HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

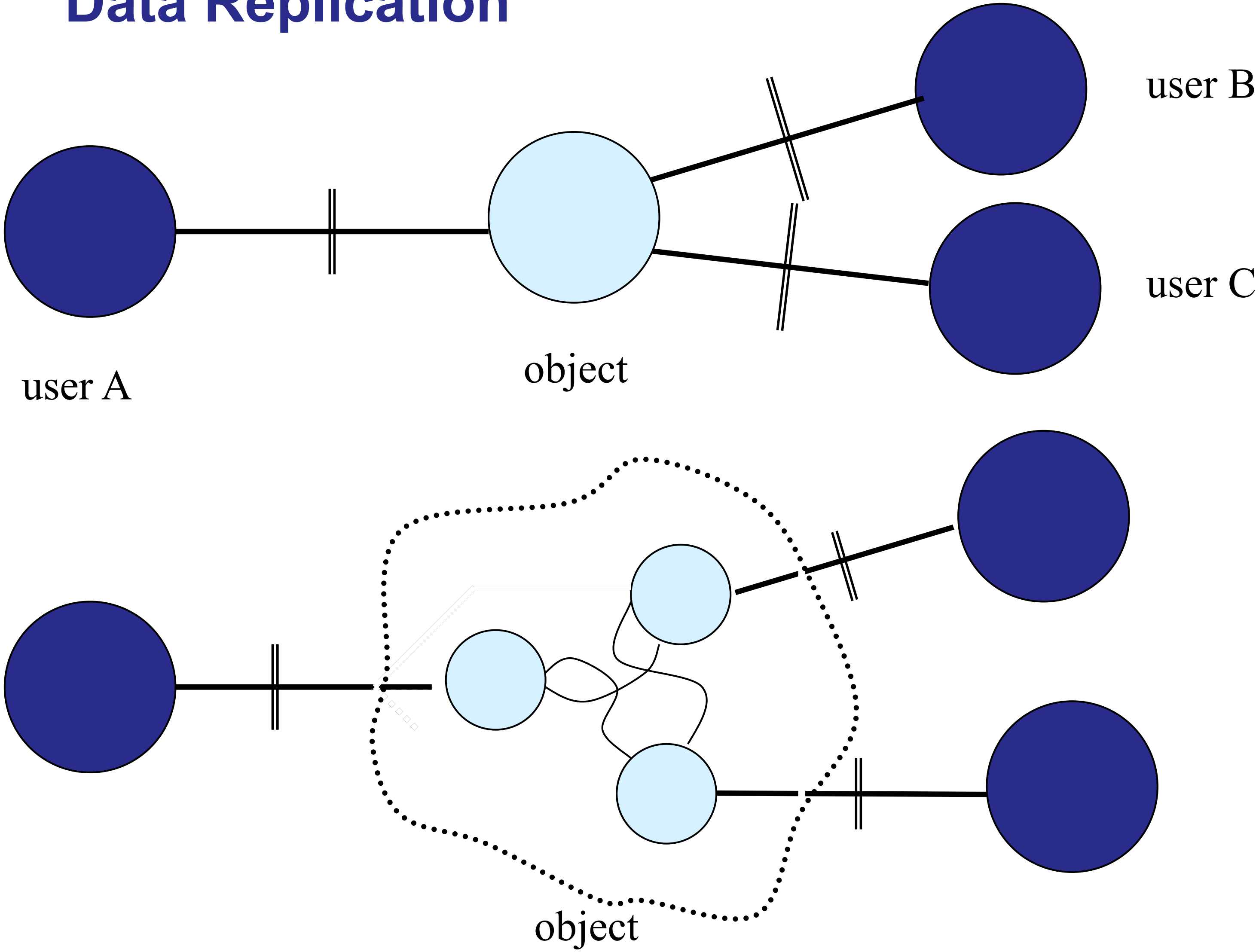
Replication and Consistency

Fall 2008
Jussi Kangasharju

Chapter Outline

- Replication
- Consistency models
- Distribution protocols
- Consistency protocols

