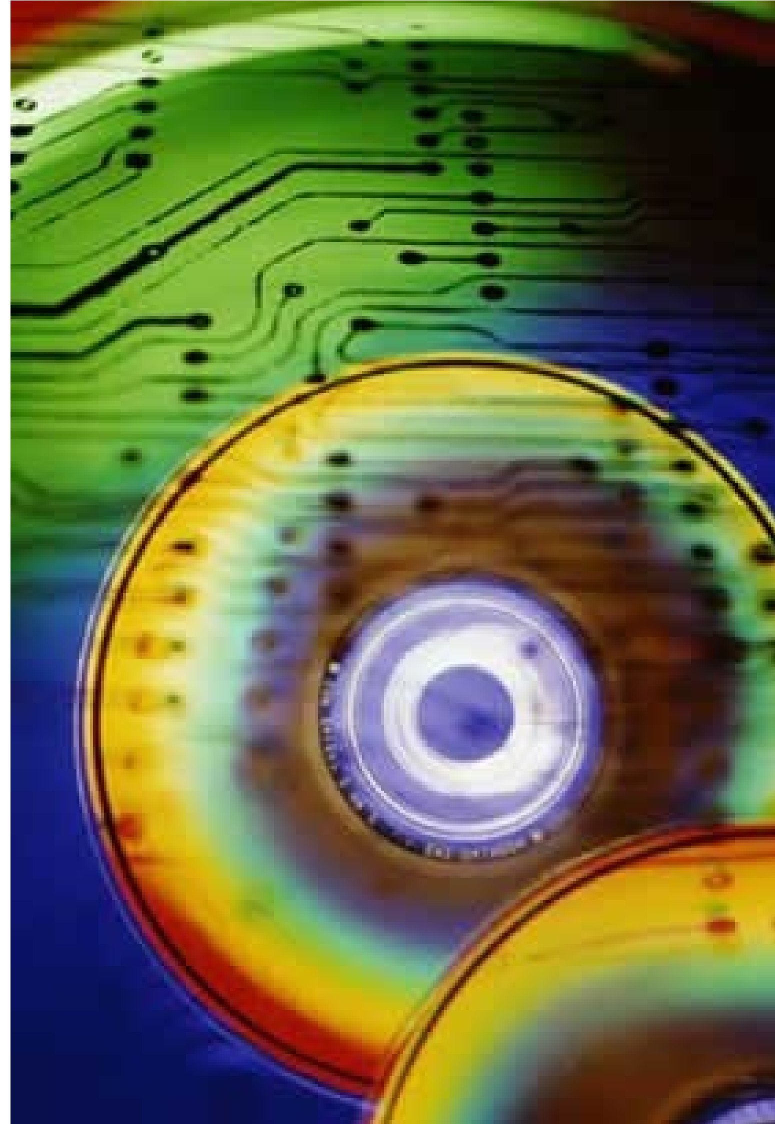
ICS 362 Distributed Systems

Distributed Systems: Part 15

Lecturer: Toby Daniel

Consistency and Replication

- Client centric
- Protocols



Client Centric Consistency

- So far we have considered how to make data consistent for a variety of processes, but maintaining data consistency might not be the important factor.
- Can you think of a situation where data consistency is not the most important factor.

