Consistency and Replication

Why Replicate Data?

- Enhance reliability.
  - guarantees correct behaviour in spite of certain faults (can include timeliness)
  - While at least one server has not crashed, the service can be supplied
- Improve performance.
  - e.g. several web servers can have the same DNS name and the servers are selected in turn. To share the load.
    - replication of read-only data is simple, but replication of changing data has overheads

More on Replication

- Replicas allows remote sites to continue working in the event of local failures.
- It is also possible to protect against data corruption.
- Replicas allow data to reside close to where it is used.
- This directly supports the distributed systems
- goal of enhanced *scalability*.
- Even a large number of replicated “local” systems can improve performance: think of clusters.

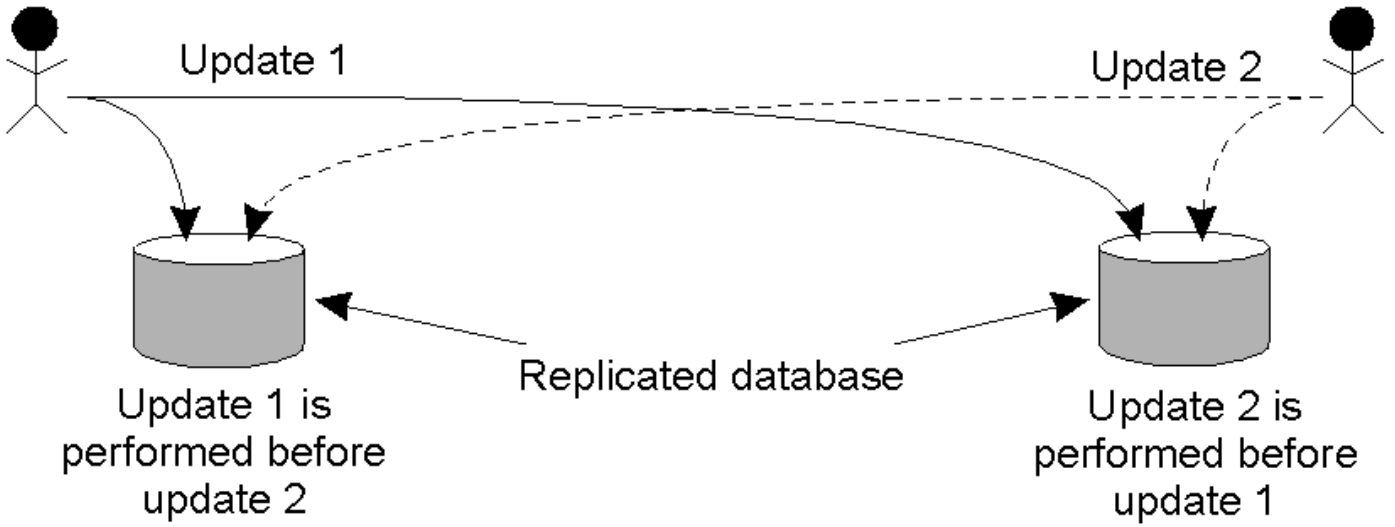
Replication and Scalability

- Replication is a widely-used scalability technique: think of web clients and web proxies.
- When systems scale, the first problems to surface are those associated with performance – as the systems get bigger (e.g., more users), they get often slower.
- Replicating the data and moving it closer to where it is needed helps to solve this scalability problem.

So, what's the catch?

It is not easy to keep all those replicas *consistent*.

What Can Go Wrong



- Updating a replicated database: Update 1 adds 100 euro to an account, Update 2 calculates and adds 1% interest to the same account.
- Due to network delays, the updates may come in different order!
- Inconsistent state

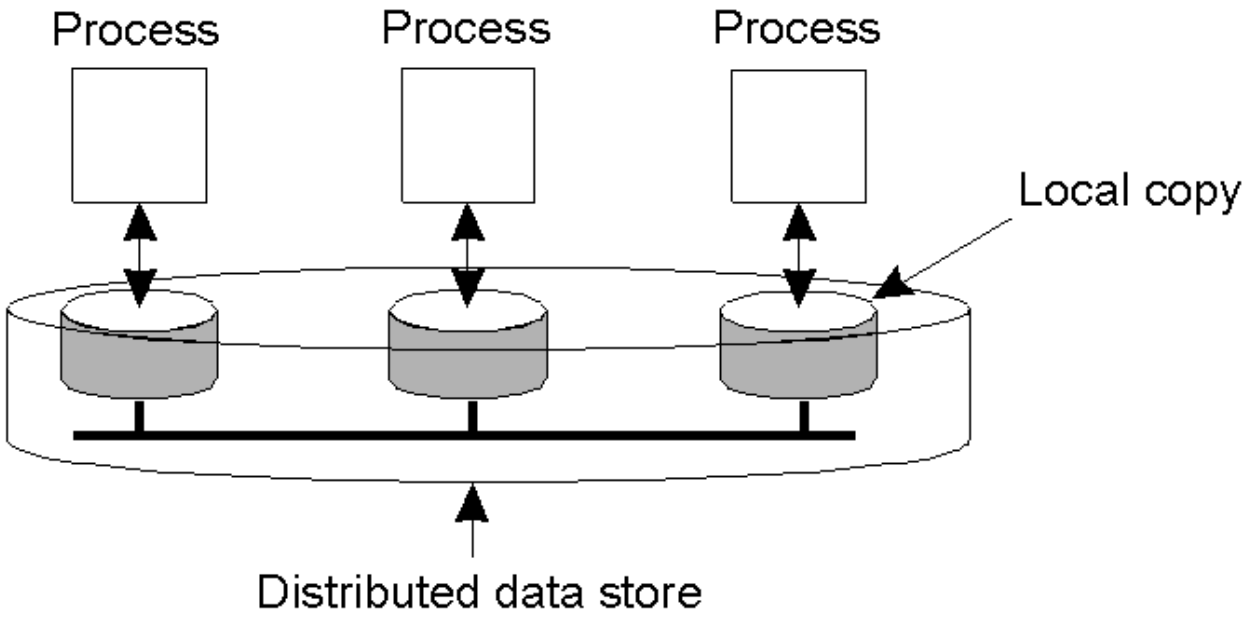
Requirements for replicated data

What is replication transparency?

