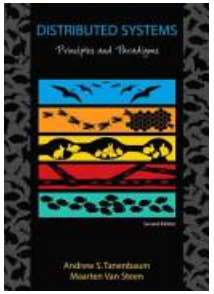
Chapter 7: CONSISTENCY AND REPLICATION

How to keep all the copies same?



Thanks to the authors of the textbook [**TS**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.

Turgay Korkmaz
korkmaz@cs.utsa.edu

Chapter 7: CONSISTENCY AND REPLICATION

- INTRODUCTION
  - Reasons for Replication, Replication as Scaling Technique
- DATA-CENTRIC CONSISTENCY MODELS
  - Continuous Consistency, Consistent Ordering of Operations
- CLIENT-CENTRIC CONSISTENCY MODELS
  - Eventual Consistency
  - Monotonic Reads, Monotonic Writes
  - Read Your Writes, Writes Follow Reads
- REPLICA MANAGEMENT
  - Replica-Server Placement, Content Replication and Placement
  - Content Distribution
- CONSISTENCY PROTOCOLS
  - Continuous Consistency
  - Primary-Based Protocols
  - Replicated-Write Protocols
  - Cache-Coherence Protocols
  - Implementing Client-Centric Consistency

Objectives

- To understand replication and related issues in DS
- To learn about how to keep multiple replicas consistent with each other

Reasons for Replication

- Data are replicated
  - To increase the reliability of a system.
  - To improve performance
    - Scaling in numbers
    - Scaling in geographical area (e.g., place copies of data close to the processes using them. So clients can quickly access the content.)
- Problems
  - How to keep replicas *consistent*
    - Distribute replicas
    - Propagate modifications
  - Cost of increased resources and bandwidth for maintaining consistent replications

(scalability?)

Does it itself Scale?

- What if there is an update?
  - **Update all in an atomic way** (sync replication)
  - To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the same order everywhere
    - Read–write conflict: a read operation and a write operation act concurrently
    - Write–write conflict: two concurrent write operations
  - This itself may create scalability problem, making the cure worse than the disease!
- Solution
  - Loosen the consistency constraint so that hopefully global synchronization can be avoided
  - Depends on application but improves performance

Data-centric

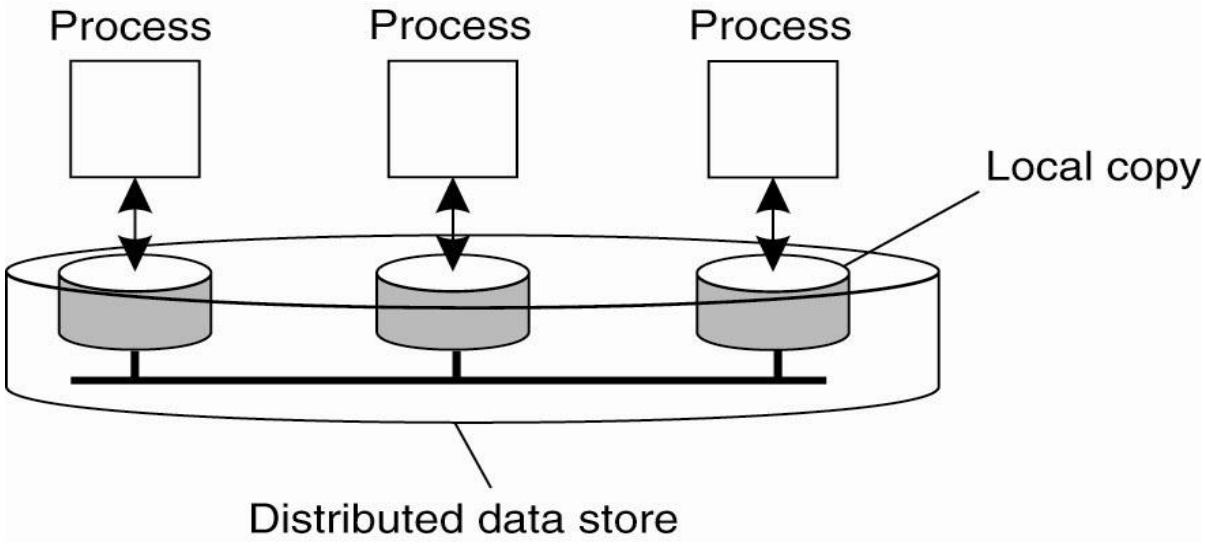
Client-centric

CONSISTENCY MODELS

Data-centric Consistency Models

A data store is a distributed collection of storages, R-W

System-wide consistency



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. Read the last write!

Without a global clock, it is hard to define precisely which write is the last one! So we need other definitions [degree/range of consistency]

Continuous Consistency

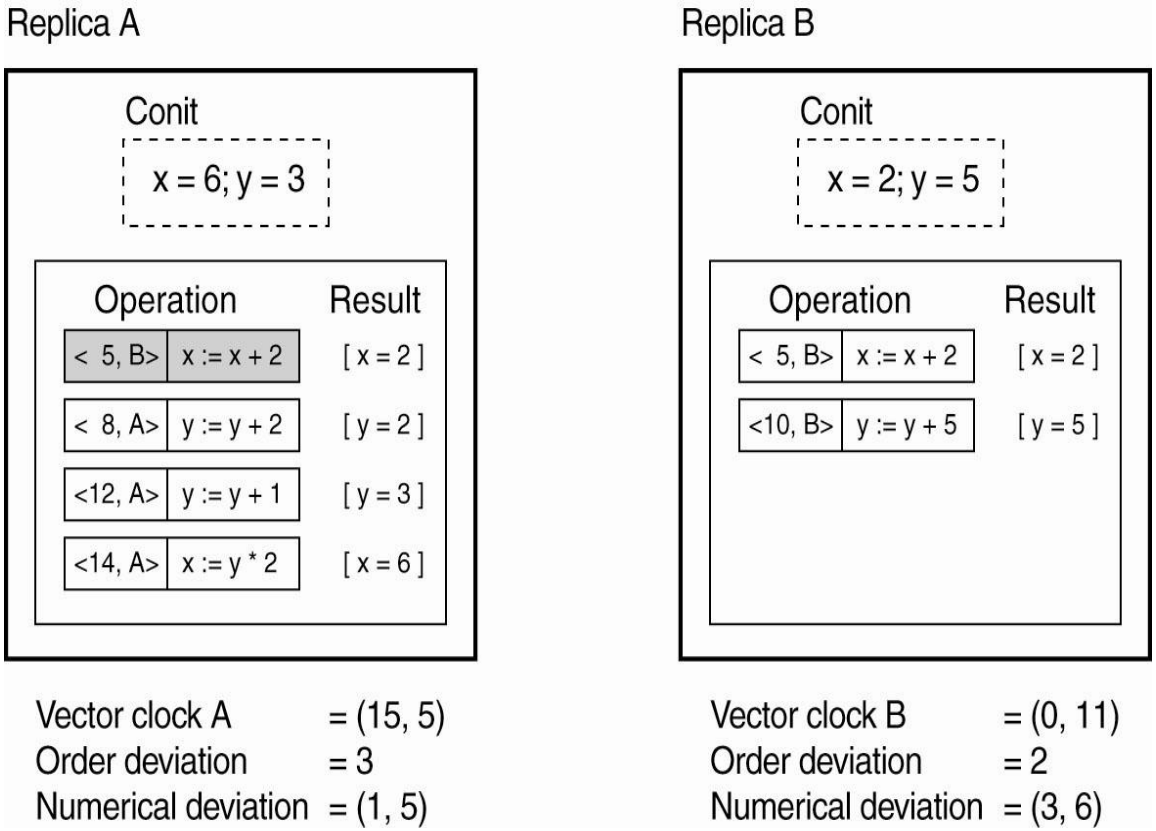
- We can actually talk about a degree of consistency:
  - replicas may differ in their numerical value
  - replicas may differ in their relative staleness
  - there may be differences with respect to number and order of performed update operations
- Examples
  - Replication of stock market prices (e.g., no more than $.02 or 0.5% difference between any two copies)
  - Duration of updates (e.g., weather reports stay accurate over some time, web pages)
  - Order of operations could be different (e.g., see next slide)

