ECE151 – Lecture 10

Chapter 6

Consistency and Replication

Summary of 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)

a) Consistency models not using synchronization operations.
b) Models with synchronization operations.

Client-Centric Consistency Models

Goal: Show how we can perhaps avoid systemwide consistency, by concentrating on what specific *clients* want, instead of what should be maintained by servers.

Background: Most large-scale distributed systems (i.e., databases) apply replication for scalability, but can support only weak consistency:

DNS: Updates are propagated slowly, and inserts may not be immediately visible.

NEWS: Articles and reactions are pushed and pulled throughout the Internet, such that reactions might be seen before postings.

Lotus Notes: Geographically dispersed servers replicate documents, but make no attempt to keep (concurrent) updates mutually consistent.

WWW: Caches all over the place, but there need be no guarantee that you are reading the most recent version of a page.

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.

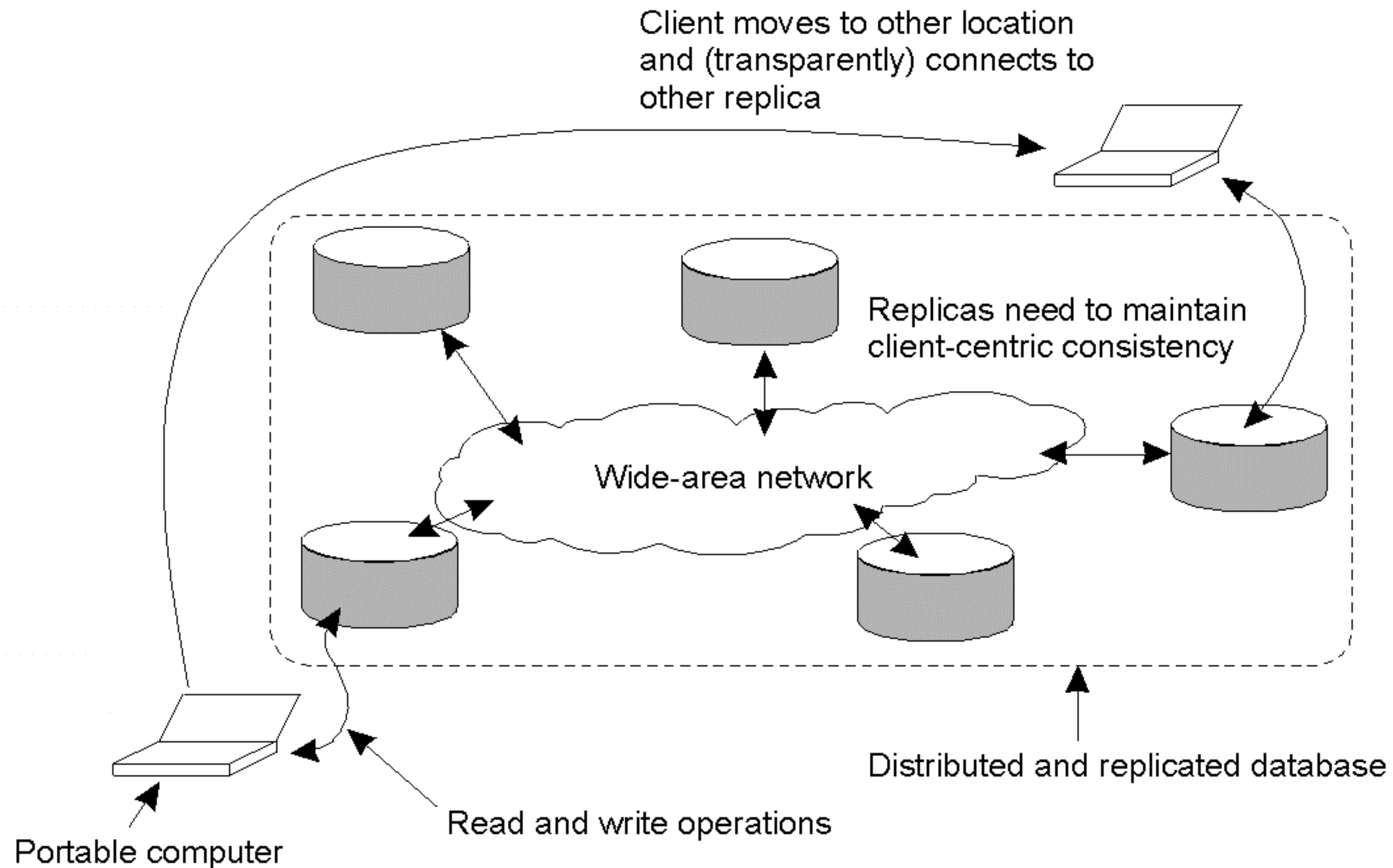
At location *A* you access the database doing reads and updates.

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*

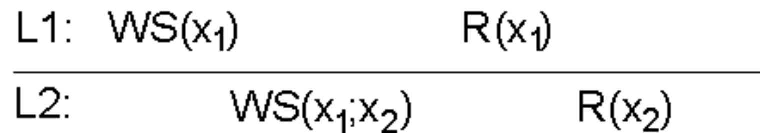
Note: The only thing you really want is that the entries you updated and/or read at *A*, are in *B* the way you left them in *A*. In that case, the database will appear to be consistent *to you*.

Eventual Consistency

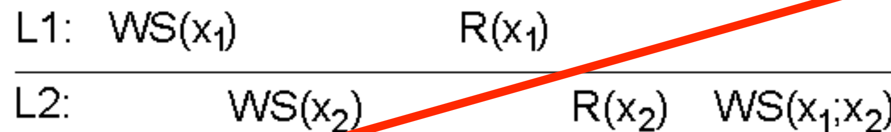


Monotonic Reads

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.



(a)



(b)

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.

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

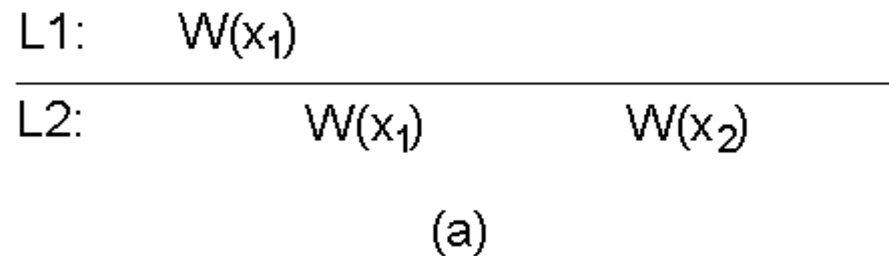
Monotonic Reads

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.

