Consistency and Replication

Why Replicate Data?

- Enhance reliability.
- Improve performance.

But: if there are many replicas of the same thing…

- How do we keep all of them up-to-date?
- How do we keep the replicas *consistent*?

- **Consistency** can be achieved in a number of ways.
  - We will study a number of ***consistency models*** as well as ***protocols*** for implementing the models.

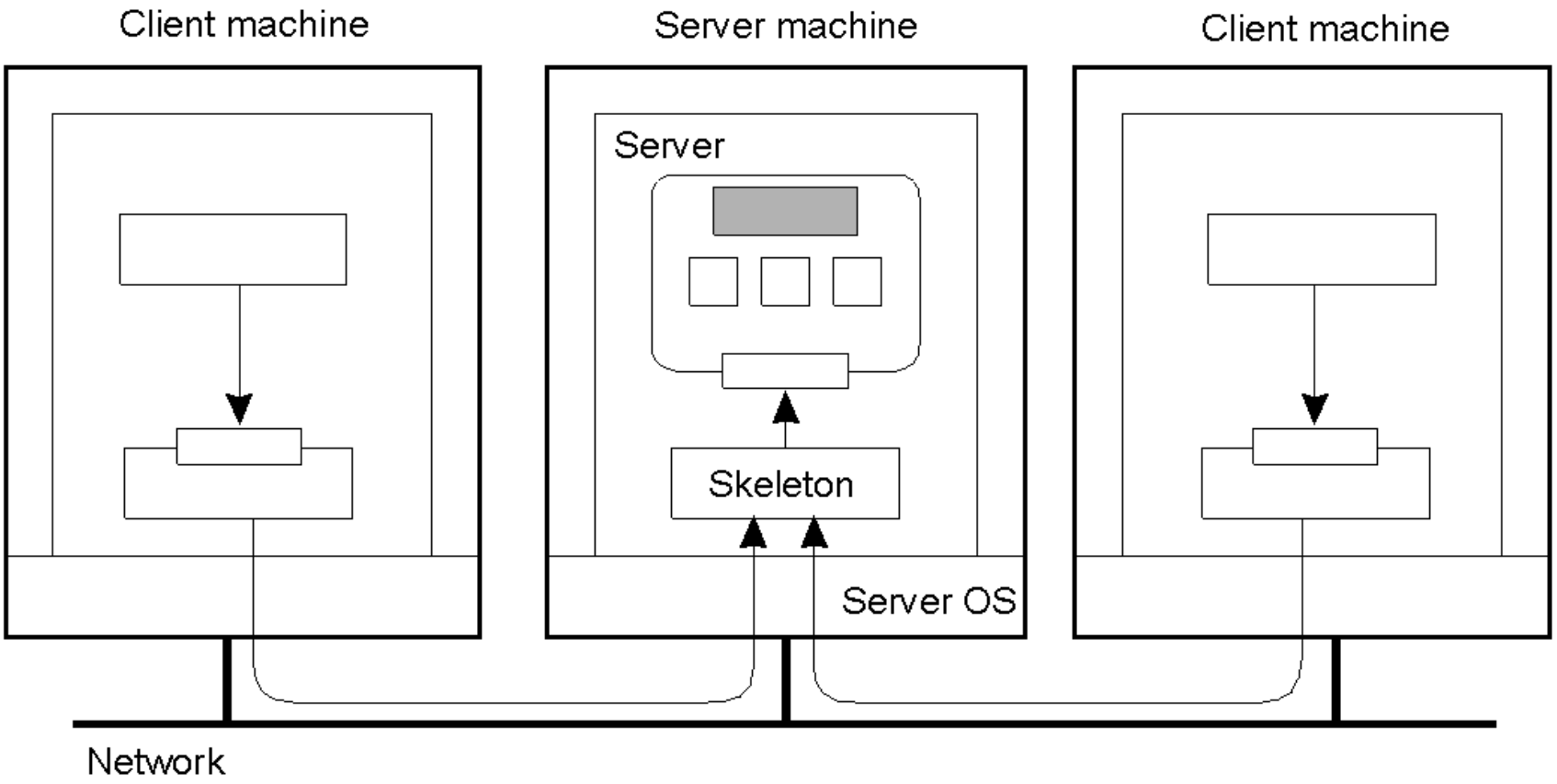
More on Replication

- Support DS goal of enhanced ***scalability***:
  - Replicas allow remote sites to continue working in the event of failures.
  - Replicas Protect against data corruption.
  - Replicas allow data to reside close to where it is used.
  - Even a large number of replicated “local” systems can improve performance: think of clusters.

So, what’s the catch?

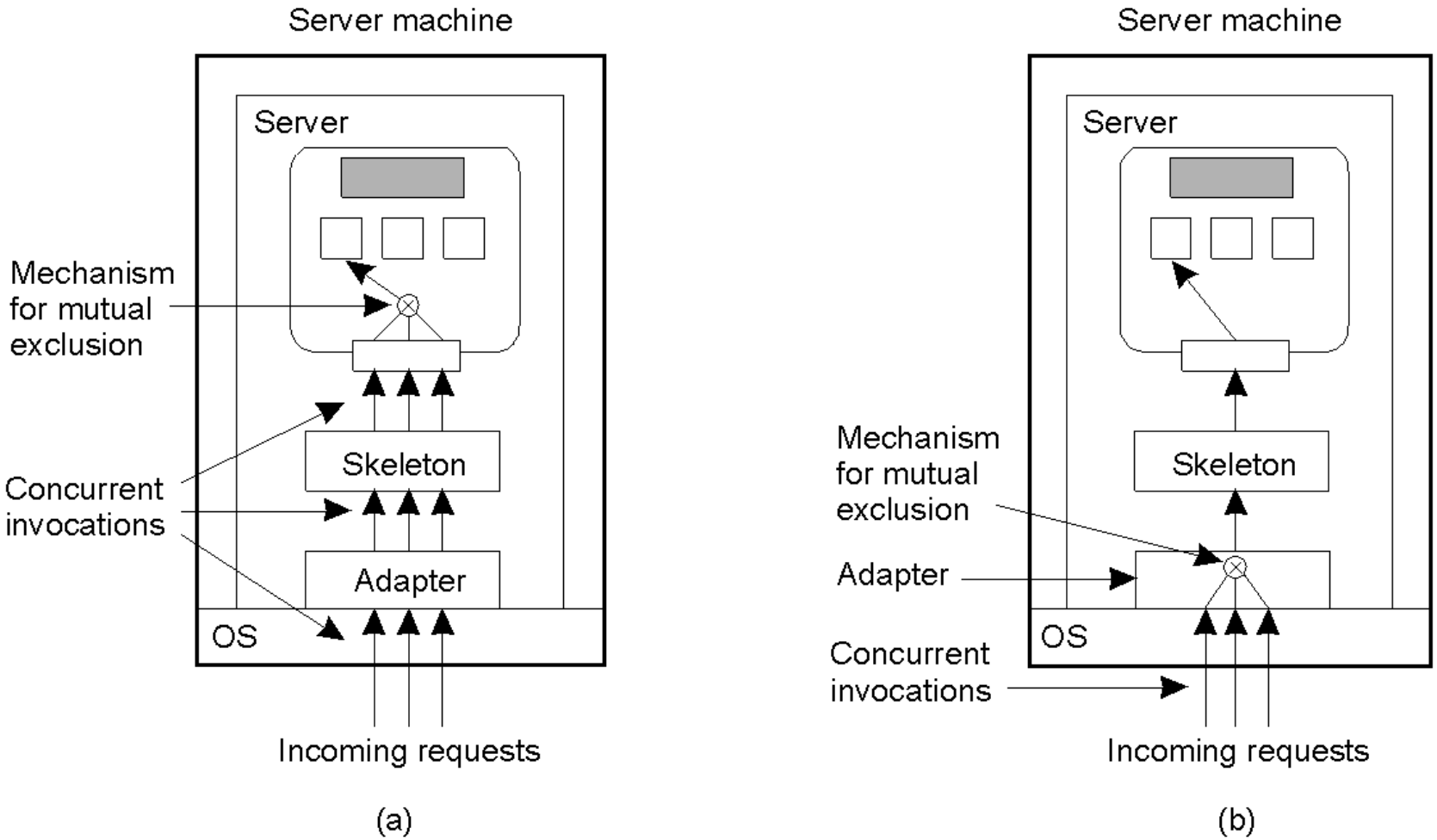
- It is **not easy** to keep all those replicas ***consistent***.