Client Centric Consistency

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

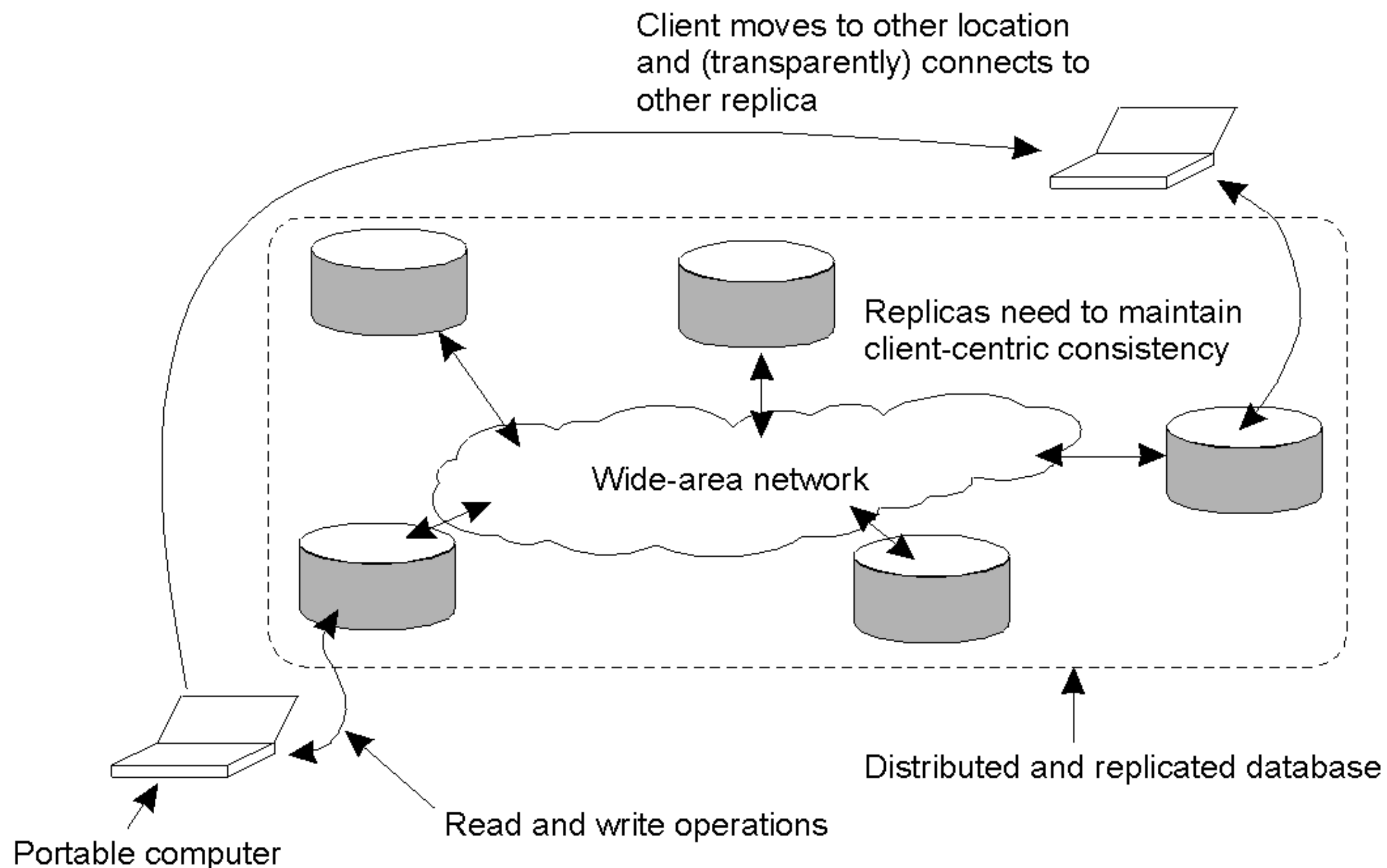
- Think of most database systems: *mainly read*.
- Think of the DNS: *write-write conflicts* do not occur.
- Think of WWW: as with DNS, except that heavy use of client-side caching is present: *even the return of stale pages is acceptable to most users*.

- These systems all exhibit a high degree of acceptable inconsistency, with the *replicas* gradually becoming consistent over time.

Eventual Consistency

- In ***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*.