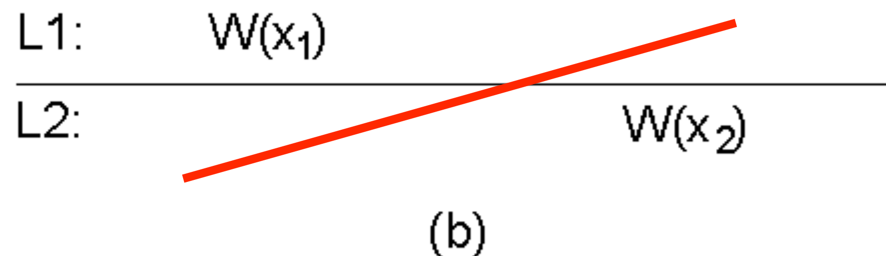
Example: Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

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.



(a)



(b)

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.

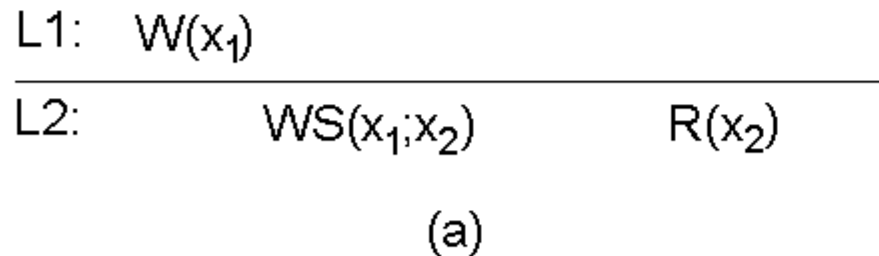
Monotonic Writes

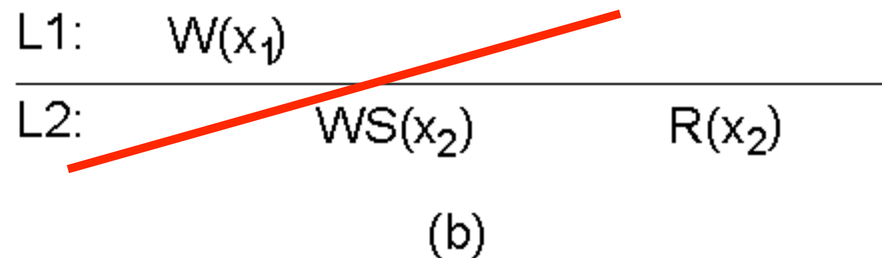
Example: Updating a program at server *S*2, and ensuring that all components on which compilation and linking depends, are also placed at *S*2.

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.





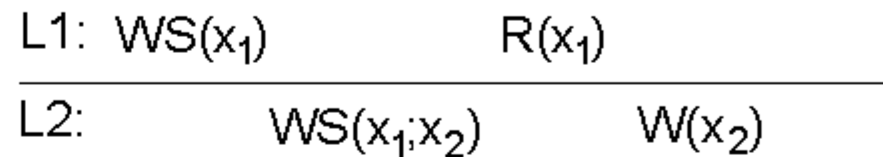
a) A data store that provides read-your-writes consistency.
b) A data store that does not.

Read Your Writes

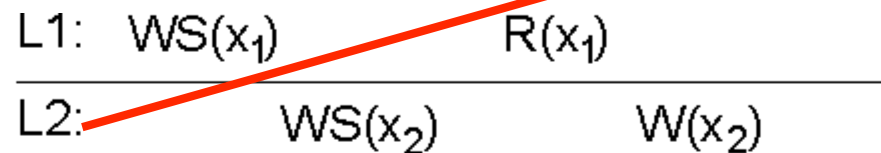
Example: Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.

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.



(a)



(b)

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

Writes Follow Reads

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

Replica Placement

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

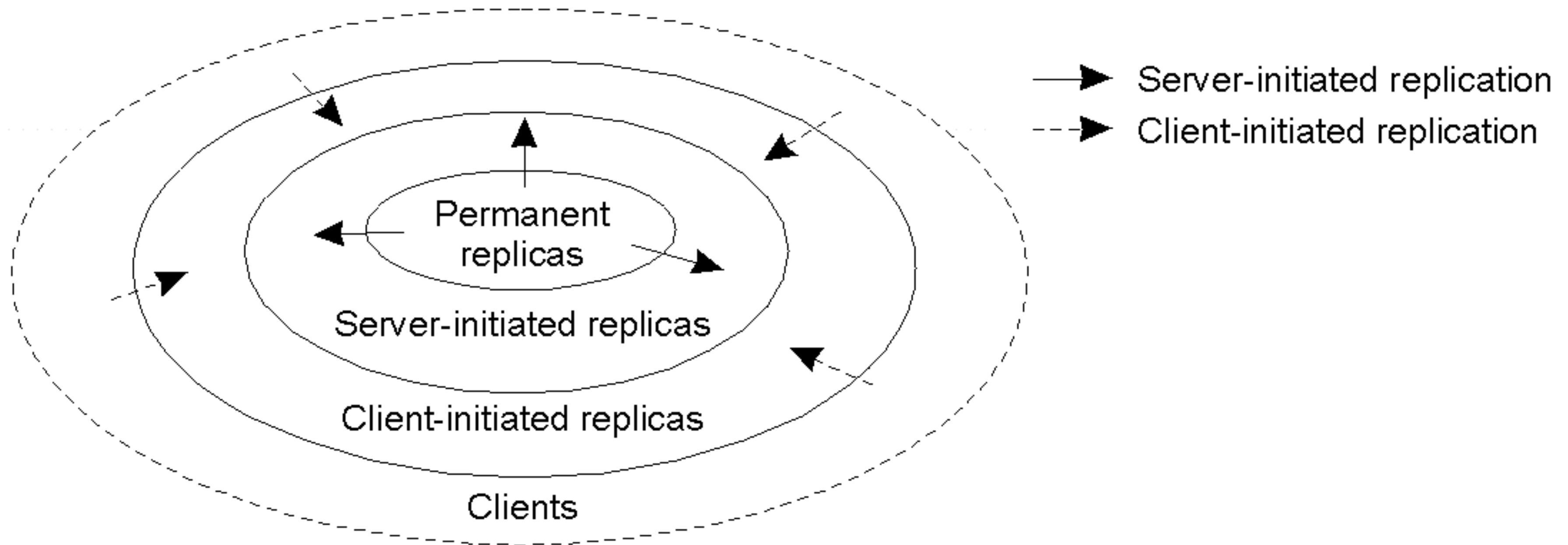
Distinguish different processes: A process is capable of hosting a replica of an object or data:

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)

