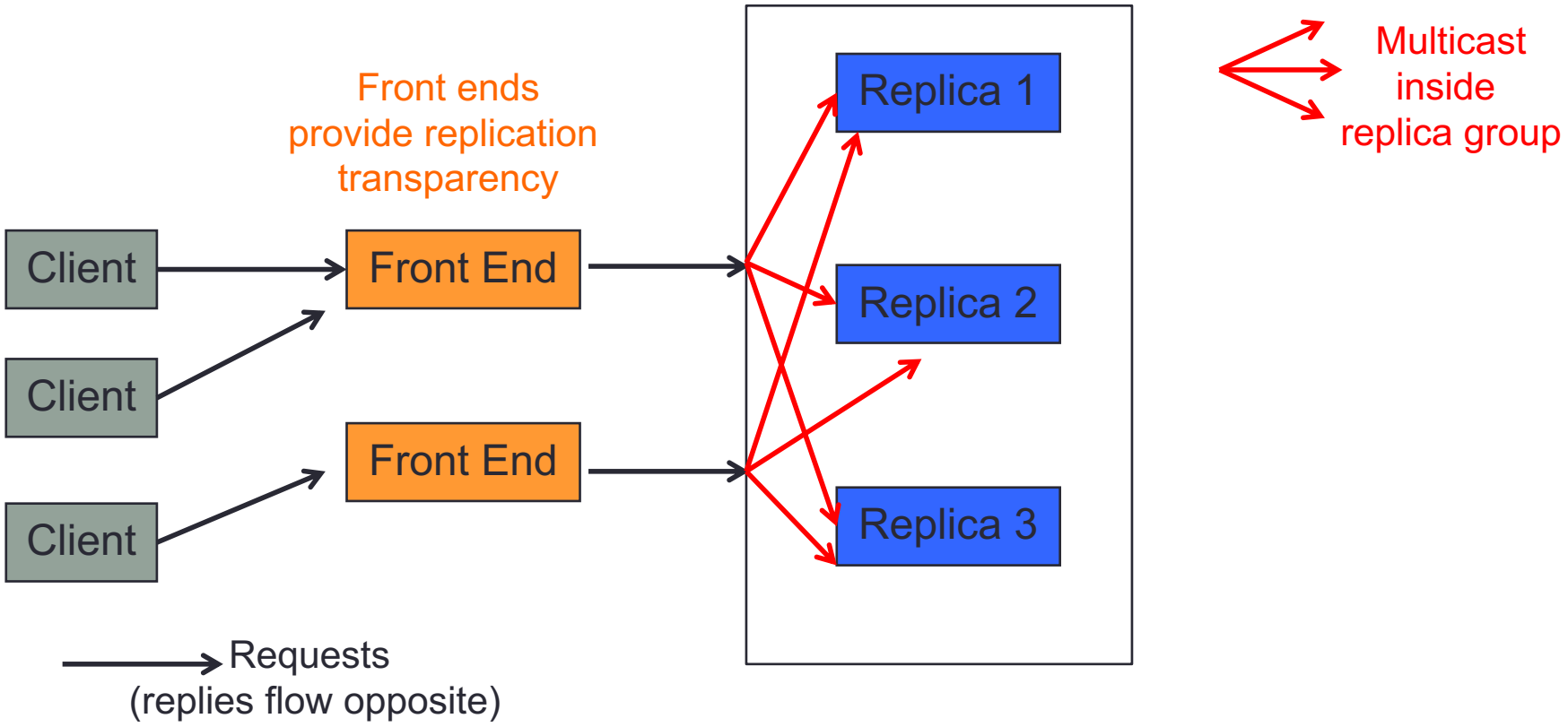
Passive (primary-backup) replication

- Implements *linearizability* if primary is correct, since primary sequences all the operations
- If primary fails, then system retains linearizability if a single backup becomes the new primary and if the new system configuration takes over exactly where the last left off
  - If primary fails, it should be replaced with a unique backup
  - Replica managers that survive have to agree upon which operations had been performed when the replacement primary takes over
  - Requirements met if replica managers organized as a group and if primary uses **view-synchronous communication** to propagate updates to backups

Active replication using multicast

- Active replication
  - Front end **multicasts** request to each replica using a *totally ordered reliable multicast*
  - System achieves sequential consistency but not linearizabilty
    - Total order in which replica managers process requests may not be same as real-time order in which clients made requests

