Chapter 6.1. Introduction

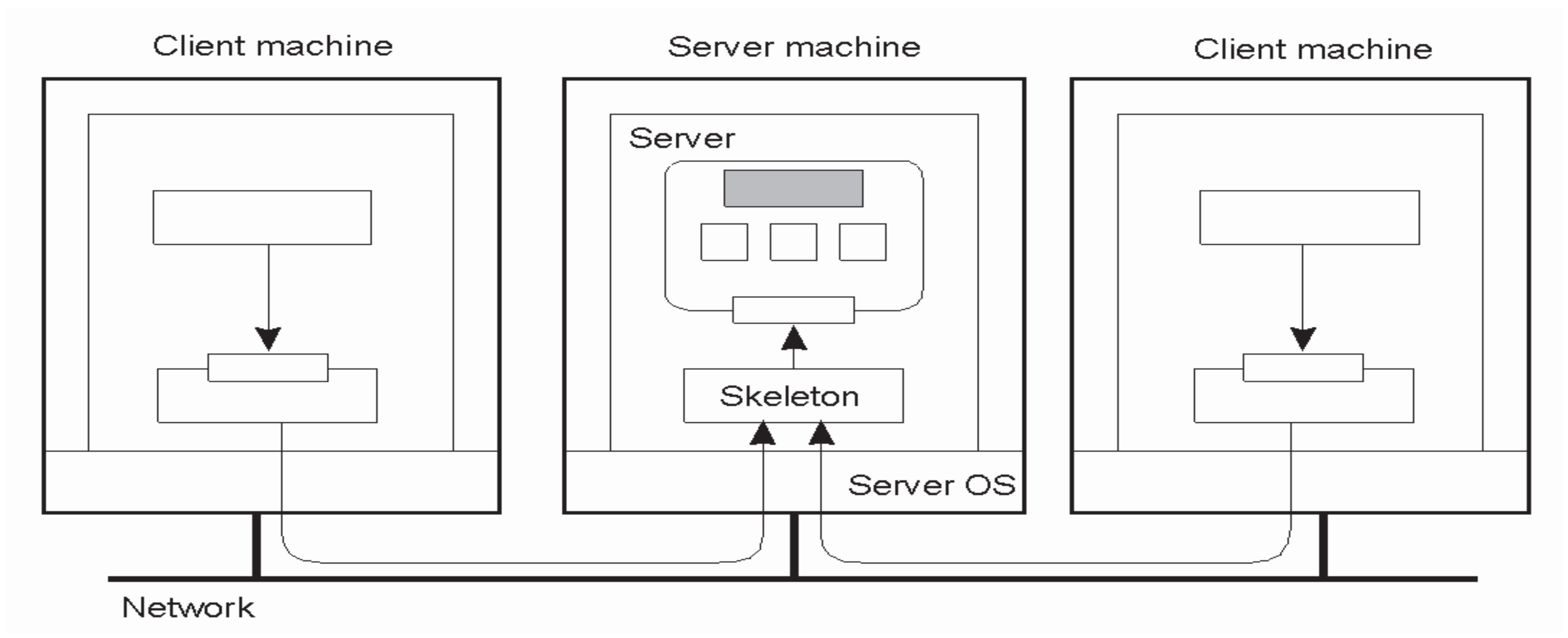
- Why Replication
  - Enhance reliability
  - Improve performance

- Two issues
  - Placement of replicas & how updates are propagated
  - How replicas are kept consistent

Object Replication

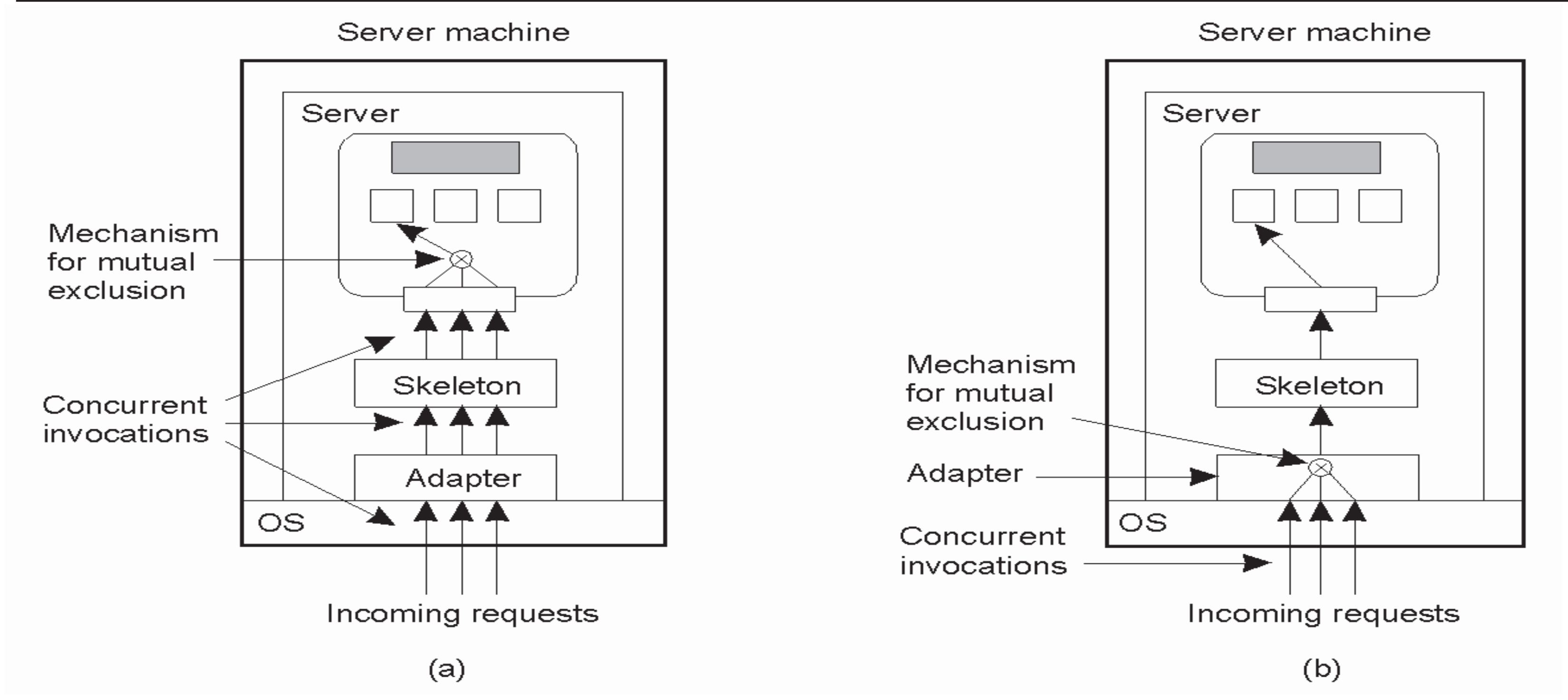
- Replicate objects instead of only data.
- Benefit:
  - ***Encapsulation*** since only can be accessed via predefined interface
- But, before considering replication, we need so solve how to protect object from multiple-access?
  - Object itself implement the means
    - Synchronized method in Java
  - Completely unprotected but objects adapter
    - E.g., object adapter uses a single thread per object
  - See the following next slide

Problem : Concurrent Object Access



Organization of a distributed remote object shared by two different clients.

Concurrent Object Access: Solutions



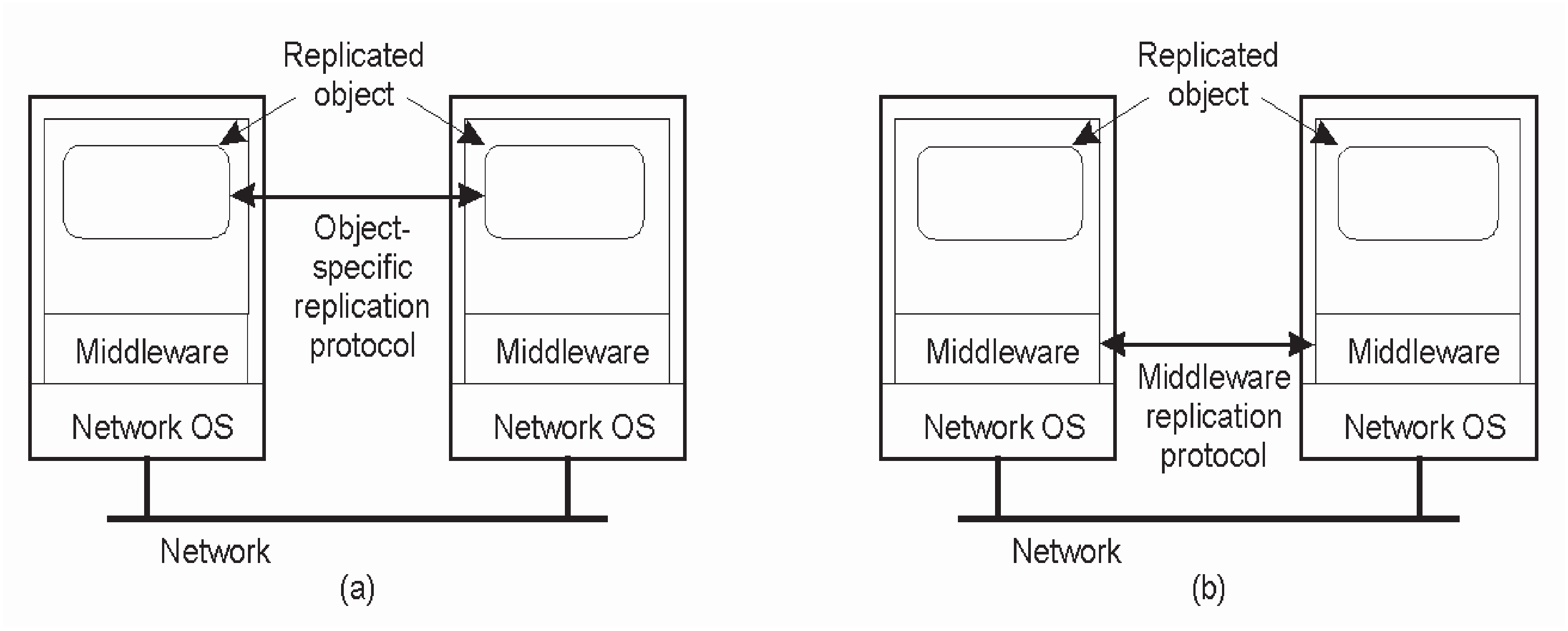
(a) A ***remote object*** capable of handling concurrent invocations on its own.

(b) A remote object for which an ***object adapter*** is required to handle concurrent invocations (relies on middleware).

Object Replication (Cont.)

- Replicate a shared remote object also faces consistency problem
- Solutions
  - The object itself is “aware” that it is replicated.
    - Object responsible for its replicas stay consistent in the presence of concurrent invocations
    - Adv. adopt object-specific strategies
  - Distributed system take care
    - Adv. simplicity for AP developers.

Object Replication: Solutions



a) A distributed system for replication-aware distributed objects.

b) A distributed system responsible for replica management – the *most common approach.*

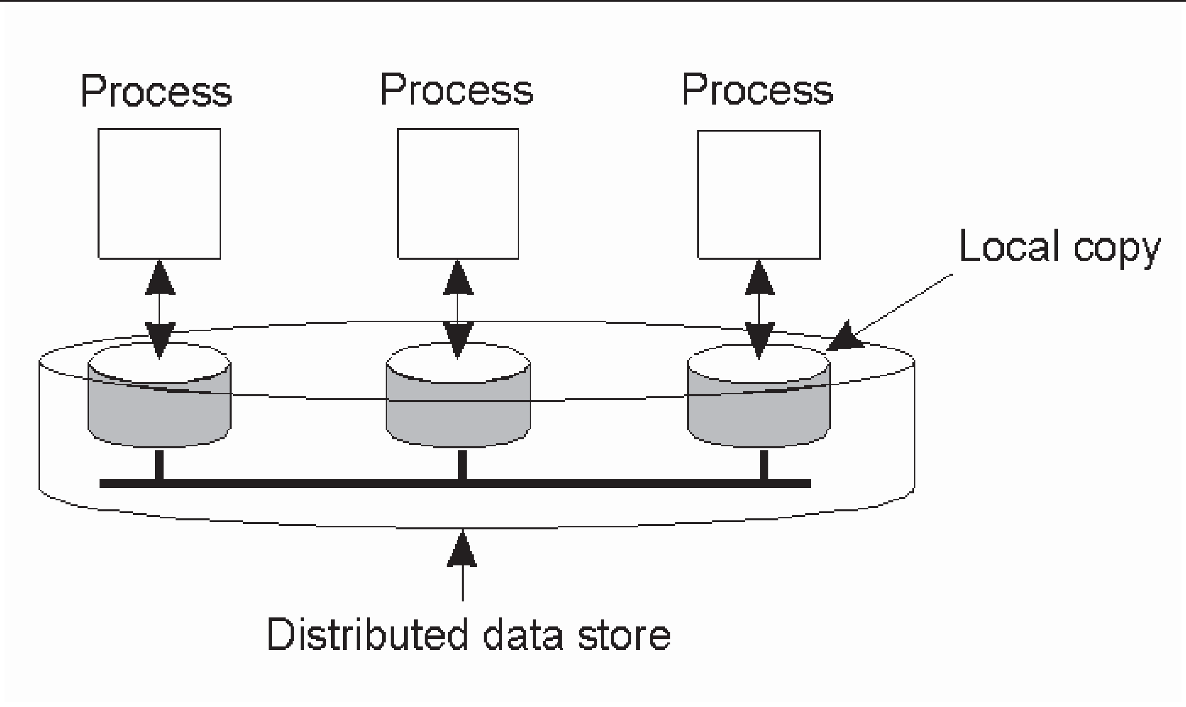
Replication as Scaling Technique

- Scalability issues trade-off
  - Performance gain since reduction access time
  - Bandwidth may lose since keep copies up to date
    - Keeping multiple copies consistent would require ***global synchronization***
- Scalability? v.s. Consistency?
- Sol.: loosen the consistency constraints
  - But copies may not always be same

Chapter 6.2. Data-Centric Consistency Models

- A ***consistency model*** is a contract between a DS data-store and its processes
  - Since it is difficult to define precisely which write operation is the last one
  - If the processes agree to the rules, the data-store will perform properly and as advertised

Data-Centric Consistency Models



- A data-store can be read from or written to by any process in a DS.
- 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.

