# Distributed Systems
# 分布式系统

Consistency and Replication

一致性和复制

# Replication of data

Why?

- To enhance reliability
- To improve performance in a large scale system

Replicas must be consistent

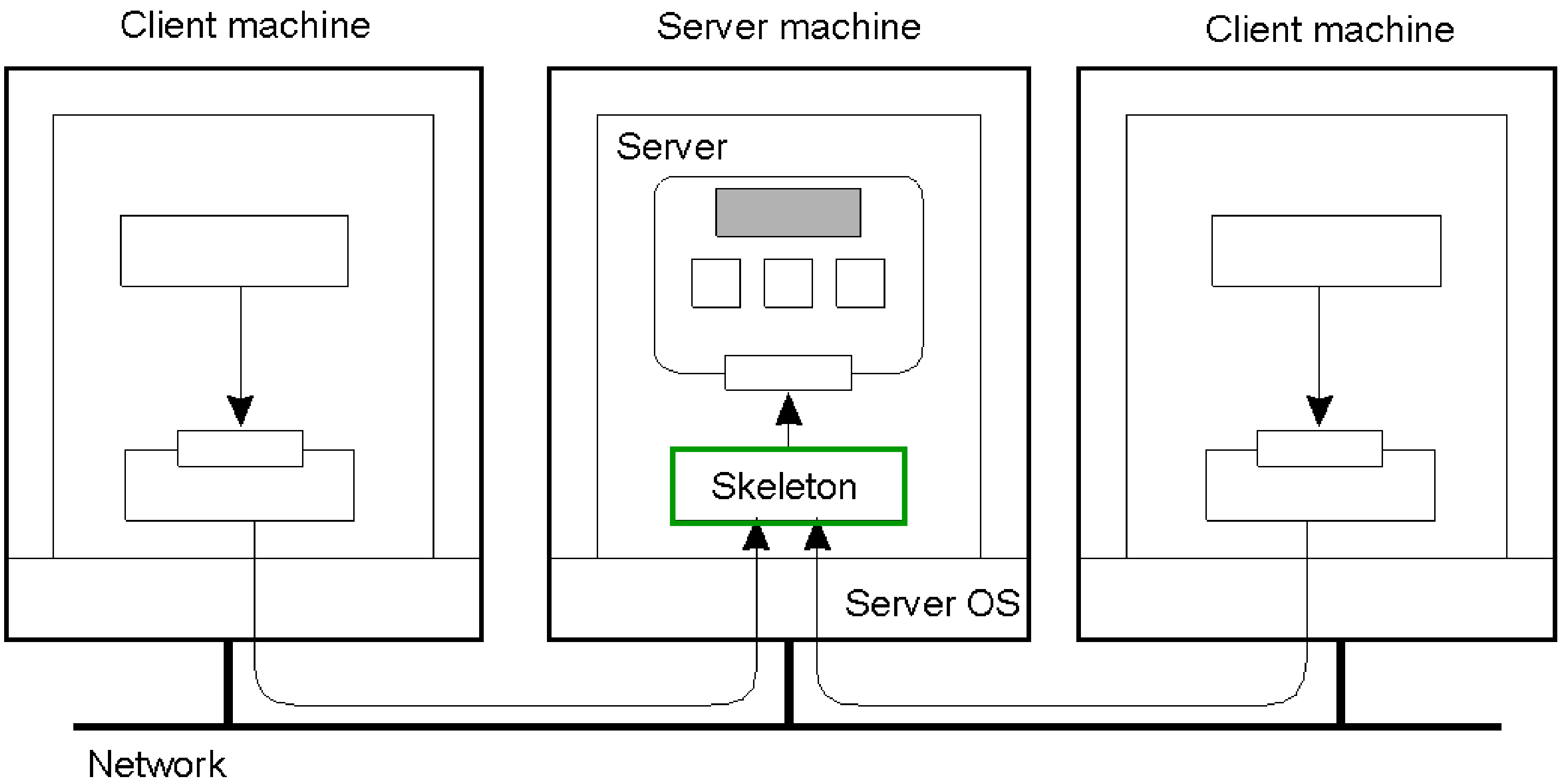
- Modifications have to be carried out on all copies
- Problems with network performance
- It is needed to handling concurrency…

Different consistency models

A consistency model is a set of rules that process obeys accessing data

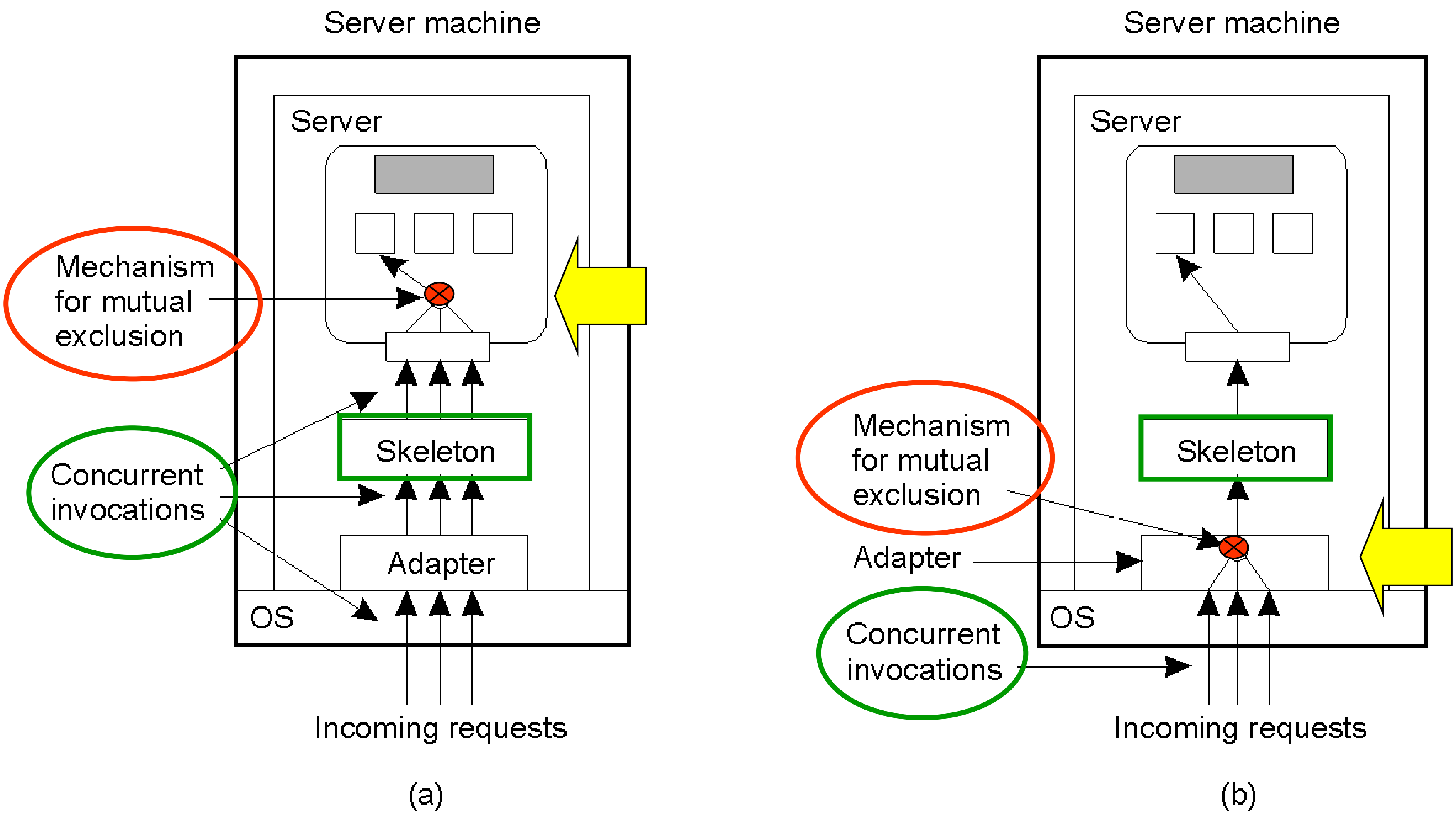
# Object Replication

How can we protect objects against multiple clients access? Synchronization...




A distributed remote object shared by different clients.