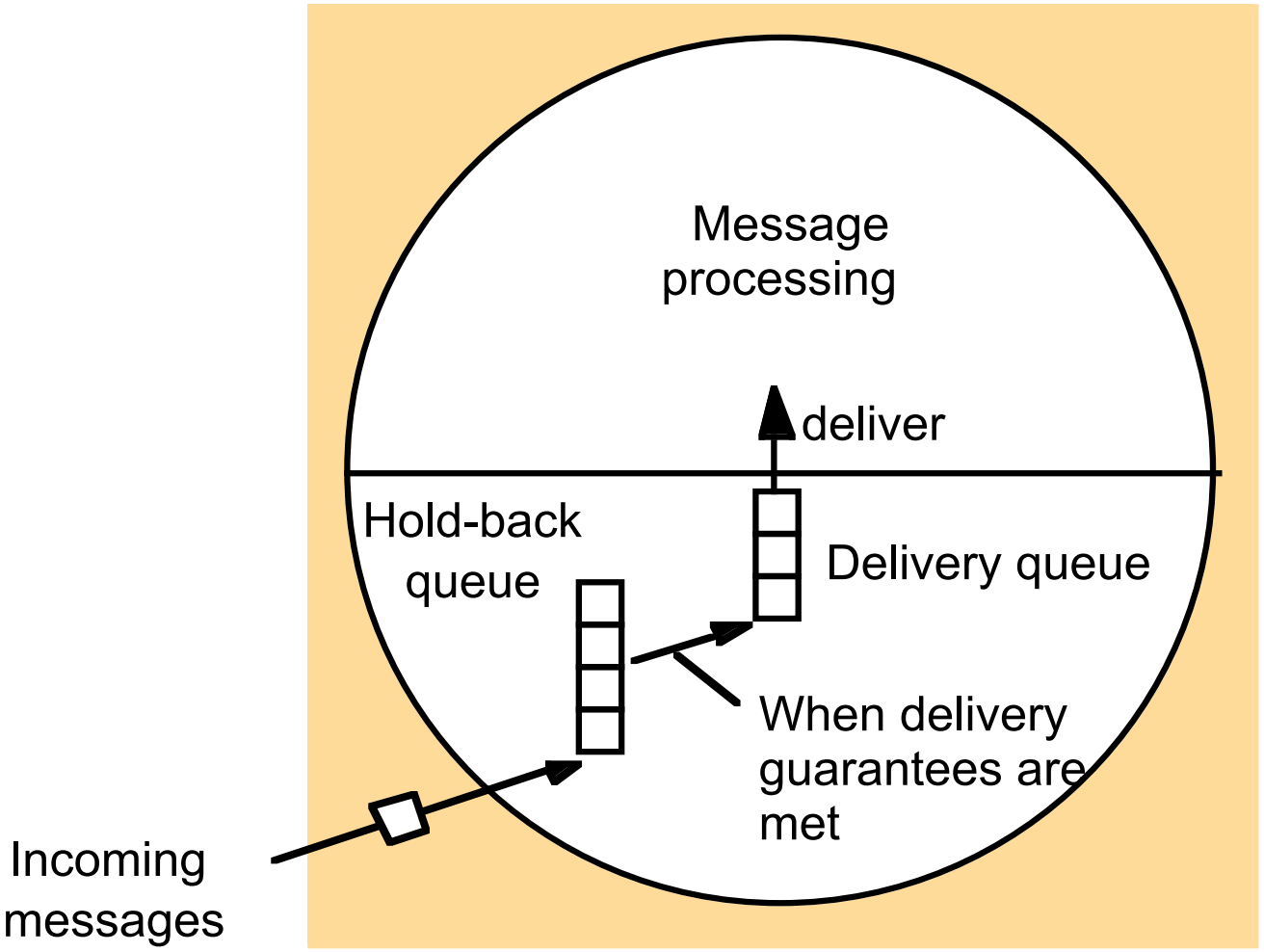
Active replication



Implementing ordered multicast

- Incoming messages are held back in a queue until delivery guarantees can be met
- Coordination between all machines needed to determine delivery order
- Three popular flavors implemented by several multicast protocols
  - FIFO-ordering
    - easy, use a separate sequence number for each process
  - Causal ordering
    - use vector timestamps
  - Total ordering
    - Use a sequencer

The hold-back queue for arriving multicast messages



FIFO ordering

- Multicasts from each sender are received in the order they are sent, at all receivers
- Don't worry about multicasts from different senders
- More formally
  - If a correct process issues (sends) multicast(*g*,*m*) to group *g* and then multicast(*g*,*m'*), then every correct process that delivers *m'* would already have delivered *m*.