- Replication transparency
  - clients see logical objects (not several physical copies)
  - they access one logical item and receive a single result
- Consistency
  - specified to suit the application,
    - e.g. when a user of a diary disconnects, their local copy may be inconsistent with the others and will need to be reconciled when they connect again. But connected clients using different copies should get consistent results. These issues are addressed in Bayou and Coda.

Data-Centric Consistency Models

- A data-store can be read from or written to by any process in a distributed system.
  - A local copy of the data-store (replica) can support “fast reads”.
- However, a write to a local replica needs to be propagated to *all* remote replicas.



- Various consistency models help to understand the various mechanisms used to achieve and enable this.

What We Study Here?

- How to express what is happening in a system
  - Guaranteeing global ordering is a costly operation, downgrades scalability
  - Maybe, for some applications *total* ordering is an overkill?
    - Weaker consistency requirements
- How to make replicas consistent
  - Update protocols

What is a Consistency Model?

A “consistency model” is a CONTRACT between a DS data-store and its processes.

If the processes agree to the rules, the data-store will perform properly and as advertised.

Consistency Models

No explicit synchronization operations

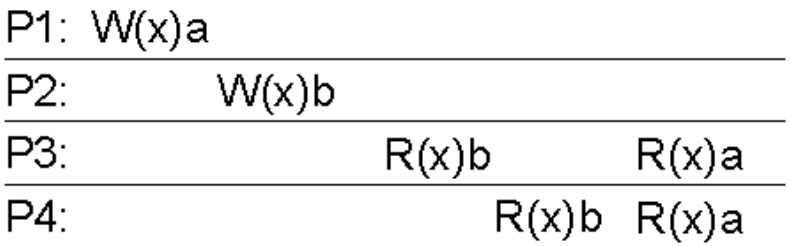
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in 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.

With explicit synchronization operations

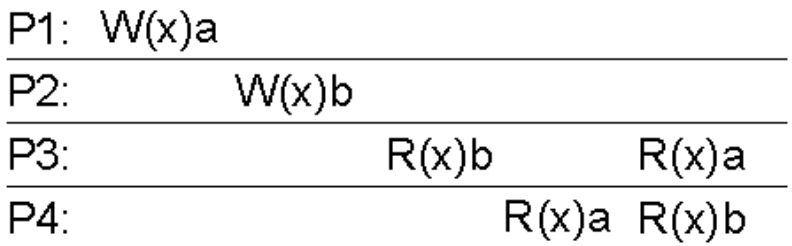
Weak	Shared data can be counted on to be consistent only after a synchronization is done.
Release	Shared data are made consistent when a critical region is exited.
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

Example: Sequential Consistency

All processes see the same interleaving set of operations, regardless of what that interleaving is.



(a)



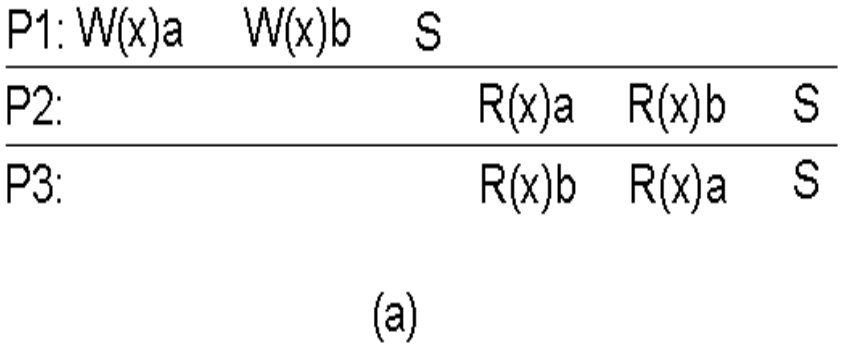
(b)

a) A sequentially consistent data-store – the "first" write occurred *after* the "second" on all replicas.

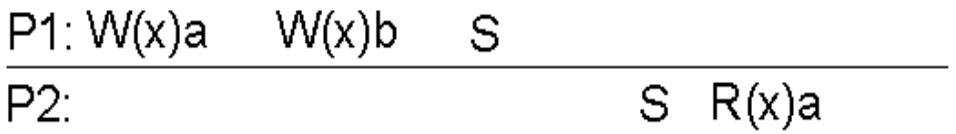
b) A data-store that is not sequentially consistent – it appears the writes have occurred in a non-sequential order, and this is NOT allowed.

Example: Weak Consistency

- In some cases we do not need to synchronize on *all* data



(a)



(b)

a) A valid sequence of events for weak consistency. This is because P2 and P3 have yet to synchronize, so there's no guarantees about the value in 'x'.

b) An invalid sequence for weak consistency. P2 has synchronized, so it cannot read 'a' from 'x' – it should be getting 'b'.

Client-Centric Consistency Models

- The previously studied consistency models concern themselves with maintaining a consistent (globally accessible) data-store in the presence of concurrent read/write operations.

- Another class of distributed datastore is that which is characterized by *the lack of simultaneous updates*. Here, the emphasis is more on maintaining a consistent view of things *for the individual client process* that is currently operating on the data-store.

More Client-Centric Consistency

How fast should updates (writes) be made available to read-only processes?

