Distributed Systems: Consistency and Replication

Alessandro Sivieri

Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy

sivieri@elet.polimi.it

http://corsi.dei.polimi.it/distsys

Slides based on previous works by Matteo Migliavacca and Alessandro Margara

Politecnico di Milano

Whoami

- Ing. Alessandro Sivieri
  - Ph.D. student @ DeepSE
  - Teaching assistant for
    - Distributed Systems
    - Middleware Technologies for Distributed Systems
- How to contact me
  - http://home.dei.polimi.it/sivieri
  - http://corsi.dei.polimi.it/distsys
  - Building 22 (via Golgi 42), third floor, int. 301
    - Please, come only with an appointment
  - Phone: 3707
  - Email: sivieri@elet.polimi.it

Politecnico
di Milano

Speaking about emails…

Mail

Contacts

Tasks

COMPOSE MAIL

Inbox

Important

Sent Mail

Drafts

Spam

Send | Save Now | Discard

From: [redacted]

To: "Alessandro Sivieri" <sivieri@elet.polimi.it>,

Add Cc | Add Bcc

Subject: [distsys] Some problems

Attach a file Insert: Invitation

« Plain Text

Politecnico
di Milano

Why Replicate Data?

- Improve performance
  - Local access is faster than remote access
  - Sharing of workload
- Increase availability
  - Data may become unavailable due to failure
    - *availability* = $1 - p^n$ (p probability of failure, n # of replicas)
  - Data may be available only intermittently in a mobile setting
    - Replicas provide support for disconnected operations
- Achieve fault tolerance
  - Related to availability
  - It becomes possible to deal with incorrect behaviors through redundancy

Politecnico di Milano

Examples

- Web caches
  - Browsers store locally a copy of the page, to improve access time
  - Caches may be present also in the network
    - e.g., on Web proxies
  - Assumption: pages are changed infrequently
- Distributed file systems
  - File replication (e.g., in Coda) allows for faster access and disconnected operations
  - User files are reconciled against servers upon reconnection
  - Assumption: conflicts (files modified by more than a user) are infrequent
- Domain Name Service
- Caches on processors
- …

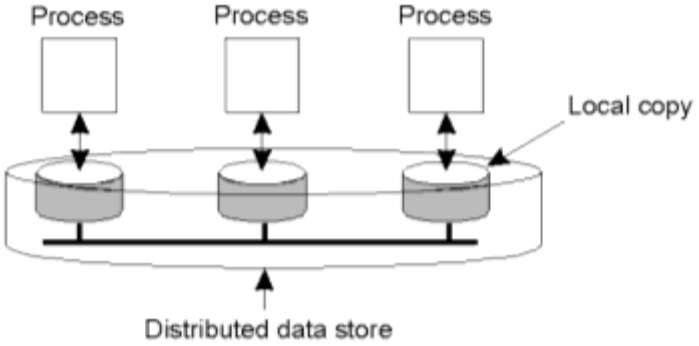
Politecnico
di Milano

Challenges

- Main problem: ***consistency***
  - Changing a replica demands changes to all the others
  - We must ensure that accessing a replica has the same effect as accessing the original
  - Goal: provide consistency with limited communication overhead
- Scalability vs. performance
  - Replication may actually degrade performance!
- Different consistency requirements depending on the application scenario
  - Data-centric vs. client-centric

Politecnico di Milano

Consistency Models



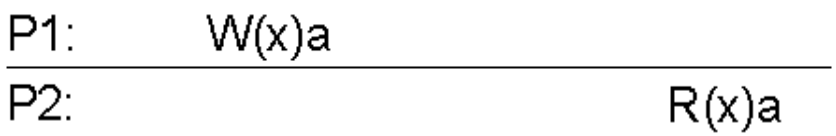
- Focus on a (distributed) ***data store***
  - E.g., shared memory, filesystem, database, …
- Ideally, a read should show the result of the last write
  - Hard to determine without a global clock
- A ***consistency model*** is a contract between the processes and the data store
  - It weakens the model by providing less guarantees
  - Tradeoffs: guarantees, performance, ease of use
- We now review the most important models
  - Operations ordered along the time axis
  - *W(x)a* means that the value a is written on the data item x
  - *R(x)a* means that the value a is read from the data item x

Politecnico di Milano

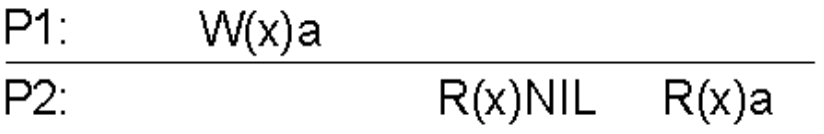
Strict Consistency

> ***"Any read on data item x returns the value of the most recent write on x"***

- All writes are instantaneously visible, global order is maintained
- "Most recent" is ambiguous without global time
- Even with global time, it can be impossible to guarantee
- In practice, possible only within a uniprocessor machine
- Example:



Consistent



NOT Consistent

Politecnico di Milano

Sequential Consistency

"The result is the same as if the operations by all processes were executed in some sequential order, and the operations by each process appear in this sequence in the order specified by its program"

- In other words:
  - operations within a process may not be re-ordered
  - all processes see the same interleaving
  - it may differ from "real" sequence
- Does not rely on time

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

Consistent

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

NOT Consistent

Politecnico
di Milano

Sequential Consistency

- Why sequential consistency?
- Shared memory on multi-processor computers
- "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs"
  - L. Lamport, ACM Transactions on Programming Languages and Systems, 1979

Politecnico di Milano

Sequential Consistency

- Consider the following program

Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

- All the following (and many others) executions are *sequential-consistent*

x = 1;	x = 1;	y = 1;	y = 1;
print (y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x,z);	print (x, y);	z = 1;
print (x, z);	print (y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111

Politecnico di Milano

Linearizability

"The system is sequentially consistent; moreover, if $ts_{OP1}(x)<ts_{OP2}(y)$ then operation OP1(x) precedes OP2(y) in the operation sequence."

- Stronger than sequential, weaker than strict
  - it assumes globally available clocks but only with finite precisions
- The previous example of sequential is also linearizable if we assume the write at P1 has a timestamp greater than the write at P2
- Practical interest only for formal verification

Politecnico
di Milano

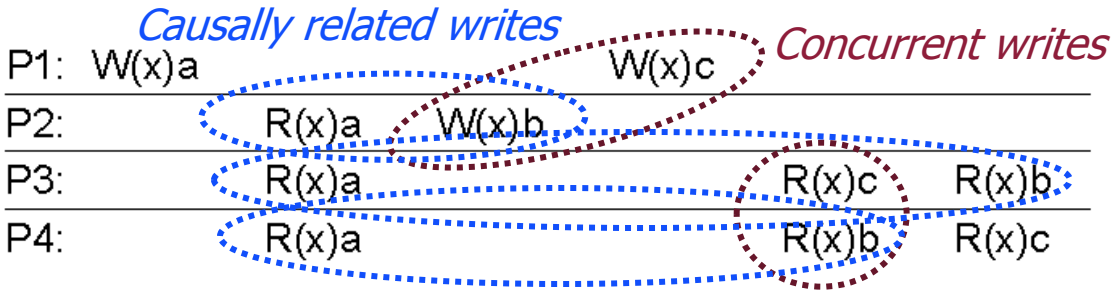
Causal Consistency

- Consider a Mailing List discussion:
  - A says "Linux rulez"
  - B says "Please, shut up. Linux sucks"
- If C first sees B and then A, he cannot understand what is going on
  - therefore, everybody else must see A's message before B's one
- Consider other two messages:
  - A says "Security exploit in Windows"
  - B says "Google $product exits beta testing"
- The two messages are not related to each other
  - the order in which they are seen from other members does not matter