Causal ordering

- Multicasts whose send events are causally related, must be received in the same causality-obeying order at all receivers
- Formally
  - If multicast(*g*,*m*) → multicast(*g*,*m'*), then any correct process that delivers *m'* would already have delivered *m*.
  - (→ stands for happens-before)

Total ordering

- Ensures all receivers receive all multicasts in the same order
- However, no guarantee is made receiving multicasts in the order of sending
- Formally
  - If a correct process P delivers message *m* before *m'* (independent of the senders), then any other correct process P' that delivers *m'* would already have delivered *m*.

Hybrid variants

- Since FIFO and causal orderings are orthogonal to total ordering, we can have hybrid ordering protocols:
  - FIFO-total hybrid protocol satisfies both FIFO and total orders
  - Causal-total hybrid protocol satisfies both Causal and total orders

Total ordering != sequential consistency

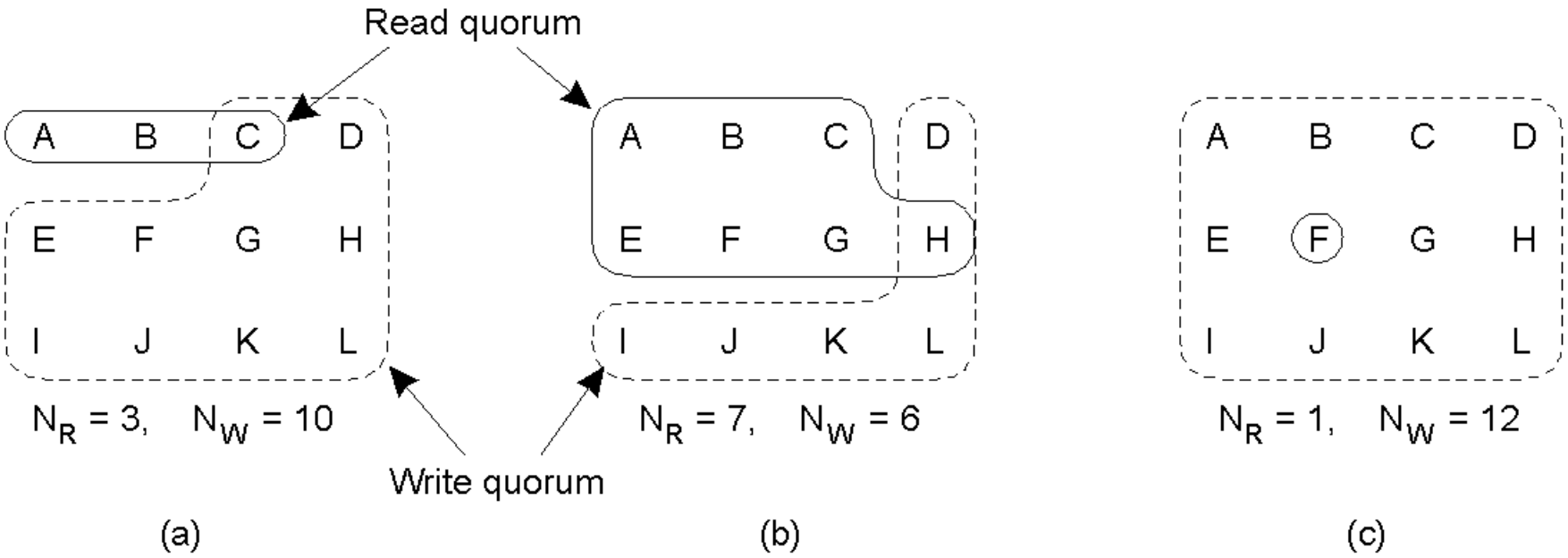
- If clients can not communicate with each other while waiting for responses, and each front end's requests are served in FIFO order, i.e., the front end awaits a response before making the next request,
- then FIFO order is automatically satisfied,
- therefore, active replication with totally ordered multicast can guarantee sequential consistency.

- If clients can communicate with each other while waiting for responses,
- then to guarantee request processing in *happened-before* order we have to replace the multicast with one that is both causally and totally ordered.

Quorum-based protocols

- Assign a number of votes to each replica
  - Let N be the total number of votes
  - Define R = read quorum, W = write quorum
  - R+W > N (total number of votes)
  - W > N / 2 (half of the votes)
  - Ensures that each read quorum intersects a write quorum, and two write quora will intersect
- Each replica has a version number that is used to detect if the replica is up to date.
- Every reader sees at least one copy of the most recent write (takes one with most recent version number).
- Only one writer at a time can achieve write quorum.