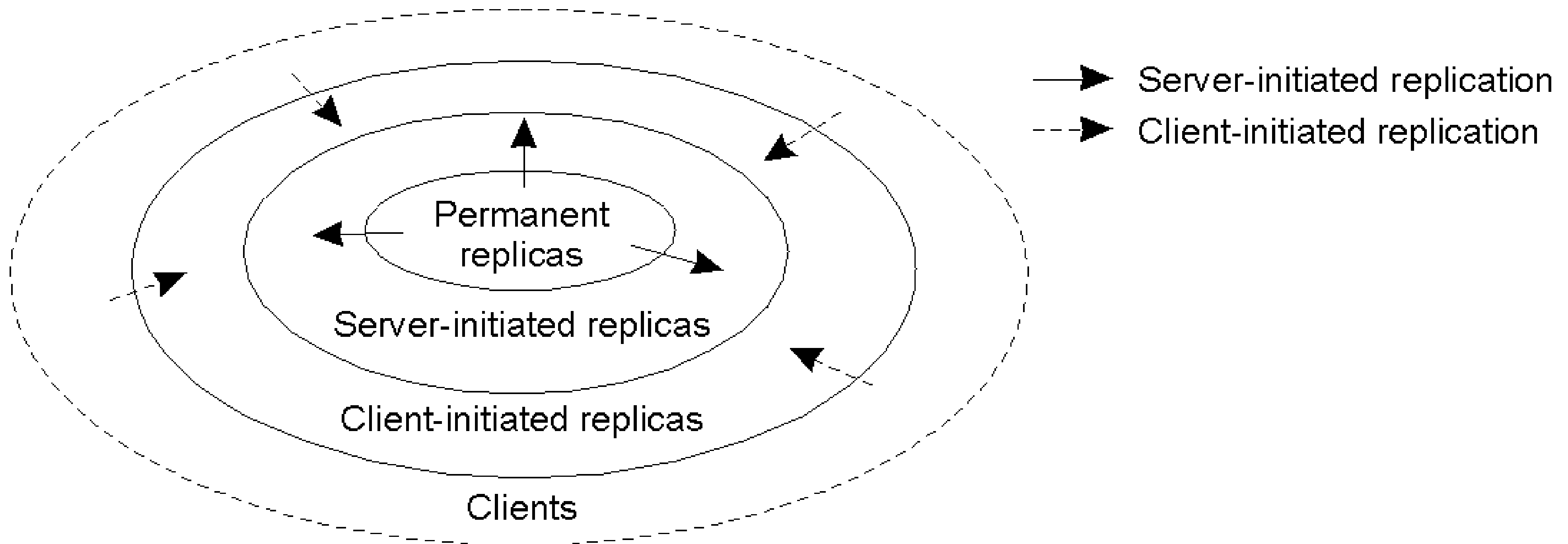
Mobile Consistency



- When the portable computer connects to a different replica, client consistency becomes a harder issue.

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

- There are three types of replica:
  1. ***Permanent replicas***: tend to be small in number, organized as COWs (Clusters of Workstations) or mirrored systems.
  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

- When a client initiates an update to a distributed data-store, what gets propagated?
- There are three possibilities:
  1. Propagate *notification* of the update to the other replicas – this is an "invalidation protocol" which indicates that the replica's data is no longer up-to-date. Can work well when there's many writes.
  2. Transfer the *data* from one replica to another – works well when there's many reads.
  3. Propagate the *update* to the other replicas – this is "active replication", and shifts the workload to each of the replicas upon an "initial write".

Push Vs Pull

Push Vs Pull

- 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!

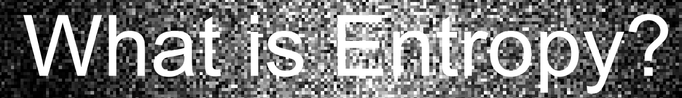
Epidemic Protocols

- 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.

Epidemic Terms

- ***Infective replica***: a server that holds an update that can be spread to other replicas.
- ***Susceptible replica***: a yet to be updated server.
- ***Removed replica***: an updated server that will not (or cannot) spread the update to any other replicas.
- The trick is to get all susceptible servers to either infective or removed states as quickly as possible without leaving any replicas out.

