Distributed Operating Systems
Consistency and Replication

dr inż. Adam Kozakiewicz

akozakie@elka.pw.edu.pl

Institute of Control and Information Engineering
Warsaw University of Technology

Consistency and Replication

Consistency and Replication

1. Introduction
2. Data-centric consistency models
3. Client-centric consistency models
4. Propagation-centric consistency models
5. Consistency protocols

Introduction

Primary reasons for replicating data:

- √ **reliability**
- √ **performance/scalability**

Reliability corresponds to fault tolerance, while performance corresponds to high availability.

There's no such thing as a free lunch:

- √ every modification must be introduced on **all** copies to ensure **consistency**,
- √ the real price of replication depends on *when* and *how* modifications need to be carried out.

Performance and scalability

Main issue: to keep all replicas consistent, all conflicting operations must generally be performed in the same order on all replicas.

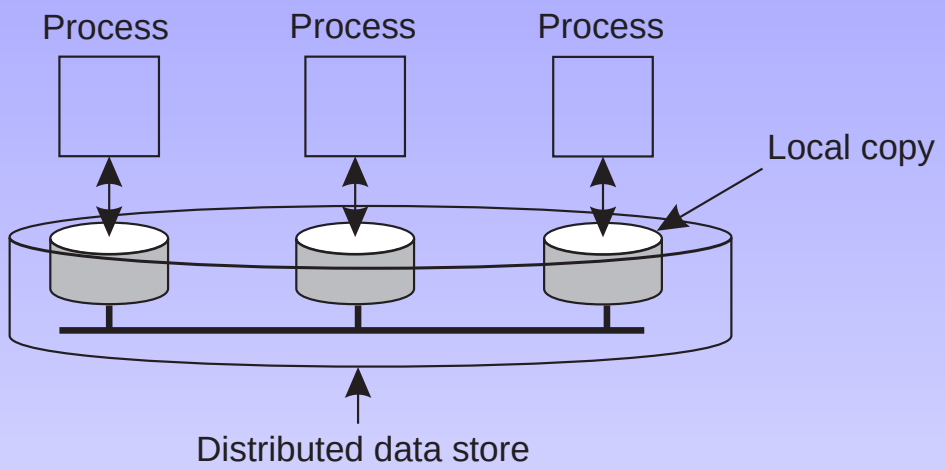
Possible conflicts (see previous lecture):

- √ read-write conflict
- √ write-write conflict

Global ordering of conflicting operations is difficult and costly to achieve, therefore limiting scalability.

Solution: weaken the consistency requirements so that global synchronization can be avoided, while the system is still usable.

Data-Centric Consistency Models (1)



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

Consistency model

A contract between a distributed data store and processes, in which the data store specifies what the results of read and write operations are in the presence of concurrency.

Data-Centric Consistency Models (2)

Strong consistency models: Operations on shared data are synchronized.

- √ strict consistency (related to time)
- √ sequential consistency (what we usually need and want)
- √ causal consistency (only causal relations are preserved)
- √ FIFO consistency (per process)

Weak consistency models: Synchronization by locking/unlocking data.

- √ general weak consistency
- √ release consistency
- √ entry consistency

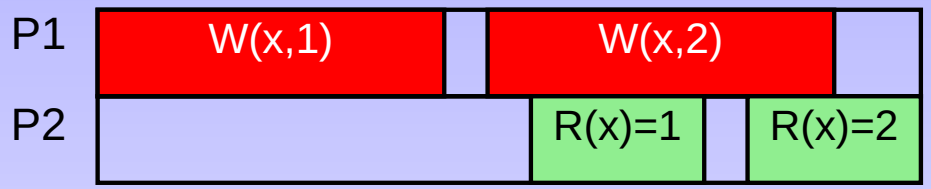
Observation: Weaker models are easier to build in a scalable manner.

Strict Consistency

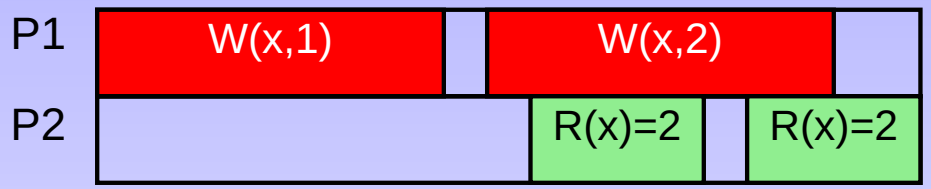
Strict consistency

> Any read to a shared data item X will return the value stored by the most recent write to X.

Impossible to obtain in distributed systems – requires a global real time clock.



(a)



(b)

Two processes operating on a shared data item, the data store is:

a. *not* strictly consistent
b. strictly consistent.

Linearizability and Sequential Consistency (1)

Serializability vs. linearizability

$ops(\sigma)$ – a sequence of call and response events appearing in the execution σ in real-time order.
τ – a legal sequence of operations in the system. If there exists τ with the following properties:

1. τ is a permutation of $ops(\sigma)$,
2. for each process p, the order of operations from that process is the same in $ops(\sigma)$ and $\tau|p$,
3. for any operations op_1 and op_2, if the response for op_1 precedes the call for op_2 in $ops(\sigma)$, then op_1 precedes op_2 in τ,

then the execution σ is **linearizable**. If condition 3 is not satisfied, but 1 and 2 are, then the execution is **sequentially consistent**.
Note: strict rules for breaking ties exist, so $ops(\sigma)$ is unique.

Condition 3 states that if two operations (from call to response) do not overlap, then they are not really concurrent and their order must be preserved.

Linearizability and Sequential Consistency (2)

Instead of dealing directly with calls and responses, assume **atomic** operations, executed in a single instant.

Alt. definition: under linearizability the global real-time order of the atomic operations must be preserved in the system.