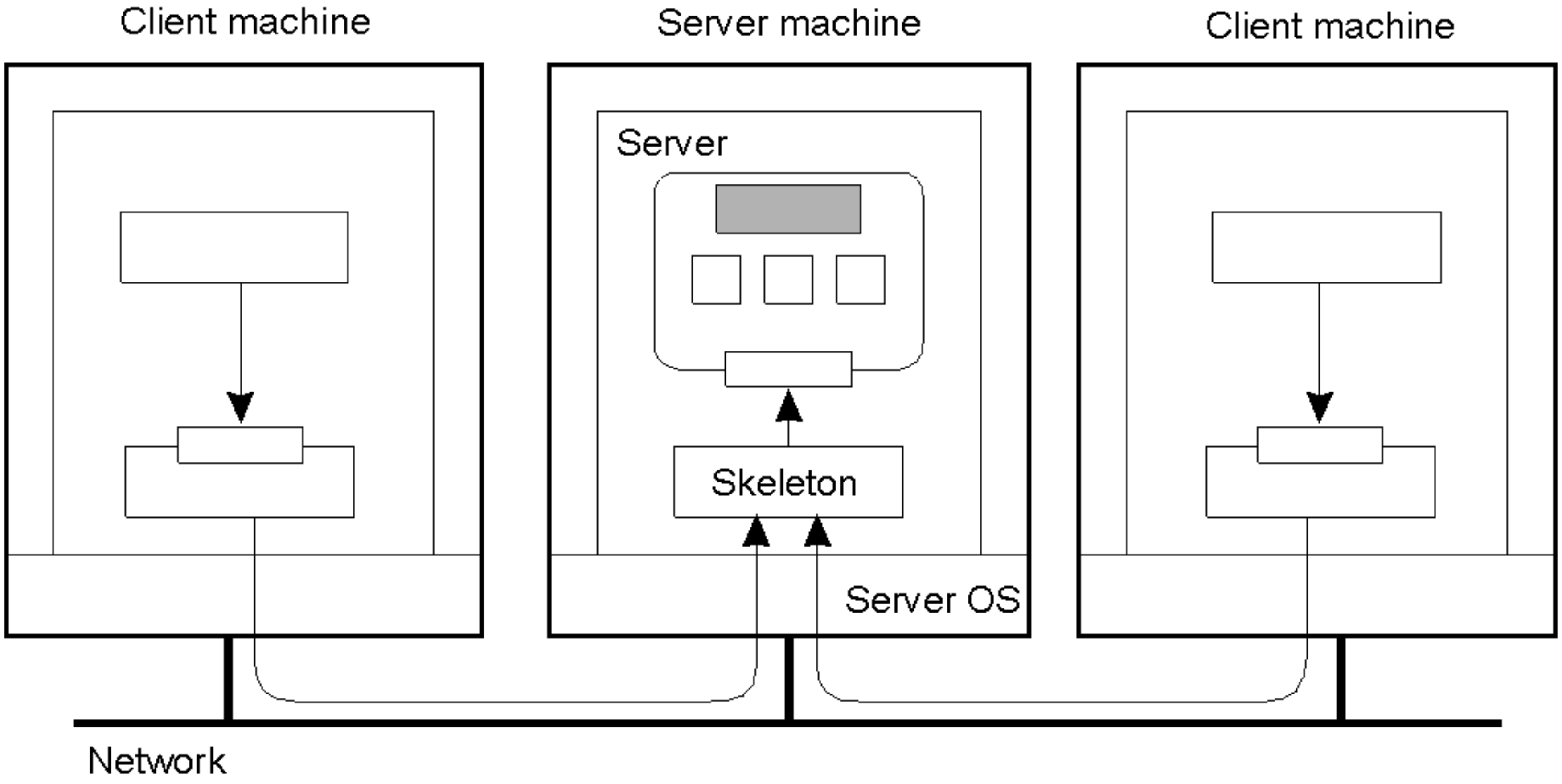
Data Replication



Reasons for Data Replication

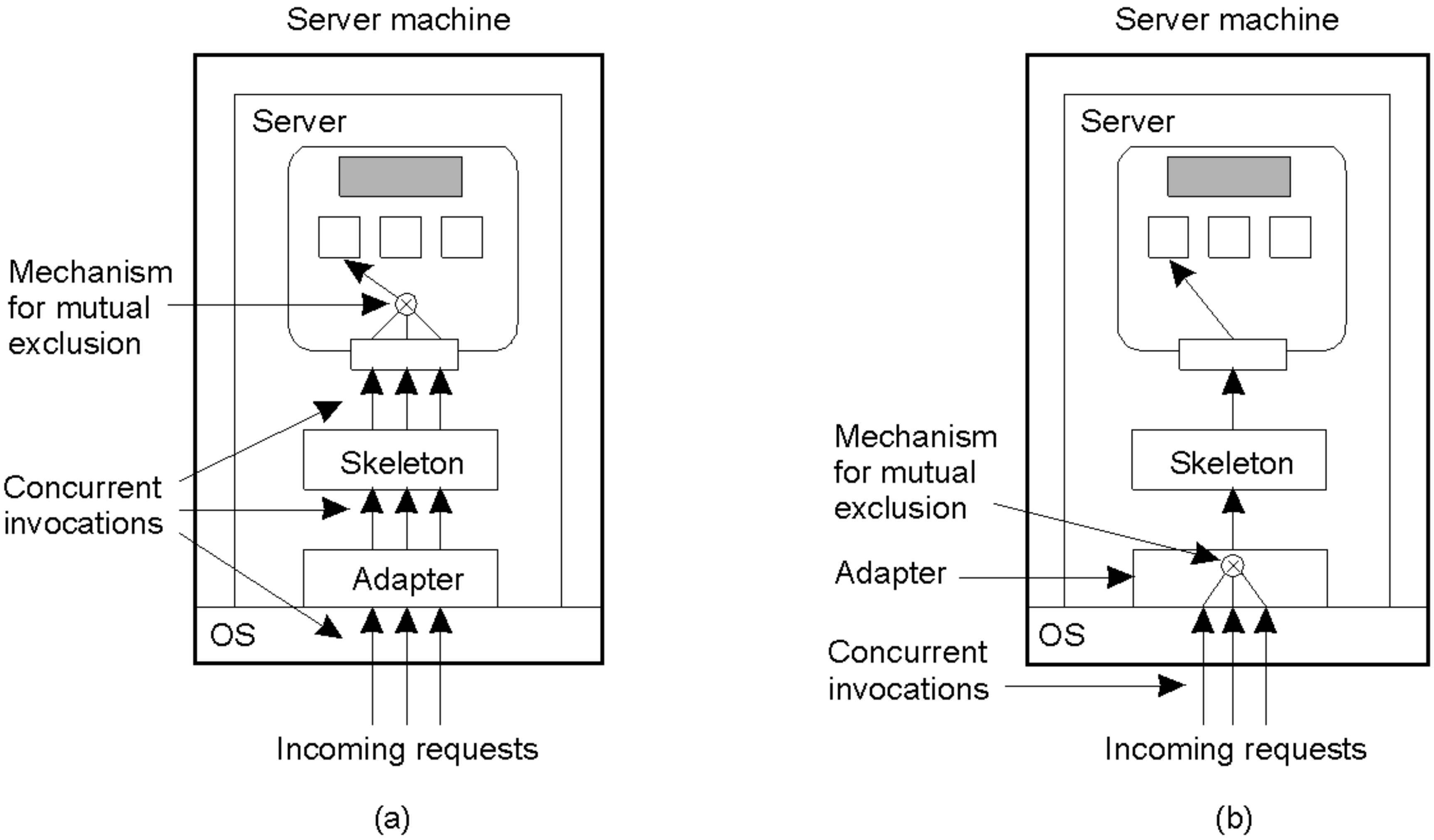
- **Dependability requirements**
  - availability
    - at least some server somewhere
    - wireless connections => a local cache
  - reliability (correctness of data)
    - fault tolerance against data corruption
