Distributed Systems

7. Replication and Consistency Models

Moving Data – What?

- Migration
  - A data block is moved from one place to the other
  - Advantage: enhanced access can be provided
  - Disadvantage: *Block-bouncing*
    - Similar to thrashing
    - Two (or more) processors play ping-pong with a block
    - Can be avoided by replication
- Replication
  - A *copy* of data is moved form one place to the other
  - Replication is
    - More general
    - More important
    - More difficult

Moving Data – Who, When, How?

- *Who* initiates the data movement?
  - The owner processor: *Push*
    - E.g. replicated file server
  - The user processor: *Pull*
    - E.g. POP3-Mail client
- *When* is data movement initiated?
  - On demand: *Reactive*
    - E.g. Web cache
  - In advance: *Proactive*
    - E.g. replicated database server
- How is consistency guaranteed?
  1. Data-Centric Consistency Models
  2. Client-Centric Consistency Models

Replication

- Reliability
  - Several copies of data make the loss of one harmless
- Performance and Scalability
  - A single server could become a bottleneck; Several copies make the system *scalable*
  - Remote data can be placed near to the client
- Disadvantages
  - The content of instances of the same data must be kept identical: data must remain consistent
  - The price for consistency management may be higher than the gain: If more write than read access
  - Consistency management may destroy the enhanced scalability features

1. Data-Centric Consistency Models (1)

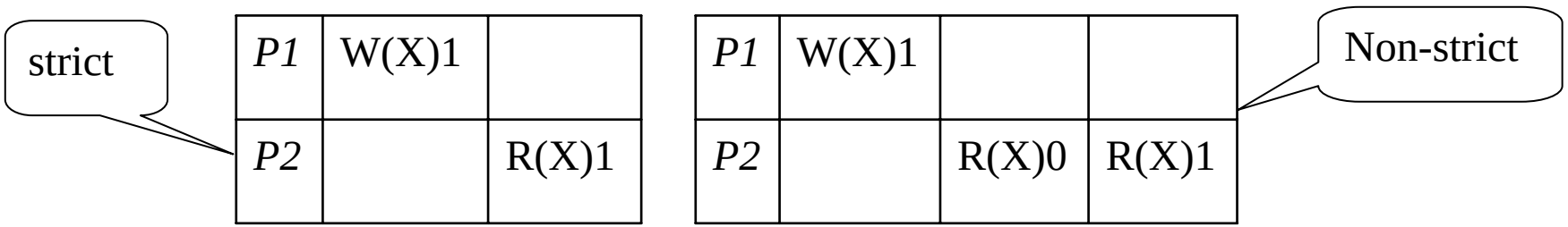
- Relevant are: *competing* accesses
- A read operation may return a stale value
- Write operations maybe observed in different order
- *Coherence* is the most strict requirement for *consistency*
  - Coherence is related to data items in isolation
  - Consistency is related also to different data items
  - Often used just as synonyms
- Ideal definition of coherence
  - A read always returns the value produced by the *latest* write
  - Which is the latest? – no global time
- We have to choose a consistency model
  - The programmers must know and accept it
- Concurrency vs. ease of programming
  - No concurrency at all: no programming difficulty
  - Full concurrency: all difficulties at the programmer

Data-Centric Consistency Models (2)

- Models grouped in two categories
  1. General access models (based on general read/write accesses)
  2. Synchronization Access Consistency (based on synch/acquire/release accesses)
- Synchronization access
  - synch(S)
  - acquire(S); release(S)
- Notation
  - R(X)v: A *read* operation on *X* results in value *v*
  - W(X)v: A *write* operation assigns the value *v* to *X*
  - Pi: Process(or) i
  - Variables are initialized to 0

1.1. General Access Consistency Models

- *Atomic* (strict) Consistency
- All reads and writes appear as they were executed atomically and sequentially
  - Like a centralized shared memory system
  - E.g. X:= 1; X:= 2; print(X); outputs always 2
- Cannot be implemented reasonably in a distr. sys.
  - We ought to serialize every resource access
  - The propagation of values between nodes takes time (e.g. the assignment X:= 2 could happen 1 ns later)



Sequential Consistency - Lamport (1)

1. The result of any execution is the same as if the operations of all the processors were executed in some sequential order and
2. The operations of any process are observed in the order, as stated in the program code

- Operations may intertwine, in the same way for all
- Costs are high and have a theoretical limit.
- If
  - Read-time=r, Write-time=w, Min-pack.-transfer-time=t
  - $r + w \geq t$ always holds
  - Faster reads make writes slower and the way around

Sequential Consistency - Lamport (2)

sequential

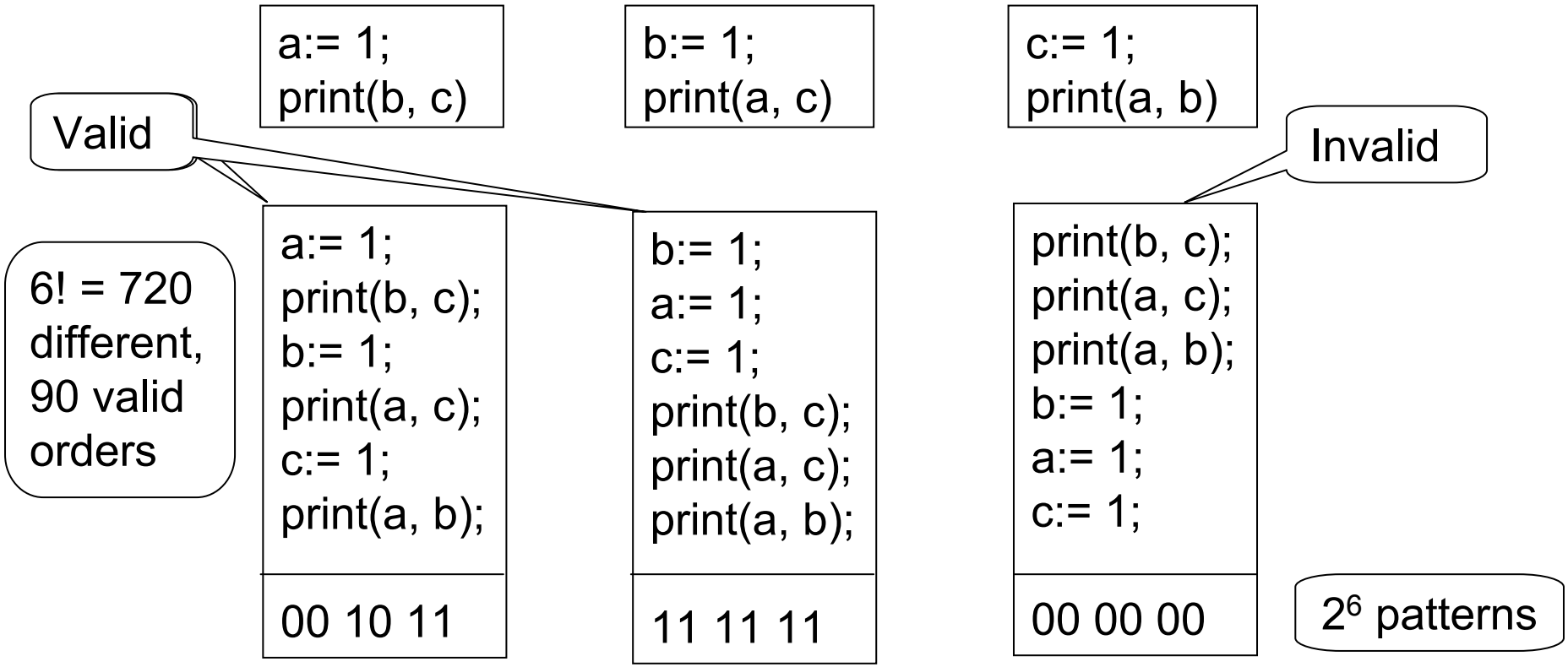
P1	W(X)1					
P2		W(X)2				
P3			R(X)2			R(X)1
P4				R(X)2	R(X)1	

