Consistency and Replication

Chapter 6

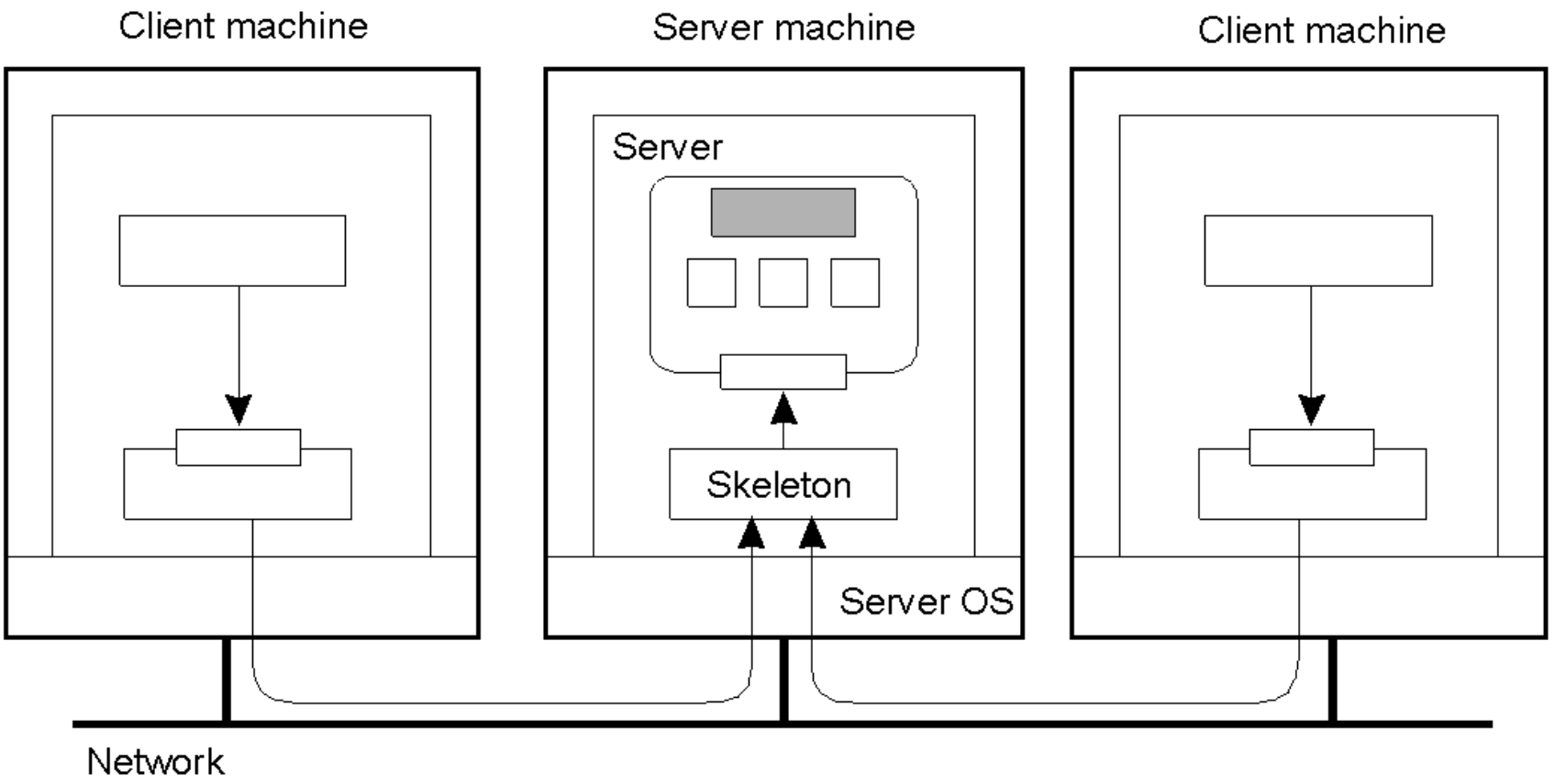
Reasons for Replication

- Data replication is a common technique in distributed systems. There are two reasons for data replication:
  - It creases the **reliability** of a system.
    - If one replica is unavailable or crashes, use another
    - Protect against corrupted data
  - It improves the **performance** of a system.
    - Scale with size of the distributed system (replicated Web servers)
    - Scale in geographically distributed systems (Web proxies)
- The key issue is the need to maintain **consistency** of replicated data.
  - If one copy is modified, others become inconsistent.

Object Replication

- There are two approaches for object sharing:
  - The object itself can handle concurrent invocation.
    - A Java object can be constructed as a monitor by declaring the object's methods to be synchronized.
  - The object is completely unprotected against concurrent invocations, but the server in which the object resides is made responsible for concurrency control.
    - In particular, use an appropriate object adapter.

Object Replication

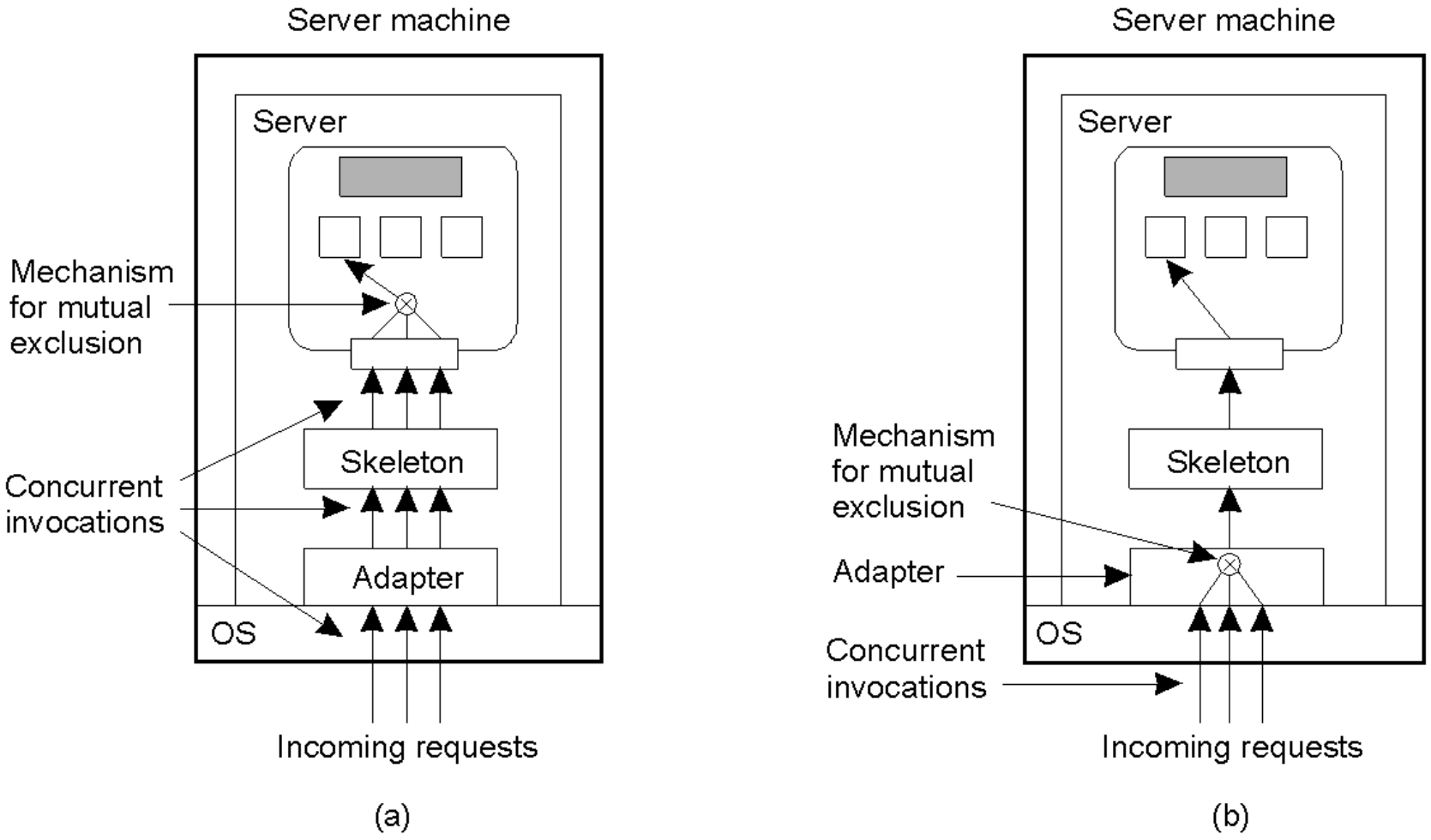


Organization of a distributed remote object shared by two different clients.

Object Replication

- There are two approaches for object replication:
  - The application is responsible for replication.
    - Application needs to handle consistency issues.
  - The system (middleware) handles replication.
    - Consistency issues are handled by the middleware.
    - It simplifies application development but makes object-specific solutions harder.

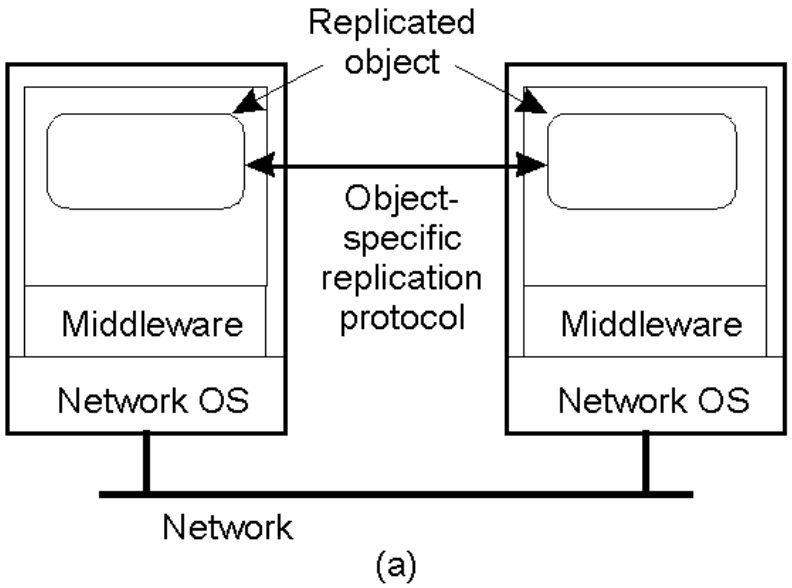
Object Replication

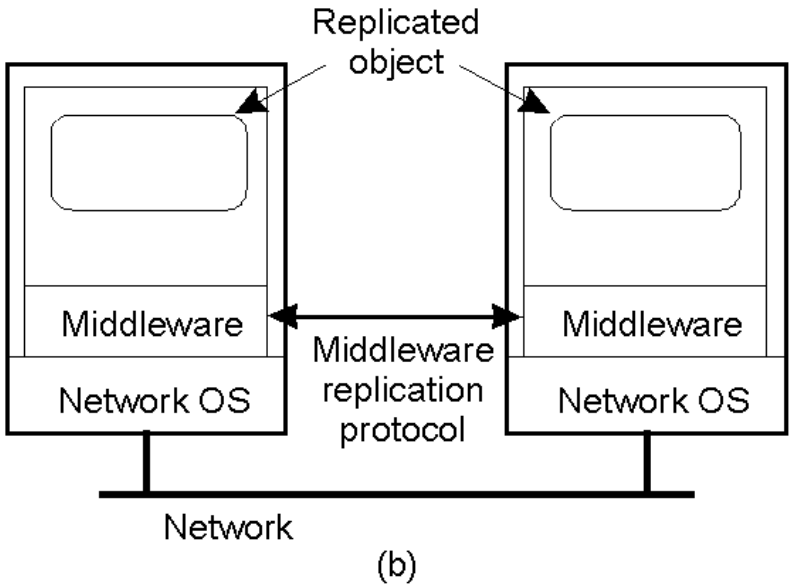


a) A remote object capable of handling concurrent invocations on its own.