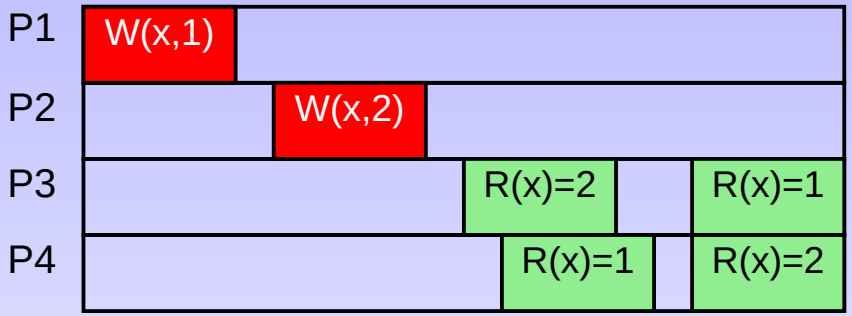
- √ The instant of the operation – any chosen moment between the call and the response,
- √ reordering doesn't really happen – we just define the instants of atomic operations.
- √ if two call-response pairs overlap, then, depending on the choice of instants of atomic operations, the operations can be executed in any order,
- √ if they don't overlap, then however we choose, one will precede the other.

Linear (or atomic) consistency = sequential consistency + global time order.

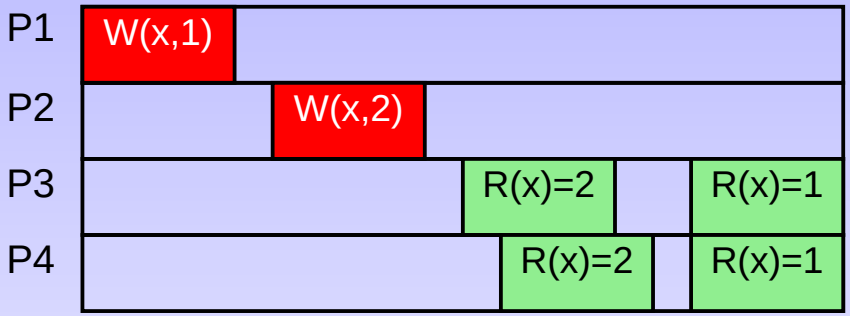
Linearizabililty and Sequential Consistency (3)

Sequential consistency

The result of any group of operations is the same as if all operations were executed in some sequential order, with the condition that operations from a single process are executed in the order specified by the program of that process.



(a)



(b)

All processes must see the same sequential order of operations:

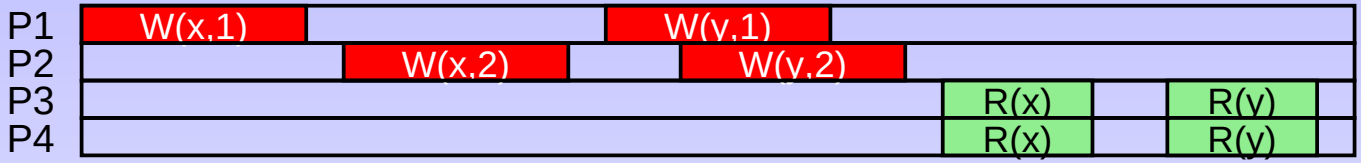
a. this data store is not sequentially consistent,

b. a sequentially consistent data store.

Linearizabililty and Sequential Consistency (4)

Linear consistency

The result of any group of operations is the same as if all operations were executed in some sequential order, with the condition that operations from a single process are executed in the order specified by the program of that process **and that global time ordering of operations is preserved**.



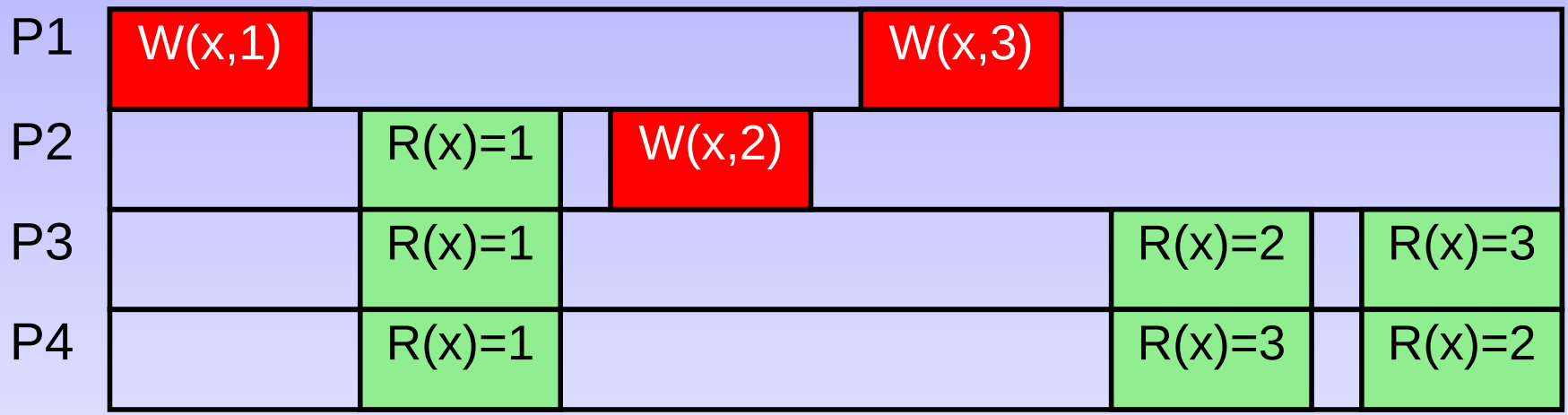
All processes must see the same sequential order of operations and global time ordering must be preserved.

P3 reads	P4 reads	Strict	Linear	Sequential
2,1	2,2	*illegal*	*illegal*	*illegal*
2,2	2,2	legal	legal	legal
2,1	2,1	*illegal*	legal	legal
1,1	1,1	*illegal*	*illegal*	legal

Causal Consistency (1)

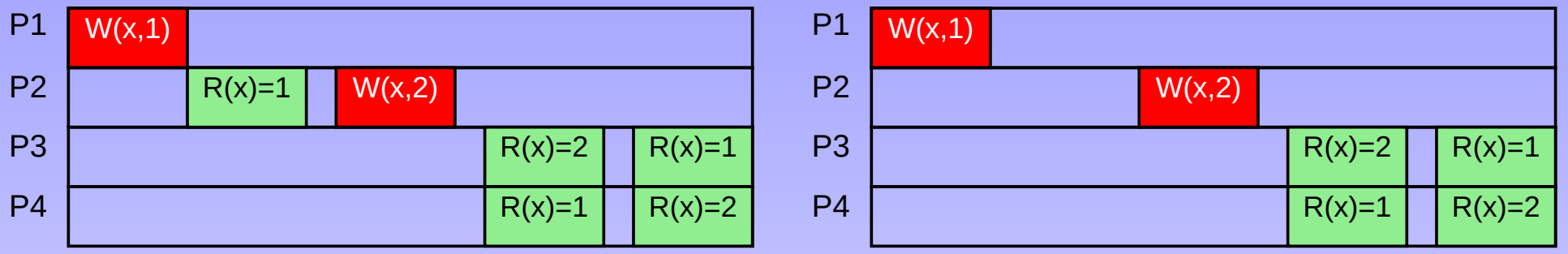
Causal consistency

All write operations that are potentially causally related will be seen by all processes in the same order. Order of concurrent write operations may be seen differently by different machines.