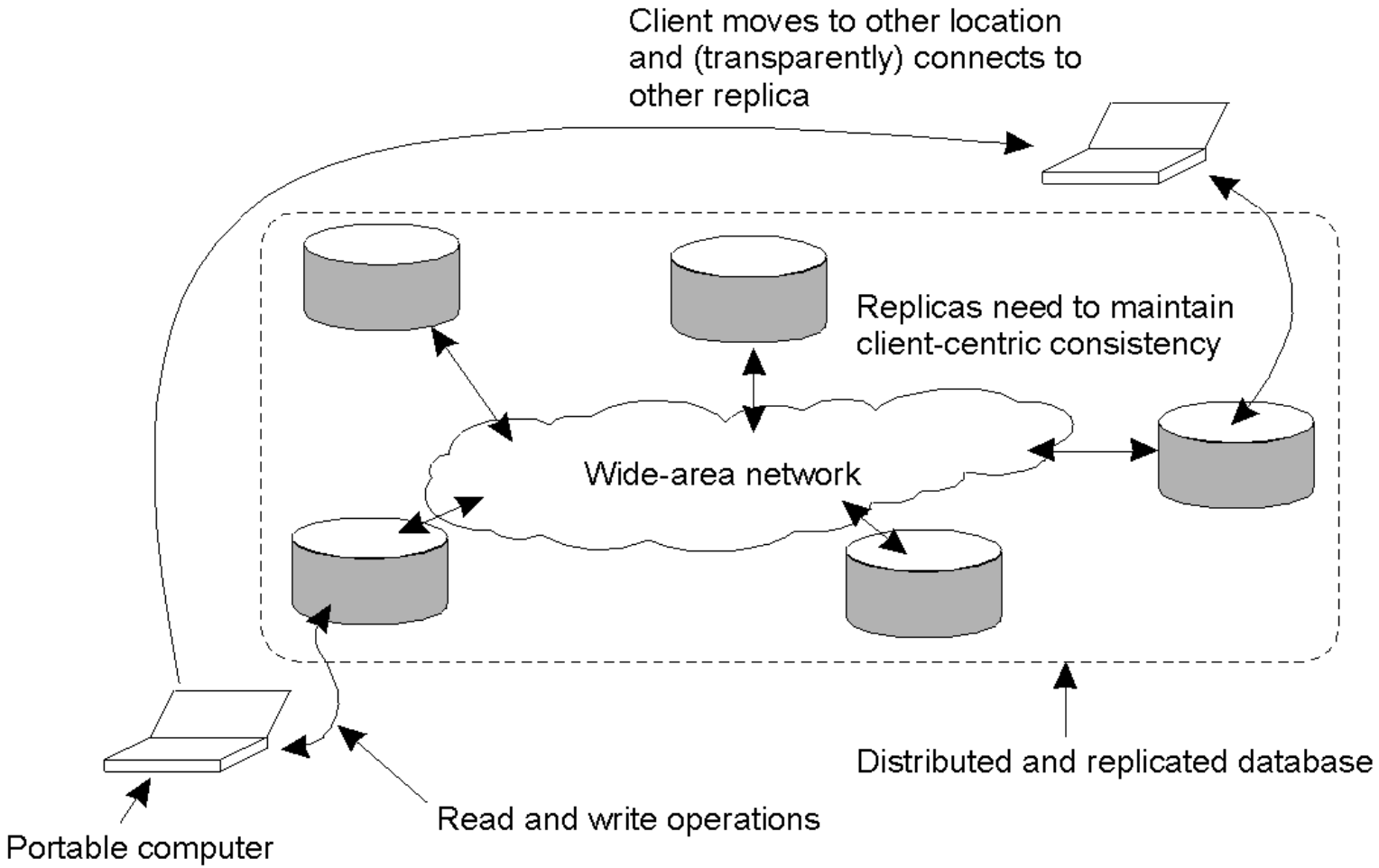
- Think of most database systems: *mainly read*.
- Think of the DNS: *write-write conflicts* do no occur.
- Think of WWW: as with DNS, except that heavy use of client-side caching is present: *even the return of stale pages is acceptable to most users*.

These systems all exhibit a high degree of acceptable inconsistency … with the *replicas* gradually become consistent over time.

Toward Eventual Consistency

- The only requirement is that all replicas will *eventually* be the same.
- All updates must be guaranteed to propogate to all replicas … *eventually*!
- This works well if every client always updates the same replica.
- Things are a little difficult if the clients are *mobile*.

Eventual Consistency: Mobile Problems

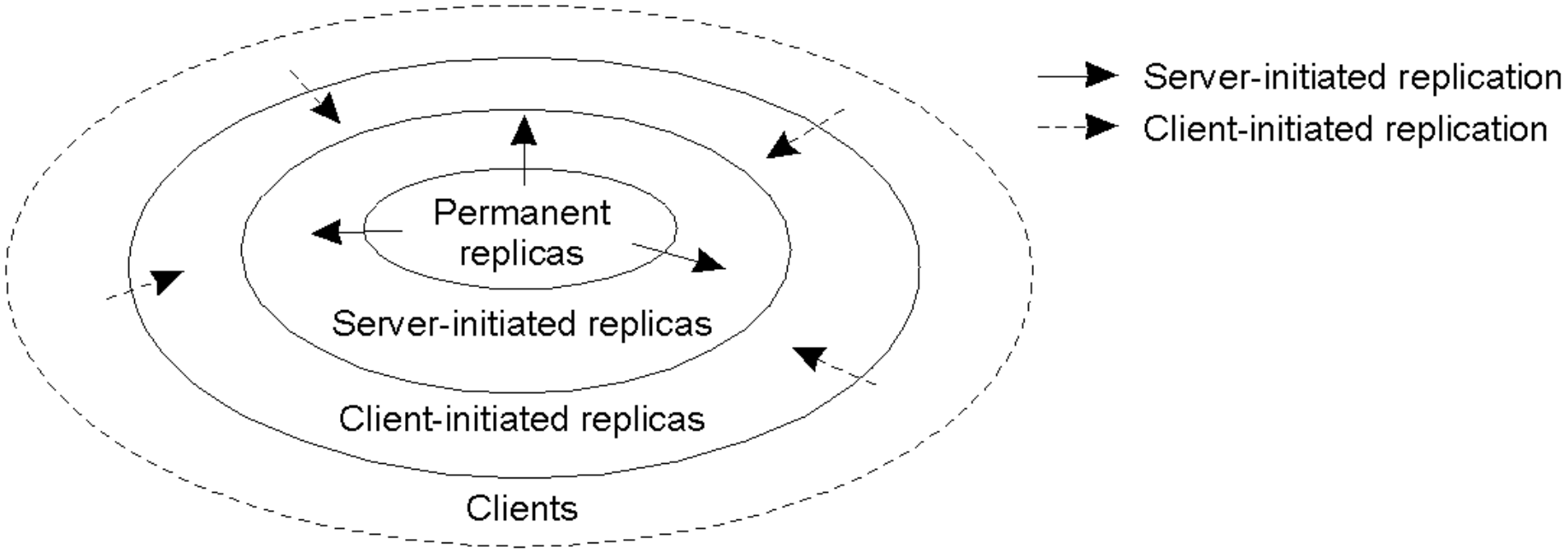


The principle of a mobile user accessing different replicas of a distributed database.