# Object Replication

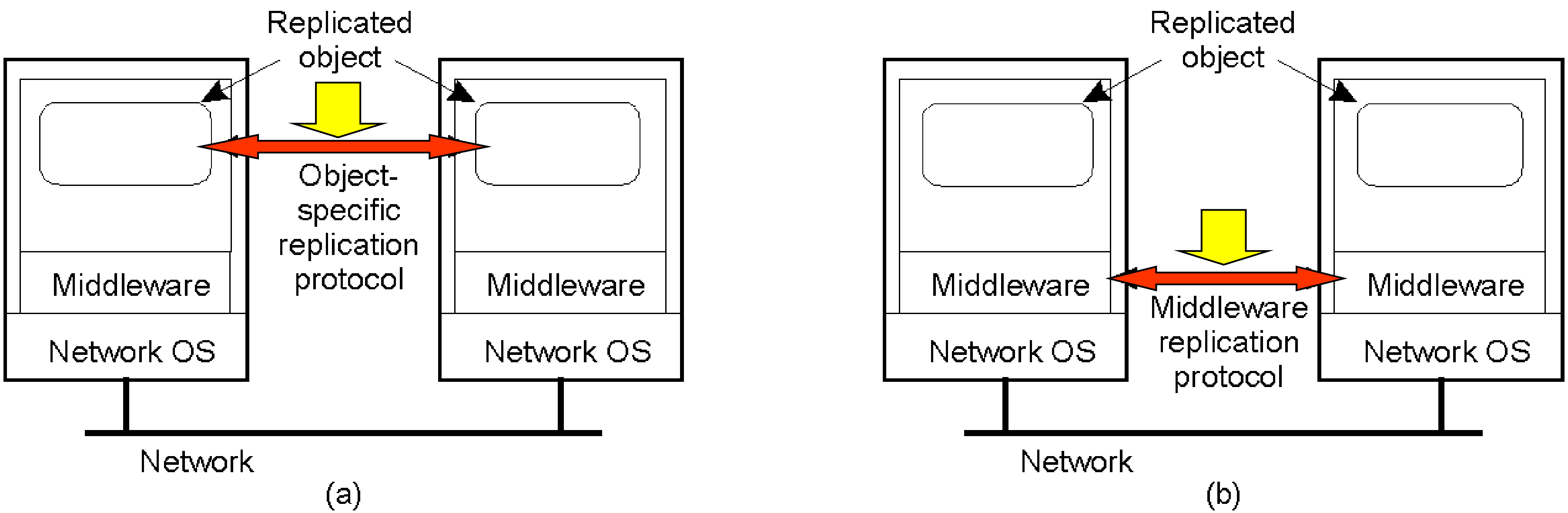



a) A remote object capable of handling concurrent invocations on its own.

b) A remote object for which an object adapter is required to handle concurrent invocations

# Object Replication

Replicas need more synchronization to ensure that concurrent invocations lead to consistent results




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

b) A distributed system responsible for replica management (simpler for application developers), it ensures that concurrent invocation are passed to the replicas in the correct order

# Replication and Scaling

Replication and caching are widely used in scaling technique, but

Keeping replicas up to date needs networks use

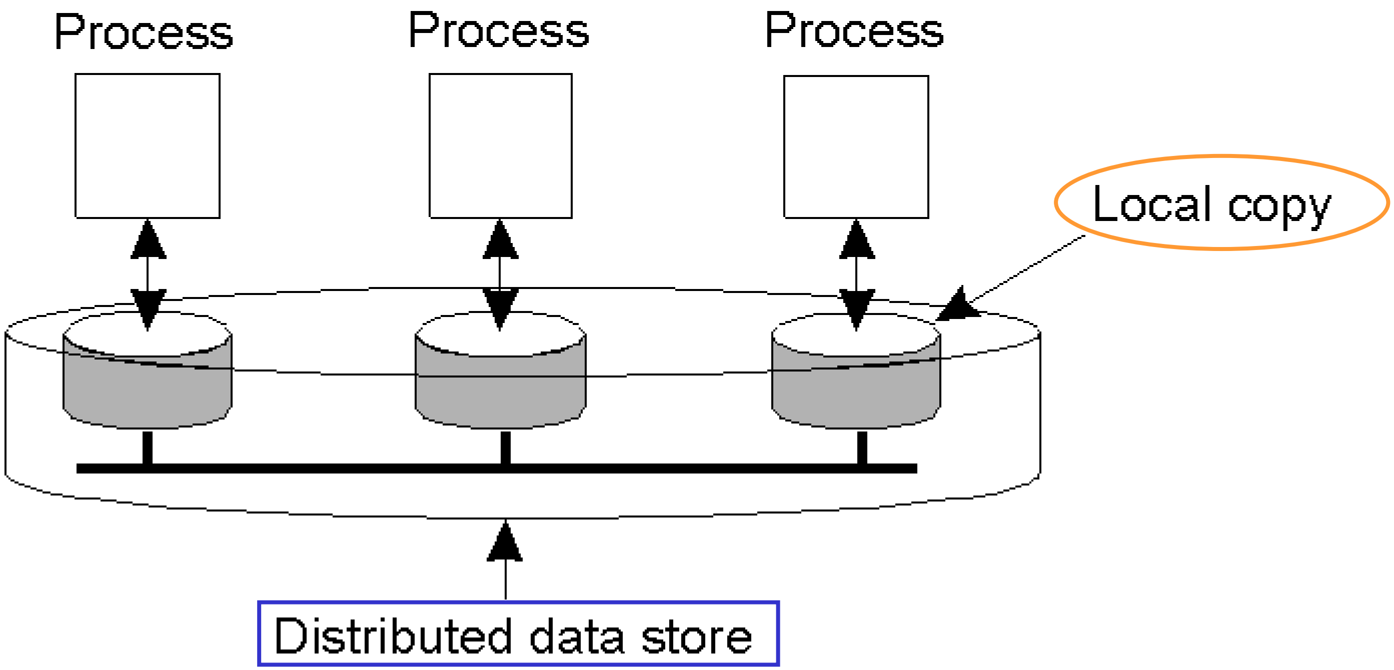
Update needs to be atomic ( → transaction)

Replicas need to be synchronized (time consuming)

Loose Consistency

In this case copies are not always the same everywhere.

# Data-Centric Consistency Models




The general organization of a logical **data store**, physically *distributed* and *replicated* across multiple machines.

Each process that can access data has its own local copy

Write operations are propagated to the other copies

# Strict Consistency

***Any read on a data item x returns a value corresponding to the result of the most recent write on x***

two operations in the same time interval are said to conflict if they operate on the same data and one of them is a write operation


(a) (b)

Behavior of two processes, operating on the same data item.

a) A strictly consistent store.

b) A store that is not strictly consistent.

Strict consistency is the ideal model but it is impossible to implement in a distributed system

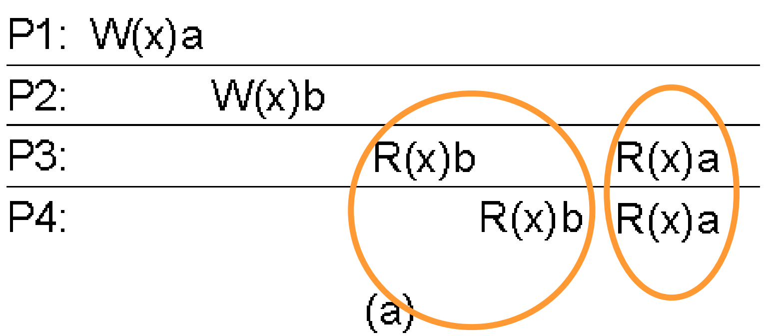
It is based on *absolute global time*.

# Linearizability and Sequential Consistency (1)

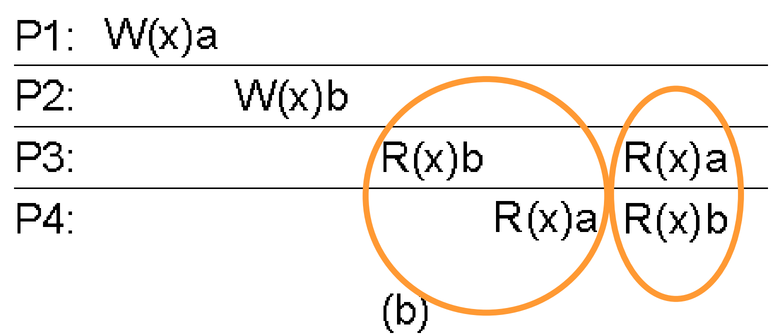
**Sequential Consistency** it is a weaker consistency model than strict consistency

***The result of any execution is the same as if the read and write operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program***

All processes see the same interleaving of operations



(a)



(b)

a) A sequentially consistent data store.
b) A data store that is not sequentially consistent.