Quorum-based protocols



Three examples of the voting algorithm:

a) A correct choice of read and write set
b) A choice that may lead to write-write conflicts
c) A correct choice, known as ROWA (read one, write all)

Possible policies

- ROWA: R=1, W=N
  - Fast reads, slow writes (and easily blocked)
- RAWO: R=N, W=1
  - Fast writes, slow reads (and easily blocked)
- Majority: R=W=N/2+1
  - Both moderately slow, but extremely high availability
- Weighted voting
  - give more votes to “better” replicas

Scaling

- None of the protocols for sequential consistency scale
- To read or write, you have to either
  - (a) contact a primary copy
  - (b) use reliable totally ordered multicast
  - (c) contact over half of the replicas
- All this complexity is to ensure sequential consistency
  - Note: even the protocols for causal consistency and FIFO consistency are difficult to scale if they use reliable multicast
- Can we weaken sequential consistency without losing some important features?

Eventual consistency

- Sequential consistency requires that at **every point**, **every replica** has a value that could be the result of the globally-agreed sequential application of writes
- This does not require that all replicas agree at all times, just that they always take on the same sequence of values
- Why not allow **temporary** out-of-sequence writes?
  - Note: all forms of consistency weaker than sequential allow replicas to disagree forever
- We want to allow out-of-order operations, but only if the effects are temporary
- All writes **eventually** propagate to all replicas
- Writes, when they arrive, are applied in the same order at all replicas