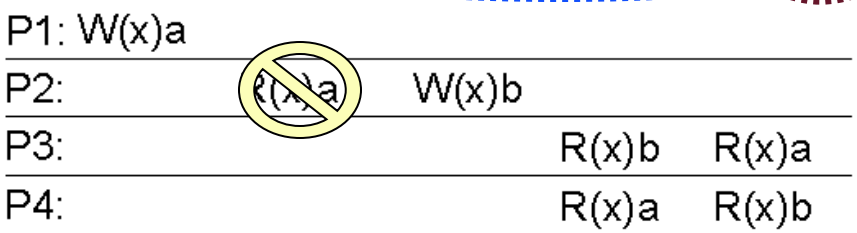
Causal Consistency

"Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in any order at different machines."

- Weakens sequential consistency based on Lamport's notion of happened-before
  - Lamport's model dealt with message passing; here causality is between reads and writes



Consistent



NOT Consistent

(it becomes so if P2's R(x) is removed)

Politecnico
di Milano

Causal Consistency

- Causal consistency defines a causal order among operations
- More precisely, causal order is defined as follows:
  - A write operation W by a process P is causally ordered after every previous operation O by the same process
    - Even if W and O are performed on different variables
  - A read operation by a process P on a variable x is causally ordered after a previous write by P on the same variable
  - Causal order is transitive
- It is not a total order
  - Operations that are not causally ordered are said to be concurrent

Politecnico
di Milano

Causal Consistency

- Why causal consistency?
- Easier to guarantee within a distributed environment
  - Less overhead
- Easier to implement
- "Causal memory meets the consistency and performance needs of distributed applications!"
  - Ahamad et al., 1994

Politecnico di Milano

FIFO Consistency

"Writes done by a single process are seen by all others in the order in which they were issued; writes from different processes may be seen in any order at different machines."

- In other words, causality across processes is dropped
- Also called PRAM consistency (Pipelined RAM)
  - If writes are put onto a pipeline for completion, a process can fill the pipeline with writes, not waiting for early ones to complete
- Easy to implement (sequence numbers on writes)

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

Consistent

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)c	R(x)b	R(x)a

Not Consistent

Politecnico di Milano

Consistency Models and Synchronization

- FIFO consistency still requires all writes to be visible to all processes, even those that do not care
- Moreover, not all writes need be seen by all processes
  - e.g., those within a transaction/critical section
- Some consistency models introduce the notion of ***synchronization variables***
  - Writes become visible only when processes explicitly request so through the variable
  - Appropriate constructs are provided, e.g., **`synchronize(S)`**
- It is up to the programmer to force consistency when is really needed, typically:
  - At the end of a critical section, to distribute writes
  - At the beginning of a "reading session" when writes need to become visible

Politecnico di Milano

Weak Consistency

1. ***Access to synchronization variables is sequentially consistent;***
2. ***No operation on a synchronization variable is allowed until all previous writes have completed everywhere;***
3. ***No read or write to data are allowed until all previous operations to synchronization variables have completed.***

- It enforces consistency on a ***group*** of operations
- It limits only the ***time*** when consistency holds, rather than the form of consistency
- Data may be inconsistent in the meanwhile

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

Consistent

P1: W(x)a	W(x)b	S				
P2:				S	R(x)a	

NOT Consistent

Politecnico di Milano

Release Consistency

- Problem: the data store cannot distinguish between a synchronization request for disseminating writes or for reading consistent data
  - Unnecessary overhead
- Solution: introduce different synchronization operations
  - ***Acquire*** indicates critical region is about to be entered
  - ***Release*** indicates critical region has just been exited
  - Do not necessarily apply to the whole store, but can be defined on single items (said to be ***protected***)

1. ***Before a read or write is performed, all previous acquires done by the process must have completed successfully;***
2. ***Before a release is allowed, all previous reads and writes done by the process must have been completed;***
3. ***Accesses to synchronization variables are FIFO consistent.***

Politecnico di Milano

More on Release Consistency

P1:	Acq(L)	W(x)a	W(x)b	Rel(L)						
P2:							Acq(L)	R(x)b	Rel(L)	
P3:										R(x)a

- Note: P2's acquire must ***wait*** for P1's release
- On release, protected data that changed are propagated to other copies
- Two ways to achieve this:
  - ***Eager*** release consistency
    - On release, all updates are pushed to other replicas
    - Potentially sends data to processes that will not use it
  - ***Lazy*** release consistency
    - On release, nothing is sent
    - On acquire, acquiring process must get latest version of the data from other processes
    - Tends to be more bandwidth efficient
- Release consistency also used also for ***barrier synchronization***
  - Program divided in phases instead of critical sections
  - To enter next phase all processes must have completed previous one
  - Implemented as a release followed by an acquire
  - Used in SIMD / SPMD processors

Politecnico
di Milano

Entry Consistency

- Explicitly associates each shared data item with a synchronization variable
  - Still accessed through acquire and release
  - Reduces update overhead, increases complexity of access
  - Improves parallelism, enabling simultaneous access to multiple critical sections
- Two modes of access to a synchronized variable
  - ***Non-exclusive***: multiple processes can hold read locks simultaneously
  - ***Exclusive***: only one process holds the lock to the variable; to be granted, must guarantee that no other process holds (even a non-exclusive) lock

Politecnico di Milano

Entry Consistency

1. *An acquire access of a synchronization variable is not allowed to perform wrt a process until all updates to the guarded shared data have been performed wrt that process;*
2. *Before an exclusive mode access to a synchronization variable by a process is allowed to perform wrt that process, no other process may hold the synchronization variable, not even in non-exclusive mode;*
3. *After an exclusive mode access to a synchronization variable has been performed, any other process' next non-exclusive mode access to that synchronization variable may not be performed until it has performed wrt to that variable's owner*.

Entry Consistency

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)						
P2:							Acq(Lx)	R(x)a				R(y)NIL
P3:									Acq(Ly)	R(y)b		

- Properties:
  - On acquire, all guarded data must be made visible
  - For exclusive mode to be granted, no other process can be holding any kind of lock
  - After a data item has been accessed in exclusive mode, all future accesses by processes (other than the one that did the write) must go through the acquire process
- Complicated to use!
  - But may be useful if encapsulated in a distributed object model

Politecnico di Milano

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 (nonunique) 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 used. Writes from different processes may not always be seen in that order

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered

Politecnico di Milano

Exercise

Consider the following schedule:

P0: W(x) 1 R (x) 2 W(x) 3

P1: R (x) 1 W(x) 2 R(x) 3

P2: R (x) K R (x) ?

Complete the table below, by showing, for each value of K, the set of values (1, 2, 3) that process P2 can read during its second operation when adopting a FIFO or a sequential consistency model.