Entropy



Anti Entropy

- Entropy: "a measure of the degradation or disorganization of the universe".
- Server P picks Q at random and exchanges updates, using one of three approaches:
  1. P only pushes to Q.
  2. P only pulls from Q.
  3. P and Q push and pull from each other.
  4. Sooner or later, all the servers in the system will be infected (updated). Works well.

Gossiping

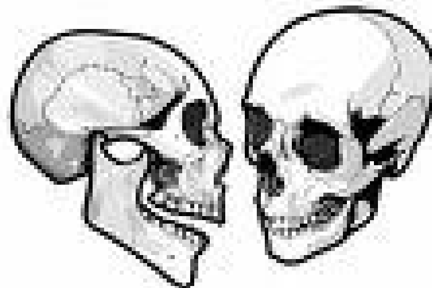
- This variant is referred to as “gossiping” or “rumour spreading”, as works as follows:
  1. P has just been updated for item ‘x’.
  2. It immediately pushes the update of ‘x’ to Q.
  3. If Q already knows about ‘x’, P becomes disinterested in spreading any more updates (rumours) and is removed.
  4. Otherwise P gossips to another server, as does Q.
- This approach is good, but can be shown not to guarantee the propagation of all updates to all servers.

A Hybrid

- A mix of anti-entropy and gossiping is regarded as the best approach to rapidly infecting systems with updates.

Removing Data...

- However, what about ***removing*** data?
  - Updates are easy, deletion is much, much harder!
  - Under certain circumstances, after a deletion, an “old” reference to the deleted item may appear at some replica and cause the deleted item to be *reactivated*!
  - One solution is to issue “Death Certificates” for data items – these are a special type of update.