Concurrent Object Access: Problem



- Organization of a distributed remote object shared by two different clients… but, how do we protect the object in the presence of multiple simultaneous access?

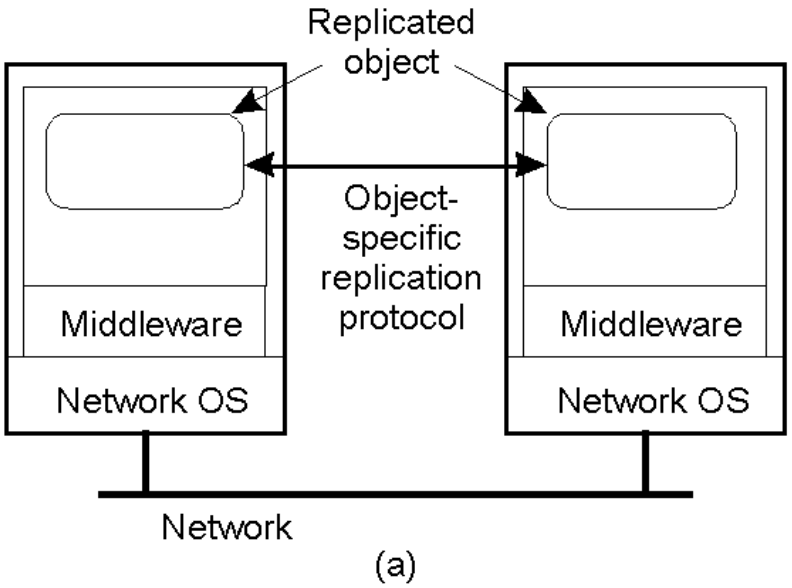
Concurrent Object Access: Solutions

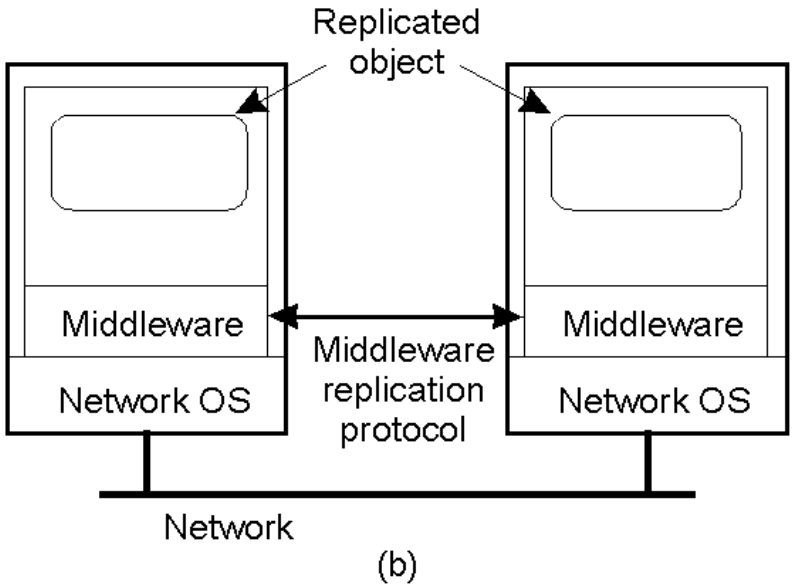


(a) Remote object capable of handling concurrent invocations on its own (e.g. Java synchronization mechanisms).

(b) Remote object for which an adapter handles concurrent invocations (e.g. CORBA middleware adapters).

Object Replication: Solutions





(a) DS for replication-aware distributed objects – the object itself is "aware" that it is replicated (flexible set-up, but can be costly since DS developer has to deal with replication/consistency).

(b) DS responsible for replica management – less flexible, but removes the burden from the DS developer (most common approach).

Replication and Scalability

- **Replication** is a widely-used scalability technique (e.g. web clients and web proxies).
  - When systems scale, the first problems are 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.

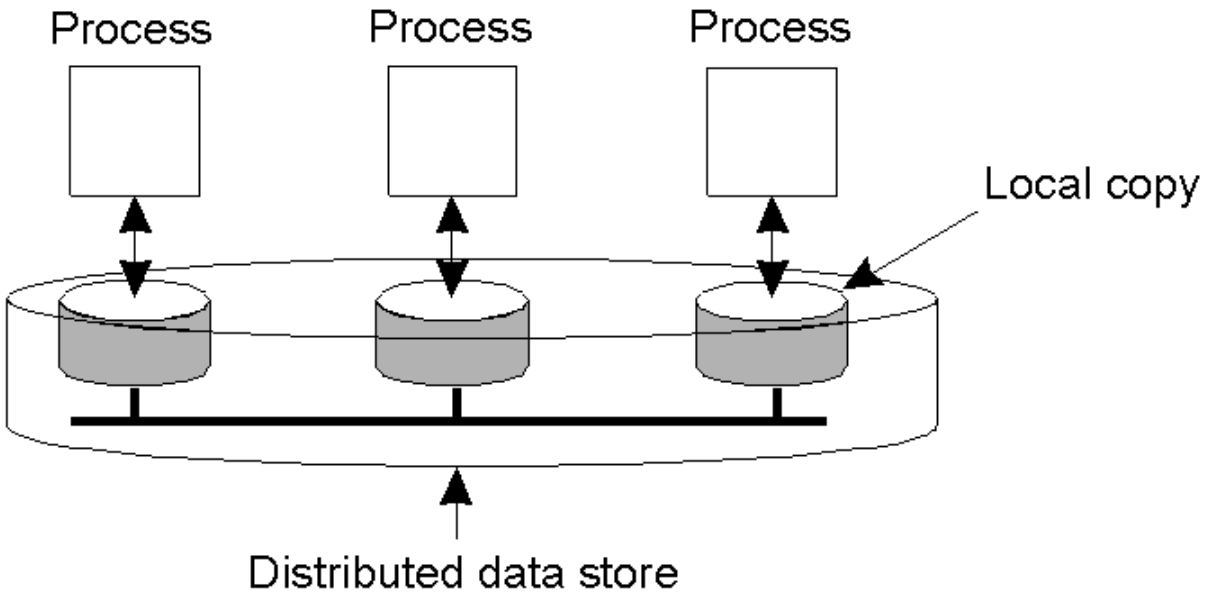
- P**roblem** remains: how to efficiently **synchronize** all of the replicas created to solve the scalability issue?

- **Dilemma**: adding replicas improves scalability, but incurs (oftentimes) on **overhead** of keeping replicas **up-to-date**!!!

- Solution often results in a **relaxation** of any ***consistency constraints***.

Data-Centric Consistency Models

- **Data-store** can be read from or written to by any process in a DS.
- L**ocal copy** of the data-store (replica) can support "**faster 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 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 as advertised.

- We start with ***Strict Consistency***, which is:
  - Any read on a data item 'x' returns a value corresponding to the result of the most recent write on 'x' (regardless of where the write occurred).

Consistency Model Diagram Notation

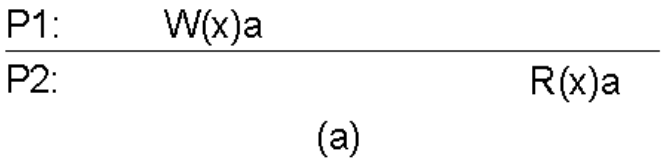
$W_i(x)a$ – a write by process ‘i’ to item ‘x’ with a value of ‘a’ (‘x’ is set to ‘a’).