P1	W(X)1					
P2		W(X)2				
P3			R(X)2			R(X)1
P4				R(X)1	R(X)2	

Non-sequential

Sequential Consistency - Lamport (3)

- Sequential Consistency is similar to the serialization problem in transactions
  - However, with much finer, single read/write level *granularity*



Causal Consistency

- Relaxation of sequential consistency
  - Only causally related writes must be observed in the same order
- This allows disjoint writes to be overlapped
- Example (causally consistent, but not sequentially)
  - W(X)1 and W(X)2 are causally related
    - P2 read X before writing it, it may rely on its previous value
  - W(X)2 and W(X)3 are causally not related
    - They can be seen by P3 and P3 in different order

P1	W(X)1			W(X)3		
P2		R(X) 1	W(X)2			
P3					R(X)3	R(X)2
P4					R(X)2	R(X)3

Processor (FIFO) Consistency

- Only writes from the same processor are observed in the same order as issued
- Easy to implement, difficult to use
  - See client-centric consistency models
- Example (processor consistent, but not causally)

P1	W(X)1						
P2		R(X)1	W(X)2	W(X)3			
P3					R(X)2	RX(1)	RX(3)
P4					R(X)1	RX(2)	RX(3)

Slow Memory Consistency

- Only writes to the same location issued by the same processor are observed in the same order
- Writes immediately visible locally and propagated slowly
- A write operation is actually a non-blocking send
- Must be used carefully, efficient even with slow connections
- Example
  - Slow memory consistent, but not processor consistent
  - W(Y)2 can be issued immediately after W(X)1
  - W(X)3 must wait, until W(X)1 is completed
    - same location
  - P2 may observe R(Y)2 before R(X)1 and R(X)3

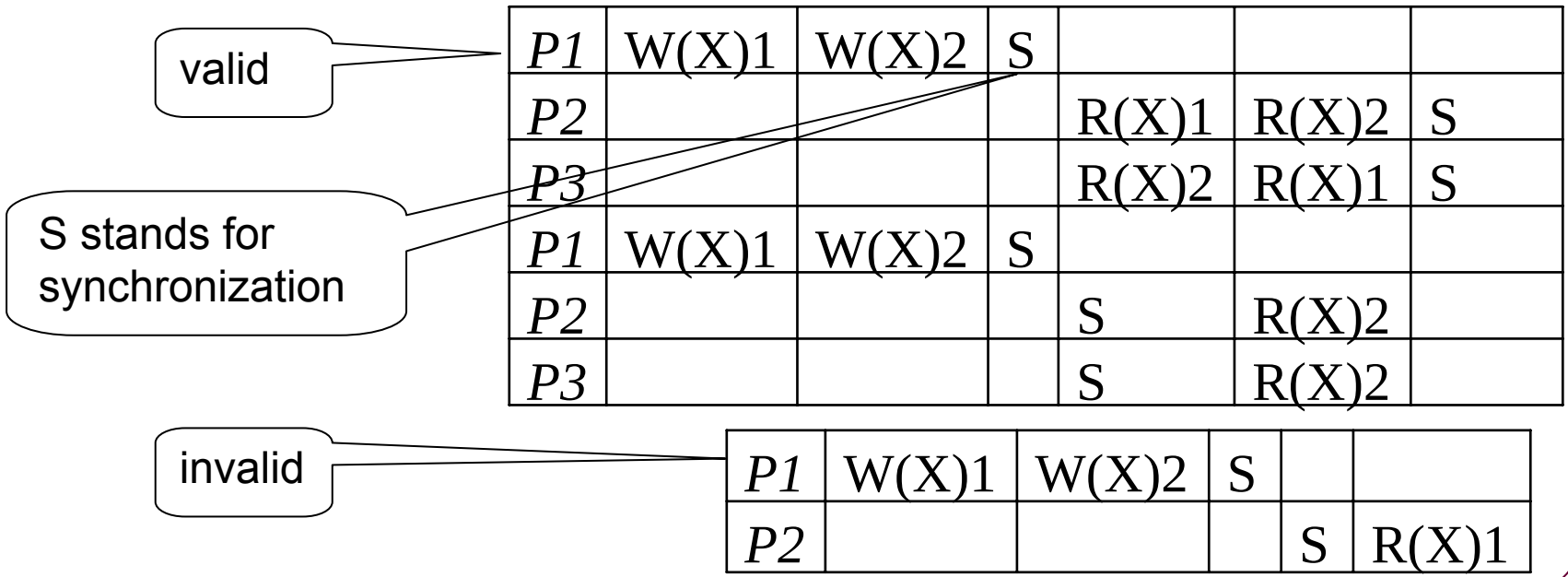
P1	W(X)1	W(Y)2		W(X)3		
P2			R(Y)2		R(X)1	R(X)3

1.2. Synchronization Access Consistency

- It suffices to guarantee consistency at synchronization points
- Two categories of memory access
  - Synchronization access
    - Read/write operations to synchronization (S-)variables
  - General competing read/write access
    - For all other shared variables

Weak Consistency

- Access to S-variables is sequentially consistent
- All previous read/write must have been completed before an access to an S-variable is issued
- Any previous S-access must have been completed before any further read/write access



Release Consistency

- In a critical section access is limited to 1 process anyway
- It suffices to make the memory consistent at leaving the CS
- We are able to distinct the start and the end of a CS
- A CS guards specific shared, *protected* variables
- *Acquire* (enters the critical region, via a lock *L*)
  - All local copies of the protected variables are consistent
  - Before release, modifications can be done locally
- *Release* (exits the critical region, defined by lock *L*)
  - Modified protected variables are broadcasted to all
- Acquire, release must be processor (not seq.) consistent

P1	a(L)	W(X)1	W(X)2	r(L)				
P2					a(L)	R(X)2	r(L)	
P3								R(X)1

Lazy Release and Entry Consistency

- Lazy Release Consistency
  - Updating is postponed until the next acquire
  - If the same process requires the same lock: no action is necessary
  - Advantageous for acquire-release pairs in a loop
- Entry Consistency
  - Each shared variable is associated with an S-variable
    - Even elements of an array have a different S-variable
  - Shared data pertaining to a critical region are made consistent when a critical region is entered
  - The granularity of parallelism is much higher – not for free

Summary of Data-Centric Consistency

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters
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 maybe seen in diff. order

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
Lazy release	Updating is postponed until the next acquire