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

Object Replication





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

b) A distributed system responsible for replica management

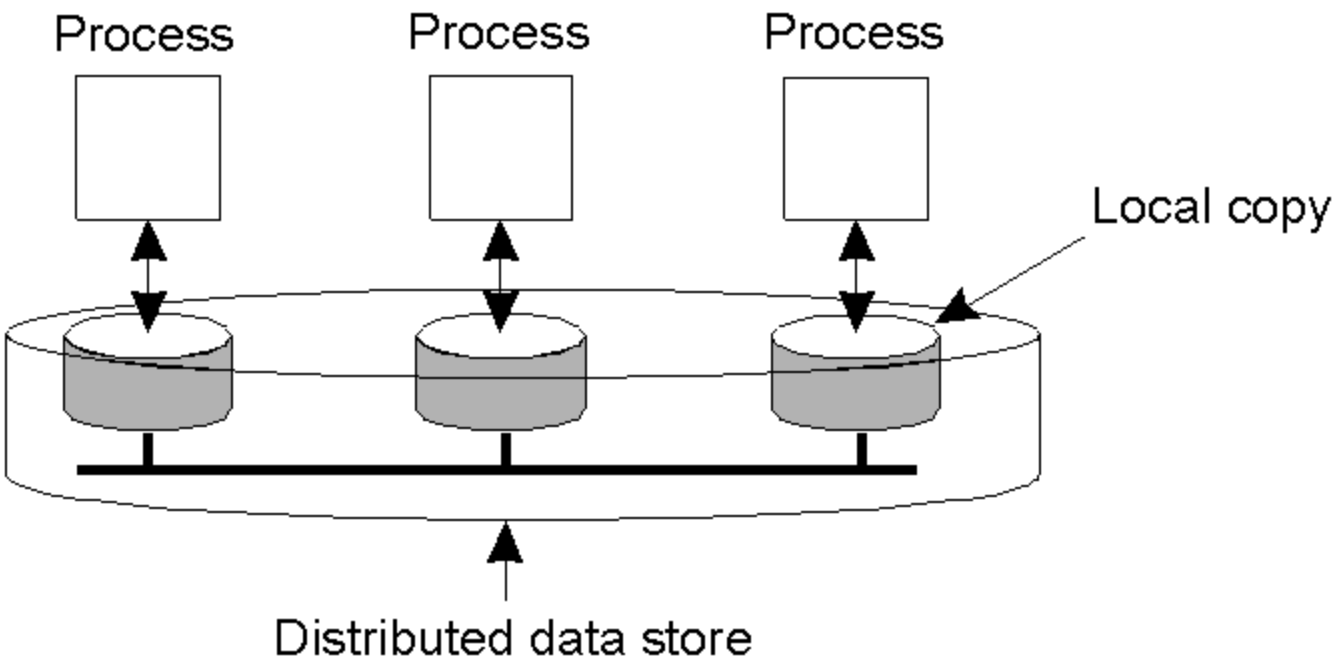
Replication and Scaling

- Replication and caching are used for system scalability.
- Multiple copies improve performance by reducing access latency but have higher network overheads of maintaining consistency.
  - Example: An object is replicated N times.
    - Consider the Read frequency R and the write frequency W
    - If $R << W$, high consistency overhead and wasted messages
    - Consistency maintenance is itself an issue
      - What semantics to provide?
      - Tight consistency requires globally synchronized clocks.

Replication and Scaling

- The solution is to loosen consistency requirements.
  - Variety of consistency semantics possible
- Consistency model (consistency semantics)
  - Contract between processes and the data store
    - If processes obey certain rules, data store will work correctly.
  - All models attempt to return the results of the last write for a read operation.
    - Differ in how last write is determined/defined

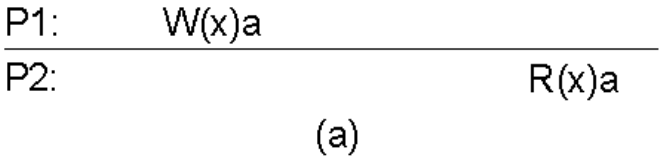
Data-Centric Consistency Models



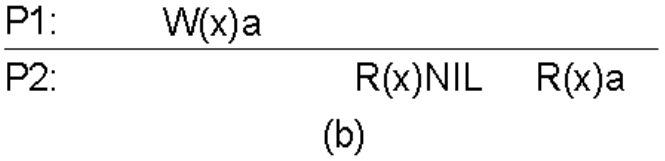
The general organization of a logical **data store**, physically distributed and replicated across multiple processes.

Strict Consistency

- **Definition:** Any read on a data item X returns a value corresponding to the result of the most recent write on X.
- This definition implicitly assumes the existence of absolute global time. Naturally available in uni-processor systems, but impossible to implement in distributed systems.



(a)

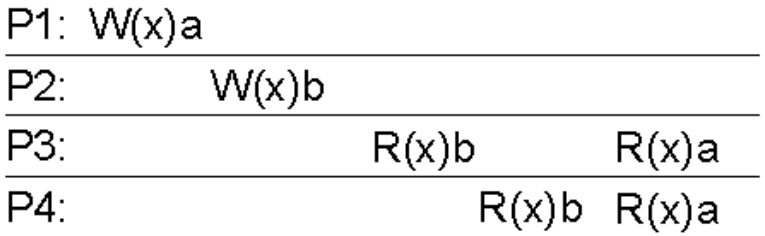


(b)

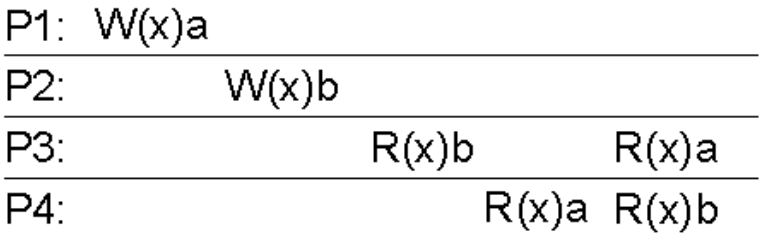
- Behavior of two processes, operating on the same data item.
  - A strictly consistent store.
  - A store that is not strictly consistent.

Sequential Consistency

- **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.
- All processes see the same interleaving of (write) operations.



(a)



(b)

a) A sequentially consistent data store.

b) A data store that is not sequentially consistent.

Linearizability

- **Definition:** 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 TSop1(x) < TSop2(y), then operation OP1(x) should precede OP2(y) in this sequence.**
- In this model, operations are assumed to receive a timestamp using a globally available clock with finite precision.
- A linearizable data store is also sequentially consistent, but it is more expensive to implement than sequential consistency
- Linearizability is primarily used to assist formal verification of concurrent programs,

Analysis of Sequential Consistency

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

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

Four valid execution sequences for the processes of the previous slide.

Sequential Consistency and Serializability

- **Definition:** Sequential consistency is comparable to serializability in the case of transactions.
- The deference is that of granularity: sequential consistency is defined in terms of read and write operations, whereas serializability is defined in terms of transactions, which aggregate such operations.
- Sequential consistency is a programmer-friendly model, but it has serious performance problems. So other weaker consistency models have been proposed.

Causal Consistency

- Causal consistency requires a total order of causally related write operations only.
  1. A read is causally related to the write that provided the data the read got.
  2. A write is causally related to a read that happened before this write in the same process.
  3. If write1 $\rightarrow$ read, and read $\rightarrow$ write2, then write1 $\rightarrow$ write2.
- Necessary condition for causal consistency:
  Writes that are potentially casually related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

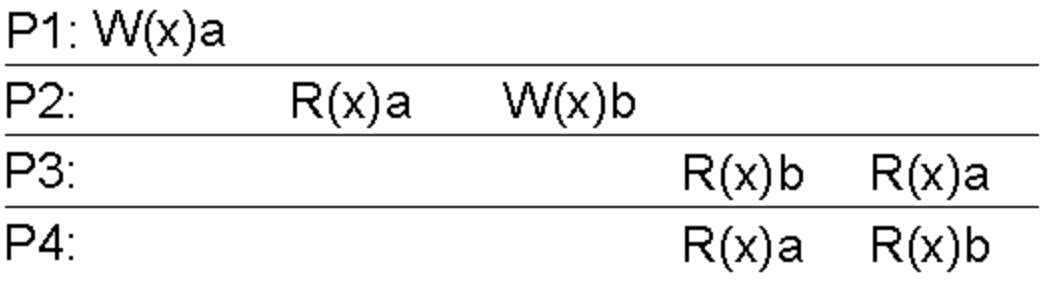
Casual Consistency

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

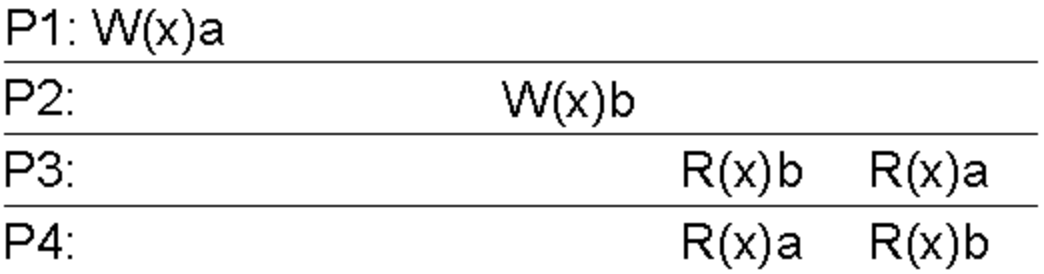
- This sequence is allowed with a casually-consistent store, but not with sequentially or strictly consistent store.

Note: W1(x)a $\rightarrow$ W2(x)b, but W2(x)b $\|$ W1(x)c

Casual Consistency



(a)



(b)

a) A violation of a casually-consistent store.

b) A correct sequence of events in a casually-consistent store.

