Distributed Systems

(3rd Edition)

Maarten van Steen Andrew S. Tanenbaum

Chapter 07: Consistency & Replication

Edited by: Hicham G. Elmongui

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

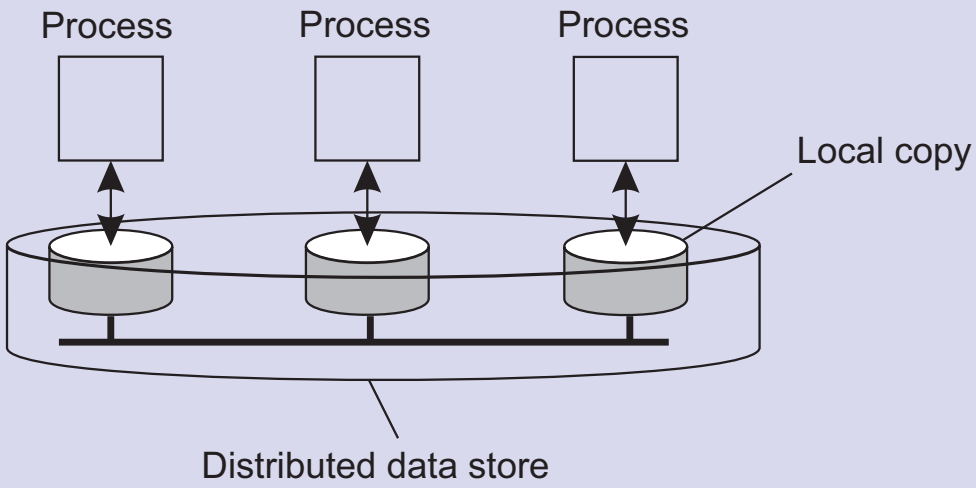
Data-centric consistency models

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:



Continuous Consistency

We can actually talk 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

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.

Behavior of two processes operating on the same data item.

P1:	W(x)a			
P2:			R(x)NIL	R(x)a

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

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

(a)

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

(b)

Causal consistency

Definition

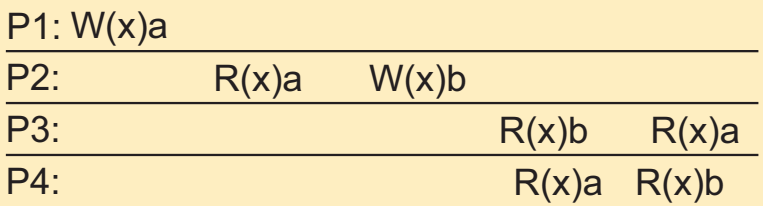
Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order by different processes.

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

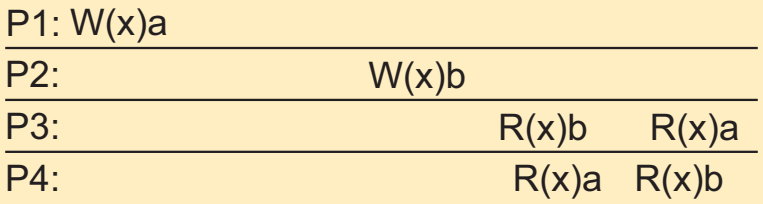
P1:	W(x)a				W(x)c		
P2:		R(x)a	W(x)b				
P3:		R(x)a				R(x)c	R(x)b
P4:		R(x)a				R(x)b	R(x)c

Causal consistency

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

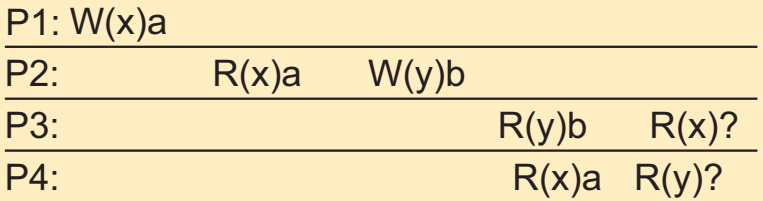


(a)



(b)

A slight modification of the figure above. What should $R_3(x)$ or $R_4(y)$ return?



Grouping operations

Definition

- Accesses to locks are sequentially consistent.
- No access to a lock 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 locks 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.

Grouping operations

A valid event sequence for entry consistency

P1:	L(x)	W(x)a	L(y)	W(y)b	U(x)	U(y)						
P2:							L(x)	R(x)a			R(y)	NIL
P3:									L(y)	R(y)b		

Observation

Entry 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?