Continuous Consistency (cont'd)

- **Conit** (**Con**sistency un**it**) specifies the data unit over which consistency is to be measured (e.g., a stock value)
- An example of keeping track of consistency deviations

Conit (contains the variables x and y)

- Each replica maintains a vector clock
- B sends A operation <5,B>: x := x +2;
- A has made this operation permanent (cannot be rolled back)
- A has three pending operations → order deviation = 3
- A has missed one operation from B, yielding a max diff of 5 units → (1,5)



How to reach a global order of operations applied to replicated data so we can provide a system-wide consistent view on data store?

Comes from concurrent programming.

Sequential consistency

Causal consistency

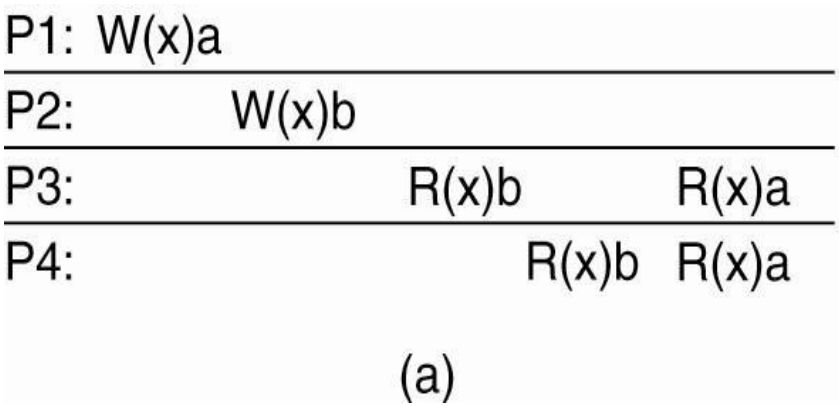
CONSISTENT ORDERING OF OPERATIONS

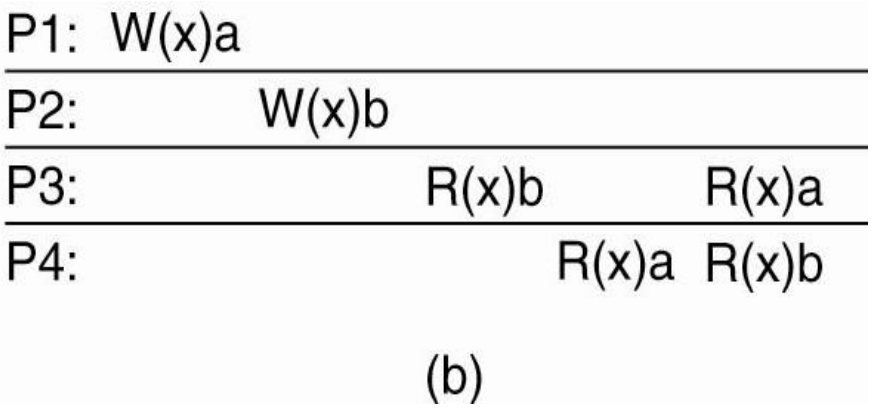
Sequential Consistency (1)

- 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. The horizontal axis is time.

it took sometime to propagate new value of x

Sequential Consistency (2)





(a) A sequentially consistent data store.

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

- Any valid interleaving of R and W is acceptable as long as all processes see the same interleaving of operations.
- Everyone sees all W in the same order

Sequential Consistency (3)

- Three concurrently-executing processes.

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

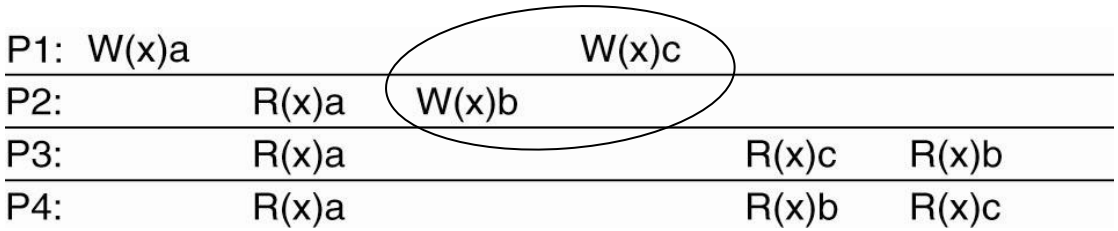
valid execution sequences

(a)	(b)	(c)	(d)
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
Signature: 001011	Signature: 101011	Signature: 110101	Signature: 111111

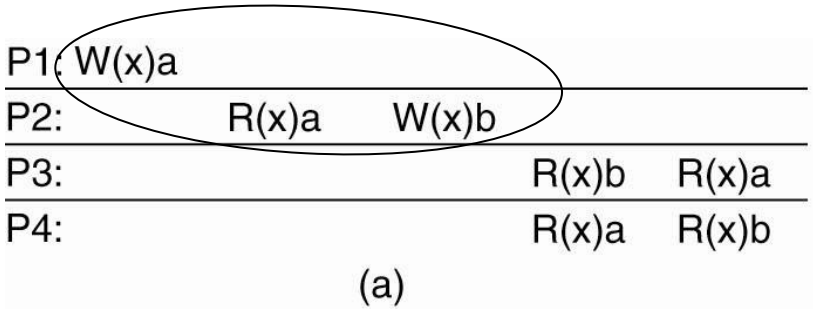
Causal Consistency (1)

