Agenda

- Client-centric consistency models
- Consistency protocols
- Pessimistic replication vs. optimistic replication
- Clocks, logical clocks, state vectors

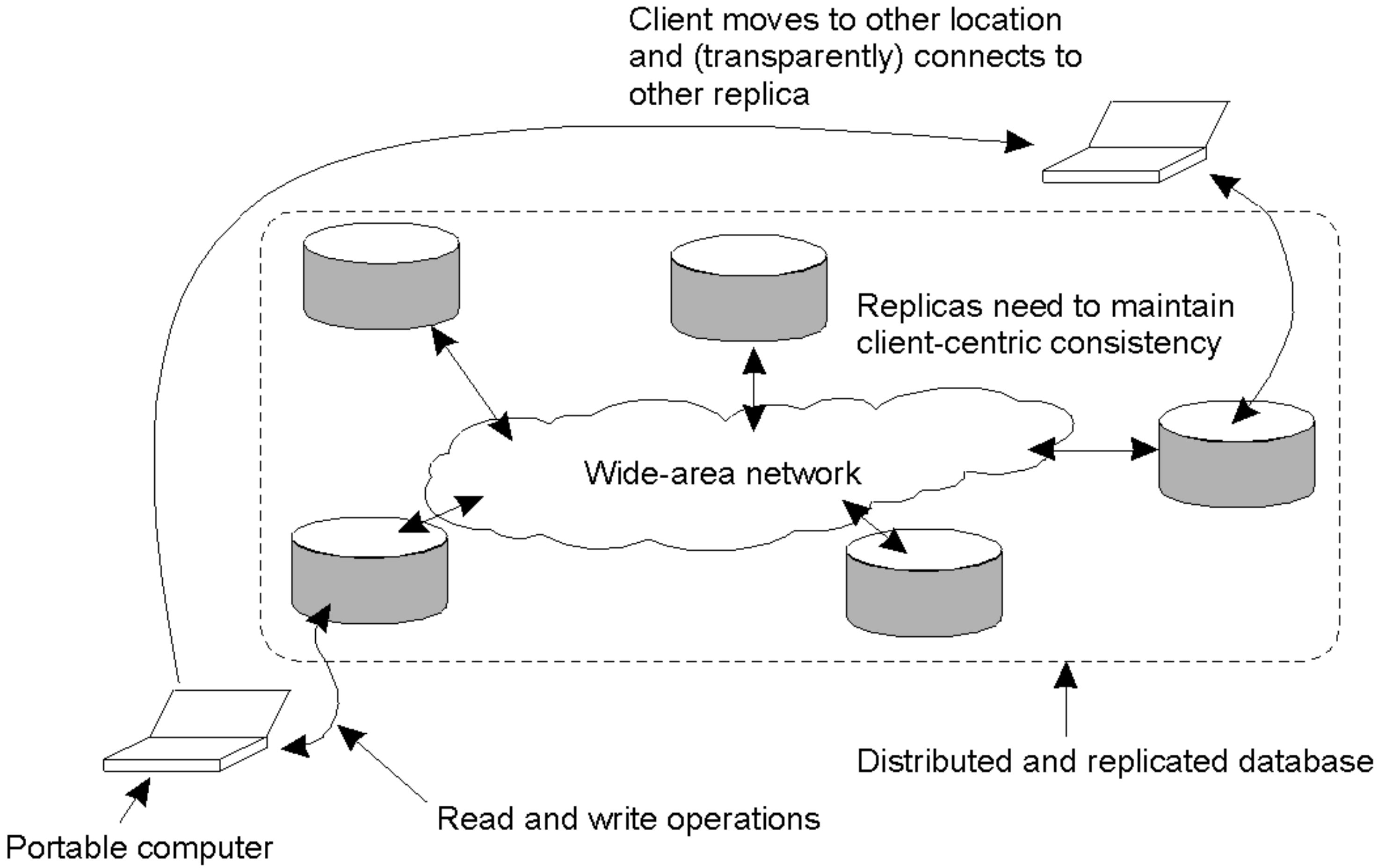
Data Centric vs. Client Centric Consistency Models

- Data-centric consistency model
  - Provides a systemwide consistent view on a data store
  - Assumption: concurrent processes may simultaneously update the data store
- Client-centric consistency model
  - Assumption: lack of simultaneous updates or when updates happen they can easily be resolved
  - Most operations involve reading data
  - Data stores offer a very weak consistency model called eventual consistency

Eventual Consistency (1)

- Concurrency appears only in a restricted form
  - Database systems
    - most processes hardly perform updates, but only read data
  - DNS
    - domains assigned to naming authorities that are allowed to update their part of the name space
    - no write-write conflicts, only read-write conflicts
    - Acceptable to propagate an update in a lazy fashion
  - World wide web
    - Web pages updated by a single authority
    - No write-write conflicts
    - Design of local caches for efficiency
- If no updates take place for a long time, all replicas become consistent

Eventual Consistency (2)



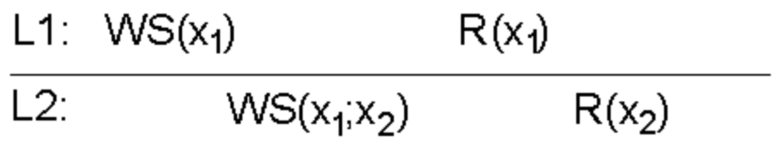
Client centric consistency

- Provides guarantees for a single client concerning consistency of accesses to a data store by that client
- A client connects to different replicas during a period of time and the differences should be made transparent
- Whenever a client connects to a new replica, that replica is brought up to date with the data that was manipulated by that client before and can reside at other replica sites
- Notations
  - $x_i[t]$ version of x at local copy L_i at time t
  - $WS(x_i[t])$ series of write operations at L_i that took place since initialization
  - $WS(x_i[t_1];x_j[t_2])$ if operations in $WS(x_i[t_1])$ have been performed at the local copy L_j at time t_2

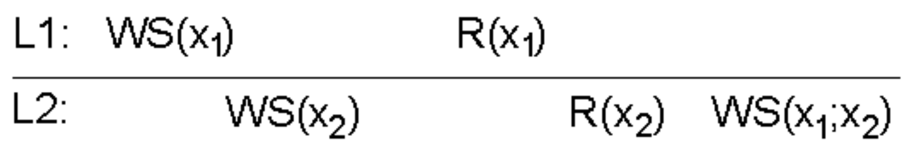
Monotonic Reads (1)