The above example shows a valid scenario for a causally consistent store, but in a strictly or even sequentially consistent store it would not be valid.

Causal consistency (2)



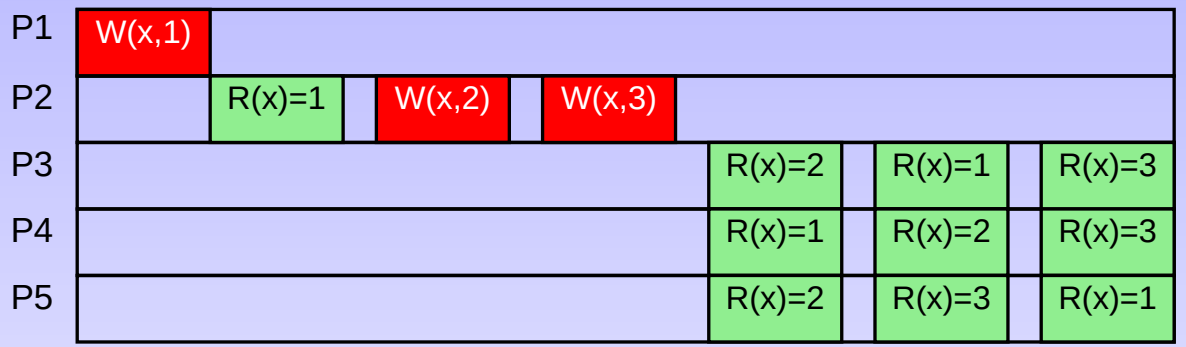
a. a causally consistent store will not allow this to happen – P3 gets a causally impossible result,

b. the causal relation is eliminated, so even though P3 gets the same result, this is a valid scenario for a causally-consistent data store.

FIFO Consistency (1)

FIFO Consistency

All write operations from one process will be seen by all other processes in the order they were issued, but there are no constraints on the order of write operations issued by different processes.



This is a valid scenario in a FIFO consistent data store!

- √ **PRAM** consistency = *pipelined* RAM, pipelining writes from a single process,
- √ easy to implement with a kind of timestamping – process number + operation number.

FIFO Consistency (2)

Process P1	Process P2
x = 1;	y = 1;
if (y == 0) kill(P2);	if (x == 0) kill(P1);

Two concurrent processes.

FIFO vs. sequential consistency:

- √ FIFO consistency: both processes may be killed!
- √ sequential consistency: however the operations are ordered, only one process may be killed.
- √ in sequential consistency the order of operations is non-deterministic, but at least all processes agree what it is. FIFO consistency doesn't even assure that.

Weak Consistency (1)

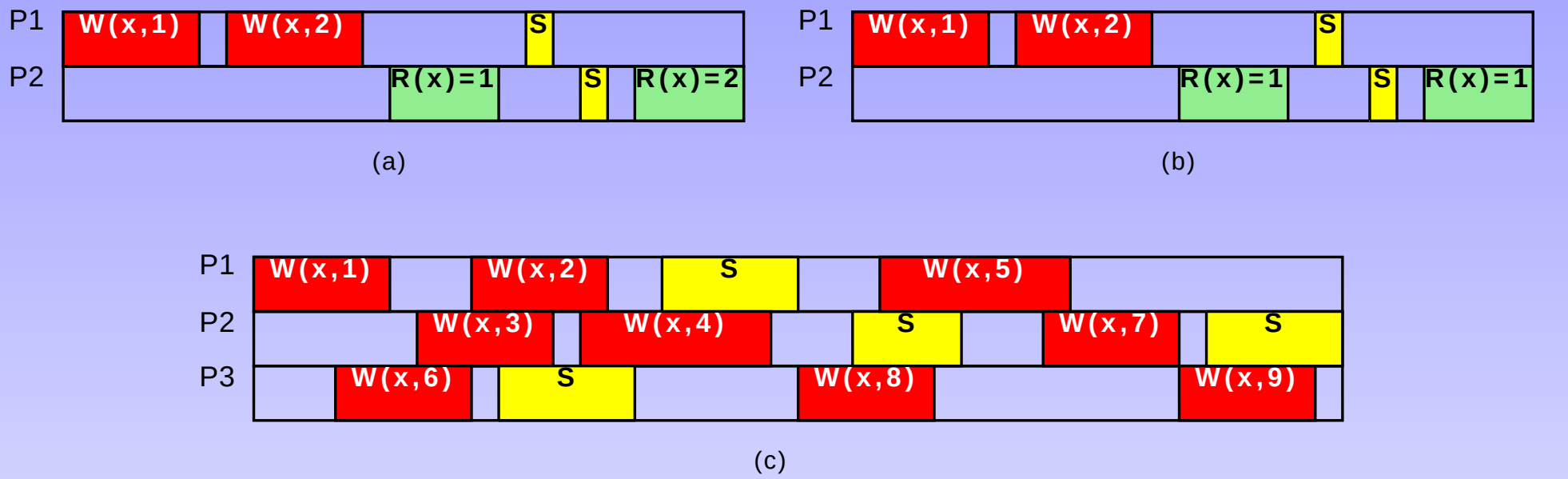
Weak consistency models

The contract for weak consistency models is two-way: consistency is guaranteed only if processes agree to respect some rules. Explicit synchronization variables are introduced. Propagation of changes to replicas happens only during explicit synchronization.

Properties:

- √ accesses to synchronization variables associated with the data store are sequentially consistent,
- √ no operation on a synchronization variable is allowed to be performed until all previous write operations have been completed everywhere,
- √ no operation on data items (read or write) is allowed to be performed until all previous operations to synchronization variables have been performed.