Consistency versus coherence

Consistency models

A consistency model describes what can be expected with respect to a set of data items when multiple processes concurrently operate on that data. The set is then said to be consistent if it adheres to the rules described by the model.

Coherence models

Coherence models describe what can be expected to hold for only a single data item. A replicated data item is said to be coherent when the various copies abide to the rules as defined by its associated consistency model.

Eventual Consistency

How fast should updates be made available to only-reading processes?

In many database systems, most processes hardly ever perform update operations; they mostly read data from the database. Only one, or very few processes perform update operations.

How often do write-write conflicts occur?

- Web pages are updated by a single authority (webmaster or page owner).
- Only the naming authority of a DNS domain updates its part of the NS.

Eventual consistency

These examples can be viewed as cases of (large scale) distributed and replicated databases that tolerate a relatively high degree of inconsistency. They have in common that if no updates take place for a long time, all replicas will gradually become consistent, that is, have exactly the same data stored.

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.

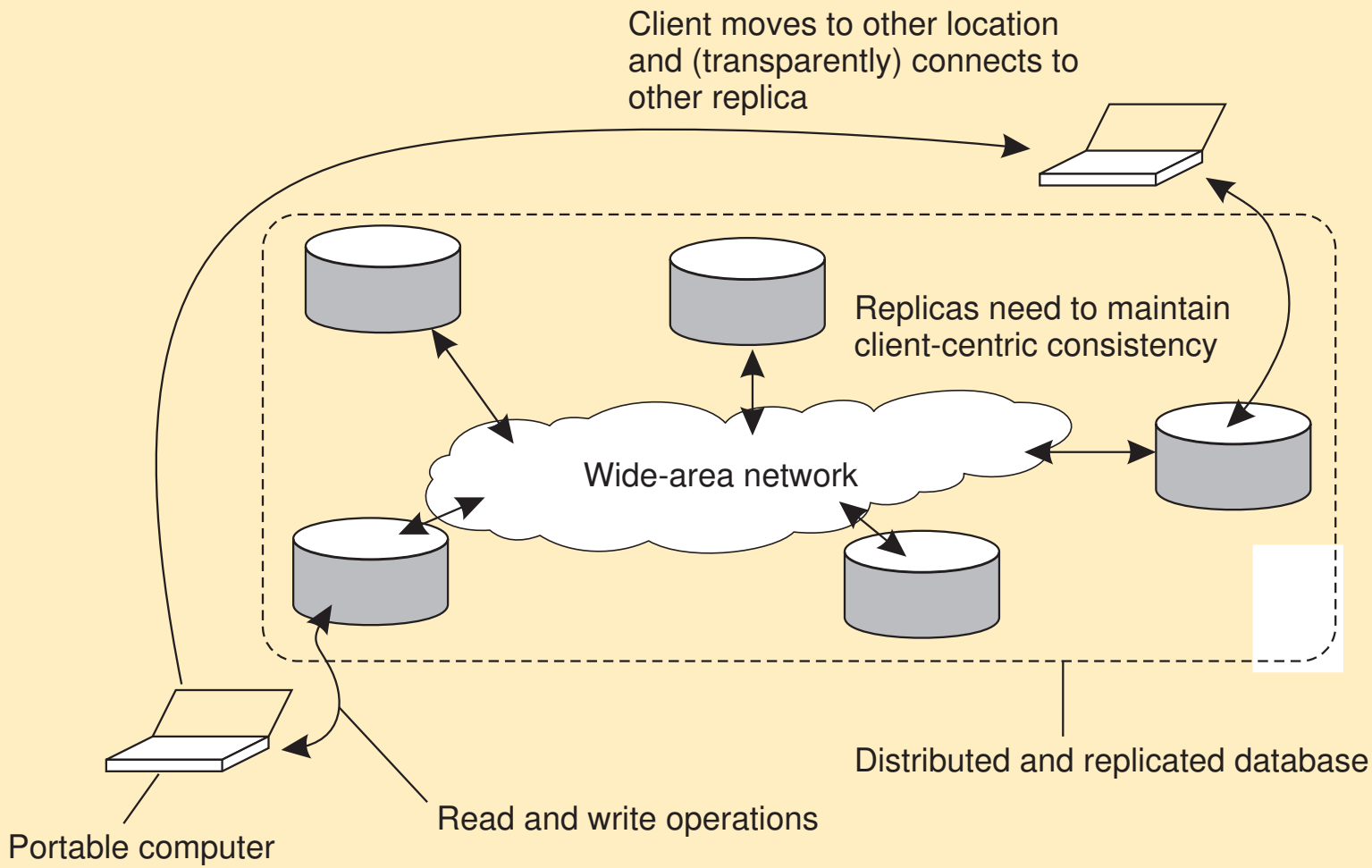
- At location *A* you access the database doing reads and updates.
- At location *B* you continue your work, but unless you access the same server as the one at location *A*, you may detect inconsistencies:
  - your updates at *A* may not have yet been propagated to *B*
  - you may be reading newer entries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

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.