- *If a process reads the value of a data item x, any succesive read operation on x by that process will always return that same value or a more recent value*
- Example:
  - A distributed email database
  - Each user’s mailbox may be distributed and replicated across multiple machines
  - Mail can be inserted at any location
  - Updates propagated in a lazy manner
  - Emails read (no remove, etc.) at location X are present when they are read later at location Y

Monotonic Reads (2)



(a)



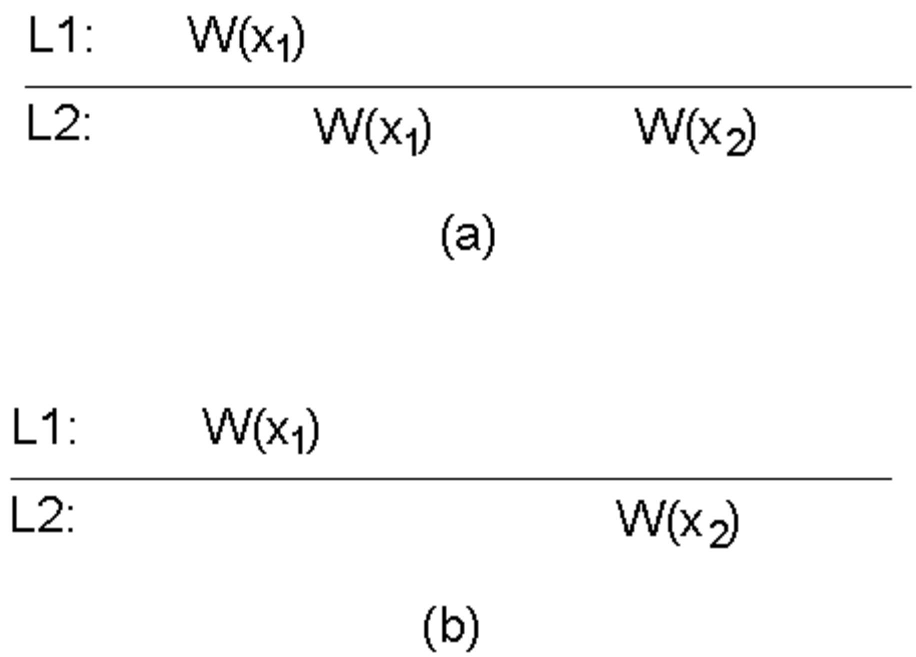
(b)

a) A monotonic-read consistent data store
b) A data store that does not provide monotonic reads.

Monotonic Writes (1)

- *A write operation by a process on a data item x is completed before any successive write operation on x by the same process*
- Similarity with FIFO consistency
- Example: updating a software library

Monotonic Writes (2)



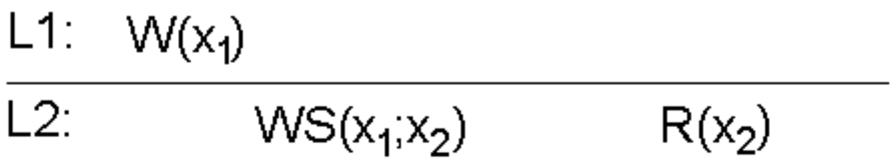
a) A monotonic-write consistent data store.

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

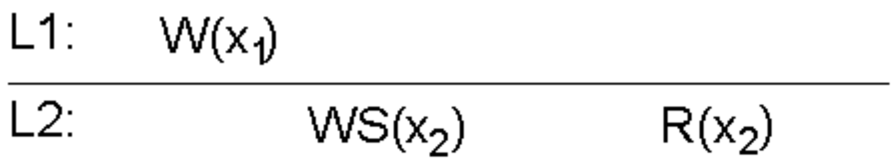
Read Your Writes (1)

- *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*
- Examples
  - Updating passwords
  - Reading and deleting mails

Read Your Writes (2)



(a)



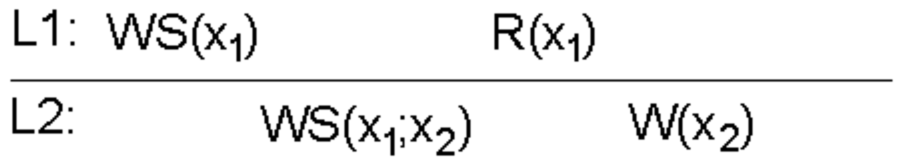
(b)

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

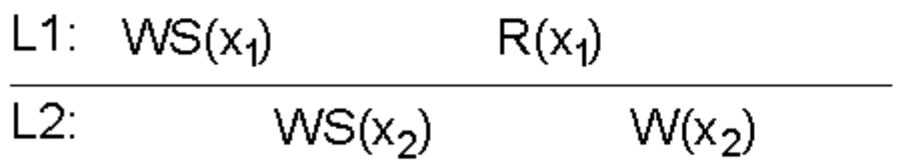
Writes Follow Reads (1)

- *A write operation by a process on a data item x following a previous read operation on x by the same process, is guranteed to take place on the same or a more recent value of x that was read*
- Example:
  - Network newsgroup that see a posting of a reaction to an article only after they have seen the original article

Writes Follow Reads (1)



(a)



(b)

