HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Chapter 4: Distributed Systems: Replication and Consistency

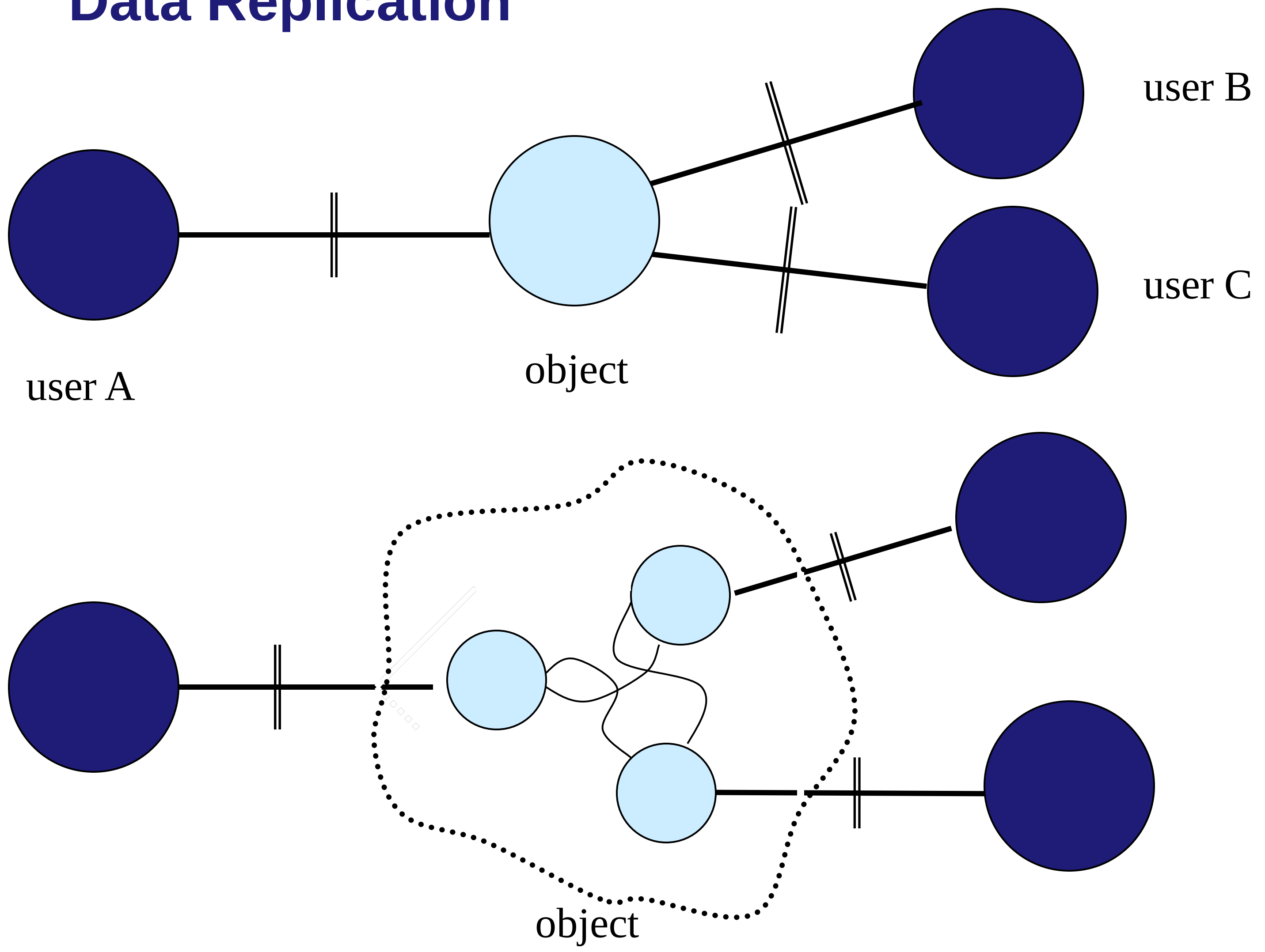
Fall 2012
Sini Ruohomaa

Slides joint work with Jussi Kangasharju et al.

Chapter Outline

- Replication
- Consistency models
- Distribution protocols
- Consistency protocols

Data Replication



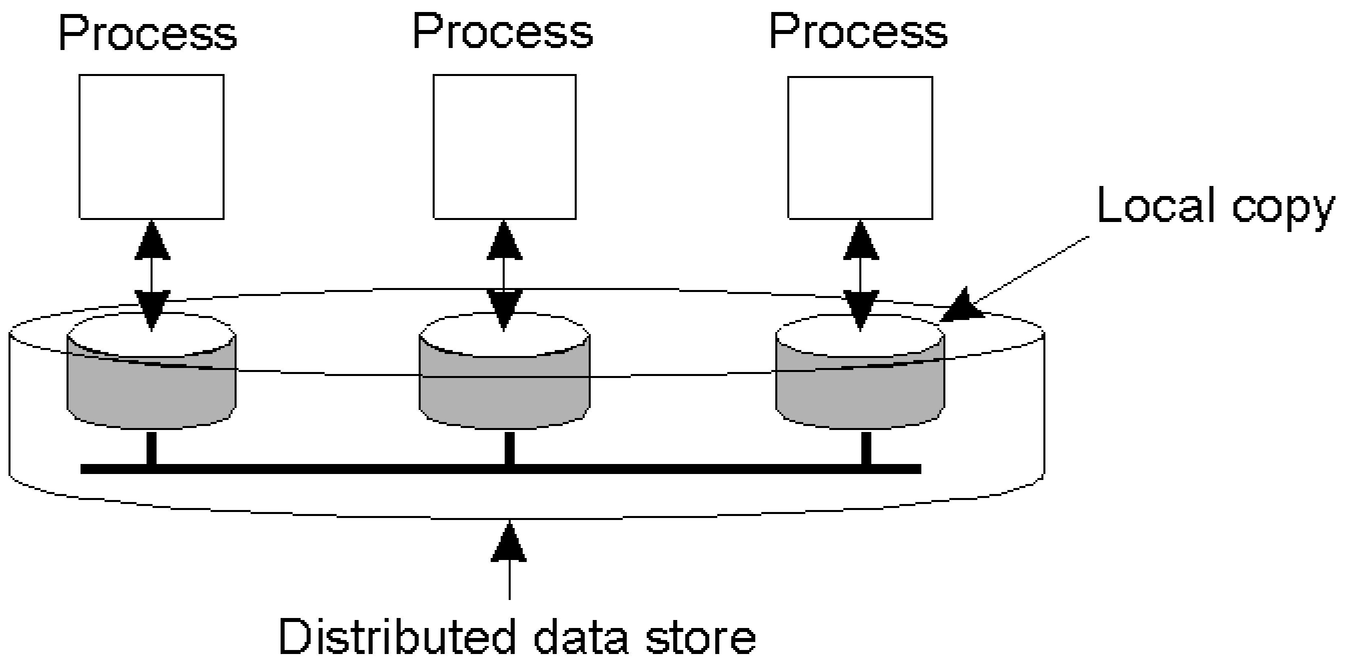
Reasons for Data Replication

- **Dependability requirements**
  - availability
    - at least some server somewhere
    - wireless connections => a local cache
  - reliability (correctness of data)
    - fault tolerance against data corruption
    - fault tolerance against faulty operations
- **Performance**
  - response time, throughput
  - scalability
    - increasing workload
    - geographic expansion
  - mobile workstations => a local cache
- **Price to be paid: consistency maintenance**
  - performance vs. required level of consistency (need not care ⇔ updates immediately visible)

Replication and Scalability

- Requirement: ”tight” consistency
  (an operation at any copy => the same result)
- Difficulties
  - atomic operations (performance, fault tolerance??)
  - timing: when exactly the update is to be performed?
- Solution: consistency requirements vary
  - **always** consistent => **generally** consistent
    (when does it matter? depends on application)
  - => improved performance
- Data-centric / client-centric consistency models

Data-Centric Consistency Models (1)



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

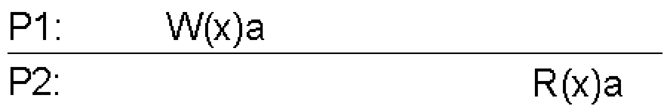
Data-Centric Consistency Models (2)

- Contract between processes and the data store:
  - processes obey the rules
  - the store works correctly
- Normal expectation: a read returns the result of the last write
- Problem: ***which write*** *is the last one?*

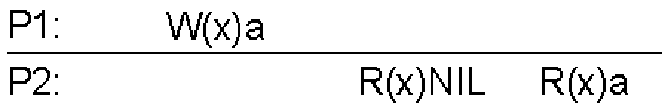
⇒a range of consistency models

Strict Consistency

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



(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.