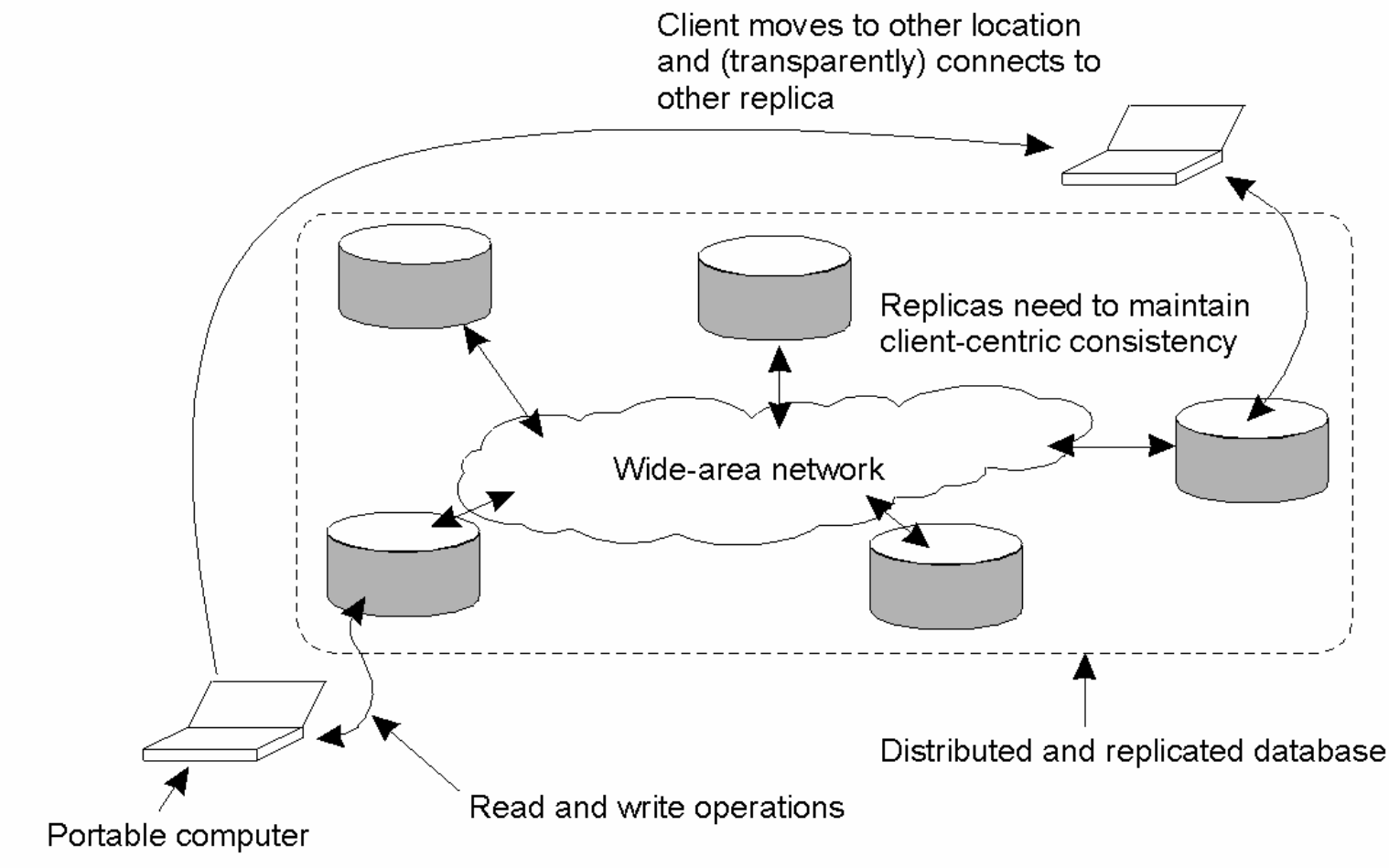
2. Client-Centric Consistency Models

- Often simultaneous updates
  - Do not happen or are easy to resolve
  - In case of databases most users only read
  - In a DNS update is made only by administrators
  - In the WWW most pages are updated only by the owner
- Eventual Consistency
  - Clients may get copies that are not the newest
    - E.g. Web Caches often return some older version
  - If for a certain object for a longer time no updates take place then at the end – eventually – all replicas become consistent

Eventual Consistency (1)

- Problems arise if
  - A client accesses different replicas
  - E.g. the client moves to another place, where another replica is seen
  - The client does not see her own changes!
- Client-centric consistency guarantees that the same client sees always consistent data
- Notation
  - $x_i[t]$: version of data item x at local copy L_i at time t
  - Maybe the result of a *series* of writes at L_i: WS ($x_i[t]$)
  - WS ($x_i[t_1]$; $x_j[t_2]$)
    - If operations of a series WS ($x_i[t]$) are also executed at a later time t_2 on a local copy of x at L_j

Eventual Consistency (2)



Monotonic Read Consistency (1)

- If a process reads the value of item x, any subsequent read returns the same or a more recent value
- E. g. distributed e-mail database
  - Mails can be inserted at different locations
  - Changes are propagated “lazy”, on demand
  - A user reads her mail in Klagenfurt
    - Assume that she does not change the mailbox, only reads the mails
  - She flies to New York and opens her mailbox
  - Monotonic read consistency guarantees that she sees all mails in her mailbox, she saw in Klagenfurt

Monotonic Read Consistency (2)

Mails have been propagated from L1

L1	$WS(X_1)$		$R(X_1)$	
L2		$WS(X_{1;}\ X_2)$		$R(X_2)$

L1	$WS(X_1)$		$R(X_1)$		
L2		$WS(X_2)$		$R(X_2)$	$WS(X_{1;}\ X_2)$

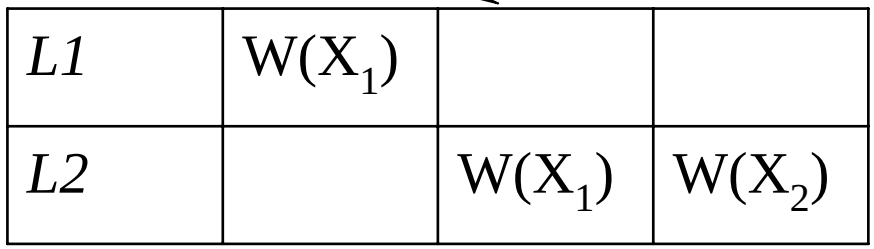
Mails not propagated from L1: no mon. read cons.

Monotonic Write Consistency (1)

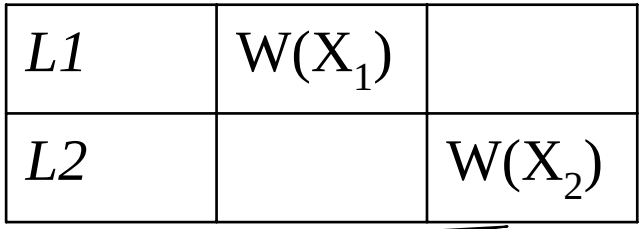
- A write by a process on an item x is completed before any successive write on x by the same process
  - A write is executed only if the result of a previous write has been propagated to all replicas
  - Similar to data-centric FIFO consistency
  - The difference is that here we are interested only on 1 process
- An update of an x may be only a partial change
  - E.g. updating (parts) of a software library
  - Monotonic write guarantees that previous updates have been already done

Monotonic Write Consistency (2)

X_1 has been propagated before $W(X_2)$



L1	$W(X_1)$		
L2		$W(X_1)$	$W(X_2)$



L1	$W(X_1)$	
L2		$W(X_2)$

X_1 not propagated before $W(X_2)$: no mon.wr. cons

Read Your Writes Consistency (1)