a) A writes-follow-reads consistent data store

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

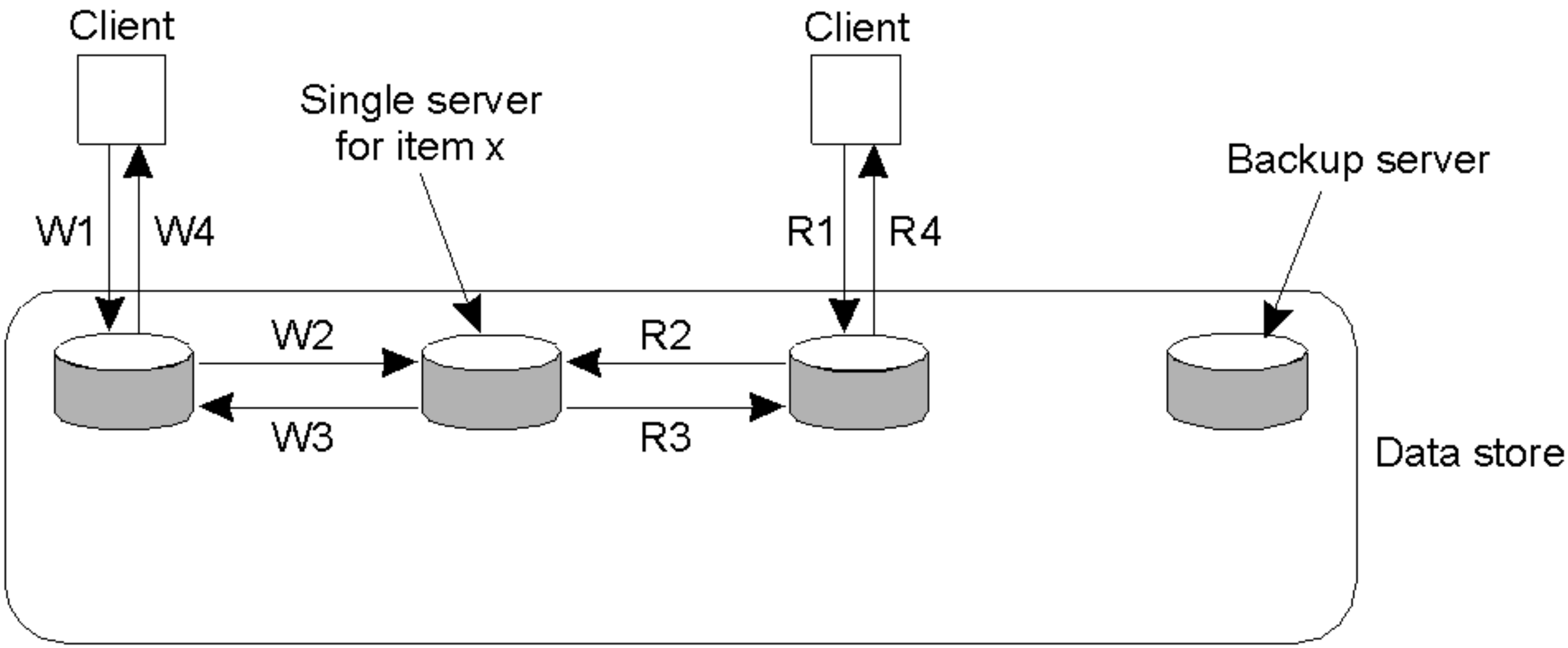
Examples Systems Consistency

- Microsoft's Windows Azure – strongly consistent storage
  - Clients see the latest written value for a data object
- Amazon Simple Storage Service (S3) - eventual consistency
  - Relaxed consistency for better performance and availability
  - Returned value for reads=object value at some past point in time but not necessarily the latest value
- Amazon's DynamoDB and Google App Engine Datastore – both eventually consistent reads and strongly consistent reads

Consistency protocols

- Describe implementation of a specific consistency model
- Primary-based protocols
  - Each data item x has an associated primary responsible for coordinating write operations on x
  - Remote-write protocols
  - Local-write protocols
- Replicated-write protocols
  - Write operations can be carried out at multiple replicas
  - Active replication
  - Quorum-based protocols

Remote-Write Protocols (1)

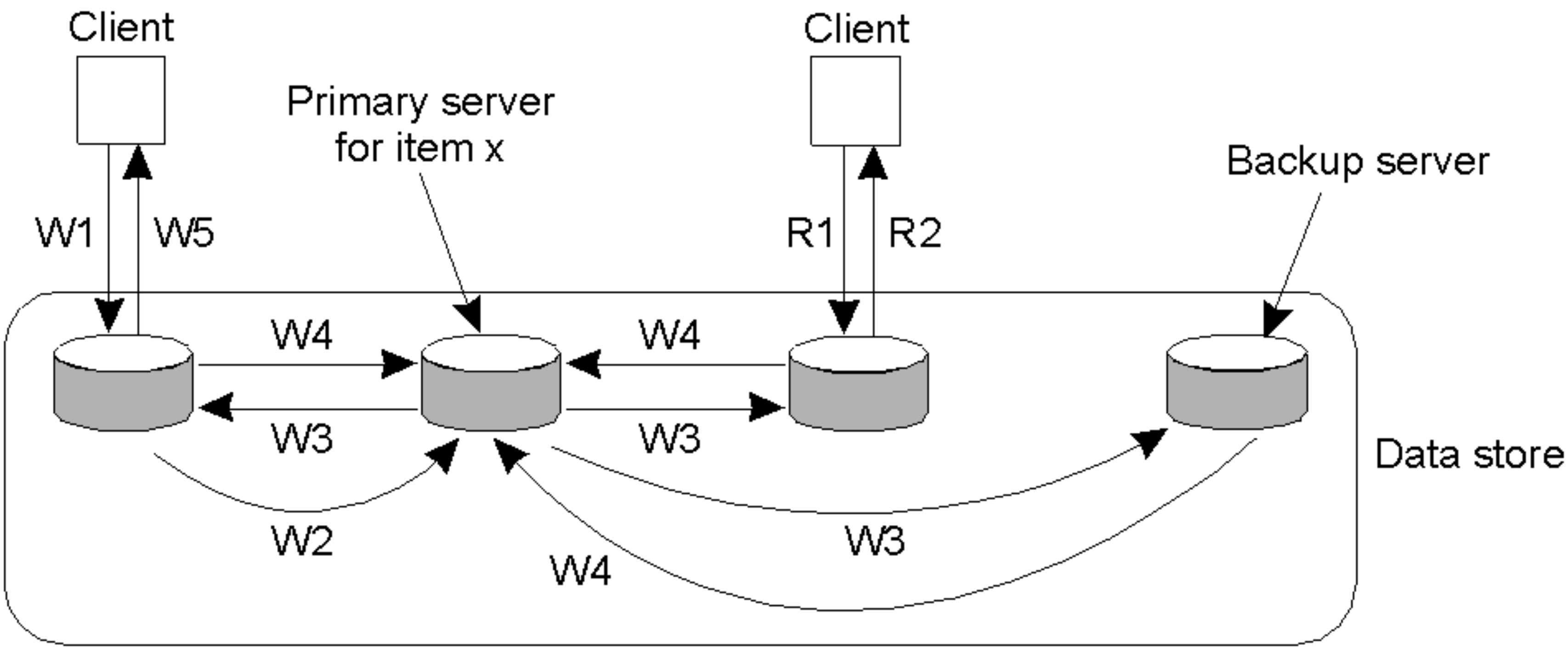


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

R1. Read request
R2. Forward request to server for x
R3. Return response
R4. Return response

- Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded.