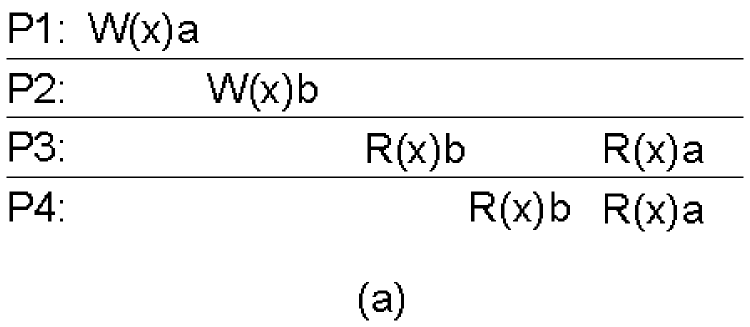
A problem: implementation requires absolute global time.

Another problem: a solution may be physically impossible.

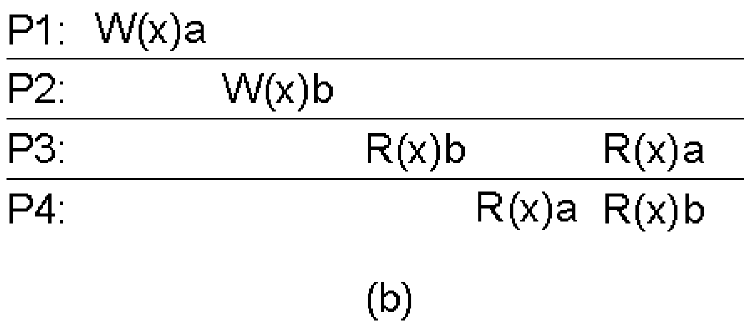
Sequential Consistency

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.

Note: nothing said about time!



A sequentially consistent data store.



A data store that is not sequentially consistent.

Note: a process sees all writes and own reads

Linearizability

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.

Linearizability: primarily used to assist formal verification of concurrent algorithms.

Sequential consistency: widely used, comparable to serializability of transactions (performance??)

Linearizability and Sequential Consistency (1)

Three concurrently executing processes

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

Initial values: x = y = z = 0

All statements are assumed to be indivisible.

Execution sequences

- 6! = 720 possible execution sequences (several of which violate program order)
- 90 valid execution sequences (within each Pi, program order is maintained)

Linearizability and Sequential Consistency (2)

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)

Example: four different execution sequences for the processes, all valid.

The contract:

the process ***must accept all valid*** results as proper answers and ***work correctly*** if **any** of them occurs.

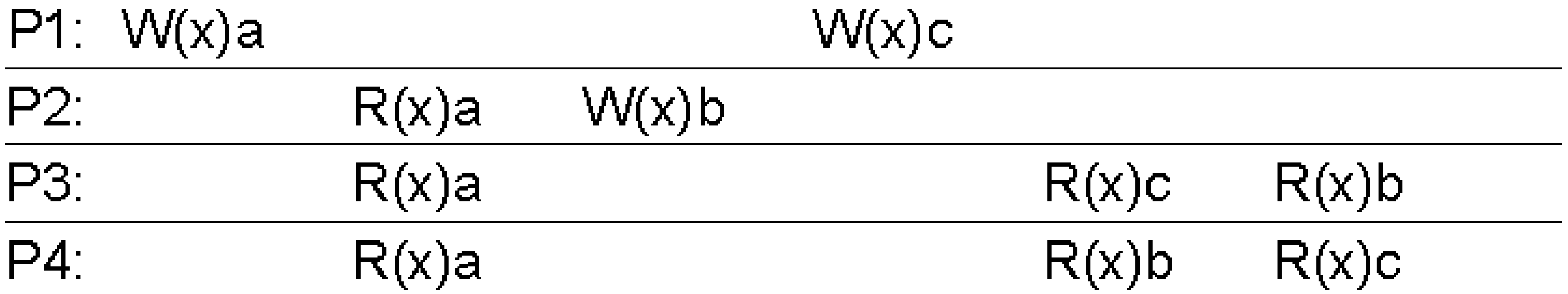
Causal Consistency (1)

Weakened from sequential consistency

Necessary condition:

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

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.

Causal Consistency (3)

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

(a)

A violation of a causally-consistent store.

A correct sequence of events in a causally-consistent store.

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

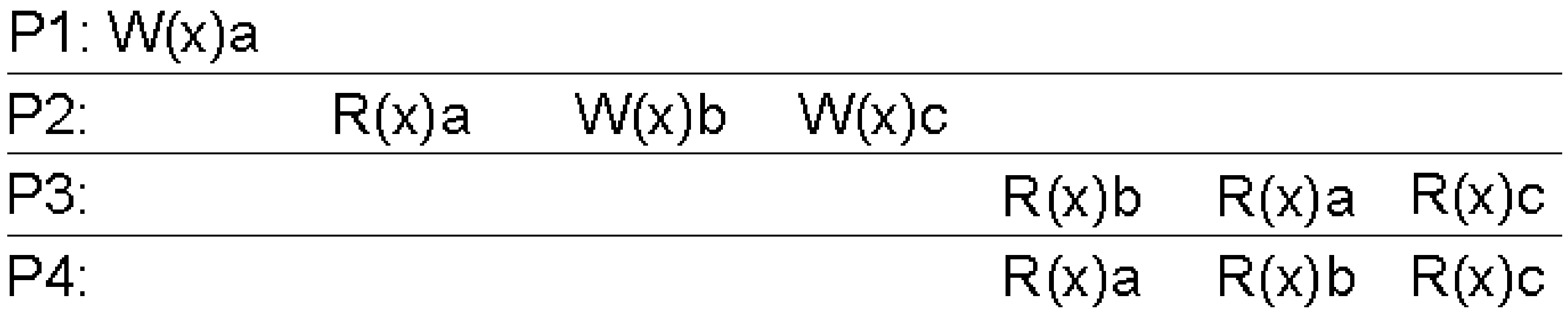
(b)

FIFO Consistency (1)

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.

FIFO Consistency (2)



A valid sequence of events of FIFO consistency

Guarantee:

- writes from a single source must arrive in order
- no other guarantees.

Easy to implement!

Summary of Consistency Models (1)

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes 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 performed. Writes from different processes may not always be seen in the same order by other processes.

Consistency models at the level of read and write operations. Next: grouping operations.

