Distributed Systems
Principles and Paradigms

Chapter 07
(version October 3, 2007)

Maarten van Steen

Vrije Universiteit Amsterdam, Faculty of Science
Dept. Mathematics and Computer Science
Room R4.20. Tel: (020) 598 7784
E-mail:steen@cs.vu.nl, URL: www.cs.vu.nl/∼steen/

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

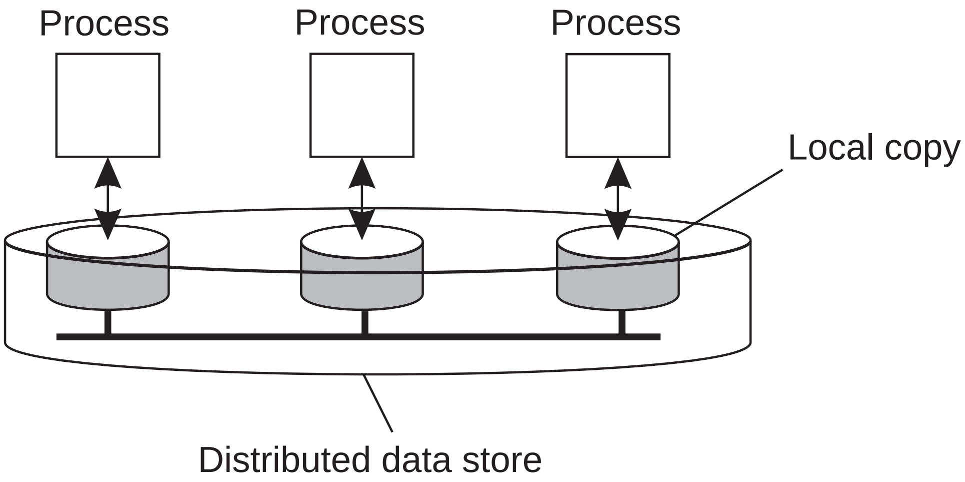
Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability

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.

Essence: A data store is a distributed collection of storages accessible to clients:



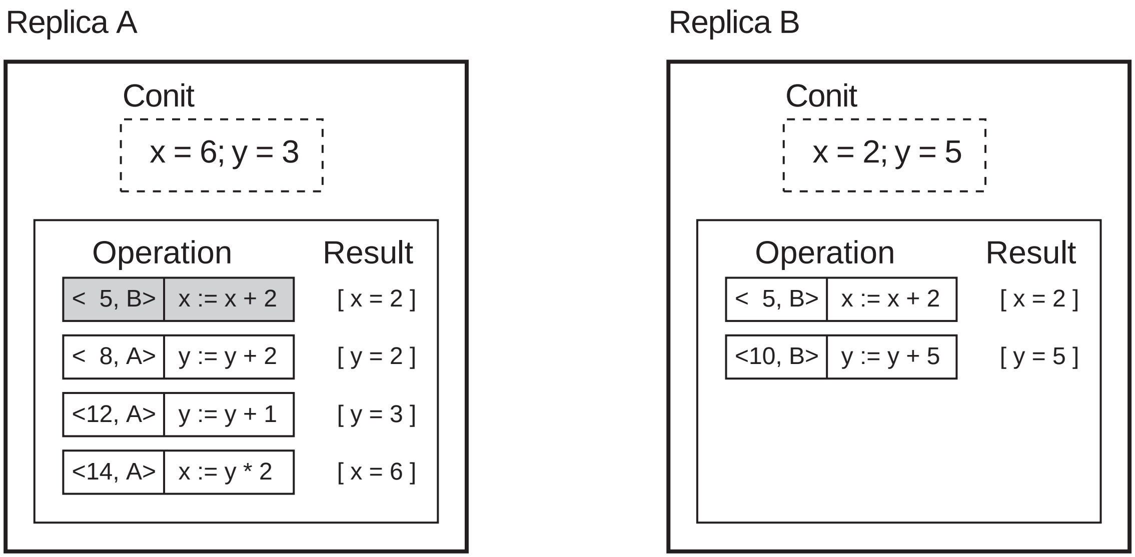
Continuos Consistency

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 differences with respect to (number and order) of **performed update operations**

Conit: conistency unit ⇒ specifies the **data unit** over which consistency is to be measured.