Value of K	Consistency model	Set of allowed values
1	FIFO	
1	Sequential	
2	FIFO	
2	Sequential	
3	FIFO	
3	Sequential	

Politecnico di Milano

Exercise

Consider the following processes (A, B, C), running concurrently and acting on variables x, y, z.
Assume the following initial values: x = 0; y = 0; z = null.

Process A	Process B	Process C

x = 1;
if (y==0) {
  z = 'A';
}
	y = 1;
	if (x==0) {
	  z = 'B';
	}
		Print z;
		Print x;
		Print y;

Complete the following table, showing which sets of values represent a valid output for process C when using a sequential, causal, or FIFO consistency model.

z	x	y	Seq	Causal	FIFO
A	0	0			
B	0	0			
A	0	1			
B	0	1			
A	1	0			
B	1	0			
A	1	1			
B	1	1			

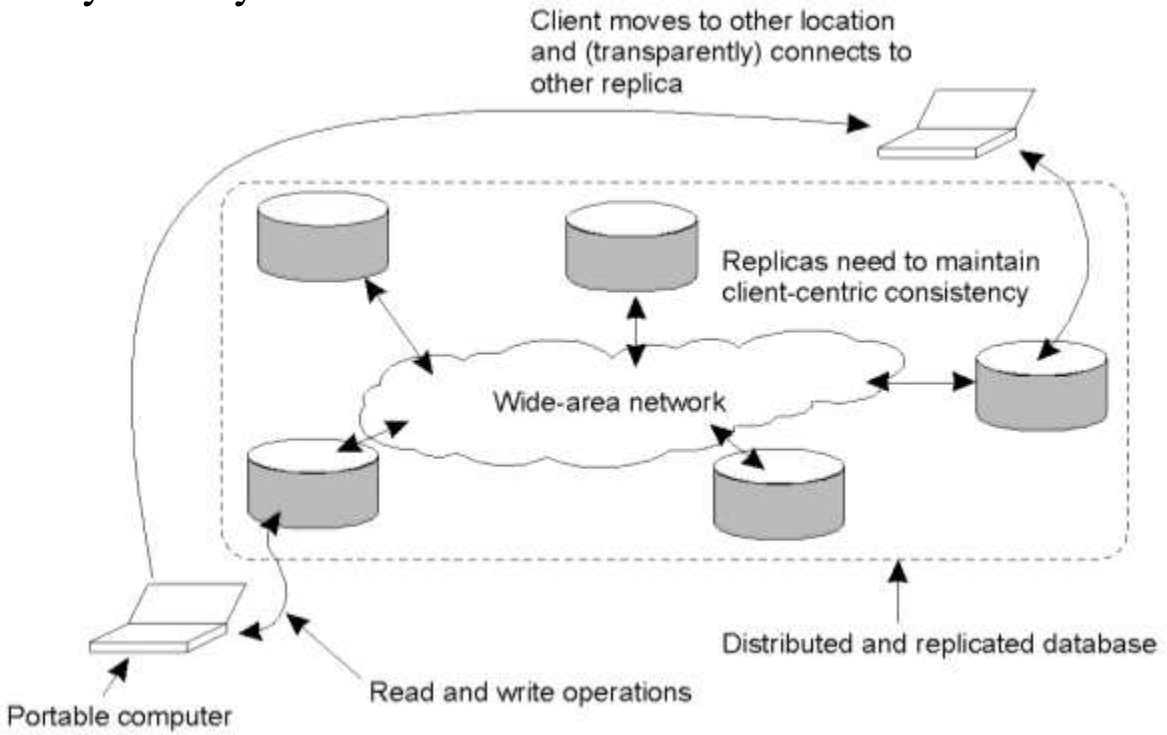
Politecnico di Milano

Eventual Consistency

- The models considered so far are ***data-centric***
  - Provide a system-wide consistent data view in the face of simultaneous, concurrent updates
- However, there are situations where there are:
  - no simultaneous updates (or can be easily resolved)
  - mostly reads
- Examples: Web caches, DNS, USENET news
- In these systems, ***eventual consistency*** is often sufficient
  - Updates are guaranteed to eventually propagate to all replicas
- However, eventual consistency is easy to implement only if reads involve always the same replica …

Client-Centric Consistency

- The problem can be tackled by introducing ***client-centric consistency*** models
  - Provides guarantees about accesses to the data store from the perspective of a single client
- Pioneered by the Bayou system



Monotonic Reads

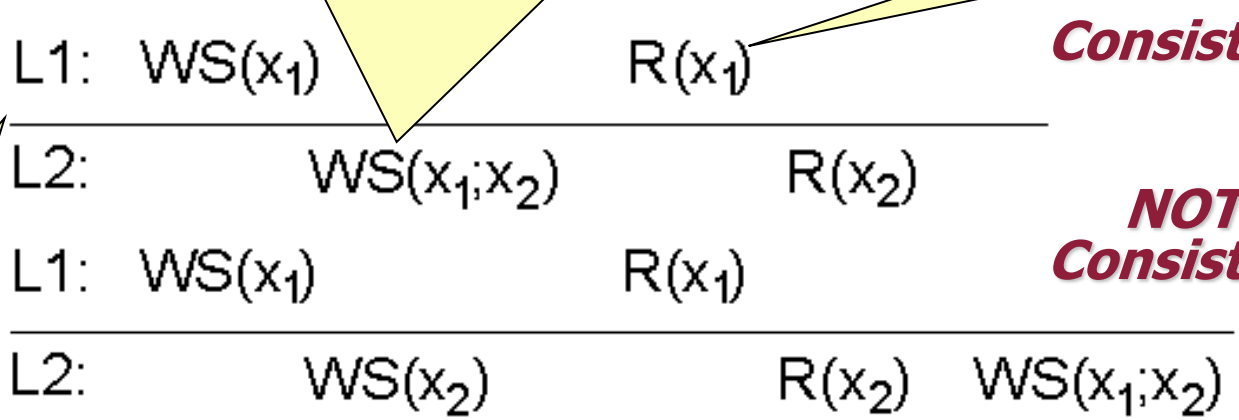
"If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value."

- Once a process reads a value from a replica, it will never see an older value from a read at a different replica
- Example: e-mail mirrors

The set of writes known at a data store contains the update of x at L1 and the update of x at L2, in this order