When the system can guarantee that a single client sees accesses to the data-store in a consistent way, we then say that “client-centric consistency” holds.

Distribution Protocols

Regardless of which consistency model is chosen, we need to decide ***where***, ***when*** and ***by whom*** copies of the data-store are to be placed.



Replica Placement Types

There are three types of replica:

1. ***Permanent replicas***: tend to be small in number, organized as COWs (Clusters of Workstations) or mirrored systems. (Think of Web-servers or backup database servers)
2. ***Server-initiated replicas***: used to enhance performance at the initiation of the owner of the data-store. Typically used by web hosting companies to geographically locate replicas close to where they are needed most. (Often referred to as "push caches").
3. ***Client-initiated replicas***: created as a result of client requests – think of browser caches. Works well assuming, of course, that the cached data does not go *stale* too soon.

Consistency and Replication: Summary

- Reasons for replication: improved *performance*, improved *reliability*.
- Replication can lead to *inconsistencies* …
- How best can we *propagate updates* so that these inconsistencies are not noticed?
- With “best” meaning “without crippling performance”.
- The proposed solutions resolve around the relaxation of any existing consistency constraints.

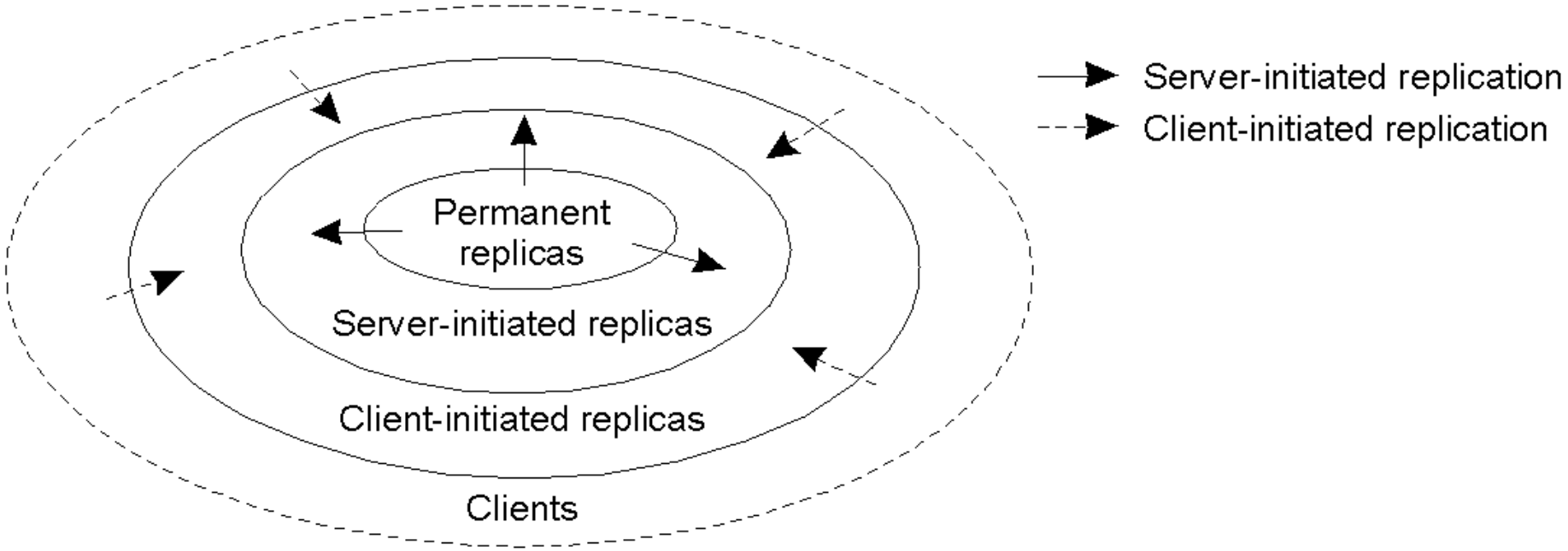
Summary, continued

- Various consistency models have been proposed
- "Data-Centric":
  - *Strict*, *Sequential*, *Causal*, *FIFO* concern themselves with individual reads/writes to data items.
  - Weaker models introduce the notion of synchronisation variables: *Release*, *Entry* concern themselves with a group of reads/writes.
- "Client Centric":
  - Concerned with maintaining consistency for a single clients' access to the distributed data-store.
    - The *Eventual Consistency* model is an example.

Consistency Protocols

Distribution Protocols

Regardless of which consistency model is chosen, we need to decide ***where***, ***when*** and ***by whom*** copies of the data-store are to be placed.



Replica Placement Types

There are three types of replica:

1. ***Permanent replicas***: tend to be small in number, organized as COWs (Clusters of Workstations) or mirrored systems. (Think of Web-servers or backup database servers)
2. ***Server-initiated replicas***: used to enhance performance at the initiation of the owner of the data-store. Typically used by web hosting companies to geographically locate replicas close to where they are needed most. (Often referred to as "push caches").
3. ***Client-initiated replicas***: created as a result of client requests – think of browser caches. Works well assuming, of course, that the cached data does not go *stale* too soon.

Update Propagation

1. Propagate *notification* of the update to other replicas
   - Often used for caches
   - Invalidation protocols
   - Works well when read-to-write ratio is small
2. Transfer the *data* from one replica to another
   - works well when there's many reads
   - Log the changes, aggregate updates
3. Propagate the *update operation* (AKA *active replicaton*)
   - Less bandwidth, more processing power

Push vs. Pull Protocols

Another design issue relates to whether or not the updates are *pushed* or *pulled*?

1. ***Push-based/Server-based Approach***: sent "automatically" by server, the client does *not* request the update. This approach is useful when a high degree of consistency is needed. Often used between permanent and server-initiated replicas.
2. ***Pull-based/Client-based Approach***: used by client caches (e.g., browsers), updates are requested by the client from the server. No request, no update!

Push vs. Pull Protocols: Trade Offs

Issue	Push-based	Pull-based
State on 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*.