FIFO Consistency

- Necessary Condition:
  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 a different order by different processes.

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

FIFO Consistency

(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 the three processes from the previous slide. The statements in bold are the ones that generate the output shown.

FIFO Consistency

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

Two concurrent processes.

Weak Consistency

- 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.
- You don't care that reads and writes of a series of operations are immediately known to other processes. You just want the effect of the series itself to be known.

Weak Consistency

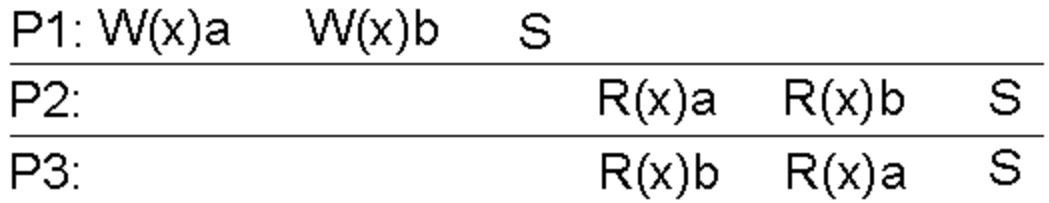
```
int a, b, c, d, e, x, y;          /* variables */
int *p, *q;                       /* pointers */
int f( int *p, int *q);           /* function prototype */

a = x * x;                        /* a stored in register */
b = y * y;                        /* b as well */
c = a*a*a + b*b + a * b;          /* used later */
d = a * a * c;                    /* used later */
p = &a;                           /* p gets address of a */
q = &b                            /* q gets address of b */
e = f(p, q)                       /* function call */
```

A program fragment in which some variables may be kept in registers.

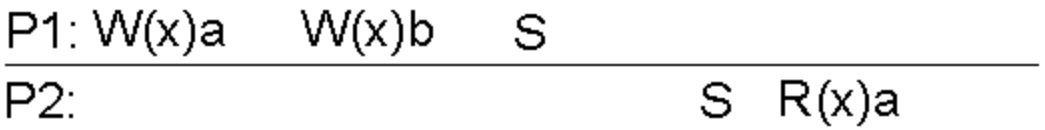
Weak Consistency

- 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.
- You don't care that reads and writes of a series of operations are immediately known to other processes. You just want the effect of the series itself to be known.

Weak Consistency



(a)



(b)

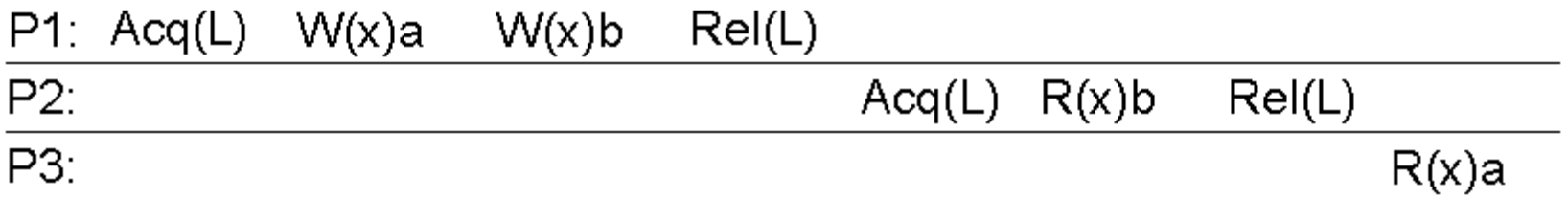
a) A valid sequence of events for weak consistency.

b) An invalid sequence for weak consistency.

- Weak consistency implies that we need to lock and unlock data (implicitly or not).

Release Consistency

- Divide access to a synchronization variable into two parts: an acquire and a release phase. Acquire forces a requester to wait until the shared data can be accessed; release sends requester's local value to other servers in data store.



A valid event sequence for release consistency.

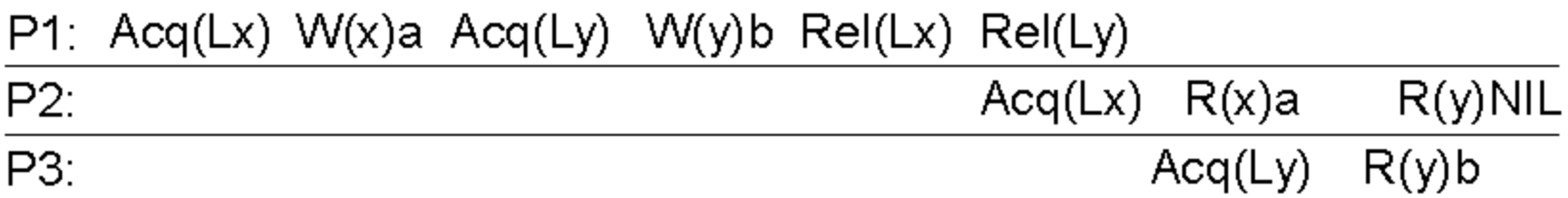
Release Consistency

- 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 FIFO consistent (sequential consistency is not required).

Entry Consistency

- 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



- A valid event sequence for entry consistency.
- Where release consistency affects all shared data, entry consistency affects only those shared data associated with a synchronization variable.

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.

ClientCentric Consistency Models

- Goal: Show how we can perhaps avoid systemwide consistency, by concentrating on what specific clients want, instead of what should be maintained by servers.
- Background: Most largescale distributed systems (i.e., databases) apply replication for scalability, but can support only weak consistency:
- DNS: Updates are propagated slowly, and inserts may not be immediately visible.
- NEWS: Articles and reactions are pushed and pulled throughout the Internet, such that reactions can be seen before postings.

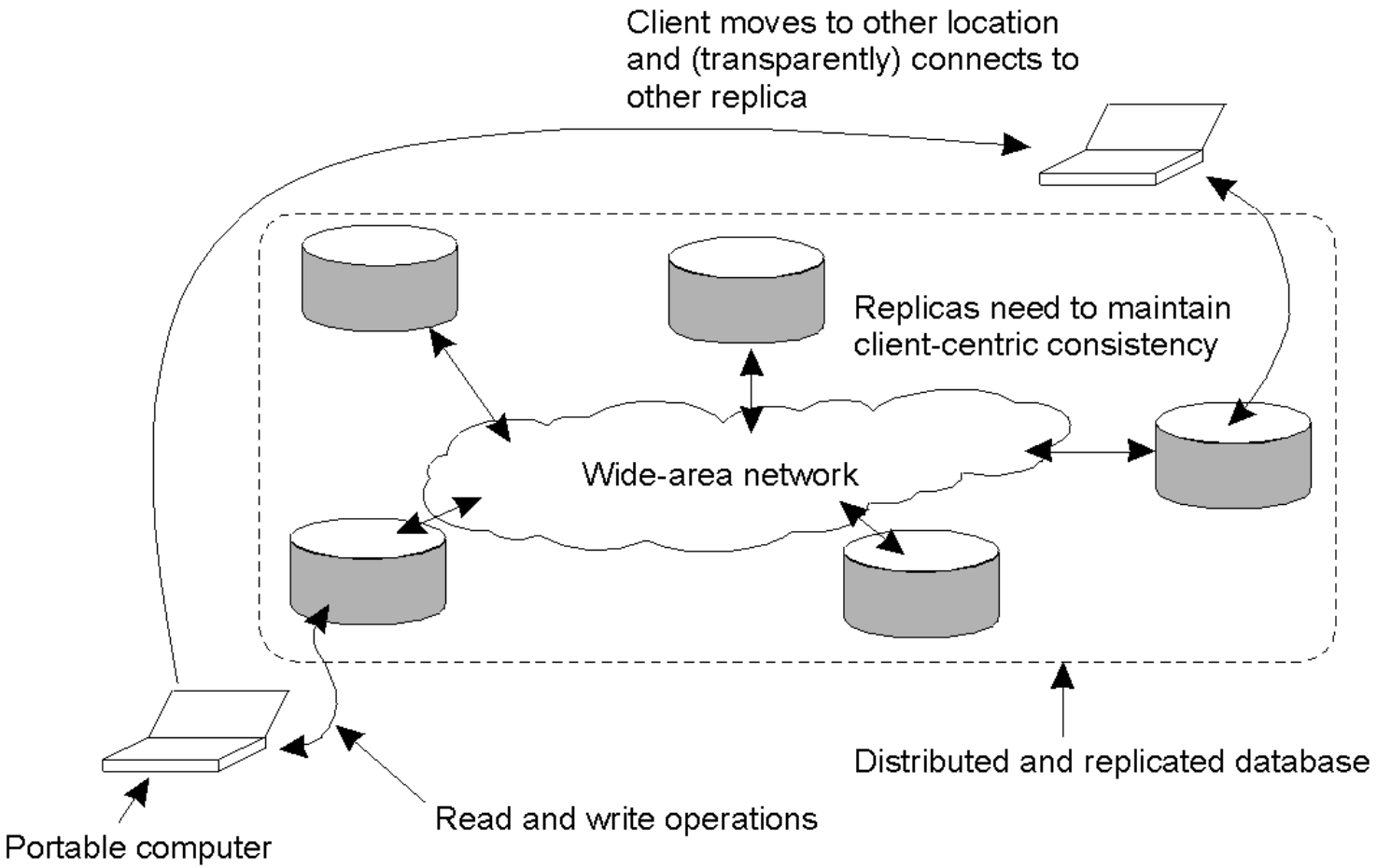
ClientCentric Consistency Models

- Lotus Notes: Geographically dispersed servers replicate documents, but make no attempt to keep (concurrent) updates mutually consistent.
- WWW: Caches all over the place, but there need be no guarantee that you are reading the most recent version of a page.

Consistency for Mobile Users

- Example: Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.
  - At location A you access the database doing reads and updates.
  - At location B you continue your work, but unless you access the same server as the one at location A, you may detect inconsistencies:
    - your updates at A may not have yet been propagated to B
    - you may be reading newer entries than the ones available at A
    - your updates at B may eventually conflict with those at A
- Note: The only thing you really want is that the entries you updated and/or read at A, are in B the way you left them in A. In that case, the database will appear to be consistent to you.

Eventual Consistency

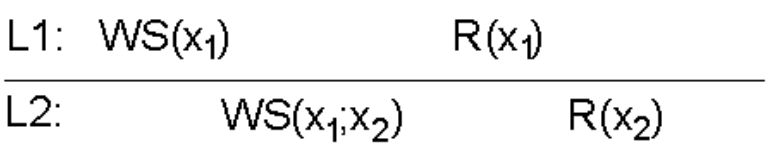


The principle of a mobile user accessing different replicas of a distributed database.