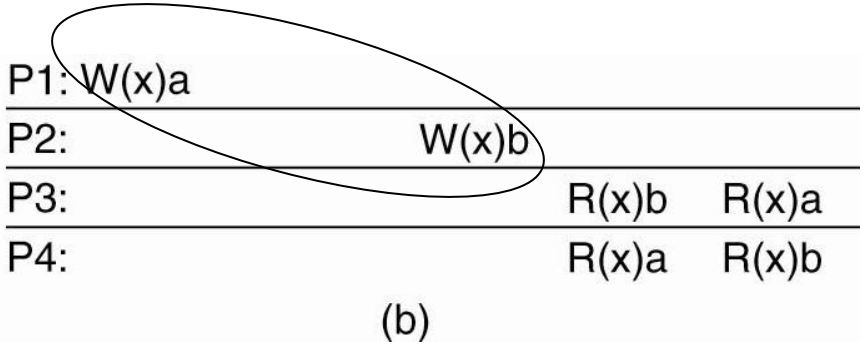
- Weakening of sequential consistency
  - Instead of all, only causally related W should be seen in the same order…
- For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:
  - Writes that are potentially causally related must be seen by all processes in the same order.
  - Concurrent writes may be seen in a different order on different machines.
- If event b is caused by an earlier event a, a→b
  - P1: Wx P2: Rx then Wy, then Wx → Wy (potentially causally related)

Causal Consistency (2)



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





(a) A violation of a causally-consistent store

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

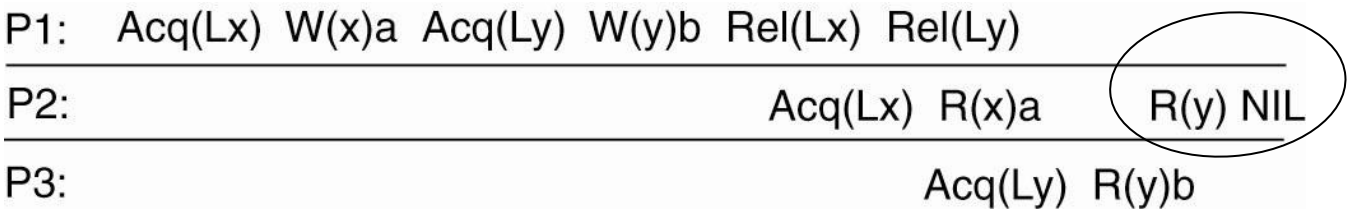
Implementing causal consistency requires keeping track of which processes have seen which writes…
Construct a dependency graph using vector timestamps…

Grouping Operations (1)

- Previous R and W granularities are due to historic reason (they were developed for shared-memory multiprocessor systems)
- In a DS, instead of making each R and W immediately known to other processes, we just want the effect of the series of such operations to be known.
- So use synchronization
  - Enter_CS … multiple R and W… Leave_CS
  - Level of granularity is increased

Grouping Operations (2)

- Semantic for Synchronization variables
  - 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.
- A valid event sequence for entry consistency.



- How to associate data with sync variables:
  - Explicit tell middleware which sync var is for which data
  - Implicit (like one lock per obj in OO)

Show how we can perhaps avoid data-centric (system-wide) consistency, by concentrating on what specific clients want, instead of what should be maintained by all servers as in data-centric models.

CLIENT-CENTRIC CONSISTENCY MODELS

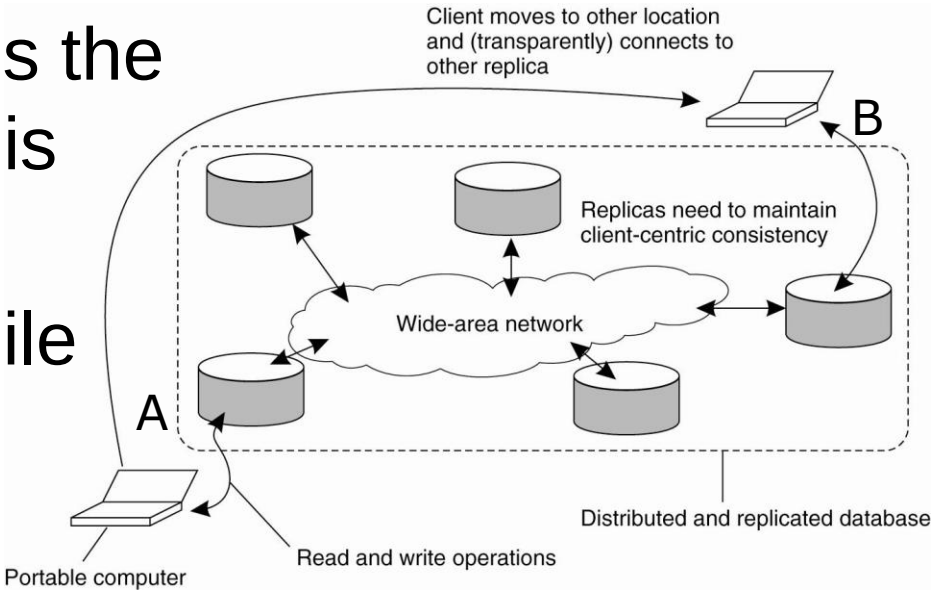
Eventual Consistency (1)

- Observation: In some applications, most processes hardly ever perform updates while a few do updates
- How fast updates should be made available to only reading processes (e.g., DNS)
  - Consider WWW pages…
    - To improve performance clients cache web pages. Caches might be inconsistent with original page for some time…
    - Eventually all will be brought up to date
- **Eventual consistency**:

  If no updates take place for a long time, all replicas will become consistent

Eventual Consistency (2)

- As long as a client access the same replica, then there is no problem…
- But when the client (mobile one) accesses different replica, then we have a problem…



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

Eventual Consistency (3)

- In the previous example, the only thing we really want is that the entries we updated and/or read at A are in B the way we left them in A.
- This way the database will appear to be consistent to us (client).
- That is what client-centric consistency is all about!
- There are four models under the following settings:
  - All R & W are performed locally and eventually propagated to all
  - Data items have an associated owner which is permitted to modify data to avoid W-W conflicts

Monotonic Reads

Monotonic Writes

Read Your Writes

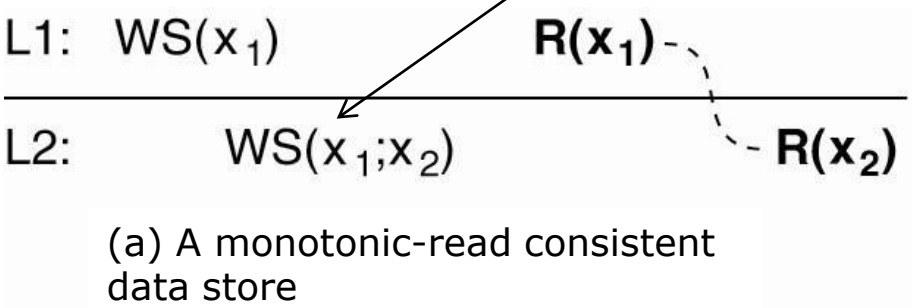
Writes Follow Reads

Monotonic Reads

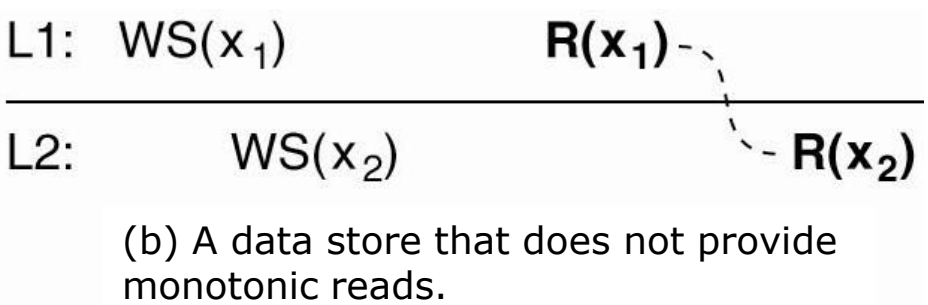
$WS(x_i[t])$ is the set of write operations (at L_i) that lead to version x_i of x (at time t)
$WS(x_i[t_1];x_j[t_2])$ indicates that it is known that $WS(x_i[t_1])$ is part of $WS(x_j[t_2])$.
Note: Parameter t is omitted from figures.