    - fault tolerance against faulty operations
- **Performance**
  - response time, throughput
  - 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)

Object Replication (1)



Organization of a distributed remote object shared by two different clients (consistency at the level of critical phases).

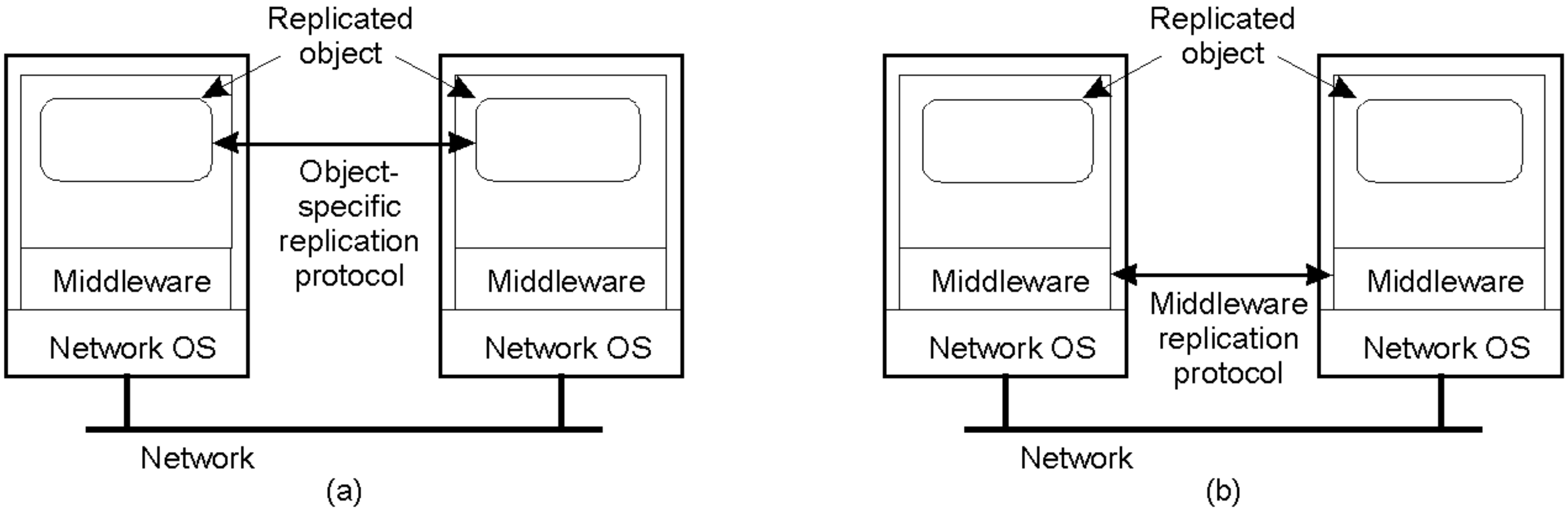
Object Replication (2)