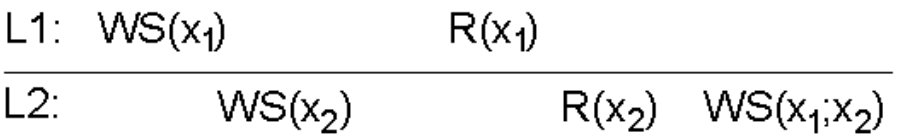
Monotonic Reads

- 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 or a more recent value.
- Example: Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.
- Example: Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different email server, that server fetches (at least) all the updates from the server you previously visited.

Monotonic Reads



(a)



(b)

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

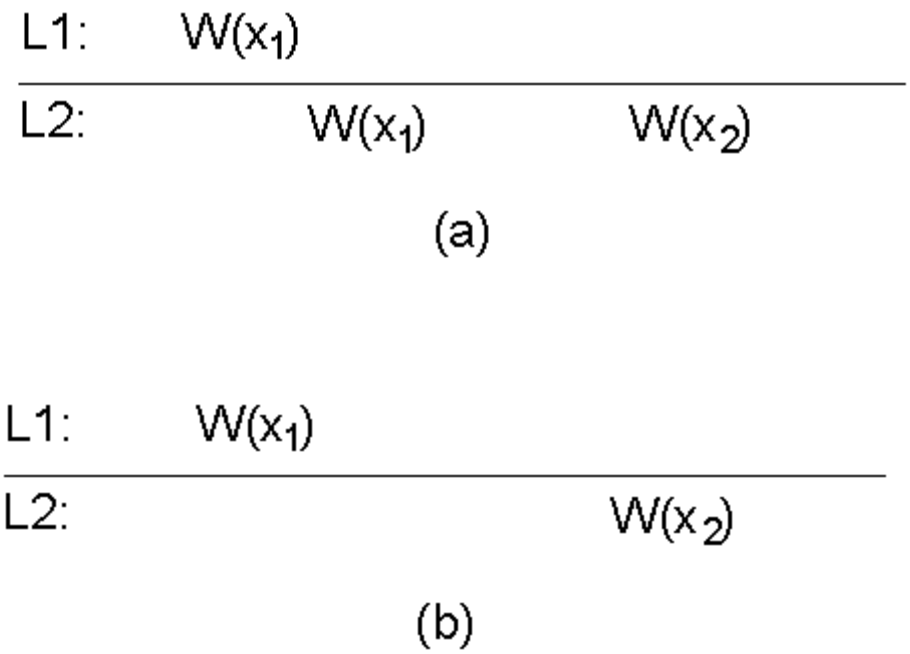
a) A monotonic-read consistent data store

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

Monotonic Writes

- If a A write operation by a process on a data item x is completed before any successive write operation on x by the same process.
- Example: Updating a program at server S2 , and ensuring that all components on which compilation and linking depends, are also placed at S2 .
- Example: Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

Monotonic Writes



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

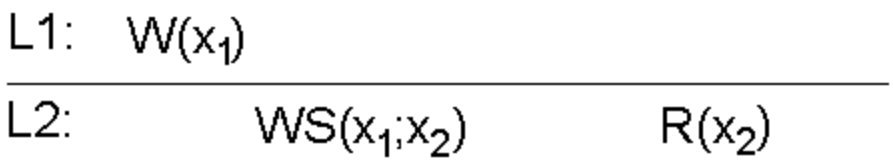
a) A monotonic-write consistent data store.

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

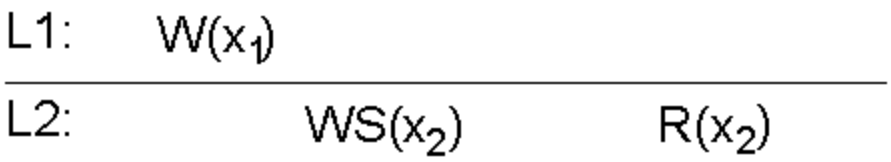
Read Your Writes

- 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.
- Example: Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.

Read Your Writes



(a)



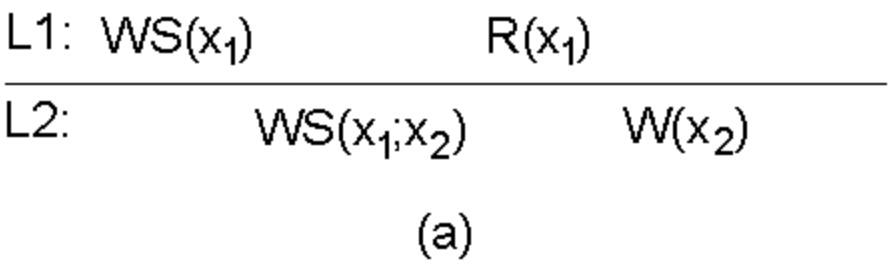
(b)

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

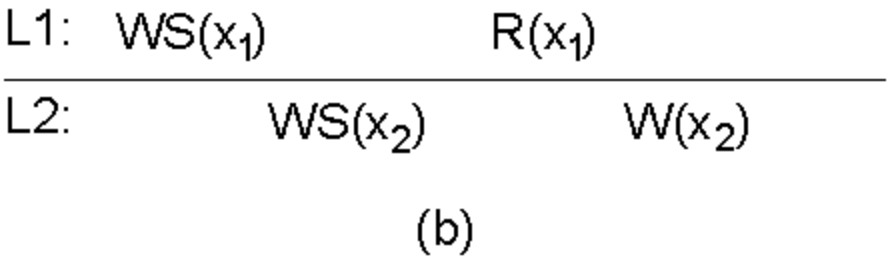
Writes Follow Reads

- 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.
- Example: Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.
- 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.
- Example: See reactions to posted articles only if you have the original posting (a read ``pulls in" the corresponding write operation).

Writes Follow Reads



(a)



(b)

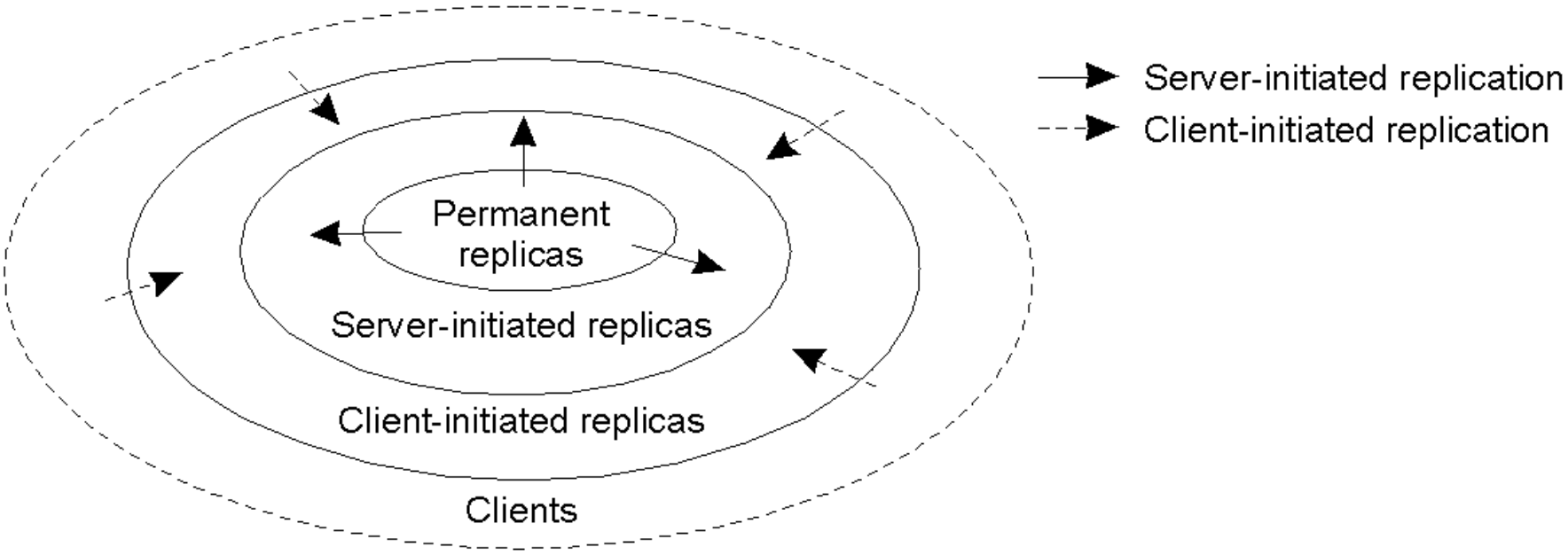
a) A writes-follow-reads consistent data store

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

Replica Placement

- The effect Model: We consider objects (and don't worry whether they contain just data or code, or both)
- **Distinguish different processes**: A process is capable of hosting a replica of an object or data:
  - **Permanent replicas**: Process/machine always having a replica
  - **Serverinitiated replica**: Process that can dynamically host a replica on request of another server in the data store
  - **Clientinitiated replica**: Process that can dynamically host a replica on request of a client (client cache)