Wanted: Less Restrictive Consistencies

- Needs
  - Tracking single read/writes too fine granularity: sometimes no need to see all writes
  - Example: updates within a critical section *(no need to update replicas when the variables are locked with mutual exclusion)*
- Replicated data and consistency needs
  - Distributed single-user applications exploiting replicas (rare!): consistency needed, but no mutual exclusion
  - Shared data: mutual exclusion **and** consistency obligatory

=> **Implement consistency maintenance around** the grouping for **critical regions**

Shared Data Consistency and Critical Sections

- Assumption: during a critical section the user has access to one replica only
- Aspects of concern
  - Consistency maintenance timing, alternatives:
    - Entry: update the active replica vs.
    - Exit: propagate modifications to other replicas vs.
    - Asynchronous: independent synchronization
  - Control of mutual exclusion:
    - Automatic vs. independent
  - Data of concern:
    - All data vs. selected data

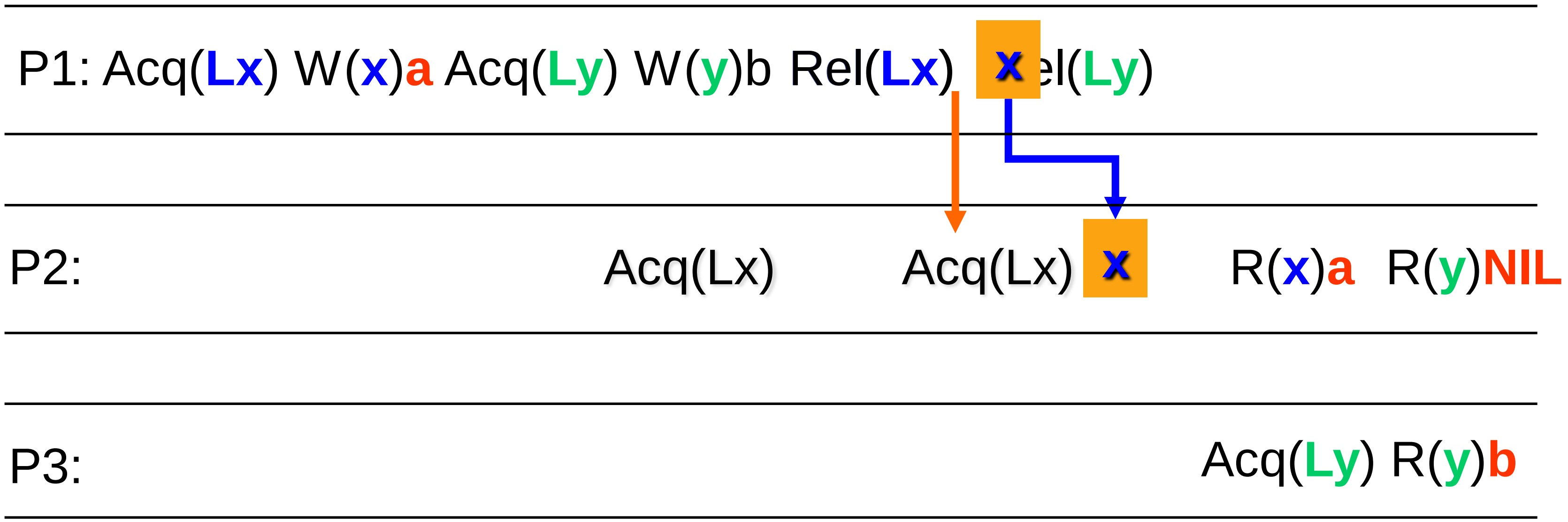
Shared Data Consistency (2)

- Weaker consistency requirements
  - Entry consistency
  - Release consistency
  - Weak consistency
- Implementation method
  - Use a control variable
    - Synchronization / locking
  - Operation
    - Synchronize
    - Lock/unlock and synchronize

Entry Consistency (1)

- Consistency combined with “mutual exclusion”
- **Each** shared data item is associated with a synchronization variable S
- S has a current owner (who has exclusive access to the associated data, which is guaranteed up-to-date)
- Process P enters a critical section: Acquire(S)
  - retrieve the ownership of S
  - the associated variables are made consistent
- Propagation of updates at the next Acquire(S) by some other process

Entry Consistency (2)


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

A valid event sequence for entry consistency.

Summary of Consistency Models (2)

Consistency	Description
Entry	Shared data associated with a synchronization variable are made consistent when a critical section is entered.
Release	All shared data are made consistent after the exit out of the critical section (and up-to-dateness checked upon entry)
Weak	Shared data can be counted on to be consistent only after an explicit synchronization is done

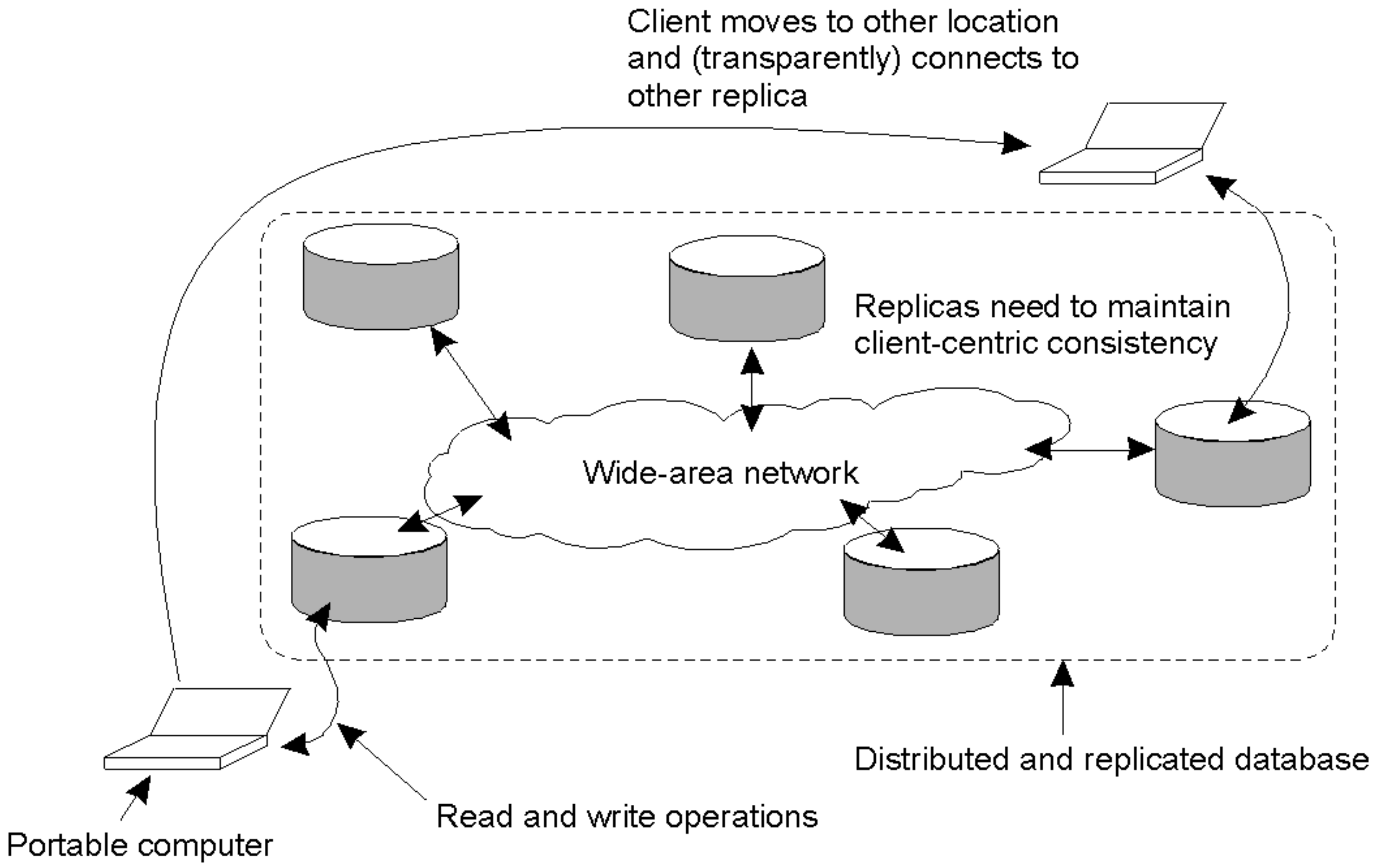
Models built around grouping operations and synchronization.