Read on x_1, the value of x at data store L1

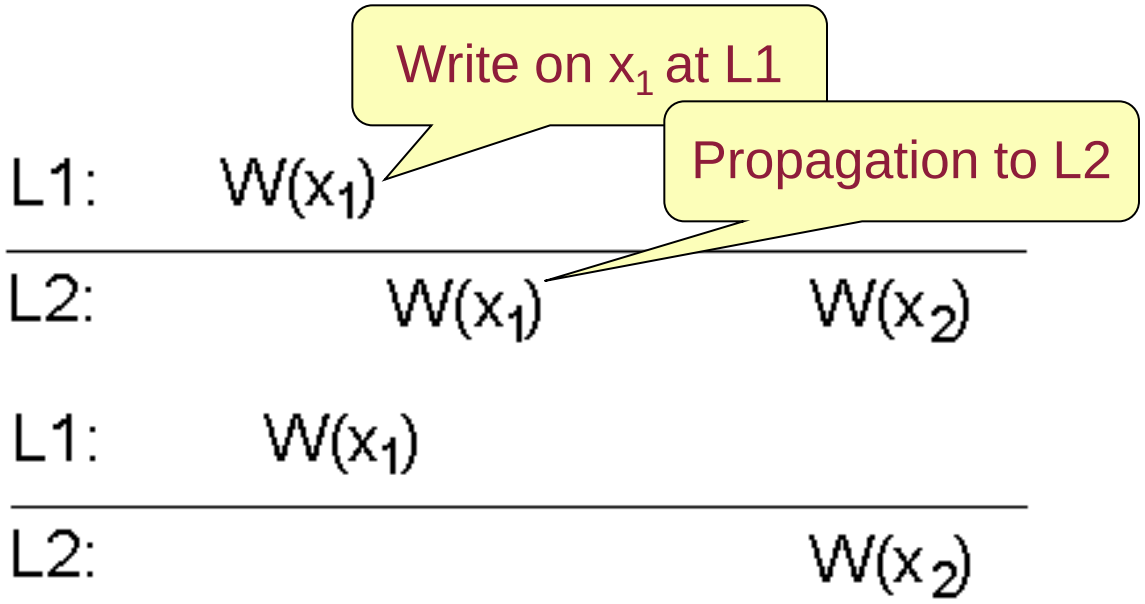
L1 and L2 are local copies of the data store, accessed by the same process



Monotonic Writes

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

- Similar to FIFO consistency, although this time for a single process
- A weaker notion where ordering does not matter is possible if writes are commutative
- x can be a large part of the data store (e.g., a code library)



Consistent

NOT Consistent

Politecnico di Milano

Read Your Writes

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

- Examples: updating a Web page, or a password

L1: $W(x_1)$

L2: $WS(x_1;x_2)$ $R(x_2)$

Consistent

L1: $W(x_1)$

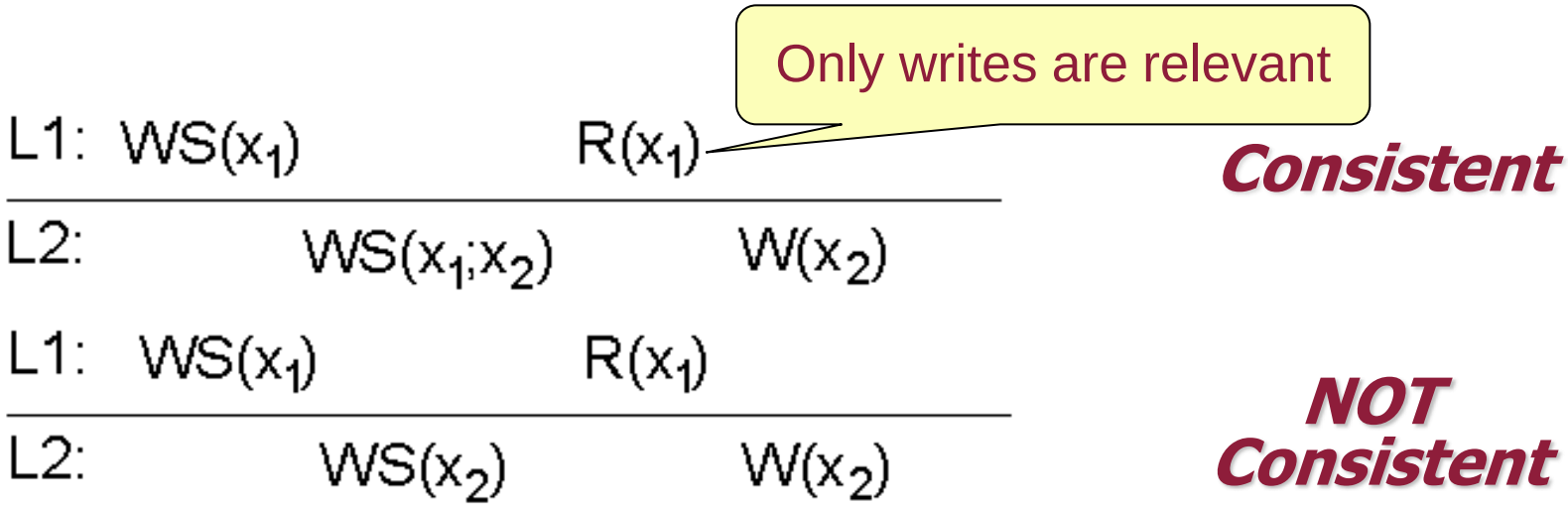
L2: $WS(x_2)$ $R(x_2)$

NOT Consistent

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 more recent value of x that was read."***

- Example: guarantee that users of a newsgroup see the posting of a reply only after seeing the original article



Politecnico di Milano

Client-Centric Consistency Implementation

- Each operation gets a unique identifier
  - e.g. ReplicaID + sequence number
- Two sets are defined for each client:
  - read-set: the write identifiers relevant for the read operations performed by the client
  - write-set: the identifiers of the write performed by the client
- *Monotonic-reads*: before reading on L2, the client checks that all the writes in the read-set have been performed on L2
- *Monotonic-writes*: as monotonic-reads but with write-set in place of read-set
- *Read-your-writes*: see Monotonic-writes
- *Write-follow-reads*: firstly, the state of the server is brought up-to-date using the read-set and then the write is added to the write-set