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

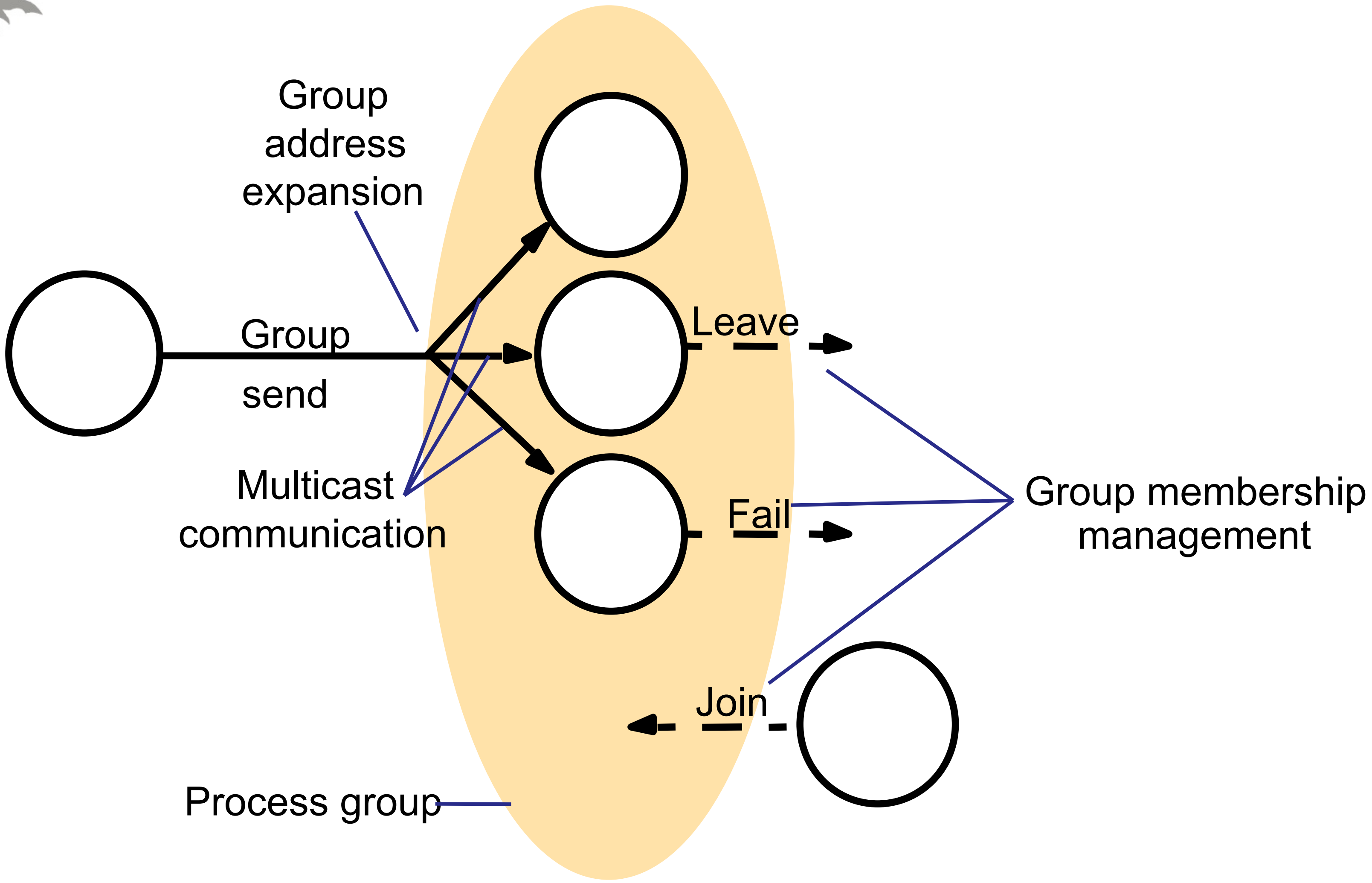
Object Replication (3)



