M120: DISTRIBUTED SYSTEMS

Consistency and Replication

*Slides are variant of slides provided by Ken Birman, Andrew S. Tanenbaum, and Maarten Van Steen

Replication

- Maintenance of copies at multiple sites
  - Enhanced reliability
    - Switch to working on other copy of a file when one replica crashes
    - Protect against corrupted data (quorum)
    - Replication of functionality – when one component fails, another takes up its job
  - Enhanced performance
    - When DS needs to scale in numbers
    - When DS needs to scale in geographical area

Replication

- Comes at a cost
  - Multiple copies lead to consistency problem
  - When one copy altered, others must be updated (uses network bw)
- How do we keep replicas consistent ?
  - Want synchronous replication? → <u>atomic</u> updates
    - <u>Requires global synchronization</u>: Replicas reach agreement on when an update is to be performed locally
    - Very costly on wide-area network

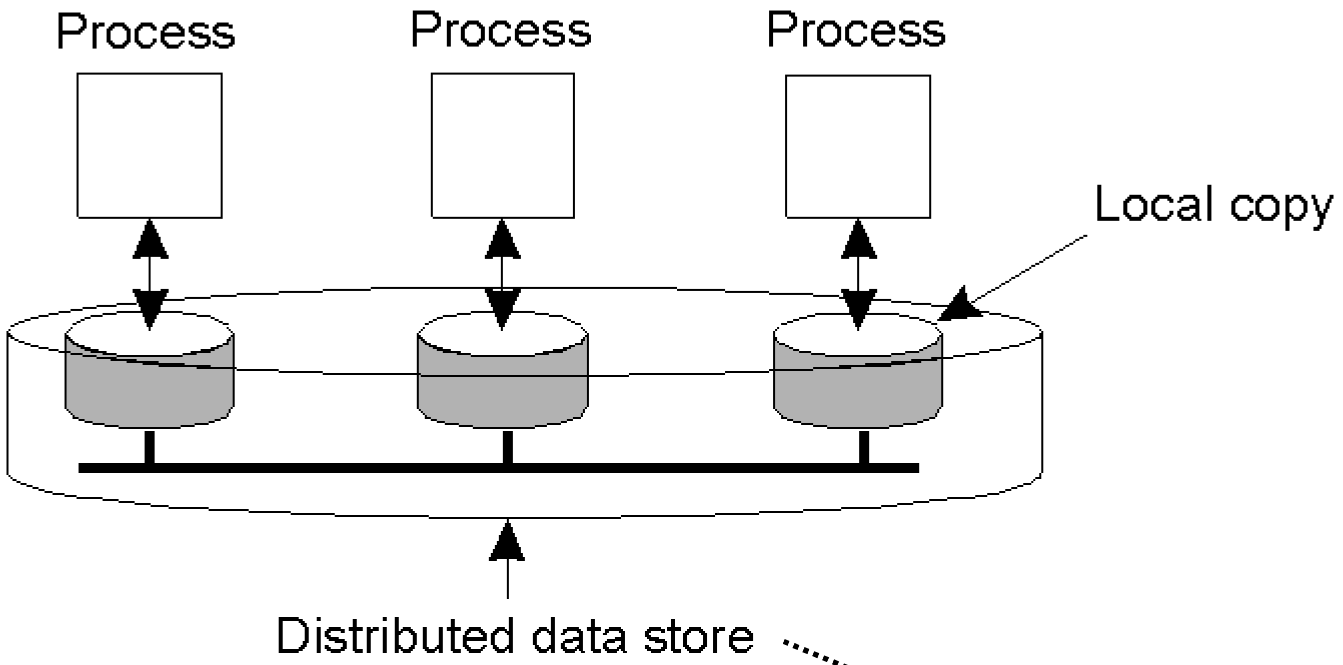
Should we bother to keep replicas consistent?

- Often improves application and thus, user experience
  - Example: Browser caching of web pages
    - Pages may get stale
    - Possible solutions?
  - Example: Disconnected operation by mobile nodes
    - Stale data & resolution of conflicting updates

In many cases, for improved user experience, we need to loosen consistency constraints
- depending on access & update pattern
- depending on location of replicas
- depending on intended usage

Data-Centric Consistency Models

- Deal with the general organization of a logical data store, physically distributed & replicated across multiple processes.



DSM, DFS, DDBMS,…

Consistency Model

- Contract between data store & processes
  - Rules for processes to obey
  - Data store promises to work correctly
- Strict consistency
  - Read is expected to return the value resulting from the most recent write operation
  - … assumes absolute global time which is impossible to achieve in distributed system
  - All writes are instantaneously visible to all
- Sequential consistency
  - All processes see the same interleaving of operations
- Relaxed consistency models

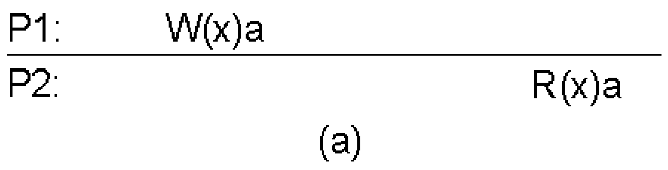
--A consistency model effectively restricts the values that a read operation can return …
--and dictates ordering of updates made at replicas

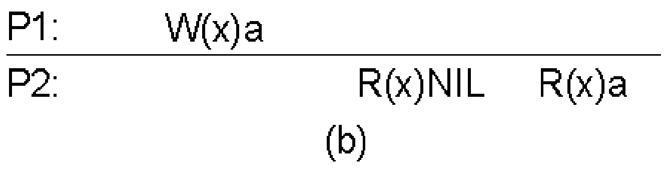
Strict Consistency (I)

- Natural & obvious definition
  - … assuming that the determination of "most recent" is unambiguous
- Suppose that x is stored only on host B
- At time t1, a process on host A "reads" x
  - … thereby sending a message to host B
- At time t2 > t1, a process on B "writes" x
- Strict consistency → The process on host A must obtain the previous value of x
  - … *regardless of the interval (t2 – t1)*
  - What happens if propagation of read request from A to B takes longer than t2 – t1?

Strict Consistency (II)

Impossible to implement in a distributed system !
(assumes all writes are instantaneously visible to all processes)





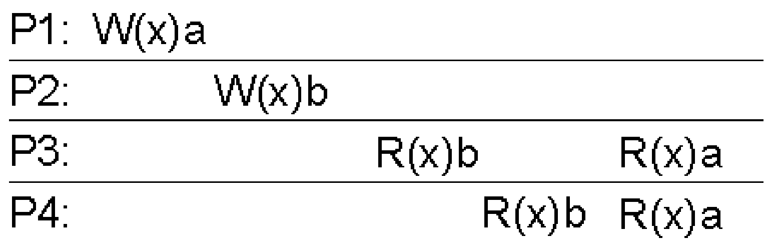
Behavior of two processes, operating on the same data item.

(a) A strictly consistent data store.