Example: Conit



Replica A

Conit

x = 6; y = 3

Operation		Result
< 5, B>	x := x + 2	[x = 2]
< 8, A>	y := y + 2	[y = 2]
<12, A>	y := y + 1	[y = 3]
<14, A>	x := y * 2	[x = 6]

Replica B

Conit

x = 2; y = 5

Operation		Result
< 5, B>	x := x + 2	[x = 2]
<10, B>	y := y + 5	[y = 5]

Vector clock A	= (15, 5)
Order deviation	= 3
Numerical deviation	= (1, 5)

Vector clock B	= (0, 11)
Order deviation	= 2
Numerical deviation	= (3, 6)

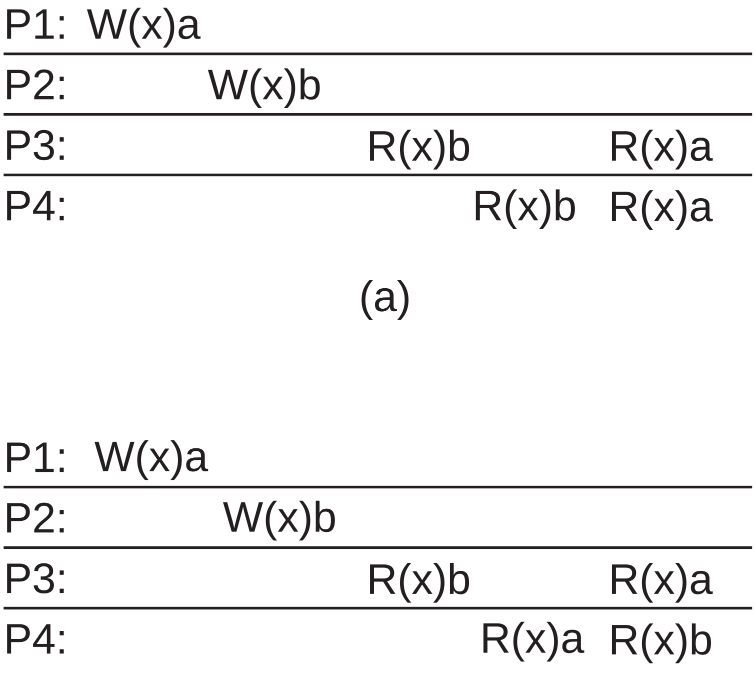
Conit: contains the variables x and y:

- Each replica maintains a **vector clock**
- B sends A operation $[\langle 5, B\rangle\colon\ x := x + 2]$; A has made this operation **permanent** (cannot be rolled back)
- A has three **pending** operations $\Rightarrow$ order deviation = 3
- A has missed **one** operation from B, yielding a max diff of 5 units $\Rightarrow$ $(1,5)$

Sequential Consistency

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.