Remote-Write Protocols (2)



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

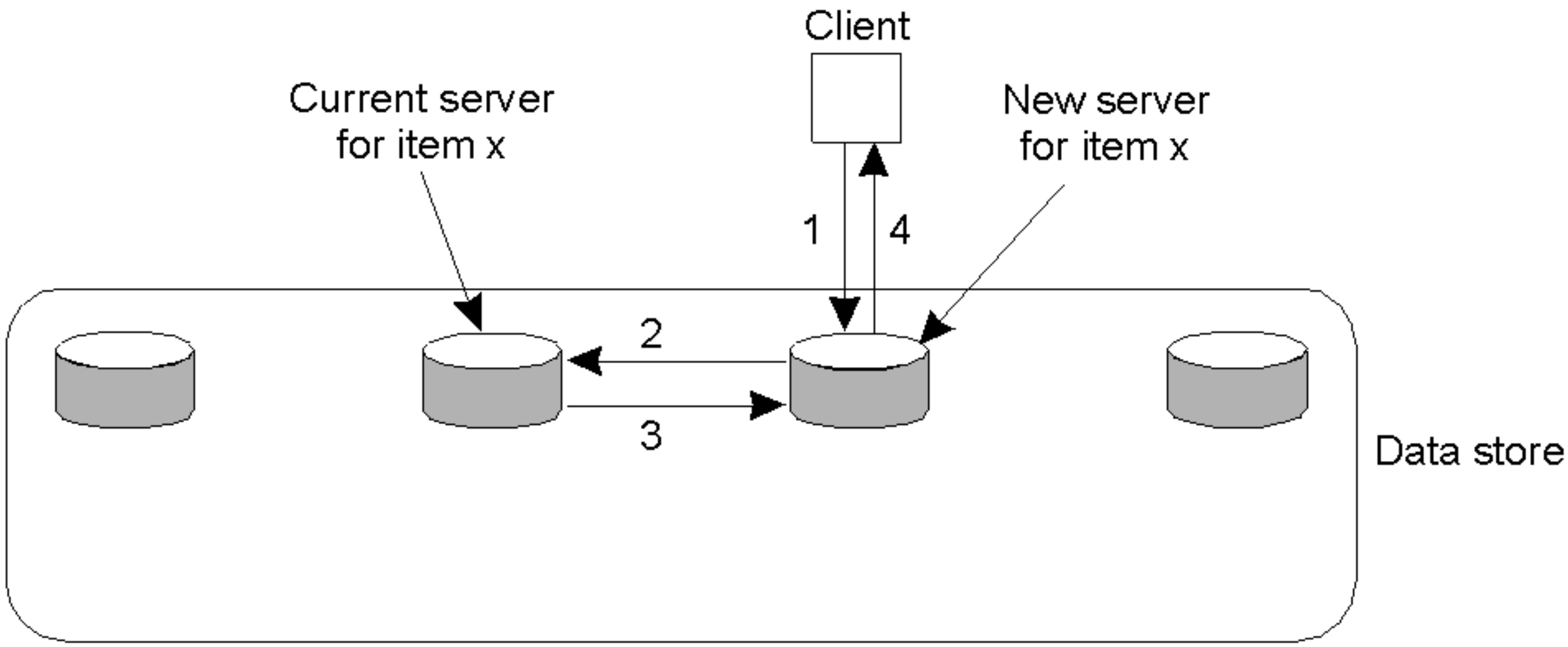
R1. Read request
R2. Response to read

- **The principle of primary-backup protocol.**

Remote-Write Protocols (3)

- Primary-backup protocol implements update as a blocking operation
- Alternative solution: non-blocking protocol
  - As soon as primary updated the local copy of x, it returns an acknowledgement
  - After that ask backup servers to perform the update
  - Fault tolerance concerns
- Implementation of sequential consistency

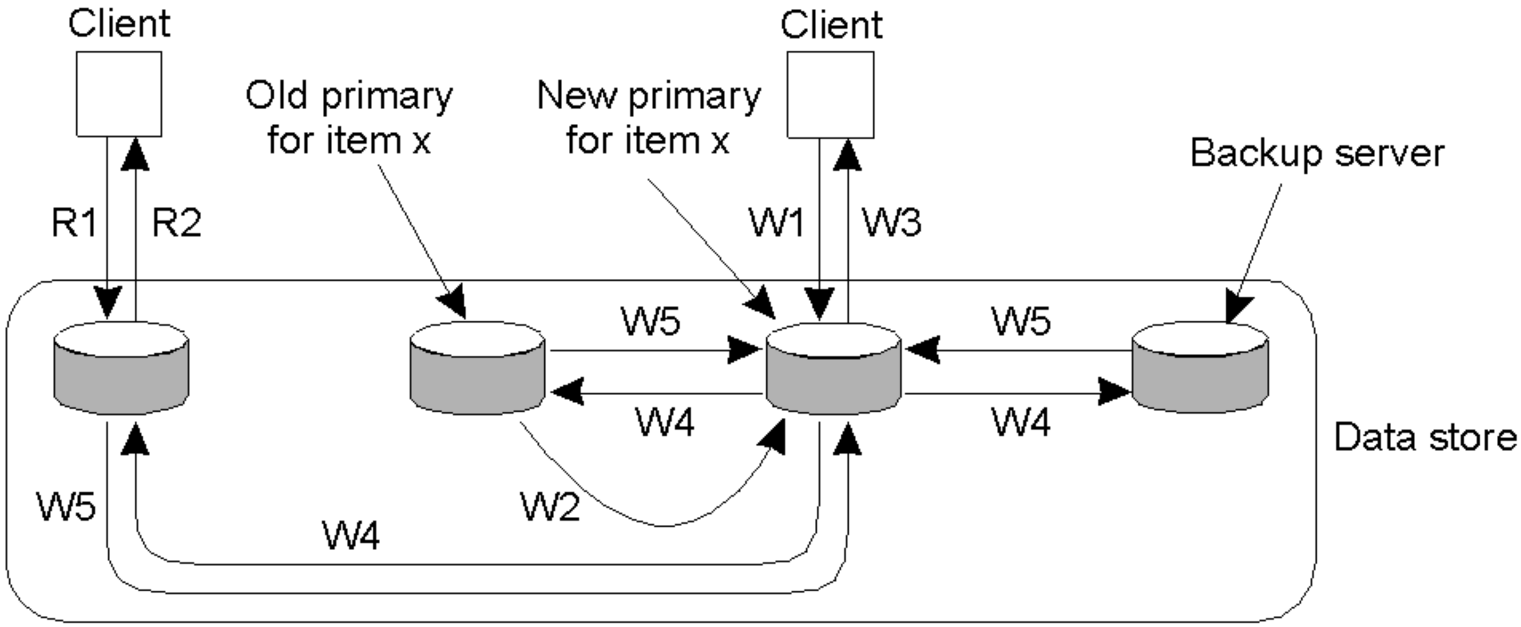
Local-Write Protocols (1)



1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

- Primary-based local-write protocol in which a single copy is migrated between processes
- Disadvantage: keeping track where each data item currently is

Local-Write Protocols (2)