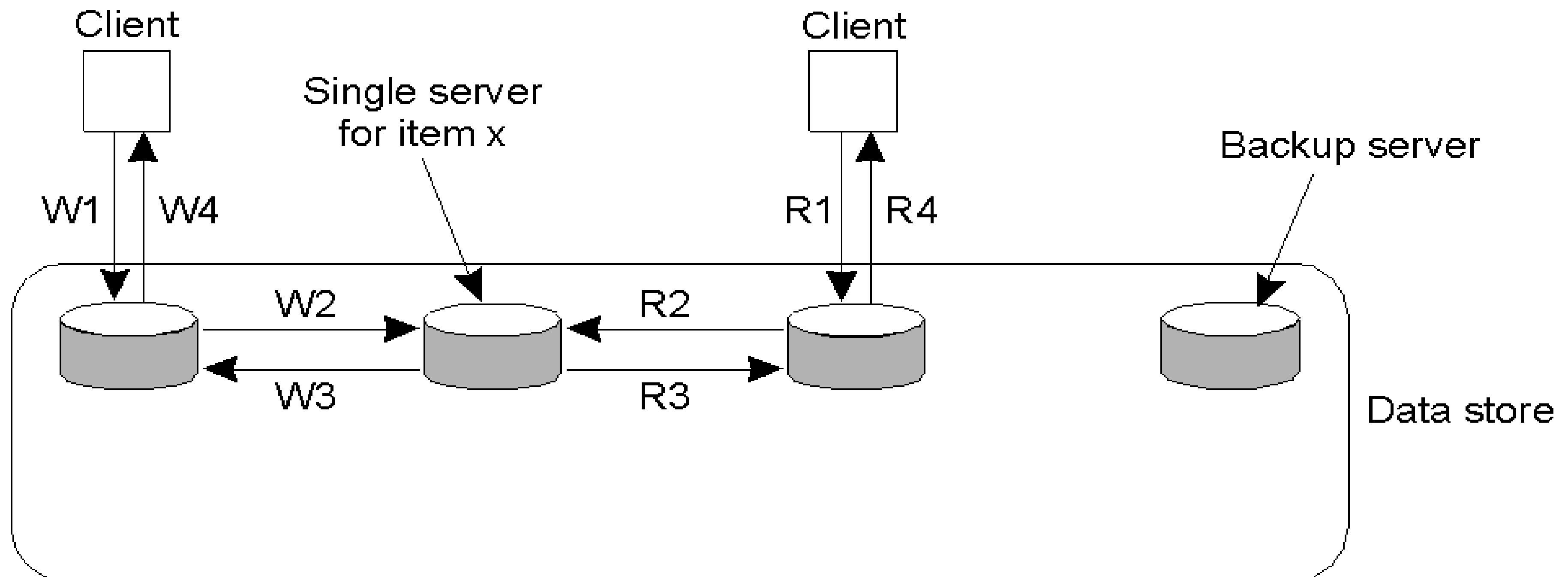


Primary Based Protocols

- 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

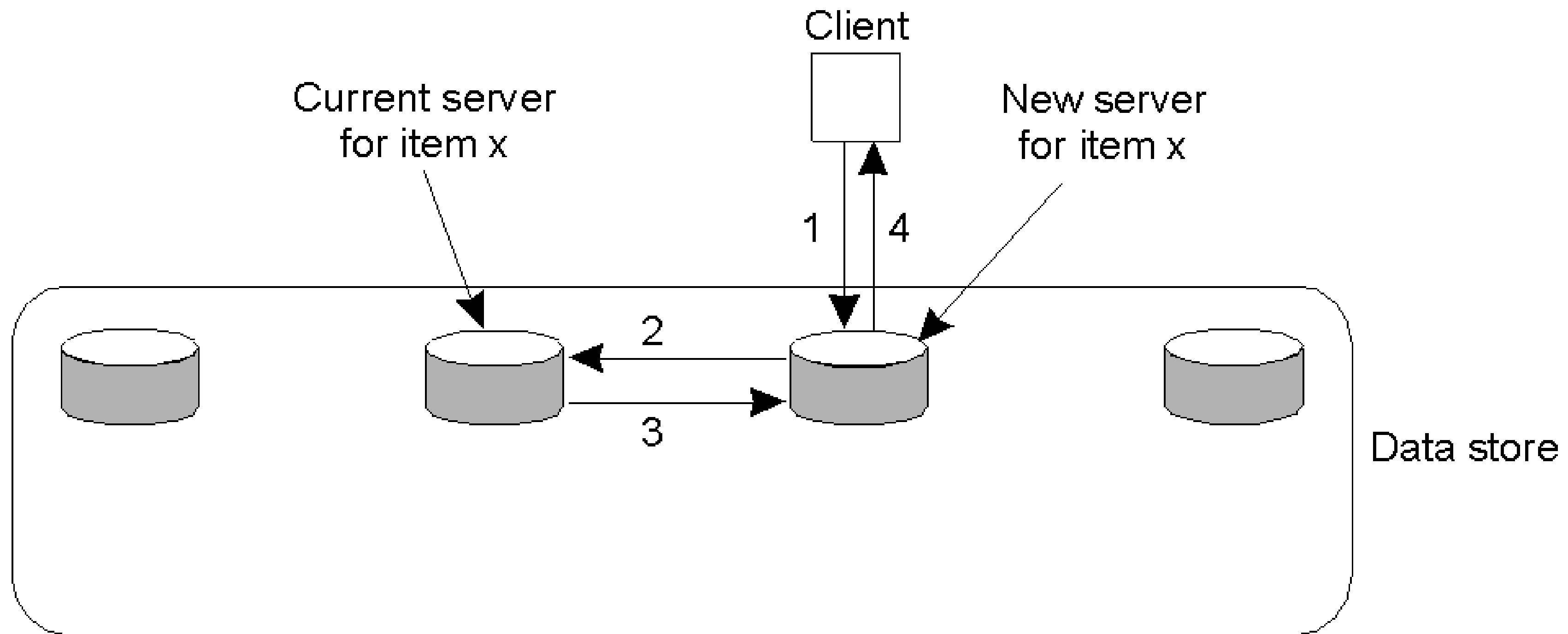
R1. Read request
R2. Forward request to server for x
R3. Return response
R4. Return response