Replica Placement

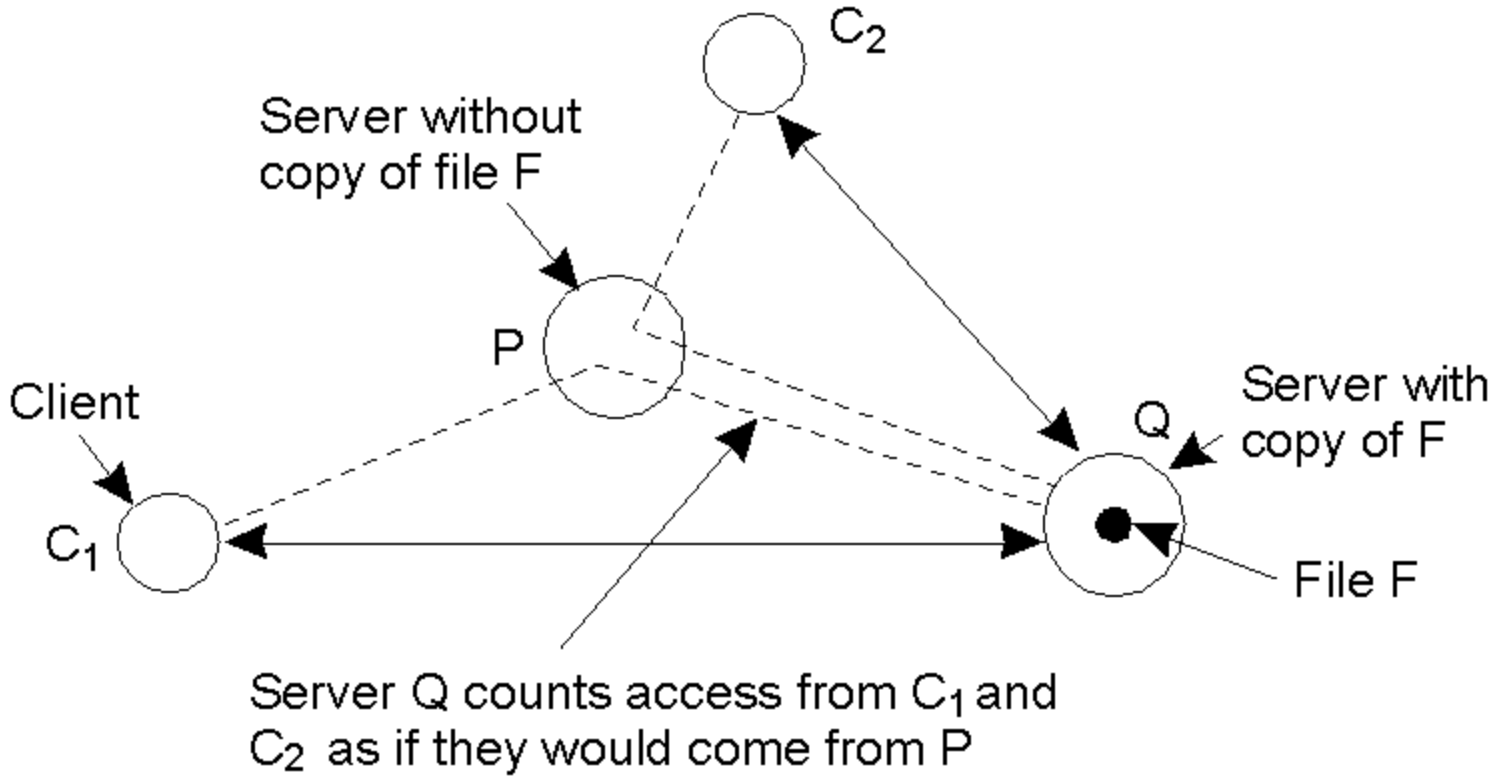


The logical organization of different kinds of copies of a data store into three concentric rings.

Server-Initiated Replicas

- The Keep track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below threshold D drop file
- Number of accesses exceeds threshold R replicate file
- Number of access between D and R migrate file

Server-Initiated Replicas



Counting access requests from different clients.

Update Propagation

- There are three possibilities of propagation update:
  - The Propagate only notification/invalidation of update (often used for caches)
  - Transfer data from one copy to another (distributed databases)
  - Propagate the update operation to other copies (also called active replication)
- No single approach is the best, but depends highly on available bandwidth and readtowrite ratio at replicas.

Pull versus Push Protocols

- There Pushing updates: serverinitiated approach, in which update is propagated regardless whether target asked for it.
- Pulling updates: clientinitiated approach, in which client requests to be updated.

Issue	Push-based	Pull-based
State of 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

Epidemic Algorithms

- **Basic idea**: Assume there are no write--write conflicts:
  - Update operations are initially performed at one or only a few replicas
  - A replica passes its updated state to a limited number of neighbors
  - Update propagation is lazy, i.e., not immediate
  - Eventually, each update should reach every replica
- **Antientropy**: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards
- **Gossiping**: A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well)

System Model

- We consider a collection servers, each storing a number of objects
- Each object O has a primary server at which updates for O are always initiated (avoiding writewrite conflicts)
- An update of object O at server S is always timestamped; the value of O at S is denoted VAL (O, S)
- T (O, S) denotes the timestamp of the value of object O at server S

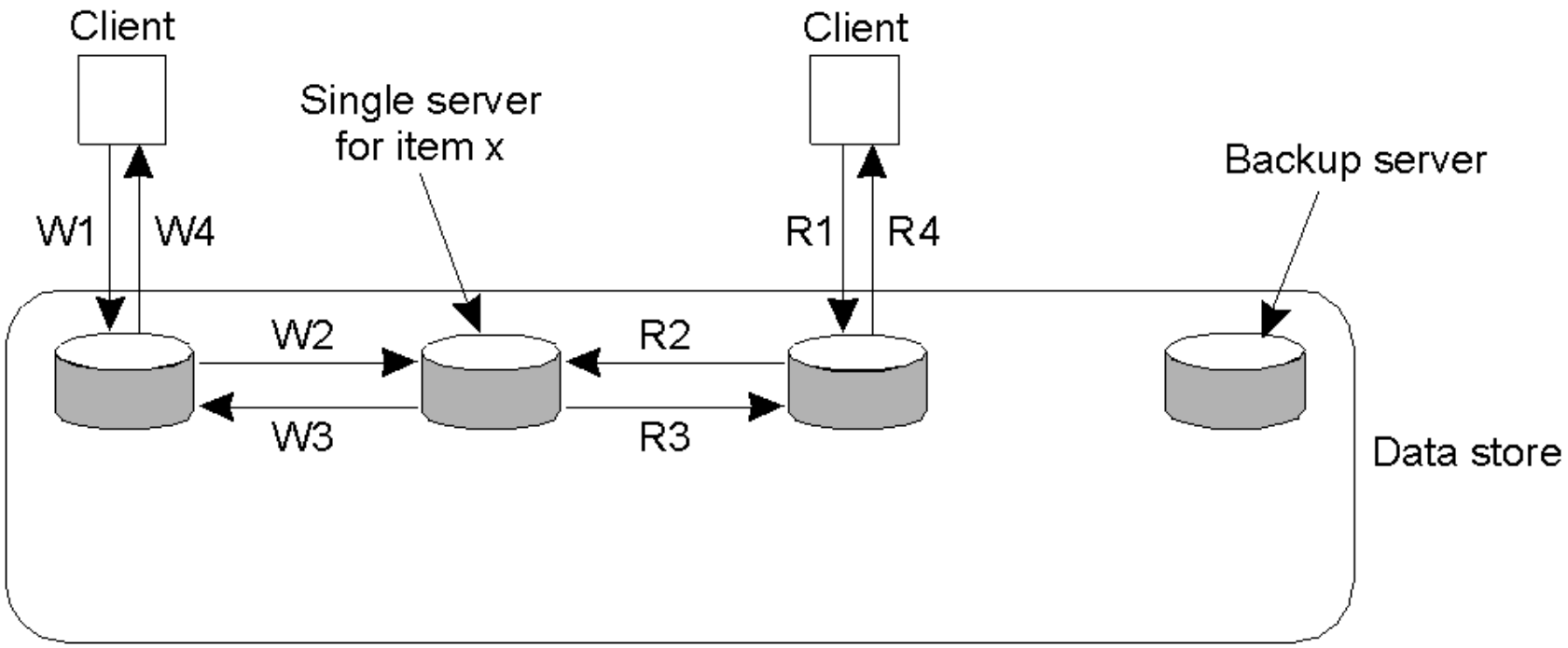
Consistency Protocols

- Consistency protocol: describes the implementation of a specific consistency model. We will concentrate only on sequential consistency.
  - Primary-based protocols
  - Replicated-write protocols
  - Cache-coherence protocols

Consistency Protocols

- Examples of primarybased protocols
  - Used in traditional clientserver systems that do not support replication.
  - Traditionally applied in distributed databases and file systems that require a high degree of fault tolerance. Replicas are often placed on same LAN.
  - Establishes only a fully distributed, nonreplicated data store. Useful when writes are expected to come in series from the same client (e.g., mobile computing without replication)
  - Distributed shared memory systems, but also mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later on).