Weak Consistency (2)



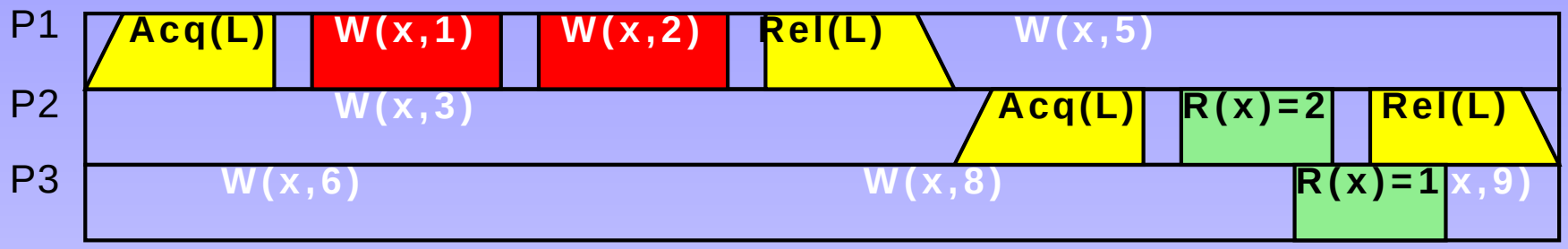
a. a valid sequence of events for weak consistency

b. an invalid sequence of events for weak consistency

c. delayed replies in case of concurrent write and synchronization events.

Observation: Weak consistency is easiest to implement for systems with full replicas.

Release Consistency (1)



A valid sequence of events for release consistency.

Release Consistency (2)

Properties:

- √ all previous aquires done by the process must have completed successfully before a read or write operation is permitted
- √ all previous read and write operations must have completed before a release is allowed
- √ accesses to synchronization variables are FIFO consistent (sequential consistency is not required!)

Notes:

- √ release consistency can be implemented in two ways: as **lazy** or **eager** release consistency
- √ **barriers** may be used instead of critical regions.

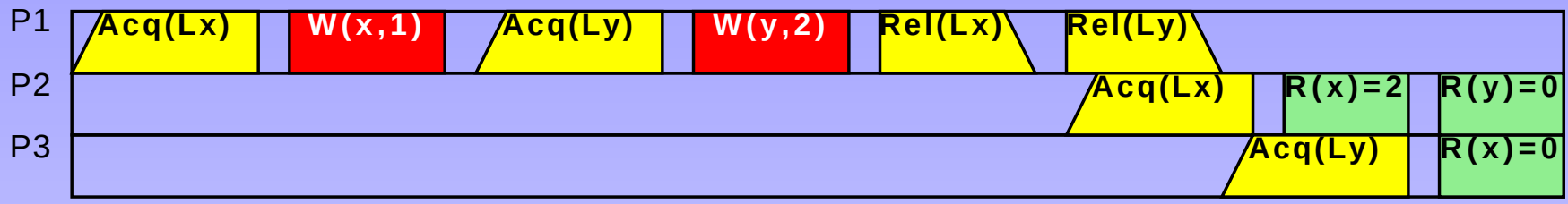
Entry Consistency(1)

Entry consistency:

- √ In release consistency all updates were propagated during release.
- √ In entry consistency model there is a separate synchronization variable for each shared data item.
- √ When acquiring the synchronization variable, the process fetches the most recent values of the data item.

Note: Release consistency is global in the data space, while entry consistency is controlled for each data item separately.
Question: Can entry consistency be made transparent to programmers? How?

Entry Consisitency (2)



A valid event sequence for entry consistency.

Data-Centric Consistency Models – Summary

Strong consistency models

Consistency	Description
Strict	Absolute time ordering of all shared accesses.
Linearizability	All processes see all shared accesses in the same order, consistent with (nonunique) global timestamps.
Sequential	All processes see all shared accesses in the same order, not related to real time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see all writes from the same process in the same order. Accesses from different processes are not ordered.

Weak consistency models

Weak	Shared data can be counted on to be consistent only after explicit synchronization.
Release	Shared data are made consistent at the exit from a critical section.
Entry	Shared data pertaining to a critical section are made consistent when the critical section is entered.

Client-Centric Consistency Models (1)

1. System model
2. Coherence models
   - √ monotonic reads,
   - √ monotonic writes,
   - √ read-your-writes,
   - √ write-follows-reads.

Client-Centric Consistency Models (2)

Goal: Avoid system-wide consistency, concentrate on the needs of clients, not the rules about data state on servers.
Background: Most of the large-scale distributed systems use replication for scalability, but very weak consistency is usually offered:

DNS updates propagate slowly, inserts take time to become visible.

NEWS pushing and pulling of articles and reactions over different routes around the world leads to replies being often seen before the original posting is available.

Lotus Notes geographically dispersed servers do replicate documents, but they don't even attempt to keep the replicas cosistent.

WWW caches all over the place, user often sees an old version of the page.

Consistency for Mobile Users

Example: A distributed database is accessed from a notebook (the front-end).

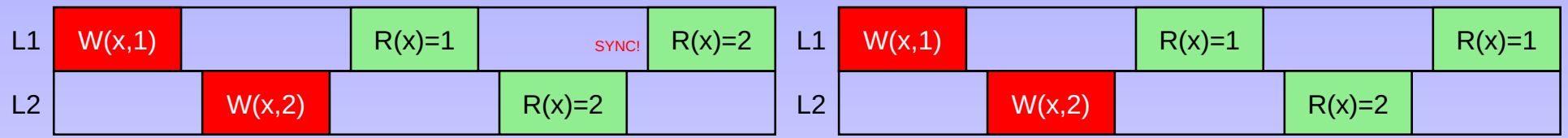
- √ at location A we read and update the database.
- √ at location B we try to continue our work, but if the server we access is different than at A, inconsistencies may appear:
  - ★ updates from A may not yet be available at B,
  - ★ newer entries than at A might be available,
  - ★ updates from B (even our own!) may conflict with the ones from A!

The simple truth: All we want is to continue our work. If the reads and writes done at A are still ok at B, the database seems consistent enough.

Monotonic Reads (1)

Monotonic reads

After the process reads the value of data item x, no successive read operations may return an earlier value of x.