Primary Backups

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

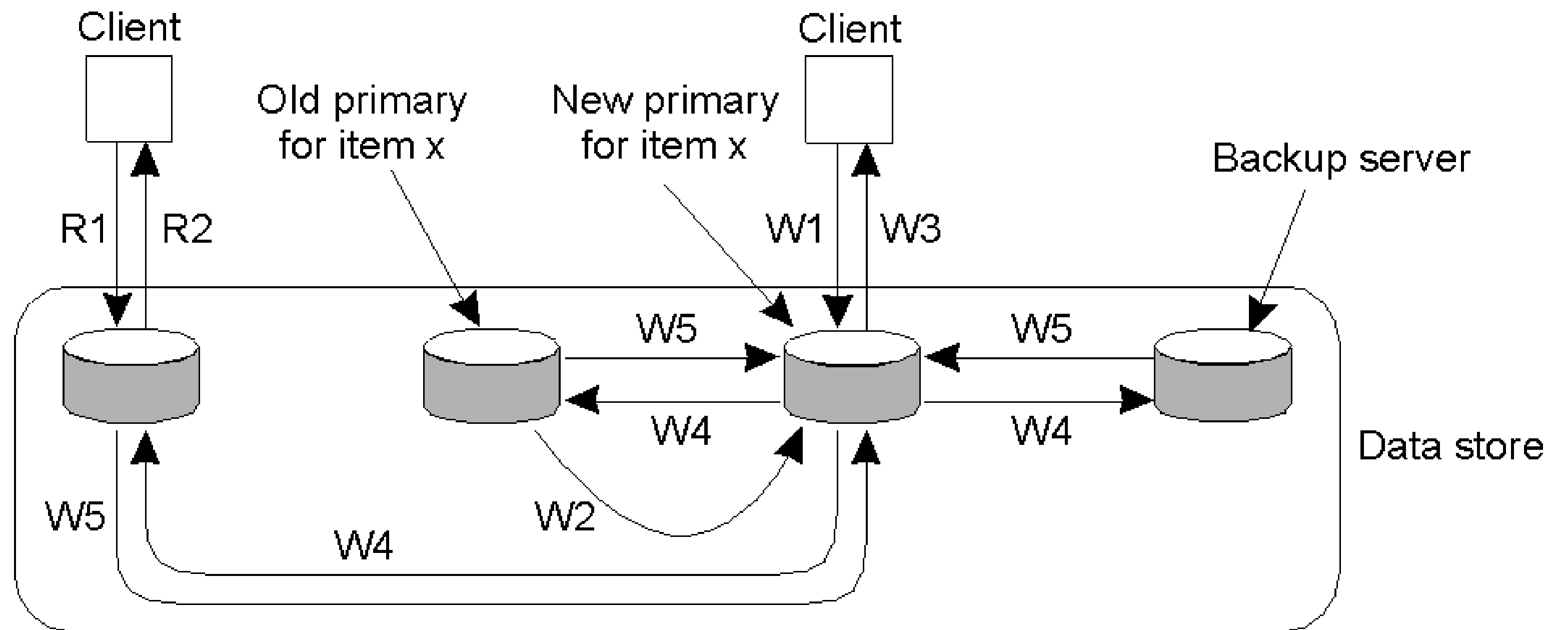


- Primary-based local-write protocol in which a single copy is *migrated* between processes (prior to the read/write).

Local Write Problems

- 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* studied earlier in this course, but scaling quickly becomes an issue.
- Processes can spend more time actually locating a data item than using it!

Local Write 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.