No reference to the timing of the operations

# Linearizability and Sequential Consistency (2)

**Linearizability** is weaker than strict consistency but stronger than sequential consistency

***The result of any execution is the same as if the read and write operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program. In addition, if $ts_{OP1}(x) < ts_{OP2}(y)$, then operation OP1(x) should precede OP2(y) in this sequence***

Operations receive a timestamp using a global clock, but with finite precision

| Process P1 | Process P2 | Process P3 | |
|---|---|---|---|
| x = 1; | y = 1; | z = 1; | write |
| print ( y, z); | print (x, z); | print (x, y); | read |

Example : three concurrently executing processes; (x, y, z) are data store items

Various (90) interleaved execution sequences are possible

# Linearizability and Sequential Consistency (3)

| | | | |
|---|---|---|---|
| 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 |
| Signature: 001011 | Signature: 101011 | Signature: 110101 | Signature: 111111 |
| (a) | (b) | (c) | (d) |

Not all signature pattern are allowed : 000000 not permitted, 001001 not permitted

Constraints:

- Program order must be maintained
- Data coherence must be respected

Data coherence : any read must return the most recently written value of the data (relatively to the single data item, without regard to other data)

# Causal Consistency (1)

When there is a read *followed* by a write, the two events are *potentially* causally related

Operation not causally related are said concurrent

Necessary condition:

***Writes that are potentially causally related must be seen by <u>all processes</u> in the same order. Concurrent writes may be seen in a different order on different machines.***

# Causal Consistency (2)

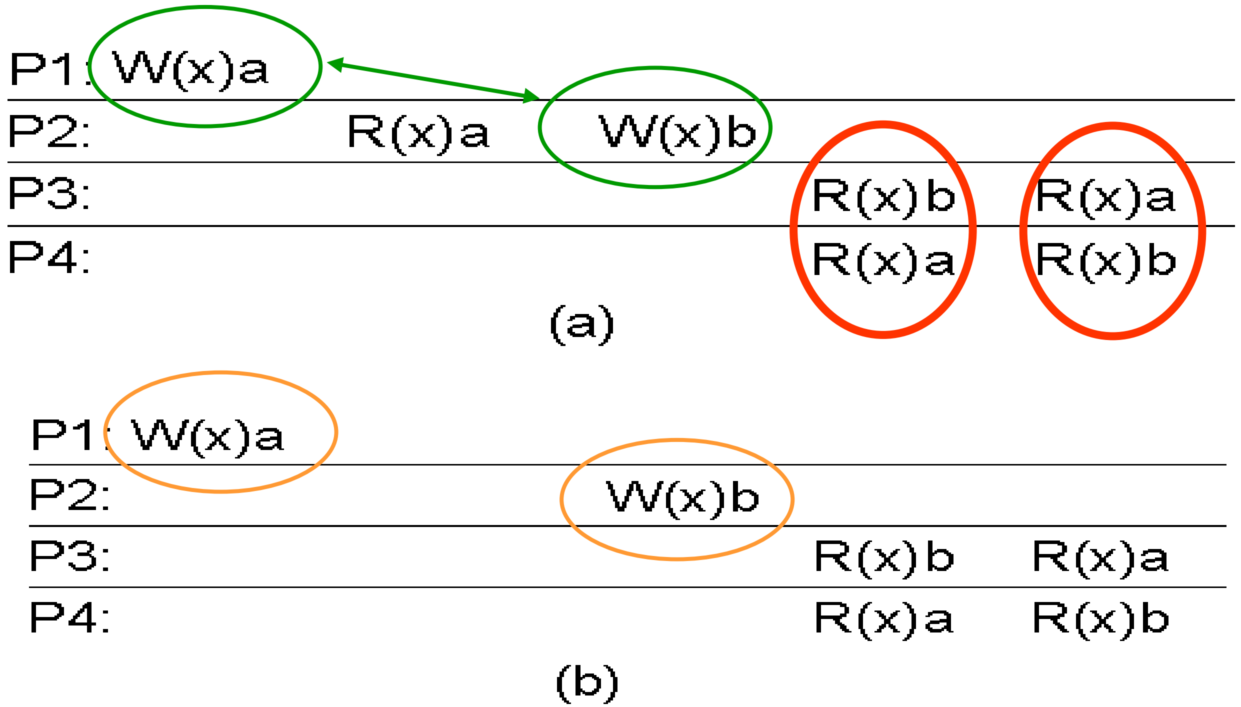
| | | | | | | |
|---|---|---|---|---|---|---|
| P1: | W(x)a | | | W(x)c | | |
| P2: | | R(x)a | W(x)b | | | |
| P3: | | R(x)a | | | R(x)c | R(x)b |
| P4: | | R(x)a | | | R(x)b | R(x)c |

This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

Note that the writes $W_2(x)b$ and $W_1(x)c$ are concurrent

Causal consistency requires keeping tracks of which processes have seen which writes

# Causal Consistency (3)




a) A violation of a casually-consistent store. $W_2(x)b$ may be related to $W_1(x)a$

b) A correct sequence of events in a casually-consistent store. $W_1(x)a$ and $W_2(x)b$ are concurrent

# FIFO Consistency (1)

Relaxing consistency requirements we drop causality

Necessary Condition:

***Writes done by a <u>single</u> process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.***

All writes generated by different processes are considered concurrent

It is easy to implement

# FIFO Consistency (2)

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 events of FIFO consistency. It is not valid for causal consistency

# FIFO Consistency (3)

| Process P1 | Process P2 | Process P3 | |
|---|---|---|---|
| x = 1;<br>print ( y, z); | y = 1;<br>print (x, z); | z = 1;<br>print (x, y); | write<br>read |

```
x = 1;
print (y, z);
y = 1;
print(x, z);
z = 1;
print (x, y);
```

Prints: 00

(a)

```
x = 1;
y = 1;
print(x, z);
print ( y, z);
z = 1;
print (x, y);
```

Prints: 10

(b)

```
y = 1;
print (x, z);
z = 1;
print (x, y);
x = 1;
print (y, z);
```

Prints: 01

(c)

The statements in bold are the ones that generate the output shown. Their concatenated output is 001001, that is incompatible with sequential consistency

# FIFO Consistency (4)

Different processes can see the operations in different order

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

The result of this two concurrent processes can be also that both processes are killed.

# Weak Consistency

We can release the requirements of writes within the same process seen in order everywhere introducing a synchronization variable.

A synchronization operation synchronize all local copies of the data store.

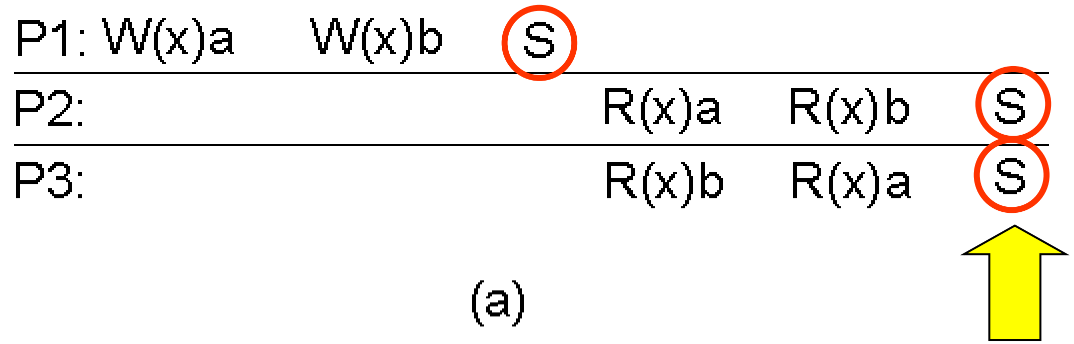
Properties of weak consistency:

- ***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.***

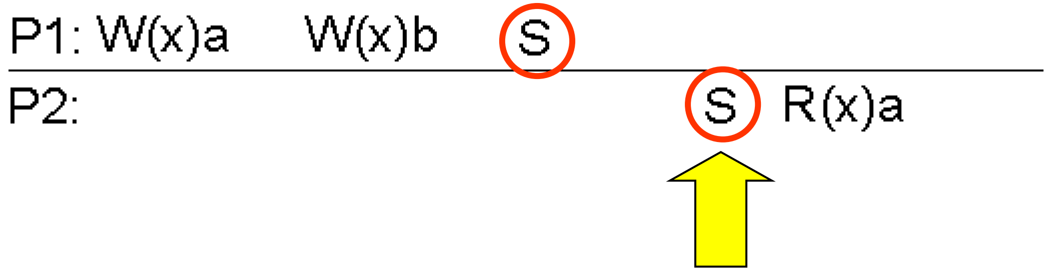
It forces consistency on a **group** of operations, not on individual write and read

It limits the time when consistency holds, not the form of consistency.