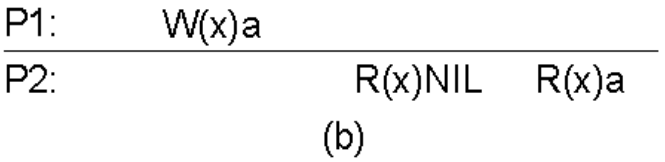
- NB: The process is often shown as ‘P_i’.

$R_i(x)b$ – a read by process ‘i’ from item ‘x’ producing the value ‘b’ (reading ‘x’ returns ‘b’).

- NB: Time moves from left to right in all diagrams.

Strict Consistency Diagrams





- Behavior of two processes, operating on the same data item:
  a) A strictly consistent data-store.
  b) A data-store that is not strictly consistent.

- With ***Strict Consistency***, all writes are *instantaneously visible* to all processes and *absolute global time order* is maintained throughout DS.
  - This is the "Holy Grail" consistency model – **not** at all **easy** in the real world, and all but ***impossible*** within a **DS**.

- So, other less strict ("**weaker**") models have been developed…

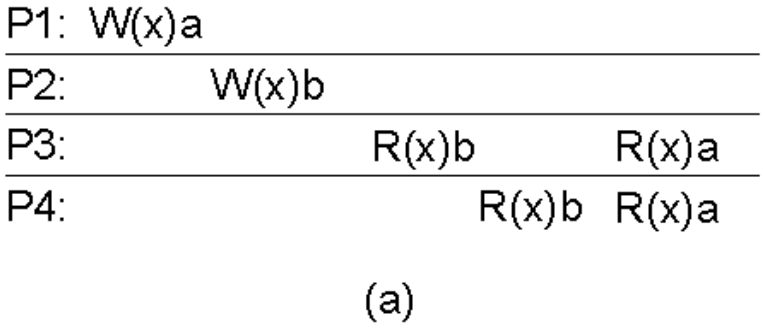
Sequential Consistency

- Weaker consistency model with relaxation of rules
  - It is also mutch easier (possible) to implement.

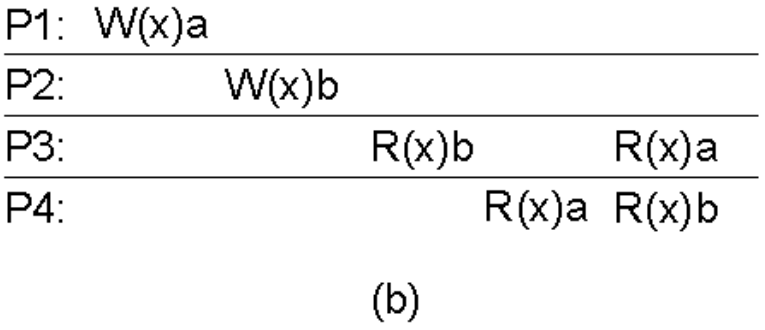
- Definition of "**Sequential Consistency**":
  - The result of any execution is the same as if the (read and write) operations by all proceses on the data-store were executed in the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

Sequential Consistency Diagrams

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



(a)



(b)

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

b) Data-store **not sequentially consistent** – it appears the writes have occurred in a non-sequential order (NOT allowed).

Problem with Sequential Consistency

- With this consistency model, adjusting the protocol to **favour reads** over writes (or vice-versa) can have a devastating **impact** on **performance** (refer to the textbook for the gory details).

- For this reason, other **weaker consistency models** have been proposed and developed.

- Again, a **relaxation** of rules allows for weaker models to make sense.

Causal Consistency

- This model distinguishes between events that are "**causally related**" and those that are not.
  - If event B is caused or influenced by an earlier event A, then causal consistency requires that every other process see event A and afterwards event B.
- Operations that are **not causally related** are said to be ***concurrent***.

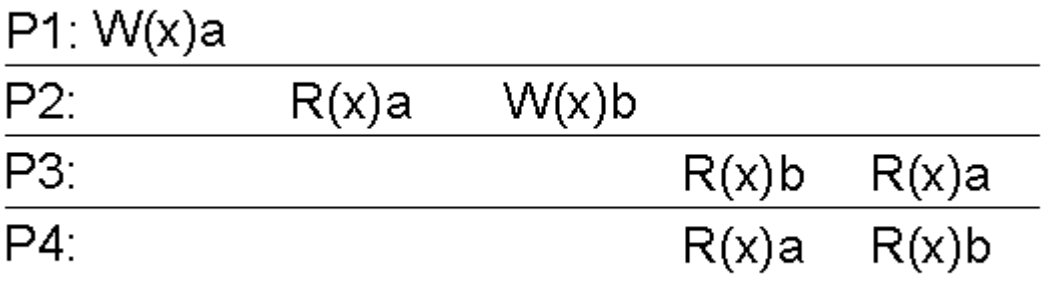
More on Causal Consistency

- A causally consistent data-store obeys this condition:
  - Writes that are potentially causally related must be seen by all processes in the same order.
  - Concurrent writes may be seen in a different order on different machines (i.e., by different processes).

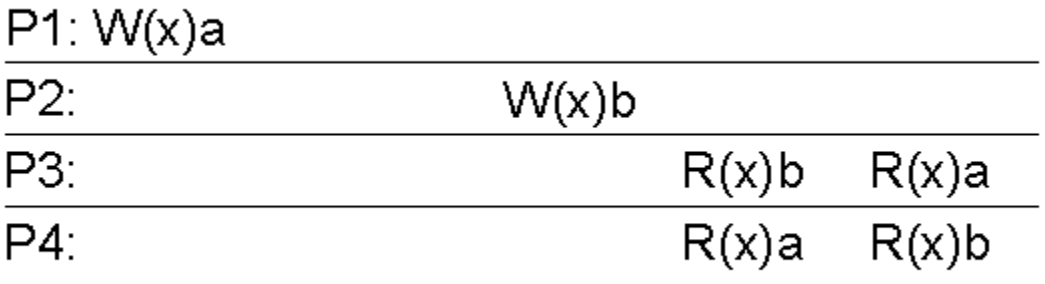
P1:	W(x)a			W(x)c	
P2:		R(x)a	W(x)b		
P3:		R(x)a		R(x)c	R(x)b
P4:		R(x)a		R(x)b	R(x)c

- This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.
  - NB: it is assumed that $W_2(x)b$ and $W_1(x)c$ are concurrent.

Another Causal Consistency Example



(a)



(b)

(a) Violation of causal-consistency: P2 write is related to P1 write due to read on ‘x’ giving ‘a’ (all processes must see them in same order).

(b) Causally-consistent data-store: the read has been removed, so the two writes are now *concurrent* (the reads by P3 and P4 are now OK).

FIFO Consistency

- Defined as follows:
  - Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in different order by processes.
  - This is also called "PRAM Consistency" (Pipelined RAM).
  - Attractive characteristic of FIFO - **easy to implement**.
  - There are no guarantees about the order in which different processes see writes – except that two or more writes from a single process must be seen in order.

FIFO Consistency Example

P1: W(x)a						
P2:	R(x)a	W(x)b	W(x)c			
P3:				R(x)b	R(x)a	R(x)c
P4:				R(x)a	R(x)b	R(x)c

- Valid sequence of FIFO consistency events.
  - NB: none of the consistency models studied so far would allow this sequence of events.

Introducing Weak Consistency

- Not all applications need to see all writes, let alone seeing them in the same order.

- This leads to "**Weak Consistency**" (which is primarily designed to work with *distributed critical regions*).

- This model introduces the notion of a "**synchronization variable**", which is used to update **all copies** of the **data-store**.