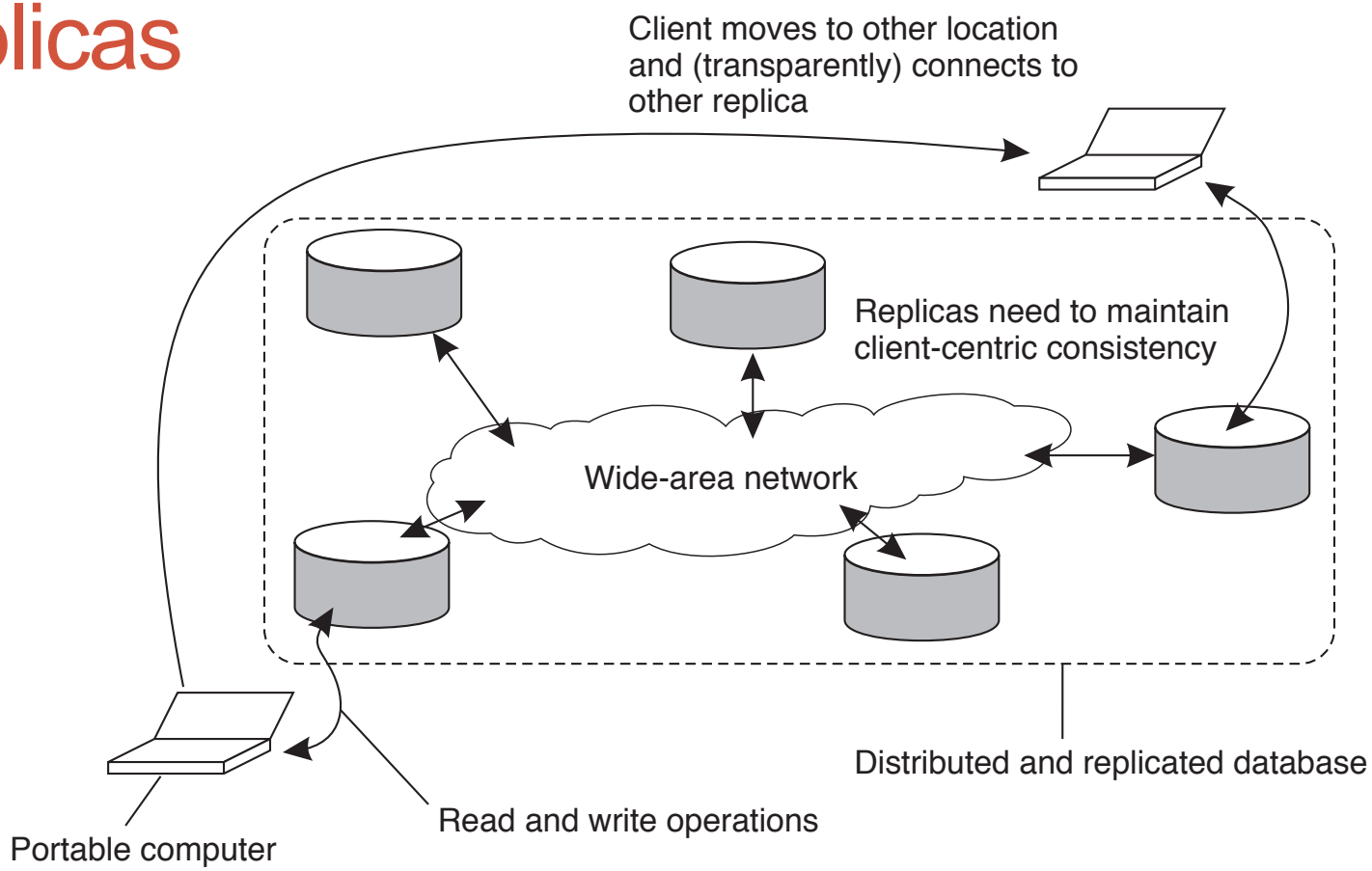
Epidemic protocols

- Update propagation for systems that only need eventual consistency
- Randomized approaches based on the theory of epidemics
  - Upon an update, try to “infect” other replicas as quickly as possible
  - Pair-wise exchange of updates (like pair-wise spreading of a disease)
  - *infective server*: server with an update it is willing to spread
  - *susceptible server*: server that is not yet updated

Spreading an epidemic

- Anti-entropy propagation model
  - A server P picks another server Q at random, and exchanges updates
  - Three approaches
    - P only pushes updates to Q
    - P only pulls new updates from Q
    - P and Q send updates to each other
  - If many infective servers, pull-based approach is better
  - If only one infective server, either approach will eventually propagate all updates
- Rumor spreading (gossiping) will speed up propagation
  - If server P has been updated, it randomly contacts Q and tries to push the update to Q; if Q was already updated by another server, with some probability (1/k), P loses interest in spreading the update any further

Example: mobile user accessing different replicas



Eventual consistency is easy to implement if a client always accesses the same replica.

If the user disconnects and reconnects to another replica, it may observe inconsistencies in the data store.

Session guarantees

- When a client moves around and connects to different replicas, strange things can happen
  - Updates you just made are missing
  - Database goes back in time
- Design choice:
  - Insist on stricter consistency
  - Enforce some “session” guarantees

Client-centric consistency models

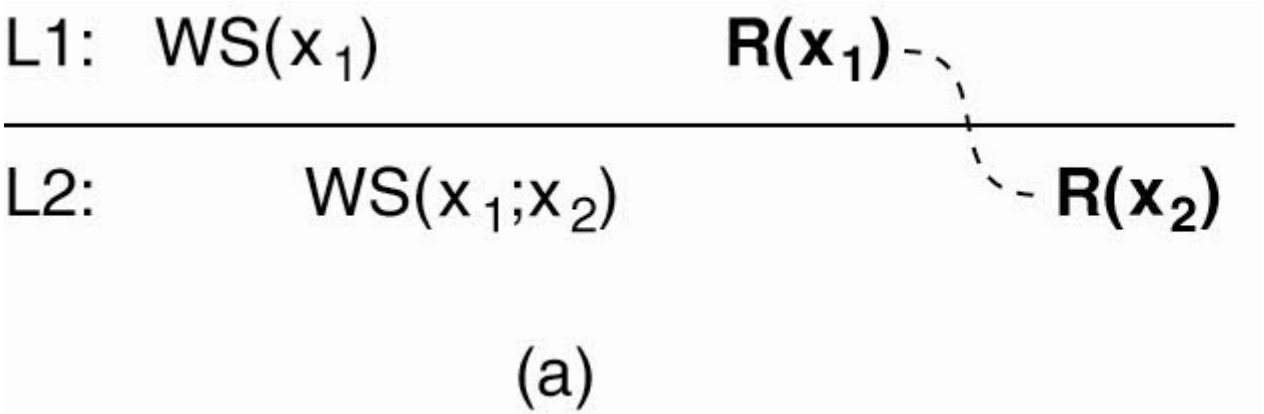
- **Client-centric consistency** is consistency from a single client's point of view
- Provides guarantees for a single client's access to a replicated data store
- No guarantees concerning accesses by different clients
- Common client centric models:
  - 1. Monotonic reads
  - 2. Monotonic writes
  - 3. Read your writes
  - 4. Write follows reads

Monotonic reads (1)

- 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 that process will always return that same value or a more recent value.