Reads by process P at two different locations (different replicas):

a. a monotonic-read consistent data store,

b. a data store that does not provide monotonic reads.

Monotonic Reads (2)

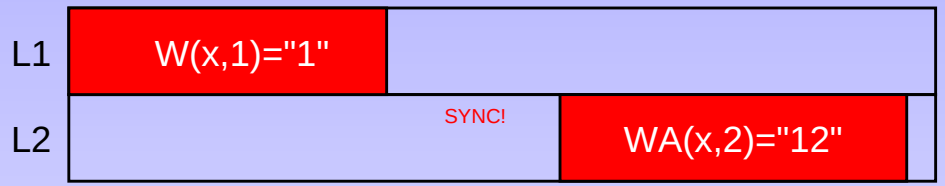
Example

Automatically reading personal calendar updates from different servers should not make any updates nonavailable – no matter which server provides the updates at the moment, you always see all updates you received so far.

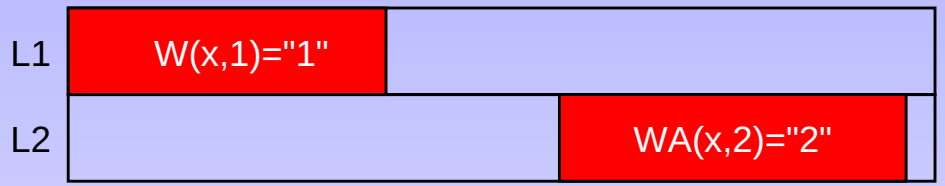
Monotonic Writes (1)

Monotonic writes

Any write operation on data item x by a process must be completed before another write operation on x from the same process will be allowed.



(a)



(b)

Write operations by one process on two different replicas of the same data store:

a. monotonic writes,

b. non-monotonic writes.

Monotonic Writes (2)

Example

Updating a program on server $S2$, may ensure that all dependencies are also placed on server $S2$.

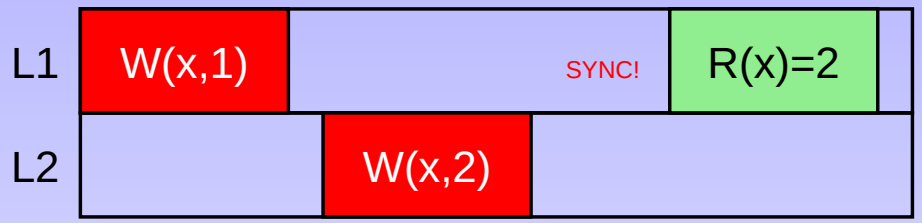
Example

Maintaining versions of replicated files in the correct order – if a new version is written, the latest previous one must first be downloaded to apply changes.

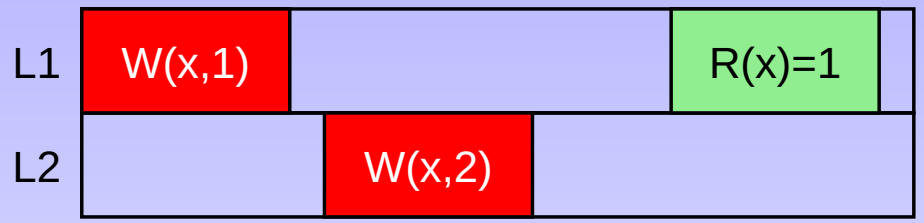
Read Your Writes

Read Your Writes

Results of a successful write operation on data item x must always be seen by successive read operations by the same process.



(a)



(b)

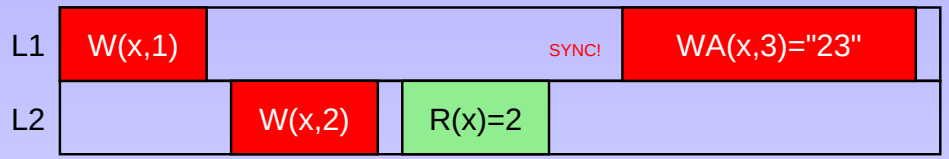
a. a data store providing read-your-writes consistency,

b. a data store providing…well, nothing of interest.

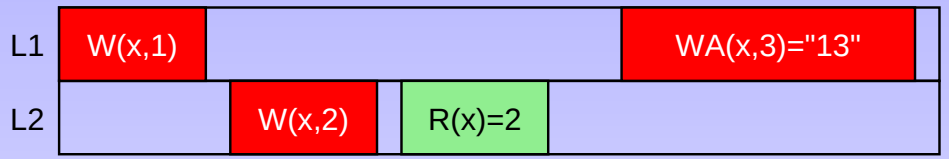
Writes Follow Reads

Writes follow reads

A write operation by a process on data item x following a read operation by the same process on the same data item must modify the same or more recent value of x.



(a)



(b)

a. a writes-follow-reads consistent data store.

b. a not-nearly-as-good-as-that data store.

Examples

Read your writes – example

Updating your own web page in a way that guarantees that your browser will see the newest version, not a copy from cache (clearing cache on update? but what about proxies?).

Writes follow reads – example

If you see a reply to a usenet posting, you should also see the posting itself.

Propagation-Centric Consistency Models

Used in very large systems or under extreme time/bandwidth requirements. Applicable only if limited lack of consistency is acceptable.

Can also be interpreted as data-centric consistency models for systems based on client-centric models.

- √ Eventual consistency
- √ Delta consistency
- √ Vector-field consistency

Eventual Consistency

Eventual consistency

Consistency model in large-scale distributed replicated databases that tolerate a relatively high degree of inconsistency.
If no updates take place for a long time, the replicas will reach consistency.
It'll get there eventually...

- √ easy to implement – periodic broadcasting or polling suffices, similarily lazy hierarchical approaches are possible,
- √ extremely resilient – since the system tolerates inconsistencies anyway, a replica may rebuild its state after a crash even without additional algorithms,
- √ normally systems of this kind are never really consistent – it is a permanent transitional state, where newest updates are only available locally.

Delta Consistency

Delta consistency

A stricter version of eventual consistency. Any write operation must become available to all readers in time shorter than δ. Depending on application, δ can vary from milliseconds to even days.

