Distributed Systems
Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science
Room R4.20, steen@cs.vu.nl

Chapter 07: Consistency & Replication

Version: November 19, 2009

vrije Universiteit *amsterdam*

Contents

Consistency & replication

- Introduction (what's it all about)
- Data-centric consistency
- Client-centric consistency
- Replica management
- Consistency protocols

Performance and scalability

Main issue

To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the the same order everywhere

Conflicting operations

From the world of transactions:

- Read–write conflict: a read operation and a write operation act concurrently
- Write–write conflict: two concurrent write operations

Issue

Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability Solution: weaken consistency requirements so that hopefully global synchronization can be avoided

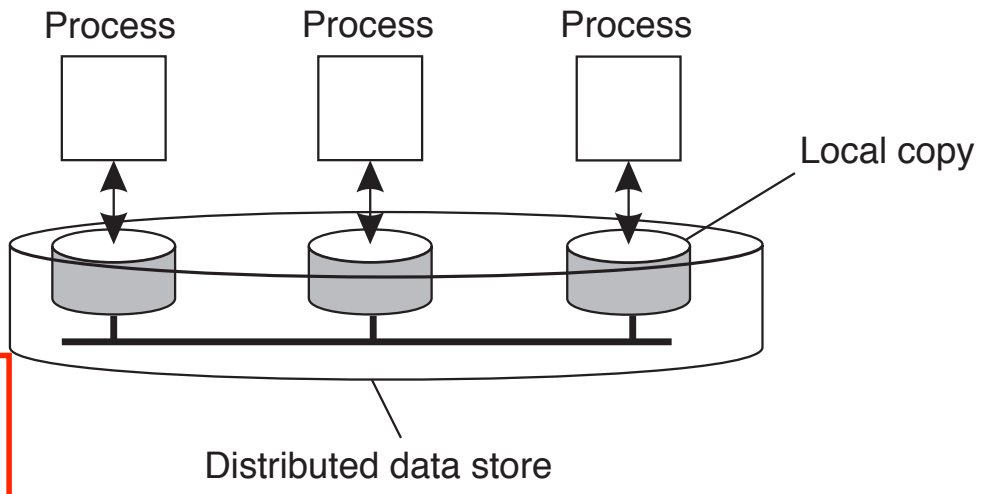
Data-centric consistency models

Consistency model

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

Essential

A data store is a distributed collection of storages:



a process that performs a read operation on a data item, expects the operation to return a value that shows the results of the last write operation on that data.

processes agree on order of operations

storage provides consistent view of data

Continuous Consistency

Observation

We can actually talk a about a degree of consistency:

- replicas may differ in their numerical value
- replicas may differ in their relative staleness
- there may be differences with respect to (number and order) of performed update operations

e.g., stock market, values cannot differ more than $0.2 or within 0.05% of each other

the frequency with which they are updated

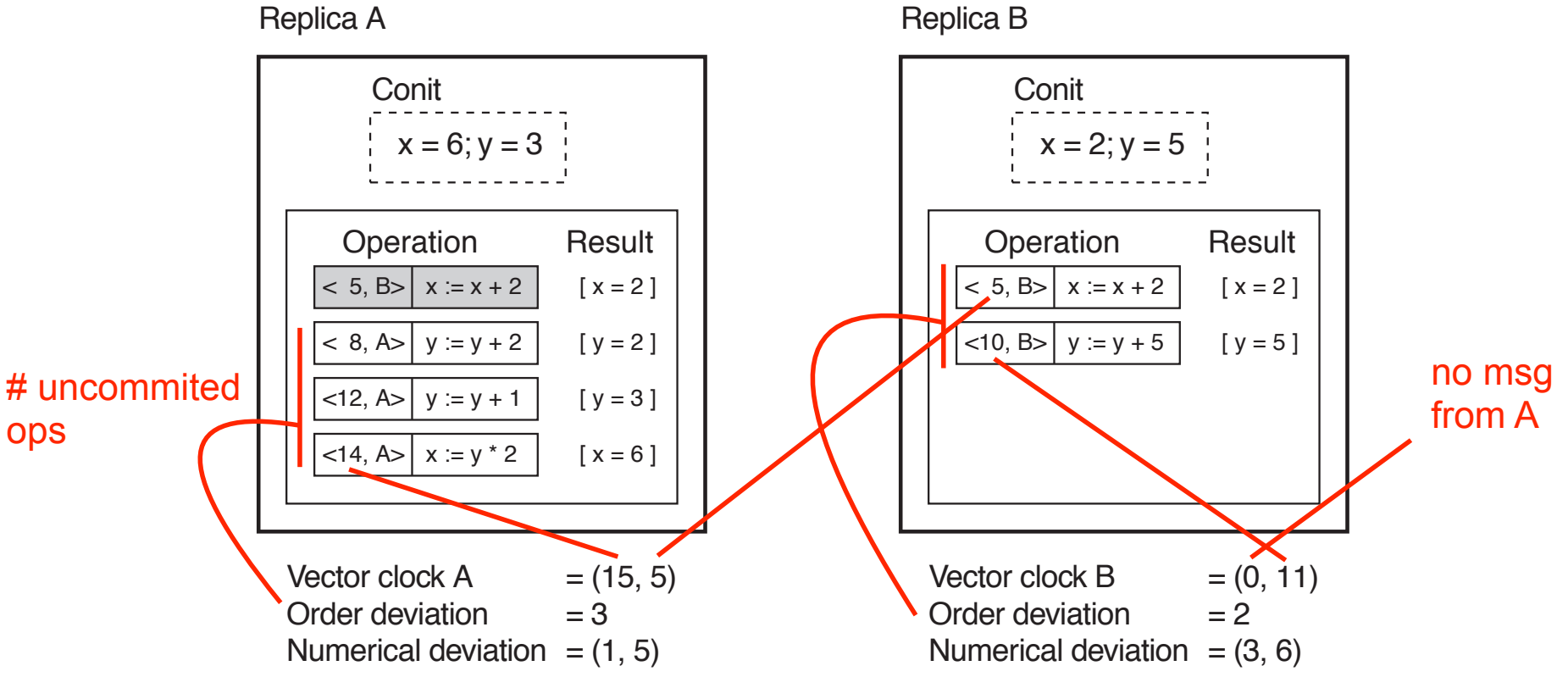
granularity

Conit

Consistency unit $\Rightarrow$ specifies the data unit over which consistency is to be measured.