- The effect of a write by a process on an item x will be seen by any successive read on x by the same process
  - A write is always completed before a successive read of the same process, regardless the location
- Examples for missing this consistency
  - We change a web page but we still get the old version from the cache – we have to push reload
  - We change the password but the propagation takes time

Read Your Writes Consistency (2)

Read-your-writes consistency

L1	$W(X_1)$		
L2		$WS(X_1; X_2)$	$R(X_2)$

L1	$W(X_1)$		
L2		$WS(X_2)$	$R(X_2)$

No read-your-writes consistency

Write Follows Reads Consistency (1)

- A write by a process on x following a previous read of the same process, is guaranteed to take place on the same or a more recent value of x.
  - Any successive write on an item x will be performed on a copy of x that is up-to-date with the value most recently read by the same process
- Examples for missing this consistency
  - In a network discussion group we get the answer to some articles we have not read yet

Write Follows Reads Consistency Consistency (2)

Write-follows-Reads consistency

L1	$WS(X_1)$	$R(X_1)$		
L2			$WS(X_{1;}\ X_2)$	$W(X_2)$

L1	$WS(X_1)$	$R(X_1)$		
L2			$WS(X_2)$	$W(X_2)$

No Write-follows-Reads consistency

Implementing Client-centric Consistency (1)

- Each write operation is assigned a globally unique identifier
  - Such an identifier (id) can be generated locally
    - E.g. based on Lamport time stamps, including processor numbers
  - The id is assigned by the server that *initiates* the operation
- For each client 2 sets of write identifiers are managed
  - The read set contains the write ids relevant for reads
  - The write set contains the ids of the writes performed by the client

Implementing Client-centric Consistency (2)

- Implementing monotonic-read consistency
  - When a client performs a read at a server, this gets the client's read-set to check whether all writes have taken place locally
  - If not, it contacts the other servers and gets and *replays* the missing writes – with help of corresponding log infos
  - Alternatively, it forwards the read to an up-to-date server
  - After performing the read, the read-set is updated
  - The algorithm requires that
    - The write-id contains the server id who made the write
    - The order of the write operations is also recorded (Lamport TS)
- Monotonic-write, read your writes and write follows reads
  - MW: The server gets the write-set before write
  - RYW: The server gets the write-set before read
  - WFR: The server gets the read-set before write

Improved Implementation of Client-centric Cons. (1)

- The read and write sets may become huge
  - The naïve algorithm described above becomes slow
- Sessions
  - The read and write operations are grouped into *sessions*
  - E.g. each application starts a new session
  - The sets are cleared after a session
- Representation of the id sets by vector timestamps
  - The first write at a server gets an write id *WID* and a timestamp *TS(WID)*
  - Further writes get only a new timestamp
  - Each server S_i maintains a vector timestamp RCVD(i), where RCVD(i)[j] = TS of the latest write initiated at S_j and seen (received and processed) by S_i
  - After a read or a write, a server returns beside the result the timestamp
  - Read and write sets are represented by vector timestamps

Improved Implementation of Client-centric Cons. (2)

- For such a set A, VT(A)[i] = max. timestamp of all operations in A that were initiated at S_i
  - This is more efficient than storing the entire set
- The union of two sets A and B is represented as

  VT(A+B); | VT(A+B)[i] = max{VT(A)[i], VT(B)[i]}
- A is contained in B, iff VT(A)[i] ≤ VT(B)[i] for all i
- When a server passes its current timestamp to the client, the client adjusts the vector timestamps of its own r-w sets

- Example for monotonic read
  - The client gets RCVD(i) from S_i
  - Vector timestamp of the client's read set: VT(Rset)
  - ∀ j: VT(Rset)[j] := max {VT(Rset)[j], VT(RCVD)[j]}
  - VT(Rset) now reflects the latest writes the client has seen
  - It will sent along with the next read, possibly to a different server S_j

Implementation of Data-Centric Replication

1. How should be the replicas distributed?
2. How should be they kept consistent?

- Replica Placement, three main types
  1. Permanent replicas
  2. Server-initiated replicas
  3. Client-initiated replicas

Permanent replicas

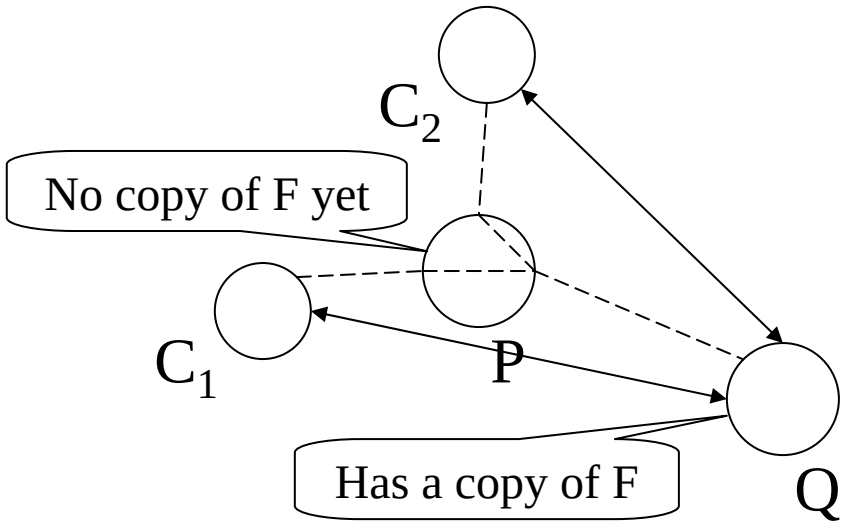
- Initial set of replicas
- Typically a small set, set up statically
- Server group on a local network
  - Important data (e.g. Web pages) are replicated "manually"
  - Requests are served by one of them – e.g. round robin
- Mirroring
  - Replication of sites at locations of great geographical distance
  - Requests are served by the nearest replica
    - E.g. mirroring French data in Berlin and Vienna

Server-initiated replicas (1)

- If the server is overloaded, it pushes replicas to places where the requests come from – *push cache*
  - Especially popular for Web hosting services (high read/write ratio)
- Definitions and a simplified algorithm
  - Requests for file *F* from clients C_1 and C_2 sharing the same "closest" server node *P* are counted together at server node *Q:* $cnt_Q(P, F)$ (as if they would come from *P*)
  - Total number of requests for a file *F*: *req(F)*
  - Replication threshold on server *S* for file *F*: *rep(S, F)*
  - Deletion threshold: *del(S, F)* *(del(S, F) < rep(S, F))*
  - If *req(F) > rep(S, F)*: try to replicate
  - If *del(S, F) < req(F) < rep(S, F):*
    - Maybe try to migrate: Replication is unnecessary but finding a better place can provide better response time