Weak Consistency Properties

- The 3 properties of **Weak Consistency**:

1. A**ccesses** to synchronization variables associated with a data-store are ***sequentially*** *consistent*.
2. No operation on a synchronization variable is allowed to be performed until all **previous writes** have been **completed** everywhere.
3. No read or write operation on data items are allowed to be performed until all previous operations to **synchronization** variables have been **performed**.

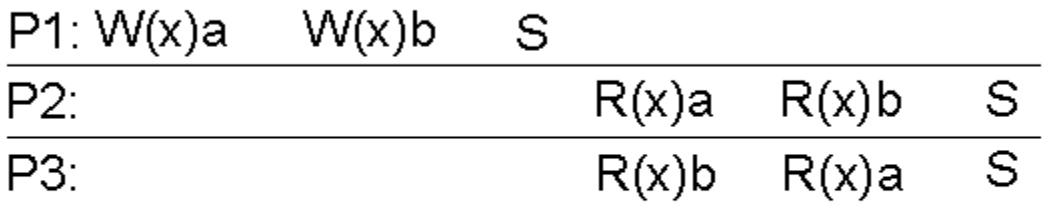
Weak Consistency: What It Means

So…

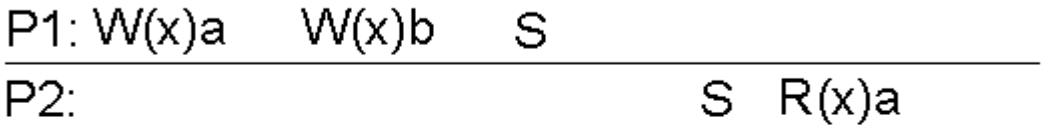
- By doing a **sync**, a process can ***force*** the just written value out to all the other replicas.
- Also, by doing a **sync**, a process can be *sure* it is getting the most recently written value before it reads.

- In essence, the weak consistency models **enforce consistency** on a ***group of operations***, as opposed to individual reads and writes (as with strict, sequential, causal and FIFO consistency).

Weak Consistency Examples



(a)



(b)

(a) Valid sequence of events for weak consistency (P2 and P3 have yet to synchronize, so there is no guarantees about value in ‘x’)

(b) Invalid sequence for weak consistency (P2 has synchronized, so it cannot read ‘a’ from ‘x’ – it should get ‘b’)

Introducing Release Consistency

- Question:
  - how does a weak consistent data-store knows that **sync** is the result of a read or a write?
  - Answer: It doesn't!

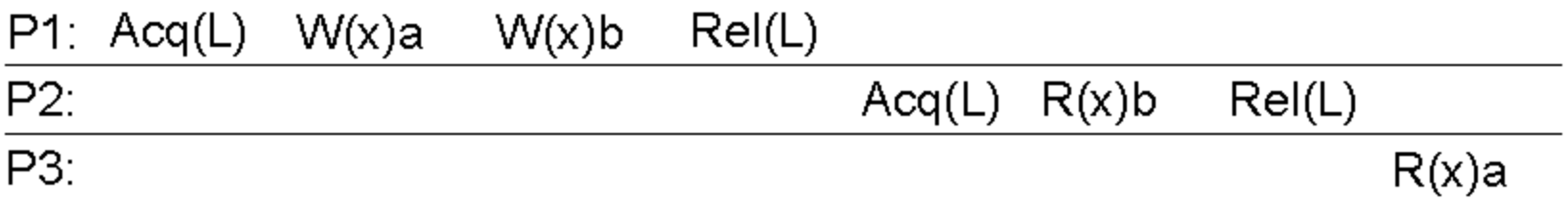
- Possible to implement efficiencies if data-store is able to determine whether the sync is a read or write.

- **Two sync variables** can be used:
  - "**acquire**" and "**release**" **-** leads to "**Release Consistency**" model.

Release Consistency

- Defined as follows:
  - When a process does an "**acquire**", the data-store will ensure that all the **local copies** of the protected data are brought **up to date** to be **consistent** with the remote ones if needs be.
  - When a "**release**" is done, protected data that have been changed are **propogated** out to the local copies of the data-store.

Release Consistency Example



- A valid event sequence for release consistency:
  - Process P3 has not performed an *acquire*, so there are no guarantees that the read of ‘x’ is consistent (data-store is simply not obligated to provide the correct answer).
  - P2 does perform an *acquire*, so its read of ‘x’ is consistent.

Release Consistency Rules

- A distributed data-store is “Release Consistent” if it obeys the following rules:

1. Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
2. Before a release is allowed to be performed, all previous reads and writes by the process must have completed.
3. Accesses to synchronization variables are *FIFO consistent* (sequential consistency not required).

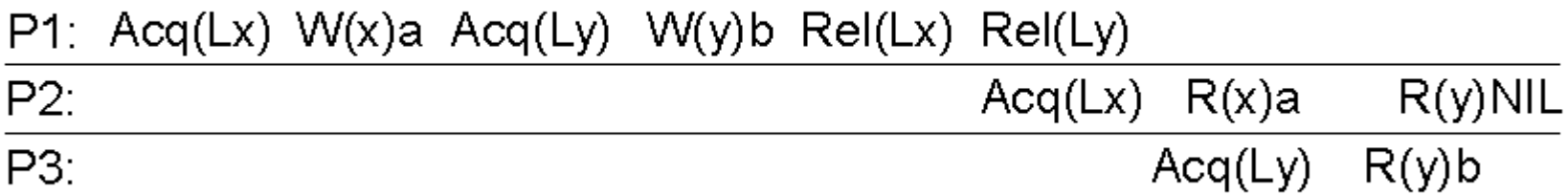
Introducing Entry Consistency

- A different twist is "**Entry Consistency**": **acquire** and **release** are still used, and the data-store meets the following conditions:

1. An acquire to synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
3. After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Entry Consistency: What It Means

- So, at an ***acquire***, all remote **changes** to guarded data must be brought **up to date**.
  - Before a write to a data item, a process must ensure that no other process is trying to write *at the same time*.



- **Locks** associate with **individual data items**, as opposed to the entire data-store.
  - NB: P2 read on ‘y’ returns NIL as no locks requested.

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 (non-unique) global timestamp.
Sequential	All processes see all shared accesses in the same order. Accesses 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 are made consistent when a critical region is entered.

(b)

(a) C**onsistency** models that do **not use synchronization** operations.

(b) Models that use **synchronization operations**: these require additional programming constructs, and allow programmers to treat the data-store *as if it is **sequentially** consistent*, when in fact it is not (“should” also offer the best performance).

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 datastores 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