Example: Conit

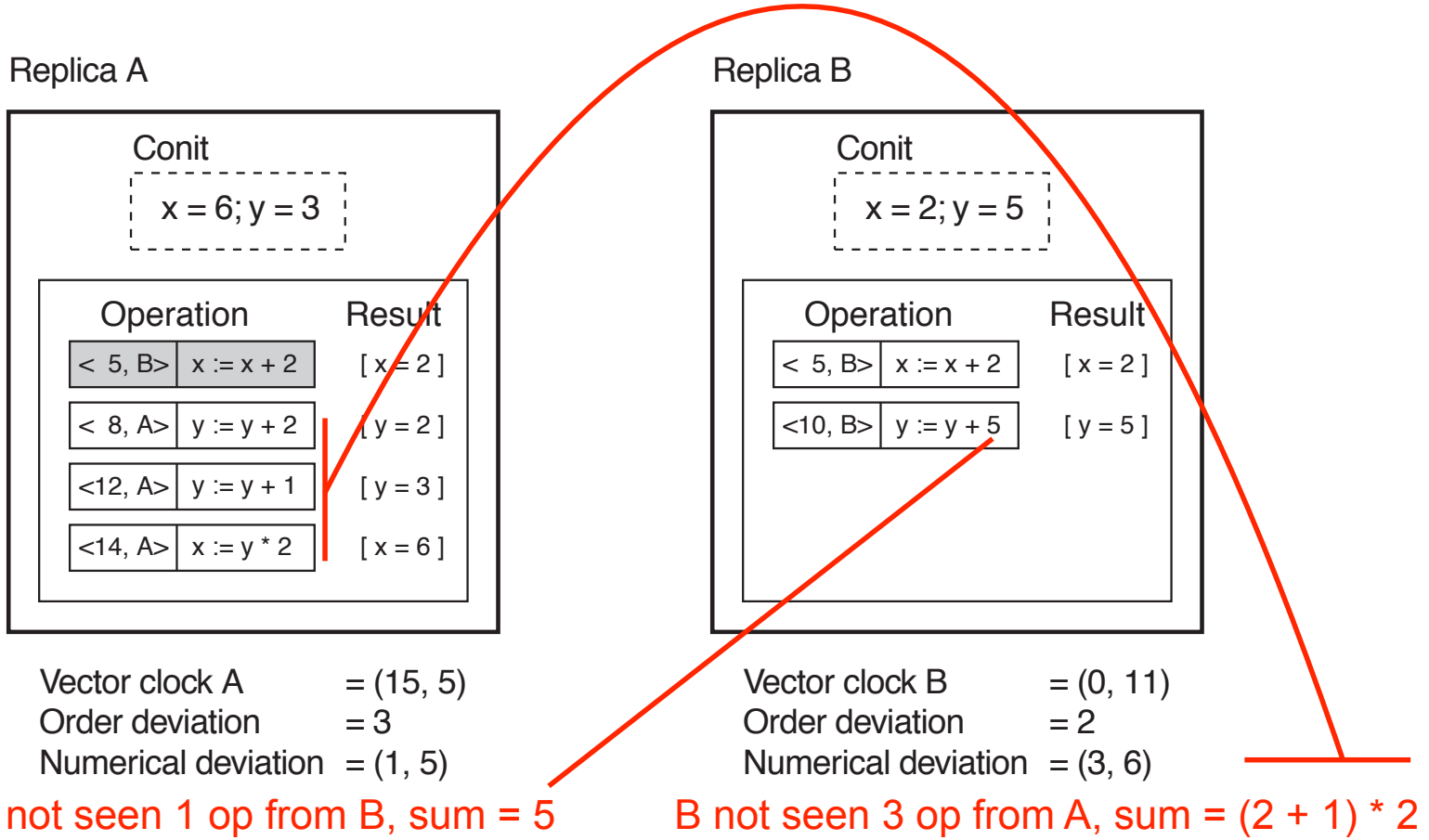
initial values (x,y) = (0, 0)



Conit (contains the variables x and y)

- Each replica maintains a vector clock
- B sends A operation $[\langle 5, B\rangle: x := x + 2]$; A has made this operation permanent (cannot be rolled back)

Example: Conit

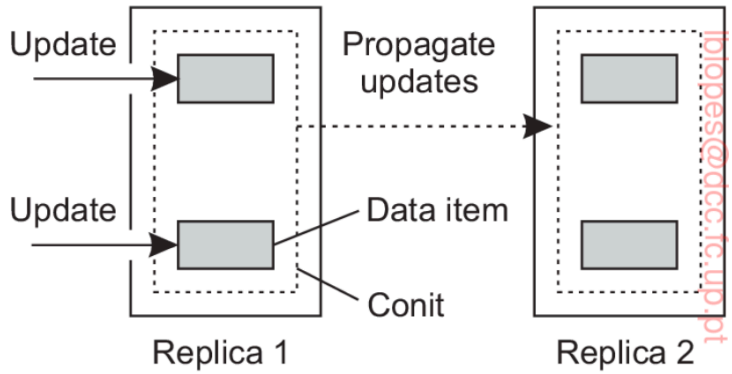


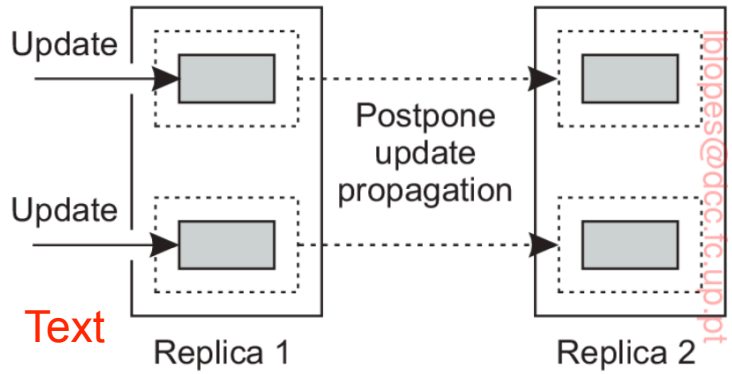
Conit (contains the variables *x* and *y*)