Hybrid Approach: Leases

Lease is a *contract* in which the server promises to push updates to a client until the lease expires.

Make lease expiration time-depended on system behavior (adaptive leases)

- Age-based leases: an object that has not changed for a while probably will not change in the near future; provide longer lease
- Renewal-frequency leases: the more often a client requests a specific object, the longer the expiration time for that client (that object).
- State-object leases: the more loaded a server is, the shorter the expiration times become

Leases and Synchronization Problems

What if client's and server's clocks are not tightly synchronized?

- The client may take a "pessimistic" view concerning the level at which its clock is synchronized with the server's clock and attempt to obtain a new lease before the current one expires.

Epidemic Protocols

- This is an interesting class of protocol that can be used to implement *Eventual Consistency*
- The main concern is the propagation of updates to all the replicas in *as few a number of messages as possible*.
- Of course, here we are spreading updates, not diseases!
- With this "update propagation model", the idea is to "infect" as many replicas as quickly as possible.
  - Removing data can be problematic

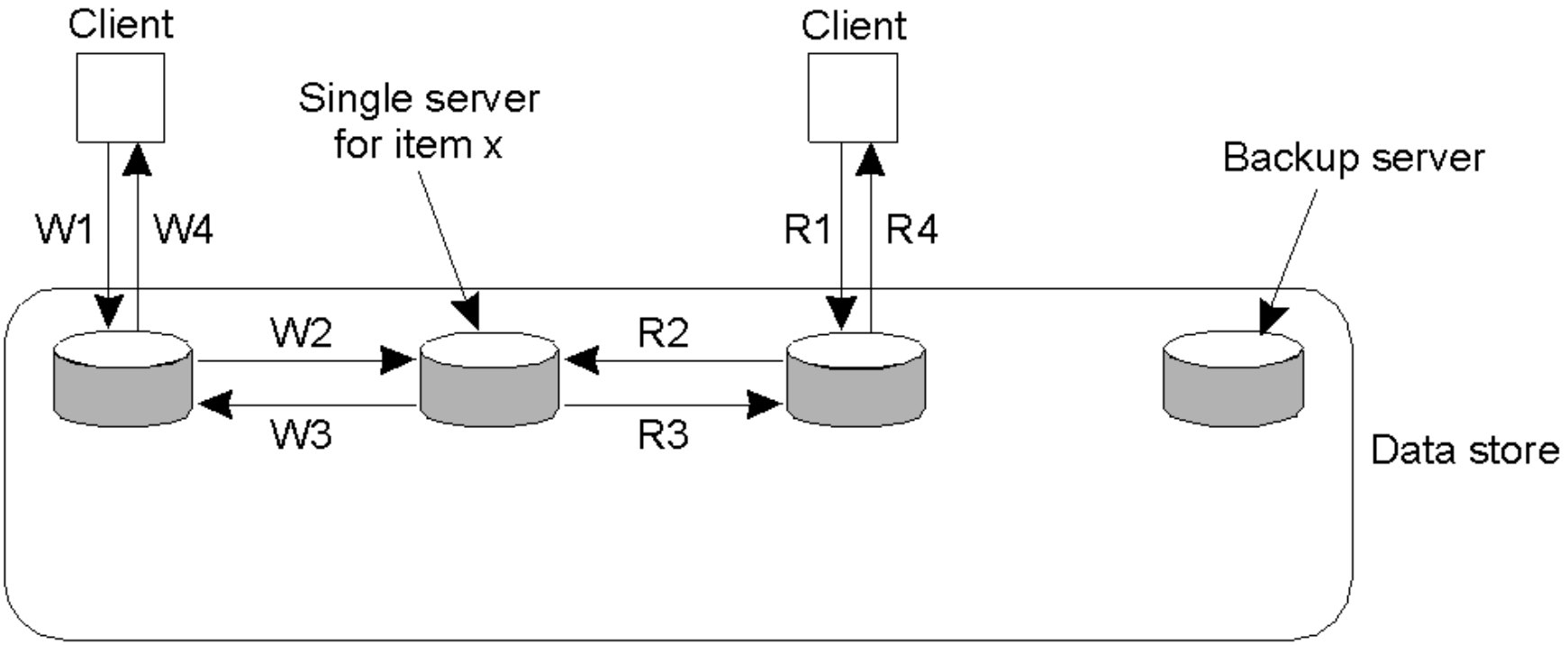
Consistency Protocols: Primary-Based Protocols

- Each data item is associated with a “primary” replica.
- The primary is responsible for coordinating writes to the data item.

- There are two types of Primary-Based Protocol:
  1. Remote-Write.
  2. Local-Write.