# Weak Consistency




(a)

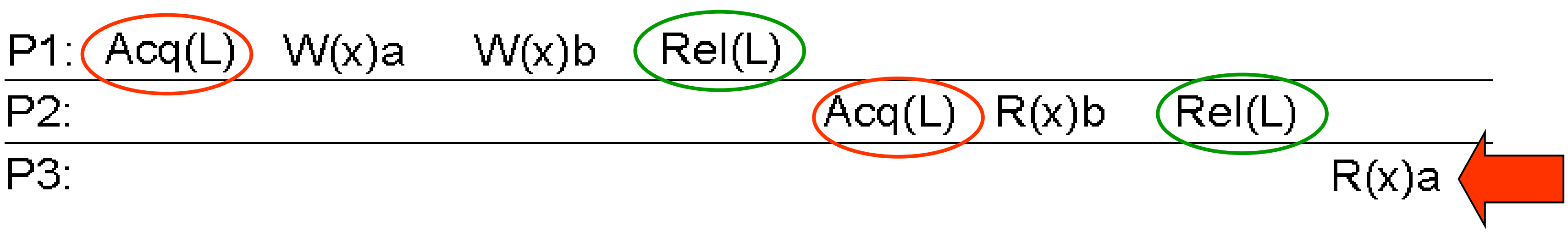



(b)

a) A valid sequence of events for weak consistency.
b) An invalid sequence for weak consistency.

# Release Consistency

If it is possible to know the difference between entering a critical region or leaving it, a more efficient implementation might be possible. To do that, two kinds of synchronization variables are needed. Release consistency : acquire operation to tell that a critical region is being entered; release operation when a critical region is to be exited




A valid event sequence for release consistency.

Shared data kept consistent are called protected

# Release Consistency

Rules:

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

Explicit acquire and release calls are required

# Entry Consistency

Many synchronization variables associated with each shared data

Conditions:

- ***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 (1)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 | |

A valid event sequence for entry consistency. Lock are associated with each data item

# Summary of Consistency Models

| Consistency | Description |
|---|---|
| Strict | Absolute time ordering of all shared accesses matters. |
| Linearizability | All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp |
| Sequential | All processes see all shared accesses in the same order. Accesses are not ordered in time |
| Causal | All processes see causally-related shared accesses in the same order. |
| FIFO | All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order |

(a)

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

(b)

a) Consistency models not using synchronization operations.

b) Models with synchronization operations.

# Client Centric Consistency

In many cases concurrency appears only in restricted form.

In many applications most processes only read data

Some degrees of inconsistency can be tolerate

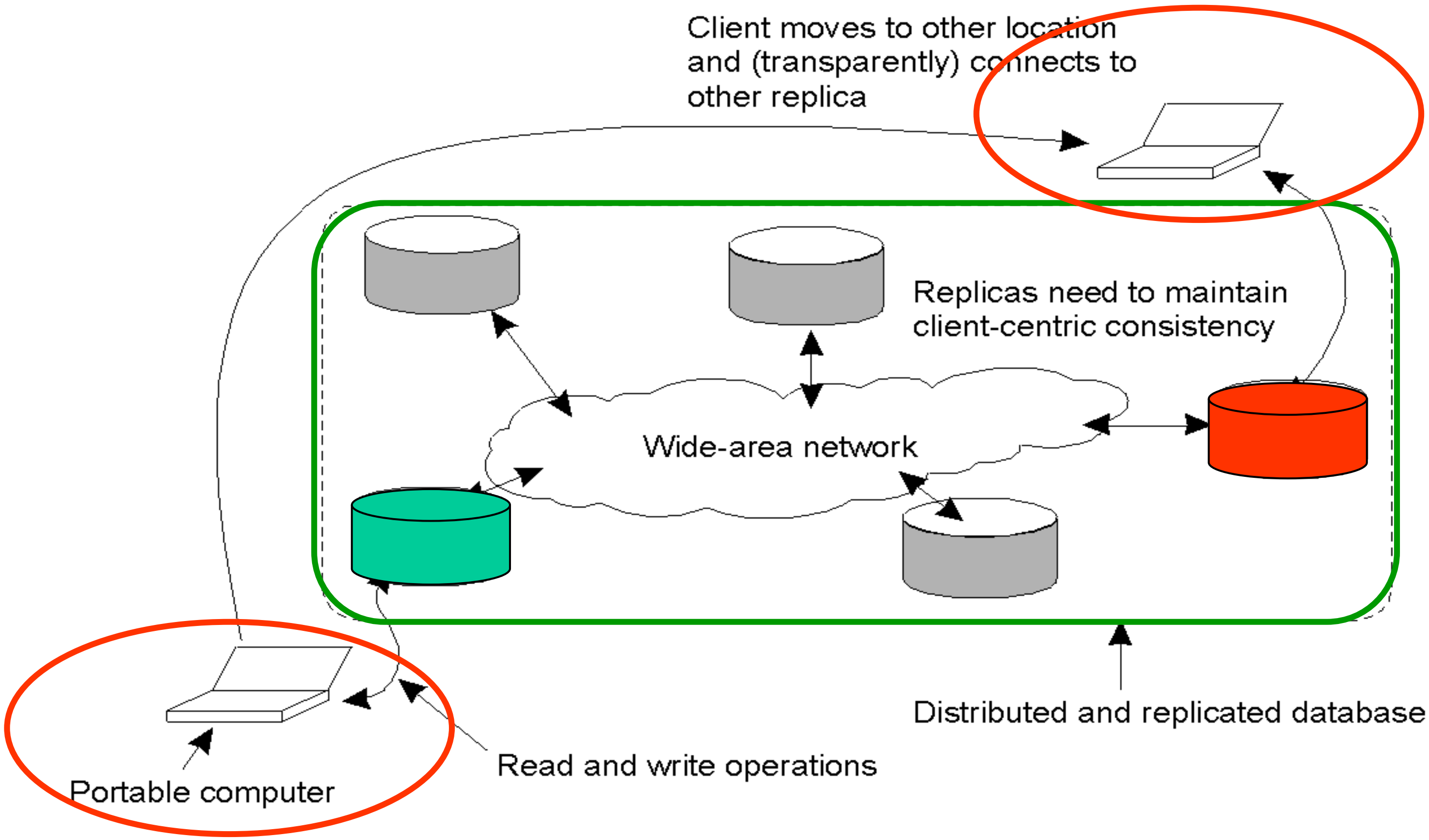
In some cases if for a long time no update takes place all replicas gradually become consistent

Eventual consistency

# Eventual Consistency

eventual consistency for replicated data is fine if clients always access the same replica

**Client centric** consistency provides consistency guarantees for a single client with respect to the data stored by that client




A mobile user accessing different replicas of a distributed database has problems with eventual consistency.

# Client centric models

Clients access distributed data store using, generally, the local copy. Updates are eventually propagated to other copies.