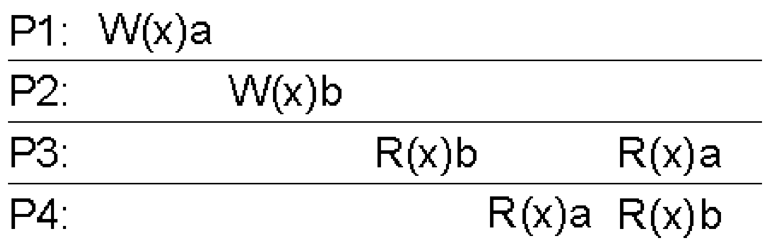
(b) A data store that is not strictly consistent.

Sequential Consistency (I)

All processes agree on the same interleaving of write operations



(a)



(b)

Time does not play a role!

a) A sequentially consistent data store.

b) A data store that is not sequentially consistent.

Sequential Consistency (II)

Process P1	Process P2	Process P3

x = 1;
print (y, z);
	y = 1;
	print (x, z);
		z = 1;
		print (x, y);

- Three concurrently executing processes.

… 90 valid execution sequences (that do not violate program order)

The processes must accept <u>all</u> of them as valid

Sequential Consistency (III)

- Four valid execution sequences for the processes of the previous slide … each yielding a <u>different</u> result.

x = 1;
print (y, z);
y = 1;
print (x, z);
z = 1;
print (x, y);
 	
Prints: 001011
 	
(a)
	x = 1;
	y = 1;
	print (x,z);
	print(y, z);
	z = 1;
	print (x, y);
	
	Prints: 101011
	
	(b)
		y = 1;
		z = 1;
		print (x, y);
		print (x, z);
		x = 1;
		print (y, z);
		
		Prints: 010111
		
		(c)
			y = 1;
			x = 1;
			z = 1;
			print (x, z);
			print (y, z);
			print (x, y);
			
			Prints: 111111
			
			(d)

Summary: 1) all processes must see all writes to all data items in the SAME order
2) individual program order must be respected

Causal Consistency (I)

- Necessary condition:
  Writes that are potentially causally related must be seen by all processes in the same order.
  - If event b is potentially caused or influenced by an earlier event a, then all processes must first see a, then see b
  - Concurrent writes may be seen in a different order on different machines.
- Example: P1 writes x. P2 reads x and writes y.
  - Reading of x and writing of y potentially causally related
- Example: P1 and P2 simultaneously write to two different items
  - Writes are not causally related
  - Operations not causally related are **concurrent**

Causal Consistency (II)

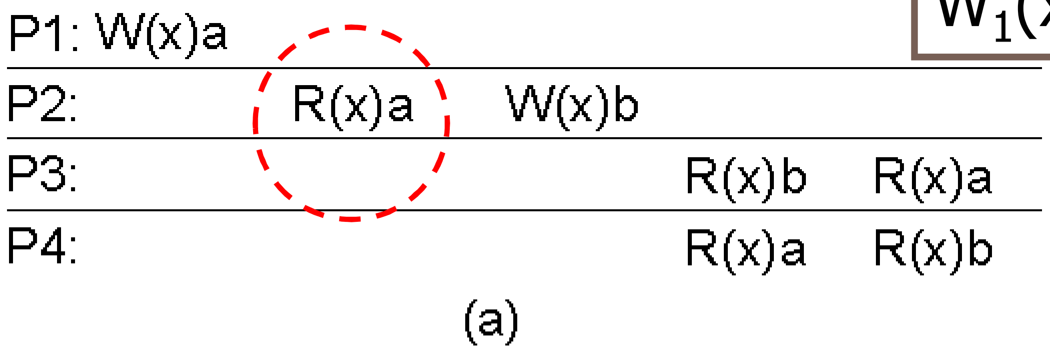
P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

$\boxed{W_2(x)b \ \& \ W_1(x)c \text{ are concurrent events}}$

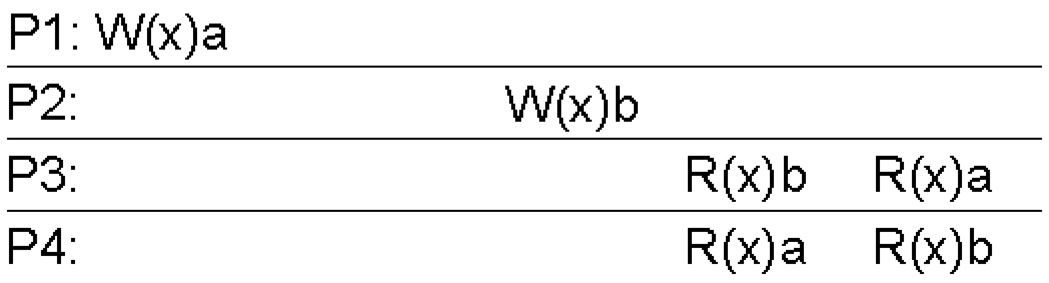
- A sequence allowed with a causally-consistent store
  - … but **not** with sequentially or strictly consistent store.
- Causal consistency requires that all processes see $W_1(x)a$ before $W_2(x)b$

Causal Consistency (III)

$W_1(x)a \rightarrow W_2(x)b$



(a)



(b)

a) violation of a casually-consistent store.

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

Implementation by tracking dependencies → vector timestamps (aka vector clocks)

Grouping Operations

- Sequential and causal consistency defined at level of reads and writes
  - Initially developed for shared memory MPs
  - Implemented at hw level
- Applications often work at different level of granularity
  - Use synchronization mechanisms for mutual exclusion and transactions
  - Reads and writes grouped together and bracketed by enter_CS & leave_CS ops
  - Process that enters critical section will be ensured that data in its local store is up-to-date, then can issue series of reads and writes freely

Synchronization Variables

- Each sync variable has an owner
  - The process that last acquired it
  - Owner can enter and exit CS without sending msgs to other processes
- A non-owner who wants to acquire the sync variable must send msg to current owner
  - Ask for ownership
  - Ask for current values of data associated with the variable
- Multiple processes can own simultaneously a sync variable in non-exclusive mode

Entry Consistency (I)

- An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
  - *All remote changes to the guarded data must be made visible*
- Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
- If another process wants to enter CS in nonexclusive mode, must first check with owner of the sync variable guarding the CS to fetch most recent copies of the guarded data from the owner of the sync variable.

Entry Consistency (II)

Each shared data item needs to be associated with a sync. variable (lock)

Can be done implicitly (by the run-time system)

Current owner per sync. variable

Only data guarded by a lock are kept consistent

Acquire makes visible all remote changes to the guarded data

```
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
```

- A valid event sequence for entry consistency.