Remote-Write Protocols

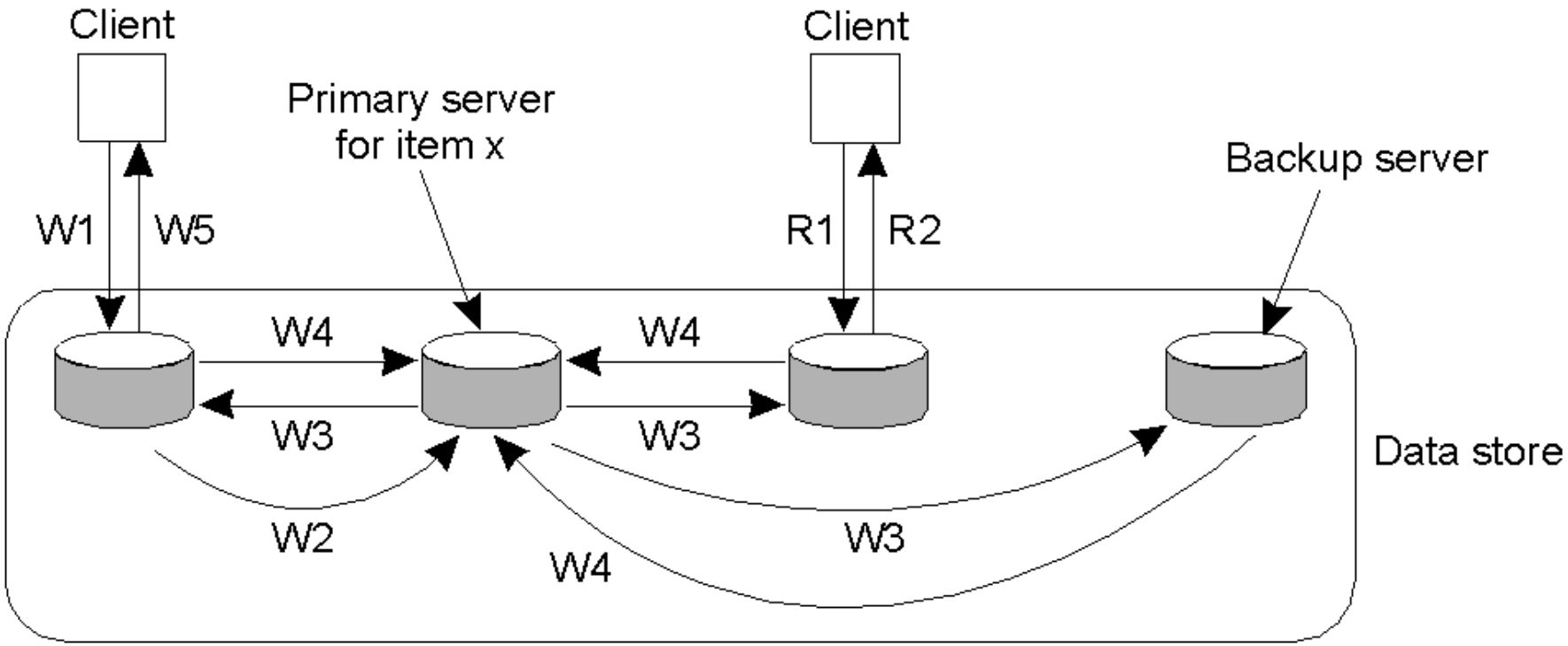
With this protocol, all writes are performed at a single (remote) server. This model is typically associated with traditional client/server systems.



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-Backup Protocol: A Variation



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

Writes are still centralised, but reads are now distributed. The *primary* coordinates writes to each of the backups.

The Bad and Good of Primary-Backup

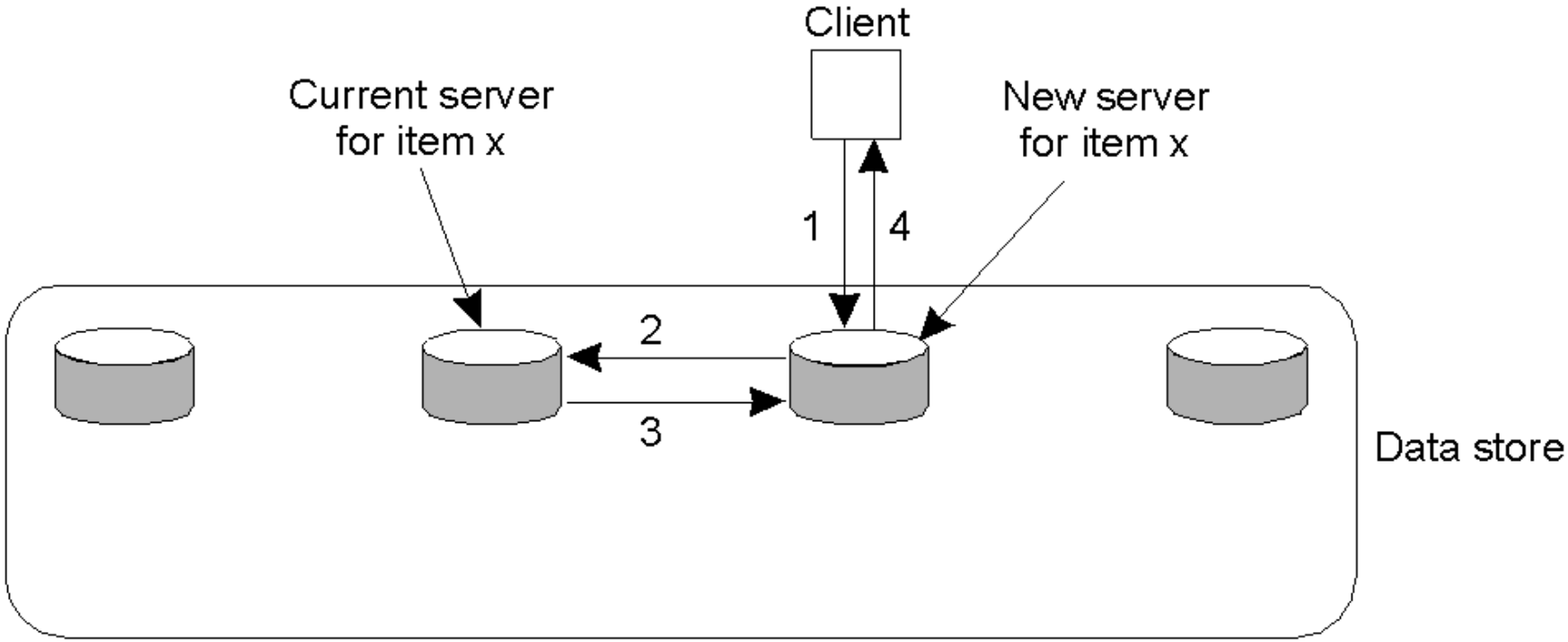
- Bad: Performance!
  - *All of those writes can take a long time (especially when a "blocking write protocol" is used).*

- Good: Using a non-blocking write protocol to handle the updates can lead to fault tolerant problems (which is our next topic).
- Good: The benefit of this scheme is, as the *primary* is in control, all writes can be sent to each backup replica *IN THE SAME ORDER*, making it easy to implement *sequential consistency*.

Local-Write Protocols

- In this protocol, a single copy of the data item is still maintained.
- Upon a write, the data item gets transferred to the replica that is writing.
- That is, the status of *primary* for a data item is *transferrable*.
- This is also called a “fully migrating approach”.

Local-Write Protocols Example



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 (prior to the read/write).