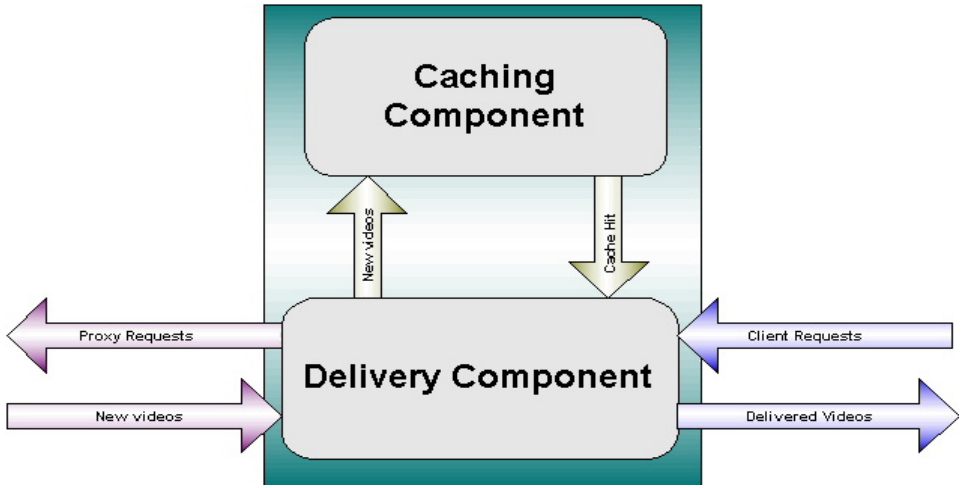
Server-initiated replicas (2)

- Replica placement at server node *Q*
  - If *req(F)* < *del(S, F)*: delete *F*, unless this is the last
  - If *req(F)* > *rep(Q, F)* & *cntQ(P, F)* > *req(F)/2*:
    ask *P* to take over a copy
  - If *P* is too busy: check further nodes, starting with the farthest



Client-initiated replicas

- Data required by the client are cached
  - At the client or
  - In a proxy
    - Many clients on the same LAN can share the same data
    - Useful for Web pages, not so much for usual files
  - Consistency – see later
- Update propagation
  - What to propagate?
  - How to propagate?



What to propagate?

- Propagate only a notification *(invalidation protocol)*
  - The replicated data are marked as *invalid*
  - Before the next use, the new version must be loaded
  - Useful if read/write ratio is low
    - Don't update large data items, maybe without a reader
- Transfer new data to all replica places
  - Useful if read/write ratio is high
  - Partial update
    - We re-send only a certain part of the data
  - Differential update
    - We re-send only a log of the changes
- Active replication
  - Propagate only the update operations
  - We let the replica holders to "replay" the changes
  - Needs less bandwidth, for the price of CPU power

How to propagate? (1)

- Push-based (or server-based) protocols
  - The replicas are pushed by the server, without a request
  - Keeps the replicas always identical
    - Needs to manage a list of clients
  - Useful, when read/write ratio is high
    - Many clients read the same cached data
    - Can be combined well with multicast
    - Updates can be aggregated and sent together

How to propagate? (2)

- Pull-based (or client-based) protocols
  - On requests, cache polls the server for modifications
  - If no modification, send cached data, otherwise reload
  - Useful, when read/write ratio is relatively low
    - E.g. with non-shared cache at the client
  - Disadv.: the response time is high if reload is needed

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

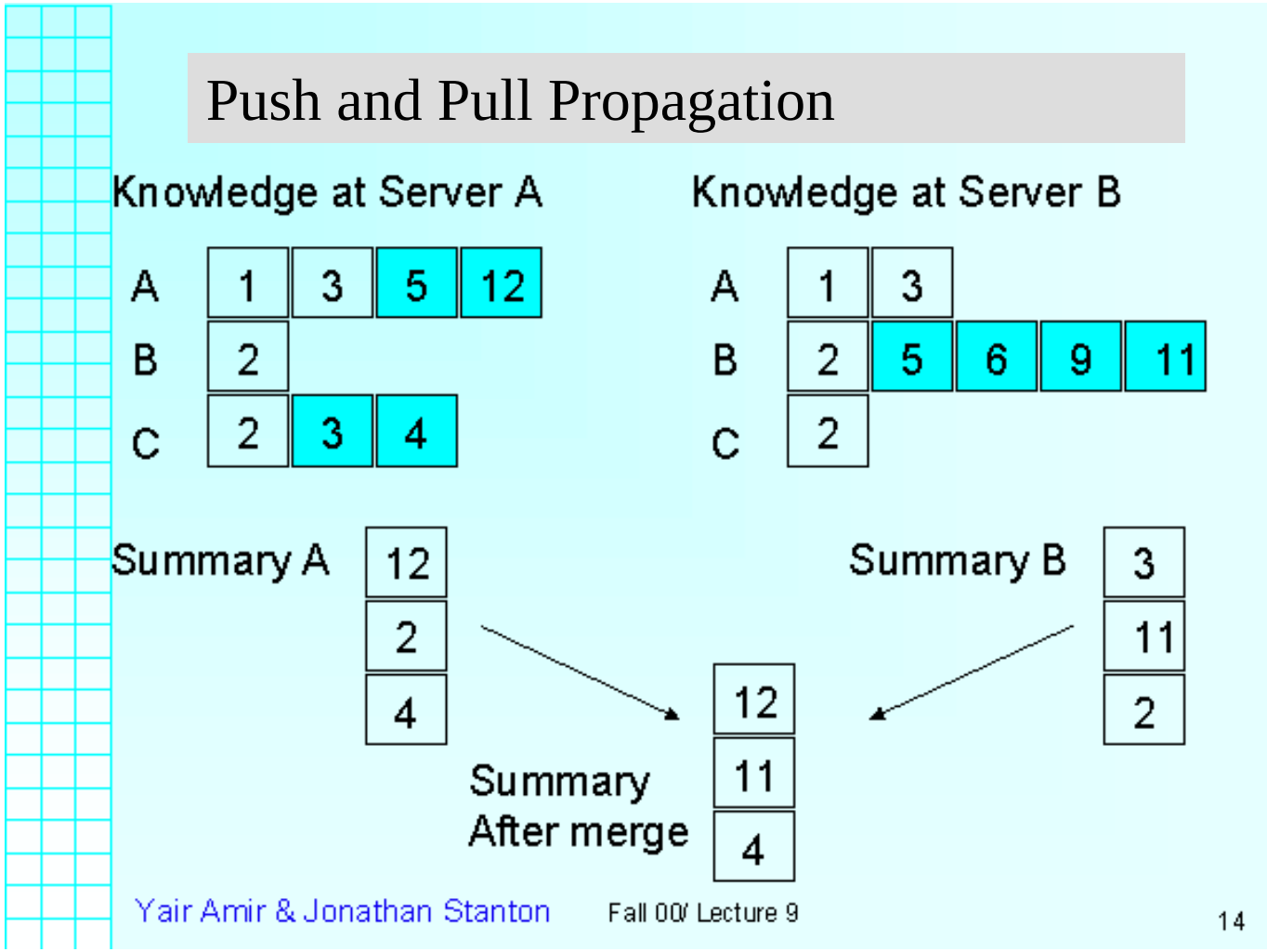
How to propagate? (3)

- Lease-based protocols
  - A combination of push and pull
  - *Lease*: The server must push changes for a specified expiration time
  - After this time the client has to pull or take a new lease
  - Age-based lease
    - Data, possibly unchanged for long, get a long-lasting lease
  - State-space based lease
    - If the state-space at the server becomes large, it sinks the expiration times. It tends to a stateless mode (exp. time = 0)
  - Renewal-frequency based lease
    - Often asked data get long-lasting lease to improve cache hit rate (data are refreshed only where they are popular)

“Epidemic” protocols (1)