Data-Centric Consistency Models

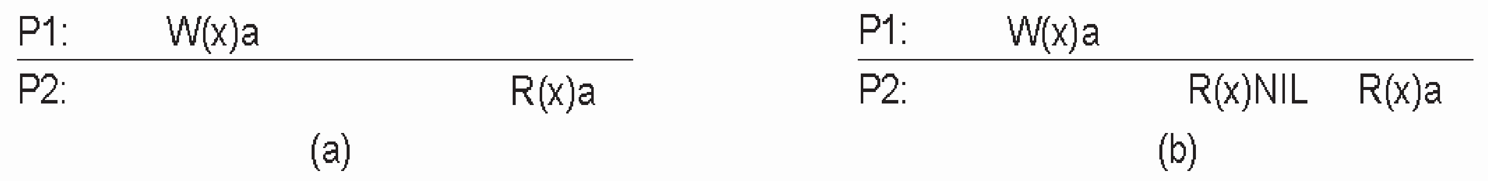
- Strict consistency
- Linearizability & Sequential consistency
- Causal consistency
- FIFO consistency
- Weak consistency
- Release consistency
- Entry consistency

Strict Consistency

- *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).*
- Most stringent consistency
  - "Most recent" must be unambiguous
- Problem: relies on *absolute global time*
  - But 後發先至
  - Impossible to implement in a DS

Strict Consistency Diagrams

- Consistency Model Diagram Notation
  - *Wi(x)a* – a write by process *i* to item *x* with a value of *a*.
  - *Ri(x)b* – a read by process *i* from item *x* producing the value *b*.

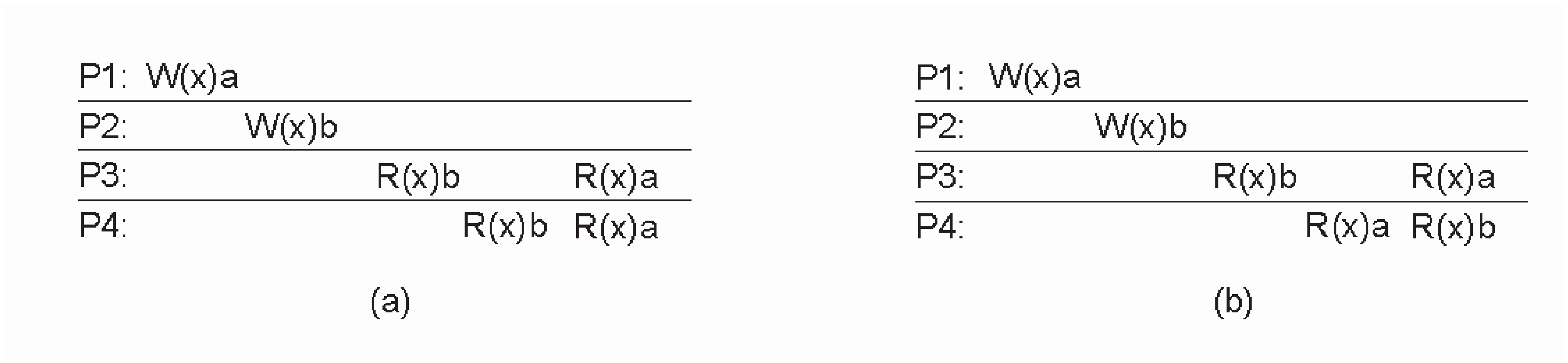


- Behavior of two processes, operating on the same data item:
  - (a) A strictly consistent data-store.
  - (b) A data-store that is not strictly consistent.

Sequential Consistency

- *Any **valid** interleaving of read and write operations is acceptable and all processes see the same interleaving of operations*
- Nothing about time
- Weaker than strict consistency

Sequential Consistency Diagrams



a) A sequentially consistent data-store
   - P3 and P4 sees the same interleaving of write operations by P1 and P2 (see P2 first and then P1)

b) A data-store that is not sequentially consistent
   - P3 and P4 do not see the same interleaving of write operations by P1 and P2

Sequential Consistency Example

Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

Fig. 6-7: Three concurrently executing processes.

x = 1;	x = 1;	y = 1;	y = 1;
print (y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x,z);	print (x, y);	z = 1;
print (x, z);	print(y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
(a)	(b)	(c)	(d)

Four valid execution sequences for the processes P1, P2, and P3.

Sequential Consistency Example (Cont.)

- There are 4 (actually more) valid sequences
- The contract between processes and the distributed shared data store
  - The DS guarantee sequential consistency
  - The process must accept all of these sequences as valid results
- Problem with sequential consistency
  - It is proven that $r+w >= t$
    - r: read time, w: write time, t: minimal packet transfer time between nodes
  - Adjust the protocol to favour reads could cause impact on write performance

Linearizability

- Linearizability
  - *All processes see the same sequential order of operations*
  - *The operations of each individual process appear in this sequence follows the order specified by its program*
  - *If timestamp(OP_1) < timestamp(OP_2), OP_1 should proceed OP_2 in the sequence*
- i.e., *sequential consistency + timestamp each operation*
- Weaker than strict consistency but stronger than sequential consistency
- Linearizability consistency is also *sequential consistency*
  - But take ordering according to *a set of synchronized clocks* into account

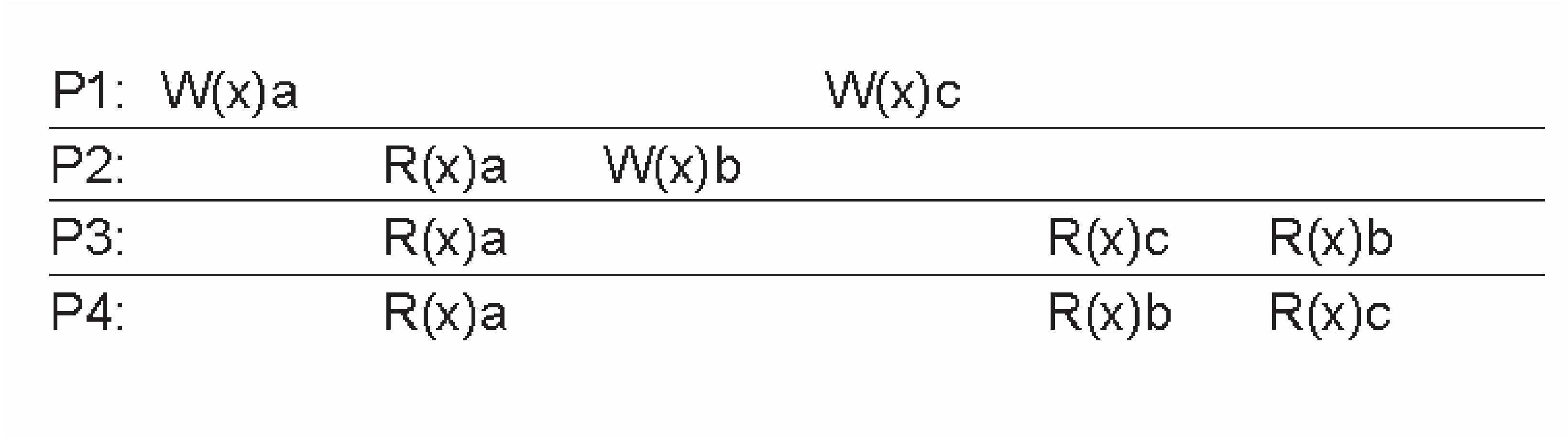
Causal Consistency

- 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*,
  - Causal consistency requires that every other process see event A, then event B.
- Operations that are not causally related are said to be ***concurrent.***

Causal Consistency (Cont.)

- Causal Consistency
  - *Writes that are potentially causally related must be seen by all processes in the same order*
  - *Concurrent events may be seen in a different order on different machine*
- Implementation: construct and maintain a *dependency graph*
  - Keep with operation is dependent on which operation
  - Use vector timestamp in previous section
    - Mention later

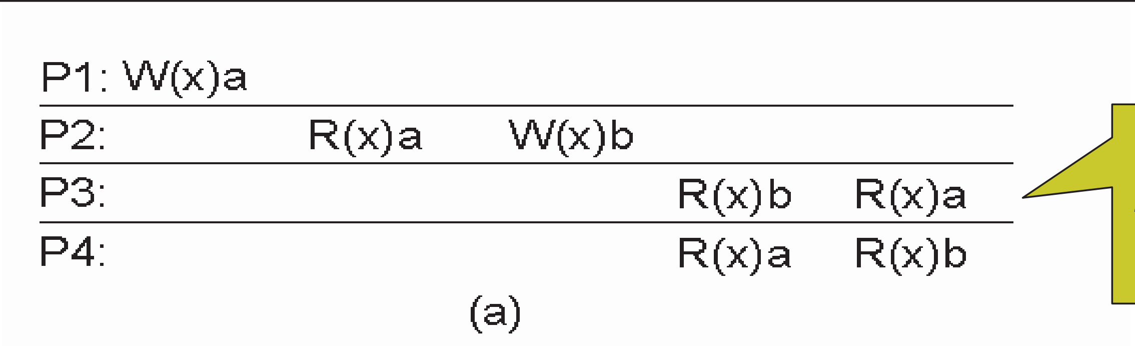
Example1



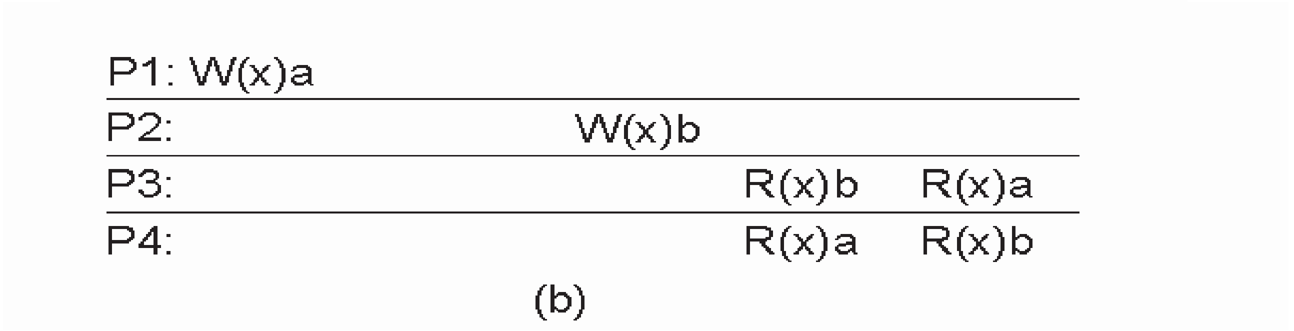
P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

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

Example2



P_3 *and* P_4 *see different order* (x)



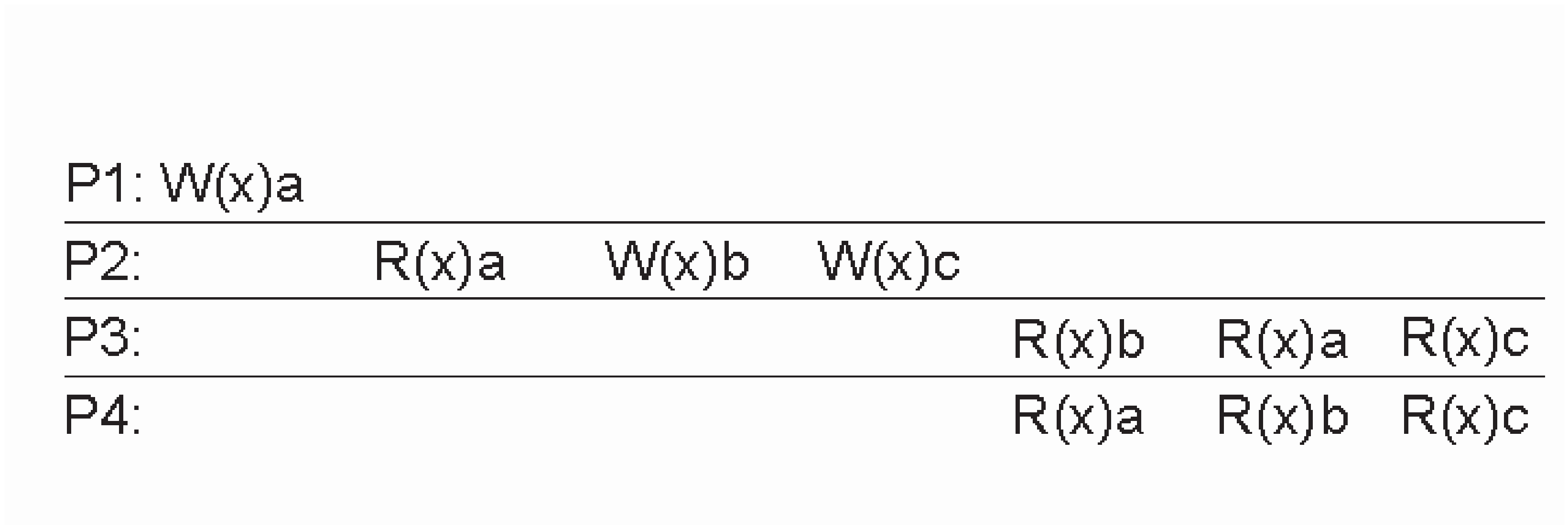
a) Violation of causal-consistency – assume $W_2(x)b$ is related to $W_1(x)a$ (all processes should see them in the same order).

b) A causally-consistent data-store: assume the two writes are now *concurrent*. The reads by P3 and P4 are now OK.