Remote-Write Protocols

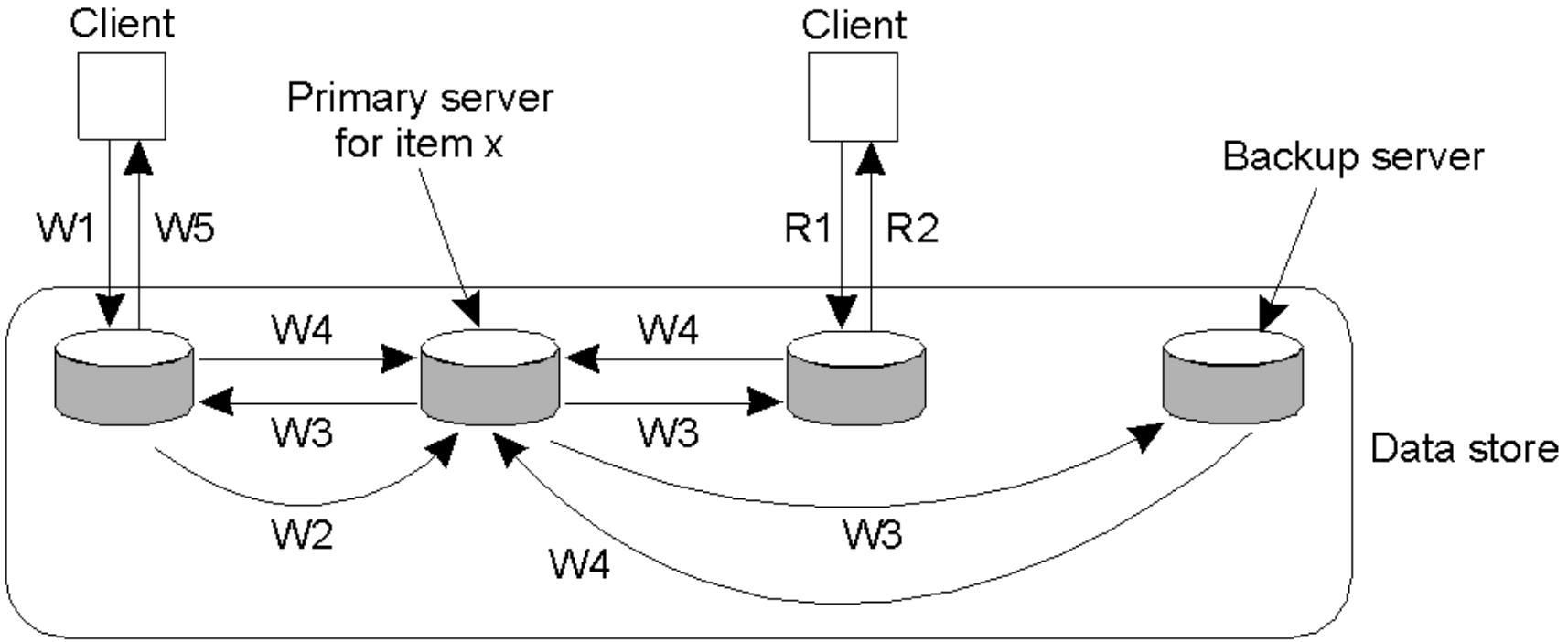


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

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

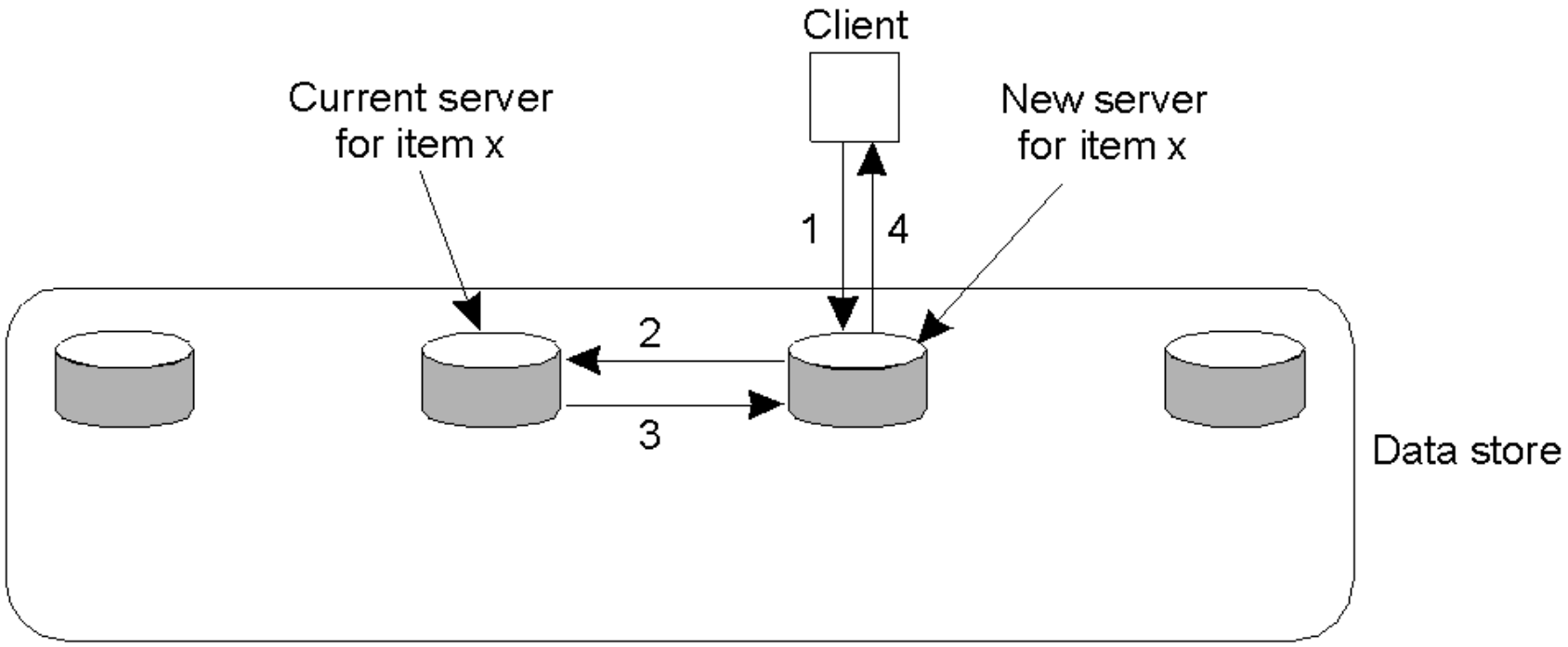


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

The principle of primary-backup protocol.

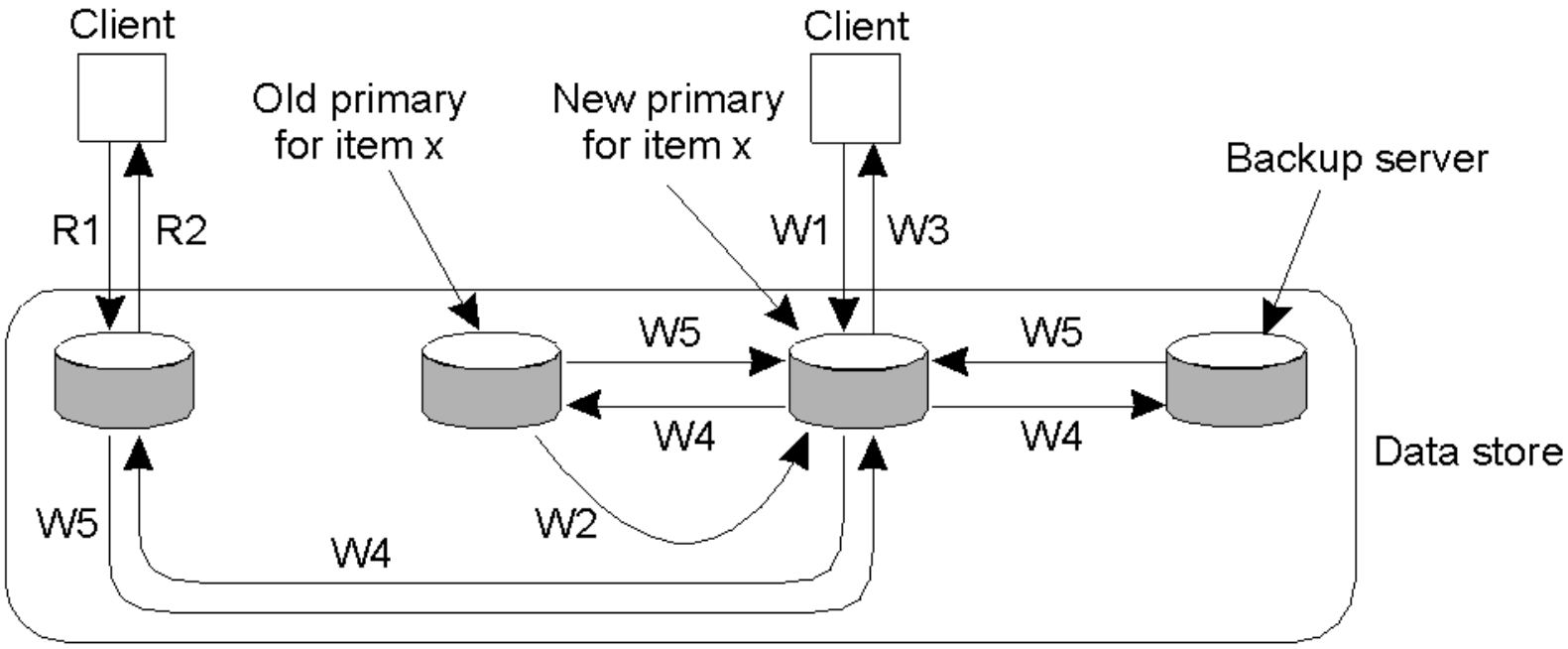
Local-Write Protocols



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.