- Spreads updates with minimal network load
- Especially useful for eventually consistent data
- Holder of a new update that should be spread: *infective*
- Not yet updated (“infected”): *susceptible*
- Server not willing or able to spread its update: *removed*
- *Anti-entropy propagation*
  - Entropy: A measure of the loss of information
  - Server A picks B randomly and exchange updates. 3 ways:
  - A only pushes its updates to B
    - Bad, if many serves are infective: they cannot push each other
  - A only pulls new updates from B
    - Good, if many servers are already infective
  - A and B send updates to each other
  - Infects *everyone* in O(log n) effort for uniformly chosen sites

“Epidemic” protocols (2)



“Epidemic” protocols (3)

- *Rumor spreading* or *gossiping*
  - If a server *P* contacts *Q* who already “knows” the new value of *x* then *P* loses interest in spreading the update, with probability 1/k
    - It makes no “fun” to spread a gossip, many people know already
  - Gossiping is a rapid way to spread updates
  - Cannot guarantee that all *s* servers will be updated
    - Remaining-susceptible rs = $e^{-(k + 1)(1 - s)}$
    - If k = 3, rs < 0,02, i.e. less than 2%
    - Gossiping must be combined e.g. with anti-entropy to finish the job

“Epidemic” protocols (4)

- *Removing data*
  - If we simply delete data then this will not be propagated
  - A server might get old copies and interpret them as new data
  - Deleted data must be recorded as normal data, but marked as deleted
  - Spreading of “death certificates”
  - After a certain time such records may be deleted finally

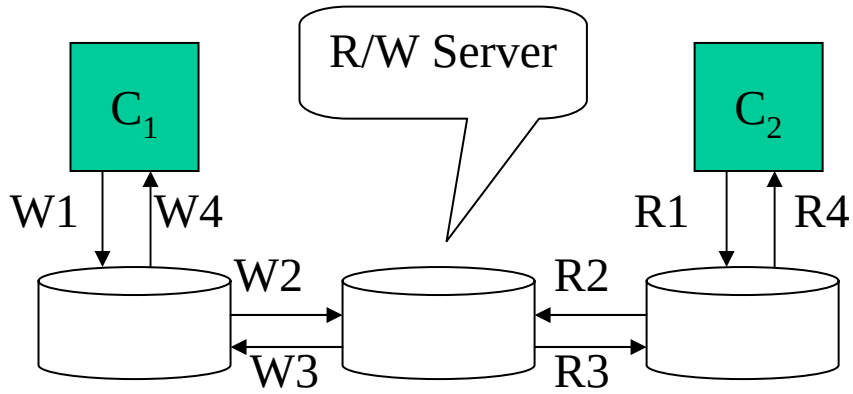
Consistency Protocols

- Implement consistency models
- Most important models are globally realized
  - Sequential consistency
  - Weak consistency with synchronization variables
  - Atomic transactions

1. Primary-Based Protocols
   - Each data item x has a primary server which is responsible for coordinating its changes
2. Replicated-Write Protocols
   - Writes can be carried out on multiple replicas

Remote-Write Protocols (1)

- All read and write operations on *x* at the same server
  - Client/server systems
  - Sequential consistency is trivial
  - No replication, no fault tolerance

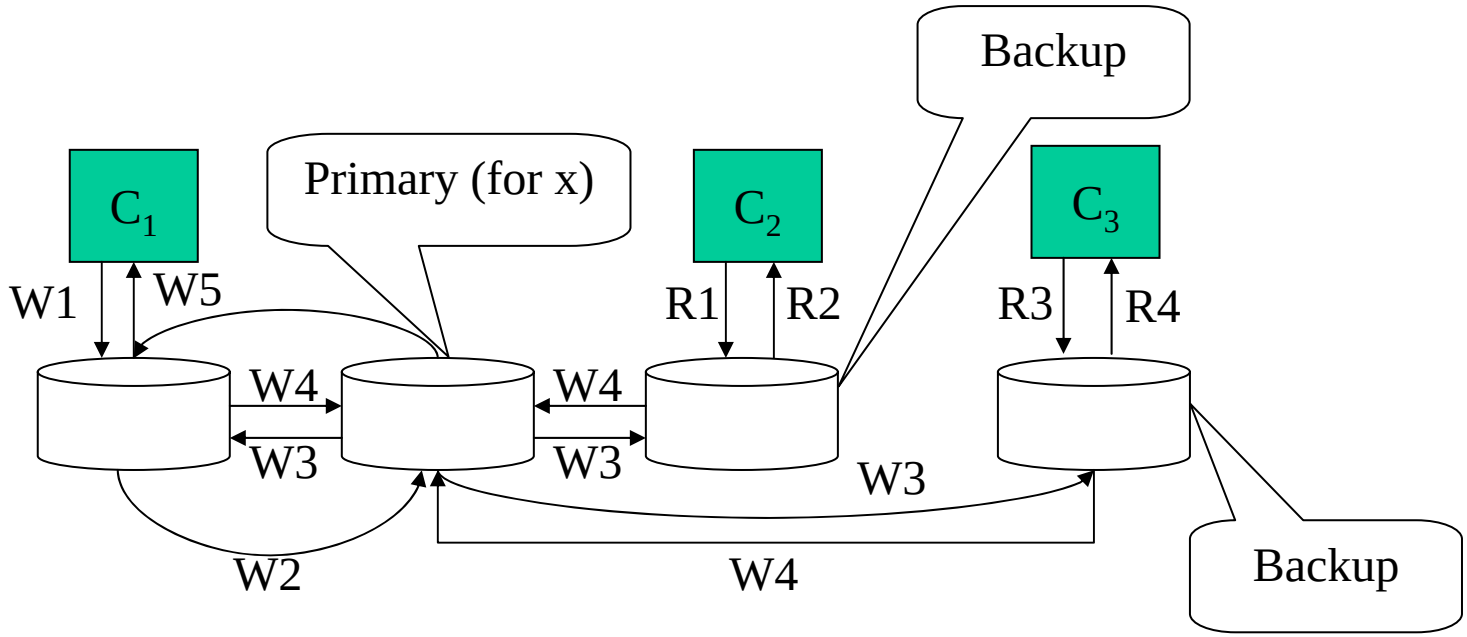


W1: Write request	R1: Read request
W2: Forward request server	R2: Forward req. to serv.
W3: Ack. Write compl.	R3: Return response
W4: Ack. Write compl.	R4: Return response

Remote-Write Protocols (2)

- *Primary-backup protocols*
  - Fault tolerance is enhanced by backup copies
  - Reads can be served locally
  - All writes are propagated to all backups
  - A write is acknowledged only after this propagation – Actually a *blocking* operation!
  - Sequential consistency is straightforward
    - The primary can order the writes
  - Performance of writes maybe poor
    - Long waiting time for propagation
  - *Non-blocking primary-backup*
    - As soon as the primary is updated the write is acknowledged
    - Fast, but fault tolerance is difficult

Remote-Write Protocols (3)