FIFO Consistency

- *Writes done by a single process are seen by all other processes in the order in which they were issued*
- *But writes form different processes may be seen in a different order by different processes*
- Easy to implement.
  - Tag each write operation with a *(process, sequence number)* pair

Example1



```
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
```

A valid sequence of FIFO consistency events.

Example2

(a)	(b)	(c)
x = 1;	x = 1;	y = 1;
print (y, z);	y = 1;	print (x, z);
y = 1;	**print(x, z);**	z = 1;
print(x, z);	print (y, z);	**print (x, y);**
z = 1;	z = 1;	x = 1;
print (x, y);	print (x, y);	print (y, z);
Prints: 00	Prints: 10	Prints: 01

- Statement execution as seen by three processes from the Fig. 6-7.
- Under FIFO consistency, different processes may see the statements executed in a different order
  - But not allowed in a sequentially consistent store

Example 3

Process P1	Process P2
x = 1;	y = 1;
if (y == 0) kill (P2);	if (x == 0) kill (P1);

- *What you expect? (Assume x=y=0 initial)*
  - *P1 is killed, P2 is killed or neither is killed (if the two assignments go first)*
  - *Actually, in FIFO consistency, both P1 and P2 would be killed*
    - *P1 reads $R_1(y)0$ before it sees P_2's $W_2(y)1$*
    - *P2 reads $R_1(x)0$ before it sees P_1's $W_x(x)1$*
    - *But impossible in sequential consistency*

Weak Consistency

- Cope with ***synchronization variable***
  - Used to synchronize all local copies of the data store
- Three properties of weak consistency
  - Accesses to ***synchronization variables*** associated with a data store are ***sequentially consistent***
  - No operation on a *synchronization variable* is allowed to be performed until all previous writes have been completed everywhere
  - 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 …
- 1st point: all processes see all operations on *synchronization variables* in the same order
  - If *P1* calls *syn.(S1*) and *P2* calls *syn.*(*S2*), all processes see either "*sync.(P1)sync.(P2)*" or "*sync.(P2)sync.(P1)*"
- 2nd point: sync. ***flushes the pineline***
  - All previous writes are guaranteed to be done as well when the synchronization is done
- 3rd point: when data item are accessed (read or write), all previous synchronization will have been completed

Weak Consistency (Cont.)

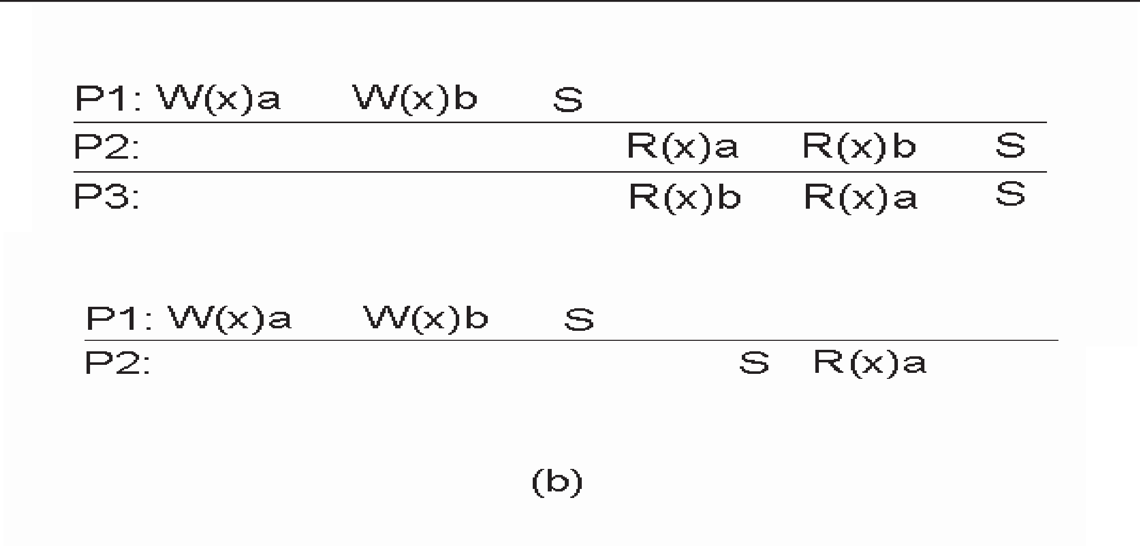
- Enforce consistency on *a group of operations*
  - Suitable when most accesses coming in clusters

- Limit only the time ***when*** consistency holds
  - When we issue an synch. variable

Example 1

- Like memory management
  - The value of a variable may be temporary put in register and updated in register
  - Result in a inconsistent memory
  - Only when some events occur, the memory will then be kept consistent with register

Example 2



P1:	W(x)a	W(x)b	S			
P2:				R(x)a	R(x)b	S
P3:				R(x)b	R(x)a	S

P1:	W(x)a	W(x)b	S		
P2:				S	R(x)a

(b)

a) A valid sequence of events for weak consistency. This is because *P2* and *P3* have not yet 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*'.

Release Consistency

- Problem of weak consistency: when synch., we have no idea
  - Either the process is finished writing the shared data (*exit critical section*)
  - Or the process is about to start reading (*enter critical section*)

Release Consistency (Cont.)

- As a result, when issues synch., weak consistency must guarantee
  - All locally initiated writes have been completed that are propagated to all other copies
  - Gather in all writes from other copies
- However, more efficient if we can tell the difference between in *entering* a C. S. or *leaving* one

Release Consistency (Cont.)

- Sol. two sync variables can be used
  - *acquire* and *release*
  - Leads to the release consistency model.
- *When acquire,*
  - *Ensure that all the local copies of the **protected data** are brought up to date*
- *When release*
  - *Protected data that have been changed are propagated out to all local copies*
- Protected data
  - Shared data that are kept consistent
- Thus, before release, all write and read take place on its local cache

Example

```
P1:  Acq(L)   W(x)a    W(x)b    Rel(L)
P2:                                     Acq(L)  R(x)b   Rel(L)
P3:                                                            R(x)a
```

- A valid event sequence for release consistency.
- *P3* has not performed an *acquire*, so there are no guarantees that the read of ‘*x*’ is consistent.
- *P2* does perform an *acquire*, so its read of ‘*x*’ is consistent.
  - Even if Acq(L) in *P2* before Rel(L) in *P1*, just delay until the release had occurred

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.*

Release Consistency

- When a release is done
  - Push out all the modified data to other processes that have a copy of the data
  - All of these processes get everything that has changed
- But, if they actually will need it?
- Sol. ***Lazy release consistency***
- The normal release consistency can be called ***eager release consistency***

Lazy Release Consistency

- At the time of release, nothing is sent anywhere
- When an acquire is done
  - Then try to get the most recent of data from the process holding the data
  - How to guarantee the most recent=> timestamp

Entry Consistency

- Similar to *release consistency*
  - Use *acquire* and *release* at the start and end of each critical section
- However, unlike to release consistency
  - Each ordinary shared data item is associated with a *synchronization variable*
  - E.g., elements of an array are associated with different synch. variable
  - Increase the amount of ***parallelism***

Entry Consistency Rules

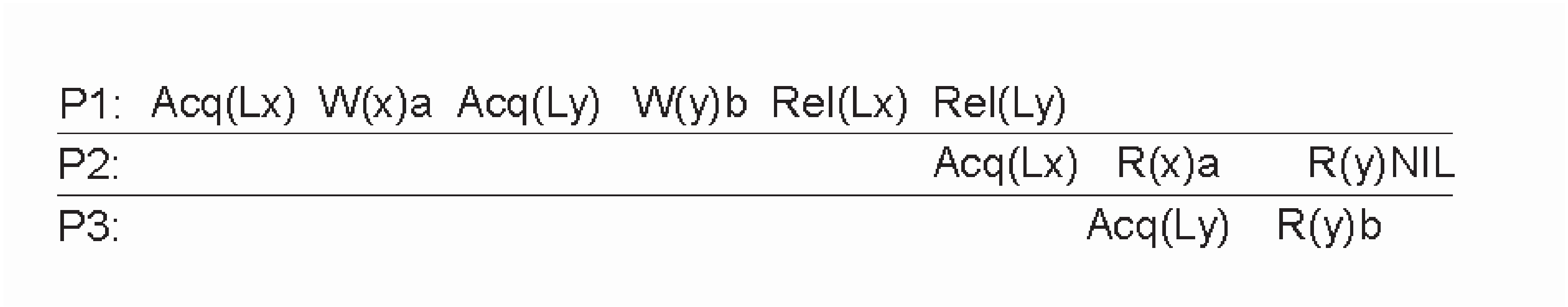
- *An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.*
- *Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.*
- *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 Rules (Cont.)

- So …
- 1st point: an *acquire* may not complete until all guarded shared data are up to date
- 2nd point: before updating a shared data item
  - A process must enter a C. S. in *exclusive mode*
    - To make sure that no other process is trying to update it at the same time
- 3rd point: if a process wants to enter a C.S. in *nonexclusive mode*
  - Check the owner of the synch. variable to fetch the *most recent copies of the guarded shared data*

Example

A valid event sequence for entry consistency.


P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)
P2: Acq(Lx) R(x)a R(y)NIL
P3: Acq(Ly) R(y)b

Locks associate with individual data items, as opposed to the entire data-store. Note: P2's read on 'y' may return NIL as no locks have been 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 (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 models that do not use synchronization operations

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) Models that do use synchronization operations.

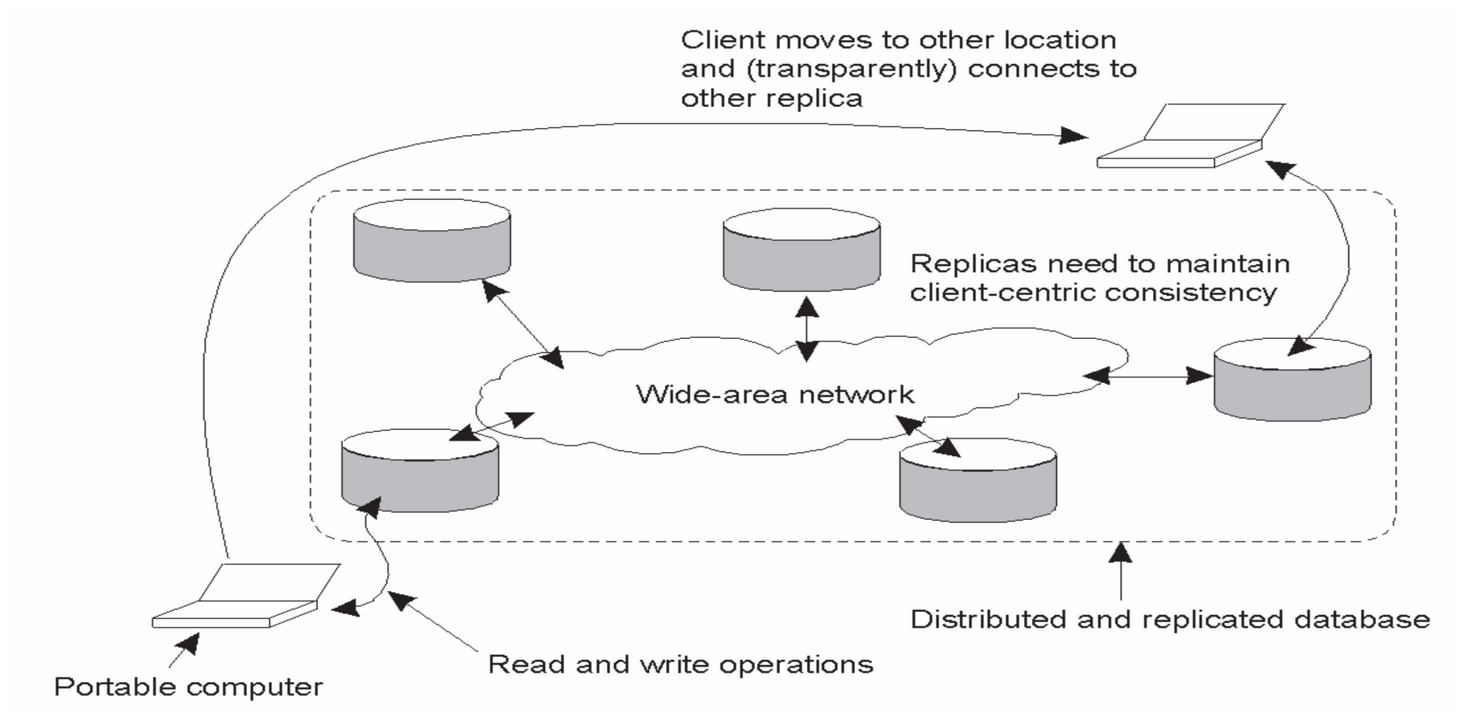
Chapter 6.3. Client-Centric Consistency Models

- Previous consistency models concern a data-store in the presence of concurrent read/write operations.
- However, some distributed data store is *lack of simultaneous updates*.
- Example
  - DNS: *write-write conflicts* do no occur
  - 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*

Eventual Consistency

- A very weak consistency model
- The only requirement is that all replicas will *eventually* be the same
  - All updates must be guaranteed to propagate to all replicas … *eventually*
- Problem:
  - Eventual consistency works well if every client always updates the same replica
  - But how about *mobile* clients ?