Summary of Data-Centric Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

Consistency versus Coherence

- Consistency model describes what can be expected w.r.t. a set of data items when multiple processes concurrently operate on that set
  - Data set is consistent if it adheres to the model’s rules
- Coherence models describe what can be expected w.r.t. a single data item
  - Data item is coherent when its various copies abide by the coherence model’s rules

Read-mostly Data Stores

- Assume a data store
  - Where concurrent conflicting updates are rare
  - And easy to resolve
  - Most operations are reads
  - Can maximize performance by following a very weak consistency model
    - Eventual consistency

Examples of Read-mostly Data Stores

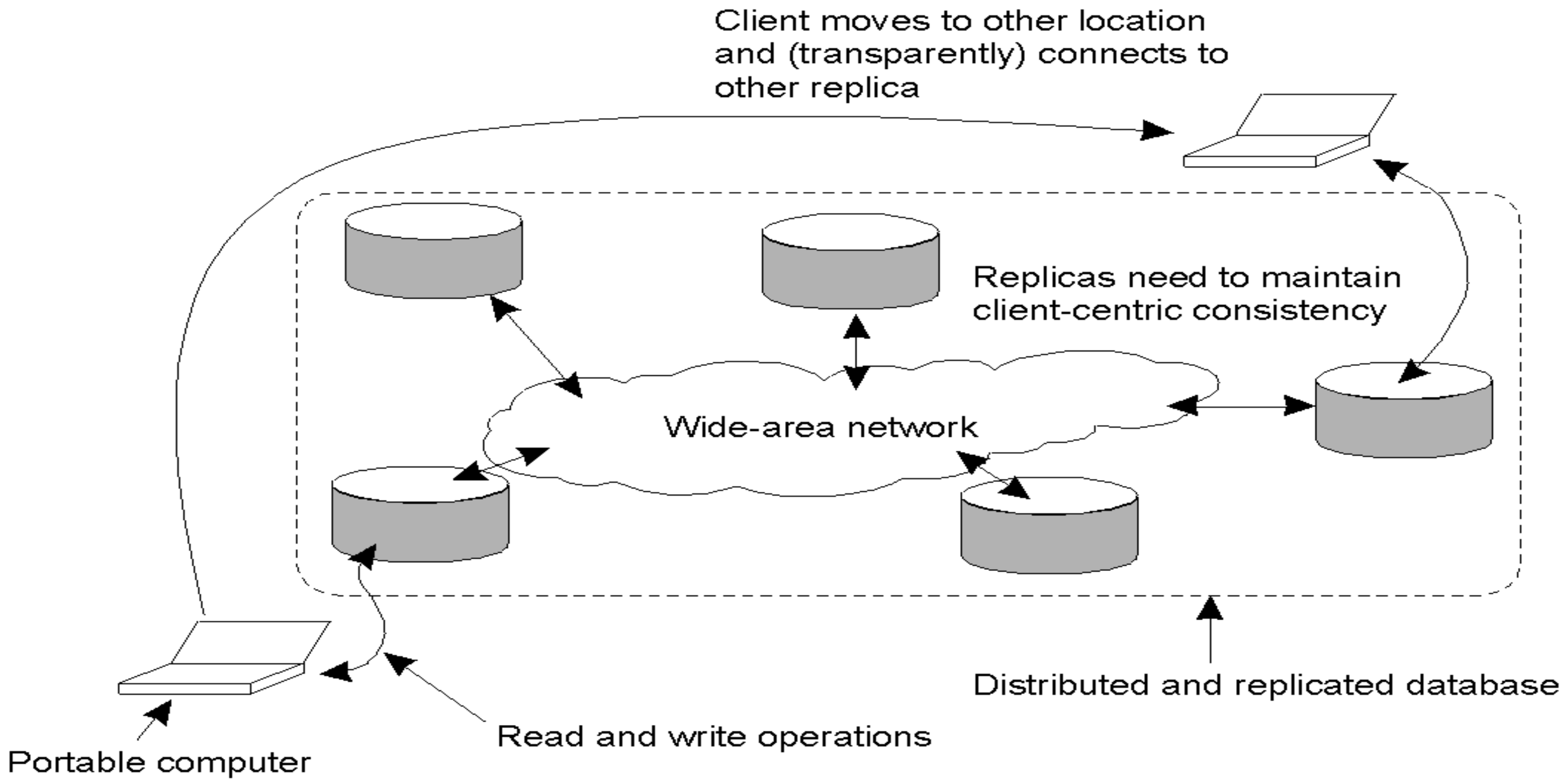
- Databases
  - Most processes rarely perform updates, mostly read from the DB
- DNS
  - Single naming authority per zone
  - “lazy” propagation of updates
- WWW
  - No write-write conflicts
  - Usually acceptable to serve slightly out-of-date pages from a cache

For read-mostly data stores: Eventual Consistency

- For data stores that tolerate a high degree of inconsistency
- If no updates take place for long time, all replicas gradually converge to identical copies
  - When few processes perform updates, write-write conflicts easy to resolve
  - Cheap to implement
- What happens when clients don’t always access the same replica?

Client-centric Consistency

- A mobile user accessing different replicas of a distributed database.


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

When users sometimes operate on different replicas, we need client-centric consistency

Client-centric Consistency

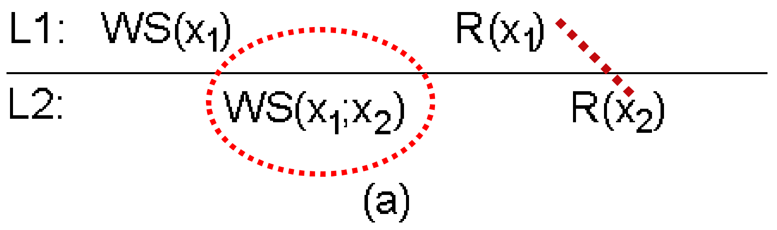
- For eventually-consistent data stores where users sometimes operate on different replicas
- *For a **single client**, consistency of the data items accessed by that client*
- No guarantees given about concurrent accesses by different clients
- Originated from Bayou work (Terry et al., 1994)
  - Weakly connected replicated storage system for a mobile computing environment
  - Four different consistency models

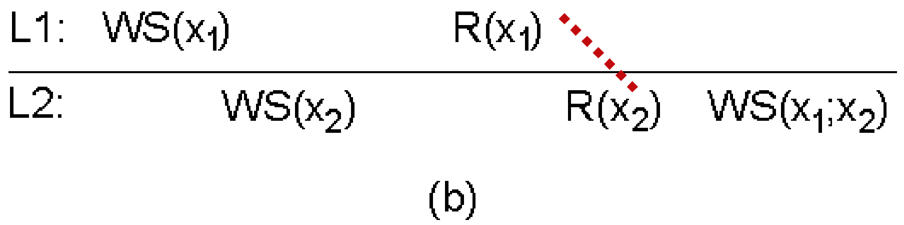
Bayou client-centric models

- $x_i[t]$: version of object x at local copy L_i at time t
  - … result of updates to a series of writes at L_i since system initialization
  - $WS(x_i[t])$: series of writes
  - $WS(x_i[t2]; x_z[t2])$: series of writes that have <u>also</u> been performed at copy $L_z$at a later time
- Assume an "owner" for each data item
  - … *avoid write-write conflicts*
- Monotonic reads
- Monotonic writes
- Read-your-values
- Writes-follow-reads

Monotonic Reads

$WS(x_1)$ is part of $WS(x_2)$





If a process has seen a value of x at time t, it will never see an older value at a later time.