- **Monotonic read**

  *If a process reads the value of a data item x, any successive read operation on x by <u>that process</u> will always return that same value or a more recent value*

- **Monotonic write**

  *A write operation by a process on a data item x is completed before any successive write operation on x by the <u>same process</u>*

- **Read your writes**

  *The effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the <u>same process</u>*

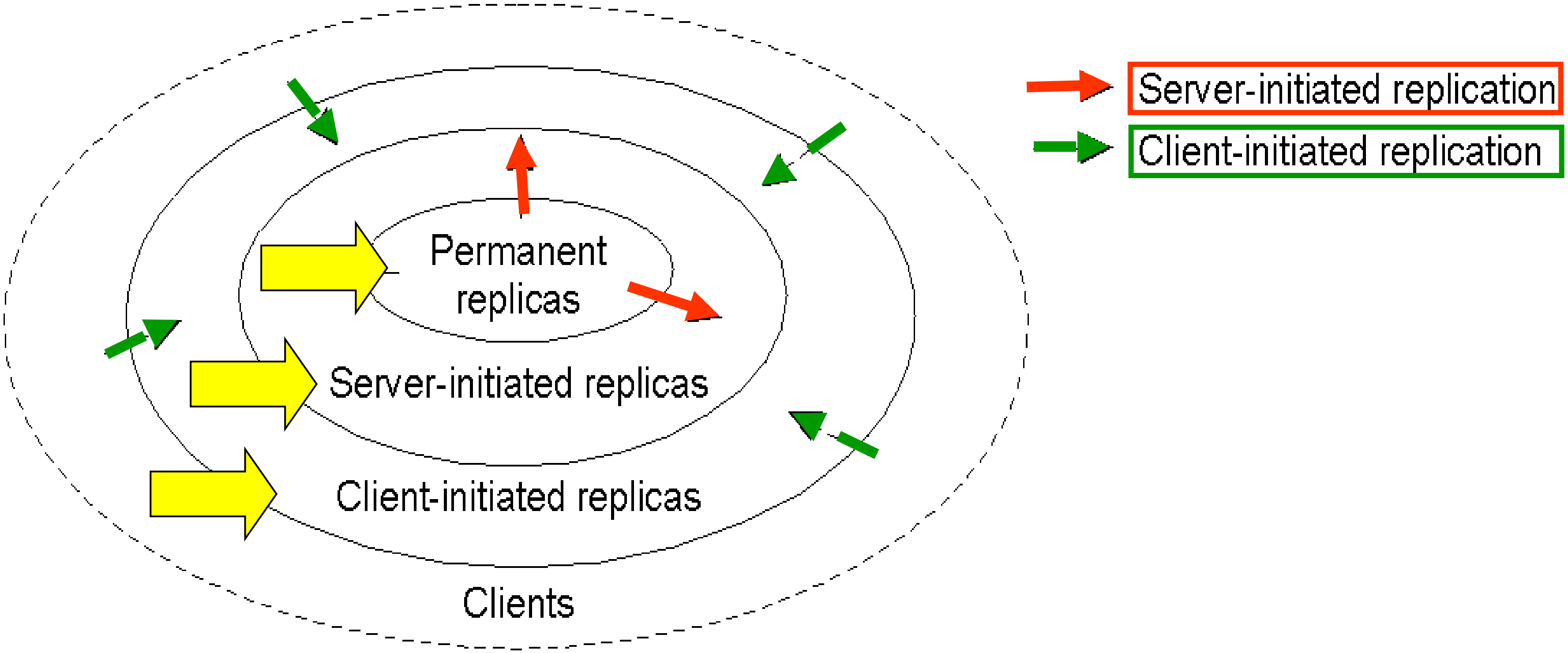
- **Writes follow reads**

  *A write operation by a process on a data item x following a previous read operation on x by the <u>same process</u>, is guaranteed to take place on the same or more recent values of x that was read*

# Distribution Protocols

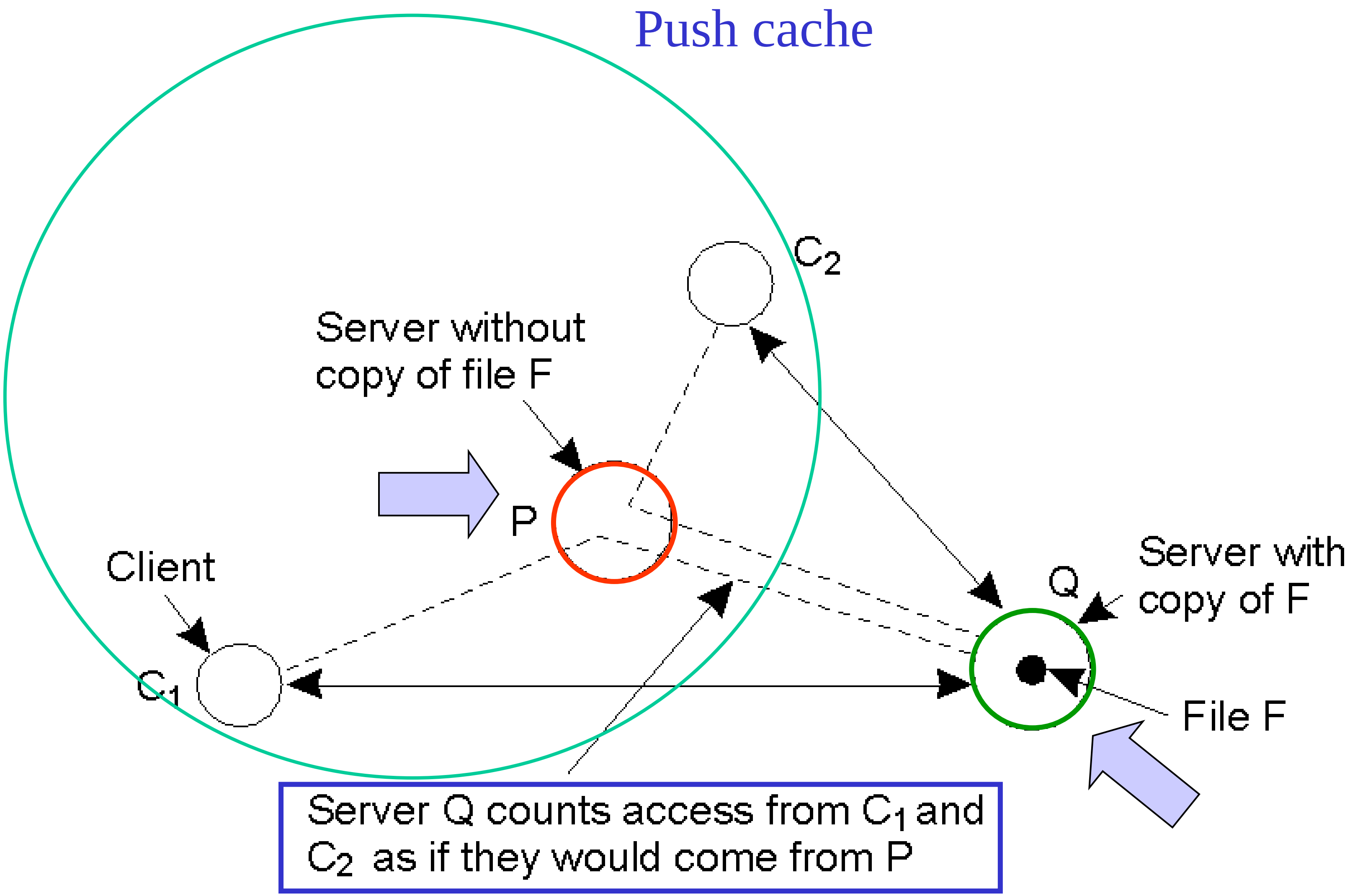
## Replica Placement

Where, when, by whom copies of data are to be placed?




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

# Server-Initiated Replicas




Web case. Counting access requests from different clients.

# Update propagation

What is to be propagated?

- Propagate only a notification of an update (Invalidation protocols) - R/W ratio: low

- Transfer data from one copy to another - R/W ratio: high

- Propagate the update operation to other copies (Active replication)

# Pull versus Push Protocols

**How is it to be propagated?**

- Push (or server) based protocols

  update are propagated to other replicas without request when a high consistency degree is needed

  I.e. Permanent to server initiated replicas

- Pull (or client) based protocols

  update are propagated to other replicas on request

I.e. Web cache

# Pull versus Push Protocols

| Issue | Push-based | Pull-based |
|---|---|---|
| State at server | List of client replicas and caches | None |
| Messages sent | Update (and possibly fetch update later for invalidation protocols) | Poll and update |
| Response time at client | Immediate (or fetch-update time) | Fetch-update time |

Hybrid propagation : lease (in a lease servers push updates with expiration time)

What kind of communication can be used?