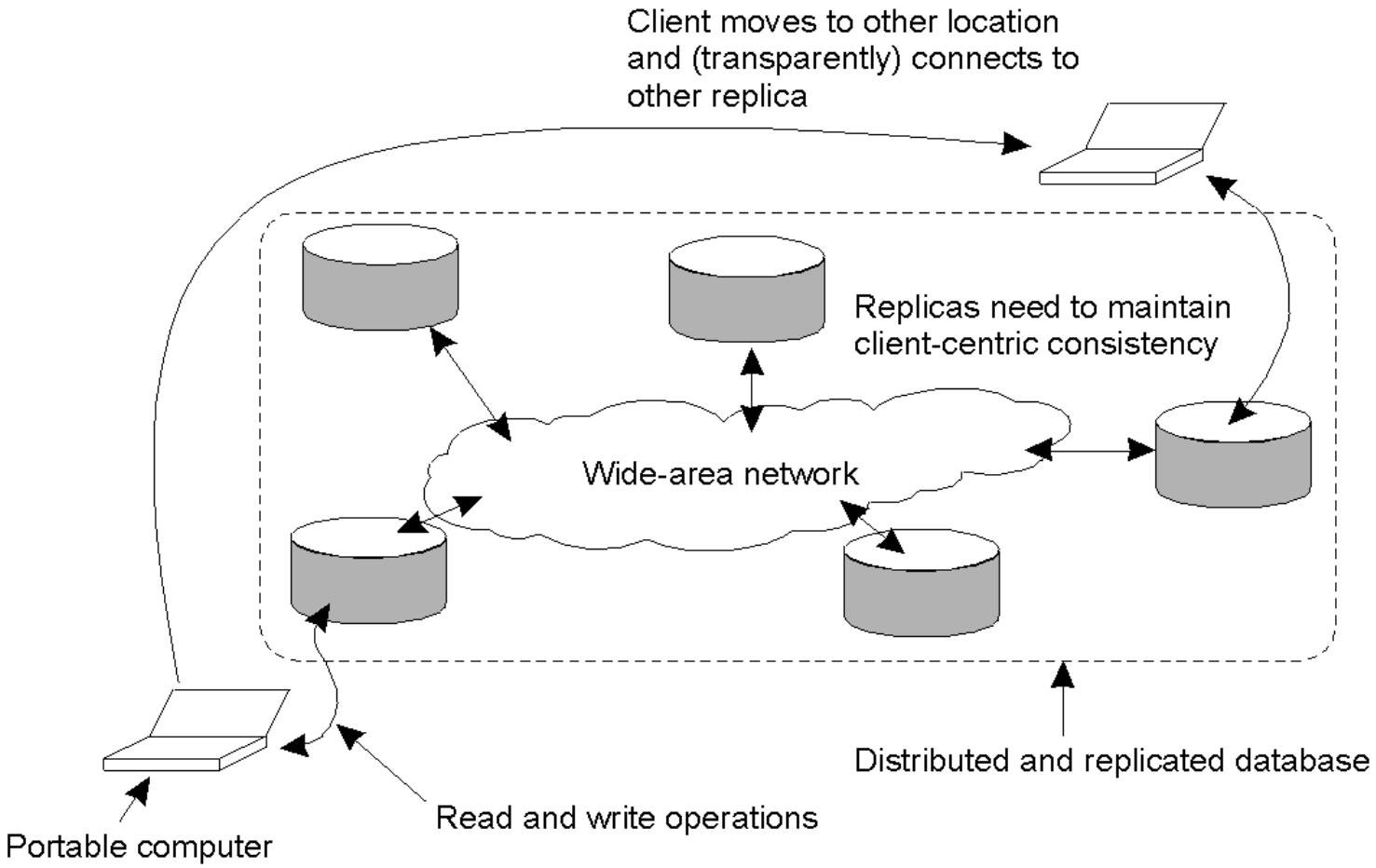
- How fast should updates (writes) be made available to read-only processes?
  - Think of most database systems: *mainly read*.
  - Think of the DNS: *write-write conflicts* do no occur.
  - Think of WWW: as with DNS, except that heavy use of client-side caching is present: *even the return of stale 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 clients are *mobile*.

Eventual Consistency: Mobile Problems



- 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.

An Example: The Bayou System

- The Bayou System implements 4 models of *Client-Centric Consistency*:
  - Monotonic-Read Consistency
  - Monotonic-Write Consistency
  - Read-Your-Writes Consistency
  - Writes-Follow-Reads Consistency

More on Bayou, 1 of 2

- 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 value or a more recent value.*

- 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.*

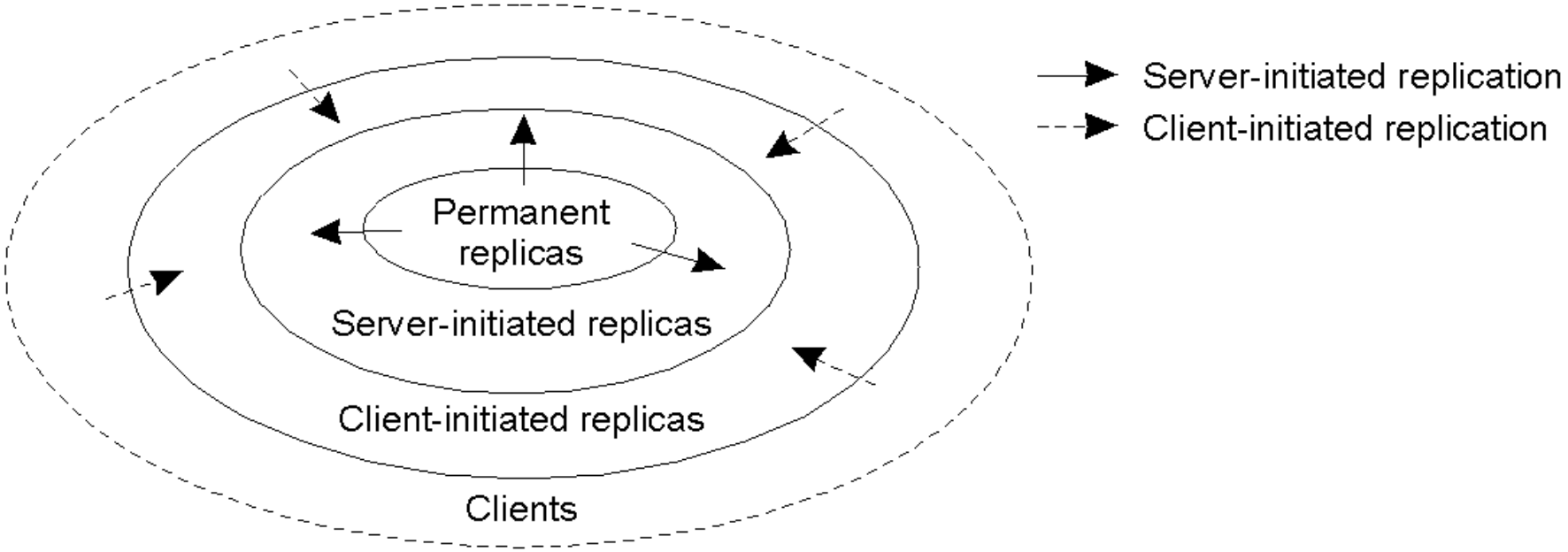
More on Bayou, 2 of 2

- 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.*

- 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.*

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 3 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 data-store. Typically used by web hosting companies to geographically locate replicas close to where they are needed most (often referred 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?

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 are 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 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 to the server.
   - Hence, if 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** schemes are possible with e.g., "**leases**":
  - Promise from a server to push updates to a client for a period of time.
  - Once lease expires, client reverts to pull-based (until another lease issued).

Epidemic Protocols

- This is an interesting class of protocols that can be used to implement ***Eventual Consistency***
  - NB: these protocols are used in Bayou
- The main concern is the propagation of updates to all the replicas in *as few a number of messages as possible*.
- This "update propagation model", intends to "infect" as many replicas as quickly as possible.
  - Of course, we are spreading updates, not diseases!

Epidemic Protocols: Terminology

- ***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 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.

The Anti-Entropy Protocol

- 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.

- Sooner or later, all the servers in the system will be infected (updated) - Works well.

The Gossiping Protocol

- 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… Oh dear.

The Best of Both Worlds

- A mix of anti-entropy and gossiping is regarded as the best approach to rapidly infecting systems with updates.
  - 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.
- Only problem remaining is the eventual removal of "old" death certificates (with which **timeouts** can help).