- *A* has three pending operations $\Rightarrow$ order deviation = 3
- *A* has missed one operation from *B*, yielding a max diff of 5 units $\Rightarrow$ (1,5)

e.g., consistency protocol: replicas cannot differ more that one update for each item (in other words: order deviation <= 1)

conit granularity





big conit
has 2 itens

2 updates on same item
=>
propagate update

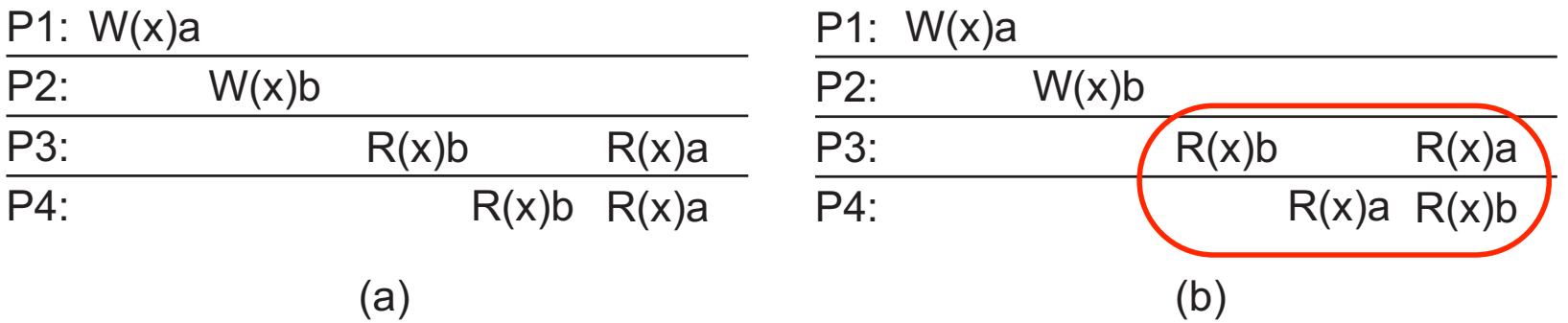
small conits
1 item each

2 updates but on different items
=>
no need to propagate immediately

Sequential consistency

Definition

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

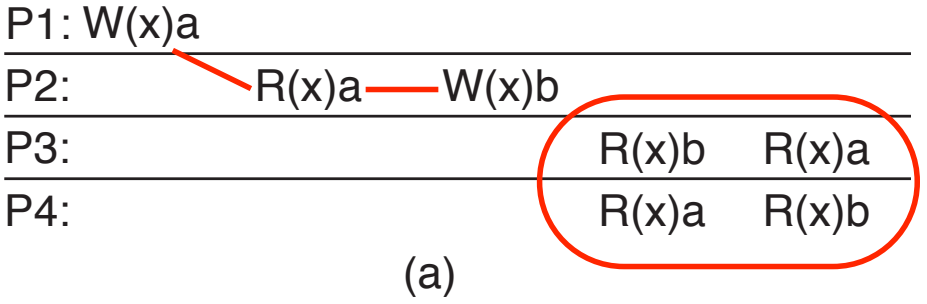


Causal consistency

Definition

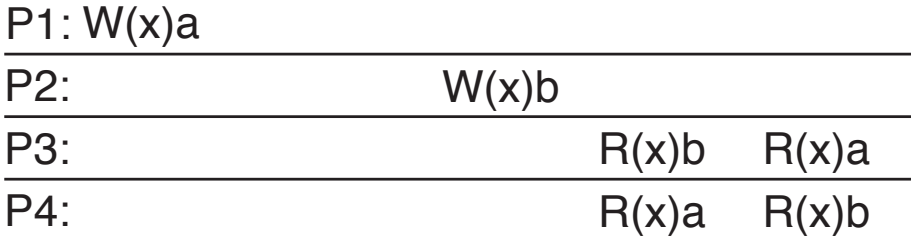
Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order by different processes.