- Unicast ( pull based approach)
- Multicast (push based approach)

# Epidemic propagation

It is used with eventual consistency and the main goal is to propagate updates with a few messages.
The infective server holds an update, the susceptible server will to be updated

## Anti-entropy model:

Three approaches:
- P pushes its updates to Q
- Q pulls new updates from P
- P and Q send their updates each other

## Gossiping

# Consistency Protocols

**Primary-based protocols**

Each data item x in the data store has an associated primary, which is responsible for coordinating write operations on x

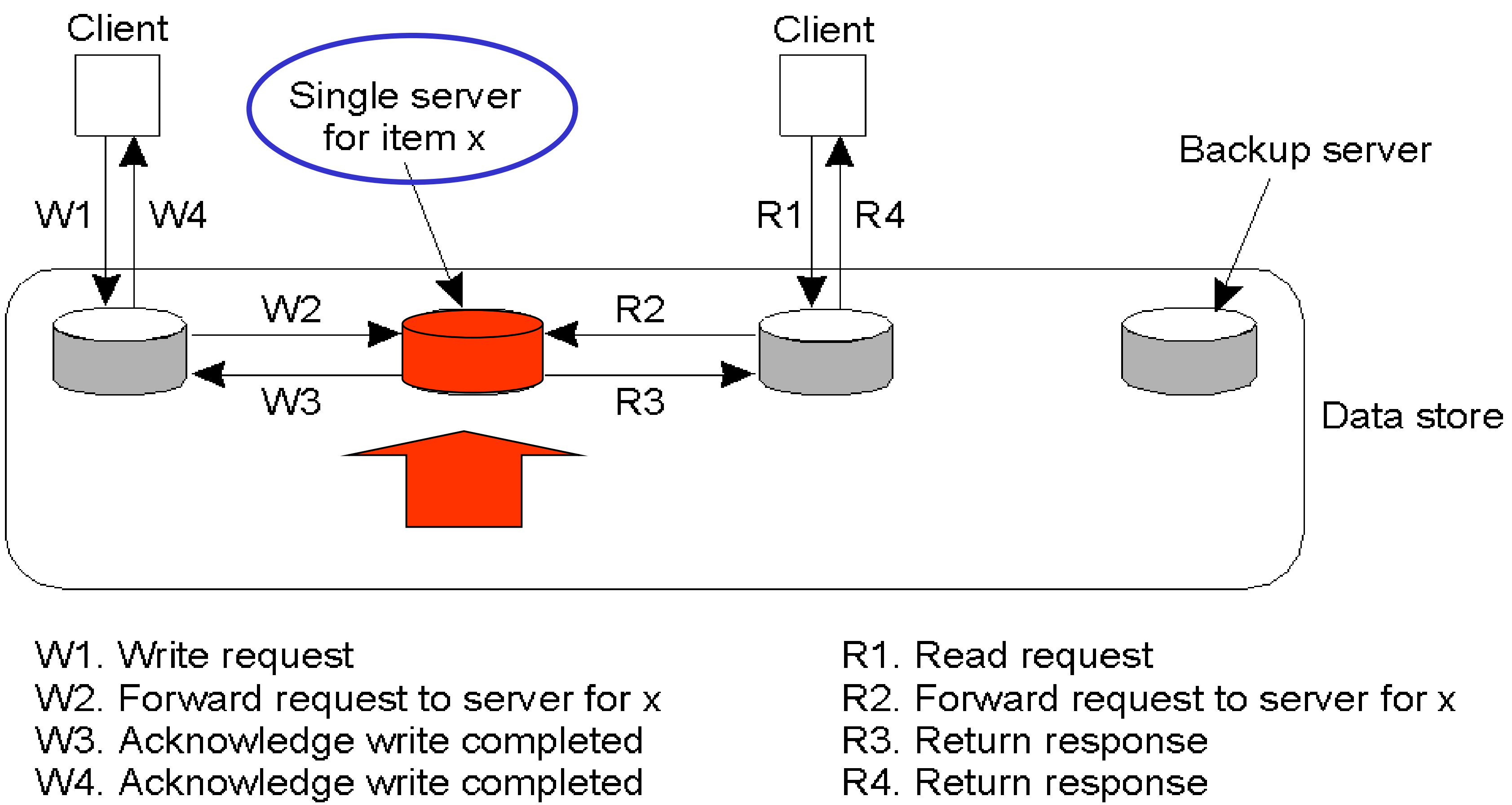
**Replicated write protocols**

Write operations can be carried out at multiple replicas instead of only one

**Cache-coherence protocols**

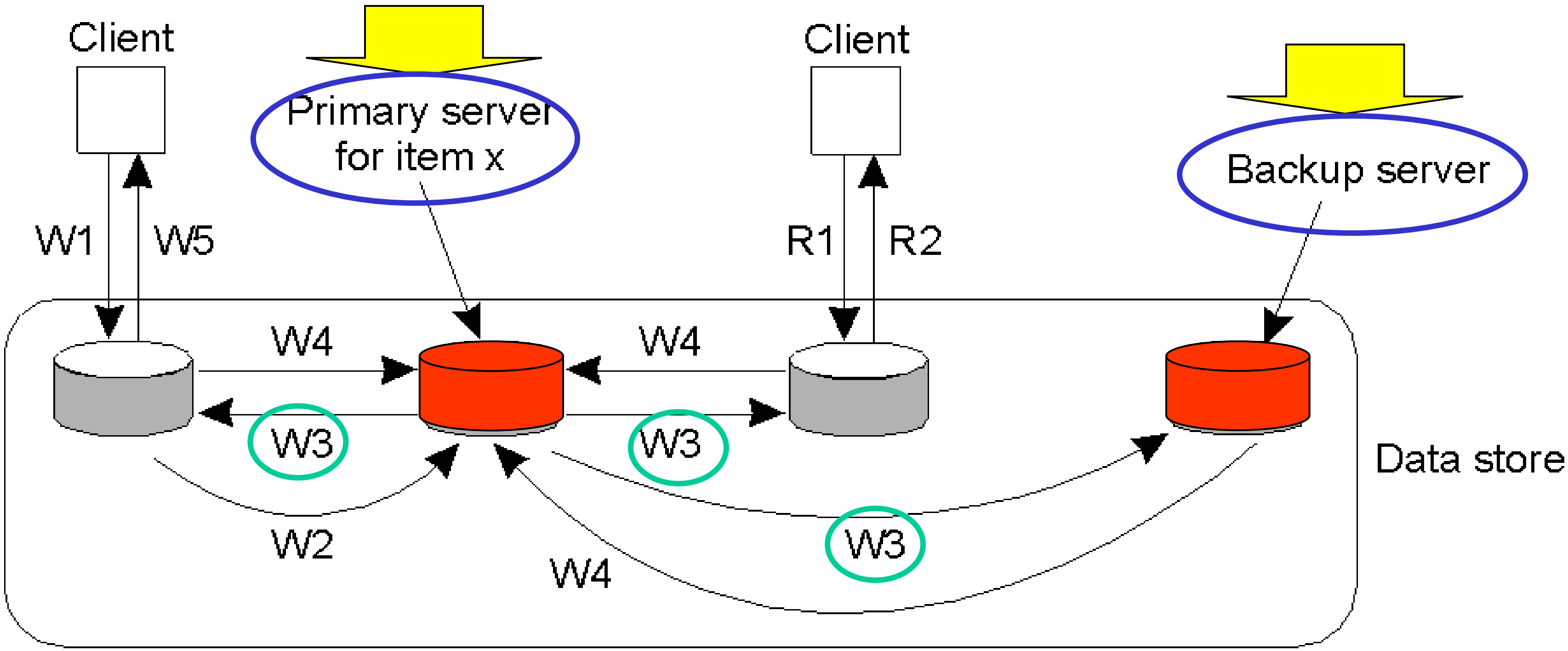
Controlled by clients instead of servers

# Remote-Write Protocols




Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded. Data can be distributed, but they are not replicated (really simple!).