Replicated Write Protocols

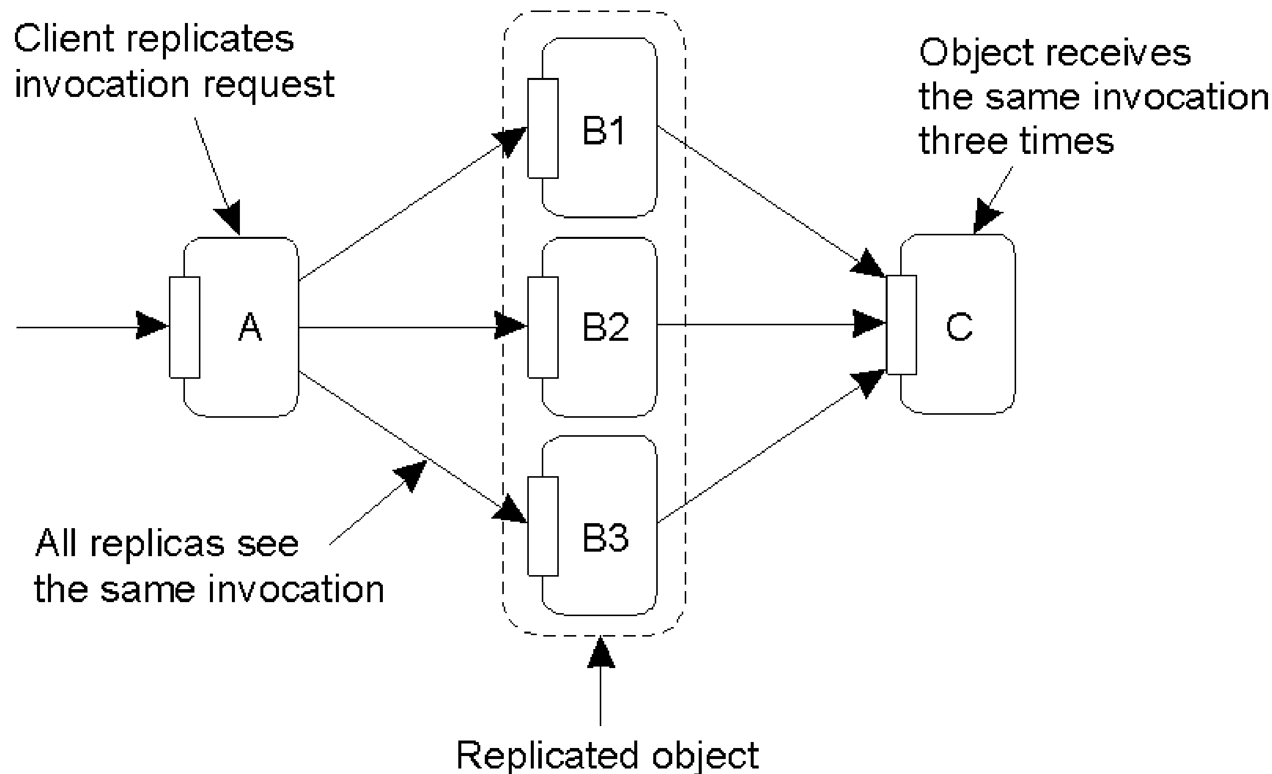
- Another name might be: “Distributed-Write Protocols”
- There are two types:
  1. Active Replication.
  2. Majority Voting (Quorums).