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

b) A distributed system responsible for replica management

Services Provided for Process Groups



CoDoKi, Figure 14.2

A Basic Architectural Model for the Management of Replicated Data

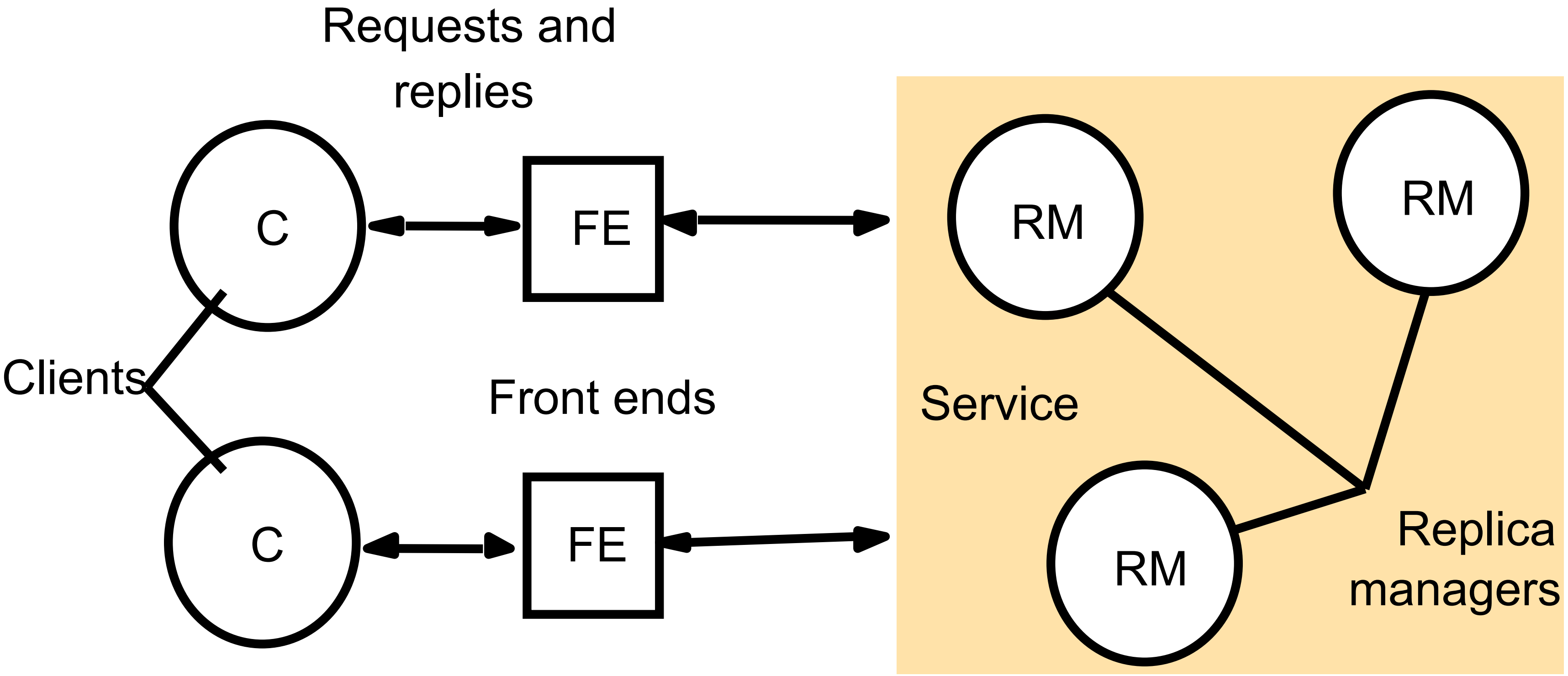