Eventual Consistency: Mobile Problems


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

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

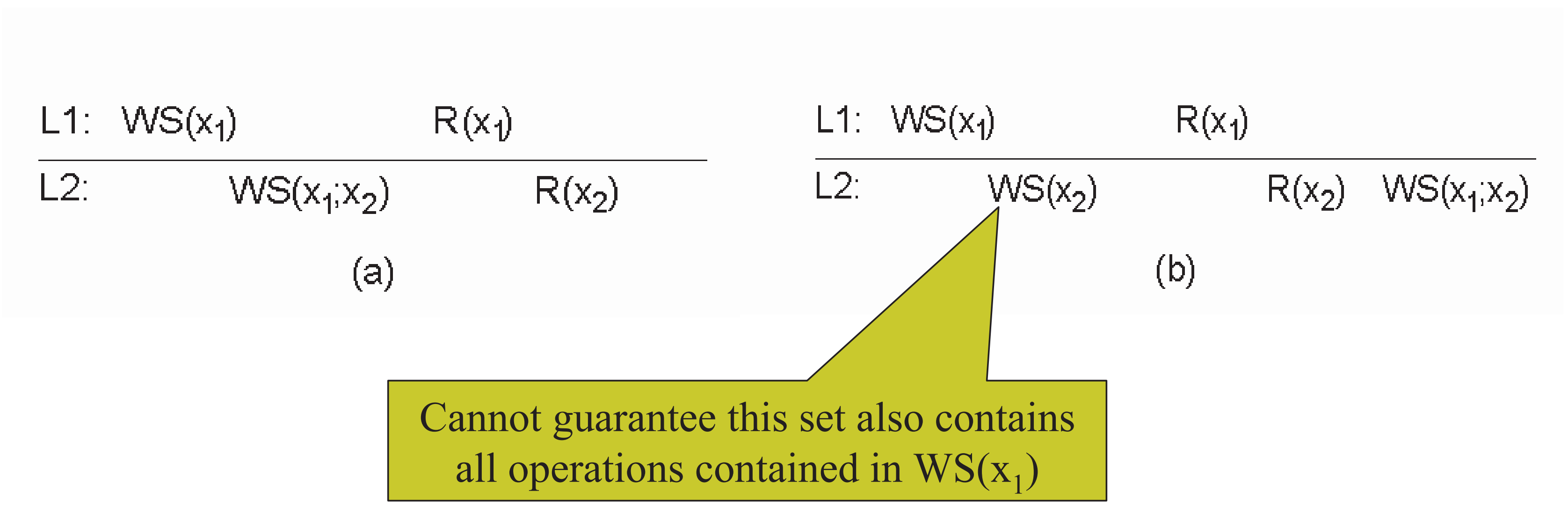
Client-Centric Consistency Models

- Sol. ***client-centric consistency model***
  - Maintain a consistent view of things *for a single client*.
- In other words, 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.
- Four models of *client-centric consistency*:
  - Monotonic-Read Consistency
  - Monotonic-Write Consistency
  - Read-Your-Writes Consistency
  - Writes-Follow-Reads 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 value or a more recent value.*
  - A process see a value of x, it will never see an older version of x at a later time.
- Notation: $WS(x_i)$
  - A series of write operations at L_i that took place since initialization
  - If operations in $WS(x_i)$ have also been performed at local copy L_j at a later time, we write: $WS(x_i, x_j)$

Example


L1: WS(x1) R(x1)
L2: WS(x1;x2) R(x2)
(a)
L1: WS(x1) R(x1)
L2: WS(x2) R(x2) WS(x1;x2)
(b)
Cannot guarantee this set also contains all operations contained in WS(x1)

- The read operations performed by a single process *P* at two different local copies of the same data store.

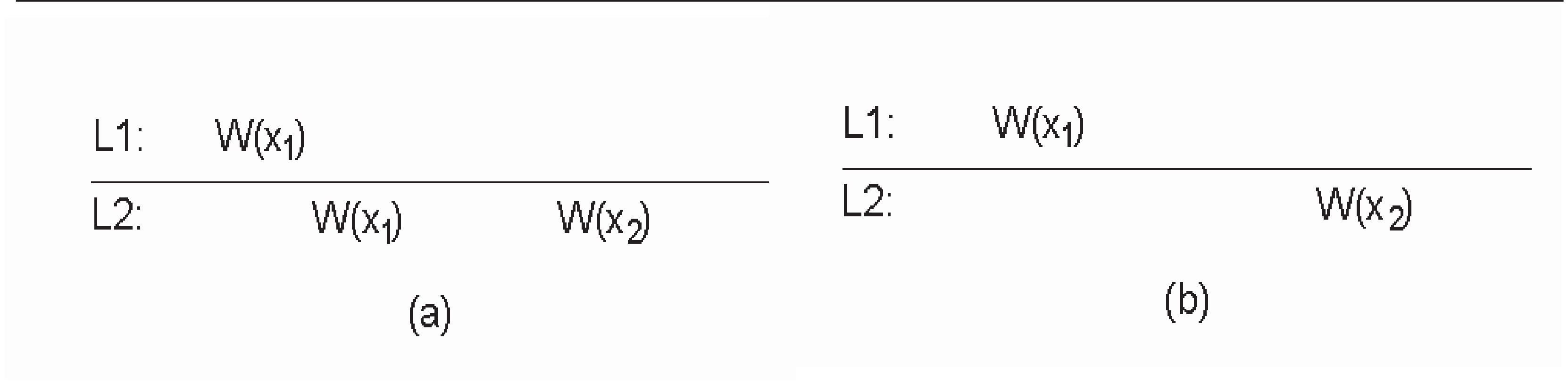
a) A monotonic-read consistent data store since WS(x_1) is part of WS(x_2)

b) A data store that does not provide monotonic reads.

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.*
- Resembles data-centric FIFO consistency
  - Write operations by the same process are performed in the correct order
  - Except we now consider only for a single process.
    - Instead of a collection of concurrent processes

Example



- The write operations performed by a single process *P* at two different local copies of the same data store

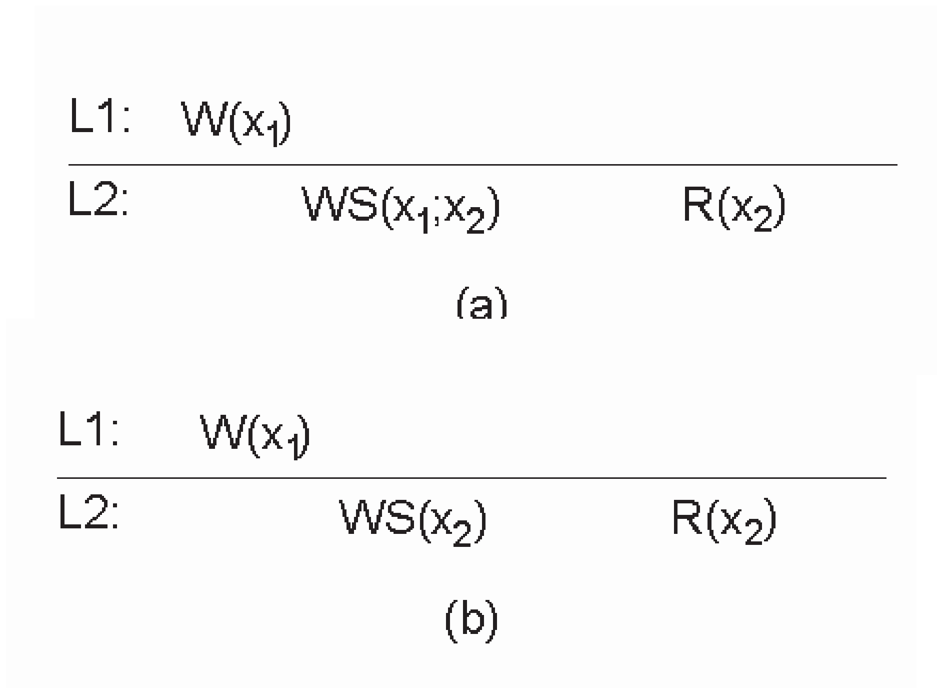
a) A monotonic-write consistent data store.

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

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 write operation is always completed before a successive read operation by the same process, no matter where the read takes place.*

Example

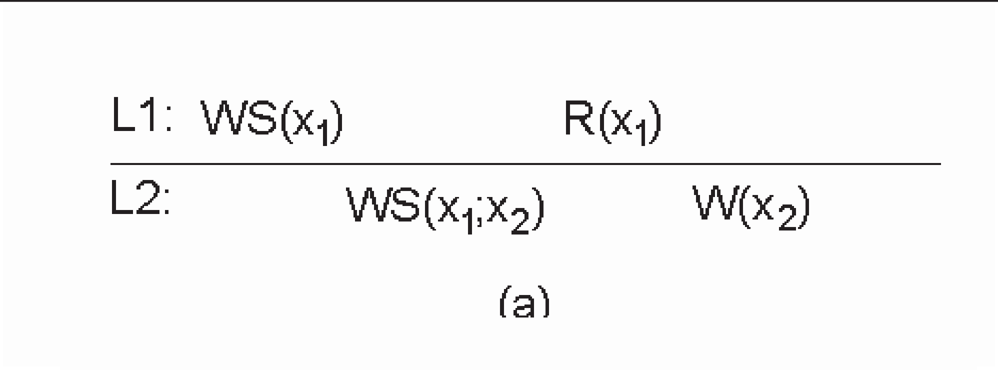


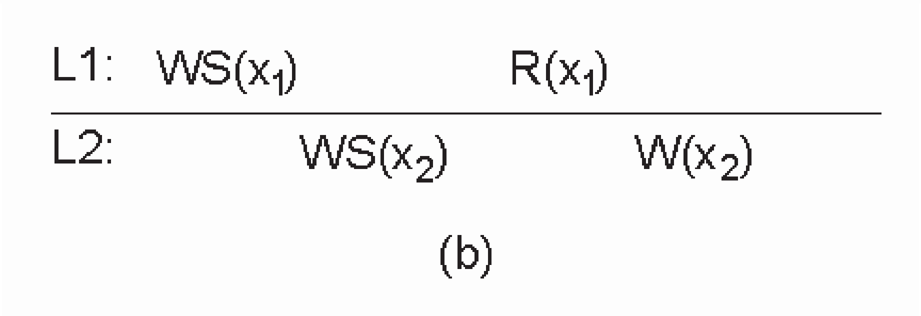
a) A data store that provides read-your-writes consistency.
b) A data store that does not.