A data store is said to provide monotonic-read consistency if the following condition holds:

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



(a) A monotonic-read consistent data store



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

IMP: When a client reads from a server, that server gets the clients R set to check if all Ws have taken place locally. If not, it contacts the other servers to ensure that it is brought up to date before read operation

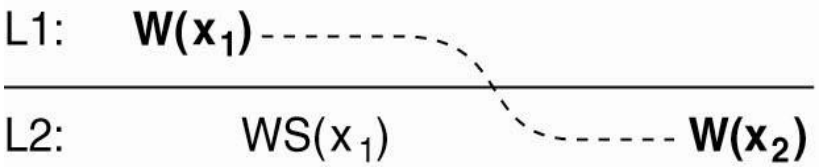
Example 1: Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

Example 2: 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.

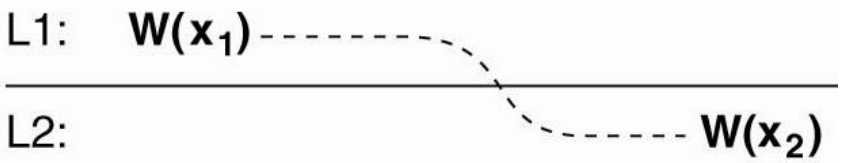
Monotonic Writes

In a monotonic-write consistent store, the following condition holds:

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



(a) A monotonic-write consistent data store.



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

IMP: When a client initiates W at a server, the server gets client's W set and makes sure identified W operations performed first and in correct order

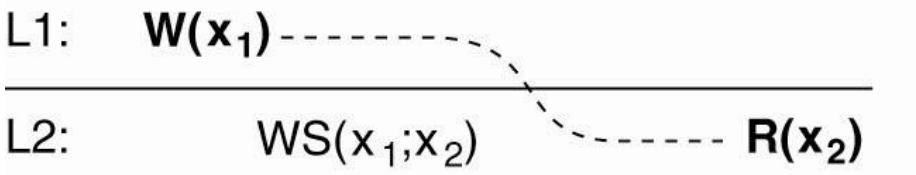
Example 1: Updating a program at server S2, and ensuring that all components on which compilation and linking depends, are also placed at S2.

Example 2: Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

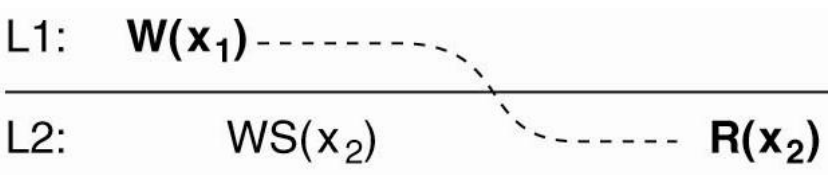
Read Your Writes

A data store is said to provide read-your-writes consistency, if the following condition holds:

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



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



(b) A data store that does not.

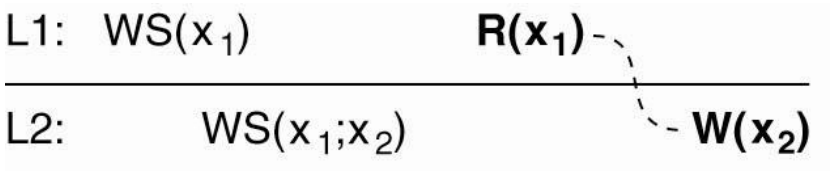
IMP: The server where the read operation is performed has seen all the write operations in client's W set. Simple fetch writes from other servers before read

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

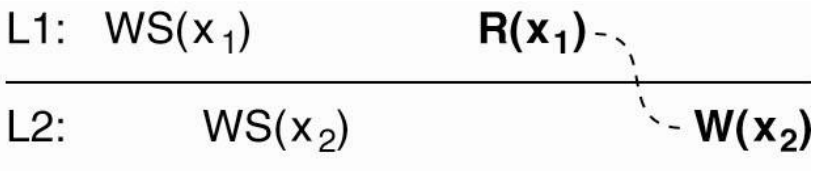
Writes Follow Reads

A data store is said to provide writes-follow-reads consistency, if the following holds:

- A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.



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



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

Bring the selected server up to date with the write operations in client's R set

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

Where, when, and by whom replicas should be placed and

Which mechanisms to use for keeping them consistent?

Placement of

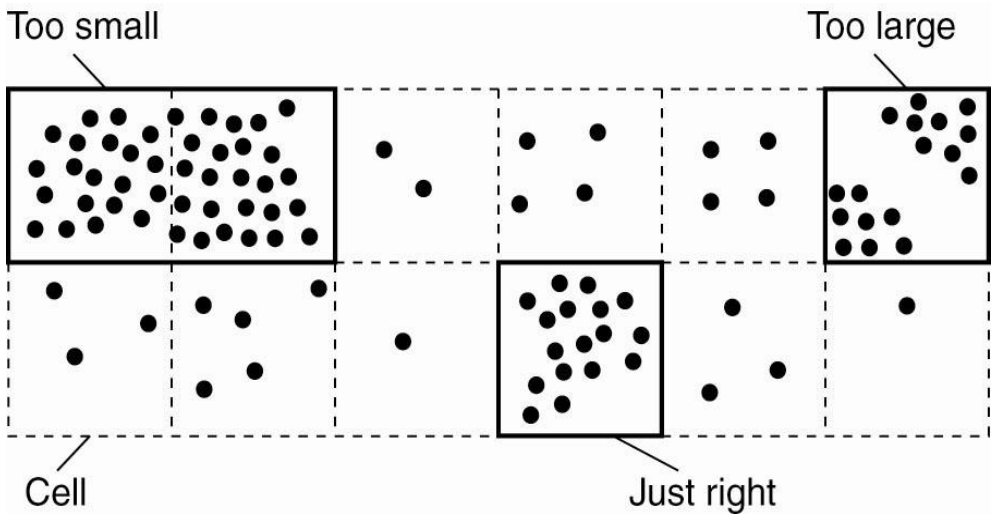
Servers (find the best location)

Content (find the best server)