Figure 14.1

The Passive (primary-backup) Model for Fault Tolerance

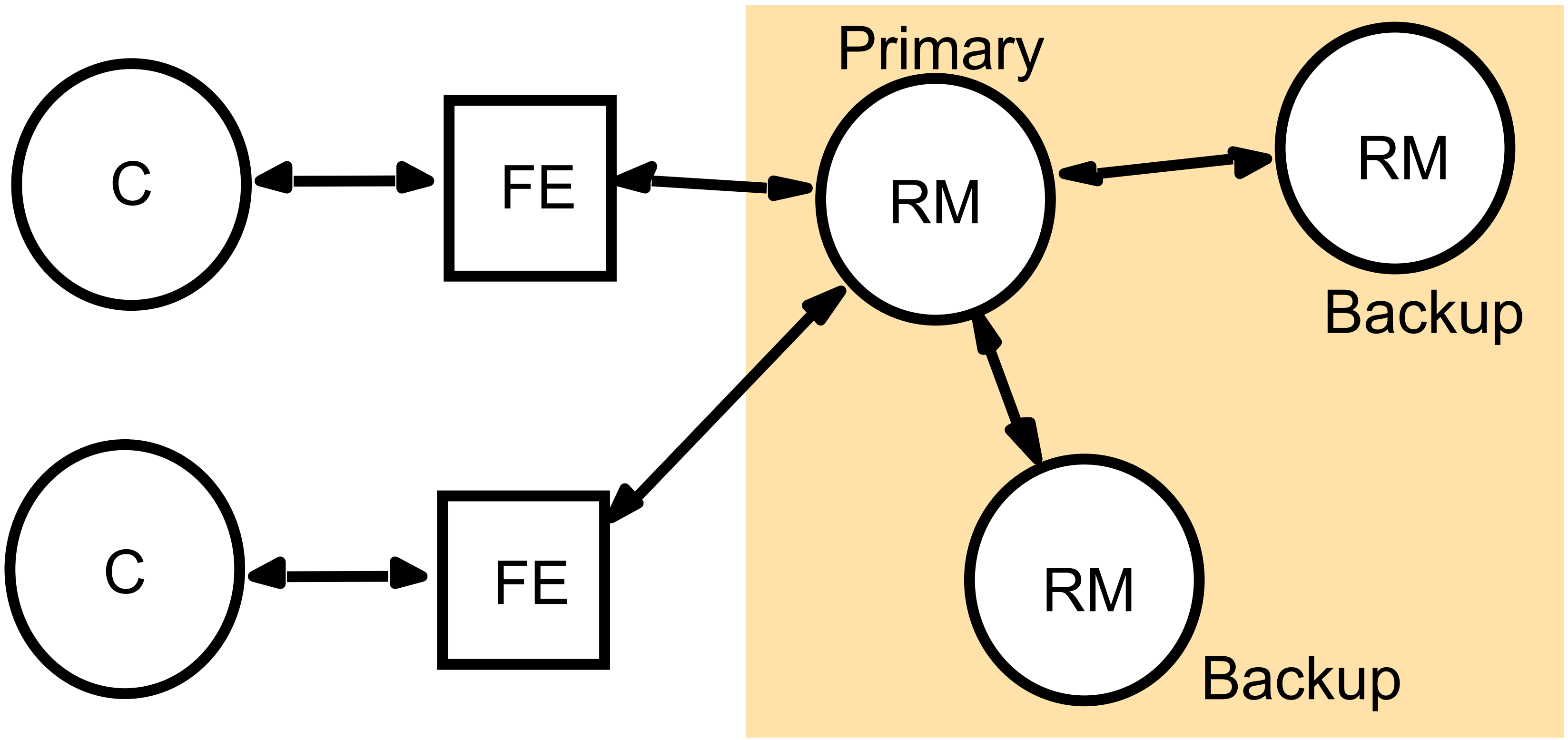


Figure 14.4

Active Replication

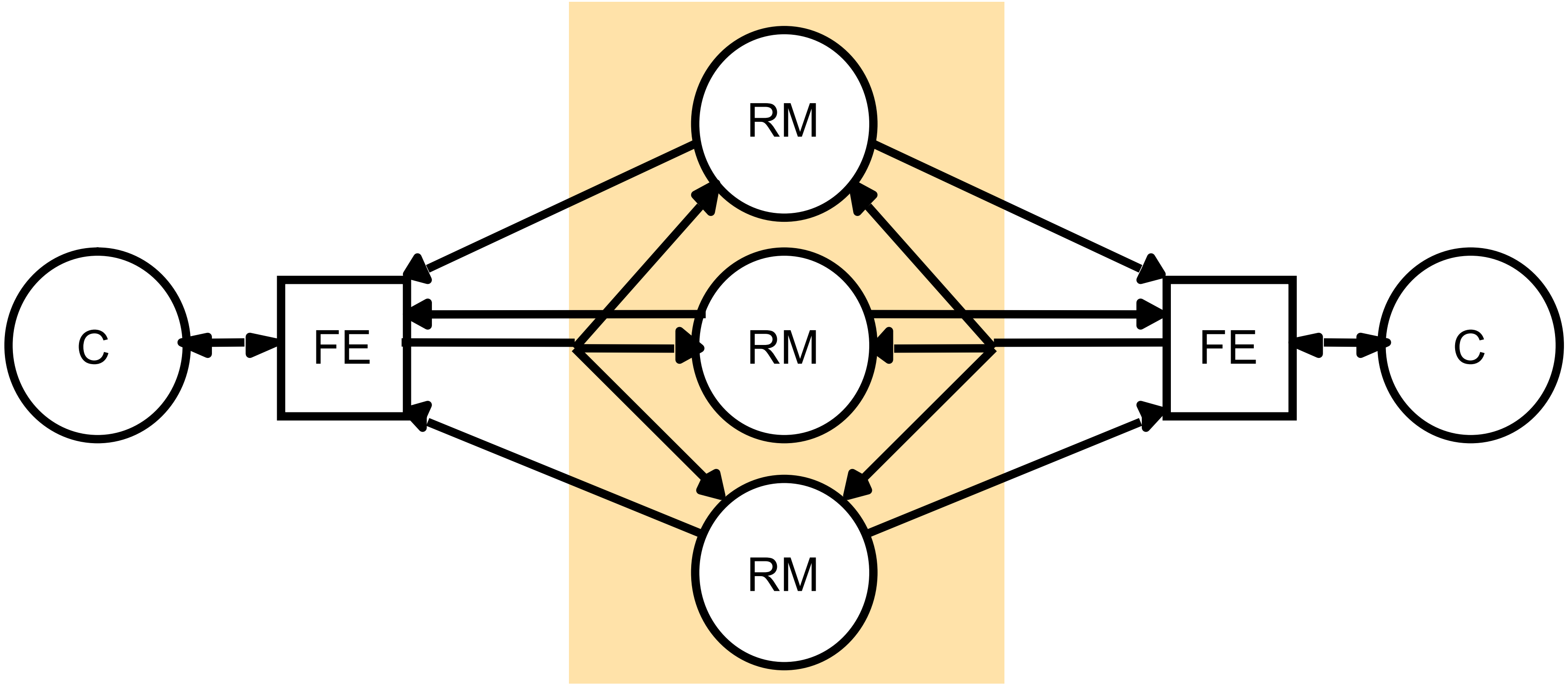
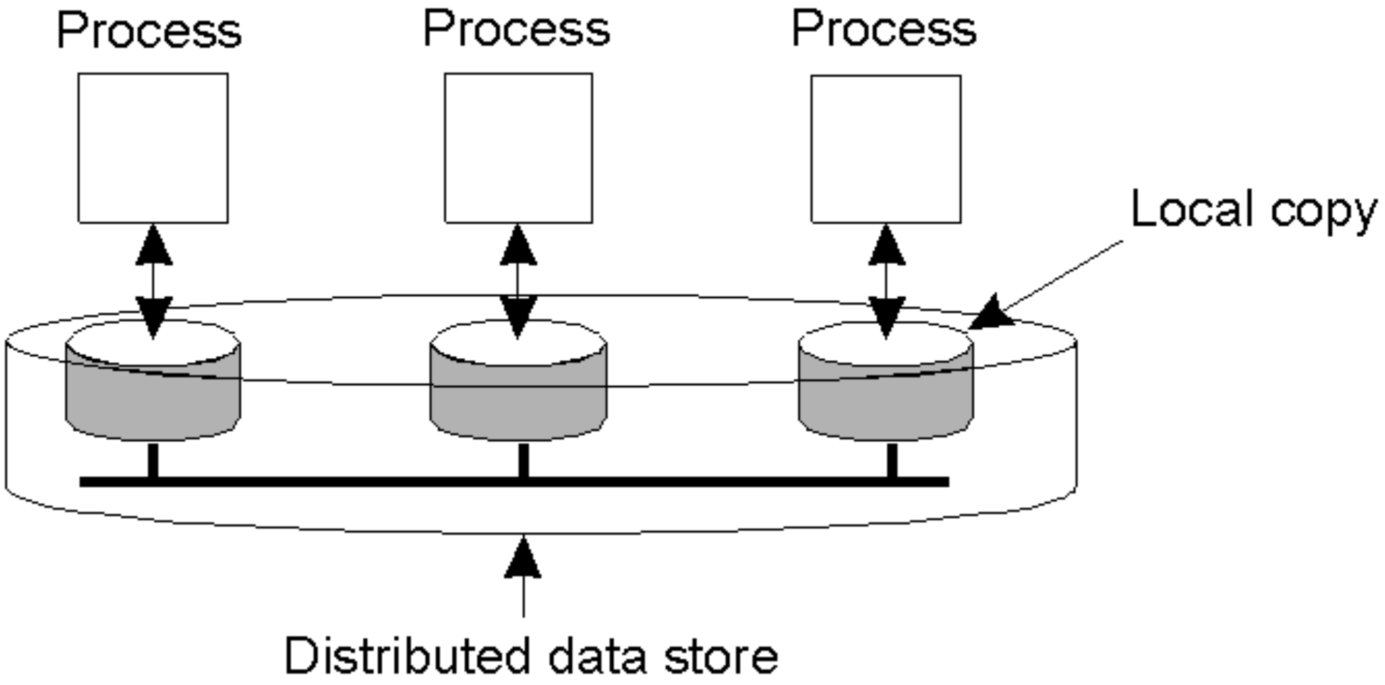


Figure 14.5

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)

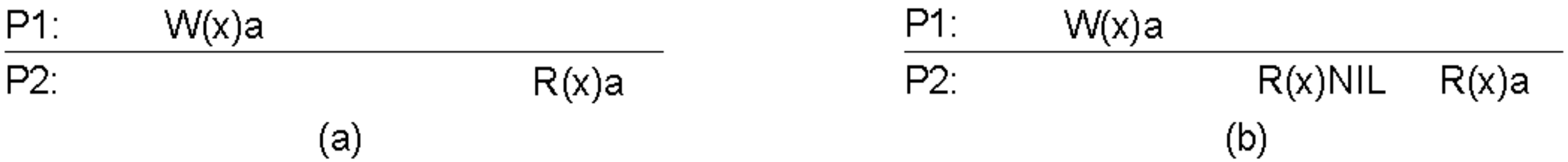
- Contract between processes and the data store:
  - processes obey the rules
  - the store works correctly

- 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.