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

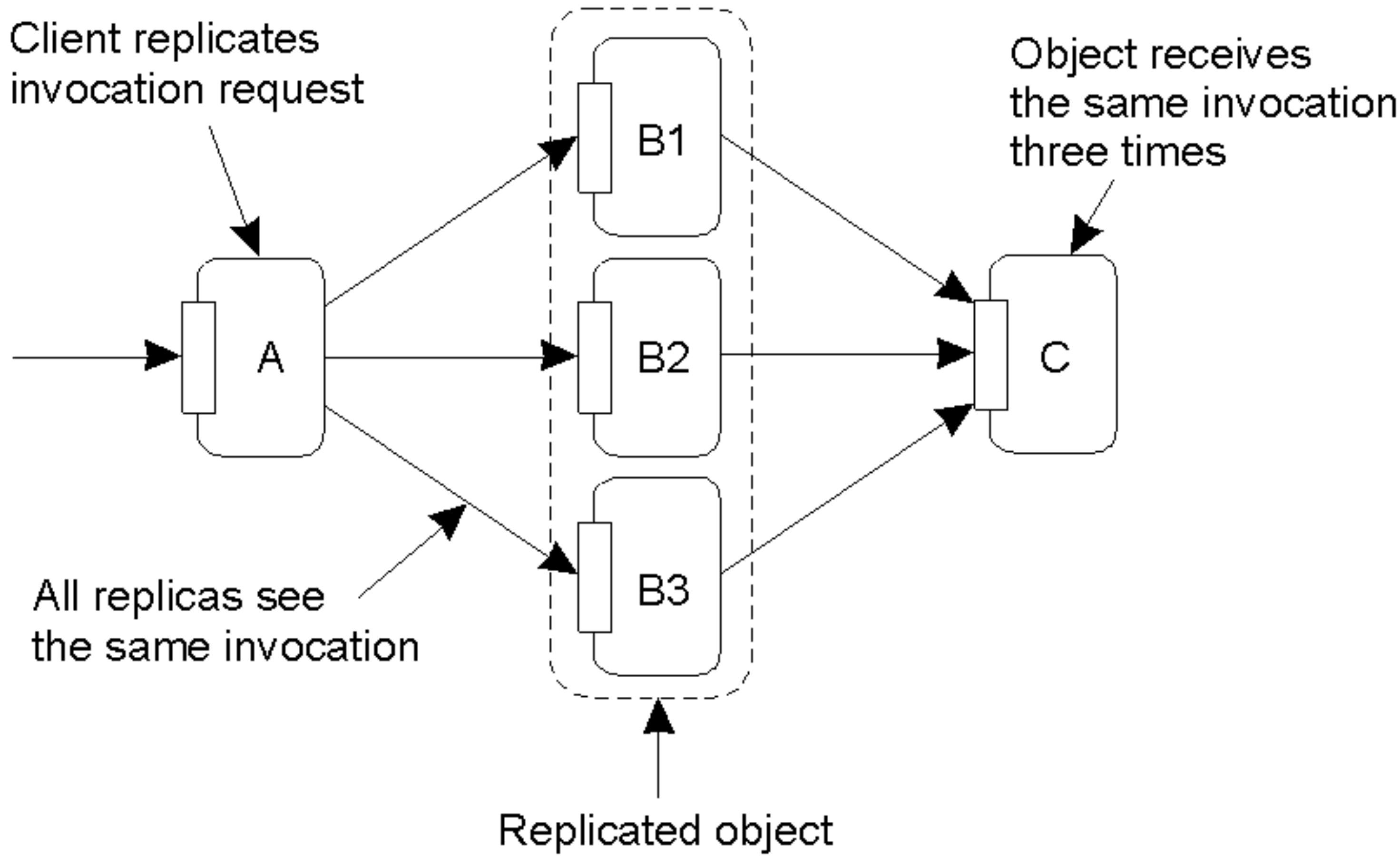
R1. Read request
R2. Response to read

- Primary-backup protocol in which the primary migrates to the process wanting to perform an update
- Advantage if nonblocking protocol: write operations carried locally, while reading can access local copies
- Protocol suitable for mobile computers

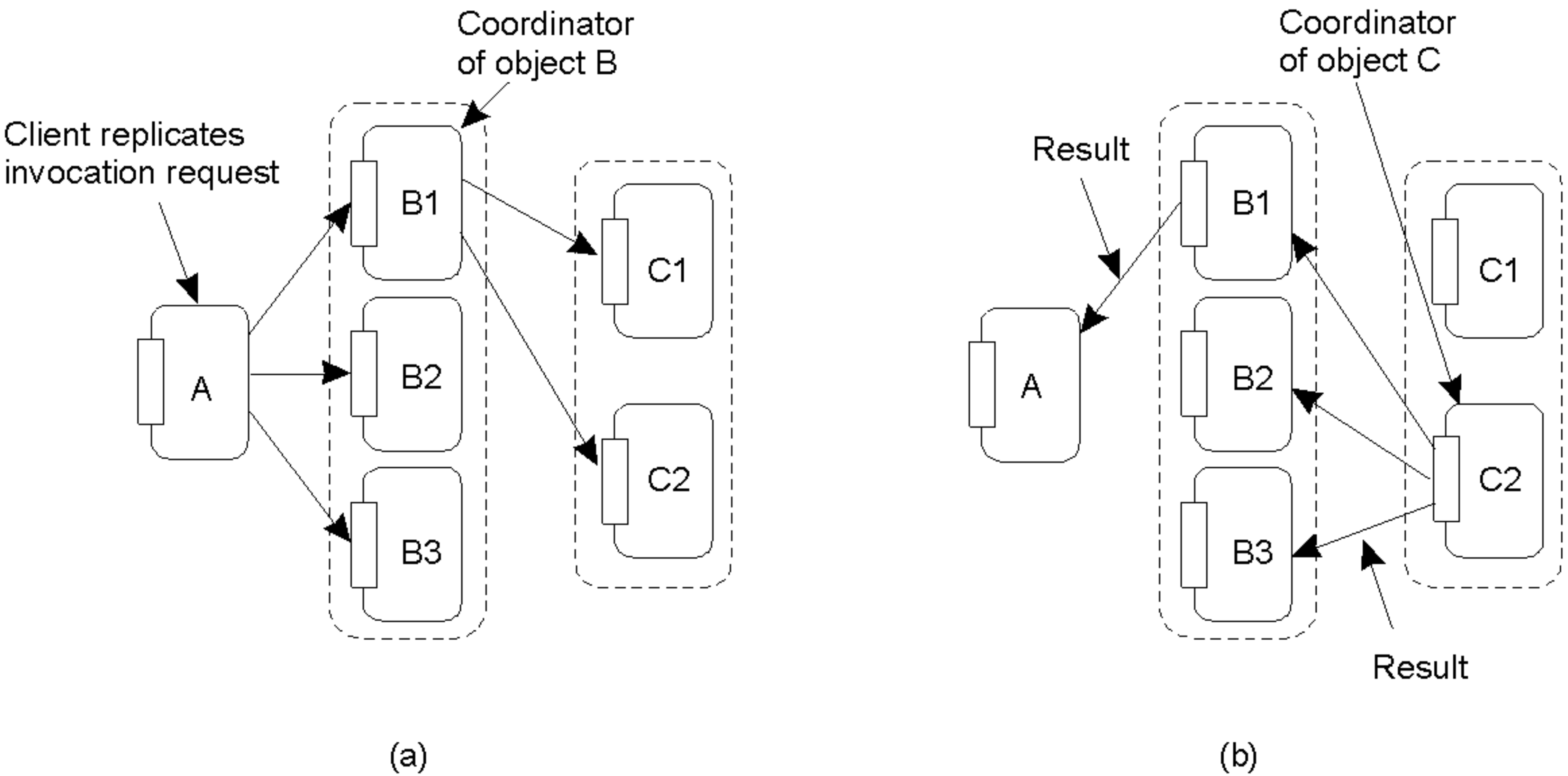
Replicated write protocols
Active Replication (1)