wrong,
Wxa and Wxb should be seen in the same order as they are causally linked

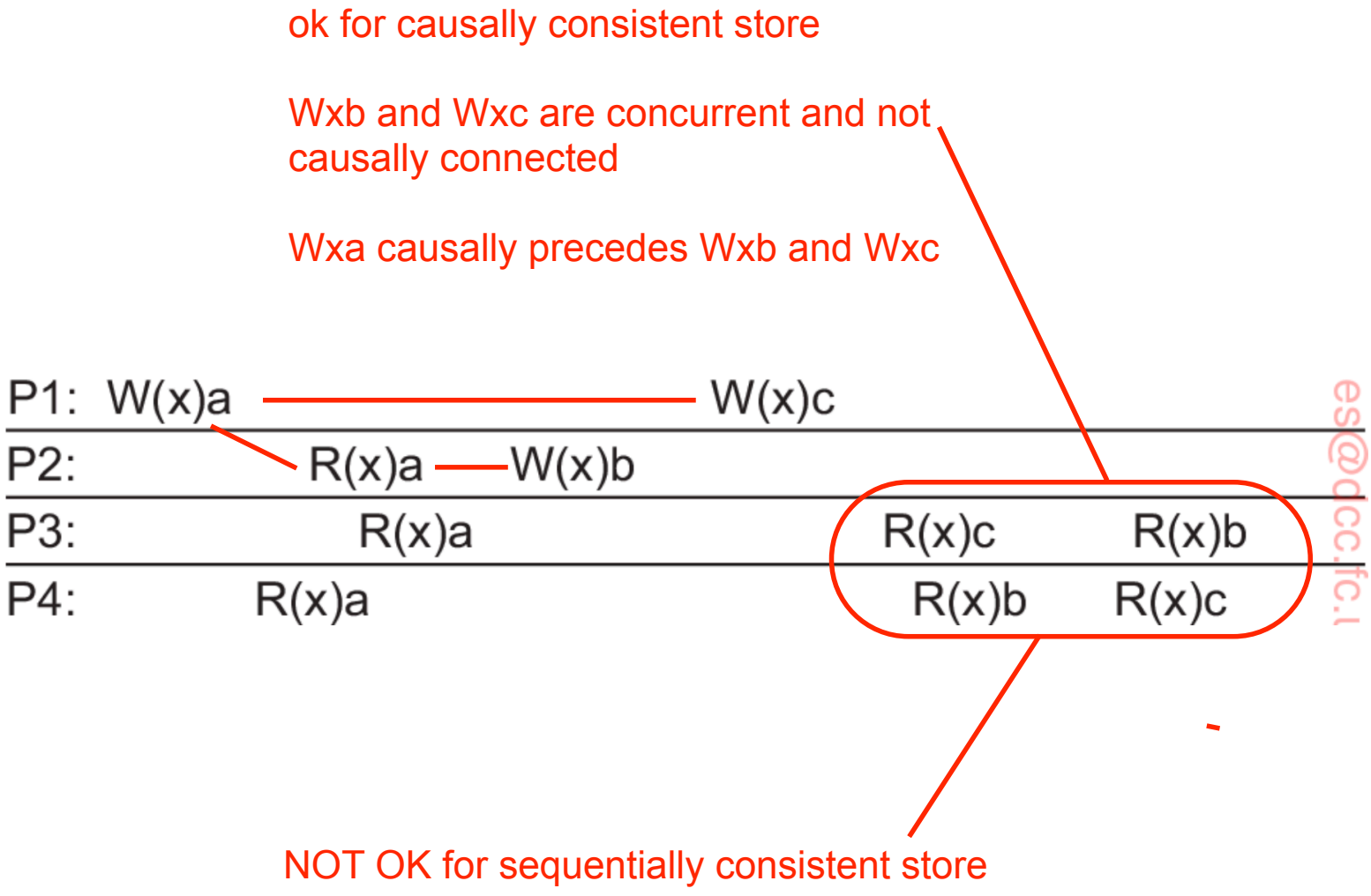


(a)

ok,
Wxa and Wxb are concurrent



(b)


ok for causally consistent store
Wxb and Wxc are concurrent and not causally connected
Wxa causally precedes Wxb and Wxc
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
NOT OK for sequentially consistent store

Process P_1	Process P_2	Process P_3
`x ← 1;`	`y ← 1;`	`z ← 1;`
`print(y,z);`	`print(x,z);`	`print(x,y);`

examples of orderings that satisfy sequential consistency

Execution 1	Execution 2	Execution 3	Execution 4
P_1: x ← 1; P_1: print(y,z); P_2: y ← 1; P_2: print(x,z); P_3: z ← 1; P_3: print(x,y);	P_1: x ← 1; P_2: y ← 1; P_2: print(x,z); P_1: print(y,z); P_3: z ← 1; P_3: print(x,y);	P_2: y ← 1; P_3: z ← 1; P_3: print(x,y); P_2: print(x,z); P_1: x ← 1; P_1: print(y,z);	P_2: y ← 1; P_1: x ← 1; P_3: z ← 1; P_2: print(x,z); P_1: print(y,z); P_3: print(x,y);
Prints: 001011 *Signature:* 00 10 11	*Prints:* 101011 *Signature:* 10 10 11	*Prints:* 010111 *Signature:* 11 01 01	*Prints:* 111111 *Signature:* 11 11 11
(a)	(b)	(c)	(d)

Grouping operations

Definition

- Accesses to synchronization variables are sequentially consistent.
- No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
- No data access is allowed to be performed until all previous accesses to synchronization variables have been performed.

Basic idea

You don't care that reads and writes of a series of operations are immediately known to other processes. You just want the effect of the series itself to be known.

A lock has shared data items associated with it, and each shared data item is associated with at most one lock.

e.g., 1 lock — 1 object, 1 lock — n db tables

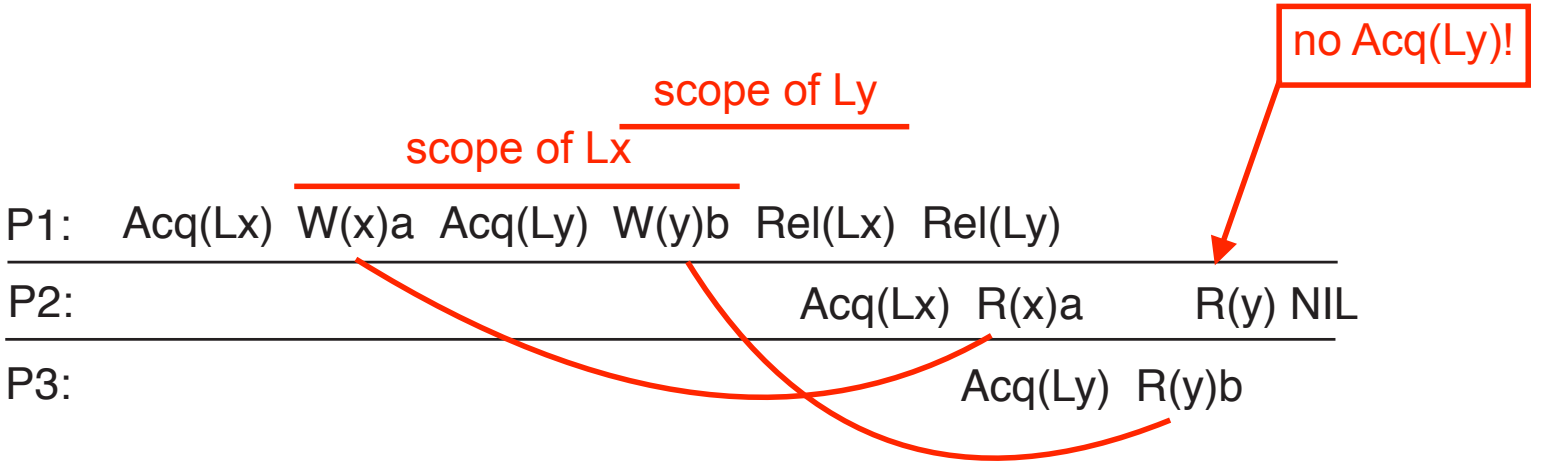
- Acquiring a lock can succeed only when all updates to its associated shared data have completed.
- Exclusive access to a lock can succeed only if no other process has exclusive or nonexclusive access to that lock.
- Nonexclusive access to a lock is allowed only if any previous exclusive access has been completed, including updates on the lock's associated data.

can do RW

only R

at the program level read and write operations are bracketed by the pair of operations Acquire_Lock() and Release_Lock()

Grouping operations



Observation

Weak consistency implies that we need to lock and unlock data (implicitly or not).