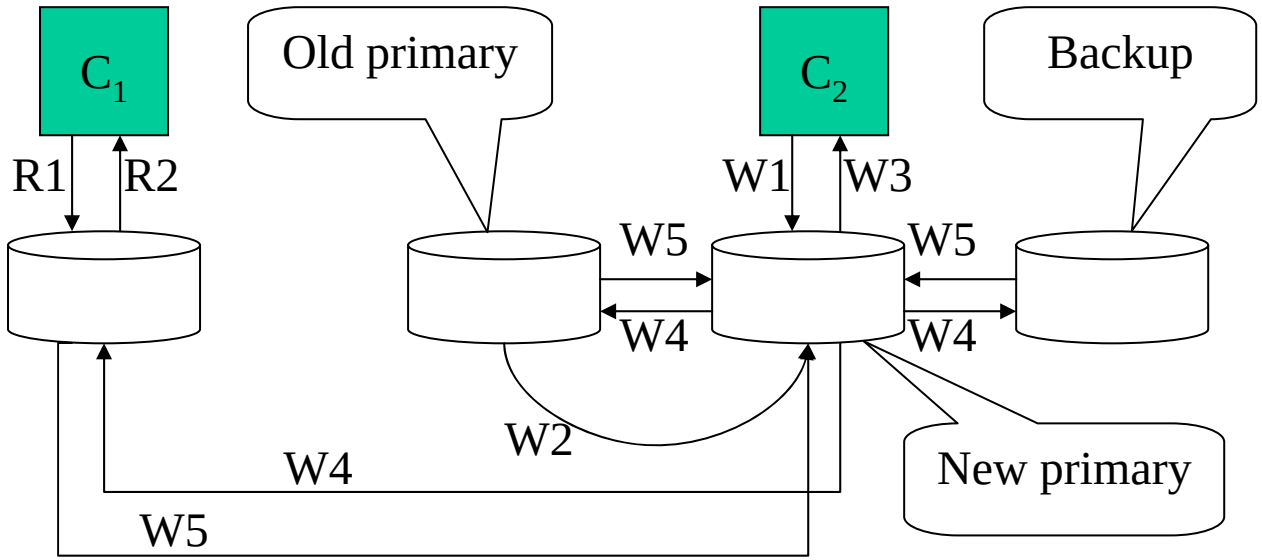


W1: Write request	W5: Ack. Write compl.
W2: Forward request to primary	
W3: Tell backups to update	R1, R3: Read request
W4: Acknowledge update	R2, R4: Response to read

Local-Write Protocols (1)

- *Full migration (used mostly in DSM)*
  - There is one single copy of each *x*
  - At a write request *x* is sent to the requester
  - Hard to know where is the actual instance
    - Broadcasting can be used on a LAN
    - Forwarding pointers may point to the proper place
  - Block-bouncing danger
- *The primary copy migrates* to the requester
  - Multiple, successive writes can be done locally
  - A non-blocking protocol must update all replicas
  - The primary may even go disconnected (mobile device) and complete updating
  - Other sites may complete reads, but not writes

Local-Write Protocols (2)

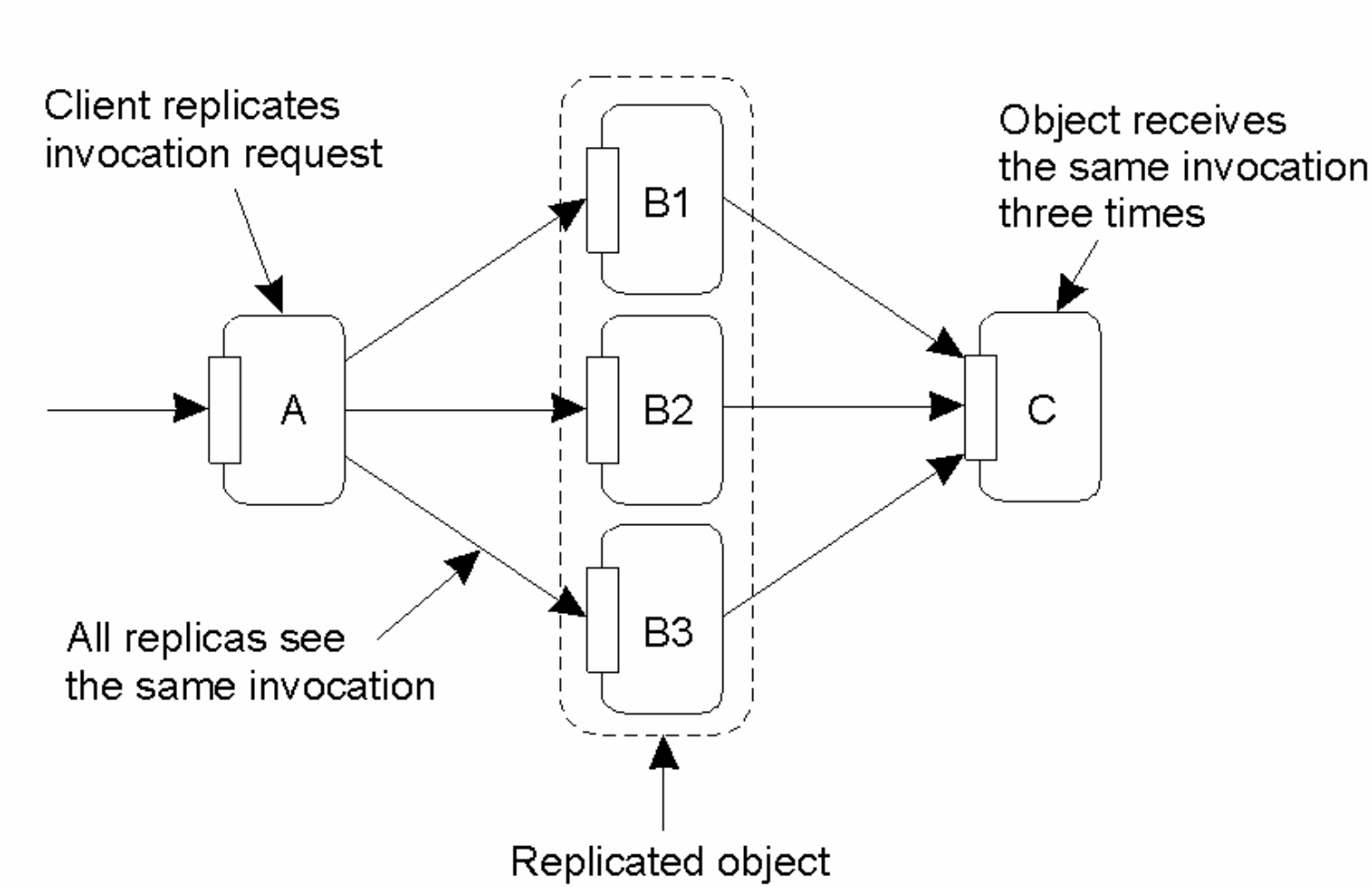


W1: Write request	W5: Ack. update
W2: Move *x* to new primary	
W3: Ack. write compl.	R1: Read request
W4: Tell backups update	R2: Response to read