- √ still relatively easy to implement – periodic broadcasting or polling suffices for large δ, but more advanced techniques may be necessary in high traffic systems with small δ (e.g. hierarchical approches),
- √ rebuilding a replica is no longer trivial – algorithms may still be simple, but the replica cannot become operational until it is updated,
- √ systems of this kind are also rarely consistent, but the inconsistency is limited.

Vector-Field Consistency

Vector-field consistency

A modification of delta consistency model. Update delay is still limited, but the data item dependent limit (δ) is selected from a vector of values. The selection is based on a dynamically changing proximity relation between the data items and the client.

- √ primarily created for multiplayer games,
- √ all properties of delta consistency,
- √ more important information is guaranteed to be more up to date,
- √ may include infinite delay for sufficiently far objects, avoiding propagation of updates if clients don't need them.

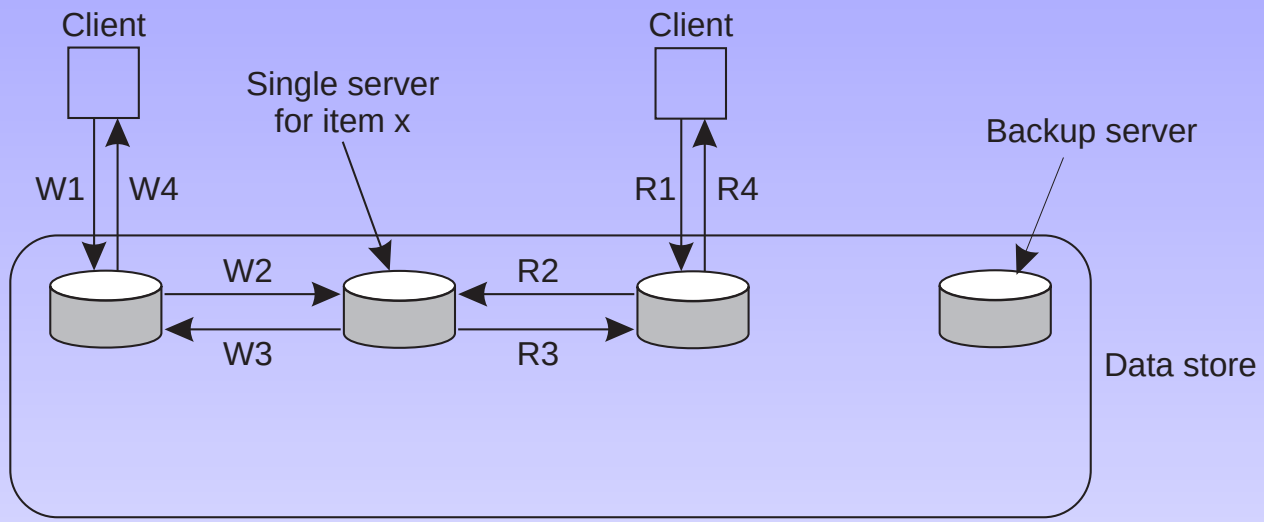
Consistency Protocols

Consistency protocol

Describes the implementation of a given consistency model. We will focus on sequential consistency.

- √ Primary-based protocols
  - ⋆ remote-write protocols,
  - ⋆ local-write protocols.
- √ Replicated write protocols,
  - ⋆ active replication,
  - ⋆ quorum-based protocols.
- √ Cache coherence protocols (write-through, write-back)

Remote-Write Protocols (1)

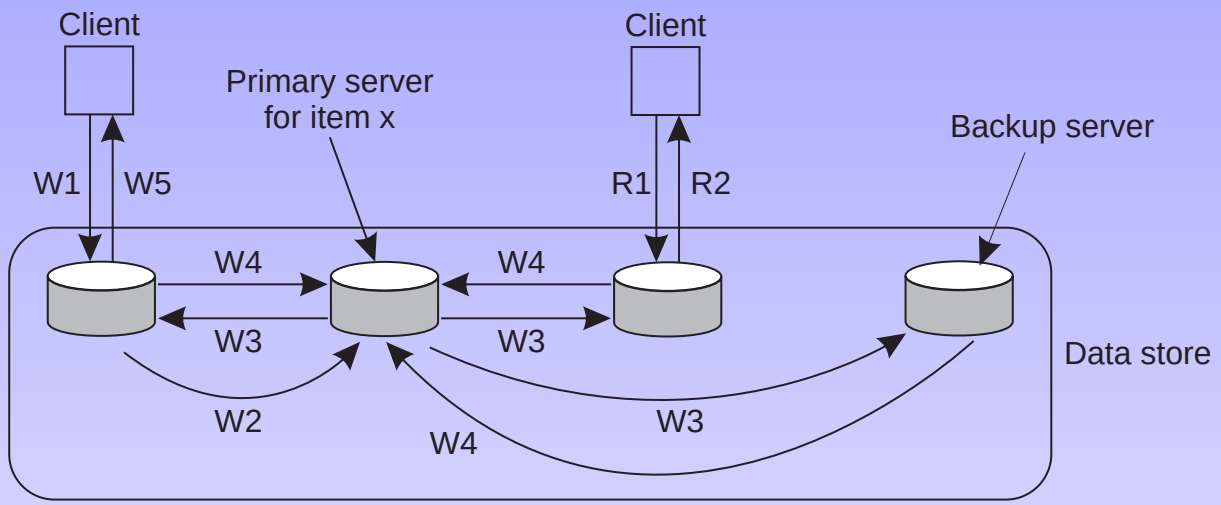


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

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

Primary-based remote-write protocol with a fixed server to which all operations are forwarded.