Eventual and Client-Centric Consistency

- Eventual consistency
  - For when most operations are “read”, few writers (or only one)
    - Simultaneous updates unlikely
  - When a relatively high degree of inconsistency tolerated
    - Examples: DNS, WWW pages
  - Basic idea: changes get propagated eventually, “no sweat!”
    - Easy to implement!

- Problem: a mobile user may access many different replicas…

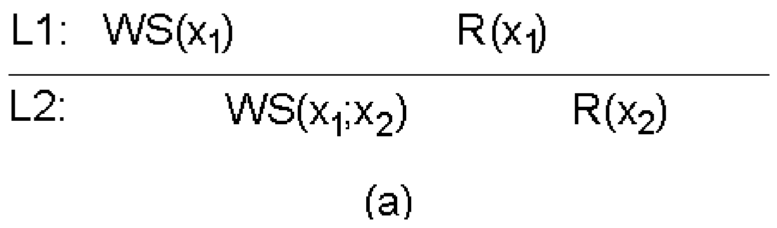
Towards Client-Centric Consistency


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

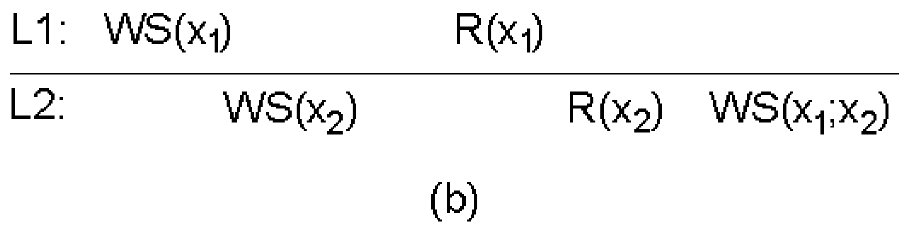
Client-Centric: Monotonic Reads

If a process reads the value of 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.

(Example: e-mail)



A monotonic-read consistent data store

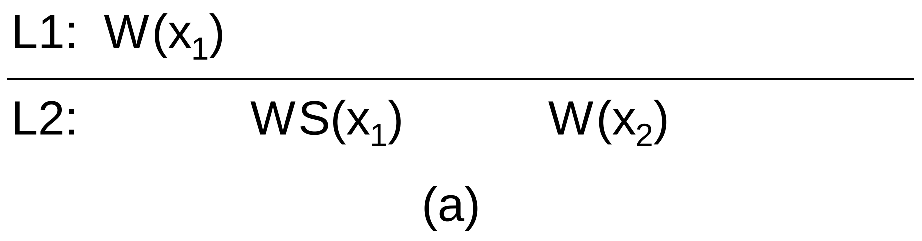


A data store that does not guarantee monotonic reads.

$WS(x_i)$: write set = sequence of operations on x at local-copy L_i

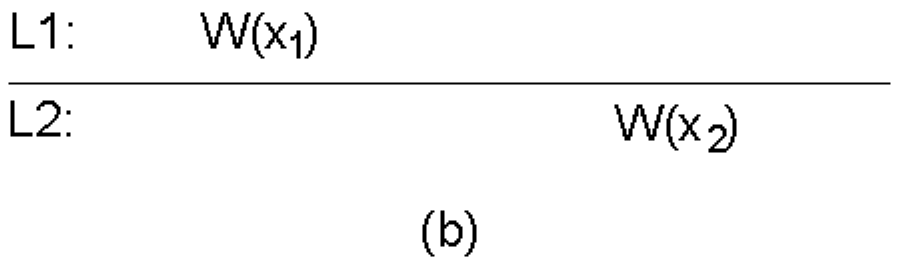
Client-Centric: Monotonic Writes

A write operation by a process on a data item x is completed before any successive write operation on x by the same process. (Example: software updates)



A monotonic-write consistent data store:

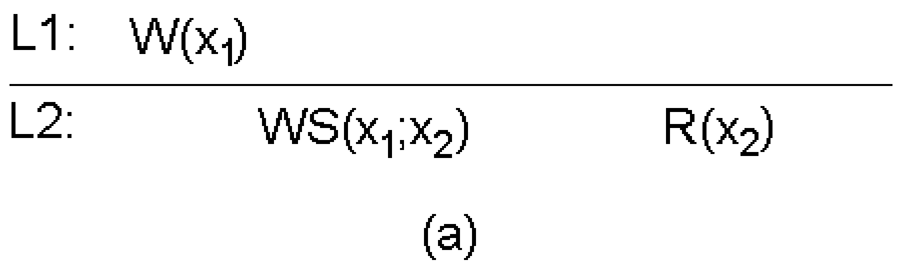
copy L2 syncs to W(x1) before another write



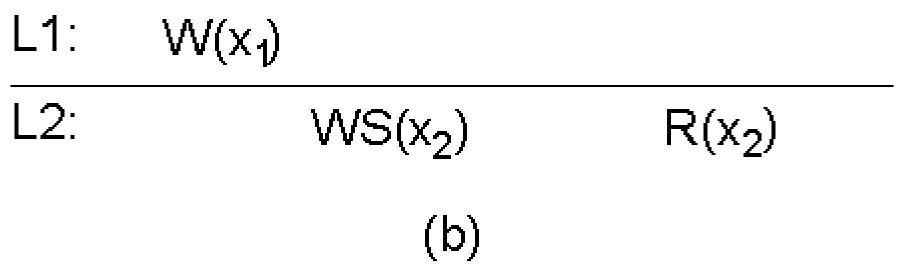
A data store that does not provide monotonic-write consistency.

Client-Centric: 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: edit www-page: L1 is editor, L2 is preview in browser)



A data store that provides read-your-writes consistency.

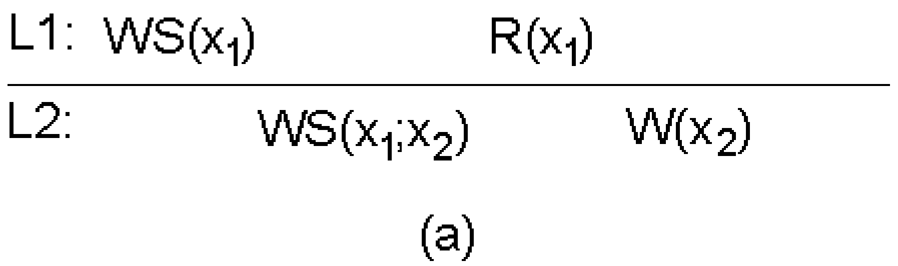


A data store that does not.

(Write set is not up-to-date before next read.)

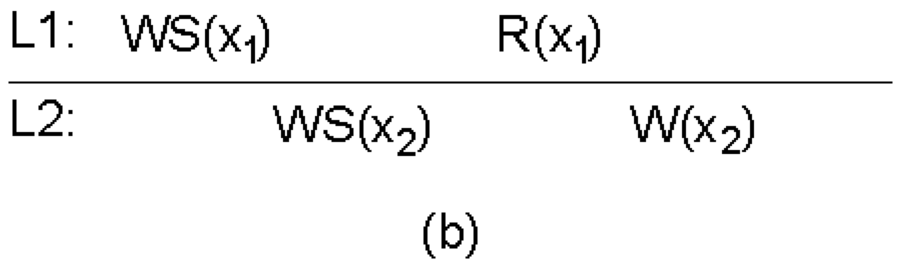
Client-Centric: Writes Follow Reads

Process P: *a write operation (on x) takes place on the same or a more recent value (of x) that was read*. (Example: bulletin board)



(a)