- Operations sent to each replica
- Operations have to be carried out in the same order everywhere
  - Need of totally-ordered multicast
    - Using Lamport timestamps
    - Using a central coordinator called sequencer
- Deal with replicated invocations

Active Replication (2)



Active Replication (3)



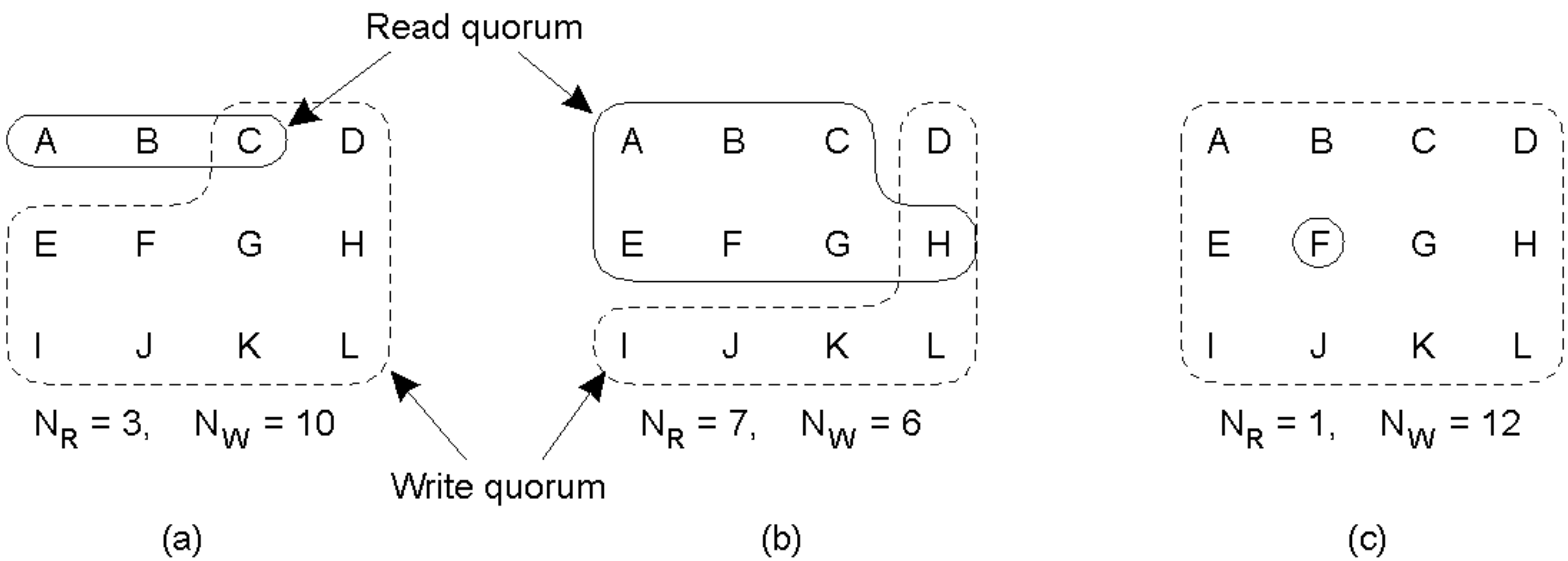
Quorum-Based Protocols (1)

- Use voting: clients request and acquire permission of multiple servers before reading/writing a replicated object
- Example distributed file system
  - File replicated on N servers
  - For an update a client must contact a majority of servers (half +1)
  - If agreement file changed and version number updated
  - For a read a client must contact at least half of servers+1 and ask them to send version numbers of the file
  - Choose the most recent version

Quorum-Based Protocols (2) -Gifford scheme

- A file with N replicas
- A read quorum (N_R servers) for reading the file
- A write quorum (N_W servers) for modifying the file
- $N_R + N_W > N$
- $N_W > N/2$

Quorum-Based Protocols (3)



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)

Pessimistic vs. optimistic replication (1)

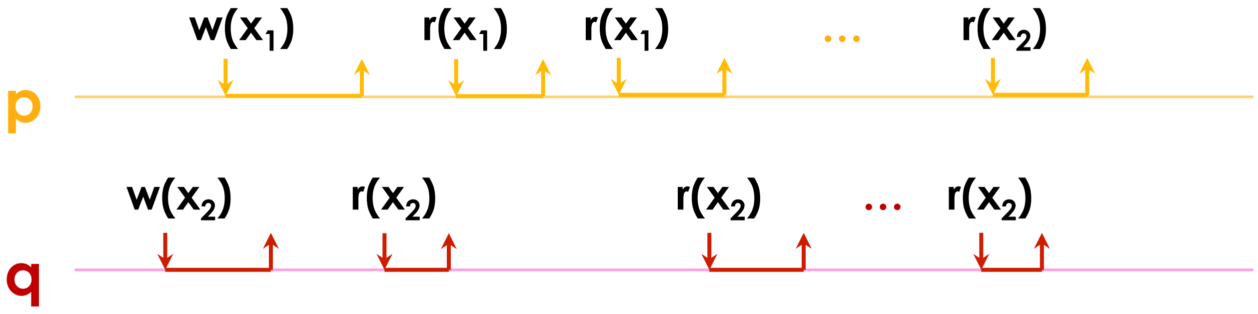
- Pessimistic replication
  - Give the illusion of one replica (no divergence)
  - Block access to a replica unless it is up-to-date
  - Example: primary-copy algorithms
    - Elect a primary replica
    - After an update primary writes the change to secondary replicas
    - If primary crashes elect a new replica
  - Bad performance and availability