REPLICA MANAGEMENT

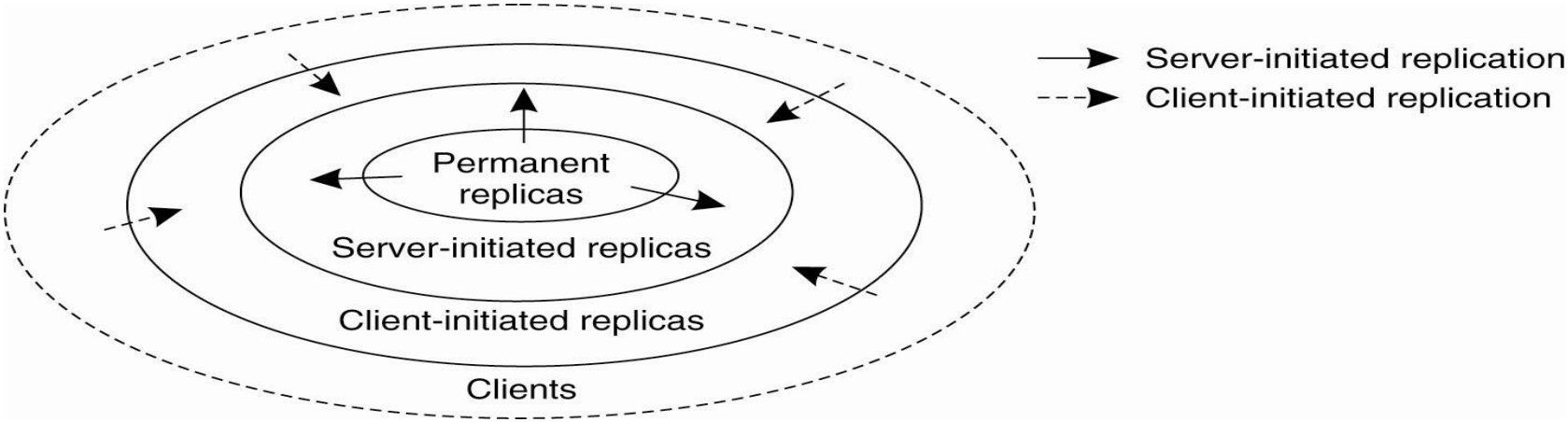
Replica-Server Placement

- K out of N locations need to be selected:
  - take distances between clients and possible locations and min distance.
- Look at AS view of the Internet, put server into ASes with larger degree
- Position nodes in m-dim geometric space and identify K largest cells
  - How to choose a proper cell size for server placement?



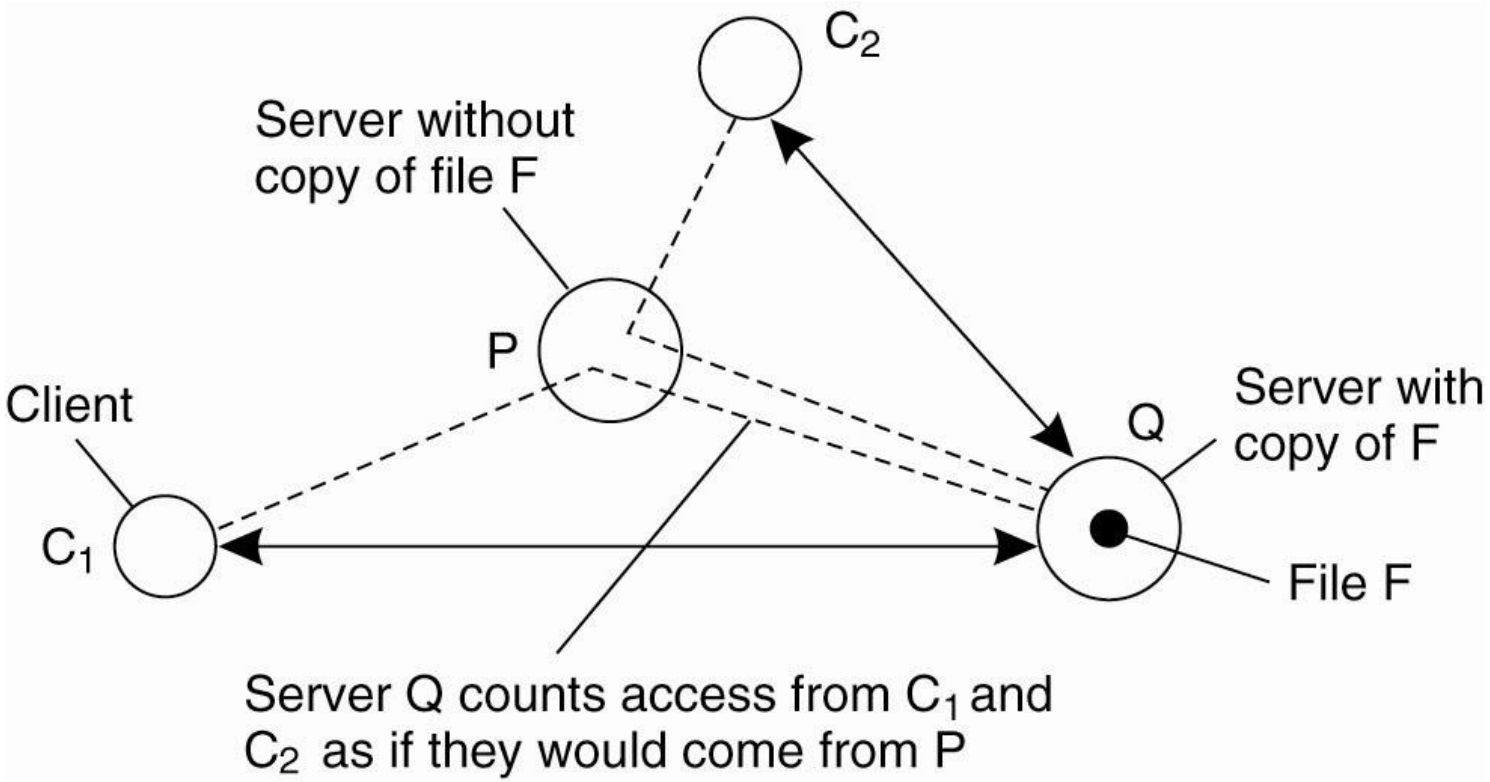
Appropriate cell size can be computed as a function of average distance between the nodes and number of required replicas O(Nxmax{log(N),K})

Content Replication and Placement



- **Permanent replicas**: Initial set of processes and machines always having a replica (web site mirrors)
- **Server-initiated replica**: Processes can dynamically host a replica on request of another server in the data store (move popular files toward clients)
- **Client-initiated replica**: Processes can dynamically host a replica on request of a client (client cache)

Server-Initiated Replicas



- How to determine which files need to be replicated and where?
- Counting access requests from different clients.

Client-Initiated Replicas

- Client caches
  - Local storage, management is left to client
- Keep it for a limited time
- For consistency client may want server to cooperate
  - (modified-since-then.. In HTML)
- Cache hit rate is important for performance
- Server-initiated is becoming more common than client-initiated…. Why?

How to propagate the updated content to the relevant replicas?