Basic architecture

The principle of a mobile user accessing different replicas of a distributed database



Client-centric consistency: notation

Notation

- $W_1(x_2)$ is the write operation by process P_1 that leads to version x_2 of x
- $W_1(x_i; x_j)$ indicates P_1 produces version x_j based on a previous version x_i.
- $W_1(x_i|x_j)$ indicates P_1 produces version x_j concurrently to version x_i.

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

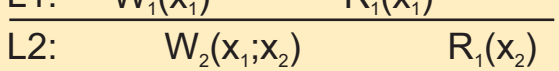
Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

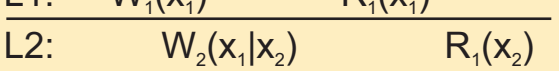
Monotonic reads

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.

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





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.

Example

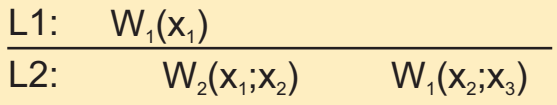
Updating a program at server S_2, and ensuring that all components on which compilation and linking depends, are also placed at S_2.

Example

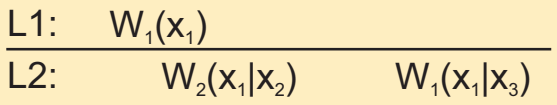
Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

Monotonic writes

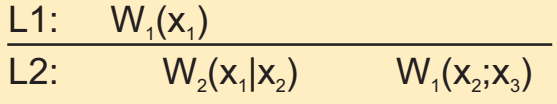
(a) A monotonic-write consistent data store. (b) A data store that does not provide monotonic-write consistency. (c) Again, no consistency as $WS(x_1|x2)$ and thus also $WS(x_1|x_3)$. (d) Consistent as $WS(x_1; x_3)$ although x_1 has apparently overwritten x_2.



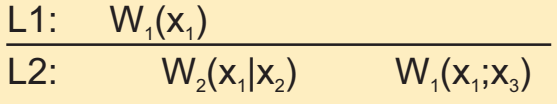
(a)



(b)



(c)



(d)

Read your writes

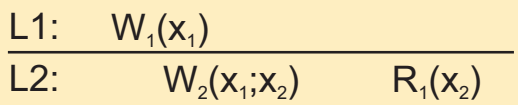
Definition

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.

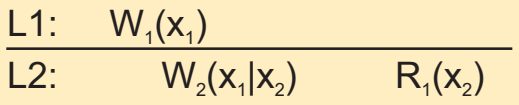
Example

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

(a) A data store that provides read-your-writes consistency. (b) A data store that does not.



(a)



(b)

Writes follow reads

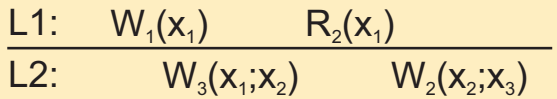
Definition

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.

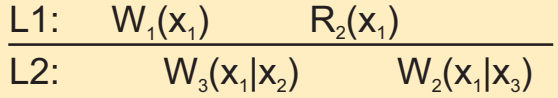
Example

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

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



(a)



(b)

Replica placement

To support replication, one has to:

- decide where replicas should be placed.
- decide when replicas should be placed.
- decide by whom replicas should be placed.
- decide which mechanisms to use for keeping the replicas consistent.

The placement problem itself should be split into two sub-problems:

- Replica-server placement is concerned with finding the best locations to place a server that can host (part of) a data store.
- Content placement deals with finding the best servers for placing content.