Write 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.
- *Any successive write operation by a process on x will be performed on a copy of x that's up to date with the value most recently read by that process*

Example





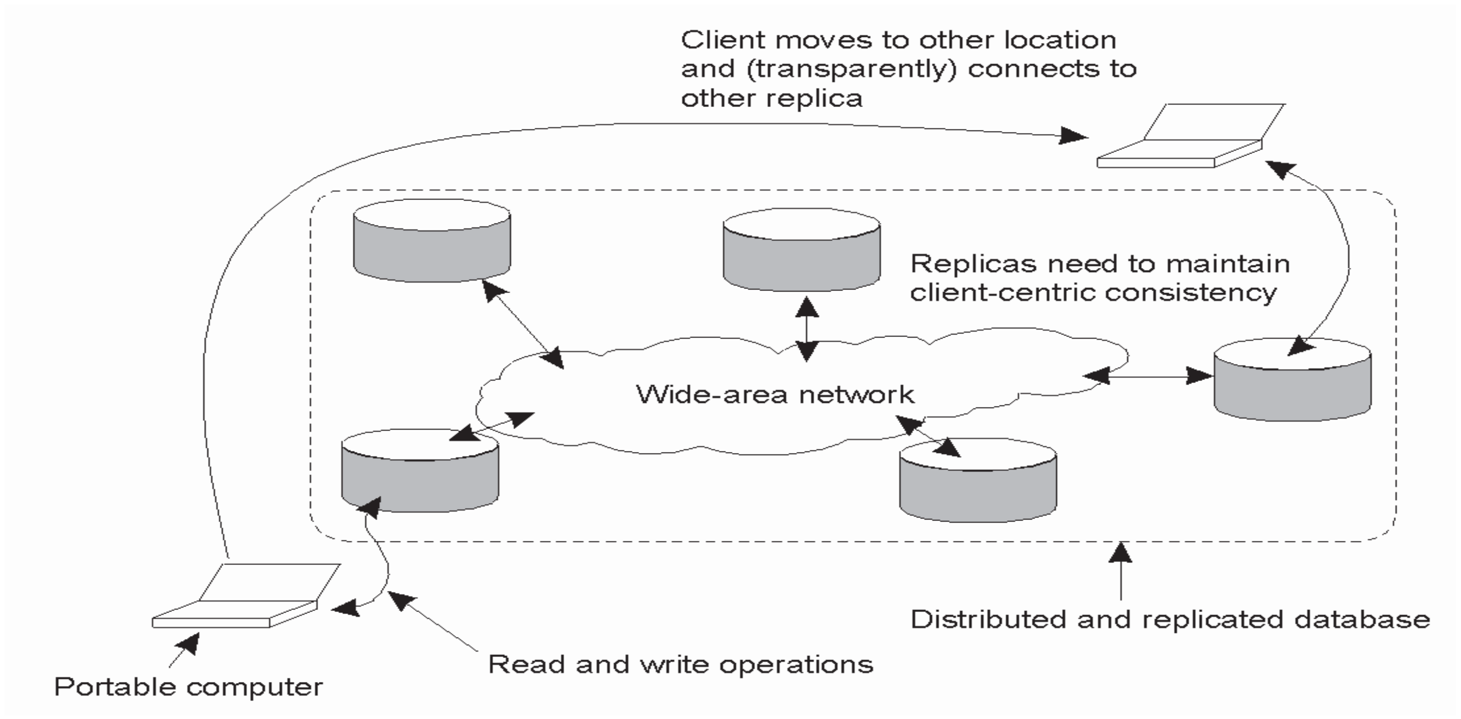
a) A writes-follow-reads consistent data store
b) A data store that does not provide writes-follow-reads consistency

Chapter 6.4.Distribution Protocols

- In this section
  - Discuss ways to propagate updates to replicas

- ***Replica Placement***
- Update Propagation
- Epidemic Protocols

Eventual Consistency: Mobile Problems


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

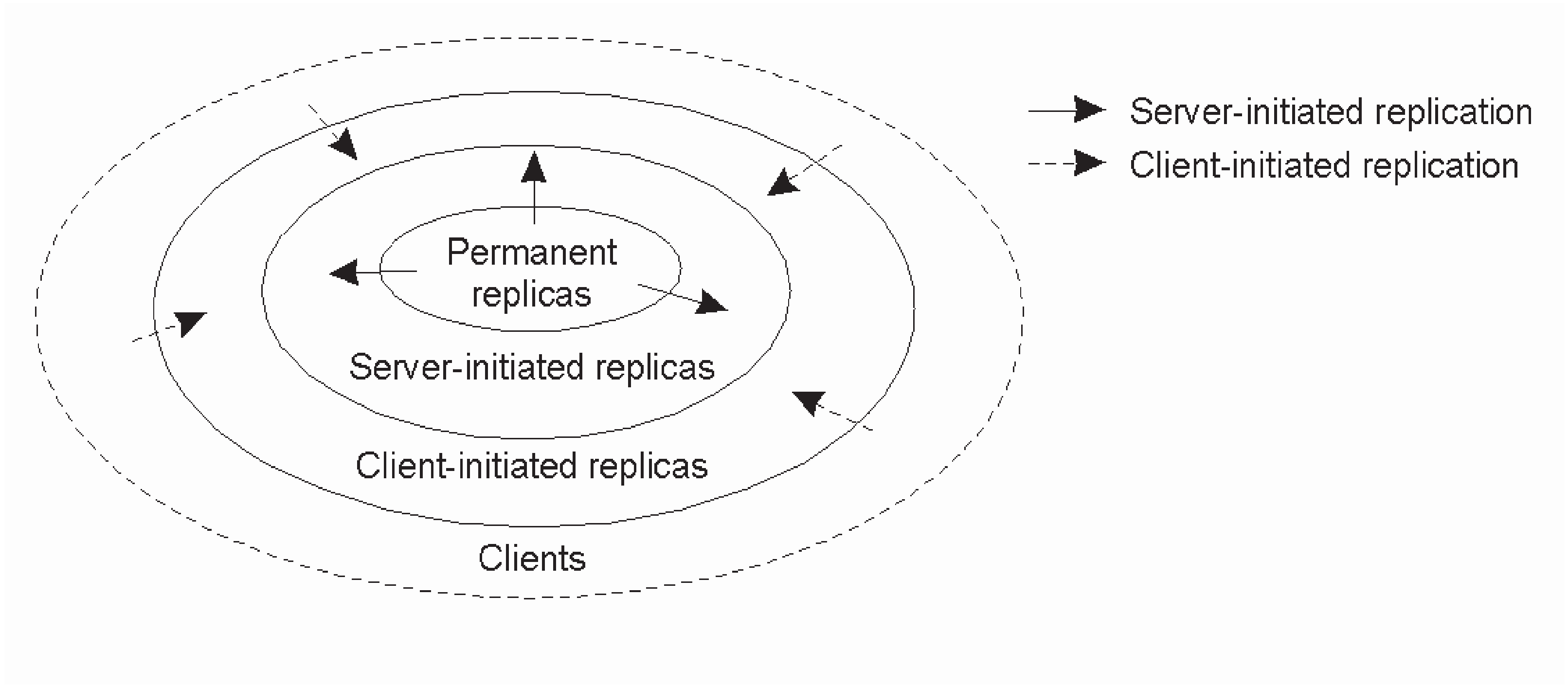
The principle of a mobile user accessing different replicas of a distributed database.

Replica Placement

- *We need to decide* ***where***, ***when*** and ***by whom*** copies of the data-store are to be placed

- Three types of copies
  - ***Permanent replicas***
  - ***Server-initiated replicas***
  - ***Client-initiated replicas***

Replica Placement


Server-initiated replication
Client-initiated replication
Permanent replicas
Server-initiated replicas
Client-initiated replicas
Clients

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

Permanent Replicas

- Number of permanent replicas is tend to be small
  - Often the initial set of replicas.

- Example: A web server can be organized by
  - Either COWs (Clusters of Workstations)
  - Or mirrored systems geographically distributed

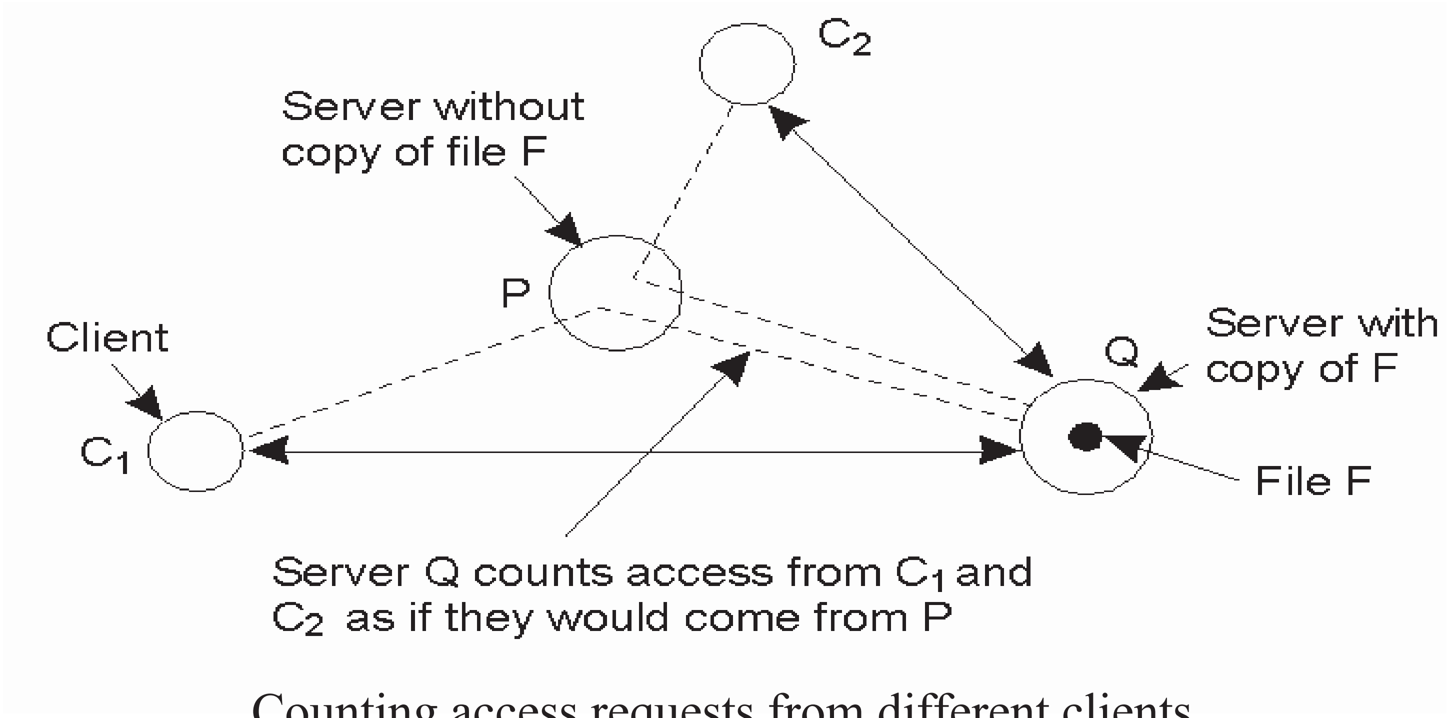
Server-Initiated Replicas

- Copies of data created by server to enhance performance.
- Also called *push caches*.
- Example: geographically create replicas close to where they are needed most.
- ***Dynamic replication*** is key for some services
- Note: principle of dynamic replication
  - Static collection of servers
  - But dynamic placement of replicated data on these servers close to demanding clients

Server-Initiated Replicas (Cont.)

- Problems: when and where?
- Keep track of all requests and where they're from.
  - Count requests from clients as if they're coming from nearest servers
- If request count drops below *del(S,F),* the deletion threshold for file, *F*, at server, *S*
  - Delete file but unless it is the last copy
- If request rises above *rep(S,F),* the replication threshold for file, *F*, at server, *S*
  - Create another replica with the one farthest away
- If request between *rep(S,F) and del(S,F)*
  - The file is allowed only to be migrated if necessary

Server-Initiated Replicas



Counting access requests from different clients.

C_1 and *C_2* share the same closest server (*P*). Then, all accesses for file *F* at *Q* from *C_1* and *C_2* are jointly counted as *$CNT_Q(P, F)$*

Client-Initiated Replicas

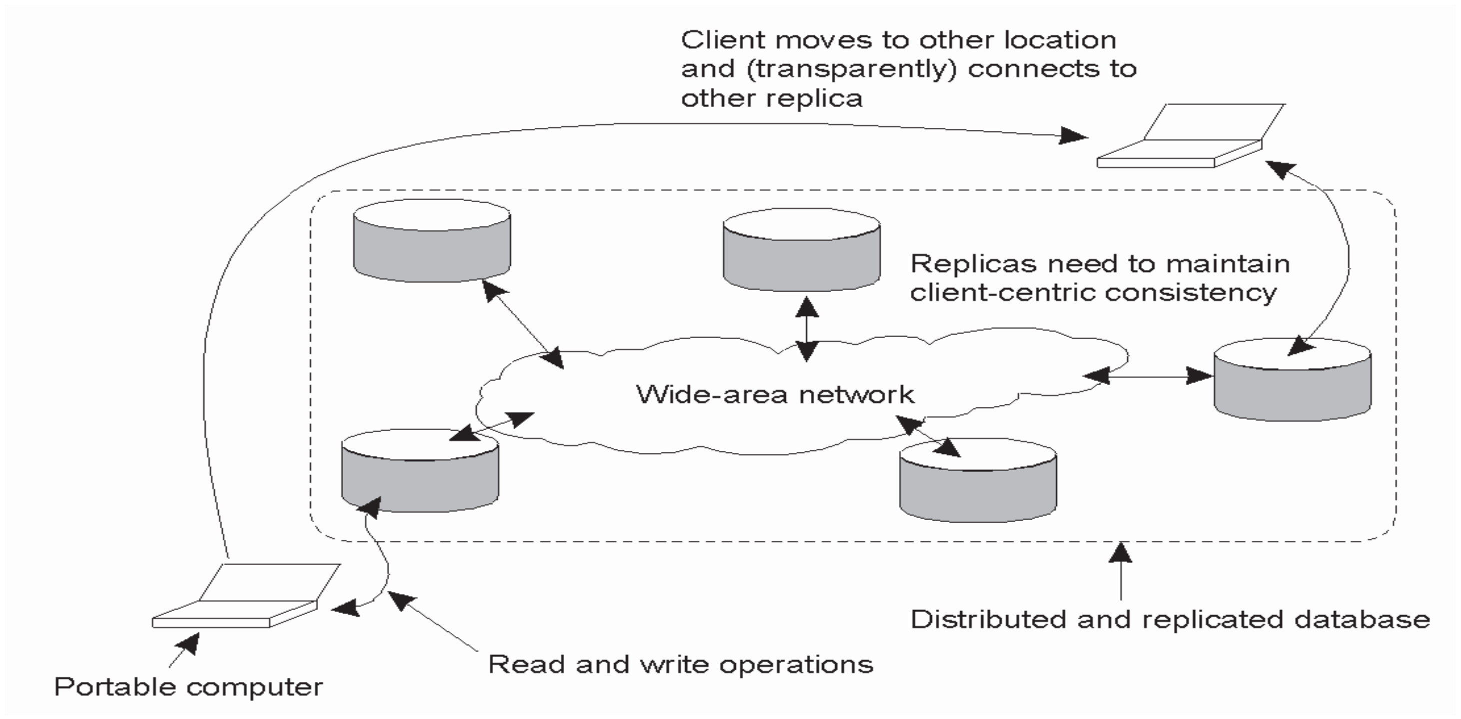
- Also known as *client caches*
- Created as a result of client requests – think of browser caches.
- Works well assuming, of course, that the cached data does not go *stale* too soon