Note: We’re talking about **interleaved** executions: there is some total ordering for all operations taken together.



```
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

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.

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

(a)

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

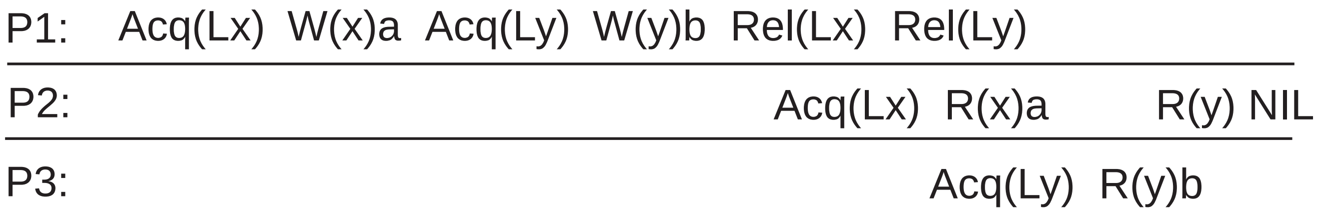
(b)

Grouping Operations (1/2)

- *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.

Grouping Operations (2/2)



P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)

P2: Acq(Lx) R(x)a R(y) NIL

P3: Acq(Ly) R(y)b

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?

Client-Centric Consistency Models

- 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.

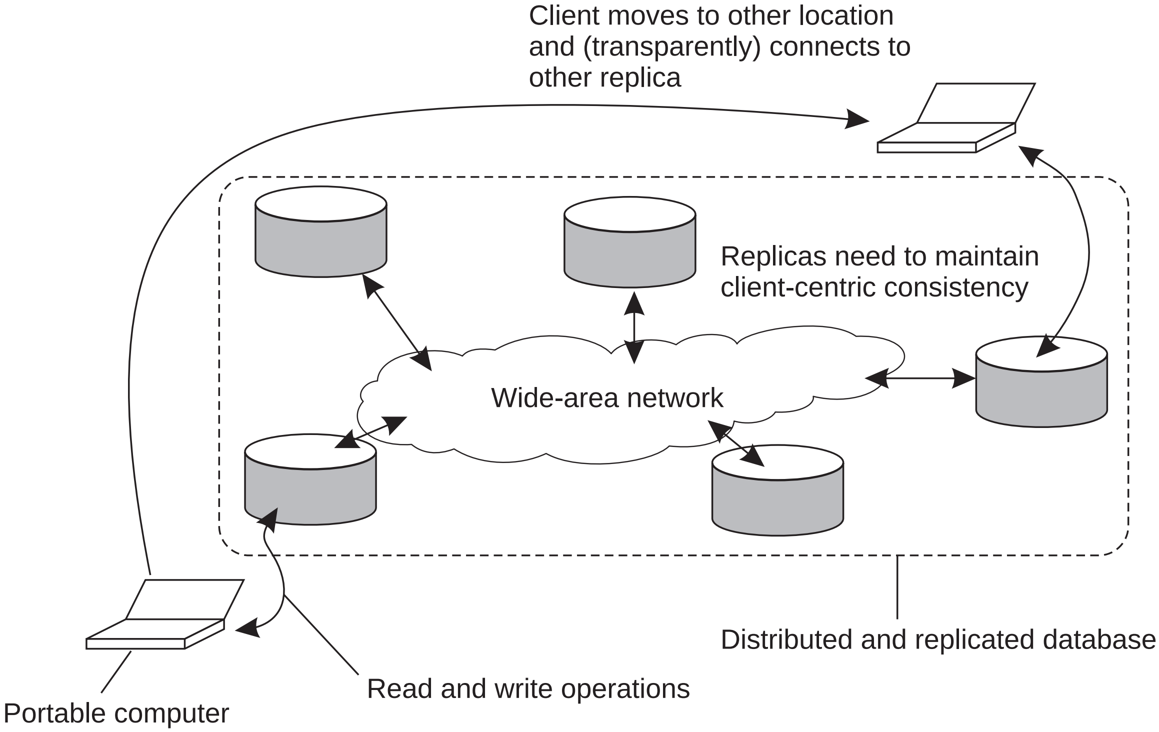
Consistency for Mobile Users

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

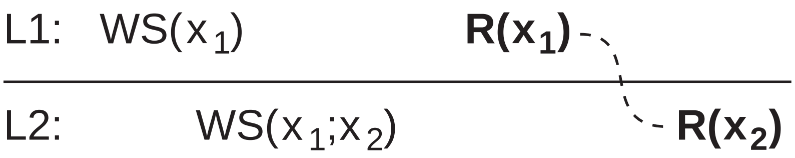
Note: The only thing you really want is that the entries you updated and/or read at A, are in B the way you left them in A. In that case, the database will appear to be consistent to you.

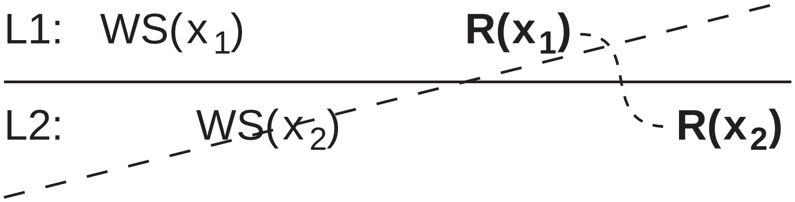