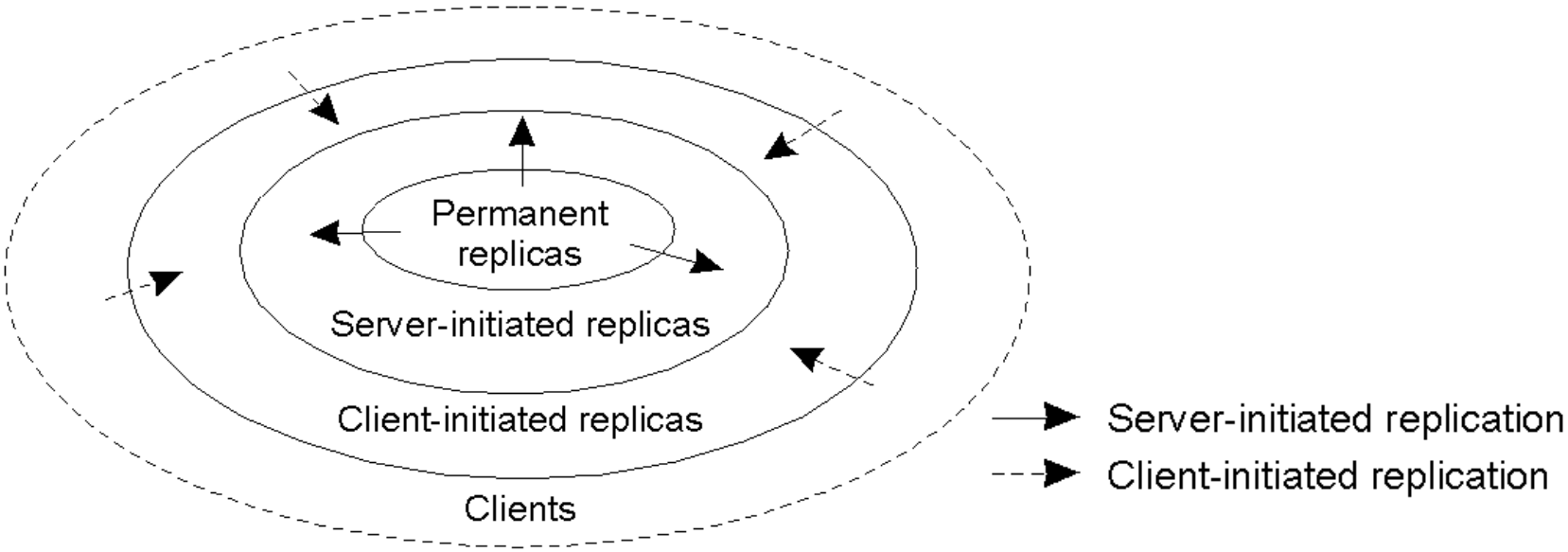
Politecnico
di Milano

Implementing Replication

- How to distribute updates, which in turn involves
  - How to place replicas
  - What to propagate ?
  - How to propagate updates between them
- How to keep replicas consistent

Replica Placement

- Permanent replicas
  - Statically configured, e.g., Web mirrors
- Server-initiated replicas
  - Created dynamically, e.g., to cope with access load
  - Move data closer to clients
  - Often require topological knowledge
- Client-initiated replicas
  - Rely on a client cache, that can be shared among clients for enhanced performance



Politecnico di Milano

Update Propagation

- What to propagate?
  - Perform the update and propagate only a notification
    - Used in conjunction with invalidation protocols: avoids unnecessarily propagating subsequent writes
    - Small communication overhead
    - Works best if #reads << #writes
  - Transfer the modified data to all copies
    - Works best is #reads >> #writes
  - Propagate information to enable the update operation to occur at the other copies
    - Also called ***active replication***
    - Very small communication overhead, but may require unnecessary processing power if the update operation is complex

Update Propagation

- How to propagate?
  - ***Push-based approach***
    - the update is propagated to all replicas, regardless of their needs
    - typically used to preserve high degree of consistency
  - ***Pull-based approach***
    - an update is fetched on demand when needed
    - more convenient if #reads << #writes
    - typically used to manage client caches, e.g., for the Web
  - Leases can be used to switch between the two
    - They were actually developed to deal with replication…

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

Comparison assuming one server and multiple clients, each with their own cache

Politecnico di Milano

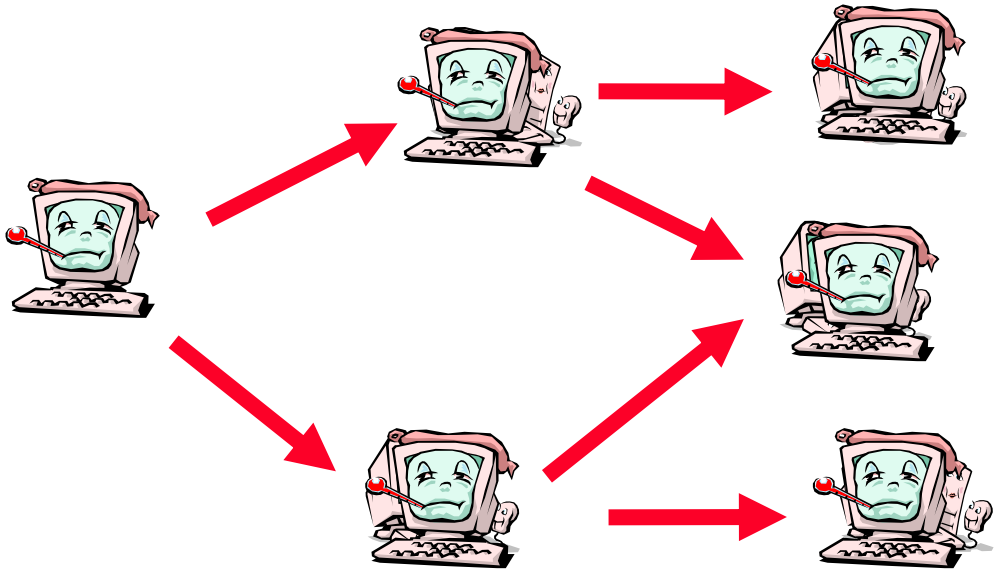
Propagation Strategies

- Anti-entropy:
  - Server chooses another ***at random*** and exchanges updates
  - Options:
    - Push (A⇨B), pull (A⇦B), or push-pull (A⇔B)
    - Positive (changes seen) vs. negative (changes missed) notifications
  - Eventually all servers receive all updates
- Gossiping (or rumor spreading)
  - An update propagation triggers another towards a different server
    - If the update has already been received, the server reduces the probability (e.g., *1/k*) to gossip further
    - Fast spreading but only probabilistic guarantees for propagation
    - May be complemented with anti-entropy
- How to deal with deletion?
  - Treat it as another update ("death certificates" with expiration)
- Several variations available
  - e.g., choosing the gossiping nodes based on connectivity, distance, or entirely at random

Epidemic Algorithms

- Based on the theory of infectious disease spreading
  - Updates (disease) spread by pairwise communication among nodes
  - Unlike diseases, here we want all nodes to become infected

A server with an update to spread is called **infective**; one that has not yet been updated is called **susceptible**; an updated one not willing to re-propagate is called **removed**



Politecnico di Milano

Properties of Epidemic Algorithms

- Intrinsically distributed and redundant
  - Scalable
  - Fault-tolerant
  - Resilient to topological changes
- Probabilistic in nature
  - Cannot provide deterministic guarantees, but
  - Enables relevant overhead savings
- Desirable property: bimodality [Birman99]
  - Almost all or almost nobody receives an update
- Gossip is not broadcast!
  - Broadcast can be regarded as a special case of gossip, with more overhead and less reliability
- Complicate to delete an item
- Applied to several fields, including multicast communication (for spreading and/or recovering events), publish-subscribe, resource location, …