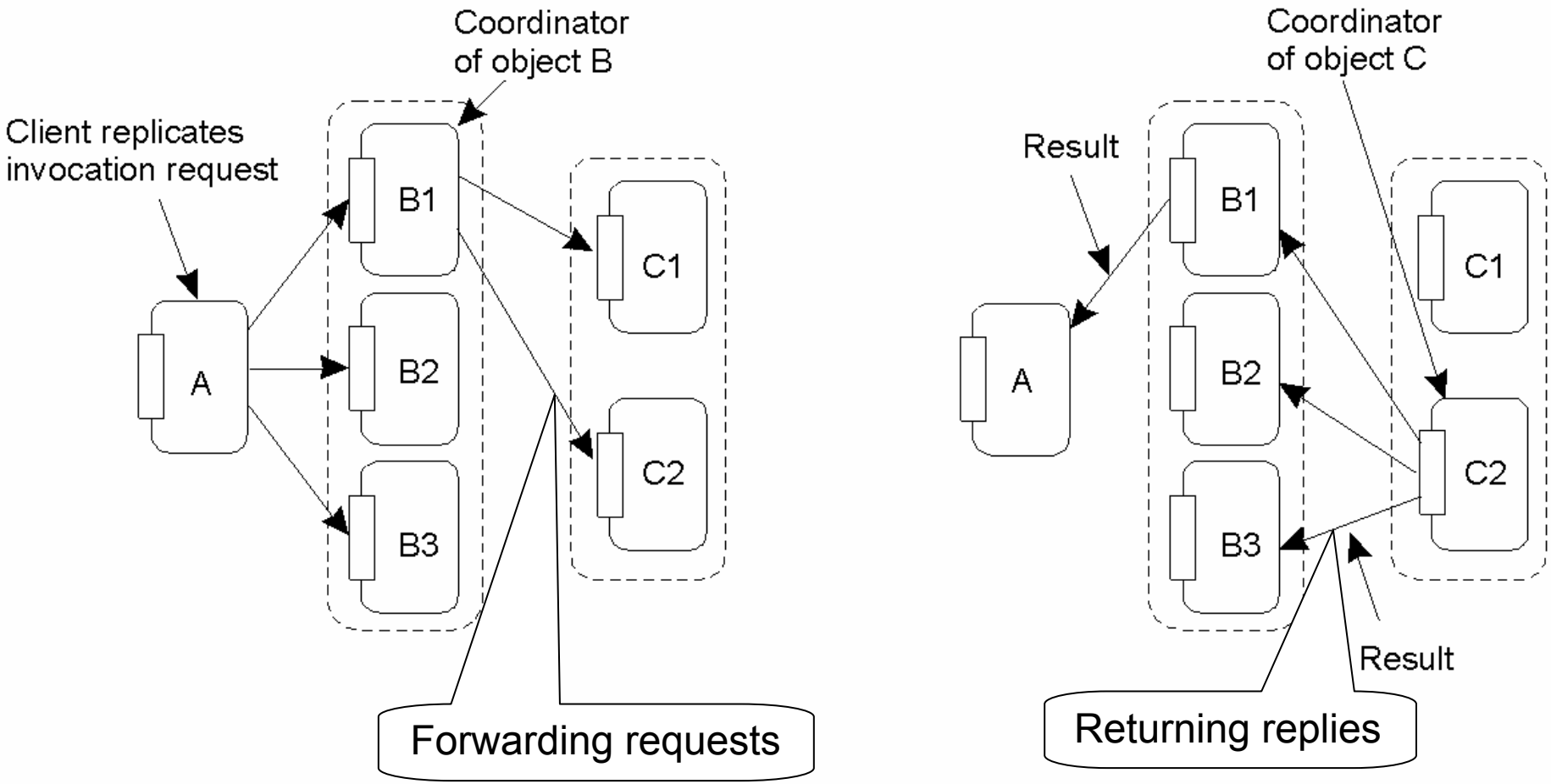
Active Replication

- Not the data but the operations are propagated
- The order of the operations must be the same everywhere
- A totally ordered multicast is necessary. Implementation by
  - *Sequencer*
    - All operations are sent first to a sequencer
    - The sequencer assigns unique sequence numbers
    - Does not scale well
    - Improvement by combination with Lamport time stamps
  - *Replicated invocations* must be prohibited
    - Operation *A* is propagated to the replicas *R1, R2, R3*
    - Operation *A* calls an operation *m* on an object *Q*
    - *Q* would receive 3 calls instead of 1 (1 from each replica)
    - Responses can be replicated in a similar way
    - One of the replicas must play the role of a *coordinator* in order to suppress unnecessary calls and responses

The problem of replicated invocations



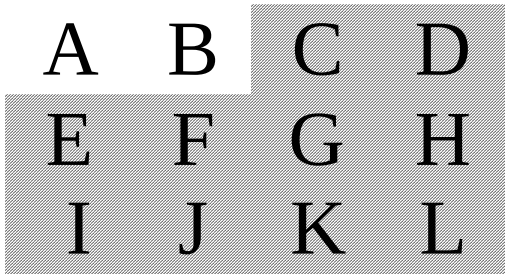
A solution for replicated invocations

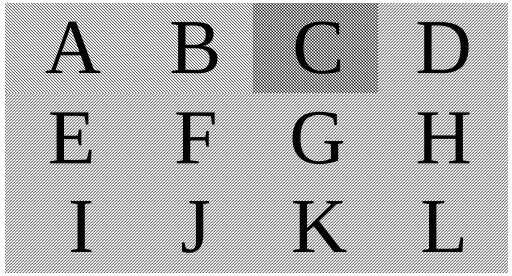


Quorum-Based Protocols (1)

- Based on *voting*
- Read quorum (N_R)
  - Num. of servers must agree on version num. for a read
- Write quorum (N_W)
  - Num. of servers must agree on version num. for a write
- If N is the total number of replicas,
  N_R and N_W must fulfill:
  - $N_R + N_W > N$
    - Prevents read-write conflicts
  - $N_W > N/2$
    - Prevents write-write conflicts
  - Examples (N = 12)

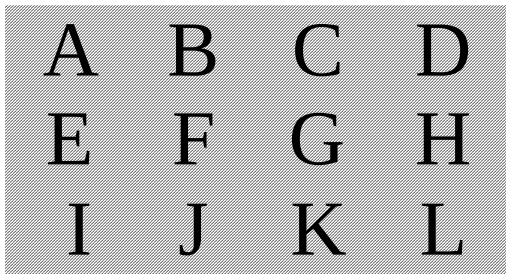
Quorum-Based Protocols (1)

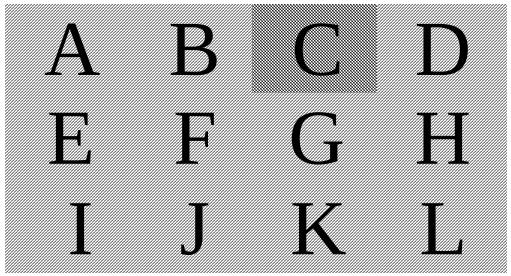




$N_W = 10$ $N_R = 3$ (read sees last update)

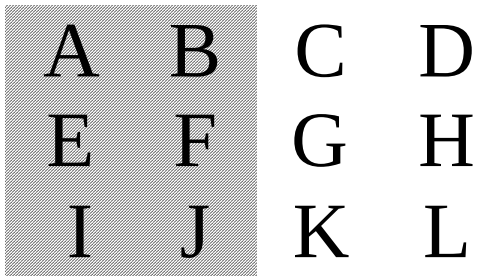
Quorum-Based Protocols (2)

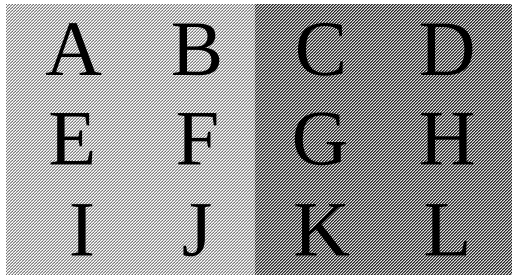




$N_W = 12 \quad N_R = 1$ (ROWA: read one, write all)

Quorum-Based Protocols (3)





$N_W = 6$ $N_R = 7$ (bad choice: $N_W > N/2$ not true)

May cause write-write conflicts