Consistency Protocols

- Specific implementation of a consistency model.
- The most widely implemented models are:

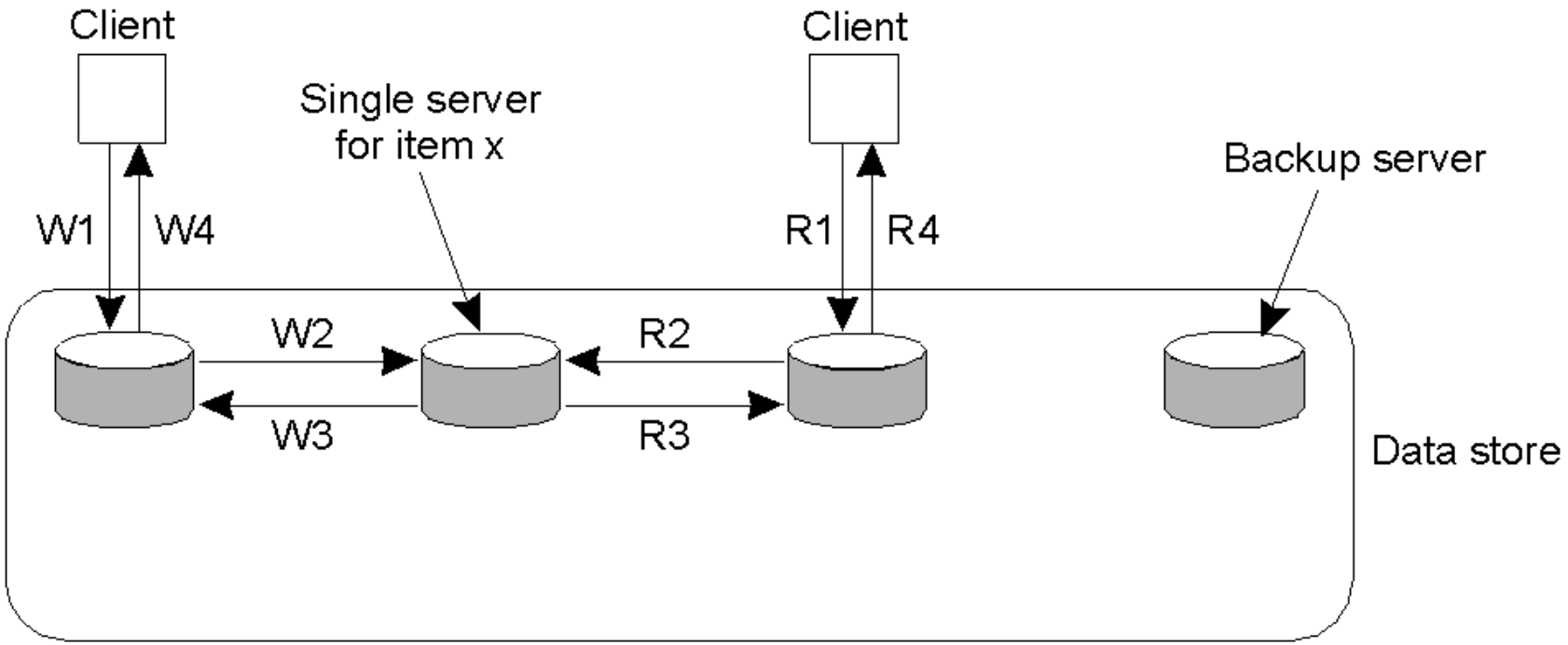
1. Sequential Consistency.
2. Weak Consistency (with sync vars).
3. Atomic Transactions (to be studied soon).

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:
  - Remote-Write.
  - Local-Write.

Remote-Write Protocols

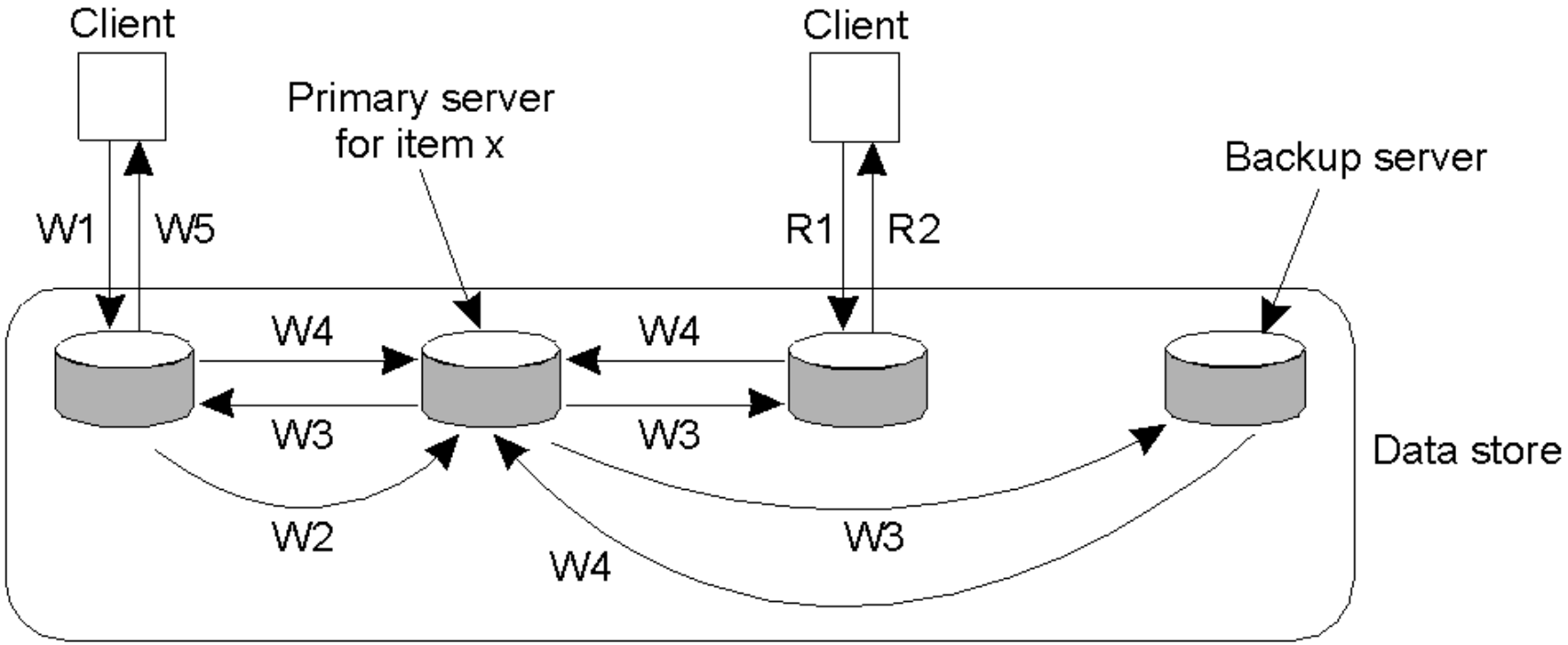
With this protocol, all writes are performed at a single (remote) server. This model is typically associated with traditional client/server systems.