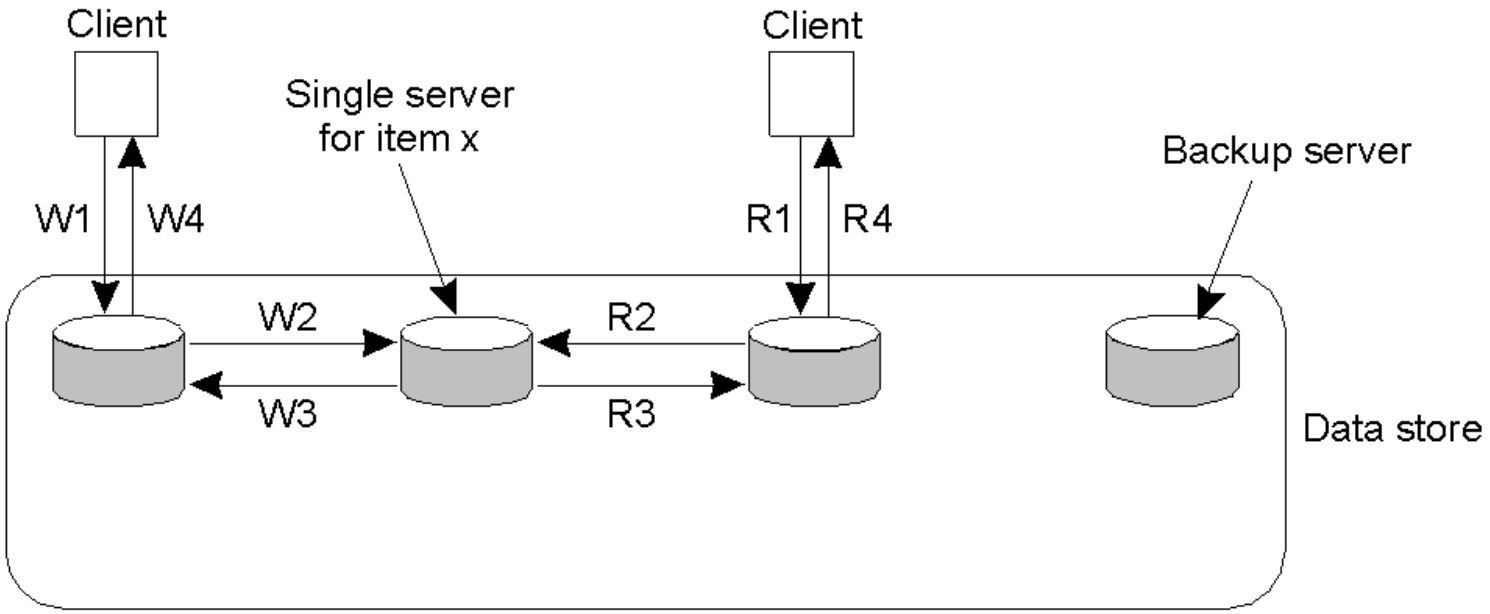
Politecnico di Milano

Primary-Based Consistency Protocols

- All writes go through a single "primary" server, in charge of keeping accesses consistent
- The primary may or may not be complemented by a "backup" server holding replicas
- Variants:
  - Remote-write vs. local write
  - With or without backups

Remote-Write

- Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded
- No backup server

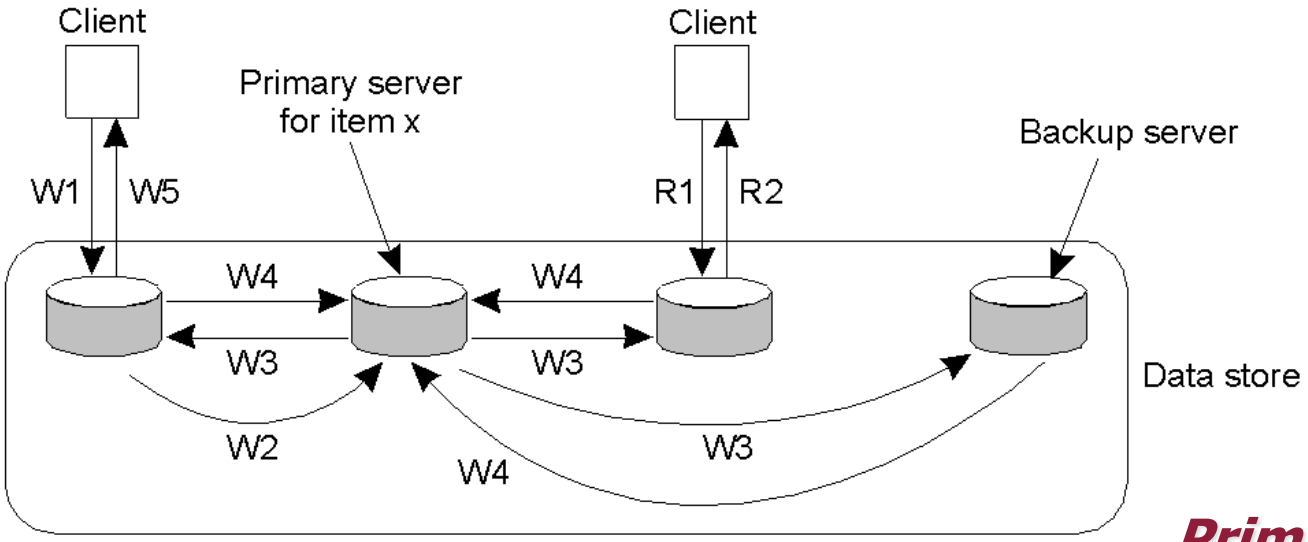


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

R1. Read request
R2. Forward request to server for x
R3. Return response
R4. Return response

Remote-Write (2)

- A remote server is in charge of managing access to data
- Unless backups are provided, no replication
  - Usually servers are somehow replicated
- Straightforward implementation of sequential consistency
- Issue: blocking client while updates propagate
  - Problem: consistency vs. performance in the presence of faults



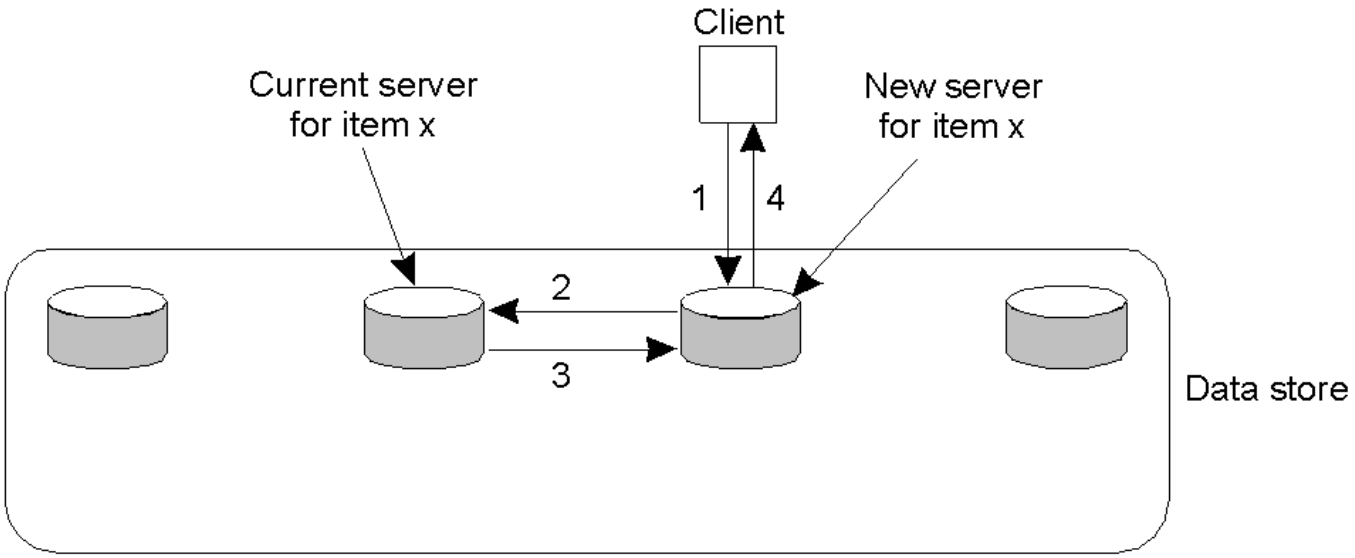
W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Primary-backup, remote-write