Question

What would be a convenient way of making this consistency more or less transparent to programmers?

associate locks and data at middleware level

Client-centric consistency models

Overview

- System model
- Monotonic reads
- Monotonic writes
- Read-your-writes
- Write-follows-reads

Goal

Show how we can perhaps avoid systemwide consistency, by concentrating on what specific clients want, instead of what should be maintained by servers.

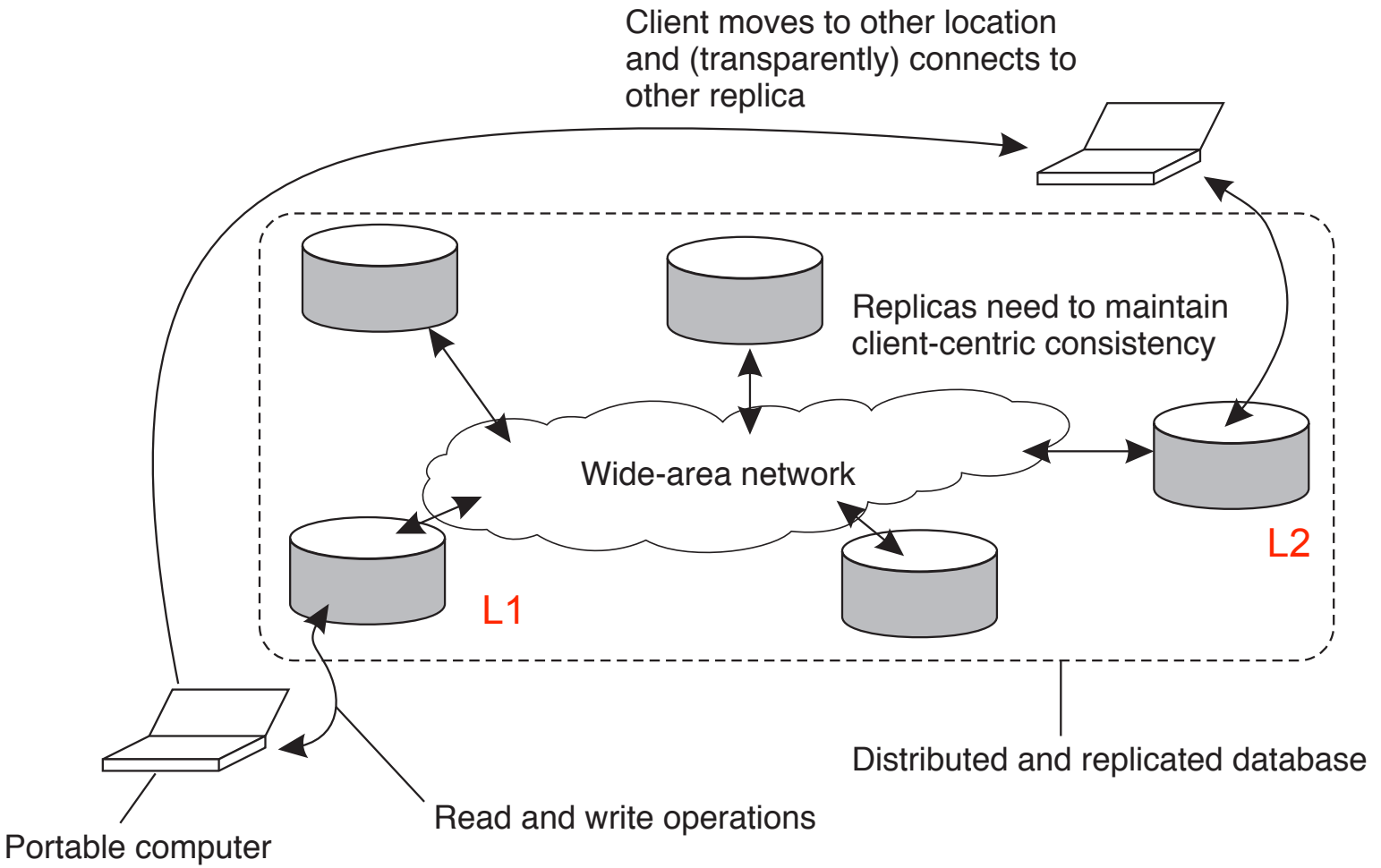
Consistency for mobile users

Example

Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.

- At location *A* you access the database doing reads and updates.
- At location *B* you continue your work, but unless you access the same server as the one at location *A*, you may detect inconsistencies:
  - your updates at *A* may not have yet been propagated to *B*
  - you may be reading newer entries than the ones available at *A*
  - your updates at *B* may eventually conflict with those at *A*

Basic architecture



Consistency for mobile users

Note

The only thing you really want is that the entries you updated and/or read at *A*, are in *B* the way you left them in *A*. In that case, the database will appear to be consistent to you.

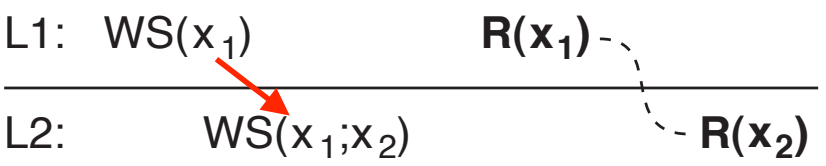
note: henceforth A = L1, B = L2

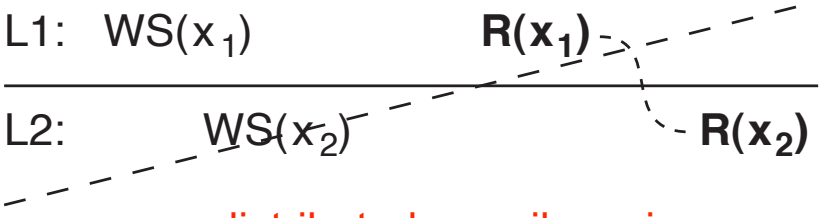
Monotonic reads

Definition

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 or a more recent value.

Read-Read





e.g., distributed e-mail service, x = e-mail messages
if you read an e-mail message x1 at L1 then, later on,
you must be able to read x1 at L2 (plus newer messages x2)
x1 must be sent from L1 to L2

Client-centric consistency: notation

Notation

- $WS(x_i[t])$ is the set of write operations (at L_i) that lead to version x_i of x (at time t)
- $WS(x_i[t_1]; x_j[t_2])$ indicates that it is known that $WS(x_i[t_1])$ is part of $WS(x_j[t_2])$.
- Note: Parameter t is omitted from figures.