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

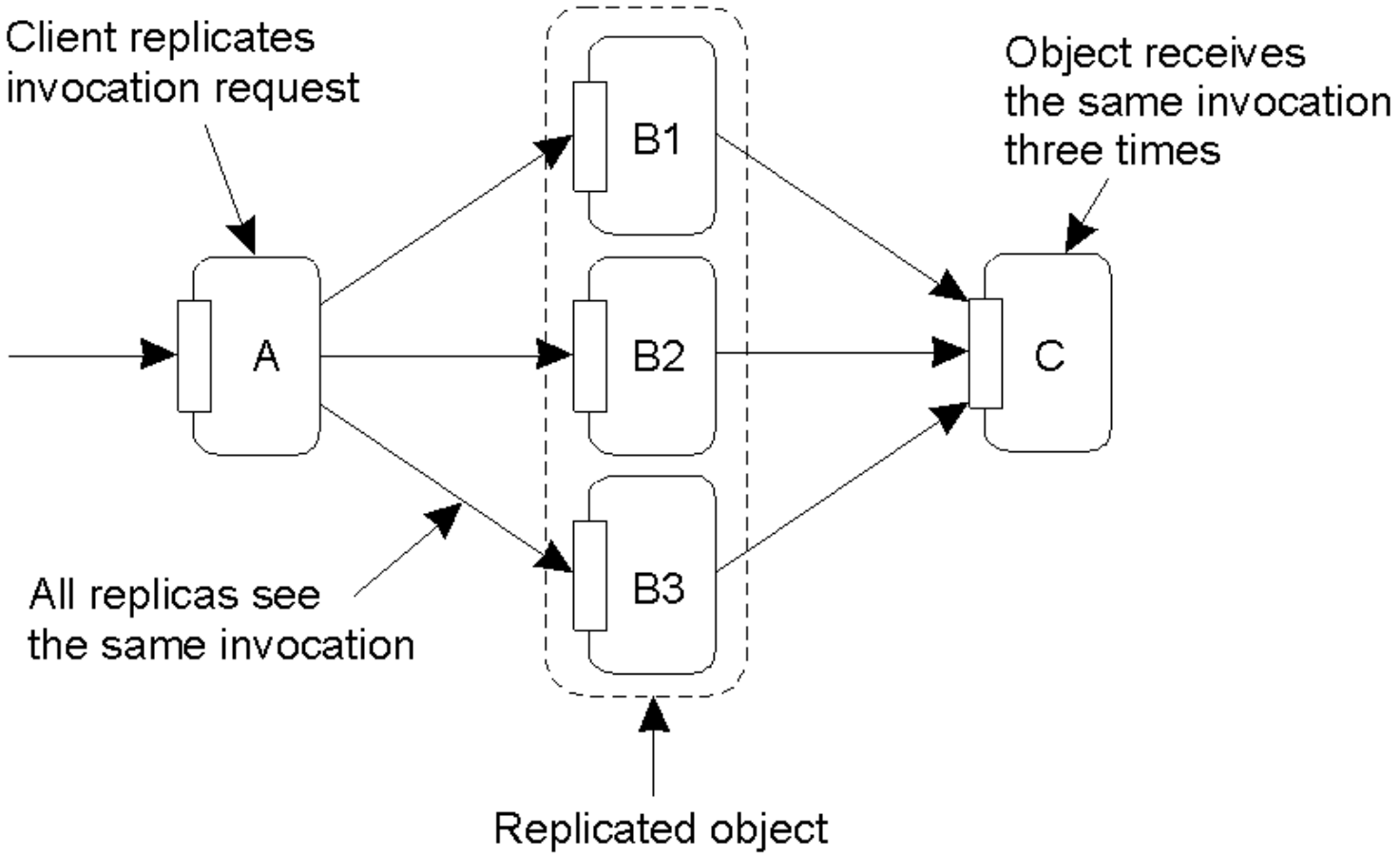
R1. Read request
R2. Response to read

Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

ReplicatedWrite Protocols

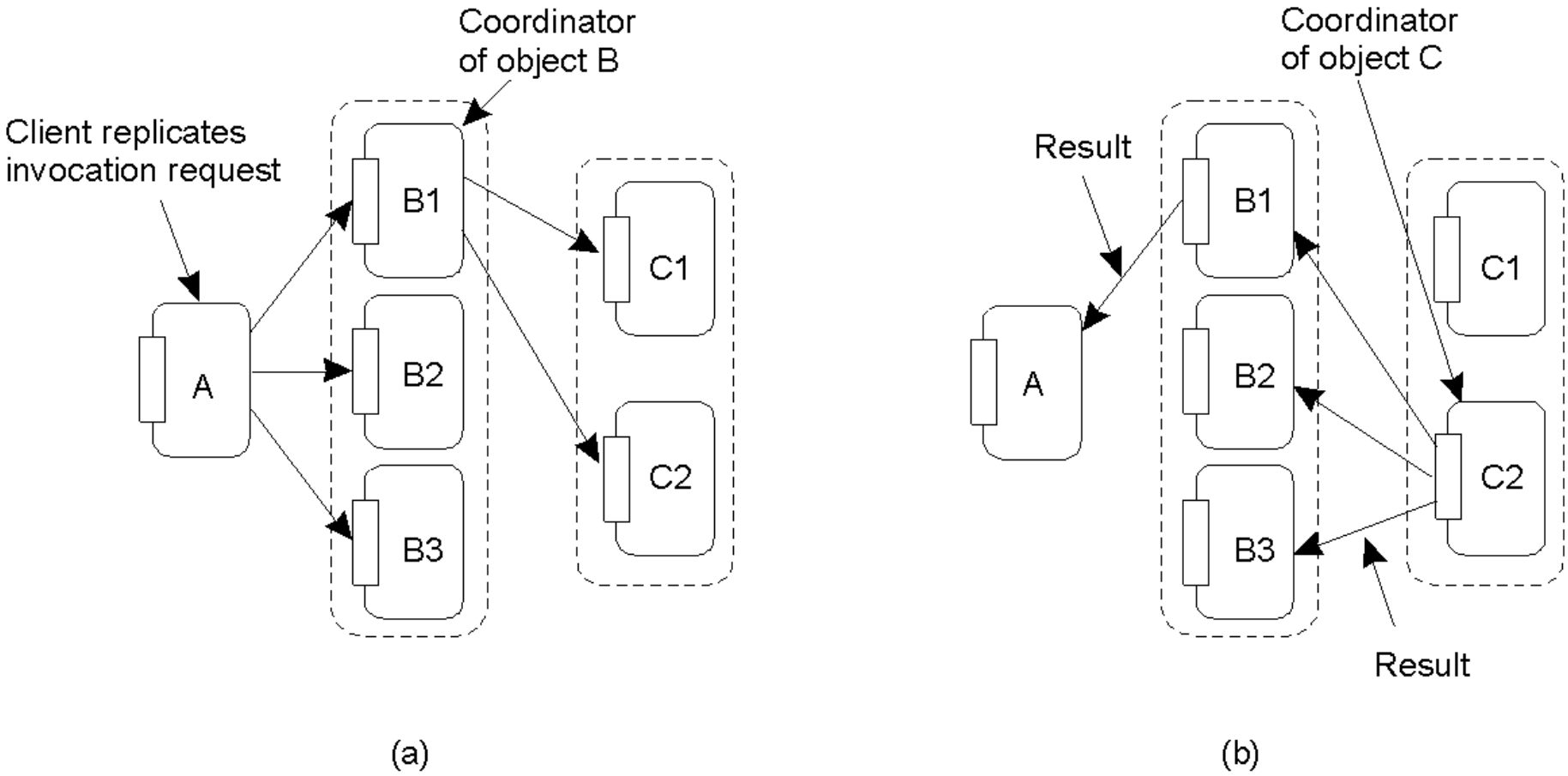
- **Active replication**: Updates are forwarded to multiple replicas, where they are carried out. There are some problems to deal with in the face of replicated invocations.
- **Replicated invocations**: Assign a coordinator on each side (client and server), which ensures that only one invocation, and one reply is sent.
- **Quorumbased protocols**: Ensure that each operation is carried out in such a way that a majority vote is established: distinguish read quorum and write quorum.

Active Replication



The problem of replicated invocations.

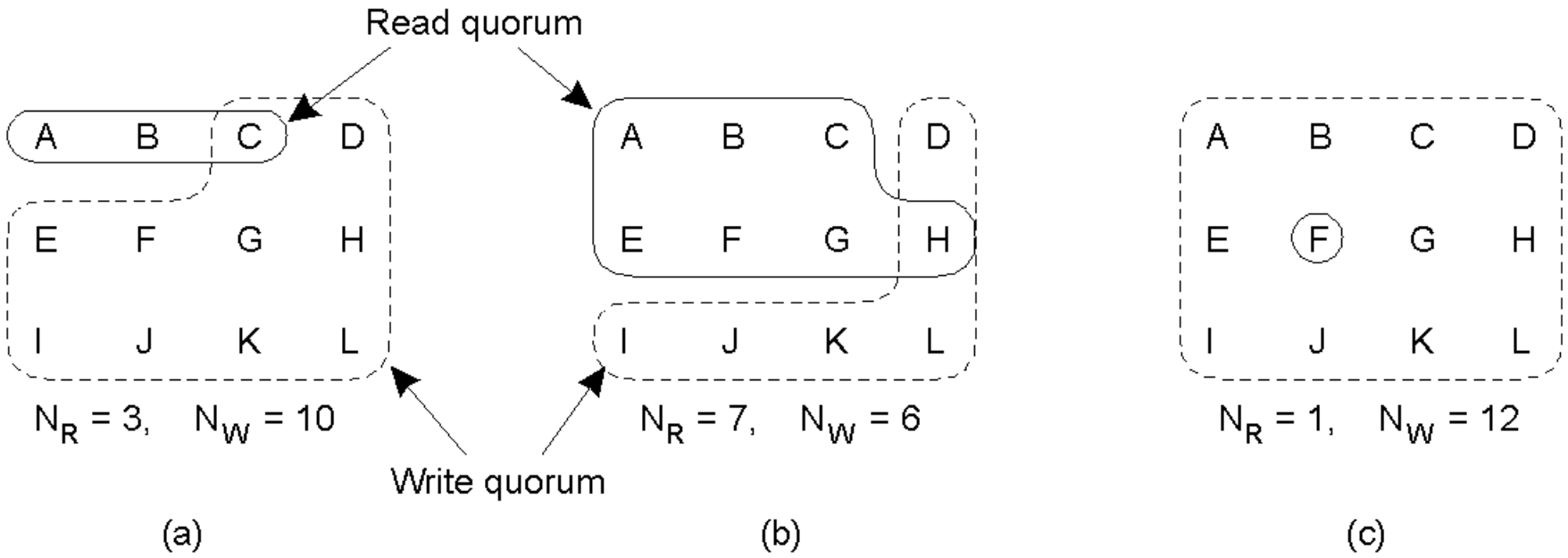
Active Replication



a) Forwarding an invocation request from a replicated object.

b) Returning a reply to a replicated object.

Quorum-Based Protocols



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

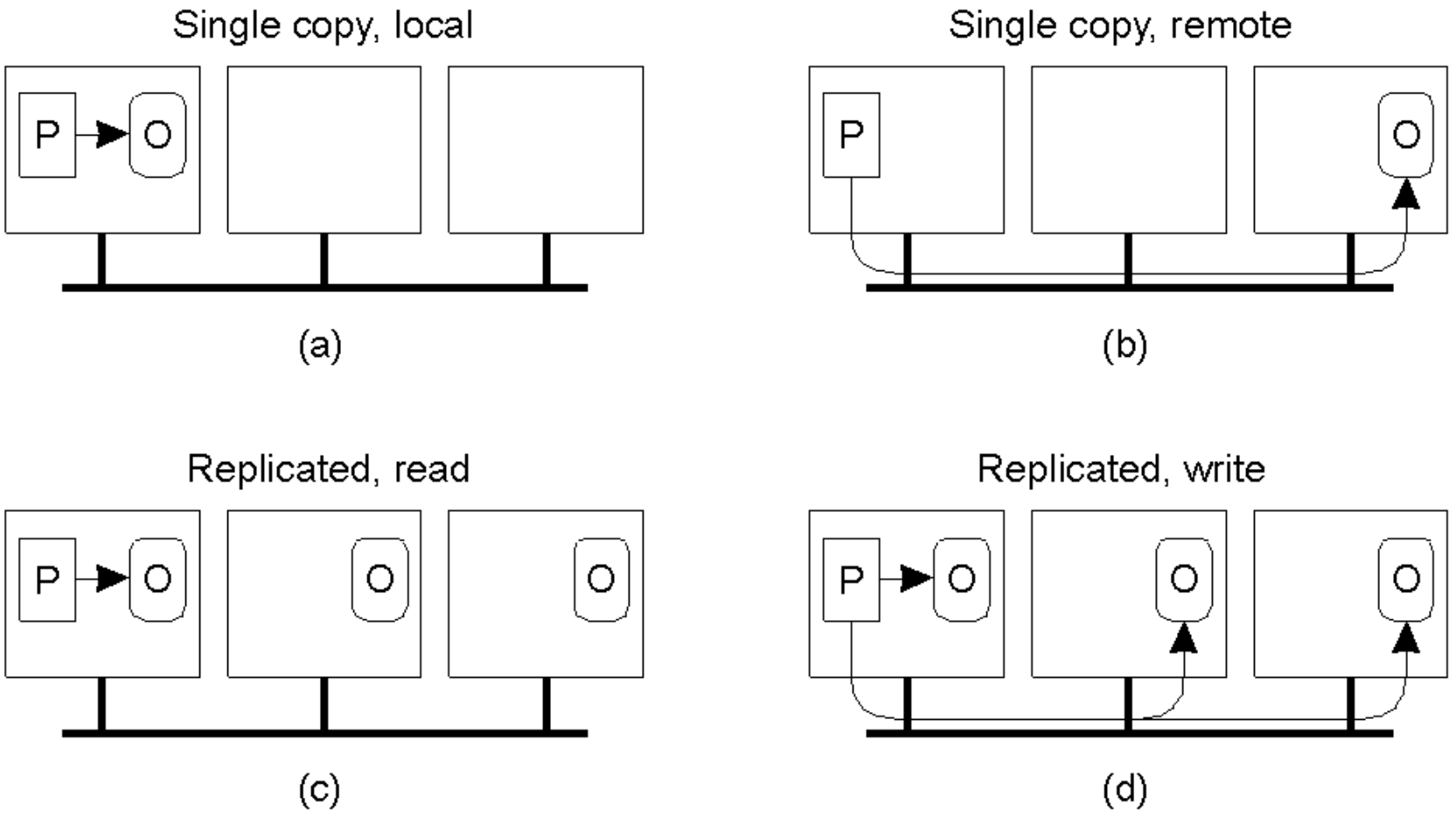
c) A correct choice, known as ROWA (read one, write all)

Orca

```
OBJECT IMPLEMENTATION stack;
  top: integer;                                  # variable indicating the top
  stack: ARRAY[integer 0..N-1] OF integer        # storage for the stack

  OPERATION push (item: integer)                 # function returning nothing
  BEGIN
    GUARD top < N DO
        stack [top] := item;                     # push item onto the stack
        top := top + 1;                          # increment the stack pointer
    OD;
  END;

  OPERATION  pop():integer;                      # function returning an integer
  BEGIN
    GUARD top > 0 DO                             # suspend if the stack is empty
         top := top – 1;                         # decrement the stack pointer
        RETURN stack [top];                      # return the top item
    OD;
  END;
BEGIN
  top := 0;                                      # initialization
END;
```

A simplified stack object in Orca, with internal data and two operations.