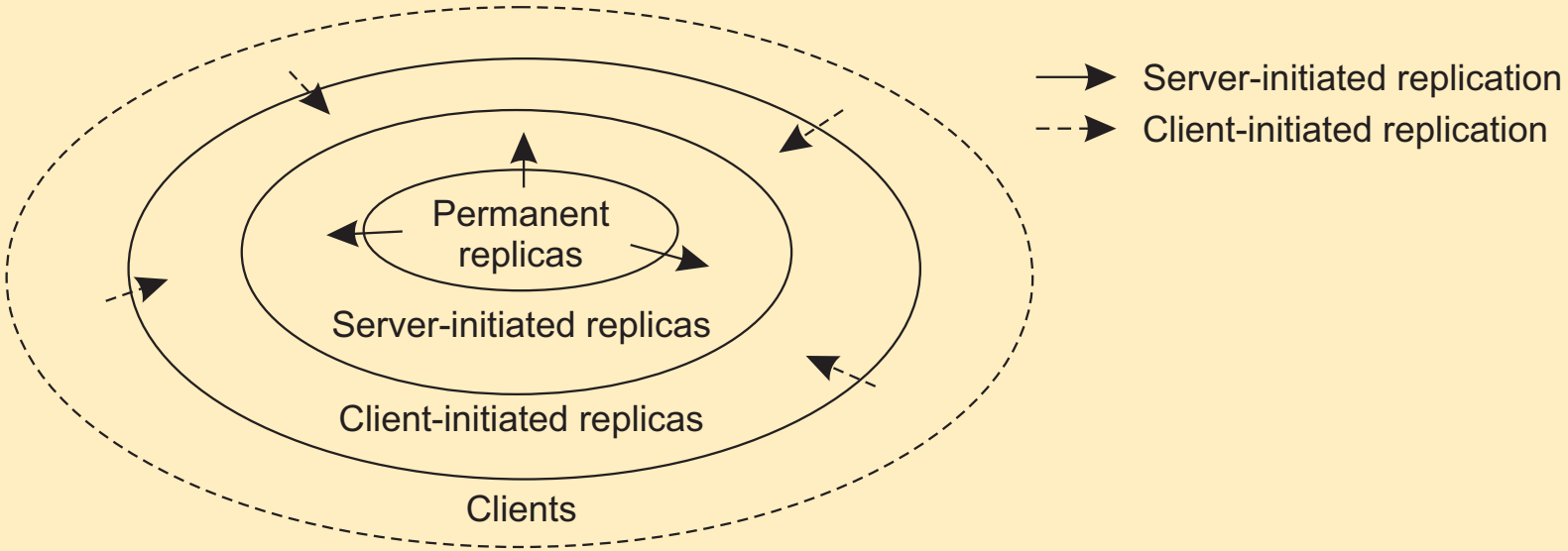
With the advent of the many large-scale data centers located across the Internet, and the continuous connectivity improvement, *precisely* locating servers becomes less critical!

Content replication

Distinguish different processes

- 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)

Logical organization of diff. kinds of copies of a data store: 3 concentric rings



Content distribution

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): passive replication
- Propagate the update operation to other copies: active replication

Note

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

Content distribution: client/server system

A comparison between push-based and pull-based protocols in the case of multiple-client, single-server systems

- 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.

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

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.

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 an 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

Why are we doing all this?

- Trying to reduce the server's state as much as possible while providing strong consistency.

Content Distribution

In many cases, it is cheaper to use available multicasting facilities.

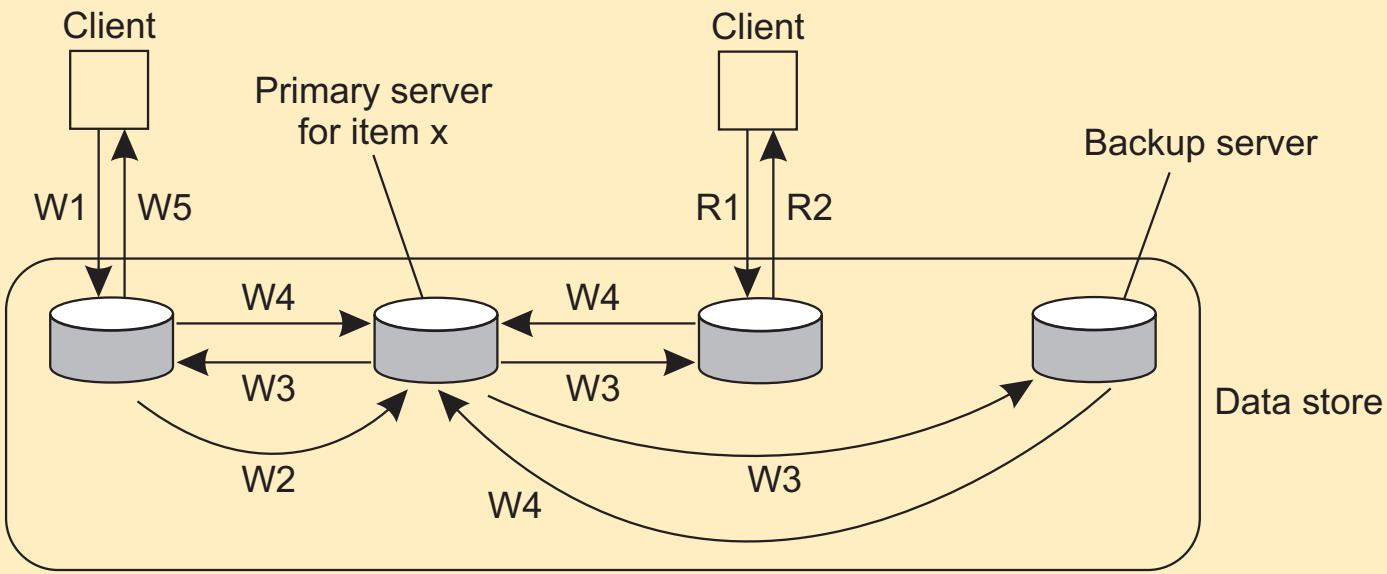
- Multicasting can often be efficiently combined with a push-based approach to propagating updates.
- When the two are carefully integrated, a server that decides to push its updates to a number of other servers simply uses a single multicast group to send its updates.