CONTENT DISTRIBUTION

State versus Operations

What is to be propagated:

1. Propagate only a notification of an update.
   - Invalidation protocols use notifications to inform others
   - + little network overhed
   - + good when W >> R (r/w is small)
2. Transfer data from one copy to another.
   - + good when W << R (r/w is high)
3. Propagate the update operation to other copies.
   - + little network overhead
   - - requires same computation power at each replica

- No single approach is the best, highly depends on available bandwidth and r/w ratio at replicas.

Pull versus Push Protocols

- Pushing updates:
  - server-initiated approach, in which update is propagated regardless whether target asked for it. + good if r/w is high
- Pulling updates:
  - client-initiated approach, in which client requests to be updated. + good if r/w is low

Issue	Push-based	Pull-based
State at 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

- We can dynamically switch between pulling and pushing using leases (a hybrid form):
- Lease is a contract in which the server promises to push updates to the client until the lease expires.

Lease-based Hybrid Form

How to determine lease expiration time?

- Make it 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
- **Question**
  - Why are we doing all this?

Unicast vs. Multicast

- +/-
- N separate send vs. one send to N servers
- Pull-based--- unicast
- Push-based– multicast (broadcast in LAN)

CONSISTENCY PROTOCOLS

Describes the implementation of a specific consistency model.

Data-centric
- Continuous consistency
- **Primary-based protocols**
- **Replicated-write protocols**
- Cache-coherence protocols

Client-centric Consistency (we already mentioned naïve ways)

How to get globally consistent ordering?

Remote-Write Protocols

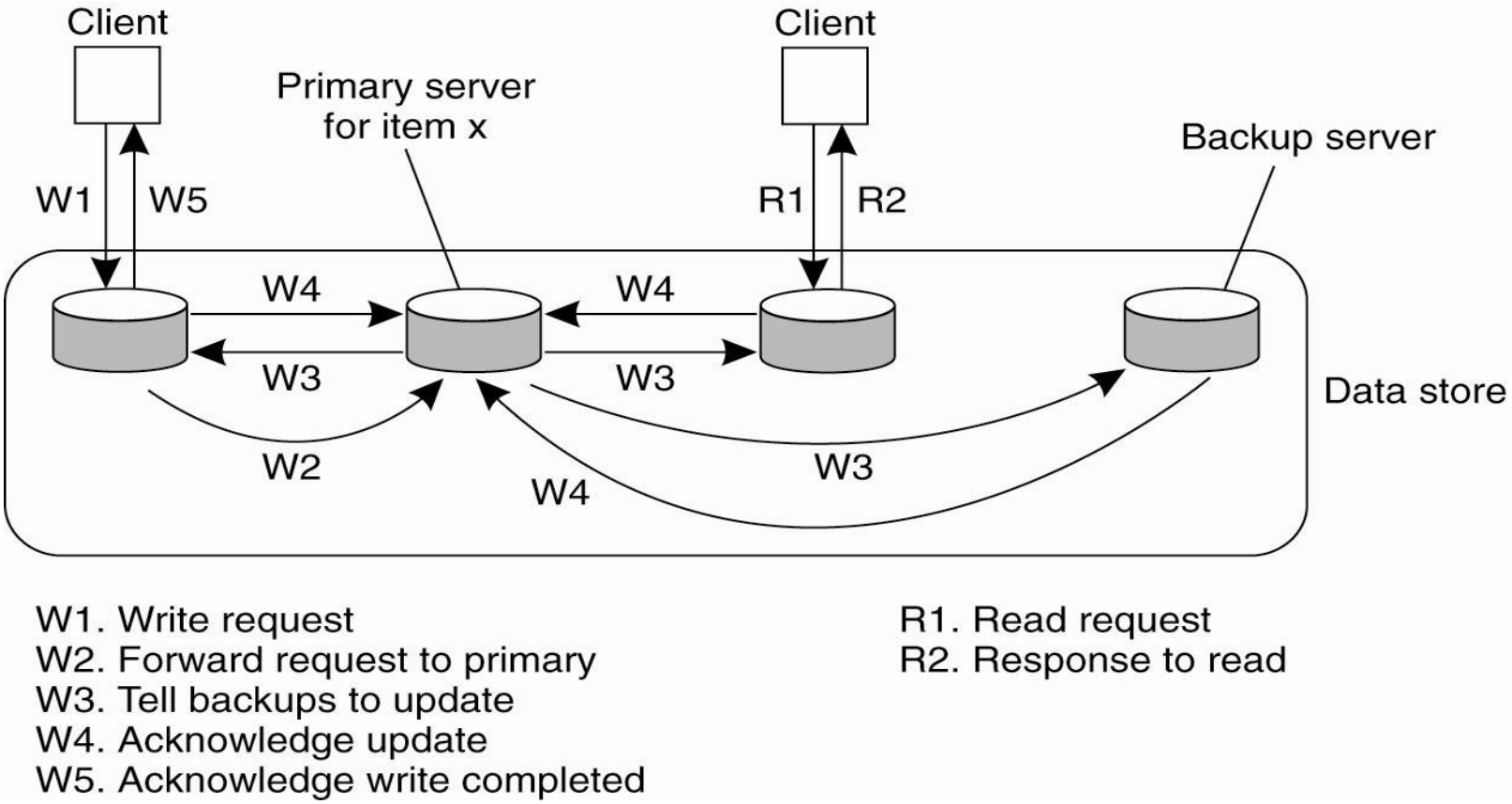
Local-Write Protocols

PRIMARY-BASED PROTOCOLS

Primary-based Protocols:

Remote-Write Protocols (primary-backup)