Pessimistic vs. optimistic replication (2)

- Optimistic replication
  - Allows replicas to diverge
    - Commit modifications immediately and propagate later
    - Observers can see different values on different sites
  - Eventual consistency
  - Mandatory for offline access
  - Better scaling

Eventual Consistency

- A history is eventually consistent (EC) when for every object x if there is a bounded amount of write operations on x in h, then eventually all the read operations observe the same state



Strong Eventual Consistency

- **Eventual delivery**: An update executed at some correct replica eventually executes at all correct replicas

- **Strong convergence**: Correct replicas that have executed the same updates have equivalent states

- No consensus in background, no need to rollback

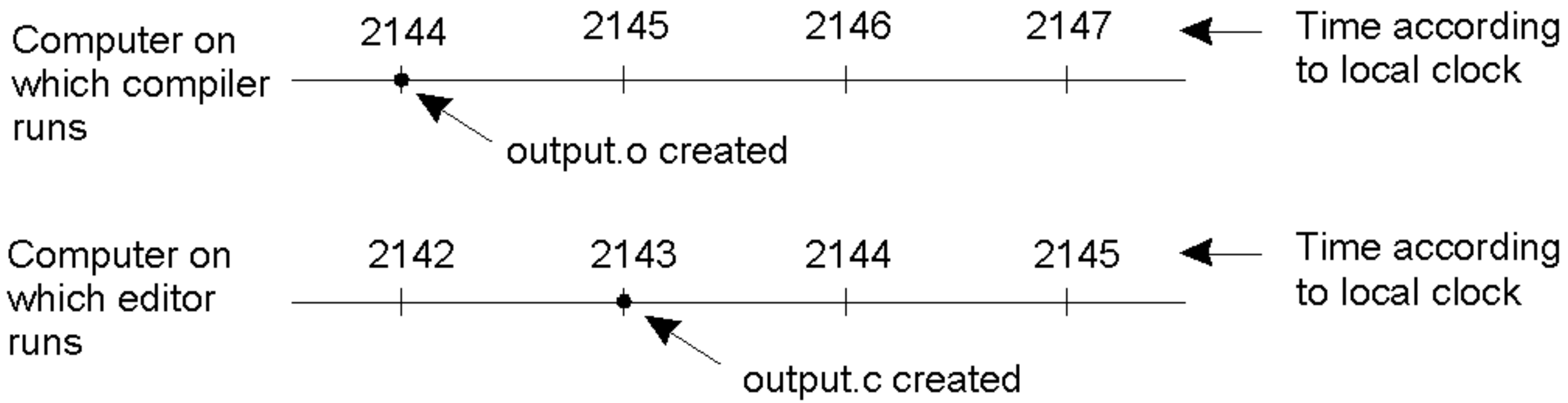
Pessimistic vs. optimistic replication (3)

- Basic principles of optimistic replication
  - N sites replicate an object
  - An object is modified by applying an operation
  - Local operations applied immediately
  - Operations broadcast to the other sites
  - Remote operations integrated and executed
  - System is correct if when it is idle all replicas are identical

Clock Synchronisation

- Time is unambiguous in a centralised system
- There is no global agreement on time in a distributed system
- Example
  - Program consisting of 100 files
  - Use of *make* to recompile only changed source files
  - If input.c has time 2151 and input.o has time 2150, then recompilation needed

Clock Synchronization



- make does not call the compiler

Logical clock

- Sufficient that all machines agree on the same time (not necessarily real time)
- Lamport 1978 – rather than agreeing on what time it is, sufficient to agree on the order in which events occur
- Previous example: if input.c is older or newer than input.o