# Remote-Write Protocols

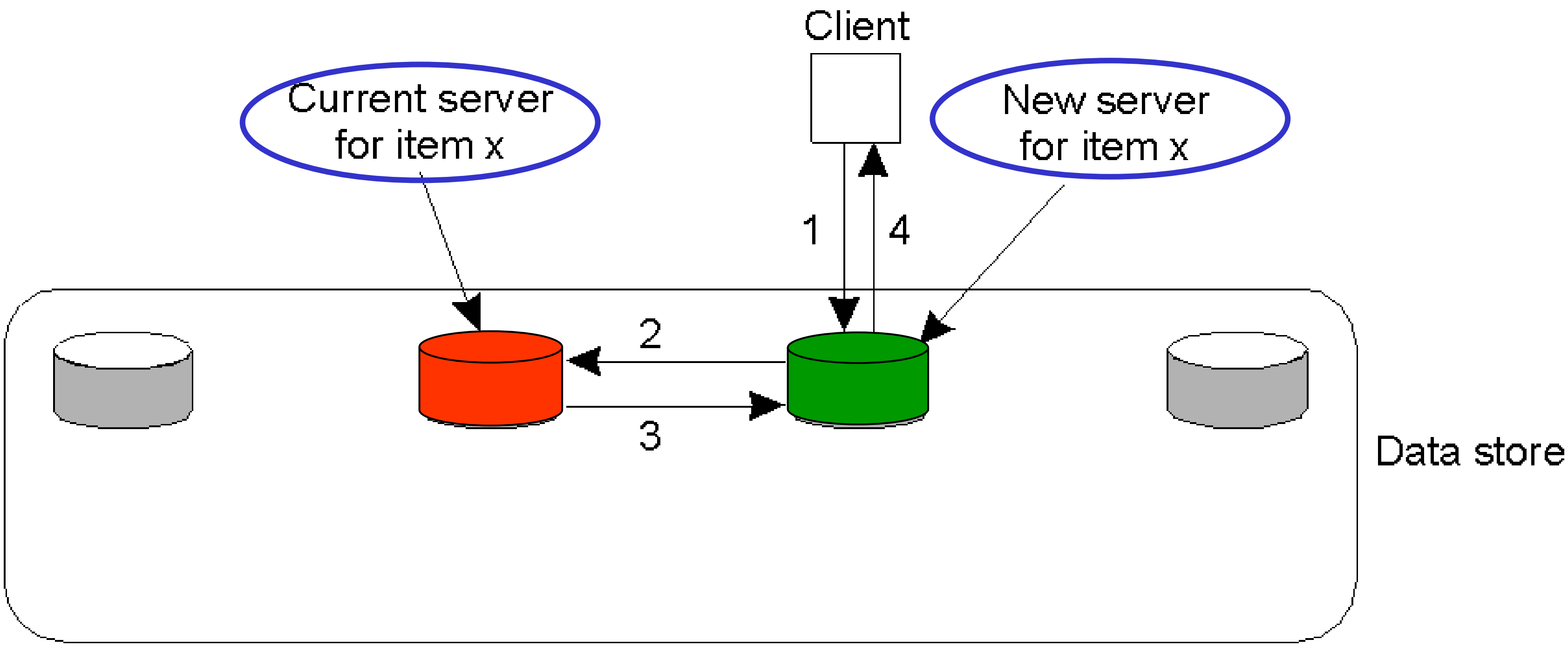



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

The principle of primary-backup protocol (time consuming). It implements sequential consistency if done as a blocking operation.

# Local-Write Protocols

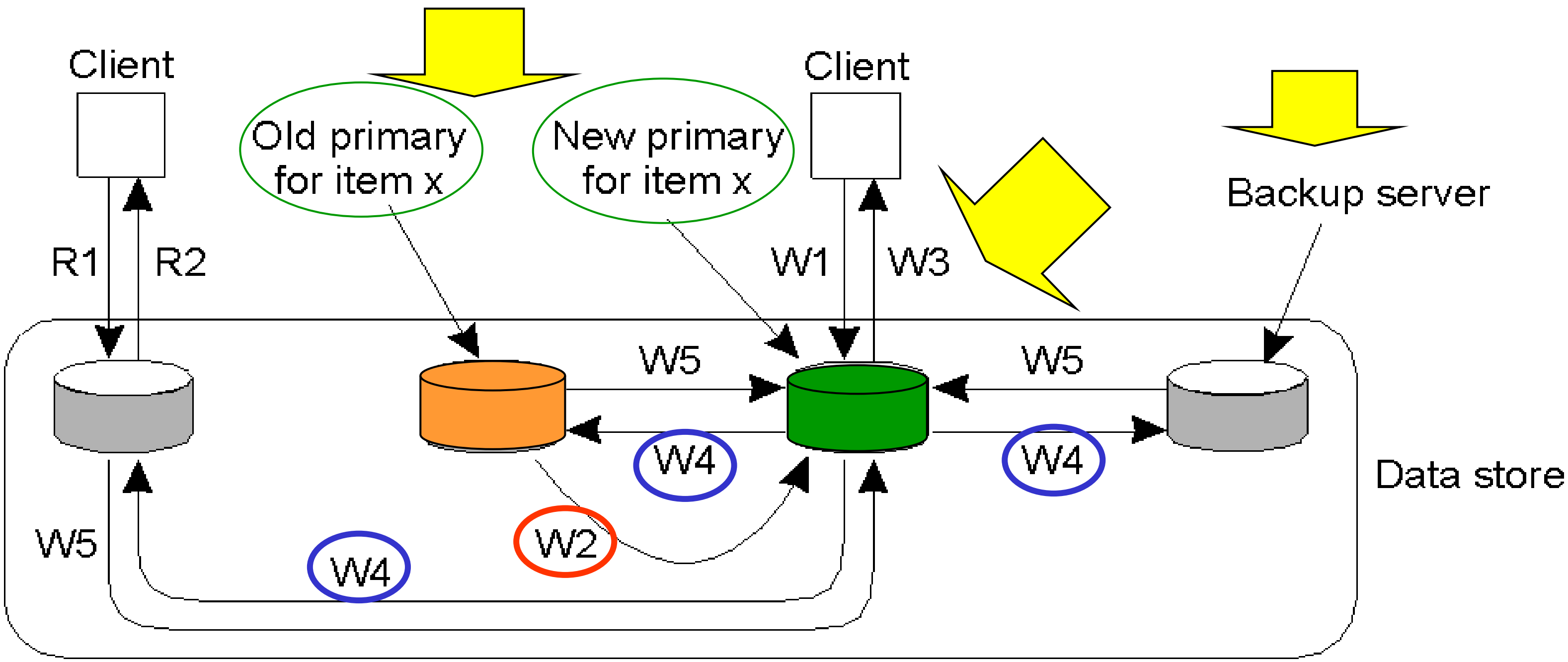



1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

Primary-based local-write protocol in which a single copy is migrated between processes (fully distributed non-replicated version of the data store).

Location information is the main problem in a widely distributed data store.

# Local-Write Protocols




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.

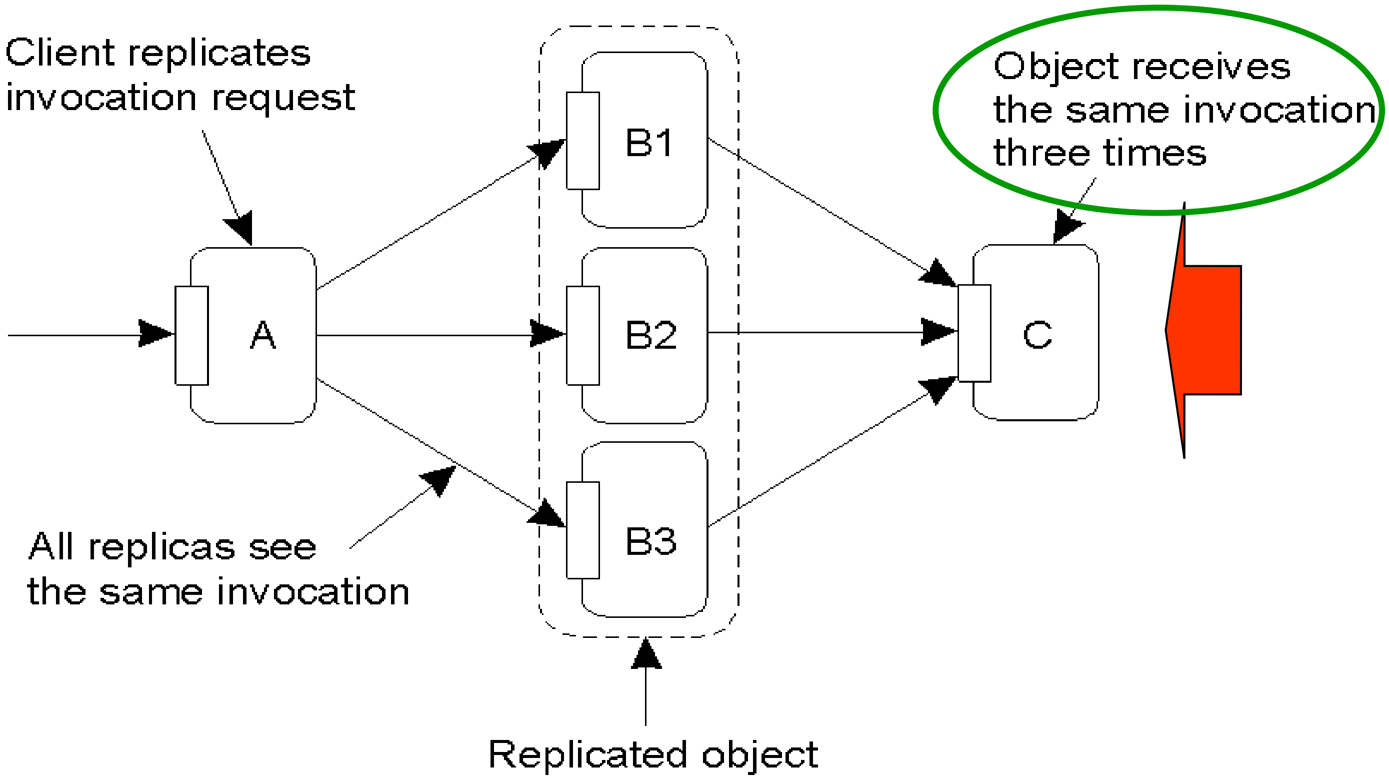
Write operations performed locally. Useful for a disconnected mobile computer