A writes-follow-reads consistent data store



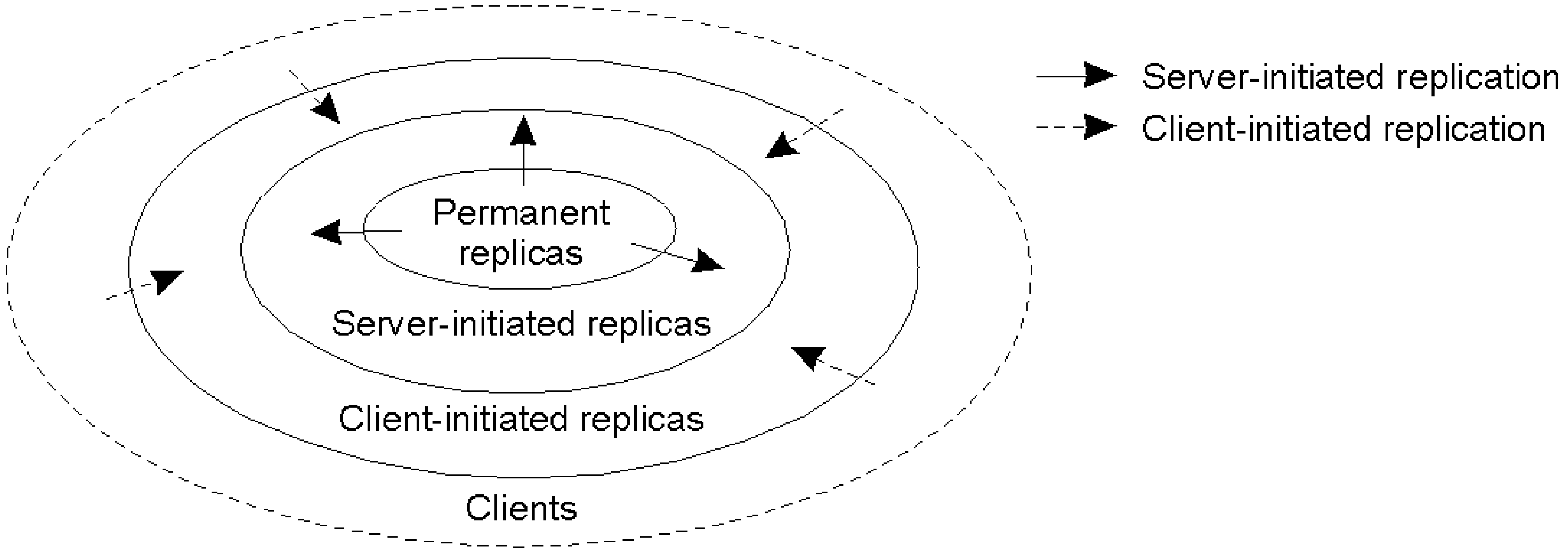
(b)

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

Distribution Protocols

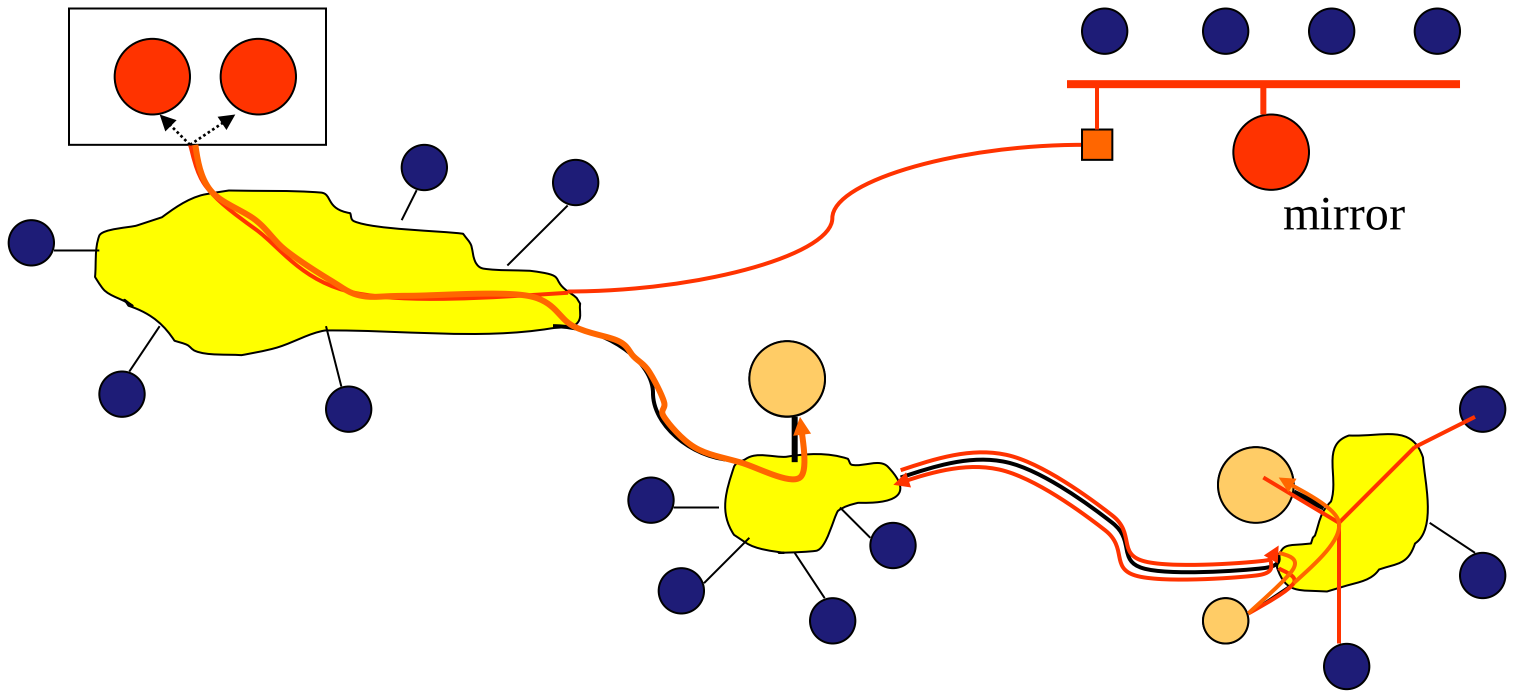
- Replica placement
- Update propagation
- Epidemic protocols

Replica Placement (1)


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.

Replica Placement (2)



● permanent replicas

● server-initiated replicas

● client-initiated replicas

○ servers

○ clients

Permanent Replicas

- *Example: a WWW site*
- The initial set of replicas: constitute a distributed data store
- Organization
  - A replicated server (within one LAN; transparent for the clients)
  - Mirror sites (geographically spread across the Internet; clients choose an appropriate one)

Server-Initiated Replicas (1)

- Created at the initiative of the data store (e.g., for temporary needs)
- Need: to enhance performance
- Also known as **push caches**
- *Example: www hosting services*
  - *a collection of servers available from provider*
  - *provide access to www files from third parties (e.g. a news site)*
  - *replicate files "close to high-demand clients" (e.g. CNN reports on Finnish celebrity cause flood of Finnish connections)*

Server-Initiated Replicas (2)

- Issues:
  - improve response time
  - reduce server load; reduce data communication load

⇒ bring files to servers placed in the proximity of clients

- Where and when should replicas be created/deleted?
- For example:
  - determine two threshold values for each (server, file): **rep** > **del**
  - #[req(S,F)] > rep => create a new replica
  - #[req(S,F)] < del => delete the file (replica)
  - Otherwise: the replica is allowed to be migrated
- Consistency: responsibility of the data store

Client-Initiated Replicas

- Called **client caches**
  - (local storage, temporary need of a copy)
- Managing left entirely to the client
- Placement
  - Typically: the client machine
  - Or a machine shared by several clients
- Consistency: responsibility of client

- More on replication in the Web in slide chapter 6

Update Propagation

- Update route: client writes to copy, who writes to {other copies}
- Whose responsibility – “push” or “pull”?
- Issues:
  - Consistency of copies
  - Cost: traffic, maintenance of state data
- What information is propagated?
  - Notification of an update (**invalidation** protocols)
  - Transfer of data itself or diff (useful if high reads-to-writes ratio)
  - Propagate the update operation: e.g. order_flight(x,y)

    (**active replication**)

Push vs. Pull propagation (1)

- Push
  - A server sends its updates to other replica servers
  - Typically used between permanent and server-initiated replicas
- Pull
  - Client asks for update / validation confirmation
  - Typically used by client caches
    - client to server: {data X, timestamp t_i, OK?}
    - server to client: OK or {data X’, timestamp t_{i+k}}

Push vs. Pull: Tradeoffs (2)

Issue	Push-based	Pull-based
Server state data	Maintain list of client replicas and caches	-
Messages sent	Updates (+ possibly fetch update*)	Polling + updates
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.