Local-Write

- The data item is migrated to the server requesting the update, which therefore becomes the new primary
  - Issue: how to locate the new primary (see previous lectures!)
- Common in distributed shared memory systems

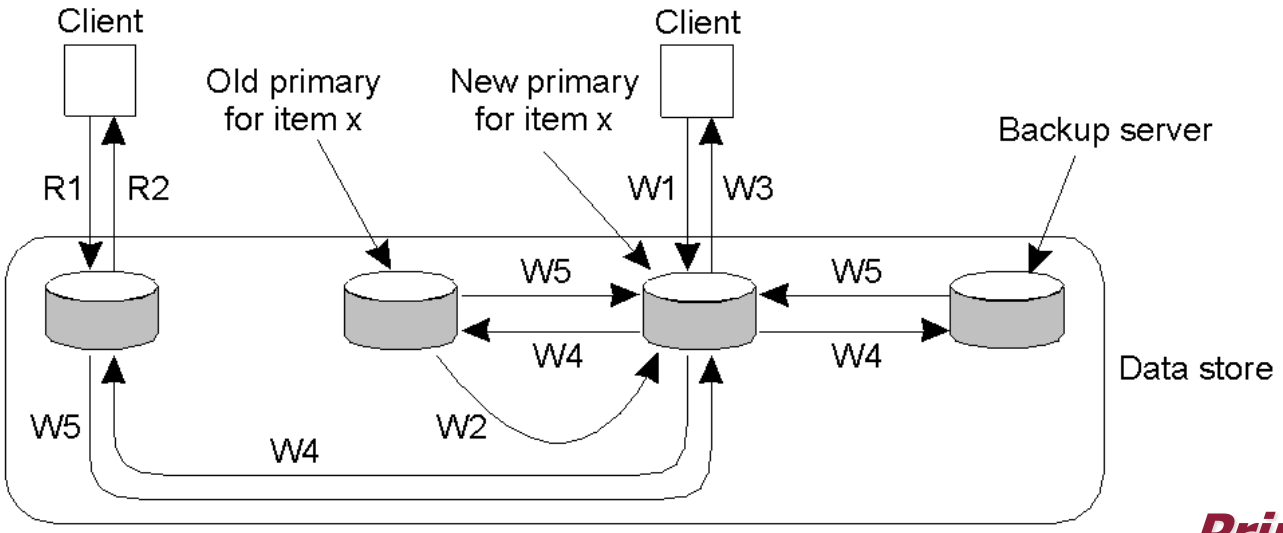


1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

Primary-based, local-write

Local-Write (2)

- The local-write with backups can be used to support disconnected operation
  - The mobile user is the only one who can update, the others can only read



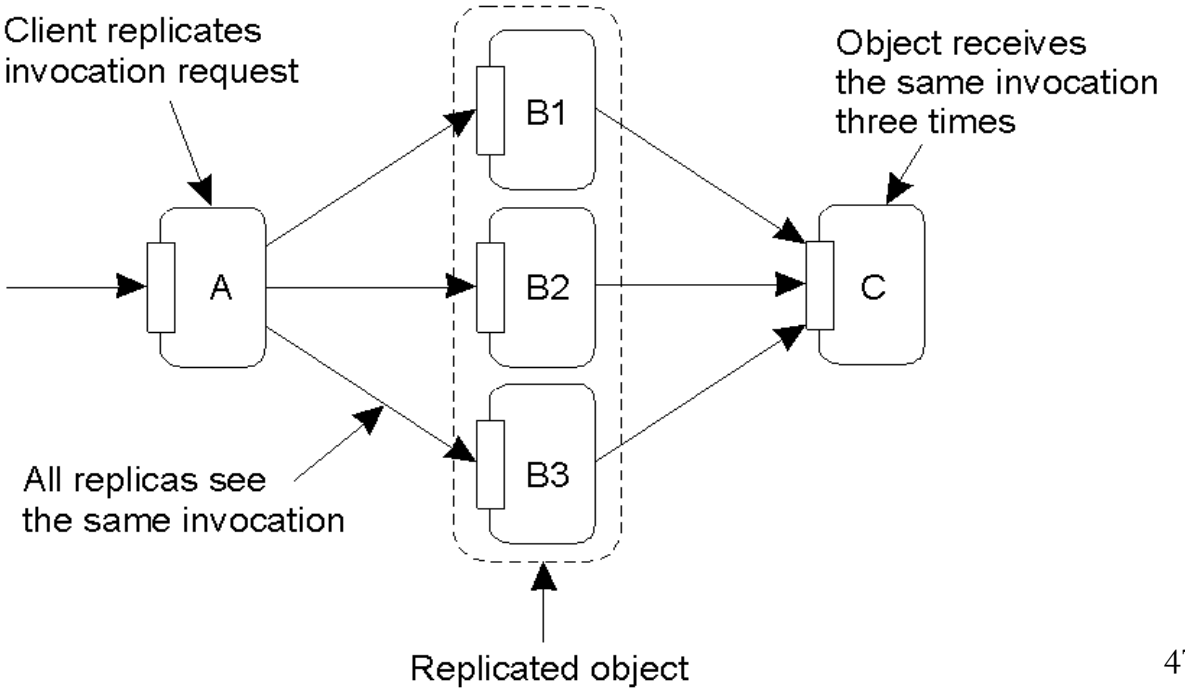
W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Primary-backup, local-write