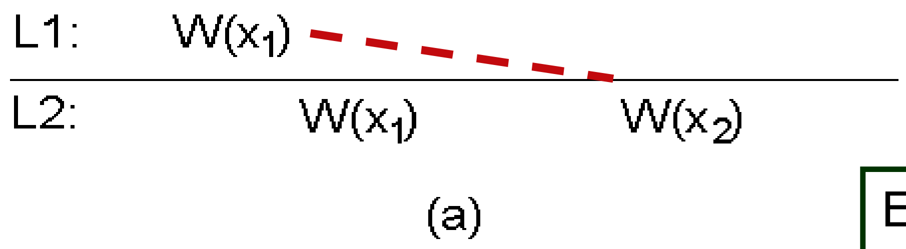
Example:
-replicated mailboxes with on-demand propagation of updates

- 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

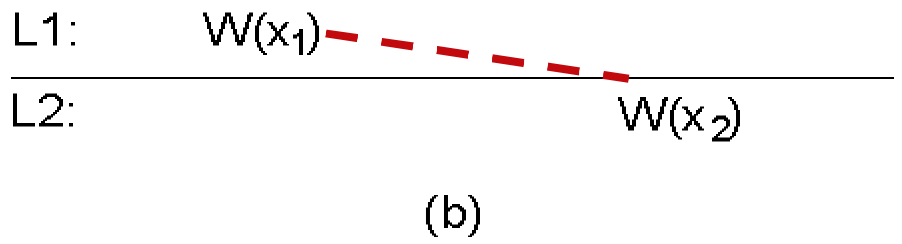
b) A data store that does not provide monotonic reads.

Monotonic Writes



If an update is made to a copy, all preceding updates must have been completed first.

Esp. important when a write may affect only part of the state of a data item



FIFO propagation of updates by each process

Example:
- s/w library

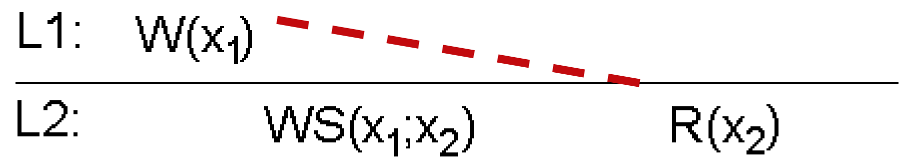
No guarantee that x at L2 has the same value as x at L_1 at the time W(x1) completed

- The write operations performed by a single process *P* at two different local copies of the same data store

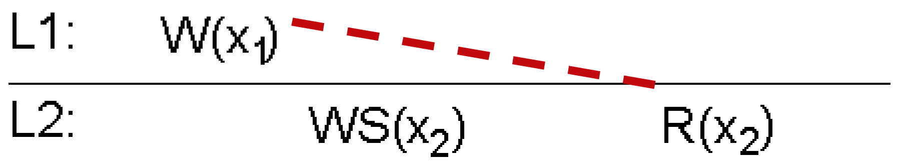
a) A monotonic-write consistent data store.

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

Read Your Writes



(a)



(b)

The effects of the previous write at L1 have not yet been propagated !

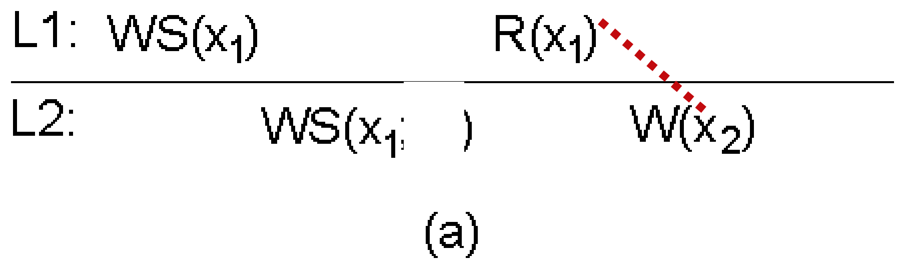
A write is completed before a successive read (by same process), no matter where the read takes place

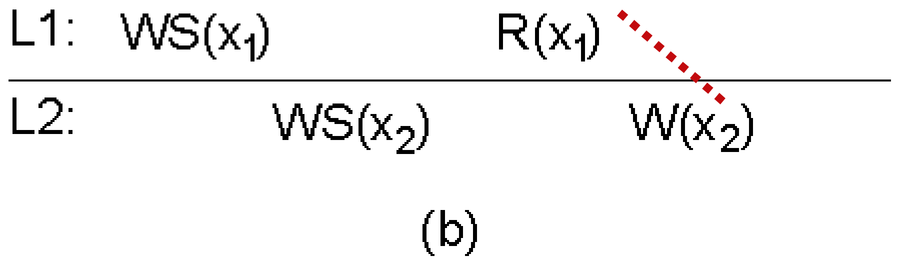
Negative examples:
- updates of Web pages
- changes of web passwords

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

b) A data store that does not.

Writes Follow Reads





Any successive write by a process will be performed on a copy that is up-to-date with the value most recently read by the process.

Example:
- updates of a newsgroup:
Responses are visible only after the original posting has been received

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

Implementing client-centric models (I)

- Globally unique ID per write operation
  - Assigned by the initiating server
- Per-client state:
  - Read set
    - Write IDs relevant to client's read operations
  - Write set
    - IDs of writes performed by client
- Major performance issue:
  - Size of read/write sets ?

Implementing client-centric models (II)

- **Monotonic read**:
  - When a client issues a read, the server is given the client's read set to check whether all the identified writes have taken place locally
    - If not, the server contacts others to ensure that it is brought up-to-date before carrying out the read
  - After the read, the client's read set is updated with the server's "relevant" writes
- **Monotonic write**:
  - When a client issues a write, the server is given the client's write set to
    - … ensure that all specified writes are performed first and in the correct order
    - Then the new write op is performed
  - The write operation's ID is appended to client's write set

Implementing client-centric models (III)

- Read-your-writes:
  - Before serving a read request, the server fetches (from other servers) all writes in the client’s write set
- Writes-follow-reads:
  - Server is brought up-to-date with the writes in the client’s read set
  - After write, the new ID is added to the client’s write set, along with the IDs in the read set
    - … as these have become “relevant” for the write just performed

Implementing client-centric models (IV)

- Read and write sets get large over time
  - Idea: Group a client's read and write operations into sessions
  - A session is typically associated with an application
    - … but may also be associated with an application that can be temporarily shutdown (eg: email agent)
  - What if the client never closes a session ?
- How to efficiently represent read & write sets ?
  - List of IDs for write operations
    - … Not all of these are actually needed !!