Monotonic reads

Example

Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

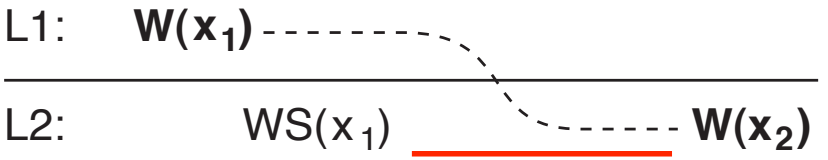
Example

Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

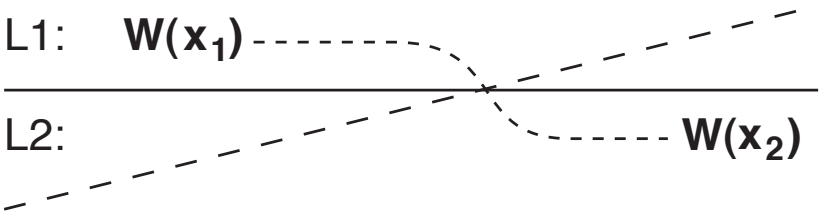
Monotonic writes

Definition

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



Write-Write



update to x from L1 at L2 is performed before update to x at L2
the client sees updates to x made at L1 before updating x at L2

Monotonic writes

Example

Updating a program at server S_2, and ensuring that all components on which compilation and linking depends, are also placed at S_2.

Example

Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

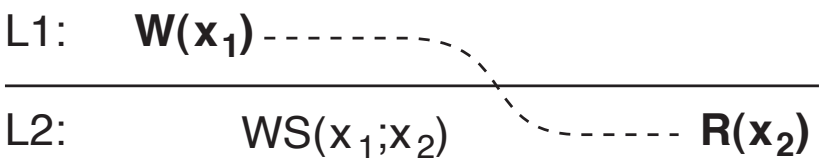
distributed software repository:
update (write) project files at L1
move to L2
update (write) project files at L2

When updating files at L2 the middleware must first fetch and apply the updates (writes) done at L1 - project is kept up-to-date from a user's perspective.

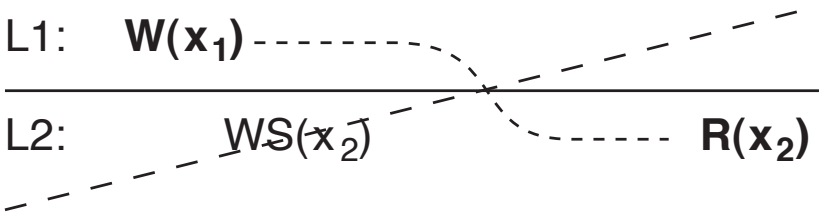
Read your writes

Definition

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.



Write-Read



Example

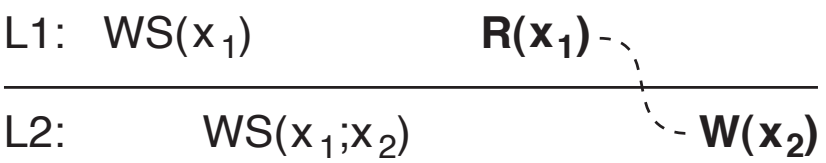
Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.

editor and browser must be supported by the same middleware that implements this consistency model.

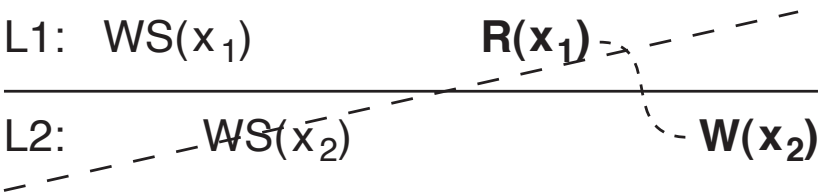
Writes follow reads

Definition

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.



Read-Write



Example

See reactions to posted articles only if you have the original posting (a read "pulls in" the corresponding write operation).

blog articles: user reads article Rx1, later replies Wx2. The write appears in all copies of the blog *after* the original article is seen also, i.e., the WSx1 is propagated.

Distribution protocols

- Replica server placement
- Content replication and placement
- Content distribution

Replica placement

Can only be handled through heuristics

Essence

N-k, assuming 1<=k<=K servers have been chosen already

Figure out what the best K places are out of N possible locations.

- Select best location out of $N - K$ for which the average distance to clients is minimal. Then choose the next best server. (Note: The first chosen location minimizes the average distance to all clients.) Computationally expensive.

O(n^2)

distance = latency / connection bandwidth

- Select the K-th largest autonomous system and place a server at the best-connected host. Computationally expensive.
- Position nodes in a d-dimensional geometric space, where distance reflects latency. Identify the K regions with highest density and place a server in every one. Computationally cheap.

region = nodes accessing same data items with low latency; O(n . max{log(N), K}) -> K= 20, N= 64k nodes -> 50000 times faster than other 2 approaches

a sub-network running same protocol and with the same admin organization / one server for each of K ASes