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

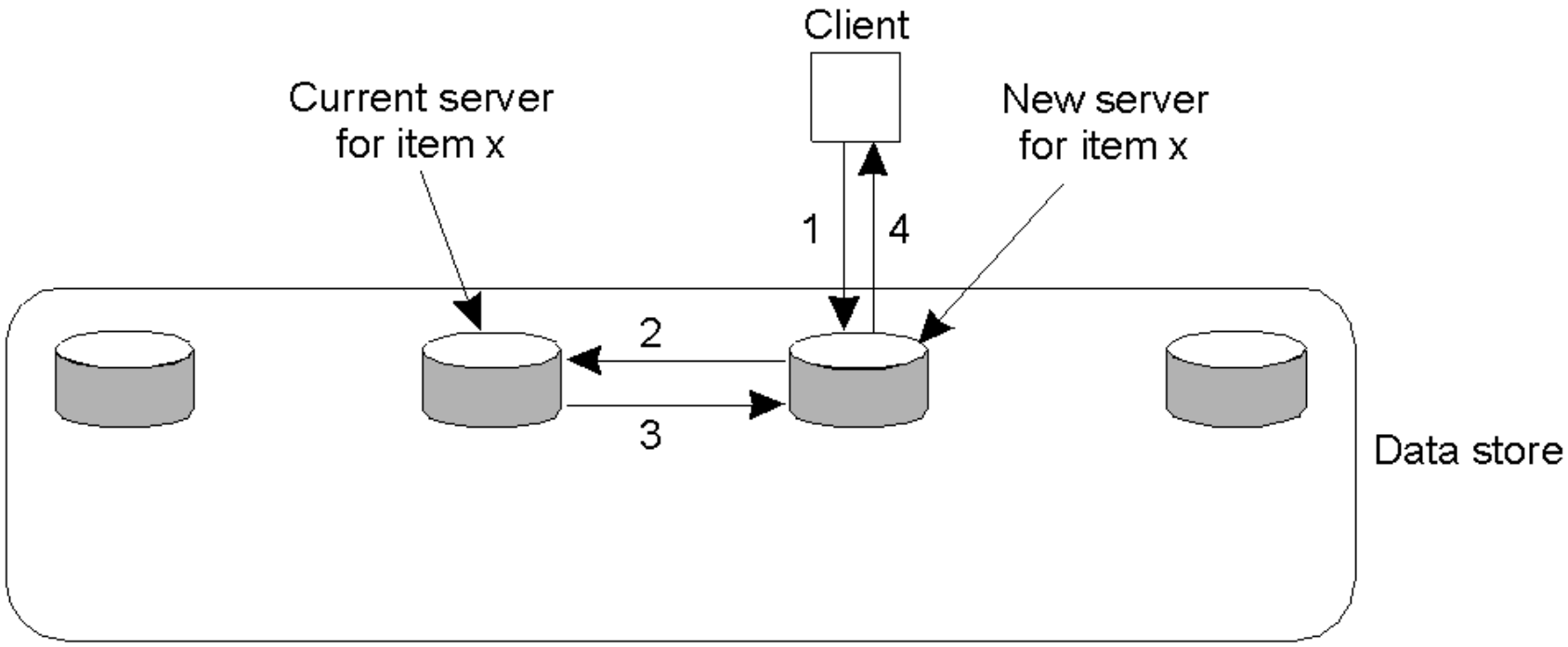
- **Bad: performance**
  - All of those **writes** can take a **long time** (especially when a "blocking write protocol" is used)
  - Using a non-blocking write protocol to handle the updates can lead to fault tolerant problems (which is our next topic).

- **Good: easy implementation**
  - 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



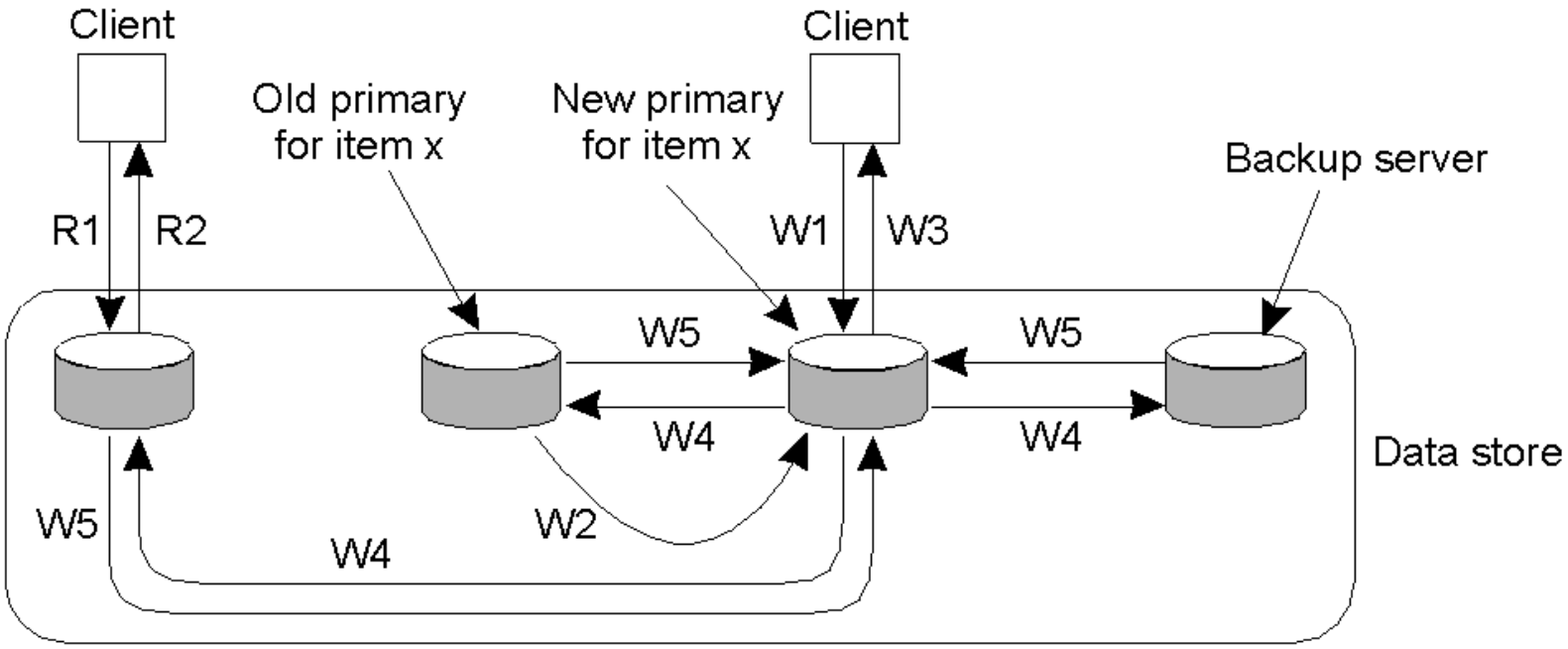
1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

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* 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 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.

Replicated-Write Protocols

- Writes can be carried out at *any* replica.
  - AKA: “Distributed-Write Protocols”

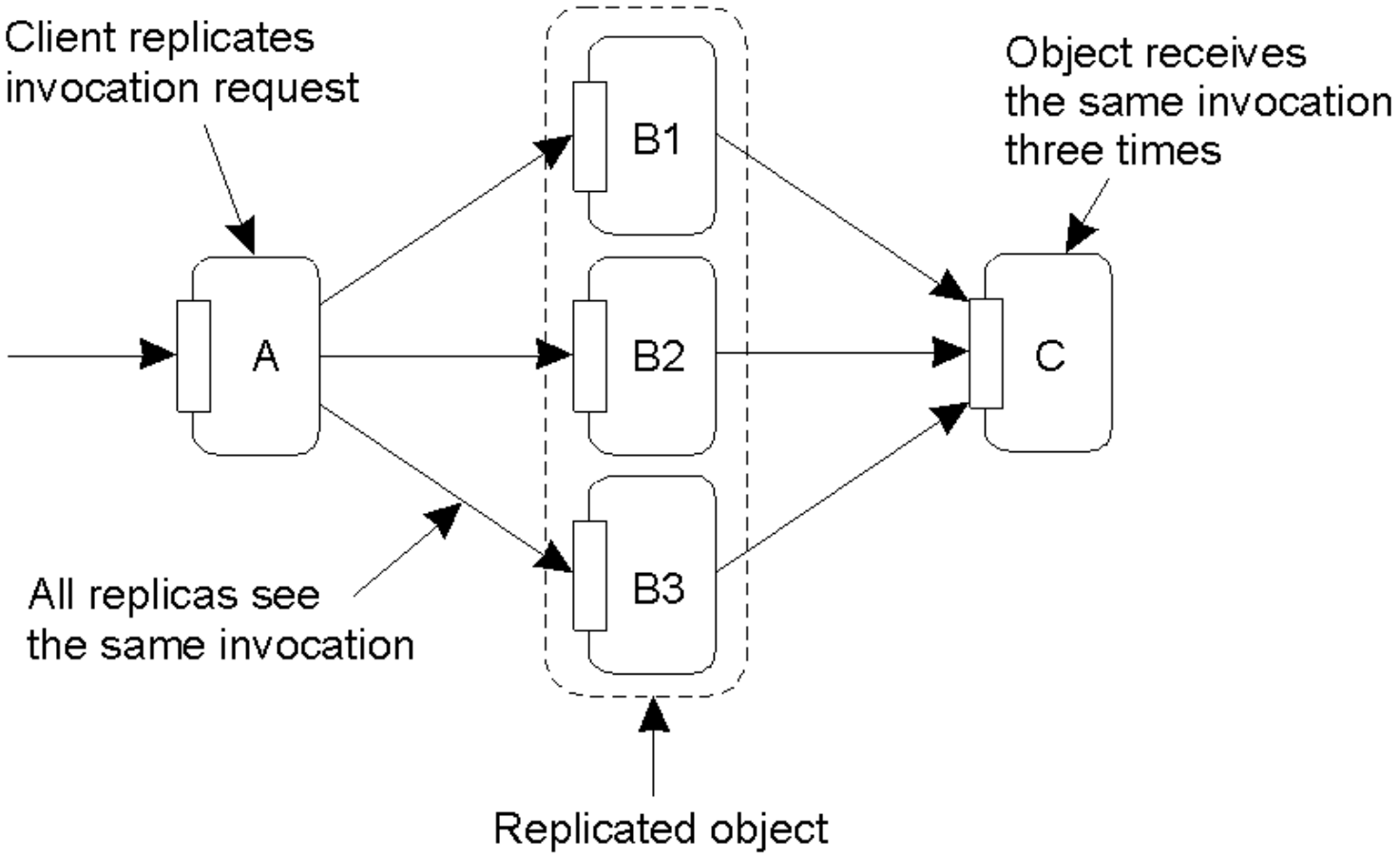
- There are 2 types:
  - Active Replication.
  - Majority Voting (Quorums).