*) When just the invalidation notice is pushed, and the update fetched later.

Pull vs. Push: Which to pick in what situation?

- Reads-to-updates ratio
  - High → push (one transfer – many reads)
  - Low → pull (check only when needed)
- Cost vs. Quality of Service (QoS) ratio
  - Factors:
    - Update rate, replica count → maintenance workload
    - Need of consistency (guarantees vs. eventual consistency)
  - Examples
    - (Popular) web pages
    - Arriving flights at the airport
- How failure prone is the data communication
  - Lost push messages → unsuspected use of stale data
  - Pull: failure of validation → known risk of using the data
  - High requirements? → combine push (data) and pull: leases

Pull vs. Push: Available Communication?

- Data communication choices
  - LAN: push & multicasting cheapest, unicasting works for pull
  - Wide-area network: unicasting (multicast not available)
- Consider how failure prone the data communication is:
  - Lost push messages → unsuspected use of stale data
  - Pull: failure of validation → known risk of using the data
- High requirements? → combine push and pull into **leases**

Push + Pull = Leases

- Switch dynamically between pushing and pulling
- A “server promise”: push updates for a certain time
- A lease expires after time t

→ the client

  - Polls the server for new updates or
  - Requests a new lease

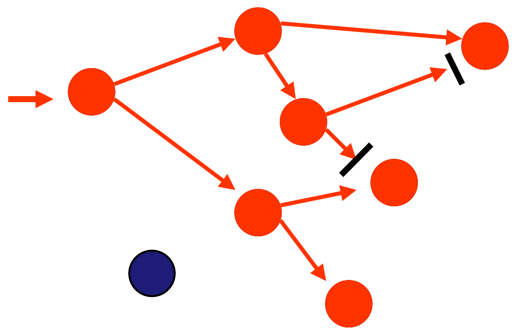
- Different types of leases
  - Age-based: {time to last modification}
  - Renewal-frequency based: long-lasting leases to active users
  - State-space overhead based: too much state to track → lower expiration times for new leases → more “stateless” operation

Information Dissemination Methods

- Example: Epidemic protocols (ch. 4.5)
  - A node with an update: **infective**
  - A node not yet updated: **susceptible**
  - A node not willing / able to spread the update: **removed**
  - **Propagation protocol example**: **anti-entropy**
    - Node **P** picks randomly another node **Q**, and…
    - Three information exchange alternatives:

      P pushes to **Q** or **P** pulls from **Q** or **P**⇔**Q** push-pull
    - Push good early, pull when many infected, push-pull best
  - **Variant of this**: **gossiping**

Gossiping (1)



P starts a gossip round (with a fixed k)

1. P selects random $\{Q_1,..,Q_k\}$ s
2. P sends the update to them
3. P becomes "removed"

Q_i receives a gossip update

If Q_i was susceptible, it starts a gossip round
else Q_i ignores the update

The textbook's variant *(for an infective P)*

```
P: do until removed
    {select a random $Q_i$ ;  send the update to $Q_i$ ;
     if $Q_i$ was infected then remove P with probability 1/k }
```

Gossiping (2)

- Coverage of our variant: depends on k *(fanout)*
  - A large fanout: good coverage, big overhead
  - A small fanout: the gossip (epidemic) dies out too soon
  - n: number of nodes, m: parameter (fixed value)

    $k = \log(n)+m \Rightarrow$

    $P\{\text{every node receives}\} = e^{(-e^{(-k)})}$

    (esim: k=2 => P=0.87; k=5 => P=0.99)
- Merits
  - Scalability, decentralized operation
  - Reliability, robustness, fault tolerance
  - No feedback implosion, no need for routing tables

Epidemic Protocols: Removing Data

The problem

1. Server P deletes data D => all information on D is destroyed

   [*server Q has not yet deleted D*]

2. Communication P ⇔Q => P receives D (as new data)

A solution: deletion is a special update *(death certificate)*

- Allows normal update communication
- A new problem: cleaning up of death certificates…
- Solution: time-to-live (TTL) for the certificate
  - After TTL elapsed: a normal server deletes the certificate
  - Some special servers maintain the historical certificates forever *(for what purpose?)*

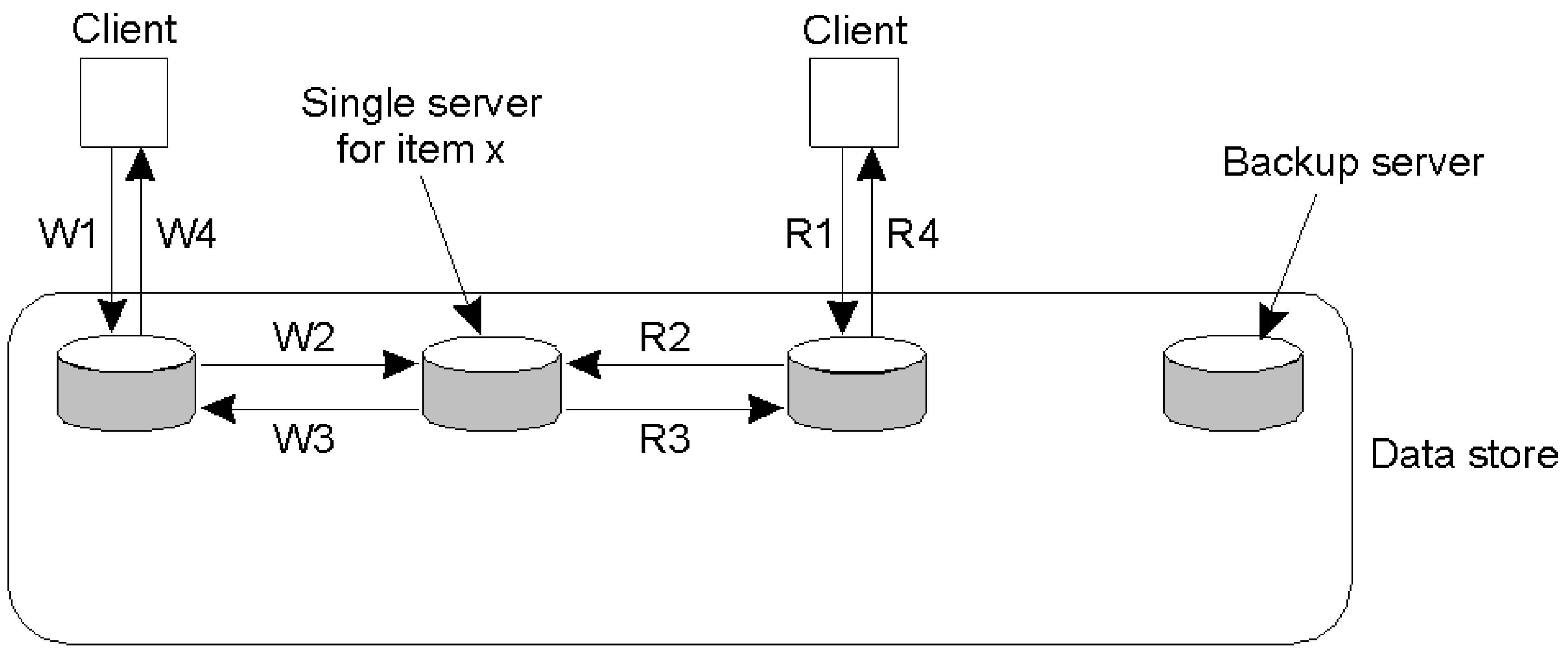
Consistency Protocols

- Consistency protocol: implementation of a consistency model
- The most widely applied models
  - Sequential consistency
  - Weak consistency with synchronization variables
  - Atomic transactions
- The main approaches
  - Primary-based protocols (remote write, local write)
  - Replicated-write protocols (active replication, quorum based)
  - (Cache-coherence protocols)