- Example of non-monotonic-read consistent:
  - Get a list of email messages
  - When attempting to read one, get “message doesn’t exist error”

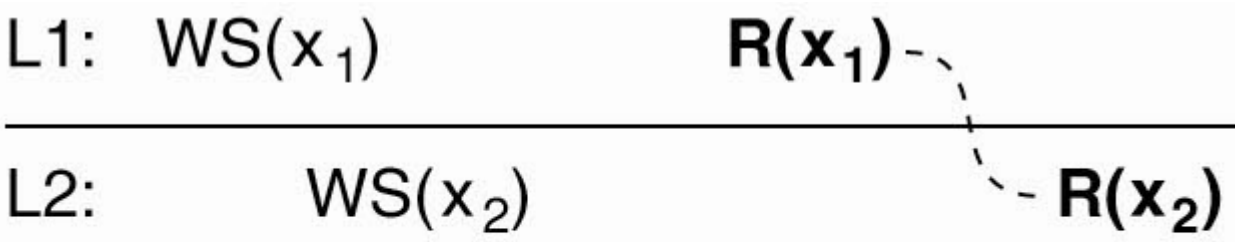
Monotonic reads (2)



The read operations performed by a single process P at two different local copies of the same data store.

(a) A monotonic-read consistent data store.

Monotonic reads (3)



(b)

The read operations performed by a single process P at two different local copies of the same data store.

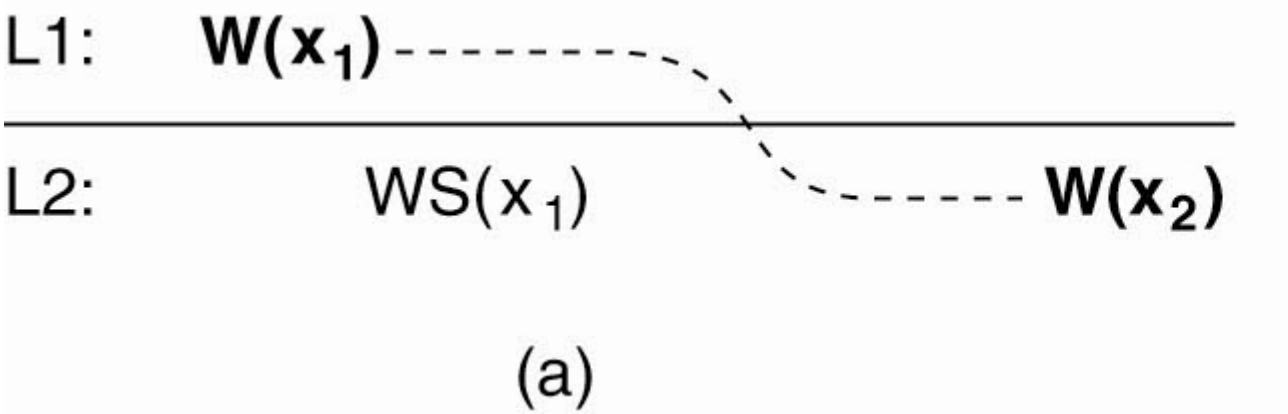
(b) A data store that does not provide monotonic reads.

Monotonic writes (1)

- In a monotonic-write consistent store, the following condition holds:
  - A write operation by a process on a data item x
  - is completed before any successive write operation on x by the same process.

- Example non-monotonic-write consistent:
  - Update to library made
  - Update to application using the library made
  - Application depending on the new library show up where the new library doesn't

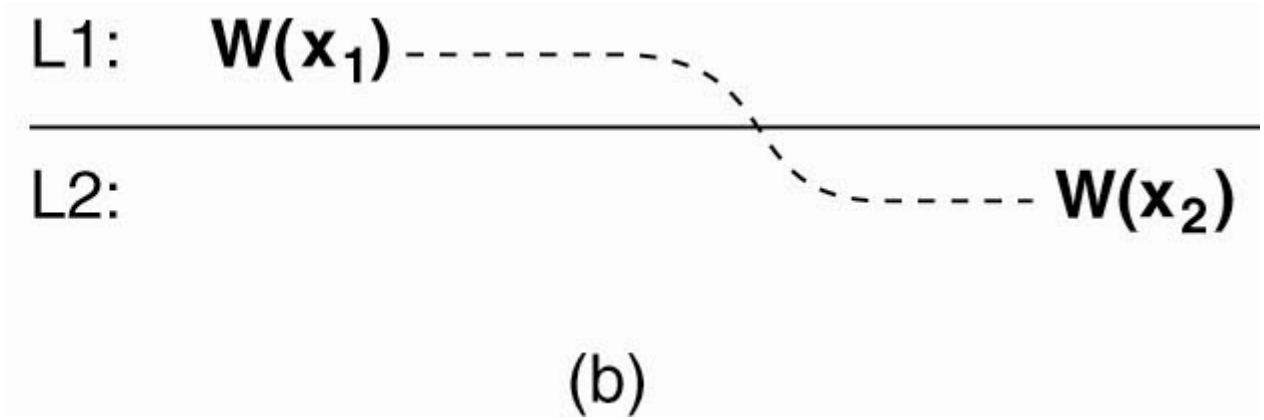
Monotonic writes (2)