Content replication

copies of the data store are created, by the owner, closer to clients

e.g. Web site - mirror replicas spread across Internet

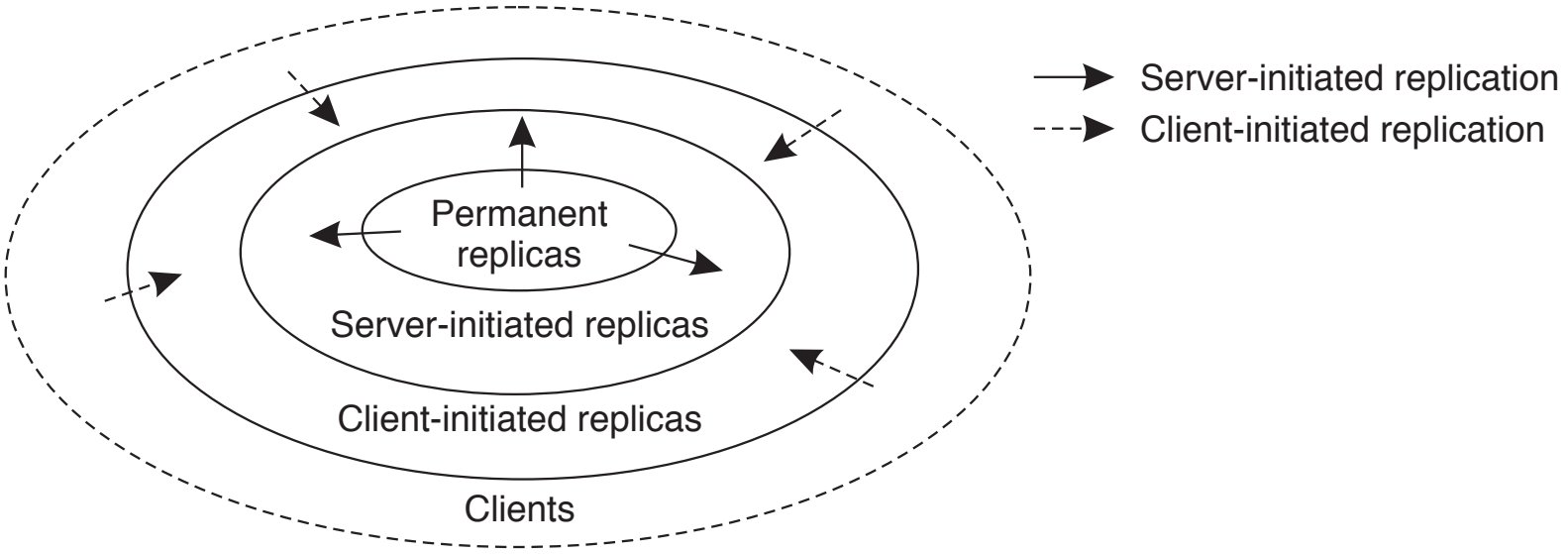
Distinguish different processes

A process is capable of hosting a replica of an object or data:

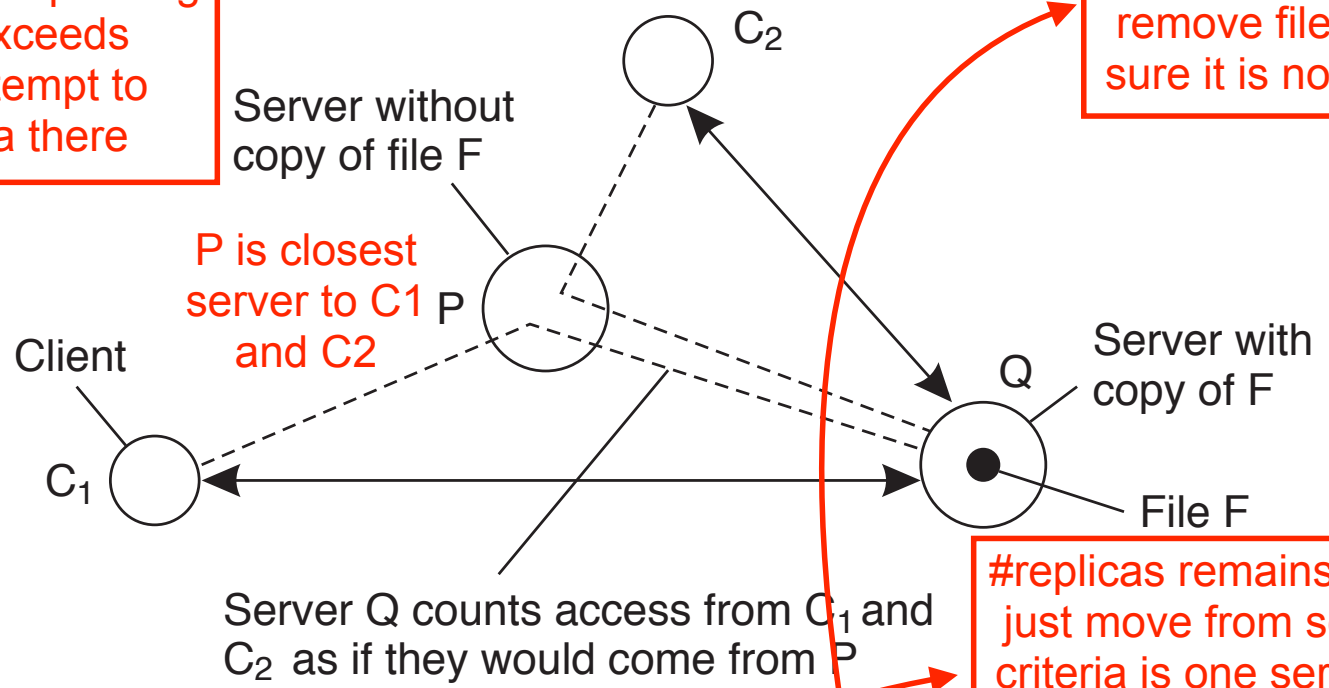
- Permanent replicas: Process/machine always having a replica
- Server-initiated replica: Process that can dynamically host a replica on request of another server in the data store
- Client-initiated replica: Process that can dynamically host a replica on request of a client (client cache)

client requests copy - cache, located in local machine or shared copy among local network

Content replication



Server-initiated replicas



- Keep track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below threshold $D \Rightarrow$ drop file
- Number of accesses exceeds threshold $R \Rightarrow$ replicate file
- Number of access between D and $R \Rightarrow$ migrate file

Content distribution

you now have several replicas of the data - how do you propagate updates among the replicas ?

Model

Consider only a client-server combination:

- Propagate only notification/invalidation of update (often used for caches)
- Transfer data from one copy to another (distributed databases)
- Propagate the update *operation* to other copies (also called active replication)

invalidated data must be fetched again, effective if R/W ratio is low

effective if data will be read several times before next update (R/W ratio is high)

Note

e.g., RPC/RMI, FT algorithms; low overhead but need extra computing resources in servers holding replicas

No single approach is the best, but depends highly on available bandwidth and read-to-write ratio at replicas.