Replicated-Write Protocols

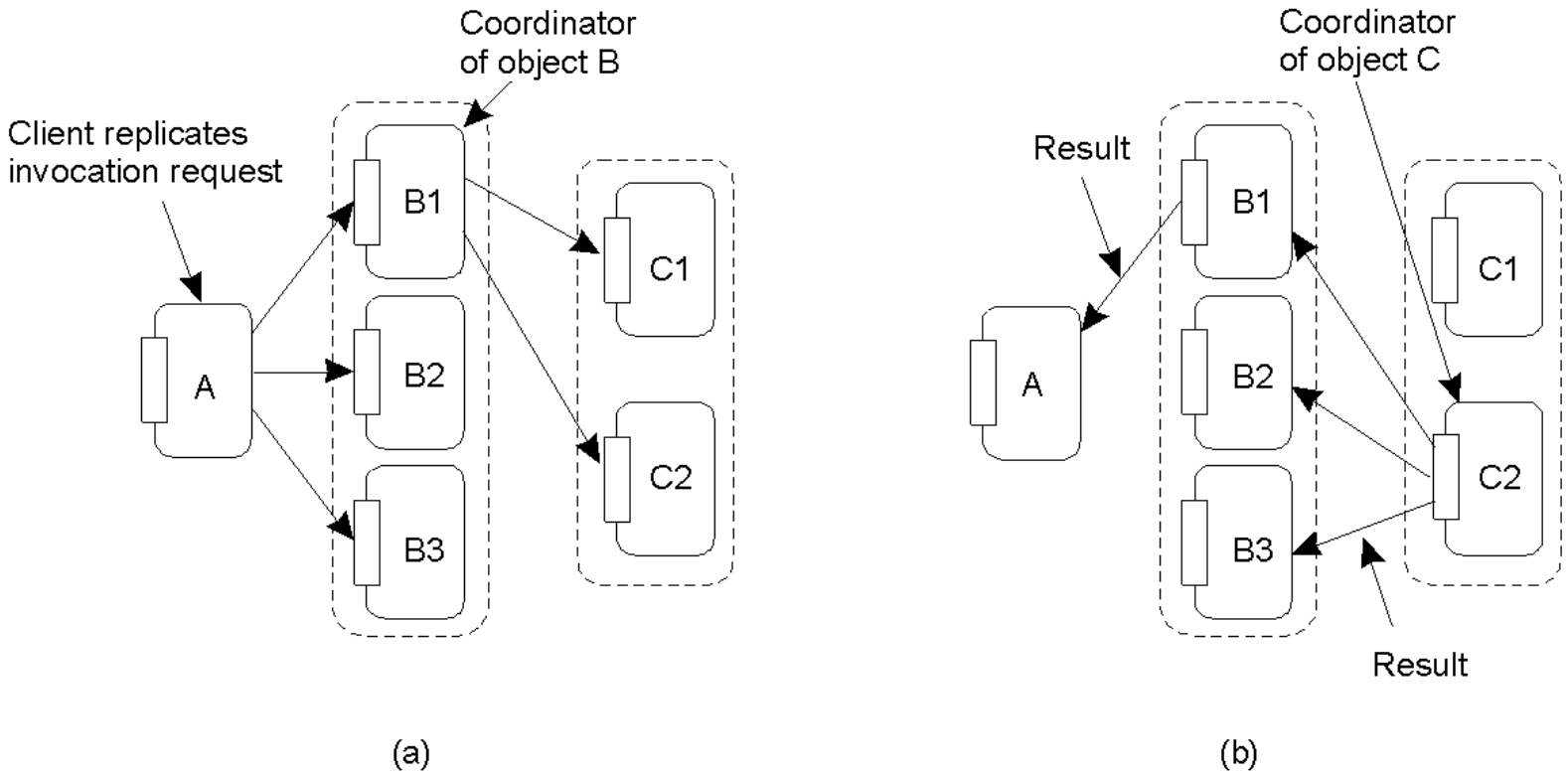
- Write operations can be carried out at multiple replicas
- *Active Replication*
  - Operations (or updates) sent to each replica
  - Ordering of operations must be preserved
    - timestamps or centralized coordinator
  - Problem with multiple invocations



Active Replication (2)

a) Forwarding an invocation request from a replicated object.

b) Returning a reply to a replicated object.

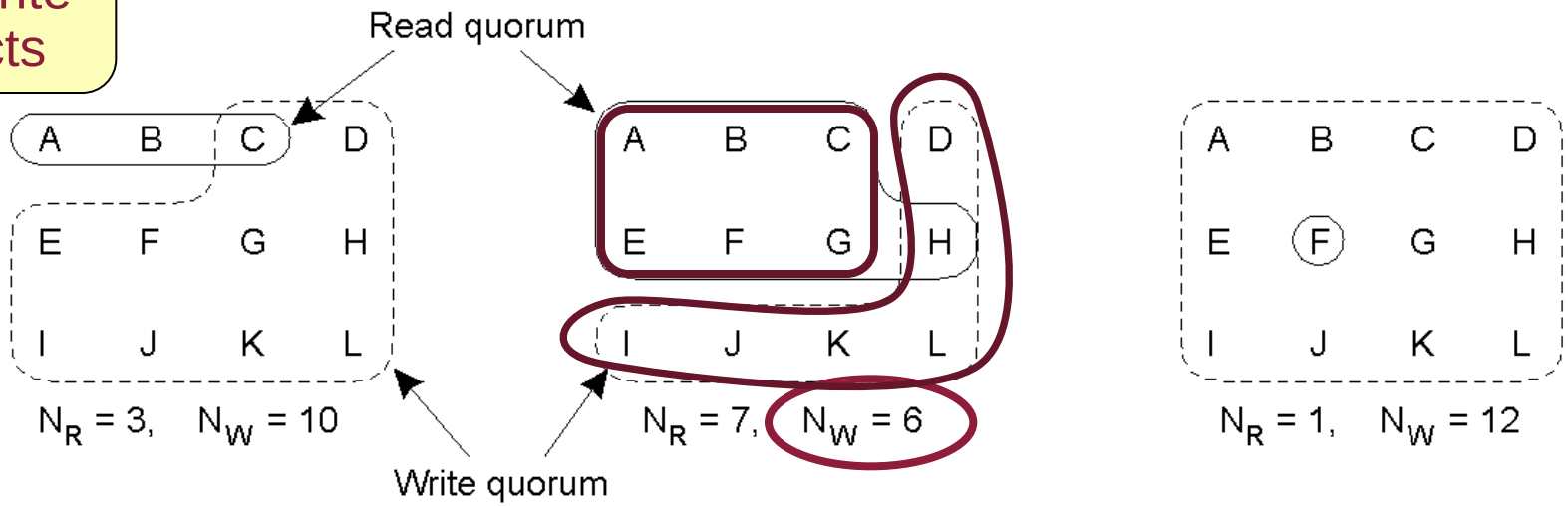


Replicated-Write Protocols (2)

- *Quorum-based*
  - An update occurs only if a quorum of the servers agrees on the version number to be assigned
  - Reading requires a quorum to ensure latest version is being read
  - Typically:
    - $N_R + N_W > N$
    - $N_W > N/2$

Avoids read-write conflicts

Avoids write-write conflicts



A correct choice of read and write set

A choice that may lead to write-write conflicts

A correct choice, known as ROWA (read one, write all)

Politecnico di Milano

Example: Bayou

- Provides data replication for high availability
  - Enabling application-specific conflict detection and resolution
  - Eventually sequentially consistent
- Transactional system:
  - The API provides **read** and **write** operations: conflicts are generated by **write** operations on the same data set
- Updates are marked as *tentative*, and become *committed* when data can produce a consistent state
- A data collection is fully replicated on a set of servers: access to any one of the servers is sufficient for a client to access data
  - Bayou does not require a client to access a specific server: updates are communicated in a peer-to-peer fashion
  - Amenable to MANETs (Mobile Ad-hoc Networks)

Politecnico di Milano

How Bayou Works

- After a client submits a **write** operation to a server, it has no further responsibility on it
  - In particular, it does not wait for its propagation to other servers
- Whenever an update operation is sent, a dependency check and a merge procedure, written by the programmer, are sent along as well
  - The dependency check contains a query and the expected result, and enables conflict detection
  - The merge procedure is run by the server if a conflict is detected
- Updates are eventually propagated using an epidemic algorithm: as long as the set of servers is not permanently partitioned, the **write** will eventually reach all servers, and thus reach consistency
  - Because writes are performed in the same order at all sites, and conflict detection and merge procedures are deterministic
- Servers propagate **writes** among themselves during pairwise interactions (called anti-entropy), during which they bounce **write** operations until they agree on the set of writes they have seen and what is the order to perform them

Politecnico
di Milano

Disadvantages

- Increased complexity for the programmer
  - Dependency and merge routines can be quite complex
- Increased complexity for the user
  - Reading *tentative* data
  - Find their data changed by the system
- Replicas are always accessible to applications, no matter whether they potentially conflict with others or not
  - Cascading conflict resolution is a potential drawback