Implementing client-centric models (V)

- Idea: Use vector timestamps to improve efficiency:
  - When server S_z accepts a write operation, it assigns to it a globally unique WID and a timestamp ts(WID)
  - Each server z maintains vector WVC(z)
    - WVC(z)[i] := timestamp of the latest write initiated at server S_i that has been received & processed at S_z
    - Server returns its current vector timestamp with its responses to read/write requests
    - Client adjusts the timestamp for its own read/write set

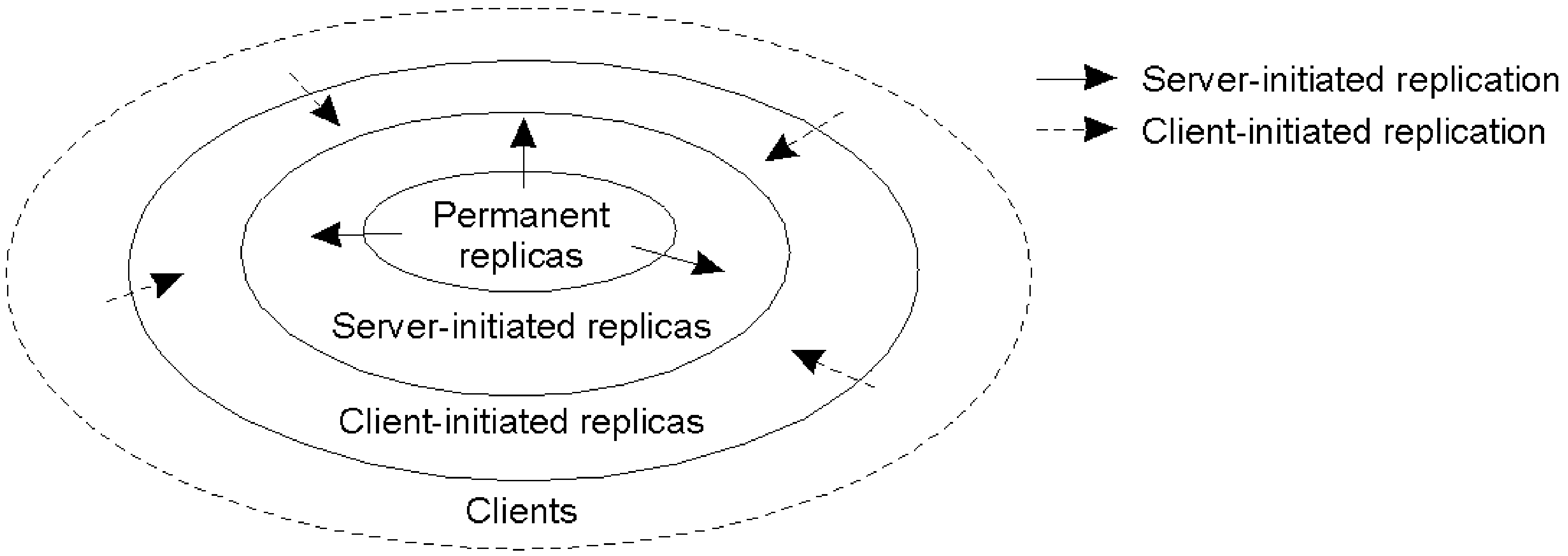
Implementing client-centric models (VI)

- Efficient representation of read/write set A:
  - VT(A): vector timestamp
    - VT(A)[z] := max. timestamp of all operations in A that were initiated at server S_z
  - Union of 2 sets of write IDs:
    - VT(A+B)[z] := max{ VT(A)[z], VT(B)[z] }
  - Efficient way to check if A is contained in B:
    - VT(A)[z] <= VT(B)[z]
    - Thus, we can check if a set A, of writes, has been applied at a server by comparing their vector timestamps rather than checking if individual write IDs of set A appear in the write set of the server

Replica Placement

- Key issue in any DS
  - Where to place replicas
    - Often affected by management/commercial issues (see Akamai example)
    - Client and network properties important factors too
  - Where to place content
    - i.e, on which replica to place a particular data item

Where to place content?


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

- The logical organization of different kinds of copies of a data store into three concentric rings.

Replica Content Placement

- Permanent copies
  - Initial set of replicas that make up the distributed data store

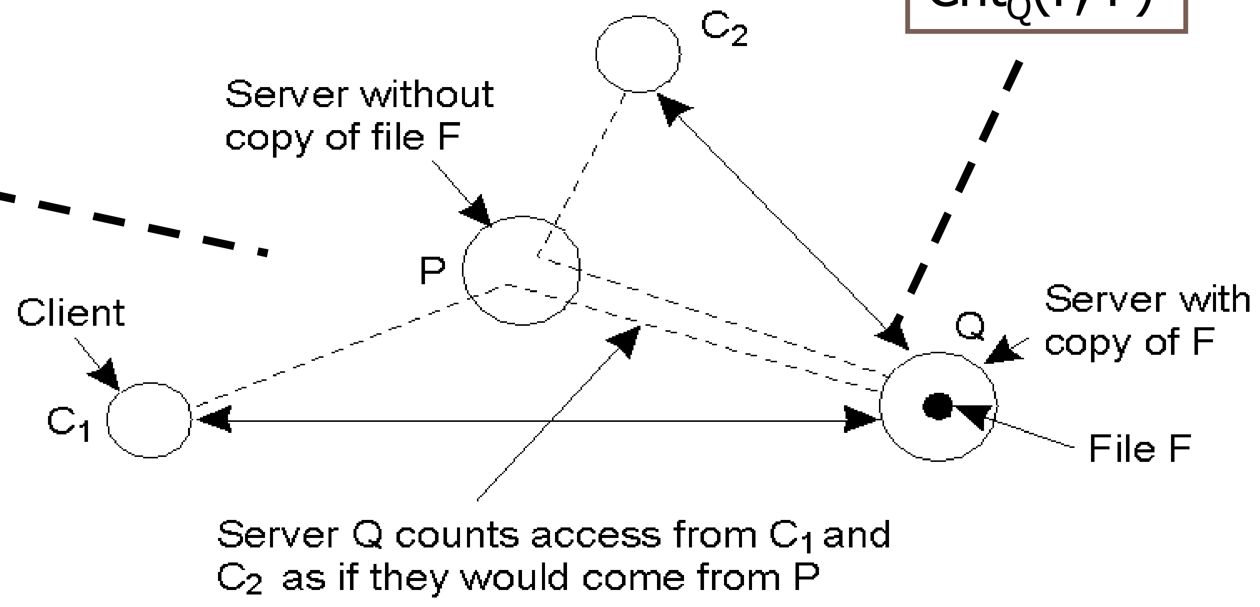
    Example from the Web:
      - Web site replicated on servers within a local cluster
      - Mirror web site on geographically distributed sites
- Server-initiated
  - Dynamic replication to handle bursts
  - Usually done for read-only data
    - Content Distribution Network (CDN)
- Client-initiated
  - Aka (client) caches
  - Improve access time to data
    - Danger of “stale” data
  - Private vs Shared caches

Server-Initiated Replicas (Rabinovich et al, ICDCS 1999)

Counting access requests from different clients.

P := closest server for both C1 & C2

$\mathrm{Cnt}_Q(P, F)$



Server Q counts access from C_1 and C_2 as if they would come from P

<u>At each server</u>:
- •Count of accesses for each file
- •Originating clients

Routing DB to determine "closest" server for client C

- •Deletion threshold: del(S, F)
- •Replication threshold: rep(S, F)

Dynamic decisions to delete/migrate/replicate file F to server S

Extra care to ensure that at least one copy remains !