In other cases, unicasting may be the most efficient solution

- With a pull-based approach, it is generally only a single client or server that requests its copy to be updated.

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

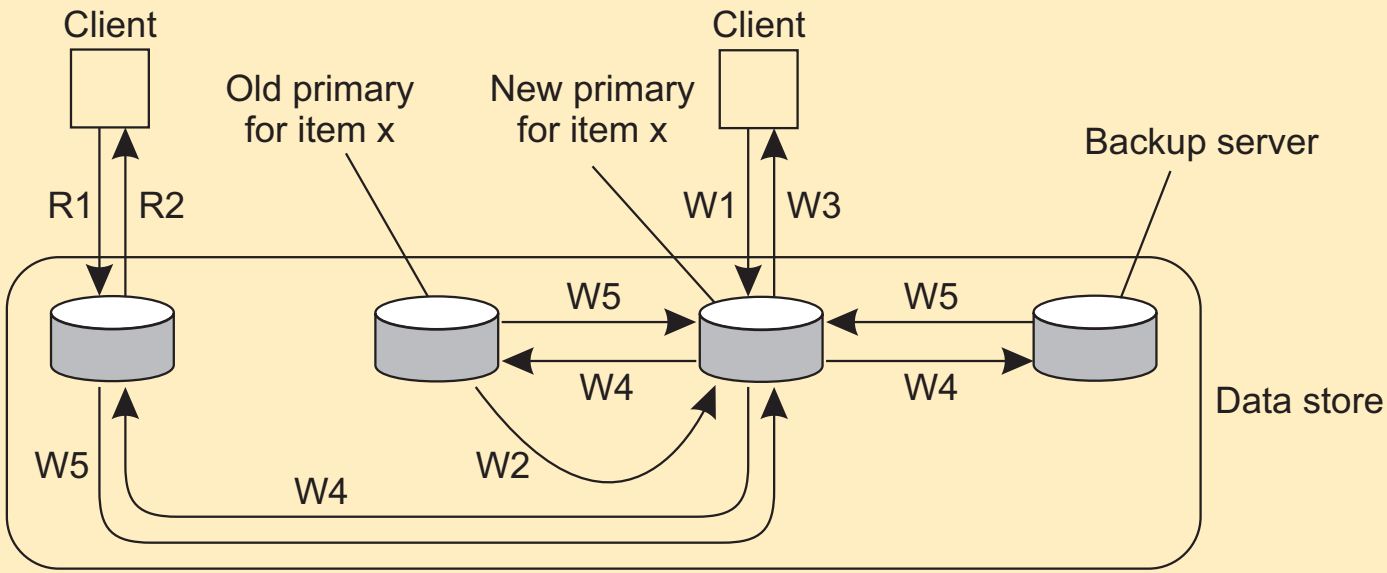
R1. Read request
R2. Response to read

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

Example primary-backup protocol with local writes

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

Replicated-write protocols

In replicated-write protocols, write operations can be carried out at multiple replicas instead of only one, as in the case of primary-based replicas.

Active replication

- Each replica has an associated process that carries out update operations.
- The write operation (or the update itself) is propagated to each replica.

A total ordered multicast mechanism is used carry out the operations

- Use a central coordinator, a sequencer, which assigns a unique sequence number to each operation.
- Operations are carried out at each replica in the order of their sequence number.

Replicated-write protocols

Quorum-based protocols

Ensure that each operation is carried out in such a way that a majority vote is established: distinguish read quorum and write quorum

- $N_R + N_W > N$
- $N_W > N/2$

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)