Chapter 6.4.Distribution Protocols

- In this section
  - Discuss ways to propagate updates to replicas

- Replica Placement
- ***Update Propagation***
- Epidemic Protocols

Eventual Consistency: Mobile Problems


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Portable computer
Read and write operations

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

Update Propagation

- Design issues
  - State versus Operations
  - Pull versus Push Protocols
  - Unicasting versus Multicasting

State versus Operations

- Three possibilities to propagate
  - Propagate only a *notification* of an update (*invalidation protocols*)
    - Use little network bandwidth.
    - Good when #write >> #read
  - Transfer *updated data* from one copy to another
    - Use a lot of bandwidth
    - Good when #read >> #write
  - Propagate the *update operation* to other copies (*active replication*)
    - Uses less bandwidth (only operation)
    - But requires more processing power at replicas

Pull versus Push Protocols

- ***Push-based*** approach (***server-based*** protocols):
  - Updates are propagated to other replicas ***without*** asking.
  - Often between *permanent replicas*, *server-initiated replicas* and large shared caches by many clients.
    - Mainly perform read operations by clients
    - Good when #read >> #write
- ***Pull-based*** approach (***client-based*** protocols):
  - Often used by client caches, e.g., browser cache
    - Mainly perform well when only a few clients shared the data
    - Good when # read << #write
  - But result in a large response time when cache miss

Push vs. Pull Protocols: Trade Offs

Issue	Push-based	Pull-based
State on server.	List of client replicas and caches.	None.
Messages sent.	Update to each client (and possibly *fetch* modified update later if update are only *invalidations*).	Poll and update.
Response time at client.	Immediate (or *fetch-update time* if update are only *invalidations*).	Fetch-update time.

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

Hybrid Scheme

- These trade-offs have lead to a hybrid scheme:
  - A **lease** is a promise by the server that it will push updates to the client *for a specified time*.
  - After expiration
    - client reverts to a pull-based approach or request another lease
- How to determine the lifetime of a lease:
  - ***Age-based:*** depend on the last time the item was modified (web data)
    - For long time unmodified data, give it long-lasting lease
  - ***Renewal-frequency based:*** how often a specific client requests its cached copy
    - If often, give it long-lasting lease
  - ***State-space overhead at the server***
    - If server is becoming overloaded, lower the expiration time of leases

Unicasting versus Multicasting

- Unicast
  - Require sending update to *N* other servers.
  - Less efficient.
- Multicast
  - Sending a message efficiently to multiple receivers if multicasting facilities is available.

Chapter 6.4.Distribution Protocols

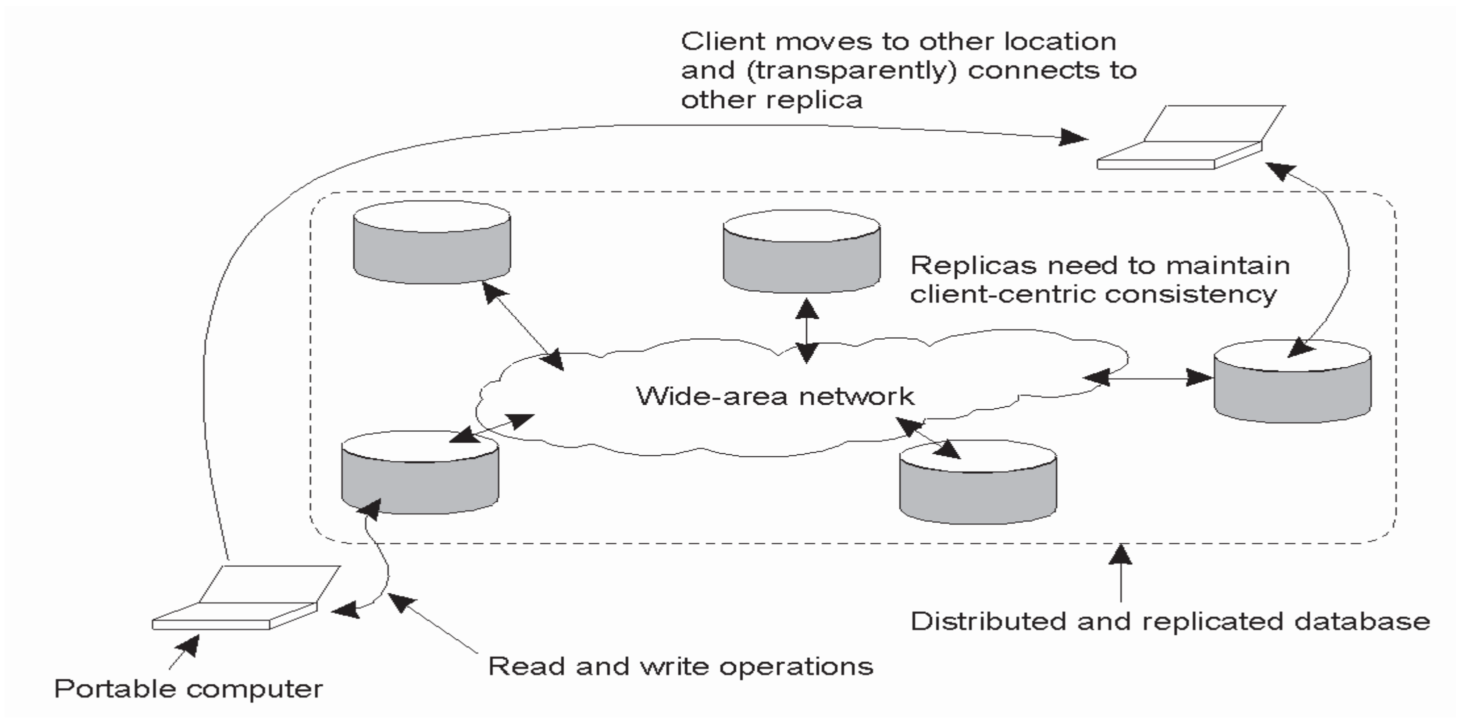
- In this section
  - Discuss ways to propagate updates to replicas

- Replica Placement
- Update Propagation
- ***Epidemic Protocols***

Epidemic Protocols

- For some data stores, provide only ***eventual consistency*** is sufficient
  - All replicas are eventually identical
- Update propagation in eventual-consistency is implemented by an ***epidemic protocols***
- Do not solve any update conflicts
  - Only concern propagating updates to all replicas in as few messages as possible.
  - Suitable for *eventual consistency* model
- **Assume** all updates for a specific data item are *initiated at a single server*.

Eventual Consistency: Mobile Problems


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

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

Epidemic Protocols: Terminology

- ***Infective server***
  - A server that holds an update that is willing to spread to other replicas.
- ***Susceptible server***
  - A server that has not yet been updated.
- ***Removed replica***
  - An updated server but is not willing or able to spread its update to any other replicas.

Update Propagation Models in Epidemic Protocols : Anti-Entropy

- ***Anti-entropy***: popular propagation model
- Server *P* picks a server *Q* at random, and exchanges updates with *Q*
- Three approaches:
  - *P* only pushes it own updates to *Q*
    - Generally good if few infected servers.
  - *P* only pulls in new updates from *Q*
    - Generally good if many infected servers
    - Spread updates is triggered by susceptible servers
  - *P* and *Q* send updates to each other
    - A push-pull approach

Update Propagation Models in Epidemic Protocols : Rumor-Spreading

- Variant of anti-entropy
- Also called ***gossiping***.
- If server *P* has just been updated
  - Randomly chooses *Q* and tries to push *the update*.
- If *Q* was already updated
  - *P* loses interest (becomes removed) with probability *1/k*, for some *k*.
- Gossiping works well
  - But can't prove that all updates eventually are received everywhere.

Epidemic Protocols For Removing Data

- Problem: spreading the deletion of a data item is hard
  - After removing *x*, an **old** copy of *x* may be considered as new data item
- Sol. keep a record of that deletion
  - i.e., spreading **death certificates**.
- However, have to recycle these death certificates
  - Otherwise, memory will be used up

Chapter 6.5: Consistency Protocols

- Consistency Protocols: describe a specific implementation of a consistency model
- ***Primary-Based Protocols***
  - Remote-Write Protocols
  - Local-Write Protocols
- Replicated-Write Protocols
- Cache-Coherence Protocols

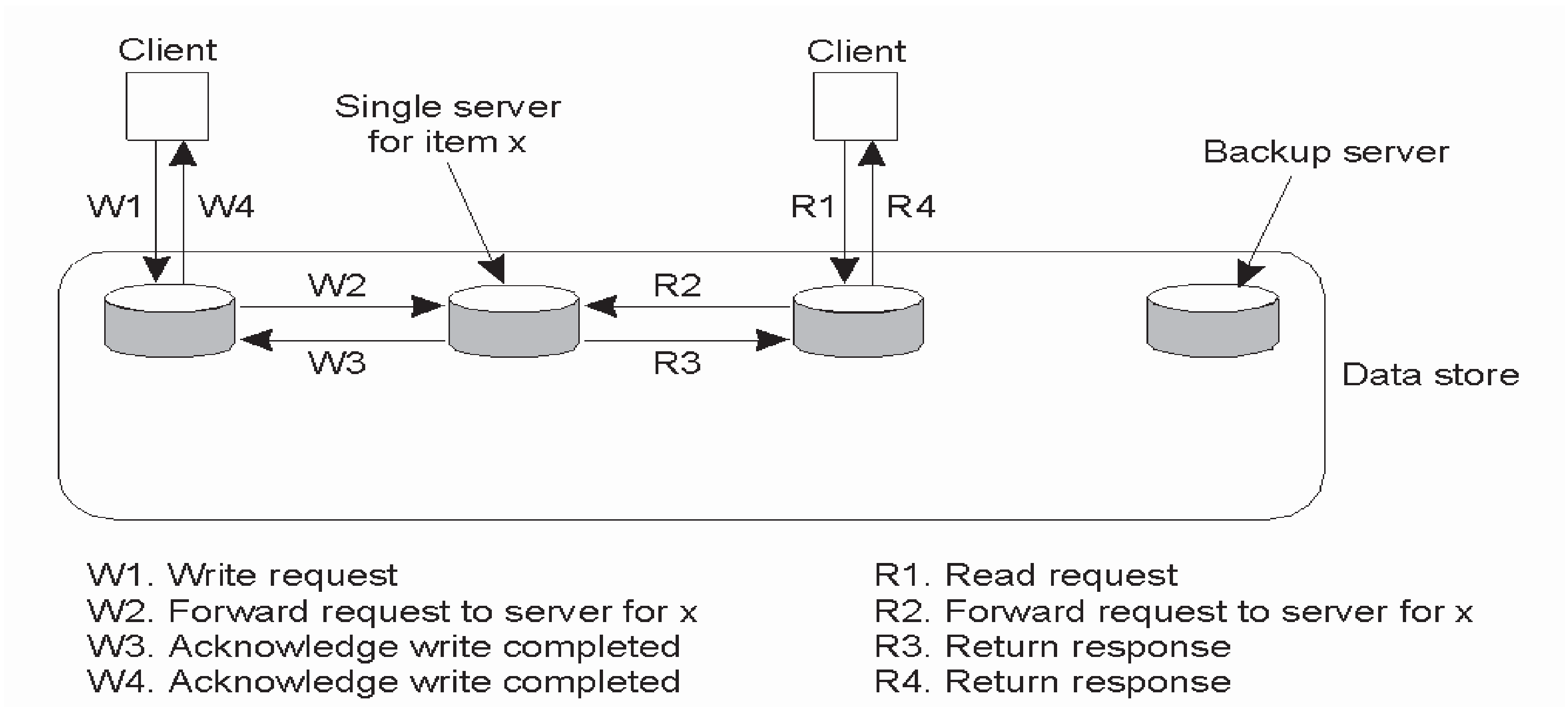
Primary-Based Protocols

- Each data item is associated with a primary.
  - The primary is responsible for coordinating writes to the data item.
- There are two types of Primary-Based Protocol,
  - Remote-Write Protocols
  - Local-Write Protocols