# Replicated Write Protocols

## Active Replication

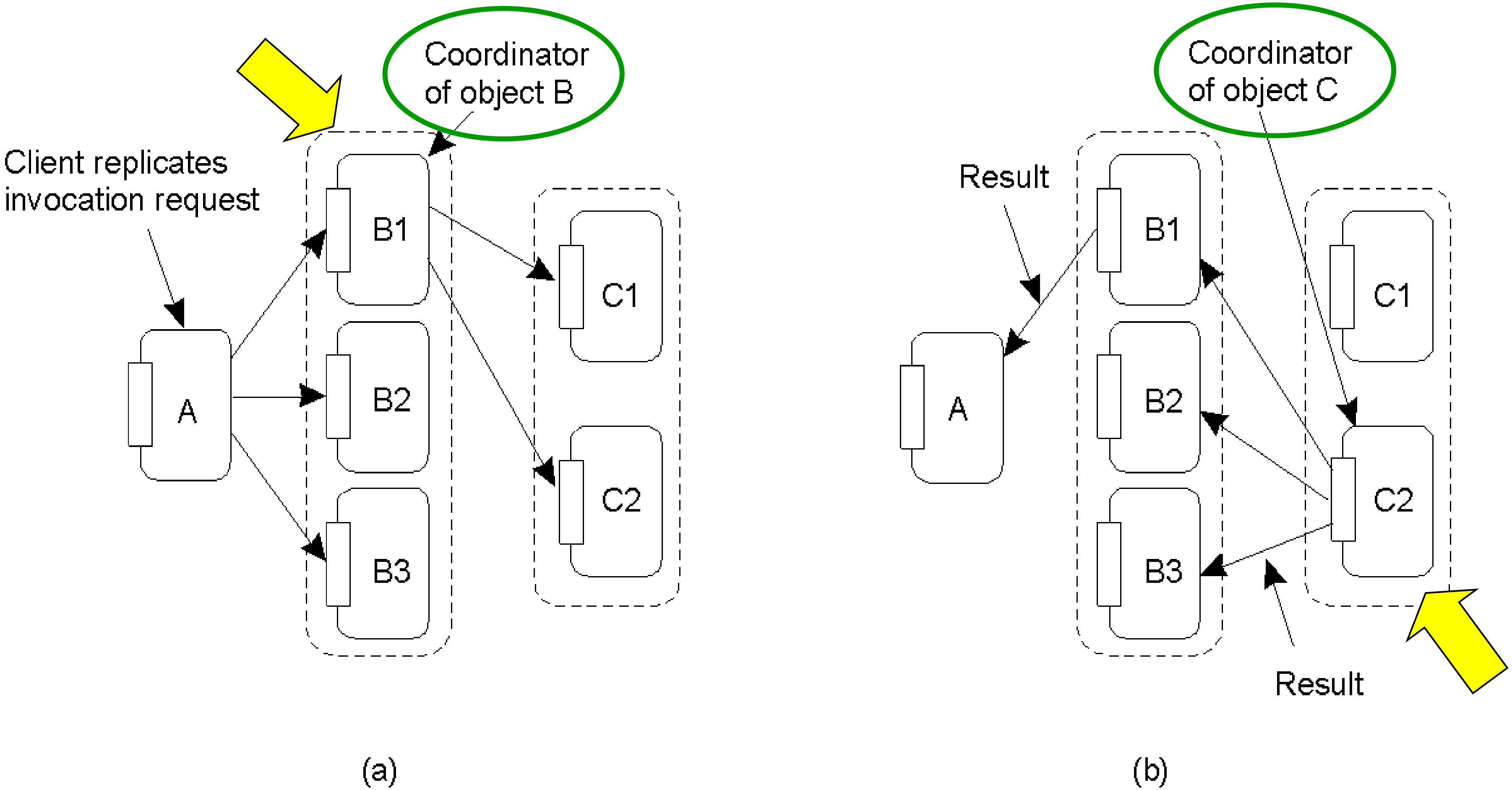
Each replica has an associated process that carries out update operations.
Updates are propagated by means of the write operation that causes the update

Upgrades need to maintain operations order (Lamport timestamps or coordinator);




Problems of replicated invocations: multiple invocations of the same object can produce errors

# Active Replication

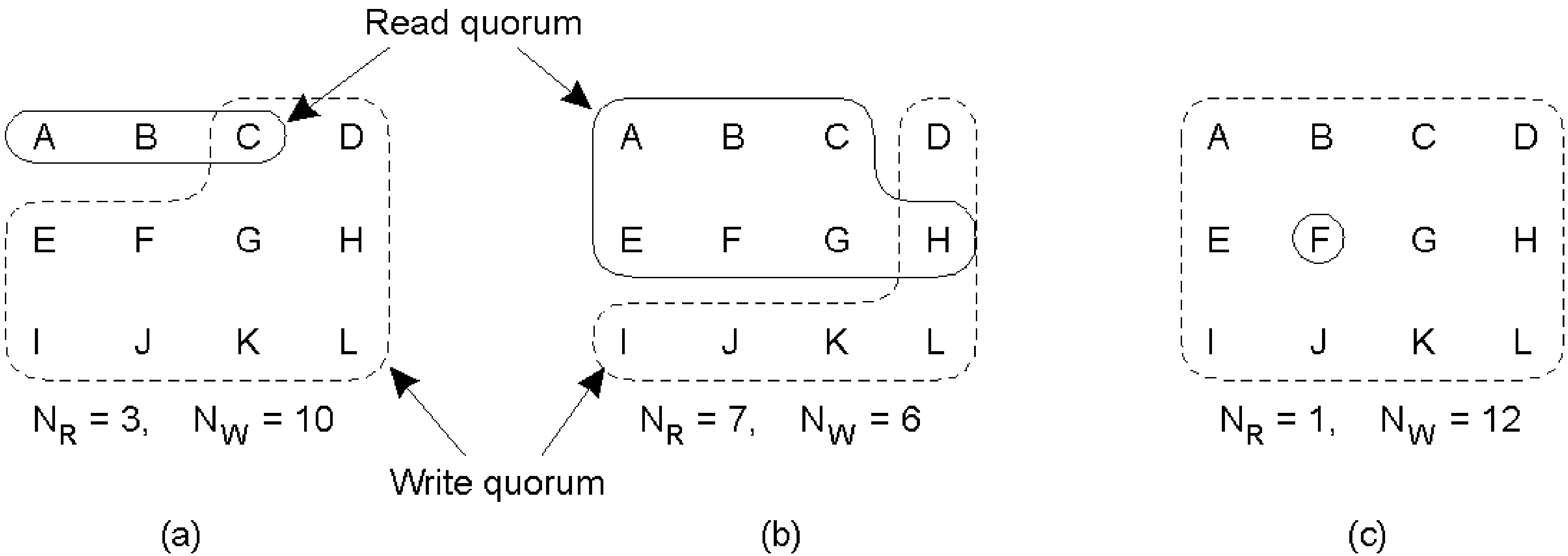



a) Forwarding an invocation request from a replicated object (via unique ID) .

b) Returning a reply to a replicated object from a replicated object.

# Replicated Write Protocols
# Quorum-Based Protocols

clients request and acquire permission of multiple server before accessing data




$N_R + N_W > N$

$N_W > N/2$

Three examples of the voting algorithm:

a) A correct choice of read and write set

b) A choice that may lead to write-write conflicts ( $N_W = N/2$)

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

# Cache Coherence Protocols

Caching can be analyzed according to different parameters

**Coherence detection strategy (when)**

- verification of consistency before cached data accessed
- no verification : data are assumed consistent
- verification after cached data used

**Coherence enforcement strategy (how)**

- no cached shared data (only at servers)
- servers send invalidation messages to all caches
- servers propagate updates

**Write-through cache**

- clients modify cached data and forward updates to servers