The write operations performed by a single process P at two different local copies of the same data store.

(a) A monotonic-write consistent data store.

Monotonic writes (3)



The write operations performed by a single process P at two different local copies of the same data store.

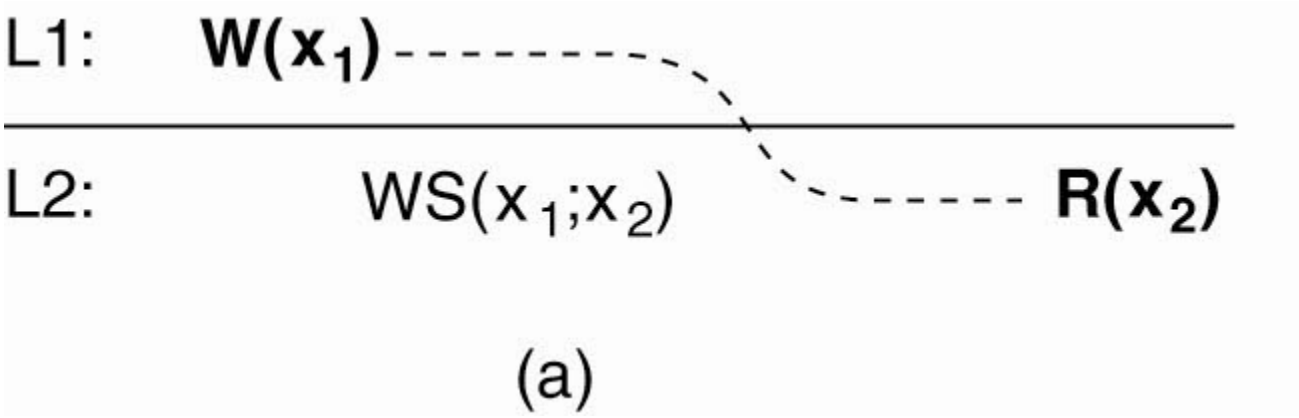
(b) A data store that does not provide monotonic-write consistency.

Read your writes (1)

- 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.

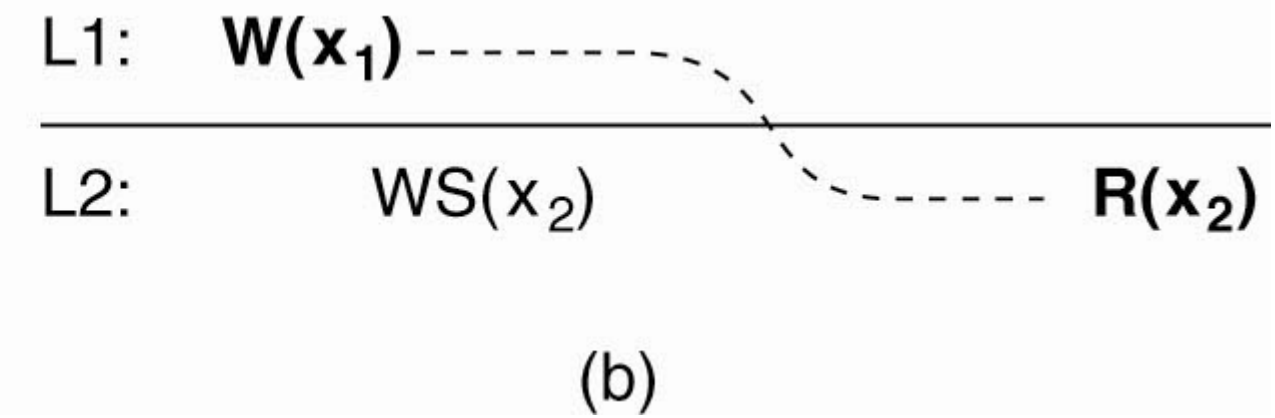
- Example non-read-your-writes consistent:
  - Deleted emails re-appear

Read your writes (2)



(a) A data store that provides read-your-writes consistency.