Remote-Write Protocols

- With this protocol, there are two variants
  - All reads/writes are performed at a single (remote) server.
    - This model is typically associated with traditional client/server systems
  - Writes are still centralized, but reads are now distributed.
    - The *primary* coordinates writes to each of the backups.
    - Called ***primary-backup protocols***

Remote-Write Protocols (1)

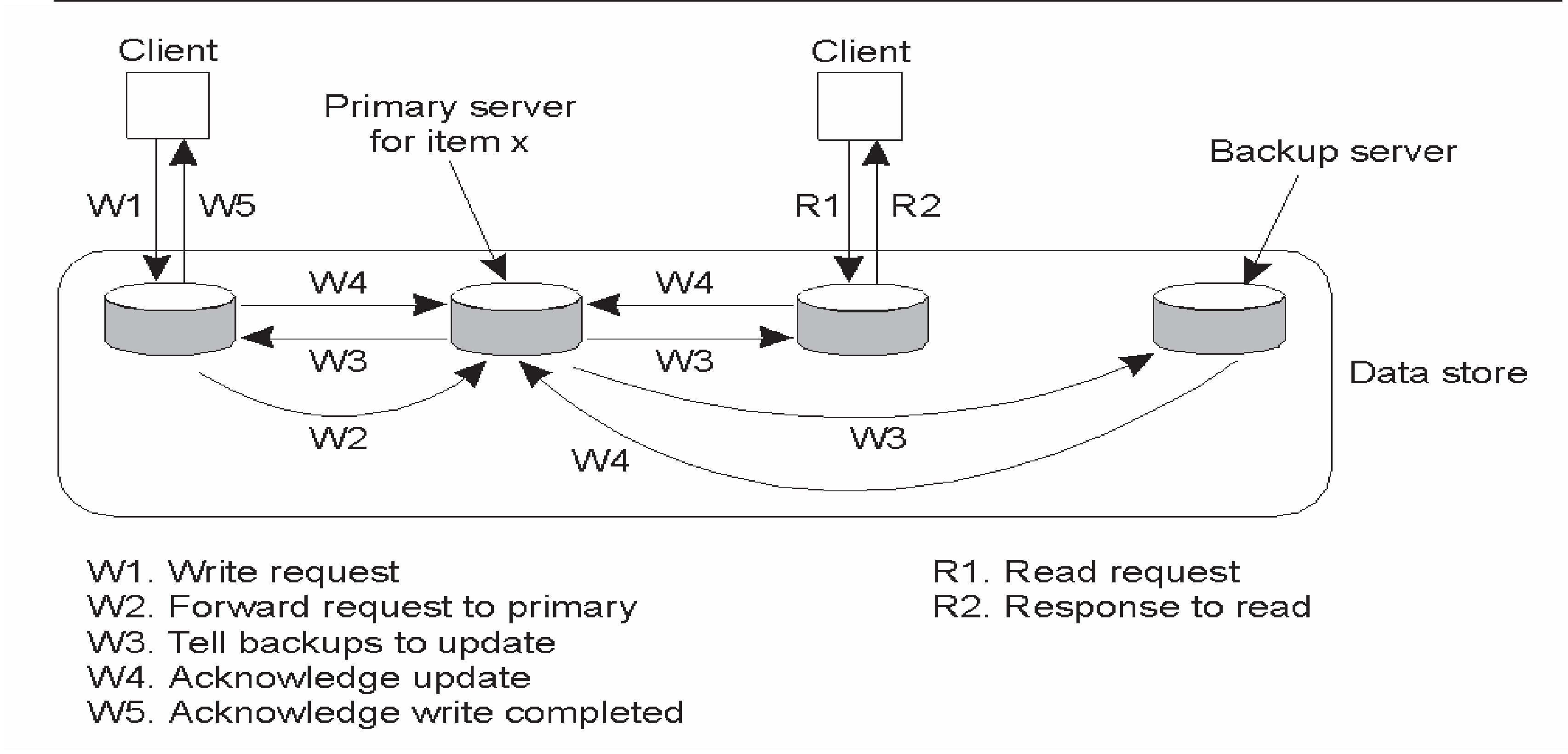


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

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

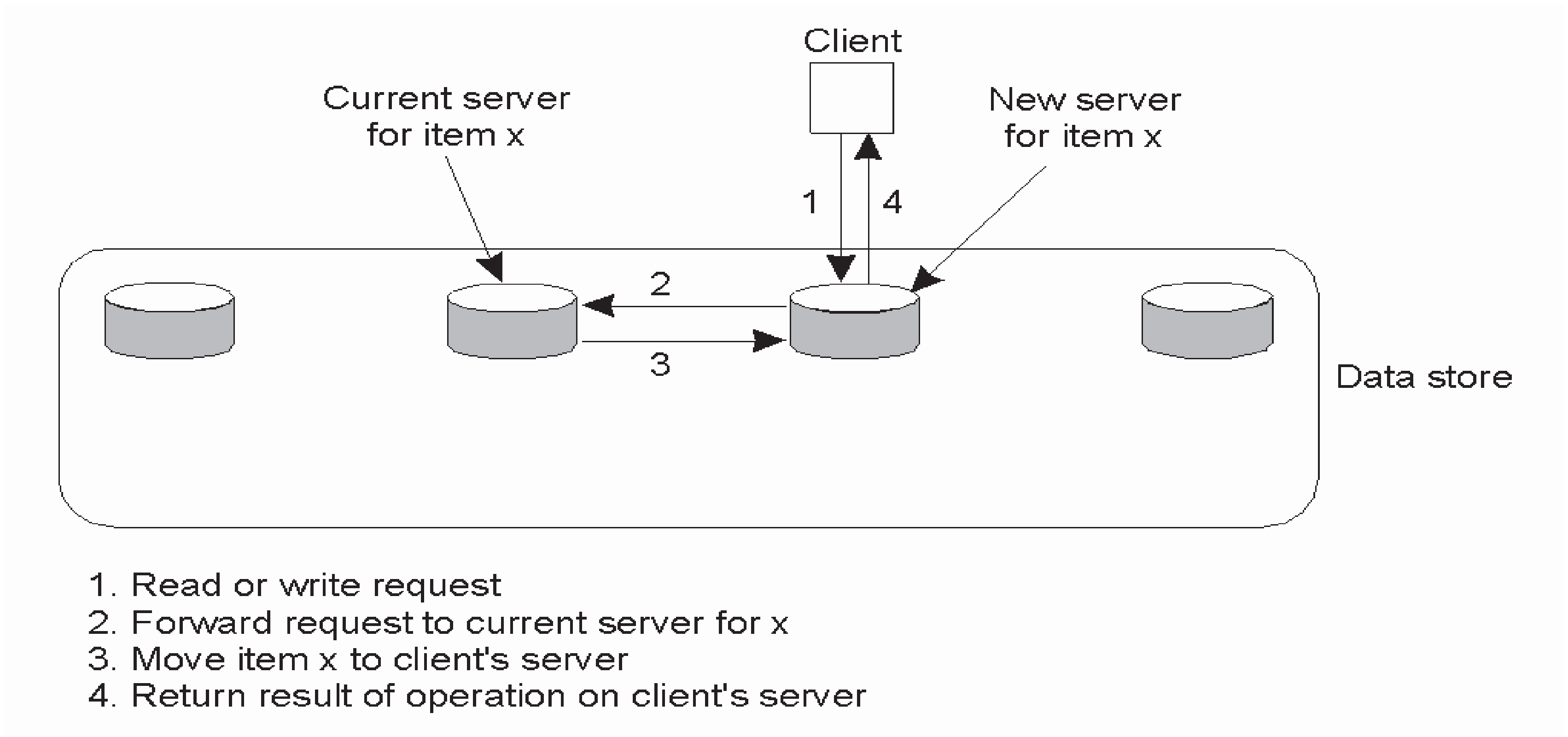
R1. Read request
R2. Response to read

The principle of primary-backup protocol.

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
- With this protocol, there are also two variants
  - All writes/reads are performed at a single (remote) server.
  - Writes are still centralized, but reads are now distributed

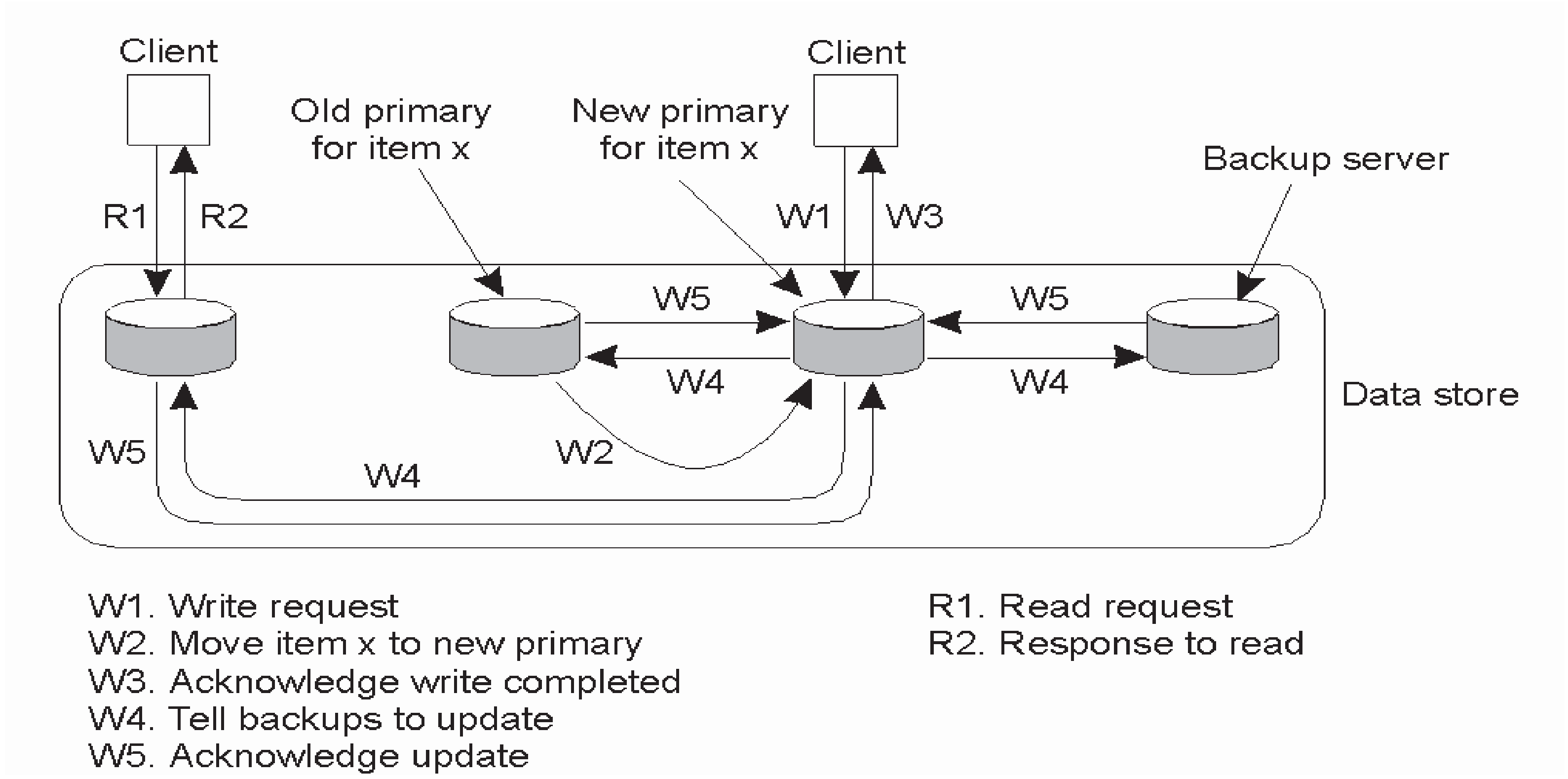
Local-Write Protocols (1)



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 (prior to the read/write).