Management of Shared Objects in Orca

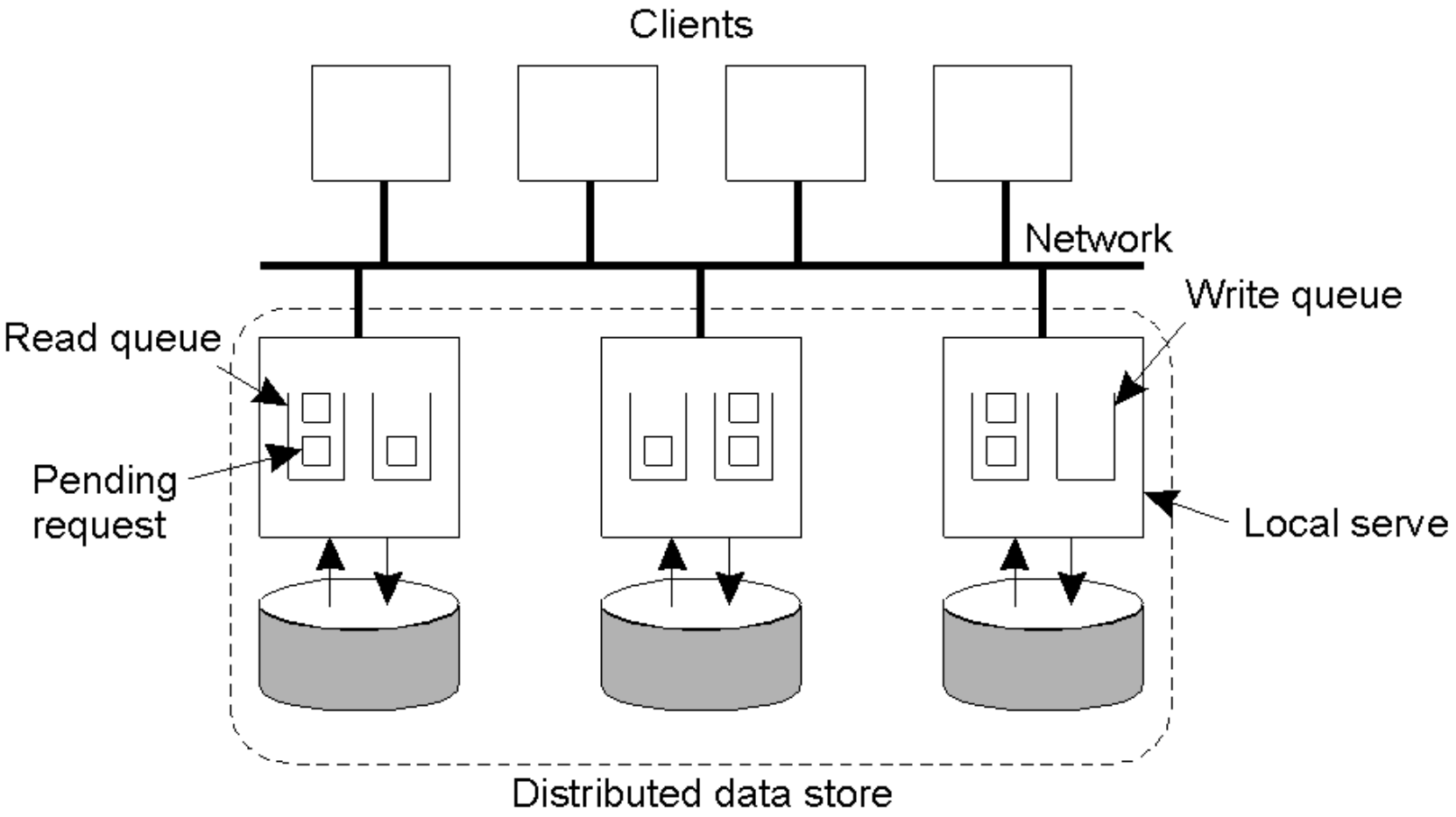


Four cases of a process *P* performing an operation on an object *O* in Orca.

Example: Lazy Replication

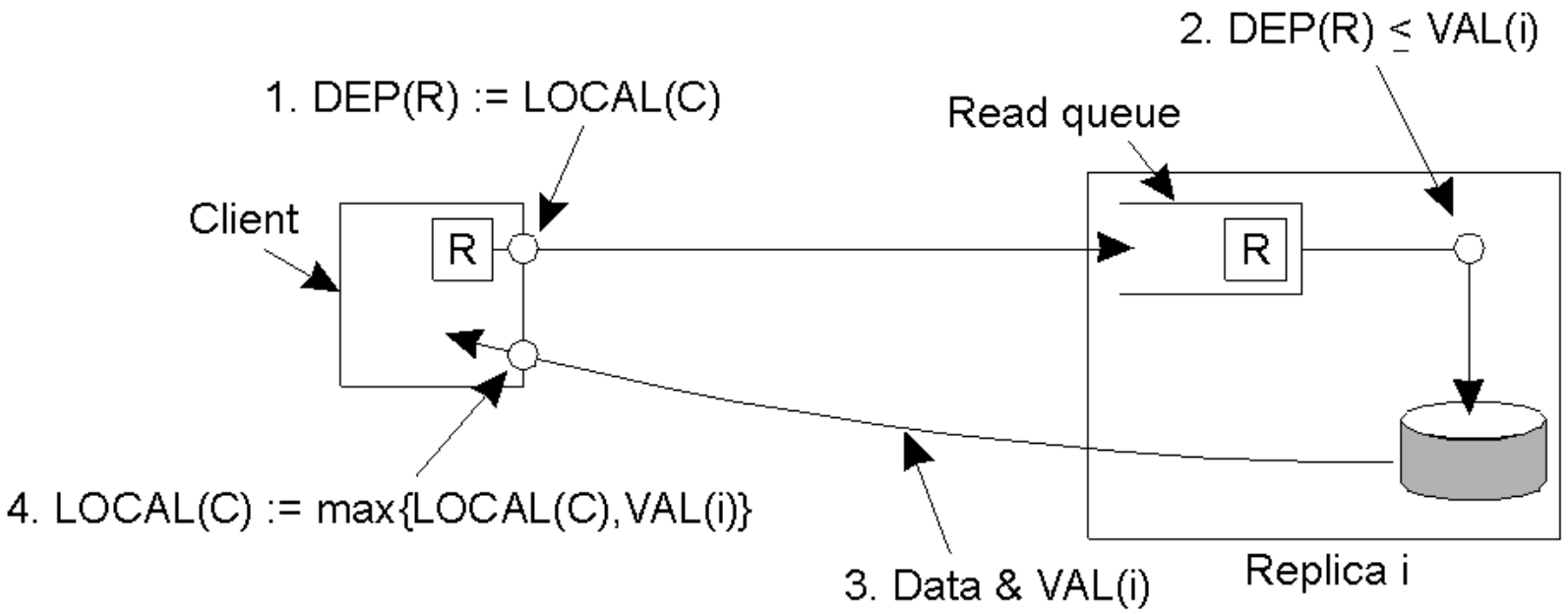
- We asic model: Number of replica servers jointly implement a causalconsistent data store. Clients normally talk to front ends which maintain data to ensure causal consistency.

Casually-Consistent Lazy Replication



The general organization of a distributed data store. Clients are assumed to also handle consistency-related communication.

Processing Read Operations

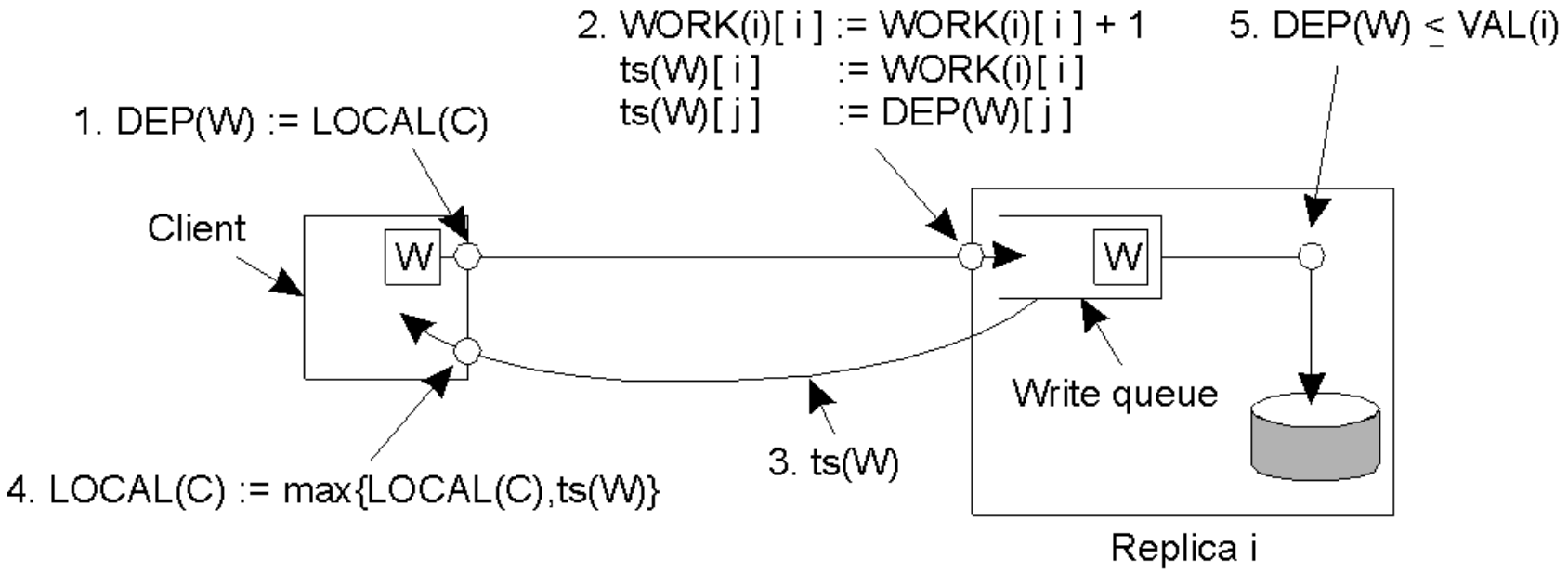


Performing a read operation at a local copy.

Lazy Replication: Vector Timestamps

- VAL(i): VAL(i)[i] denotes the total number of write operations sent directly by a front end (client). VAL(i)[j] denotes the number of updates sent from replica #j.
- WORK(i): WORK(i)[i] total number of write operations directly from front ends, including the pending ones. WORK(i)[j] is total number of updates from replica #j, including pending ones.
- LOCAL(C): LOCAL(C)[j] is (almost) most recent value of VAL(j)[j] known to front end C (will be refined in just a moment)
- DEP(R): Timestamp associated with a request, reflecting what the request depends on.

Processing Write Operations



Performing a write operation at a local copy.