Basic Architecture


Client moves to other location
and (transparently) connects to
other replica
Replicas need to maintain
client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

Monotonic Reads (1/2)

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.





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

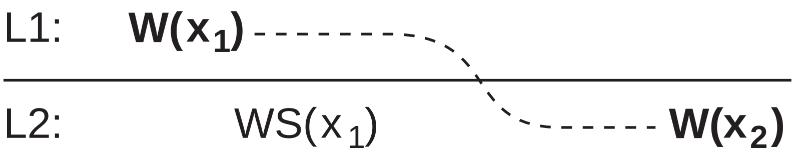
Monotonic Reads (2/2)

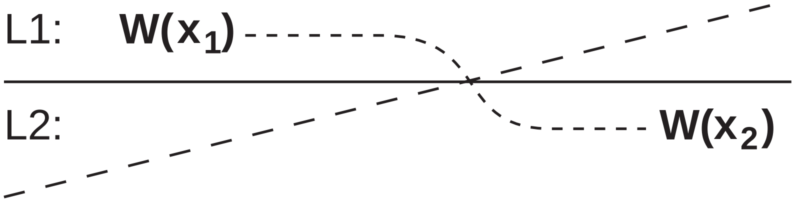
Example: Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

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

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



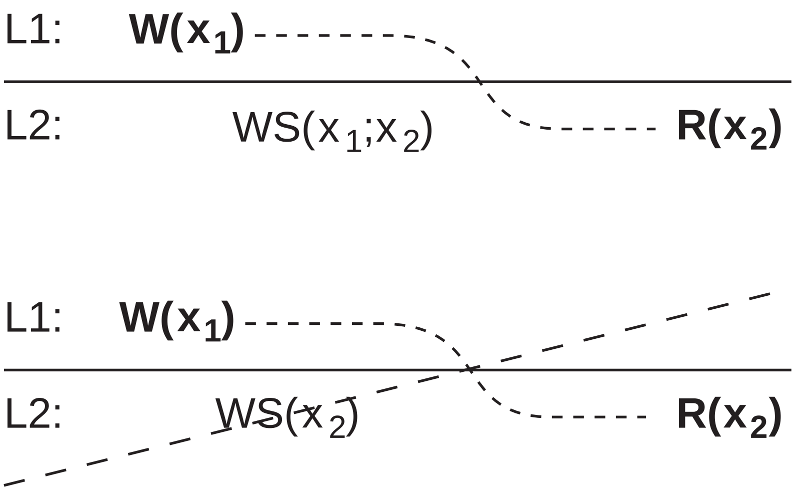


Example: Updating a program at server S_2, and ensuring that all components on which compilation and linking depends, are also placed at S_2.

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

Read Your Writes

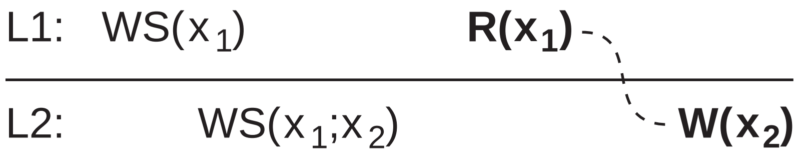
The effect of a write operation by a process on data item x, will always be seen by a successive read operation on x by the same process.



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

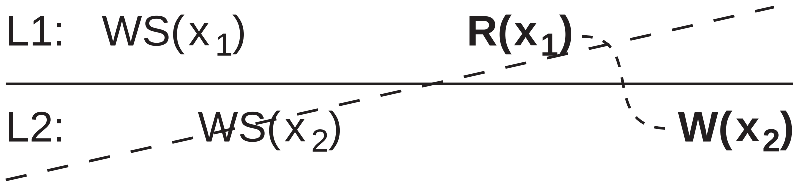