One more model: Atomic transactions

- Transaction: collection of operations, forms a logical unit for consistency and recovery
- Example: order a flight ticket, withdraw money from account
- Initialize, read and/or modify objects, commit
  - Or abort and roll back changes
- The ACID properties for transactions:
  - Atomic: either “nothing happened” or “all done” (for outsiders)
  - Consistent: takes system from one valid state to another
  - Isolated: transactions do not affect each other; serializability
  - Durable: after a commit, the state will survive even crashes etc

Primary-Based Protocols: Remote-Write Protocols (1)

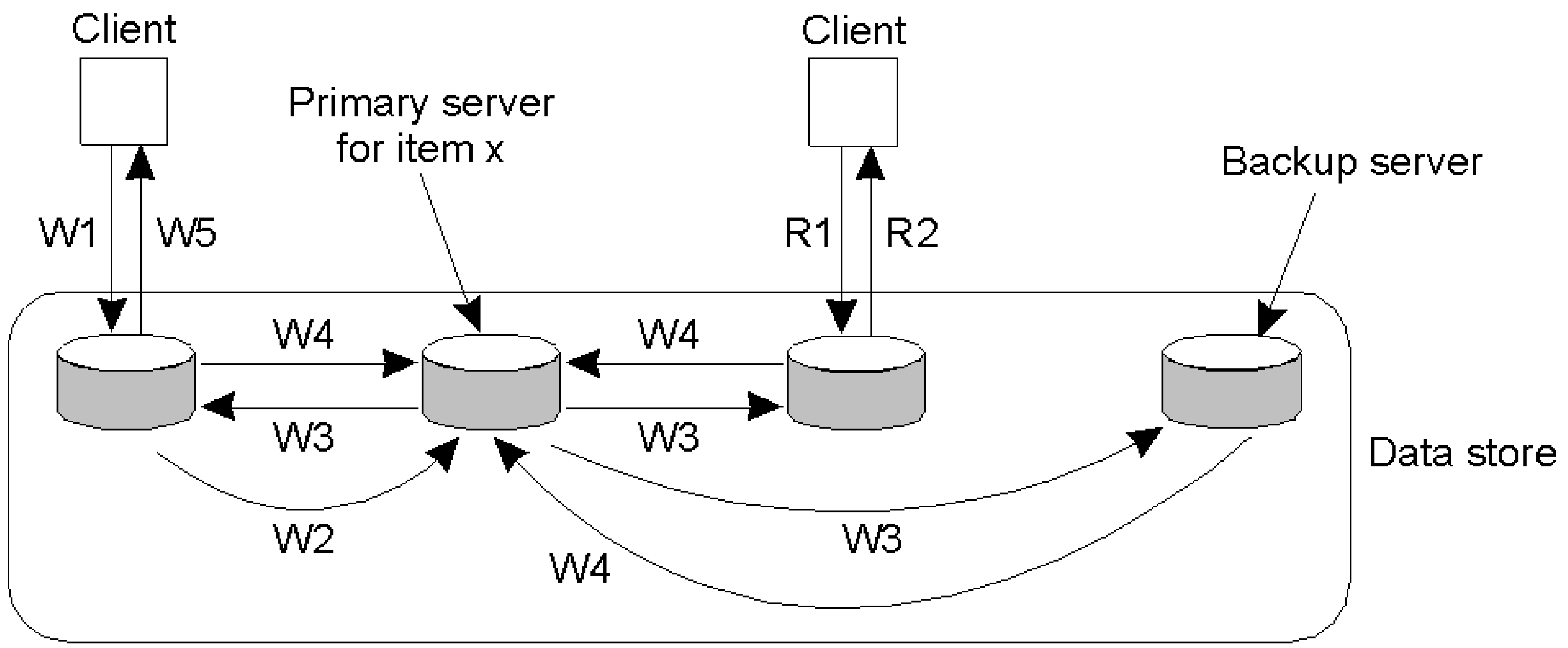


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

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.

Primary-Based Protocols: Remote-Write Protocols (2)



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

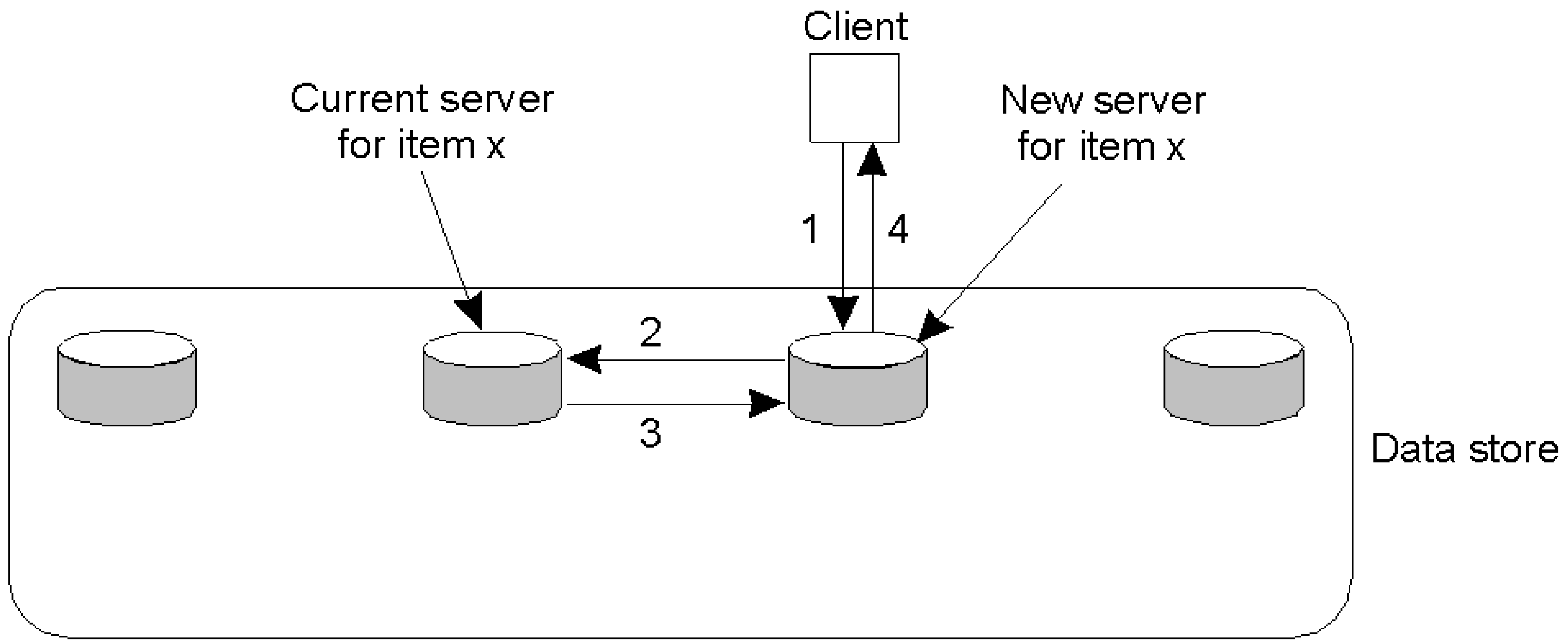
R1. Read request
R2. Response to read

Thus implements:

Sequential consistency
Read Your Writes

Primary-backup protocol.