Replica Placement



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

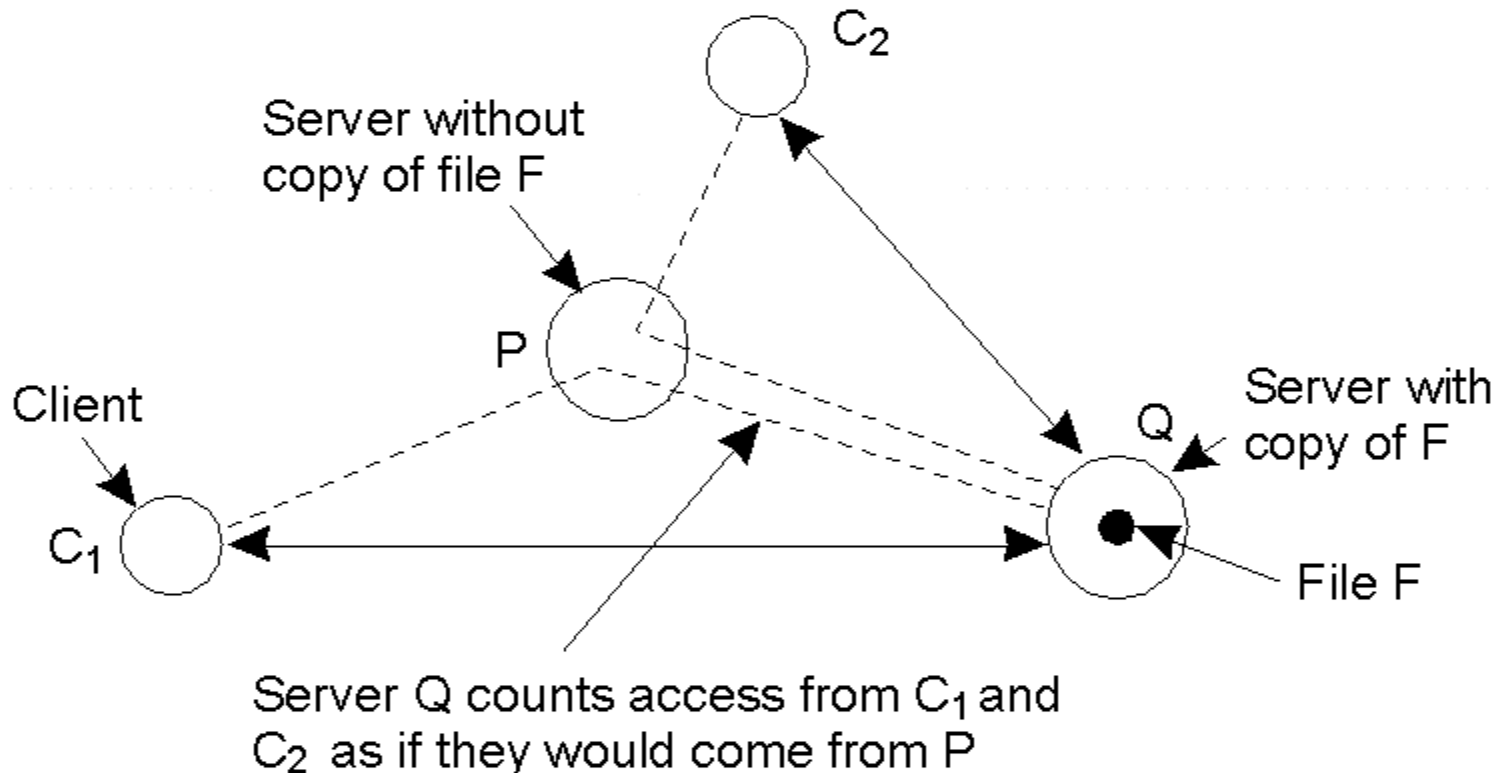
Server-Initiated Replicas

Keep track of access counts per file, aggregated by considering server closest to requesting clients

Number of accesses drops below threshold *D* drop file

Number of accesses exceeds threshold *R* replicate file

Number of access between *D* and *R* migrate file



Update Propagation

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.

Pull versus Push Protocols

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 that the update be sent to it.

Issue	Push-based	Pull-based
State of 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

A comparison between push-based and pull-based protocols in the case of multiple client, single server systems.

Update Propagation

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

Epidemic Algorithms

General background

Update models

Removing objects

Principles

Basic idea: Assume there are no write–write conflicts:

- Update operations are initially performed at one or only a few replicas
- A replica passes its updated state to a limited number of neighbors
- Update propagation is lazy, i.e., not immediate
- Eventually, each update should reach every replica

Anti-entropy: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards

Gossiping: A replica which has just been updated (i.e., has been **contaminated**), tells a number of other replicas about its update (contaminating them as well).

System Model

We consider a collection servers, each storing a number of objects

Each object O has a *primary* server at which updates for O are always initiated (avoiding write-write conflicts)

An update of object O at server S is timestamped

The value of O at S is denoted $VAL(O, S)$

$T(O,S)$ denotes the timestamp of the value of object O at server S

Anti-Entropy

Basic issue: When a server S contacts another server S^* to exchange state information, three different strategies can be followed:

Push: S only forwards all its updates to S^* :

if $T(O,S^*) < T(O,S)$ then $VAL(O,S^*) \Leftarrow VAL(O,S)$

Pull: S only fetches updates from S :

if $T(O,S^*) > T(O,S)$ then $VAL(O,S^*) \Leftarrow VAL(O,S)$

Push-Pull: S and S exchange their updates by pushing and pulling values

Observation: if each server periodically randomly chooses another server for exchanging updates, an update is propagated in $O(\log(N))$ time units.

Question: Why is pushing alone not efficient when many servers have already been updated?

Gossiping

Basic model: A server S having an update to report, contacts other servers. If a server is contacted to which the update has already propagated, S stops contacting other servers with probability 1 k.

If s is the fraction of ignorant servers (i.e., which are unaware of the update), it can be shown that with many servers:

$$s = e^{-(k+1)(1-s)}$$

k	s
1	0.2000
2	0.0600
3	0.0200
4	0.0070
5	0.0025

Observation: If we really have to ensure that all servers are eventually updated, gossiping alone is not enough

Deleting Values

Fundamental problem: We cannot remove an old value from a server and expect the removal to propagate.

Instead, mere removal will be undone in due time using epidemic algorithms

Solution: Removal has to be registered as a special update by inserting a *death certificate*

Next problem: When to remove a death certificate (it is not allowed to stay for ever):

Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection)

Assume death certificates propagate in finite time, and associate a maximum lifetime for a certificate (at risk of not reaching all servers)

Note: it is necessary that a removal actually reaches all servers.

Question: What's the scalability problem here?