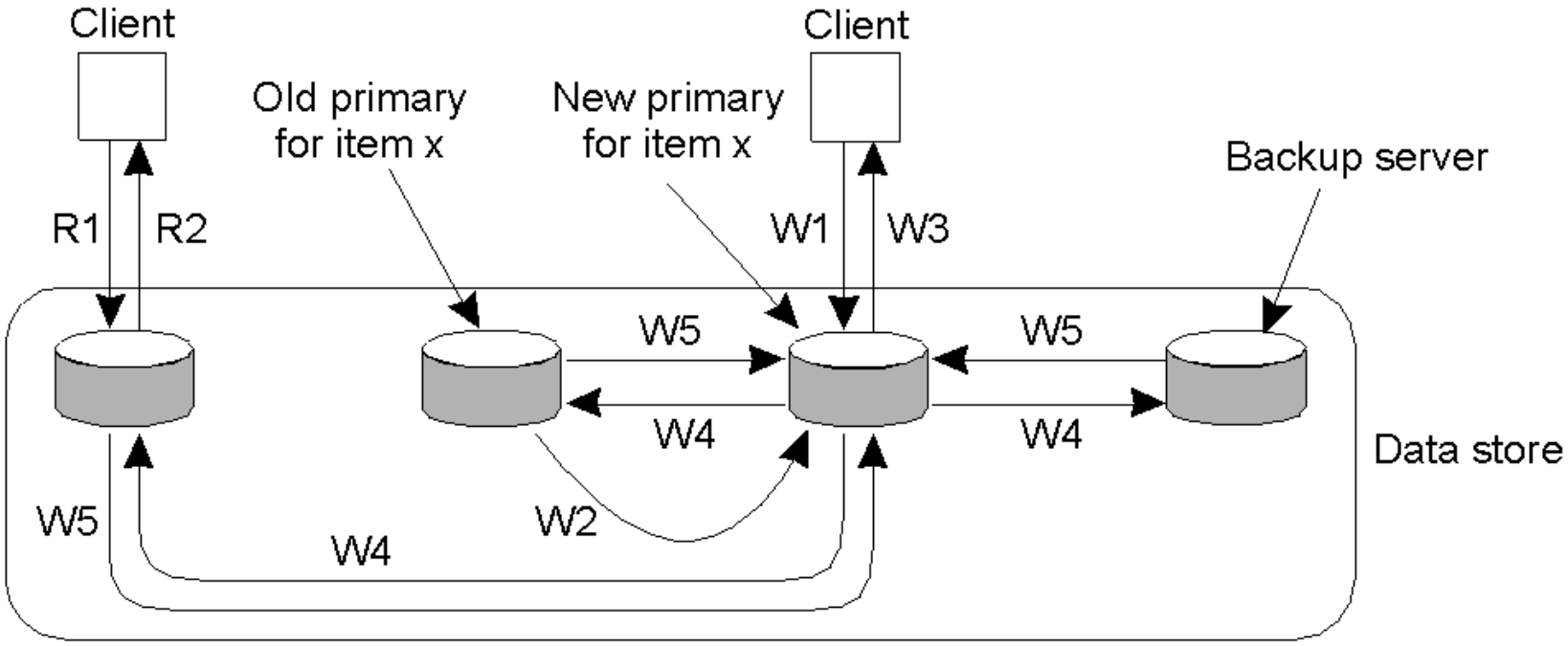
Local-Write Issues

- The big question to be answered by any process about to read from or write to the data item is:
  - *"Where is the data item right now?"*

- It is possible to use some of the *dymanic naming technologies,* but scaling quickly becomes an issue.

- Processes can spend more time actually locating a data item than using it!

Local-Write Protocols: A Variation



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, *then updates the backups*. Consequently, reads are much more efficient.

Consistency Protocols: Replicated-Write Protocols

- With these protocols, writes can be carried out at *any* replica.
- Another name might be: “Distributed-Write Protocols”