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 DS.

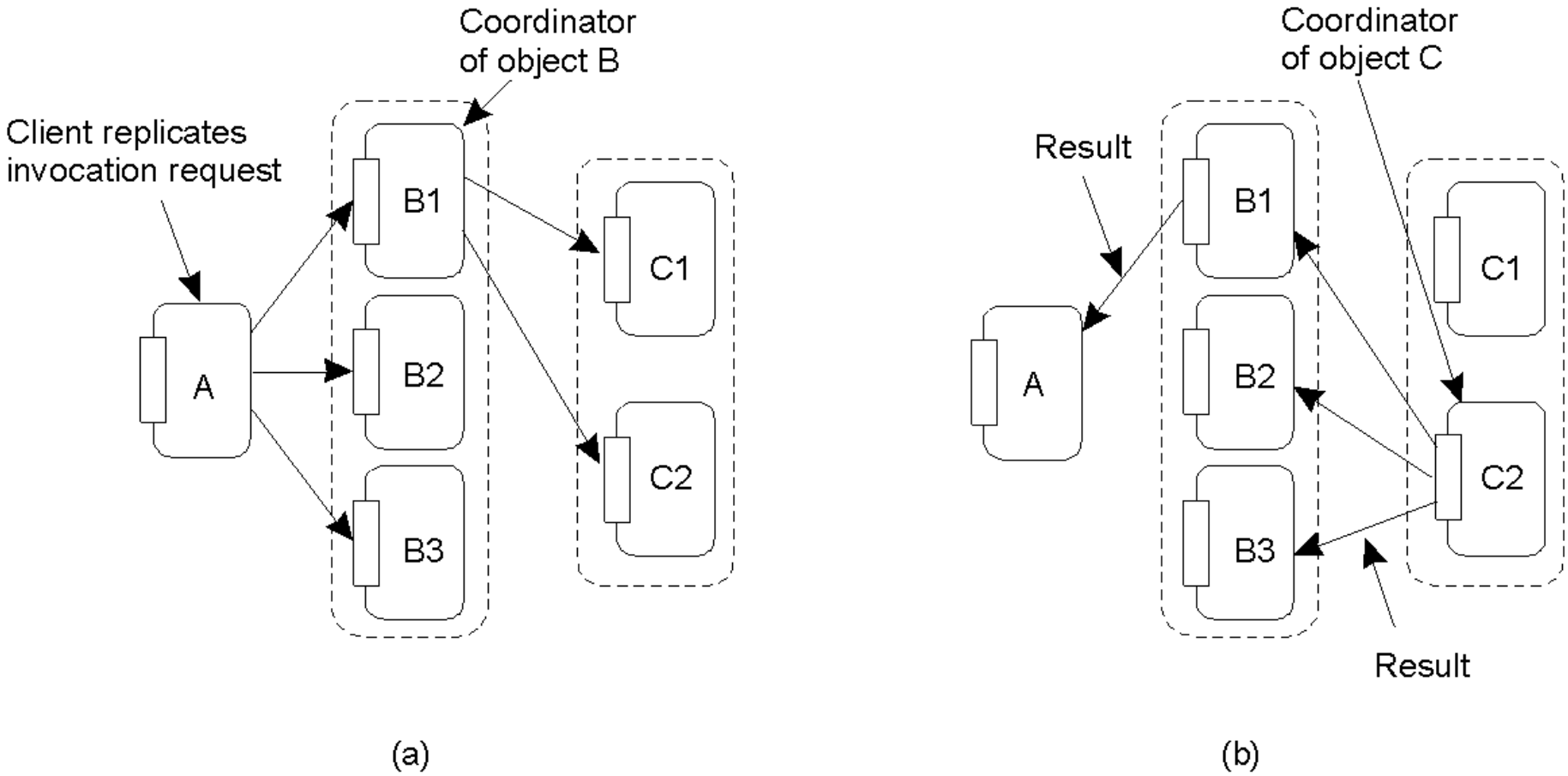
- An alternative/variation is to use a ***sequencer***, which is a process that assigns a unique ID# to each update, which is then propogated to all replicas.
  - Can lead to another problem: *replicated invocations*.

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’ is not 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'.

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.

Quorum Protocols: Generalisation

$$N_R + N_W > N$$

$$N_W > N/2$$

Cache-Coherence Protocols

- These are a special case, as the cache is typically controlled by the client, *not* the server.

- Coherence **Detection** Strategy: when are inconsistencies detected?
  - Statically at compile time: extra instructions inserted.
  - Dynamically at runtime: code to check with the server.

- Coherence **Enforcement** Strategy: how are caches kept consistent?
  - Server Sent: invalidation messages.
  - Update propagation techniques.

- Combinations are possible.

What about Writes to the Cache?

- ***Read-only Cache***: updates are performed by the server (i.e., **pushed**) or by the client (i.e., **pulled** whenever the client notices that the cache is *stale*).

- ***Write-Through Cache***: the client modifies the cache, then sends the updates to the server.

- ***Write-Back Cache***: delay the propagation of updates, allowing **multiple updates** to be made locally, then sends the most recent to the server (this can have a dramatic *positive* impact on performance).

Consistency and Replication: Summary

- Reasons for replication: improve…
  - *Performance*
  - *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”.

- Proposed solutions focus on **relaxation** of existing **consistency** constraints.

Summary, continued

- Various **consistency models** have been proposed:
  - *Strict*, *Sequential*, *Causal* and *FIFO* concern themselves with individual reads/writes to data items.
  - Weaker models introduce the notion of sync variables:
    - *Release* & *Entry* concerned with groups reads/writes.
  - These models are known as "Data-Centric".

- "Client Centric" models also exist:
  - Concerned with maintaining consistency for a single clients' access to the distributed data-store.
  - The ***Eventual Consistency*** model is an example.

End of Summary

- Distribute ("propagate") updates:
  - *WHAT* is propagated,
  - *WHERE* it is propagated
  - *WHOM* is propagated

- *Distribution Protocols* and *Consistency Protocols* designed to facilitate propagation of updates.

- Most widely implemented schemes are those that support *Sequential Consistency* or *Weak Consistency* with *Sync Variables*.