Consistency Protocols

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

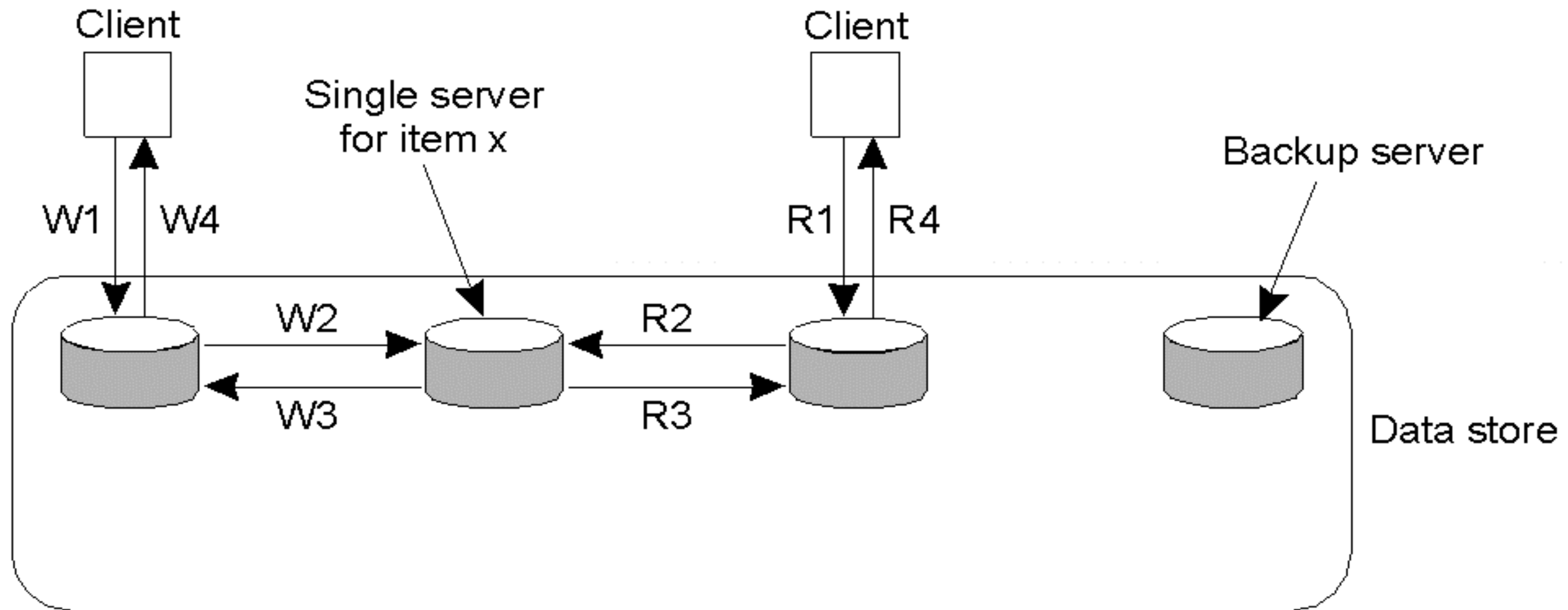
We will concentrate only on sequential consistency.

- Primary-based protocols
- Replicated-write protocols
- Cache-coherence protocols

Primary-Based Remote-Write Protocols

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

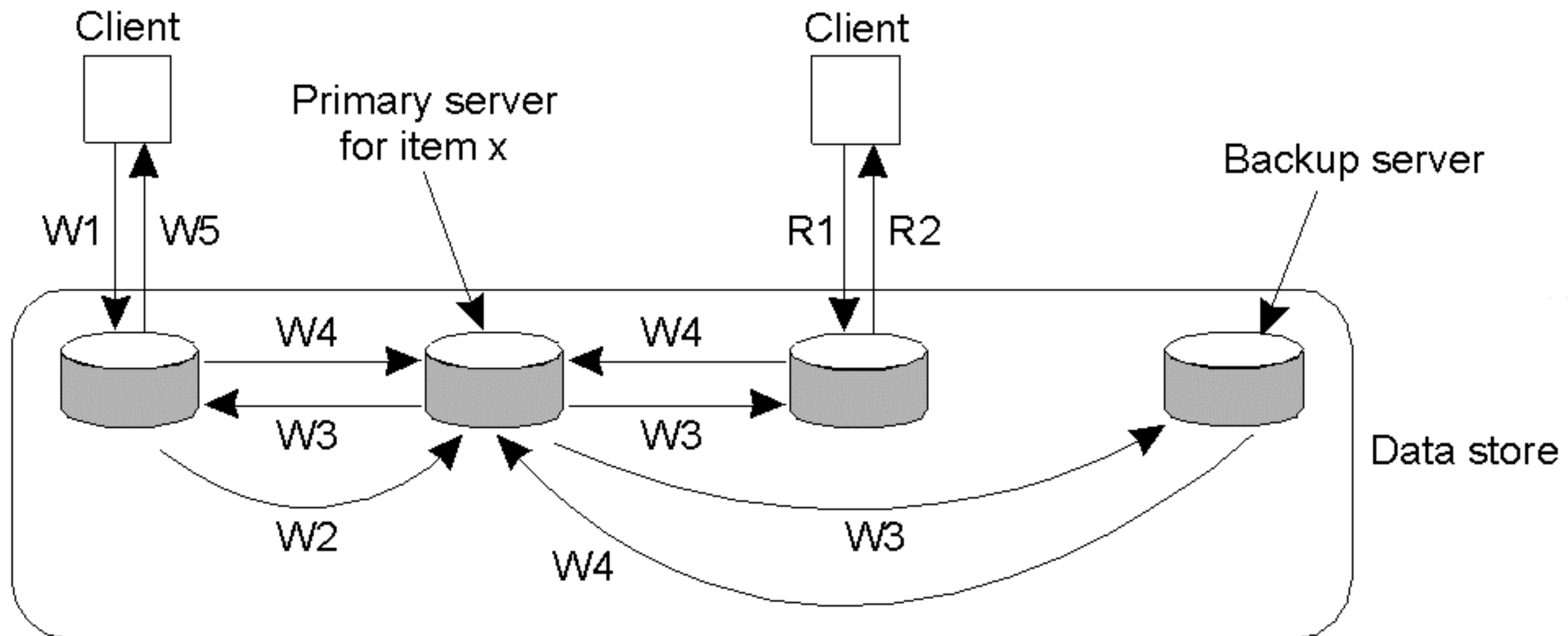
Used in traditional client-server systems that do not support replication.