Writes Follow Reads

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



L1: WS(x_1) **R(x_1)**

L2: WS(x_1;x_2) **W(x_2)**



L1: WS(x_1) **R(x_1)**

L2: WS(x_2) **W(x_2)**

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

Distribution Protocols

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

Replica Placement

Essence: 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.
- 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.

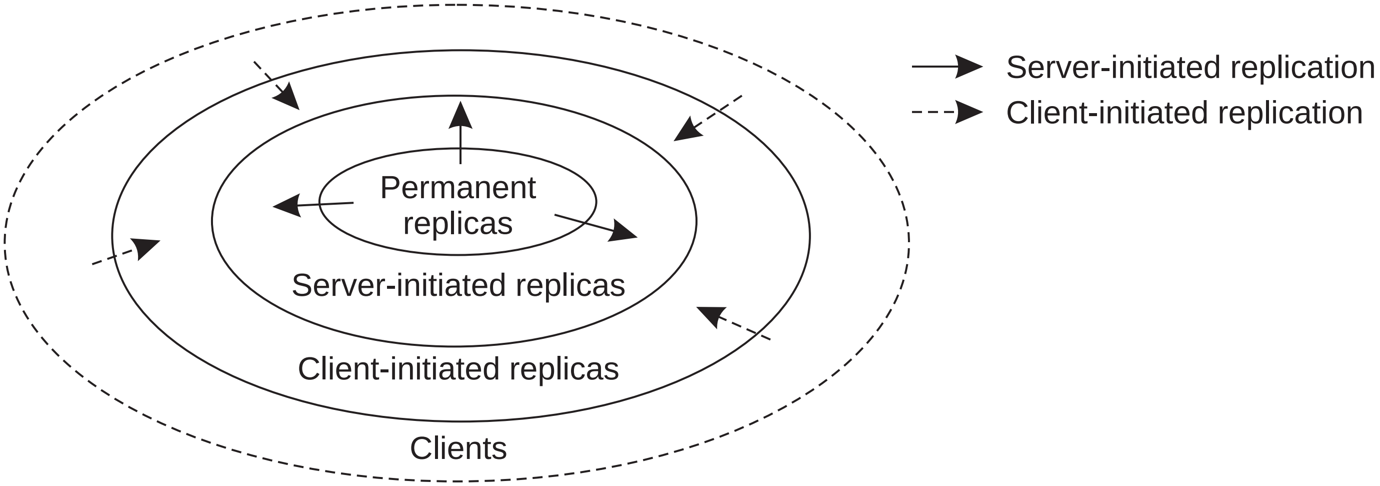
Content Replication (1/2)

Model: We consider objects (and don't worry whether they contain just data or code, or both)

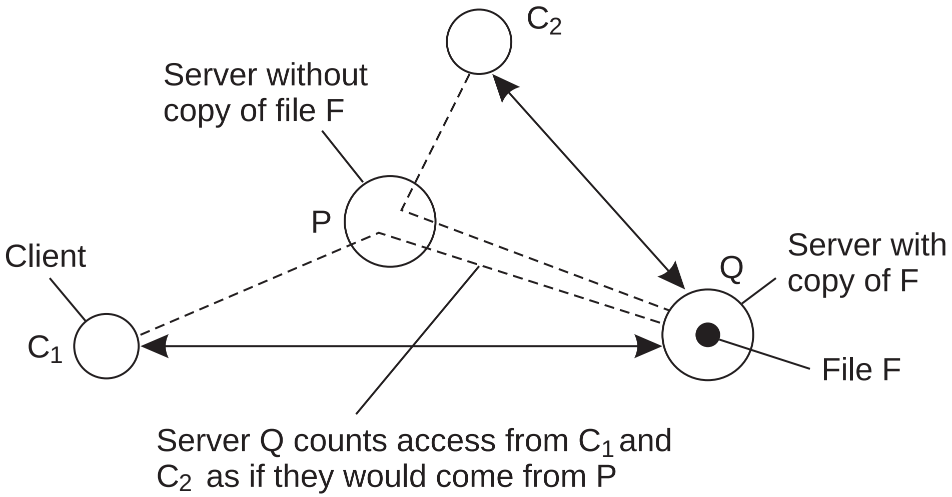
Distinguish different processes: A process is capable of hosting a replica of an object or data:

- **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)

Content Replication (2/2)


Server-initiated replication
Client-initiated replication
Permanent replicas
Server-initiated replicas
Client-initiated replicas
Clients

Server-Initiated Replicas


C2
Server without copy of file F
P
Client
C1
Q
Server with copy of F
File F
Server Q counts access from C1 and C2 as if they would come from P