Read your writes (3)

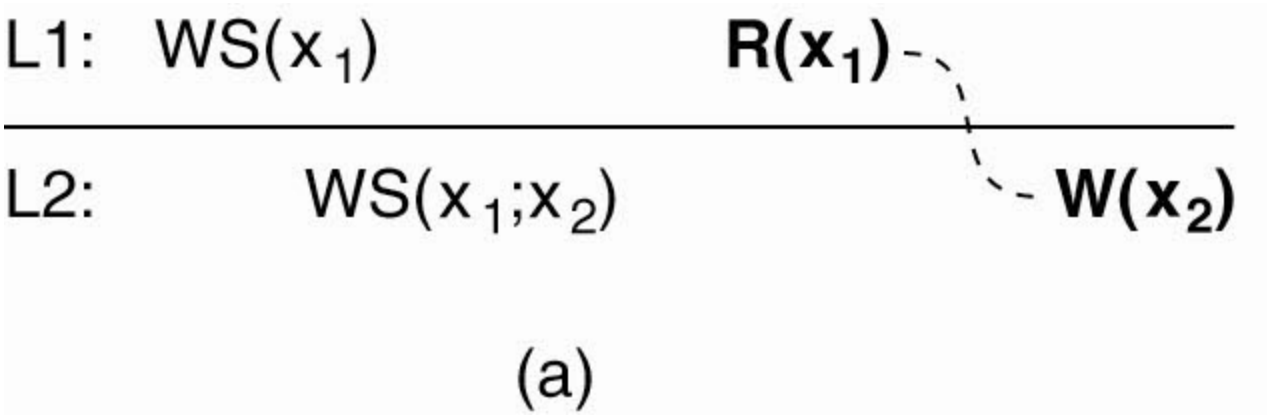


(b) A data store that does not.

Writes follow reads (1)

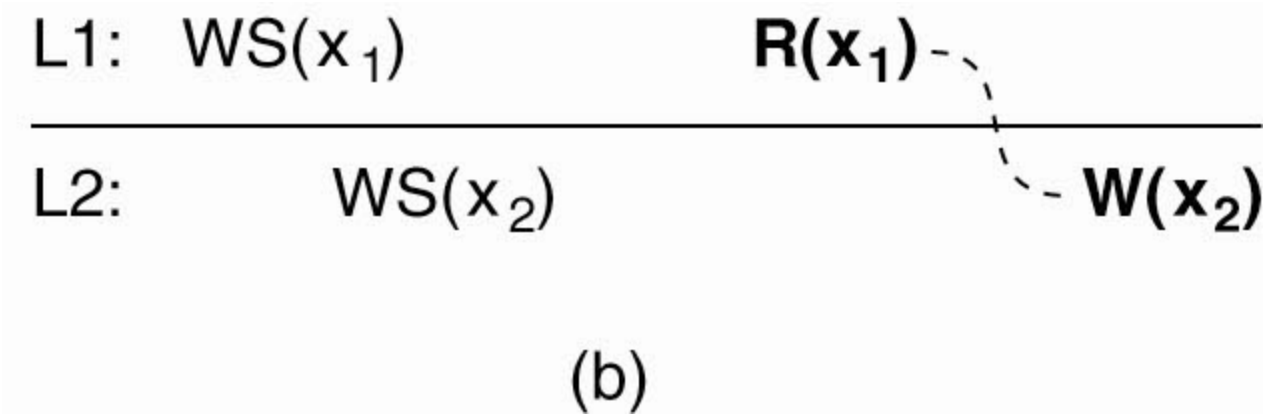
- A data store is said to provide writes-follow-reads consistency, if the following holds:
  - A write operation by a process …
    - on a data item x following a previous read operation on x by the same process …
    - is guaranteed to take place on the same or a more recent value of x that was read.

Writes follow reads (2)



(a) A writes-follow-reads consistent data store.

Writes follow reads (3)



(b) A data store that does not provide writes-follow-reads consistency.

Supporting session guarantees

- Two sets:
  - Read-set: set of writes that are relevant to session reads
  - Write-set: set of writes performed in session
- Update dependencies captured in read sets and write sets
- Causal ordering of writes
  - Use vector timestamps

- We will see more examples of eventually consistent systems in next lecture.

Reading

- Chapter 7 of TBook
- Chapters 15 and 18 of CBook