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 Protocols



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

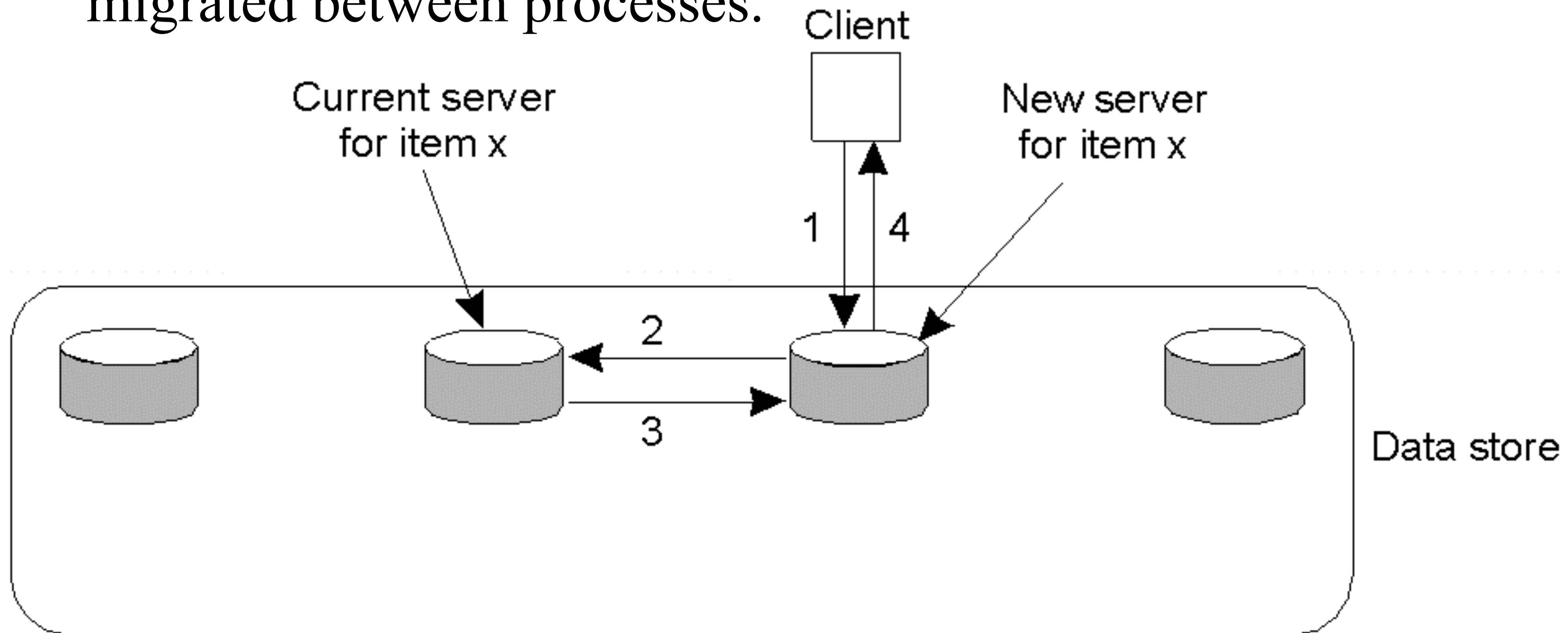
R1. Read request
R2. Response to read

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 Local-Write Protocols

Primary-based local-write protocol in which a single copy is migrated between processes.

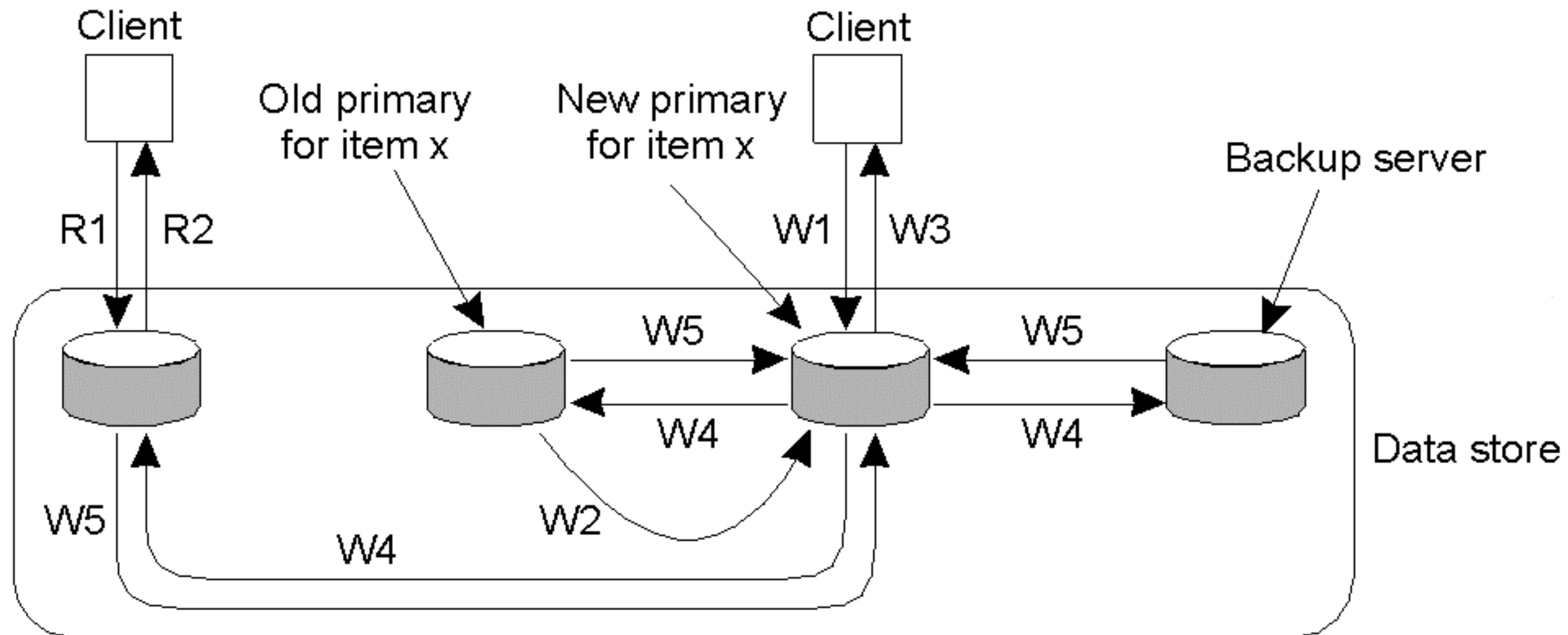


1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

Establishes only a fully distributed, nonreplicated data store. Useful when writes are expected to come in series from the same client (e.g., mobile computing without replication)