Content distribution

mostly used when replicas need a high degree of consistency, greedy

- Pushing updates: server-initiated approach, in which update is propagated regardless whether target asked for it.
- Pulling updates: client-initiated approach, in which client requests to be updated.

used for more relaxed consistency

Issue	Push-based	Pull-based
1:	List of client caches	None
2:	Update (and possibly fetch update)	Poll and update
3:	Immediate (or fetch-update time)	Fetch-update time

1: State at server

2: Messages to be exchanged

3: Response time at the client

e.g.
notification/invalidation -> push-based,
active replication -> push-based,
client centric consistency -> pull based

Content distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Content distribution

Issue

Make lease expiration time dependent on system's behavior (adaptive leases):

- Age-based leases: An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- Renewal-frequency based leases: The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- State-based leases: The more loaded a server is, the shorter the expiration times become

Unicast vs. Multicast: multicast usually used for push-based (e.g., active replication), unicast for pull-based (e.g., single client asks for copy of data)

Consistency protocols

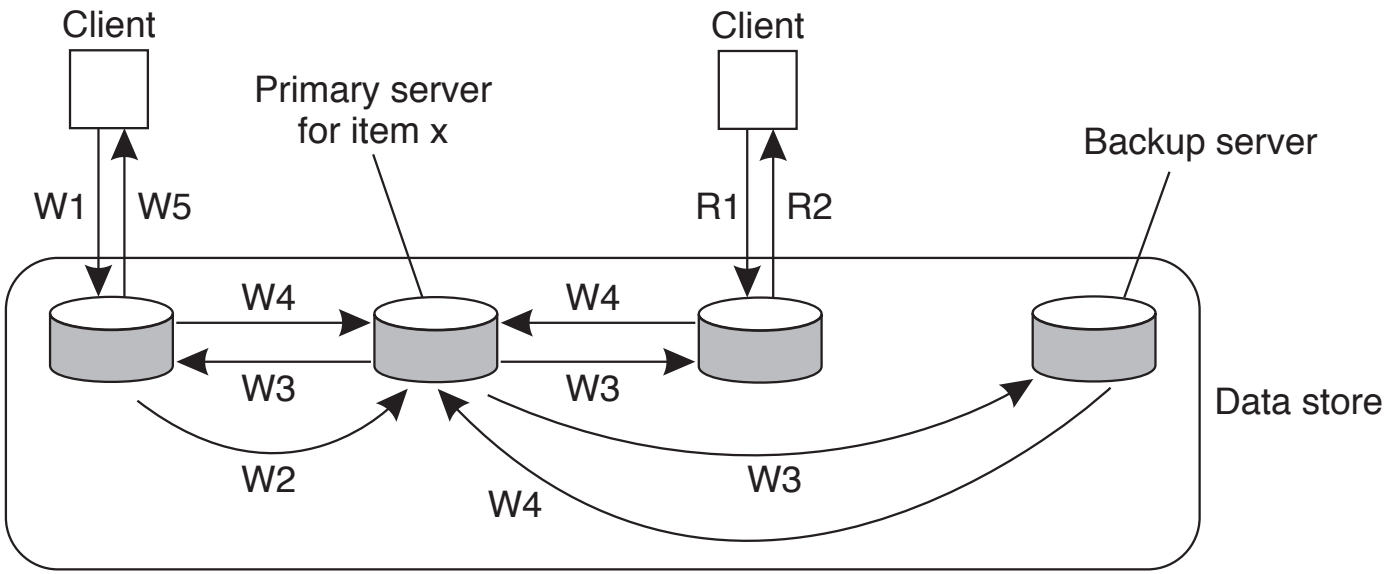
Consistency protocol

Describes the implementation of a specific consistency model.

- Continuous consistency
- Primary-based protocols
- Replicated-write protocols

Primary-based protocols

Primary-backup protocol



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

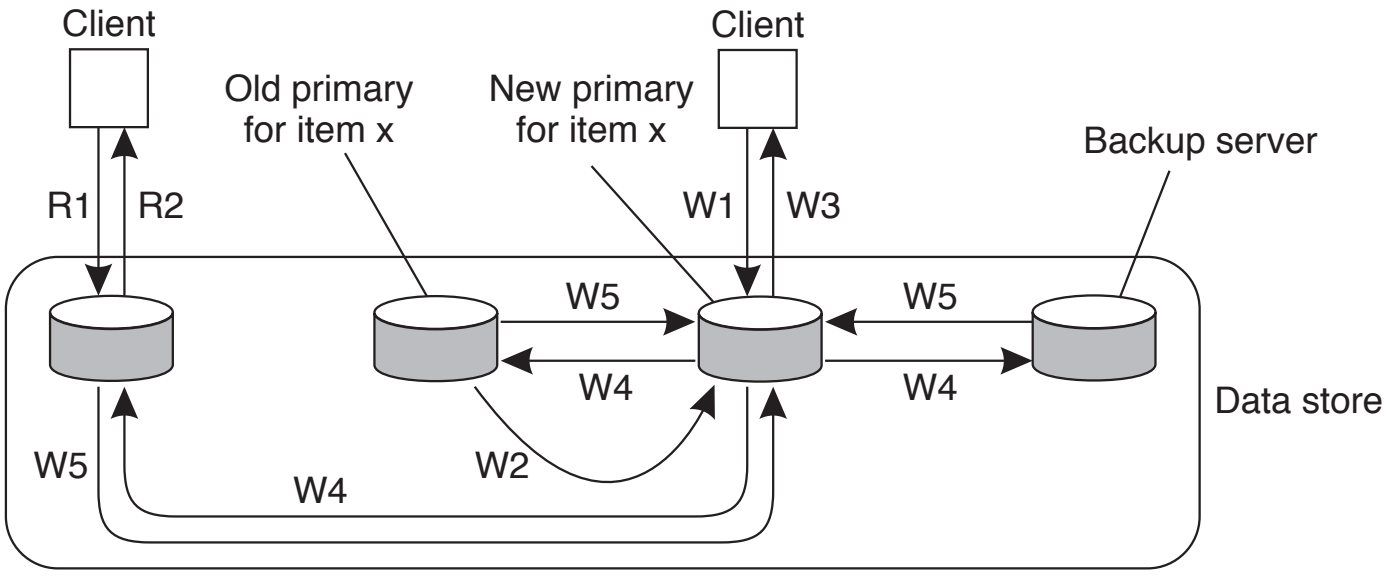
Primary-based protocols

Example primary-backup protocol

Traditionally applied in distributed databases and file systems that require a high degree of fault tolerance. Replicas are often placed on same LAN.

Primary-based protocols

Primary-backup protocol with local writes



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-based protocols

Example primary-backup protocol with local writes

Mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later on).