- 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 (1/3)

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)

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

Content Distribution (2/3)

- 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 (3/3)

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.

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

Question: Why are we doing all this?

Consistency Protocols

Consistency protocol: describes the implementation of a specific consistency model.

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

Continuous Consistency: Numerical Errors (1/2)

Principle: 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 $origin(W)$. $TW[i,j]$ are the writes executed by server S_i that originated from S_j:

$$TW[i,j] = \sum\{weight(W) | origin(W) = S_j \text{ \& } W \in log(S_i)\}$$

Note: 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:

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

Continuous Consistency: Numerical Errors (2/2)

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

Approach: Let every server S_k maintain a **view** $TW_k[i,j]$ of what it believes is the value of $TW[i,j]$. This information can be gossiped when an update is propagated.

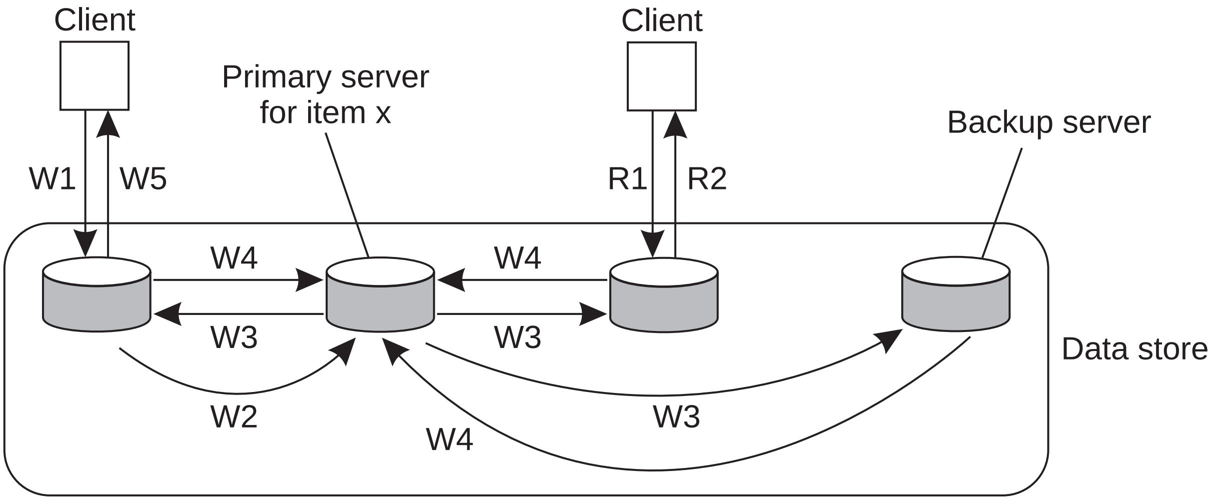
Note: $0 \leq TW_k[i,j] \leq TW[i,j] \leq TW[j,j]$

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

Note: Staleness can be done analogously, by essentially keeping track of what has been seen last from S_i (see book).

Primary-Based Protocols (1/2)

Primary-backup protocol:



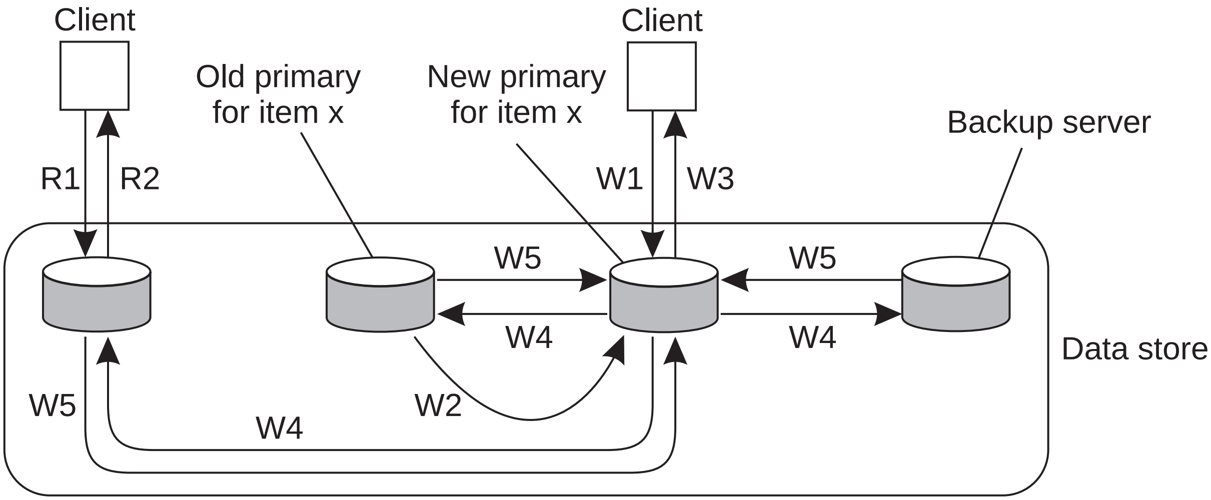
W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Example: 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 (2/2)

Primary-backup protocol with local writes:



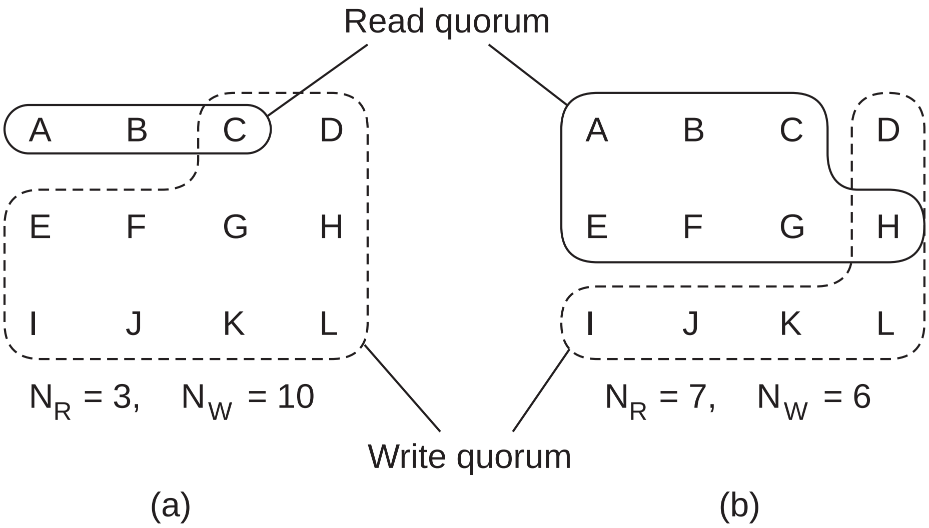
W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

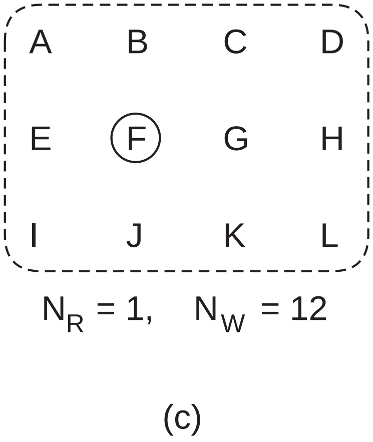
R1. Read request
R2. Response to read

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

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**:


Read quorum
A B C D
E F G H
I J K L
$N_R = 3, \quad N_W = 10$
A B C D
E F G H
I J K L
$N_R = 7, \quad N_W = 6$
Write quorum
(a)
(b)


A B C D
E F G H
I J K L
$N_R = 1, \quad N_W = 12$
(c)