Primary-Based Local-Write Protocols

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

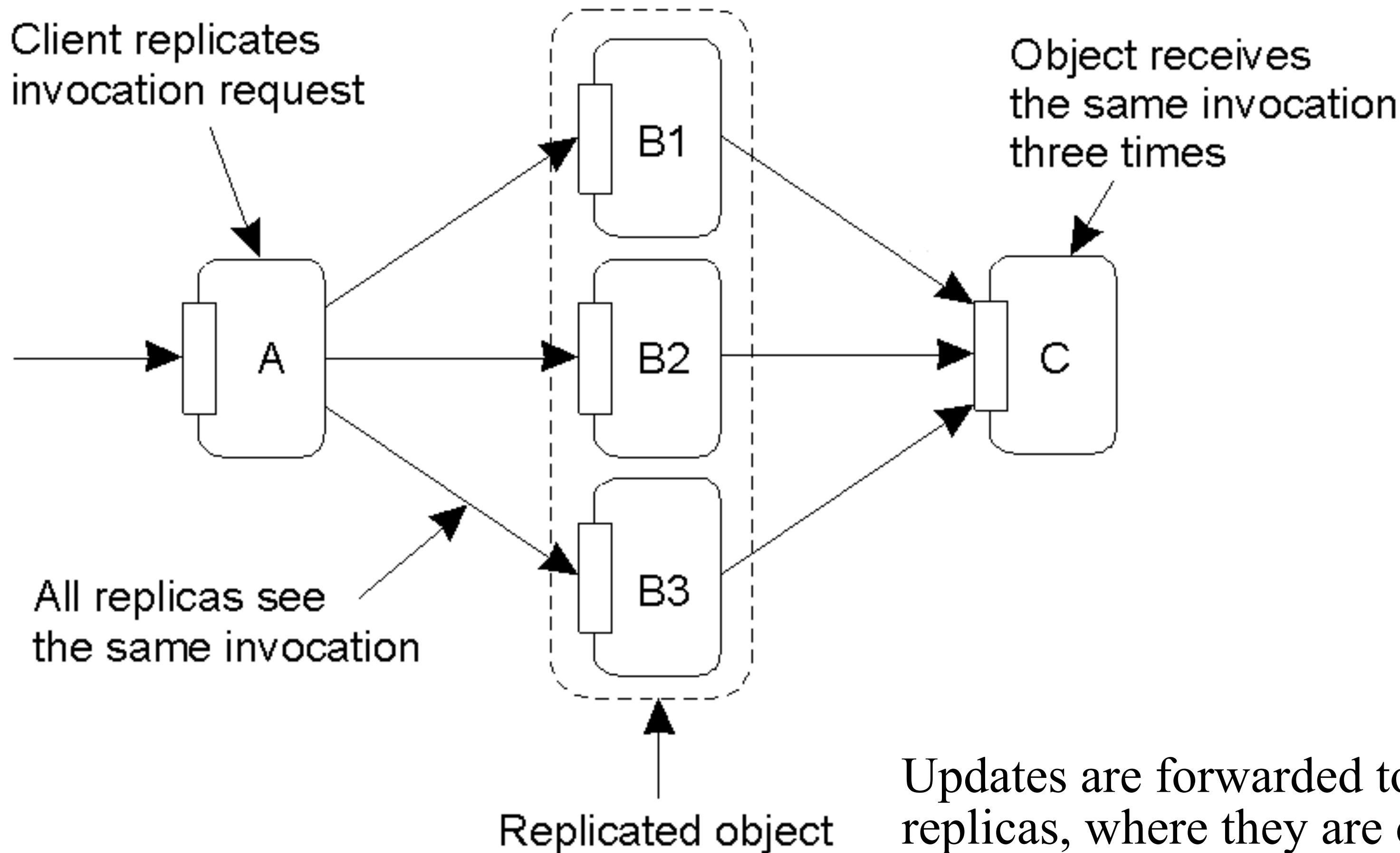


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

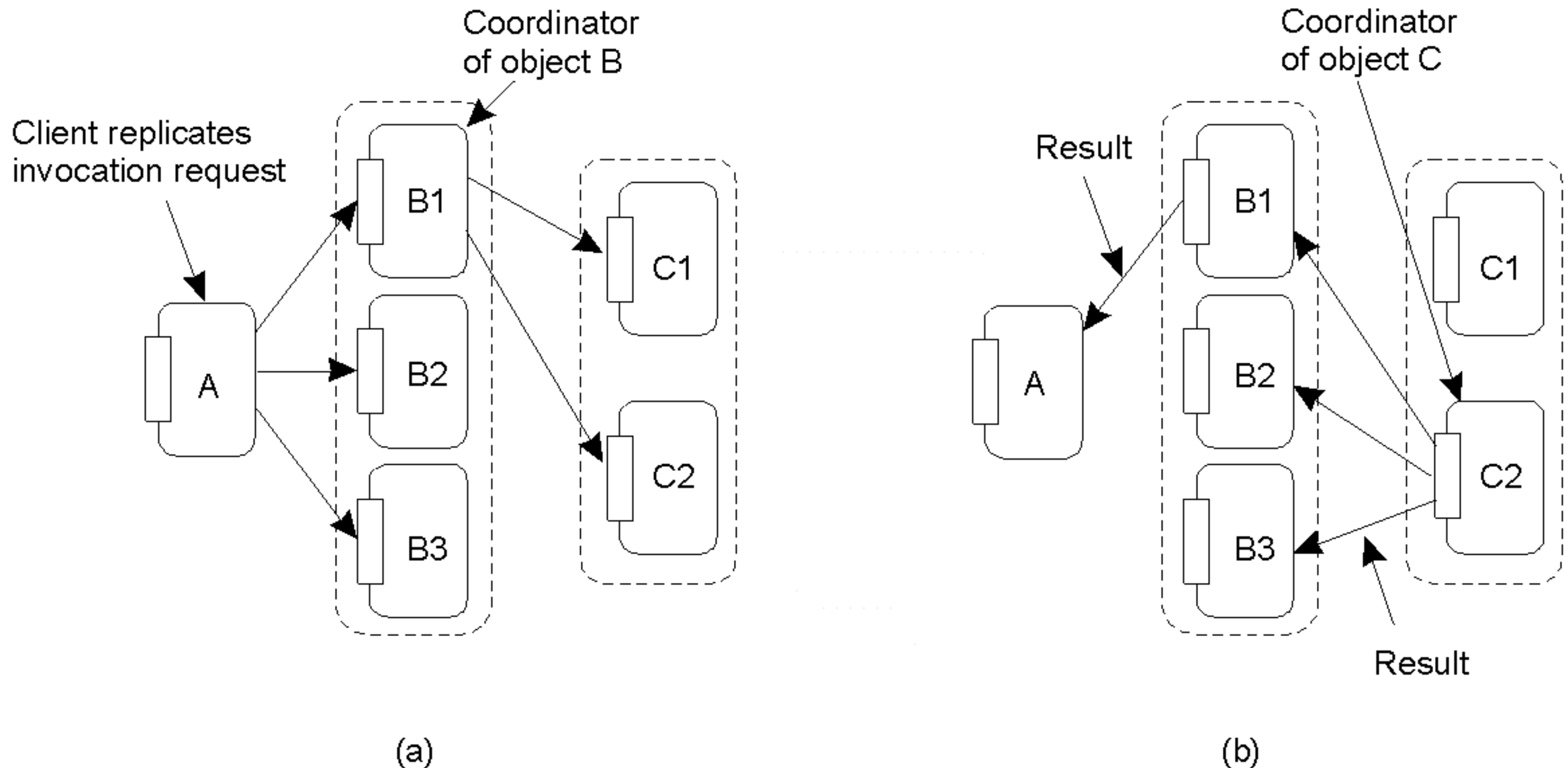
Distributed shared memory systems, but also mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later on).