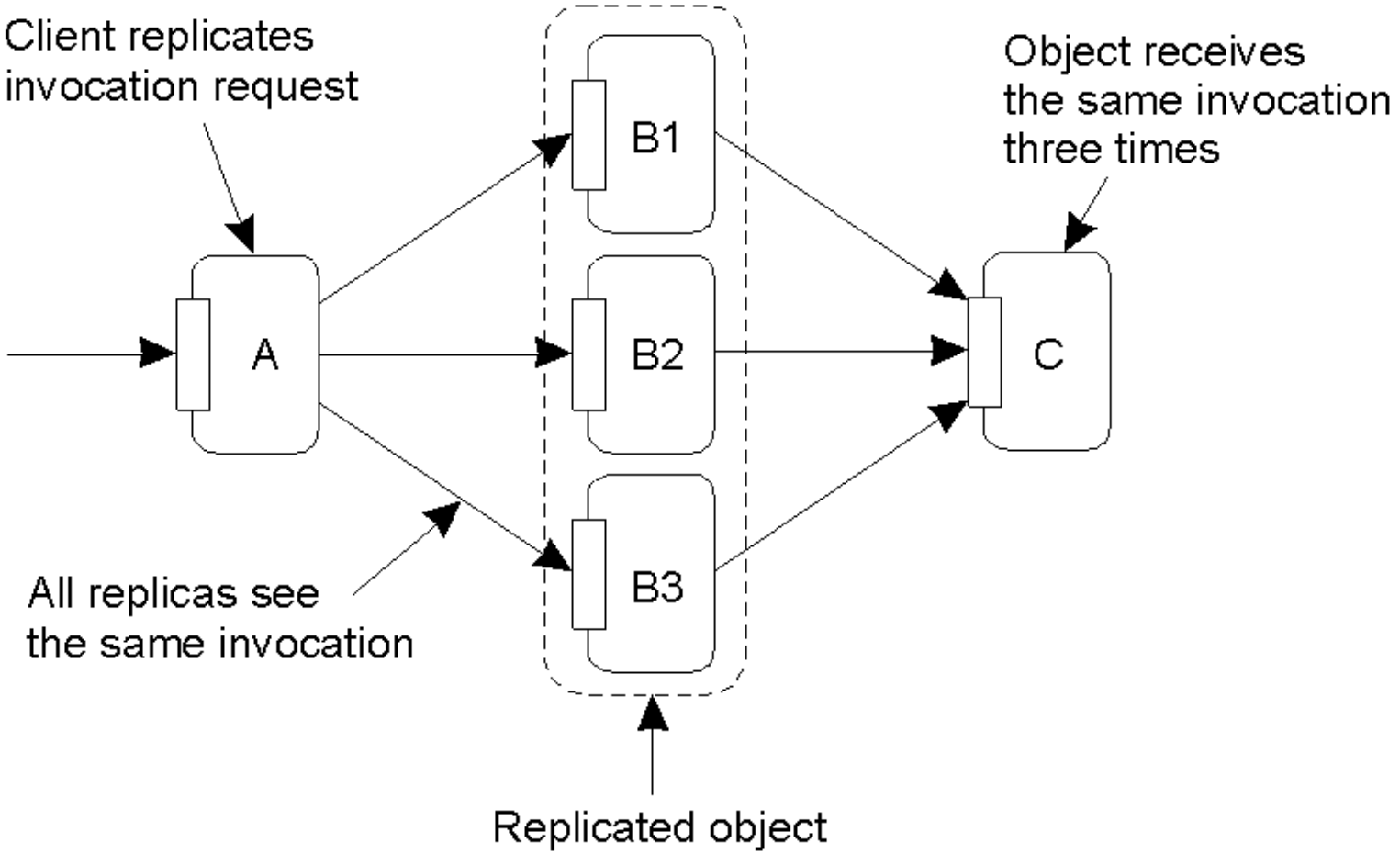
Example: Active Replication

- A special process carries out the update operations at each replica.
  - Lamport's timsestamps can be used to achieve total ordering, but this does not scale well within Distributed Systems.

- An alternative/variation is to use a *sequencer*, which is a process that assigns a unique ID# to each update, which is then propogated to all replicas.

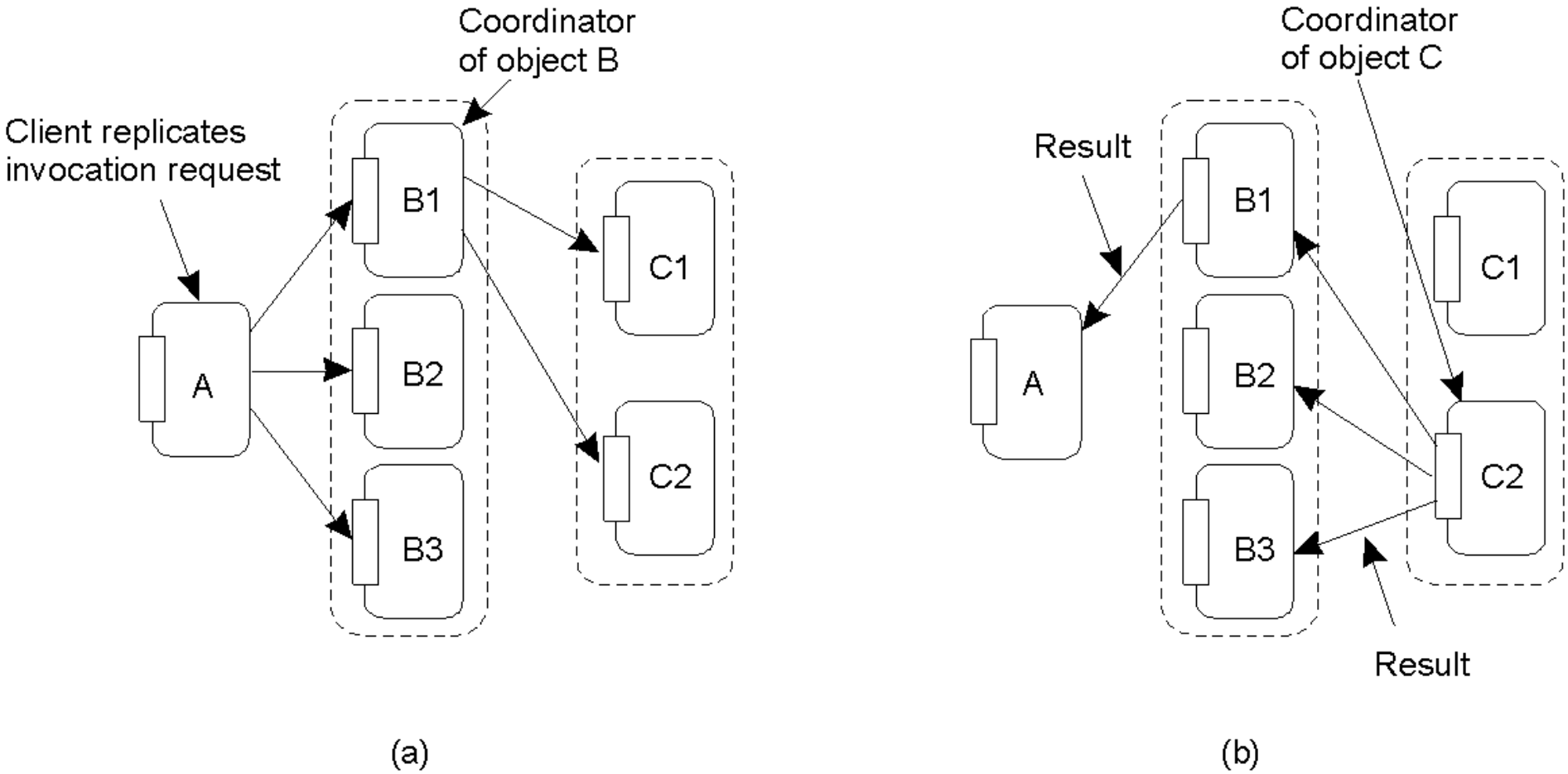
- This can lead to another problem: *replicated invocations*.

Active Replication: The Problem



The problem of replicated invocations – 'B' is a replicated object (which itself calls 'C'). When 'A' calls 'B', how do we ensure 'C' isn't invoked three times?

Active Replication: Solutions



a) Using a coordinator for ‘B’, which is responsible for forwarding an invocation request from the replicated object to ‘C’.

b) Returning results from ‘C’ using the same idea: a coordinator is responsible for returning the result to all ‘B’s. Note the single result returned to ‘A’.

Summary

- To distribute (or “propagate”) updates, we draw a distinction between *WHAT* is propagated, *WHERE* it is propagated and by *WHOM*.
  - Epidemic protocols
  - Primary-based protocols
  - Active replication