Remote-Write Protocols (2)

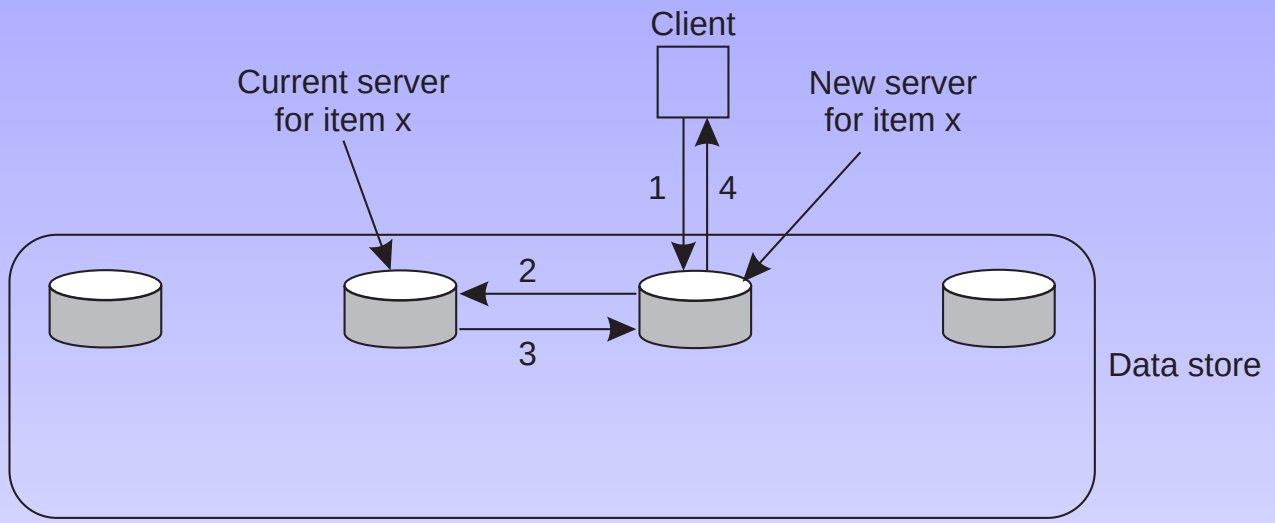


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

A (ang. *primary-backup*) protocol: read operations are allowed on the local copy, write operations are forwarded to the fixed primary copy.

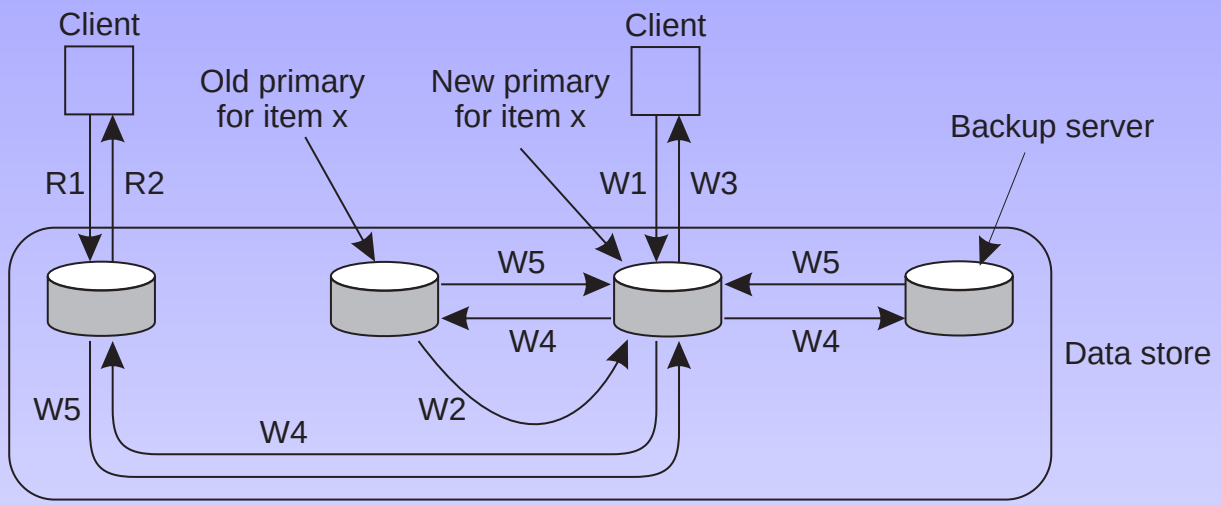
Local-Write Protocols (1)



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 protocol with a single copy being migrated between processes.

Local-Write Protocols (2)