Active Replication



Updates are forwarded to multiple replicas, where they are carried out. There are some problems to deal with in the face of replicated invocations:

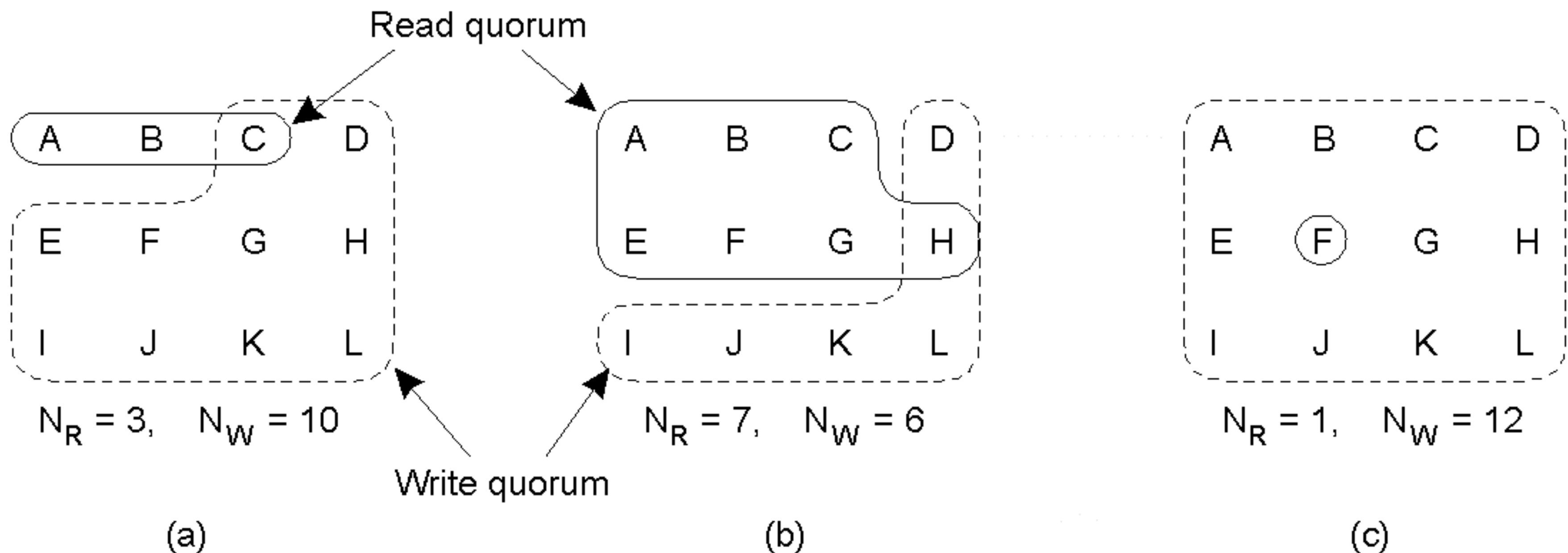
Active Replication



Replicated invocations: Assign a coordinator on each side (client and server), which ensure that only one invocation, and one reply, is sent:

Quorum-Based 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**:



Three examples of the voting algorithm:

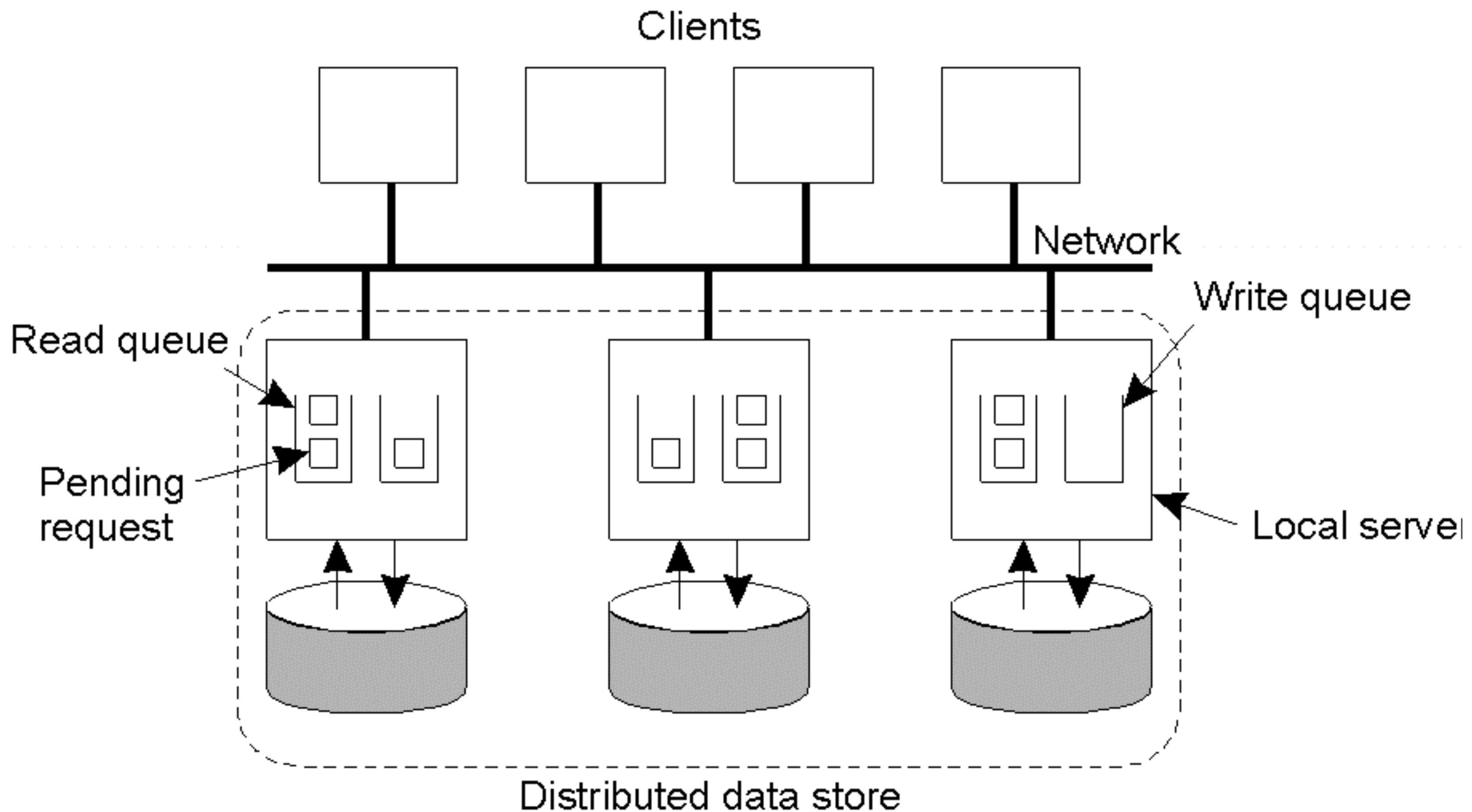
a) A correct choice of read and write set

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

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

Casually-Consistent Lazy Replication

Basic model: Number of replica servers jointly implement a causal-consistent data store. Clients normally talk to **front ends** which maintain data to ensure causal consistency.