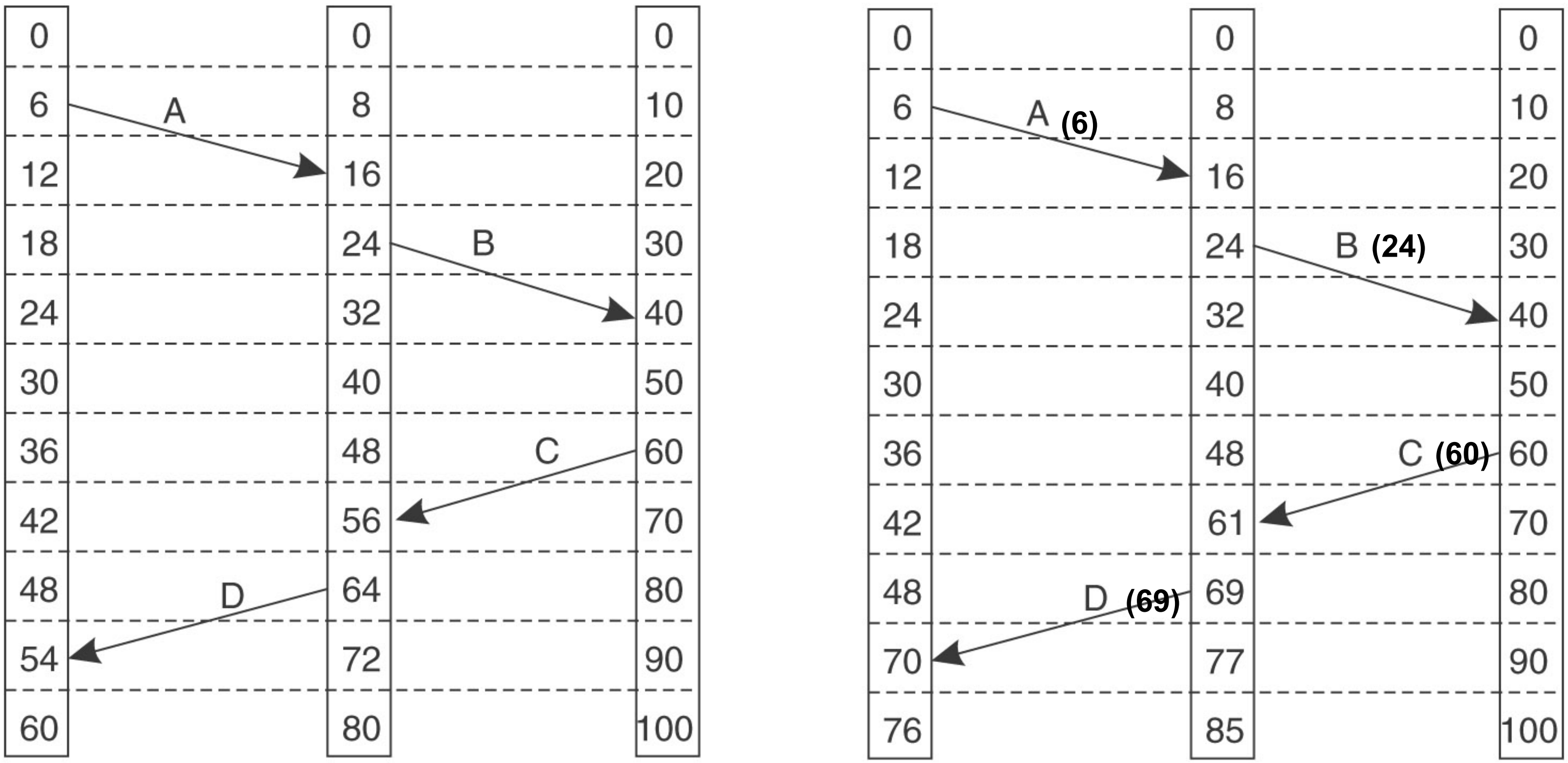
Lamport timestamps

- Happens-before relation
- *a*→*b* (*a* happens before *b*)
- Two situations:
  - If *a* and *b* are events in the same process and *a* occurs before *b*, then *a*→*b*
  - If *a* is the event of a message being sent by one process and b is the event of the message being received by another process, then *a*→*b*. A mesage cannot be received before or at the same time it is sent
- If *a*→*b* and *b*→*c* then *a*→*c*
- If neither *a*→*b* nor *b*→*a* then *a is concurrent with b*

Lamport timestamps

- For every event *a* assign *C(a)* on which all processes agree
- If $a \rightarrow b$ then $C(a) < C(b)$
- Clock time must always increase
- Lamport solution
  - Each message carries the sending time
  - If receiver clock < time of the arrived message, then receiver forwards its clock to 1 + sending time

Lamport timestamps



(a) (b)

Lamport timestamps

- If *a* happens before *b* in the same process then *C(a)<C(b)*
- If *a* and *b* represent the sending and receiving of a message, *C(a)<C(b)*
- For all distinctive events *a* and *b*, *C(a)≠C(b)*
  - Attach the number of the process to the lower order of the time
  - If *a* generated by process 1 at time 40 and *b* generated by process 2 at time 40, then *C(a)=40.1* and *C(a)=40.2*

Vector timestamps

- Lamport timestamps limits
  - if $C(a)<C(b)$ does not imply that $a \rightarrow b$
  - $a \parallel b$ does not imply $C(a)=C(b)$
- Example: posting articles and reactions to posted articles
- Lamport timestamps do not capture causality
- Vector timestamps capture causality
  - If VT(a)<VT(b), then a causally precedes b
  - Each process P_i maintains V_i
    - $V_i[i]$ = the no. of events that occured so far at P_i
    - If $V_i[j]=k$ then P_i knows that k events occurred at P_j

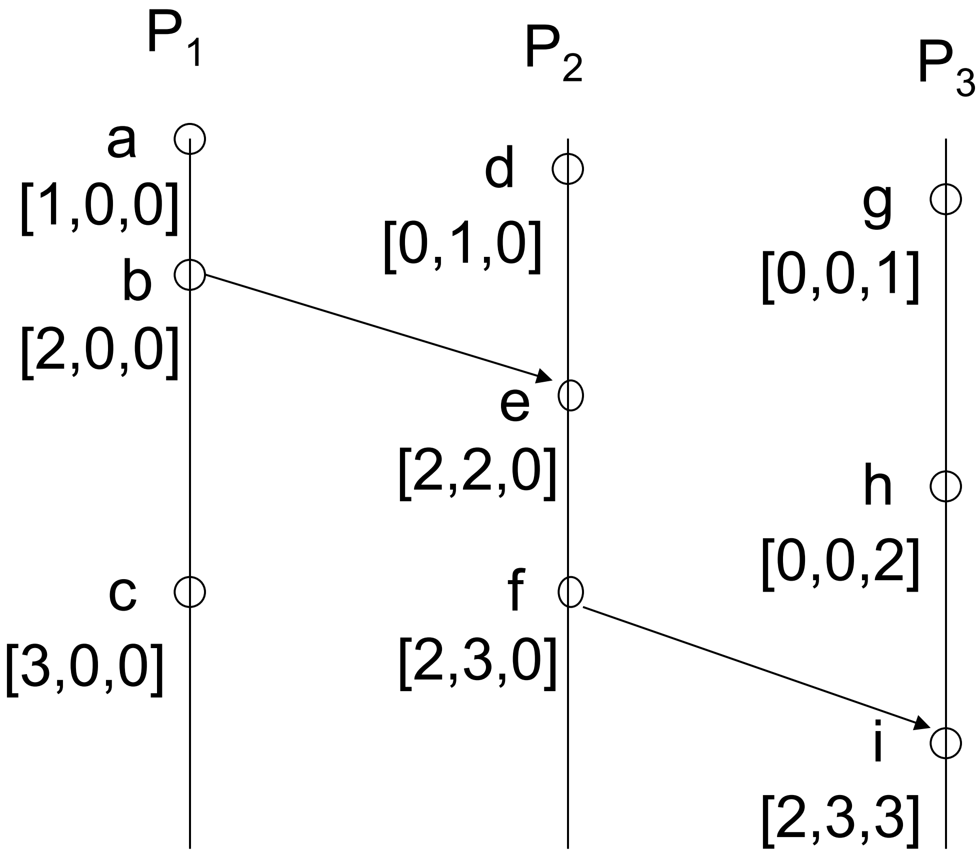
Vector timestamps

- Comparison of two vectors
  - $V=W$ iff $\forall$ i $V[i]=W[i]$
  - $V<W$ iff forall i $V[i] \leq W[i]$ and $\exists i$ $V[i]<W[i]$
  - $[1,2,0] < [3,2,1]$
  - $[0,1,1] \nless [1,0,1]$

Vector timestamps – computation rules

- Process Pi
  - Initialisation: $\forall k\ Vi[k]=0$
  - Local event: $Vi[i]= Vi[i]+1$
  - Sending message m : $Vi[i]= Vi[i]+1$, then send (m,Vi)
  - Receiving message (m,Vj):
    - $\forall\ k\ Vi[k]=\max(Vi[k], Vj[k])$
    - $Vi[i]=Vi[i]+1$

Vector timestamps – example



State vector