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

Chapter 6.5: Consistency Protocols

- Consistency Protocols: describe a specific implementation of a consistency model

- Primary-Based Protocols
  - Remote-Write Protocols
  - Local-Write Protocols
- ***Replicated-Write Protocols***
- Cache-Coherence Protocols

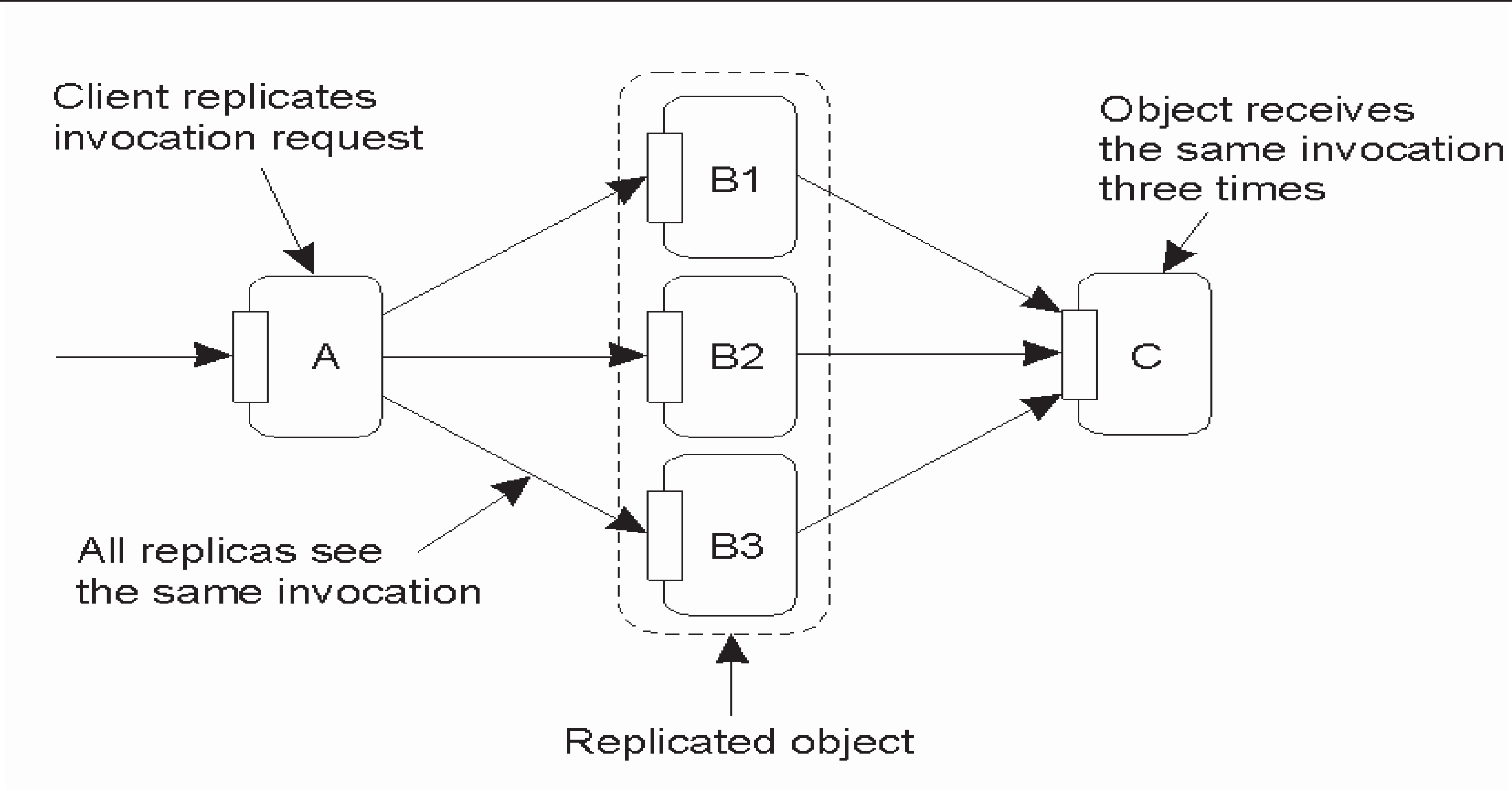
Replicated-Write Protocols

- With these protocols, writes can be carried out at *multiple* replica, instead of one in primary-based replicas
- Another name might be: "Distributed-Write Protocols"
- There are two types:
  - Active Replication.
    - An operation is forwarded to all replicas
  - Quorum-Based Protocol
    - By Majority Voting

Active Replication

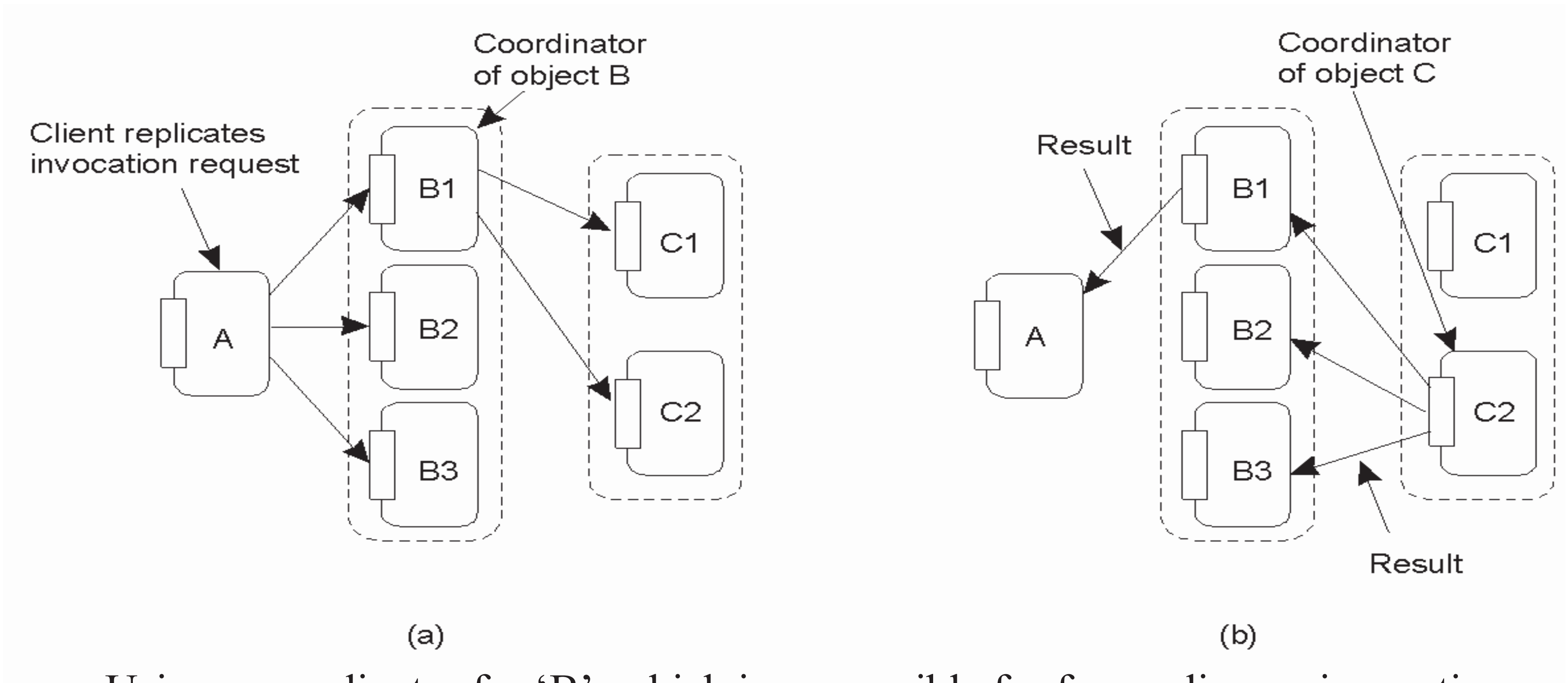
- A special process carries out the update operations at each replica
  - Updates are propagated to each replica
- Problem
  - Operation must be carried out in the same order everywhere
    - Sol 1. *totally-ordered multicast*
      - Lamport's timsestamps can be used to achieve total ordering, but this does not scale well
    - Sol 2. by a *central coordinator* called ***sequencer***
      - Assigns a unique ID# to each update, which is then propagated to all replicas, but still does not scale well
      - Also strongly resemble primary-based consistency protocol
  - *replicated invocations: next slide*

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'.

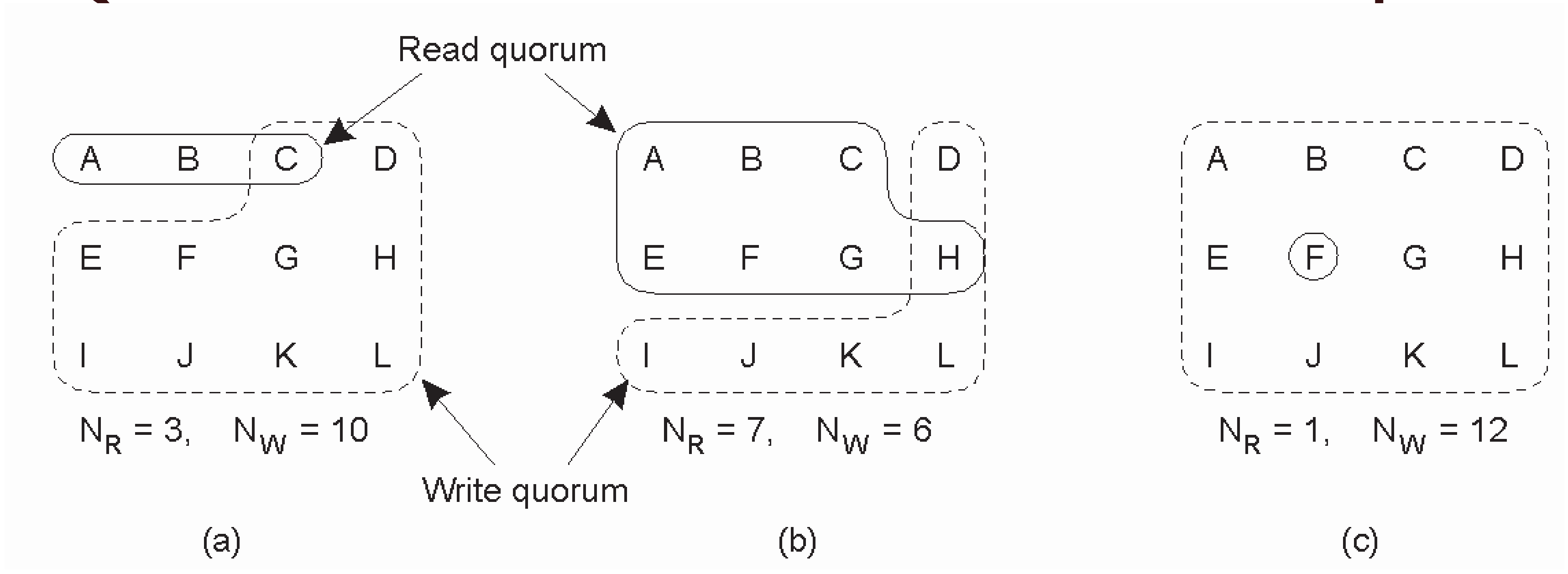
Quorum-Based Protocols

- Clients must acquire permissions from *multiple replicas* before reading/writing a replicated data item.
- Example.
  - To update a file, a process must get approval from a majority of the replicas ($> N/2$)
    - These replicas also need to agree *to perform the write*.
  - To read, a process also contacts a majority of the replicas ($> N/2$)

Quorum Protocols: Generalisation

- N_R: read quorum
- N_W: write quorum
- Must obey the following two rules
  - $N_R + N_W > N$: avoid read-write conflicts
  - $N_W > N/2$: avoid write-write conflicts
- See the next slide

Quorum-Based Protocols: Example



- A correct choice of read and write set

a) A choice that may lead to write-write conflicts

  a) If a write set is (D,H,I,J,K,L), another write set is (A,B,C,E,F,G) => conflict

b) A special choice, known as ROWA (read one, write all)

Chapter 6.5: Consistency Protocols

- Consistency Protocols: describe a specific implementation of a consistency model

- Primary-Based Protocols
  - Remote-Write Protocols
  - Local-Write Protocols
- Replicated-Write Protocols
- ***Cache-Coherence Protocols***

Cache-Coherence Protocols

- Ensure a client cache is consistent with the server-initiated replicas
- *Coherence Detection Strategy*
  - When inconsistencies are detected
- *Coherence Enforcement Strategy*
  - Determine how caches are kept consistent with the copies stored at servers

Coherence Detection Strategy

- When inconsistencies are actually detected ?
  - Statically at compile time:
    - Perform analysis prior to execution
    - If may lead to inconsistencies
      - Insert extra instructions to avoid inconsistency.
    - Used in conventional computer systems
  - Dynamically at runtime: check with the server
    - Applied in distributed system

Coherence Detection Strategy (Cont.)

- Runtime checking example: a distributed database
- There are three different approaches
  - Before start transaction, must verify first
    - Transaction cannot proceed until its consistency has been validated.
  - Let the transaction proceed while verification is taking place.
    - An optimistic approach
    - Assume cached data is up-to-date, but abort if assumption wrong
  - Verify the cached data only when the transaction committed
    - Also an optimistic approach

Coherence Enforcement Strategy

- How are caches kept consistent with copies in server ?
  - Server send invalidation to all caches when a data item is modified.
  - Server simply propagate the update.

What about Writes to the Cache

- *Read-only Cache*:
  - Updates are performed only by server (ie, pushed)
  - Client pulled the data whenever the client notices that the cache is *stale*.
- *Write-Through Cache*:
  - Client modifies the cache, then sends the updates to the server.
- *Write-Back Cache*:
  - Delay the propagation of updates
  - Allow multiple updates to be made locally and then sends the most recent to the server