How to propagate updates to replicas?

- **State vs Operations**
  - Notification of an update
    - Invalidation protocols
    - Best for low read/write ratio (%)
  - Transfer data from one copy to another
    - Transfer of actual data … or delta of changes
    - Batching to save communication overhead
    - Best for relatively high read/write %
  - Propagate the update "params/ops/description" to other copies
    - Active replication
- **Pull vs Push**
  - Push → replicas maintain a high degree of consistency
    - Updates are expected to be of use to many read-only clients
  - Pull → best for low read/write %
  - Hybrid scheme based on lease model
- **Unicast vs Multicast**
  - Push → multicast group
  - Pull → single server or client requests an update

Pull versus Push Protocols

Stateful server: keeps track of all caches

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (or notification of update and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

- Comparison between push-based & pull-based protocols in the case of a single server, multiple client system.

Leases

- A promise by a server that it will push updates for a specified time period
  - After expiration, client has to "pull" for updates
- Alternatives:
  - Age-based leases
    - Depending on the last time an item was modified
      - Long-lasting leases for items that are expected to remain unmodified
  - Renewal frequency-based leases
    - Short-term leases for clients that only occasionally ask for a specific item
    - Effect: server only tracks clients where its data are popular and gives them high consistency
  - Leases based on state-space overhead at the server:
    - Lower expiration time as the server becomes overloaded
    - Effect: needs to track fewer clients since leases expire more quickly

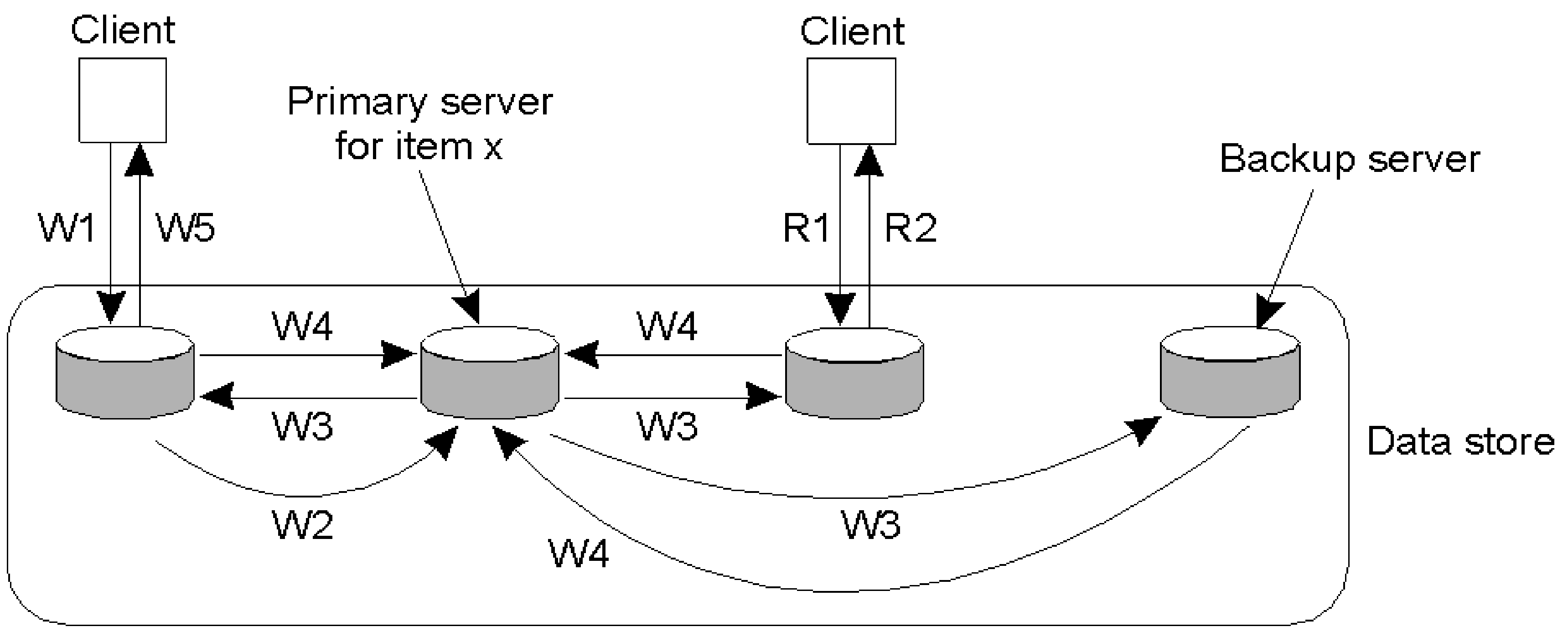
Back to implementation of consistency models

(i.e., consistency protocols)….

Primary-based protocols

- Distributed app developers tend to use consistency models that are easy to understand
  - More complex models with better performance often ignored
- For models that handle consistent ordering of operations
  - Sequential consistency very popular
  - Primary-based (sequential consistency) protocols prevail
    - Each data item has associated primary in charge of coordinating writes to that item
    - Remote-write vs local-write protocols
  - Replicated-write (sequential consistency) protocols also widely used