Lazy Replication: Vector Timestamps

VAL(i): VAL(i)[i] denotes the total number of write operations sent directly by a front end (client). VAL(i)[j] denotes the number of updates sent from replica #j.

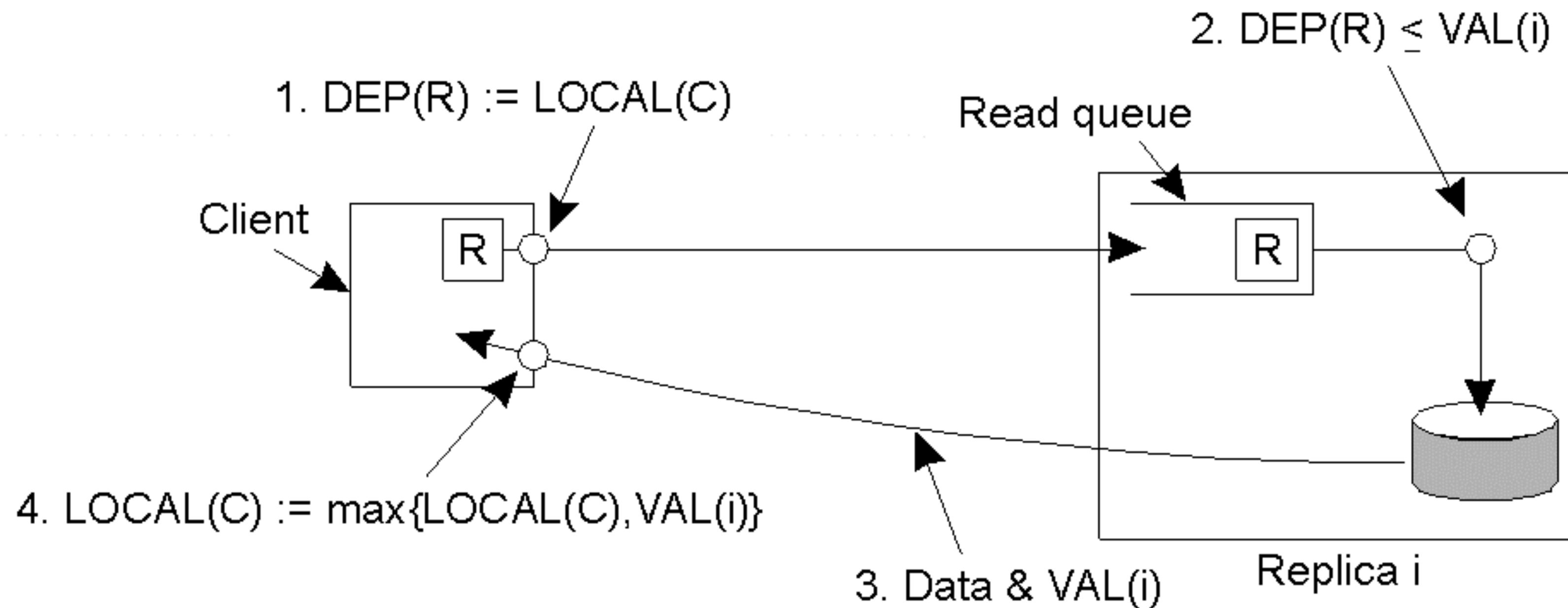
WORK(i): WORK(i)[i] total number of write operations directly from front ends, including the pending ones. WORK(i)[j] is total number of updates from replica #j, including pending ones.

LOCAL(C): LOCAL(C)[j] is (almost) most recent value of VAL(j)[j] known to front end C (will be refined in just a moment)

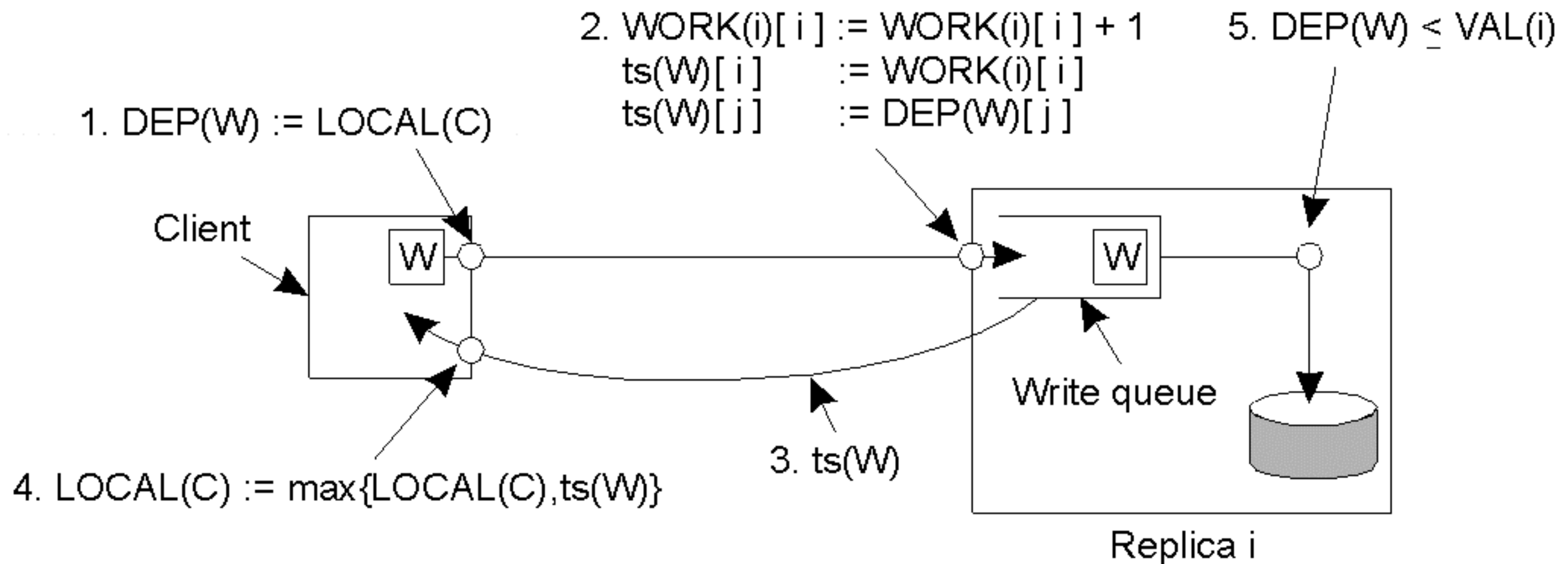
DEP(R): Timestamp associated with a request, reflecting what the request depends on.

Processing Read Operations

Performing a read operation at a local copy.



Processing Write Operations



Performing a write operation at a local copy.