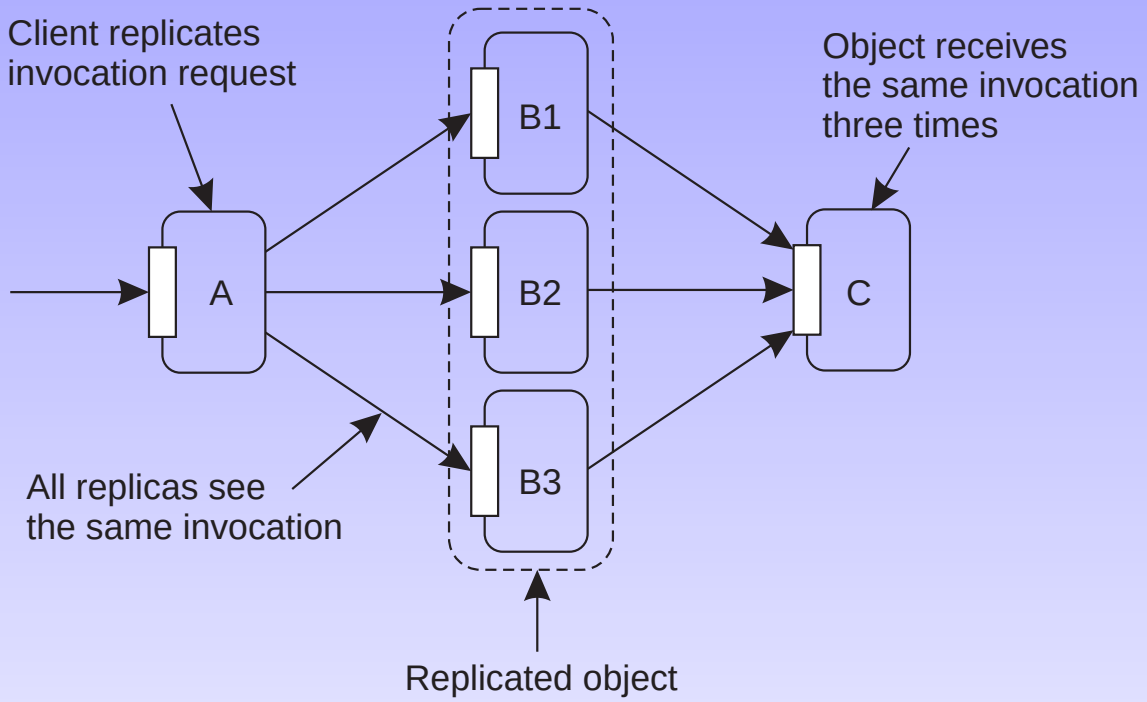


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

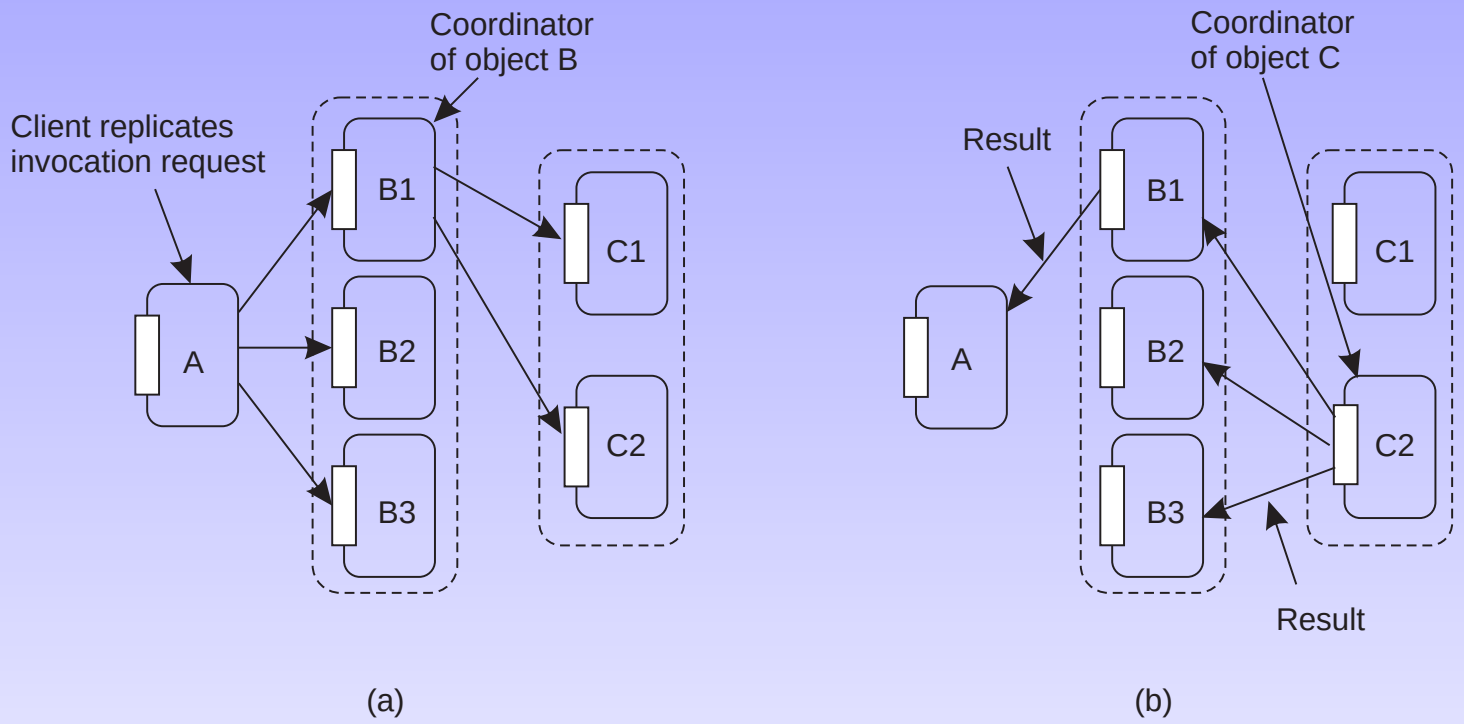
Primary-backup protocol with the primary being migrated to the process performing an update.

Active Replication (1)



The problem with replicated objects.

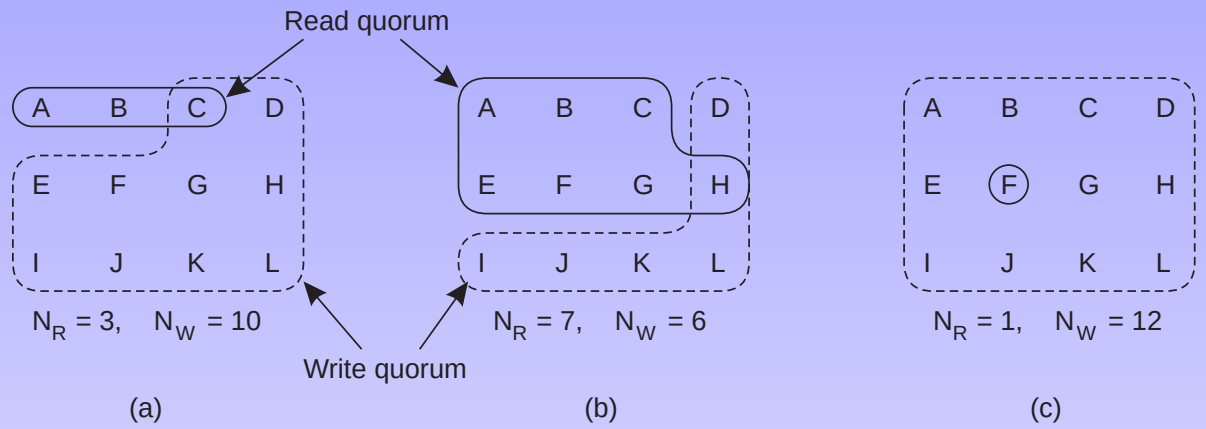
Active Replication (2)



a. invocation request forwarding for replicated objects,

b. reply forwarding for replicated objects.

Quorum-Based Protocols



Three examples of a voting algorithm:

a. a correct choice of the read and write sets,

b. an incorrect choice that may allow write-write conflicts,

c. another correct choice, known as ROWA (read one, write all).

Constraints: $N_R + N_W > N$ and $N_W > N/2$

Cache Coherence Protocols

Cache coherence strategies:

- √ coherence detection strategy – *when* are the inconsistencies detected?
- √ coherence enforcement strategy – *how* can the caches be kept consistent with the copies stored at servers?

When processes modify data:

- √ read-only cache – updates possible only by servers,
- √ write-through cache – client may modify cached data, all updates are forwarded to servers,
- √ write-back cache – client may modify cached data, but propagation of updates is delayed and multiple writes are allowed to take place before servers are informed.