Remote-Write (Primary-based) Protocols



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

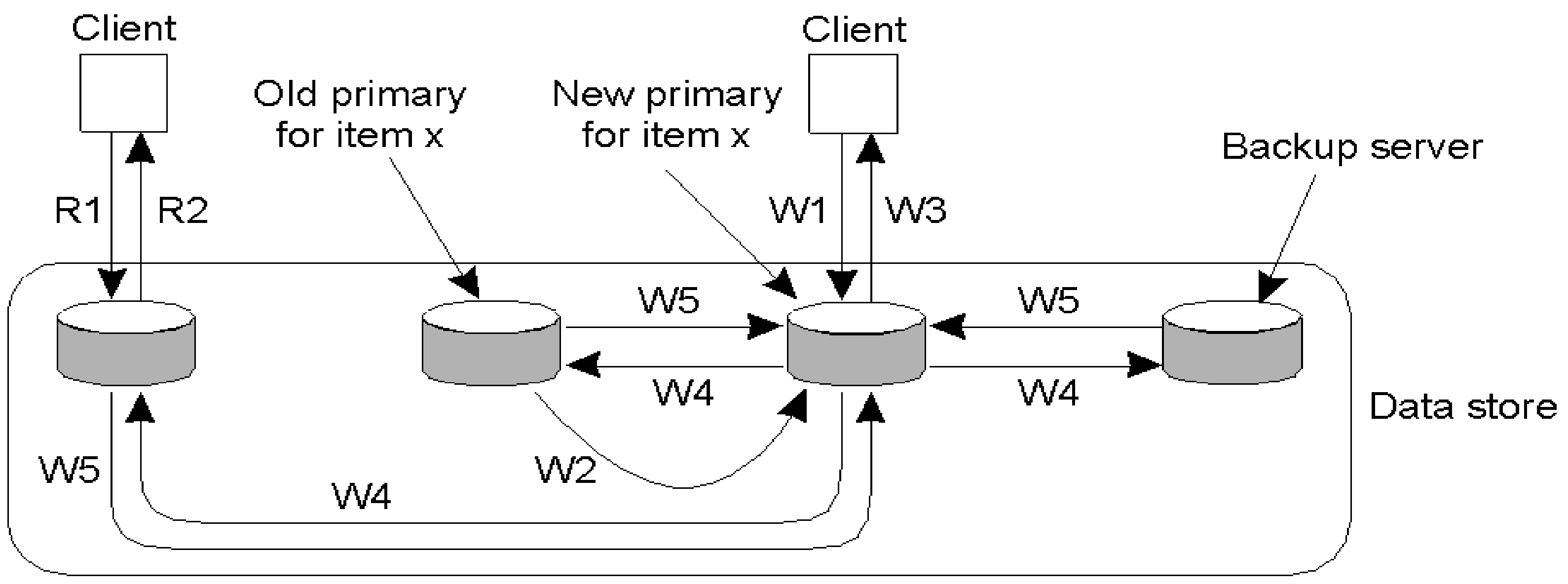
R1. Read request
R2. Response to read

- Also called primary-backup protocol
- All write ops for item x forwarded to a fixed server; read ops can be done locally

Blocking vs Non-blocking updates

- Blocking updates
  - … straightforward implementation of sequential consistency
    - The primary orders all updates
    - Processes see the effects of their most recent write
    - But, response time is delayed
- Non-blocking updates
  - … reduce blocking delay for the process that initiated the update
    - The process only waits until the primary's ACK
  - But, fault tolerance not ensured

Local-Write (Primary-based) Protocols



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

Suitable for disconnected operation

- Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

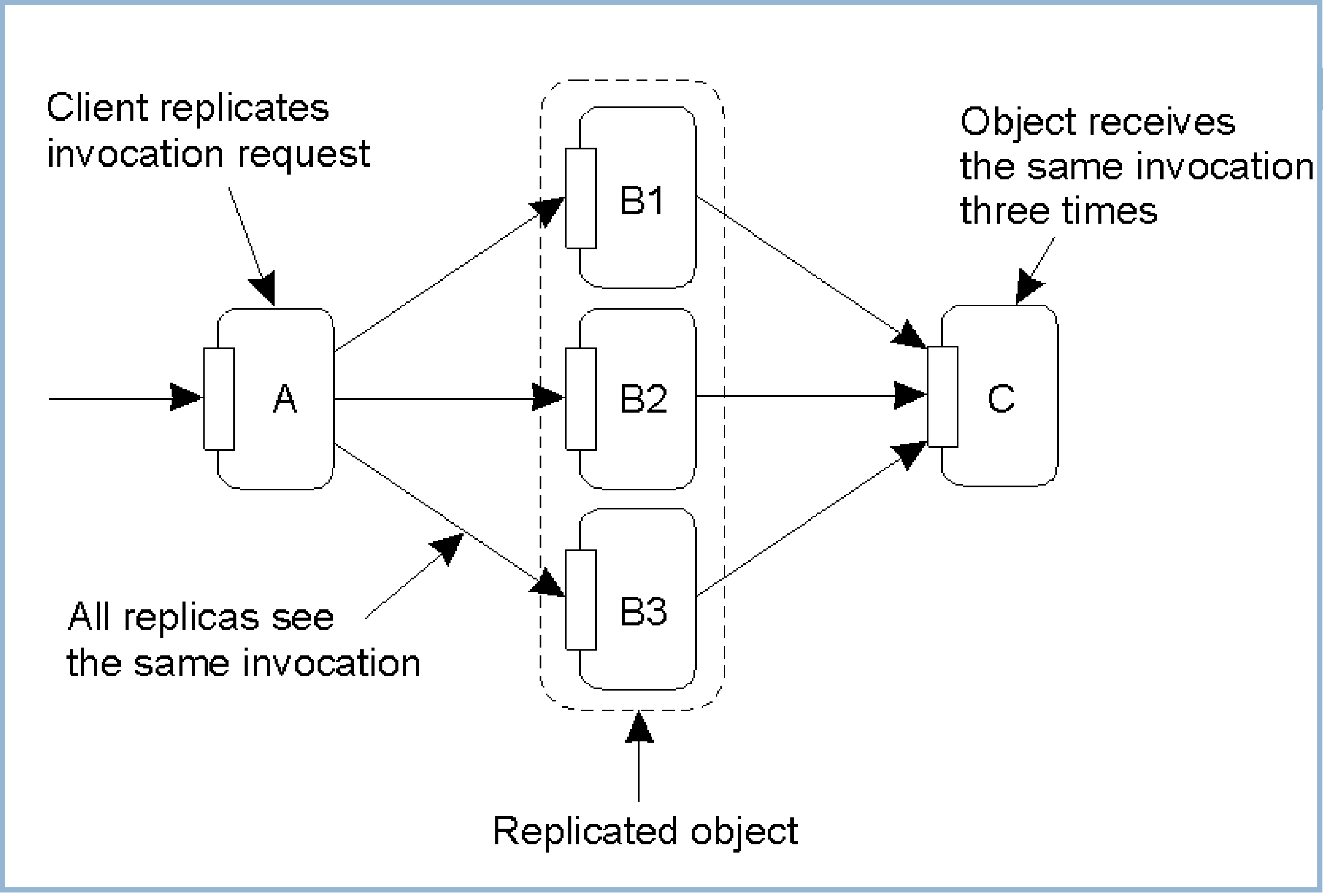
Replicated-Write Protocols

- Write operations can be carried out at multiple replicas instead of one
- Active replication
  - Write operation forwarded to all replicas
- Majority voting
  - Require clients to acquire permission from multiple servers before reading/writing data item

Active replication (I)