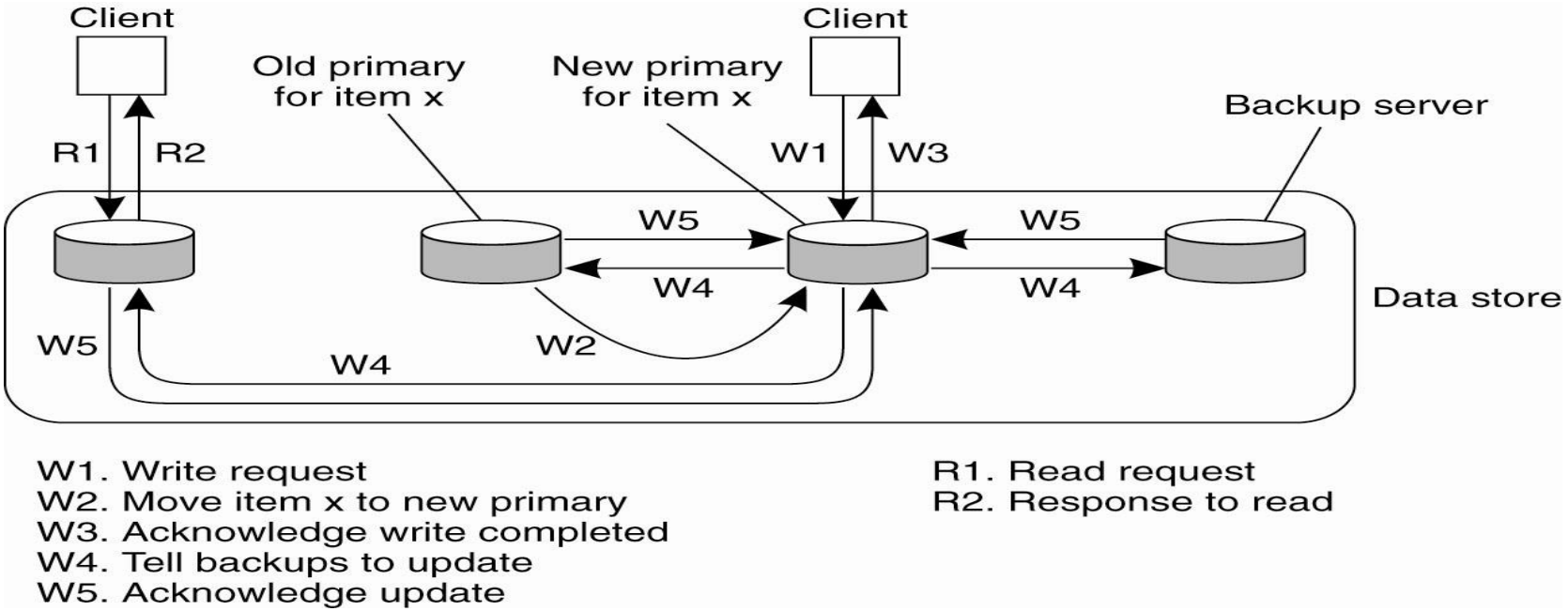
- All W need to be forwarded to a fixed single server
- Straightforward implementation of sequential consistency
- - blocking (non-blocking update is possible)



Primary-based Protocols:

Local-Write Protocols

- the primary copy migrates between the processes wanting to perform an update.



- also useful for mobile computers that can operate in disconnected mode (mobile node becomes the primary before disconnect)
- Can be nonblocking for better performance

Carry out W operations at multiple replicas

instead of one as in primary-based protocol

Active Replication

Quorum-Based Protocols

REPLICATED-WRITE PROTOCOLS

Replicated-Write Protocols:

Active Replication

- An operation is sent to every replica
- Execute operations in the same order everywhere,
- For this, we need a **totally-ordered multicast** (which can be implemented using Lamport's logical clock)
- But it does not scale well in large DS
- Instead use a central coordinator (sequencer) that assigns a unique sequence number to each W and forwards it to all replicas…
  - +/- ?

Replicated-Write Protocols:
Quorum-Based Protocols (1)

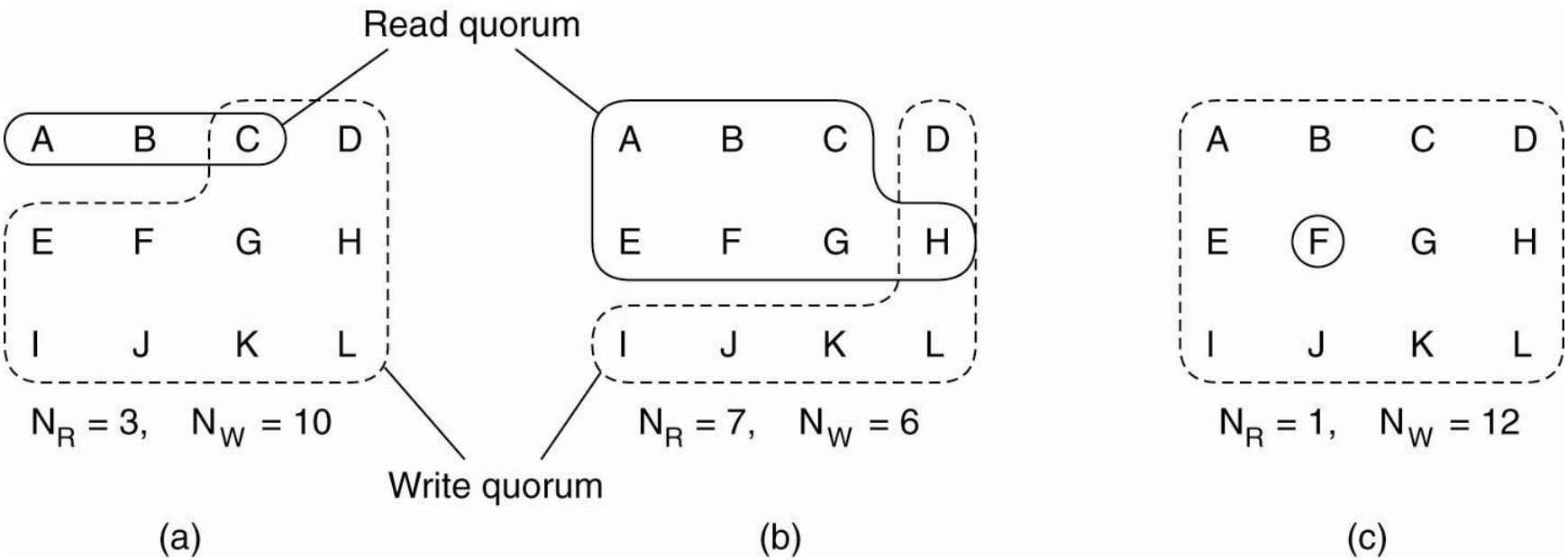
- Ensure that each operation is carried out in such a way that a majority vote is established:
- Example: Suppose a file is replicated on N servers
  - To update a file, contact at least N/2+1 servers. If they agree, change the file and associate a new version #
  - To read, contact at least N/2+1 servers. If all version numbers are the same, this is the most recent version…
- Gifford's scheme generalized this idea by distinguishing
  - N_R: read quorum and
  - N_w: write quorum that are subject to:

prevent read-write conflicts $N_R + N_w > N$

prevent write-write conflicts $N_w > N/2$

Replicated-Write Protocols:

Quorum-Based Protocols (2)



(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).