Active Replications

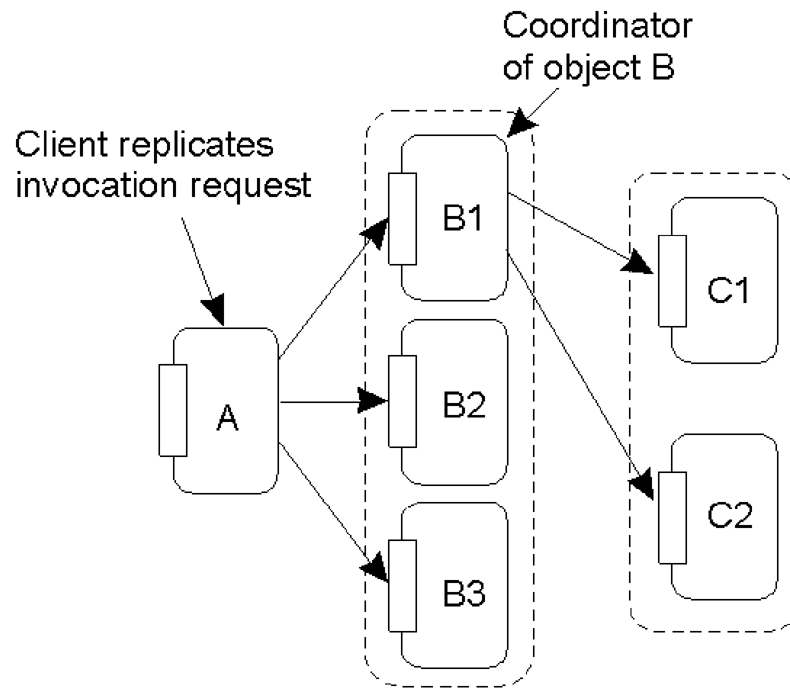
- A special process carries out the update operations at each replica.
- Lamport's timsestamps can be used to achieve total ordering, but this does not scale well within Distributed Systems.
- An alternative/variation is to use a *sequencer*, which is a process that assigns a unique ID# to each update, which is then propogated to all replicas.
- This can lead to another problem: *replicated invocations*.

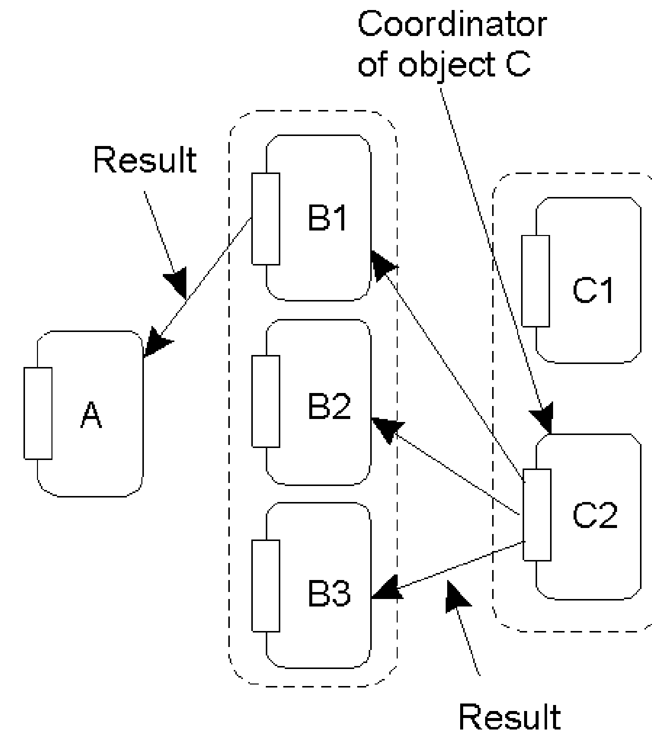
Active Replication Problem

- 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 - Solution





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’.

Quorum Based Protocols

- Clients must request and acquire permissions from multiple replicas before either reading/writing a replicated data item.
- Consider this example:
  - A file is replicated within a distributed file system.
  - To update a file, a process must get approval from a majority of the replicas to perform a write. The replicas need to agree *to also perform the write*.
  - After the update, the file has a new version # associated with it (and it is set at all the updated replicas).
  - To read, a process contacts a majority of the replicas and asks for the version # of the files. If the version # is the same, then the file must be the most recent version, and the read can proceed.