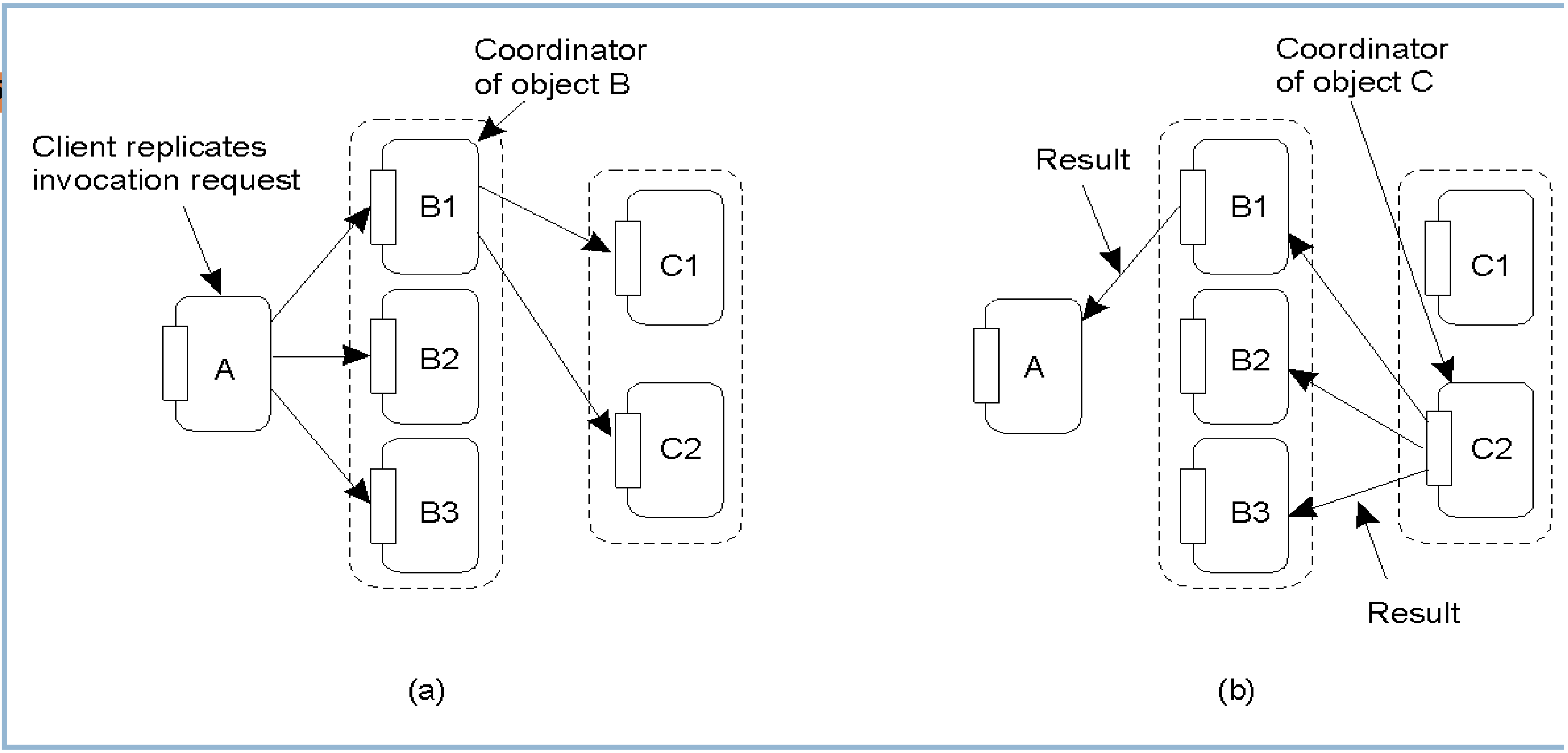
- Each replica has a process that carries out write operations
- Write operations from clients propagated to all replicas
- Write operations must be done in same order everywhere
  - Need totally-ordered multicast mechanism (e.g., Lamport's logical clocks), **not scalable**
  - OR, use **central coordinator** that orders operations and assigns unique sequence number

Active Replication (II)


Client replicates invocation request
Object receives the same invocation three times
A
B1
B2
B3
C
All replicas see the same invocation
Replicated object

- The problem of replicated invocations.

Active Replication (III)



(a) Forwarding an invocation request from a replicated object.

(b) Returning a reply to a replicated object (from a replicated object).

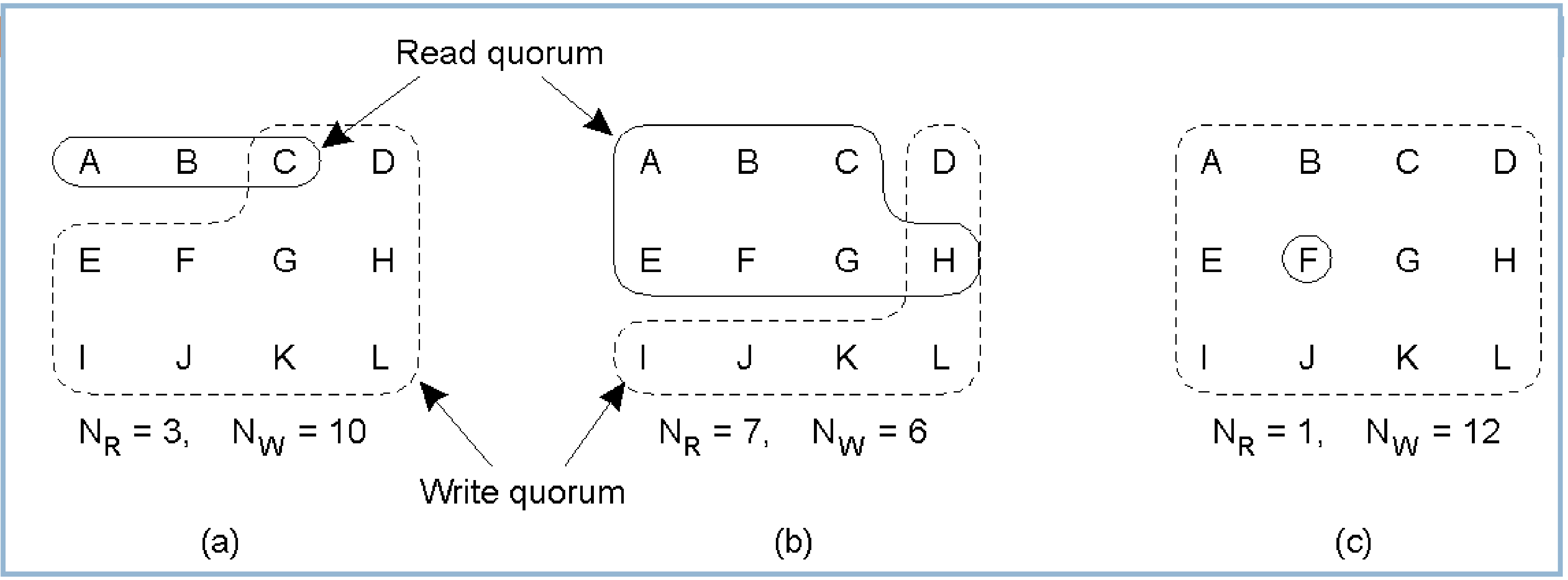
Quorum-Based Protocols

- Require clients to get permission from multiple servers before either reading or writing to a replicated data item
- Example: DFS with file replicated on N servers
  - To write, client must find **at least N/2 + 1 (majority)** servers to agree, servers write, file gets new version number
  - To read, client must find **at least N/2 + 1 (majority)** of servers and ask for version numbers, if number are same, this is most recent version of the file

Gifford's Quorum Scheme (1979)

- Version numbers or timestamps per copy
- Obtain quorum before read/write:
  - R votes before read
    - To read a file, client must find quorum of R or more servers with same version number
  - W votes before write
  - **W > N/2 → prevents write-write conflicts**
  - **(R + W) > N → prevents read-write conflicts**
- Any quorum pair must contain common copies
  - In case of partition, it is not possible to perform conflicting operations on the same copy

Gifford's Quorum Scheme



<u>Three examples of the voting algorithm</u>:

a) A correct choice of read & write set

b) A choice that may lead to write-write conflicts

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