$N_R + N_w > N$ prevent read-write conflicts

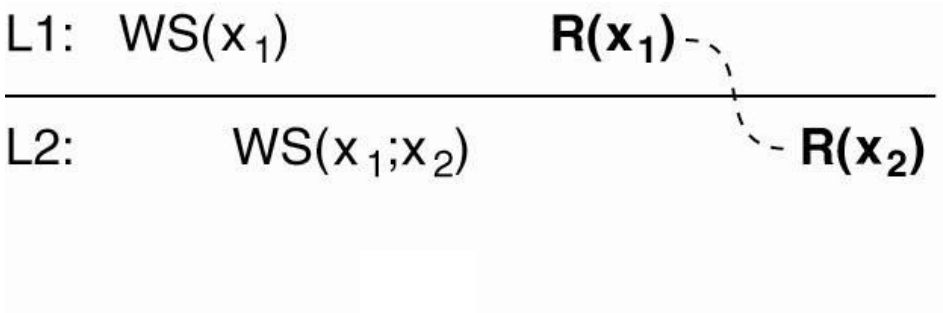
$N_w > N/2$ prevent write-write conflicts

Straightforward if performance issues are ignored

CLIENT-CENTRIC CONSISTENCY

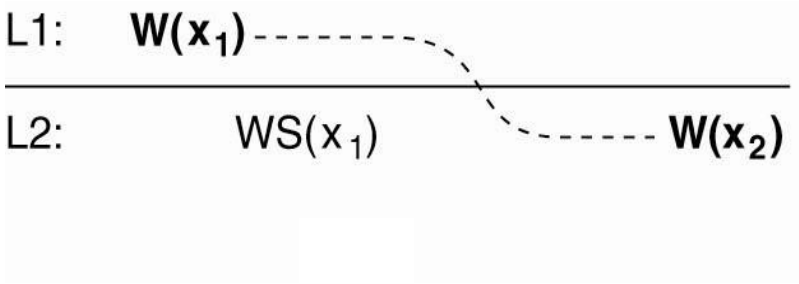
A Naïve implementation

- Each W is assigned a globally unique ID by the server to which W is submitted
- For each client, keep track of two sets of W:
  - R set: W relevant for the R performed by the client
  - W set: W performed by the client
- Monotonic Reads
  - When a client reads from a server, that server gets the clients R set to check if all Ws have taken place locally. If not, it contacts the other servers to ensure that it is brought up to date before read operation

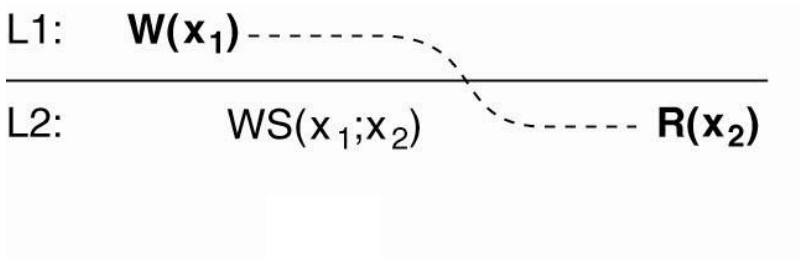


A Naïve implementation

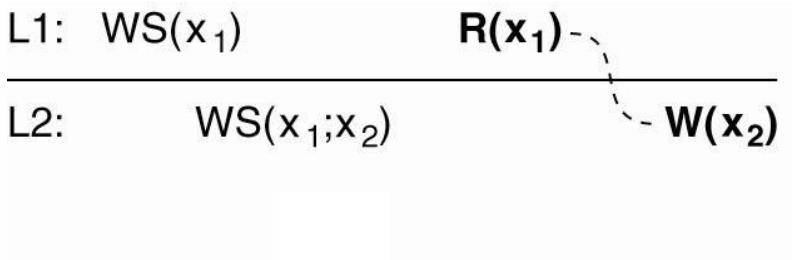
- Monotonic Writes
  - When a client initiates W at a server, the server gets client's W set and makes sure identified W operations performed first and in correct order



- Read Your Writes
  - The server where the read operation is performed has seen all the write operations in client's W set. Simple fetch writes from other servers before read



- Writes Follow Reads
  - Bring the selected server up to date with the write operations in client's R set



EXTRAS

Bounding numerical error

Bounding Staleness

Bounding Ordering Deviations

CONTINUOUS CONSISTENCY

Continuous Consistency:

Bounding numerical error (1)

- Consider a data item *x* and let *weight(W)* denote the numerical change in its value after a write operation *W*. Assume that $\forall W$: *weight(W)* > 0.
- *W* is initially forwarded to one of the N replicas, denoted as *origin(W)*.
- S_i keeps track of L_i of writes performed on its own local copy
- TW[i, j] are the writes executed by server S_i that originated from S_j: (aggregated writes submitted to S_i):
- TW[i, j] = Σ\{weight(W) | origin(W) = S_j and W $\in$ L_i\}

Continuous Consistency:

Bounding numerical error (2)

- Actual value $v(t)$ of x:

$$v(t) = v_{init} + \sum_{k=1}^{N} TW[k,k]$$

value v_i of x at replica i:

$$v_i = v_{init} + \sum_{k=1}^{N} TW[i,k]$$

Problem

We need to ensure that $v(t) - v_i < \delta_i$ for every server S_i.

Continuous Consistency:

Bounding numerical error (3)

General Approach:

- Let every server S_k maintain a **view** $TW_k[i, j]$ of what it believes is the value of TW[i, j].

  Note that: $0 \leq TW_k[i,j] \leq TW[i,j] \leq TW[j,j]$

- This information can be **gossiped** when an update is propagated.

Solution

- S_k sends operations from its log L_k to S_i when it sees that $TW_k[i,k]$ is getting too far from TW[k,k], in particular, when $TW[k,k] - TWk[i,k] > \delta_i / (N - 1)$.

Continuous Consistency:

Bounding Staleness Deviation

Solution (analogous to previous one)**:**

- Let every server S_k maintain a **real-time vector clock** $RVC_k[i] = T(i)$ to keep track of what has been seen last from S_i
- Suppose servers are loosely synchronized …
- Then, when S_k notes that $T(k) - RVC_k[i]$ is about to exceed the given time limit, then it starts pulling writes that are originated from S_i with a timestamp latter than $RVC_k[i]$

Continuous Consistency:
Bounding Ordering Deviation

- Each server will have a queue for tentative writes for which the actual order needs to be determined
- Specify the maximum length for these queues
- When the length at S_k exceeds the limit, S_k will no longer accepts any new writes and try to commit tentative writes after negotiating the correct order with other servers (i.e., we need global ordering of tentative writes)
- For this, primary-based or quorum-based protocols can be used which are discussed next….

Special case of replication controlled by client, but from consistency point of view they are similar to what we discussed so far…

Much work is done in the context of shared-memory systems and use hardware support

Solution should be software based in the context of DS

When inconsistencies are actually detected?

How caches are kept consistent with the copies at server?

CACHE-COHERENCE PROTOCOLS

Cache-coherence Protocols (when?)

- Static: compiler inserts instructions to deal with inconsistency
- Dynamic: (in DS) a check is made with the server to see if the cached data is modified
  - A distributed database may want to make sure that cached data is consistent before using it in a transaction
  - (optimistic) let process proceed while verification taking place. if it is consistent then performance improves; otherwise, abort transaction…
  - Check when about to commit

Cache-coherence Protocols (how?)

Do not allow shared data to be cached:

- Simple but limits performance improvements

Suppose shared data is allowed to be cached

- If the modification is done at server:
  1. Let the server send invalidation msg to all clients or
  2. Propagate the update
  - Which one would you select? Why?
- If the modification is done at clients:
  - Write-through cache
  - Write-back cache

Cache-coherence Protocols (how?)

- Write-through cache
  - Clients modified cached data and forward updates to server
  - Similar to primary-based local write (clients cache is temp-primary)
  - Client should have exclusive write permission to avoid w-w conflicts
- Write-back cache
  - Group multiple updates to further improve performance
- Used in distributed file systems…