Primary-Based Protocols: Local-Write Protocols (1)

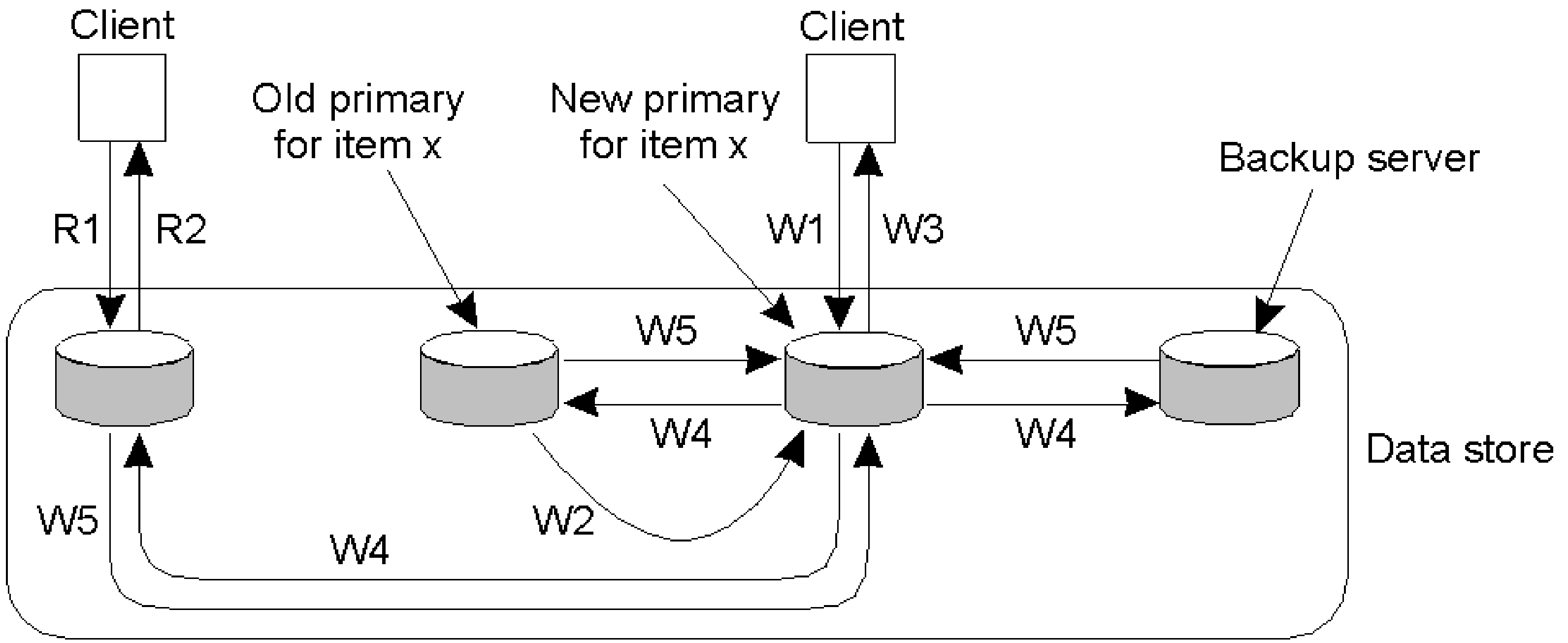


1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

Mobile workstations!

Name service overhead!

Primary-based local-write protocol in which a single copy is migrated between processes.



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

Example: Mobile PC <= primary server for items to be needed

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

Replicated-Write Protocols: Quorum-Based Protocols: "Let's Vote!"

- Write operations can be carried out at **multiple** replicas
  - E.g. active replication: forward an update operation to all replicas
  - Requires totally-ordered multicast (or a sequencer – similar to a primary)
- Could guarantee consistency by making updates transactions, but…
  - Performance?
  - Sensitivity for availability (all or *nothing*) ?
- Solution: Quorum-based protocols
  - A **subgroup of available** replicas **is allowed** to update data
- A **quorum** is a group which is large enough for the operation.
- Problem in a partitioned network:
  - If the groups cannot communicate, each group must decide independently whether it is allowed to carry out operations

Quorum-Based Voting

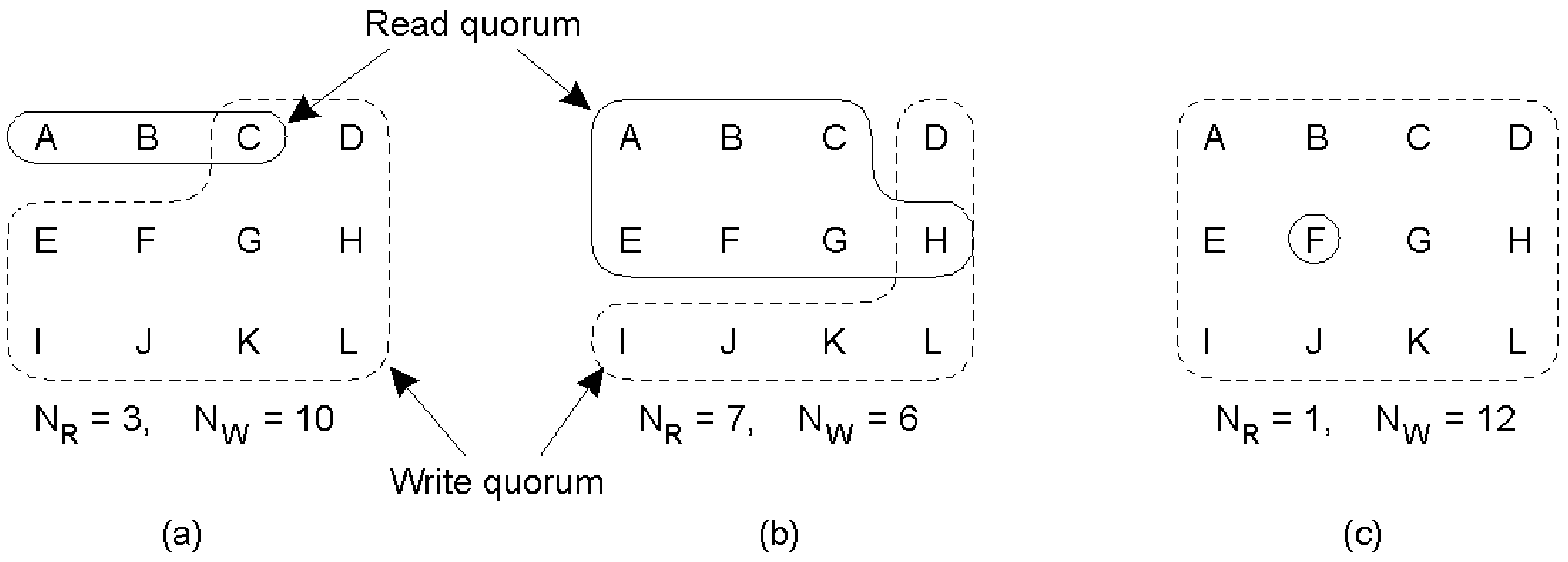
Read

- Collect a read quorum
- Read from any up-to-date replica (the newest timestamp)

Write

- Collect a write quorum
- If there are insufficient up-to-date replicas, replace non-current replicas with current replicas *(consider: why?)*
- Update all replicas belonging to the write quorum.

Notice: each replica may have a different number of votes assigned to it.



Three voting-case examples:

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)

The constraints:

1. $N_R + N_W > N$
2. $N_W > N/2$

Quorum Methods Applied

- Possibilities for various levels of “reliability”
  - Guaranteed up-to-date: collect a full quorum
  - Limited guarantee: insufficient quora allowed for reads
  - Best effort
    - Read without a quorum
    - Write without a quorum - if consistency checks available!
- Transactions involving replicated data
  - Collect a quorum of locks
  - Problem: a voting processes meets another ongoing voting
    - Alternative decisions: abort wait continue without a vote
    - Problem: a case of distributed decision making

      (figure out a solution)

Chapter Summary

- Replication
- Consistency models
- Distribution protocols
- Consistency protocols