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

a) A strictly consistent store.
b) A store that is not strictly consistent.

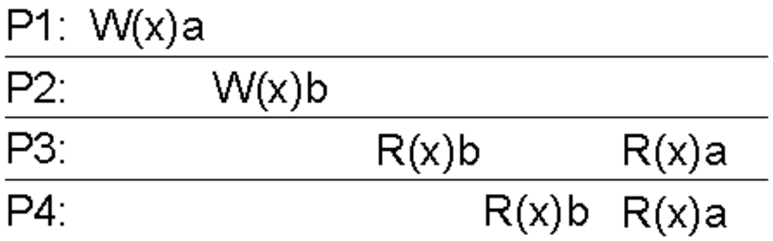
A problem: implementation requires absolute global time.

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!



P1: W(x)a

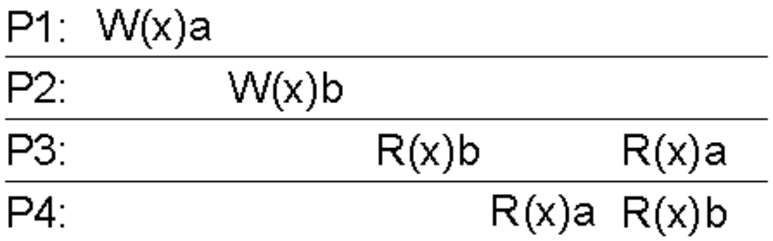
P2: W(x)b

P3: R(x)b R(x)a

P4: R(x)b R(x)a

(a)

A sequentially consistent data store.



P1: W(x)a

P2: W(x)b

P3: R(x)b R(x)a

P4: R(x)a R(x)b

(b)

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

- 720 possible execution sequences (several of which violate program order)
- 90 valid execution sequences

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)

Four valid execution sequences for the processes.

The contract:

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

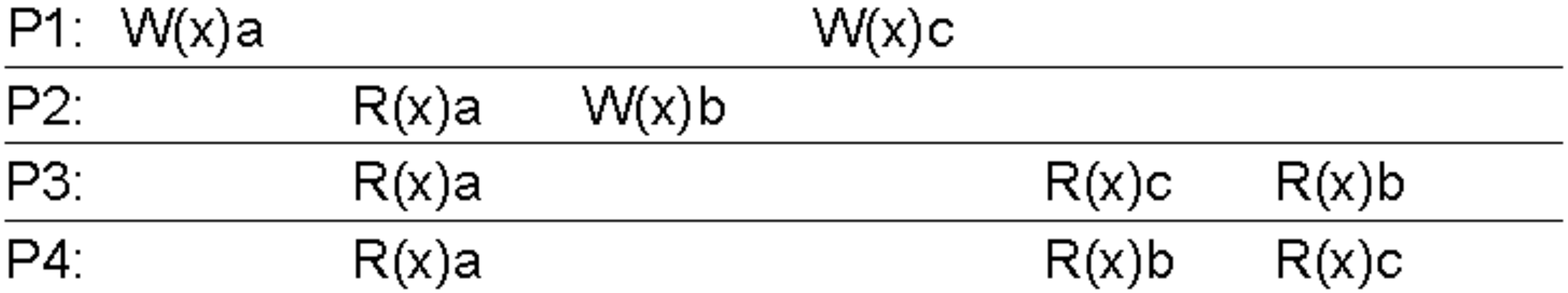
Causal Consistency (1)

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.

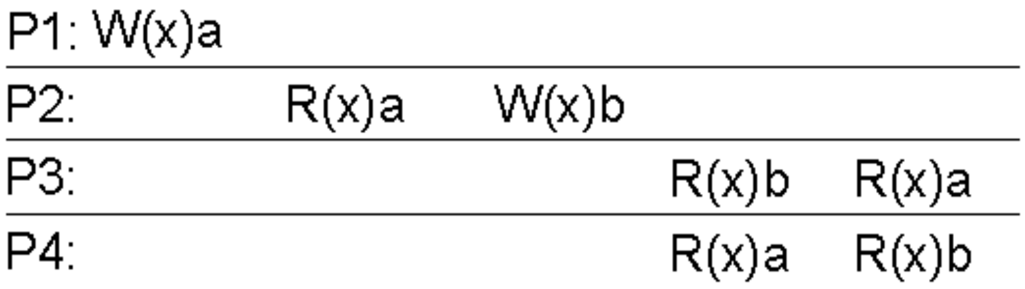
Causal Consistency (2)



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.

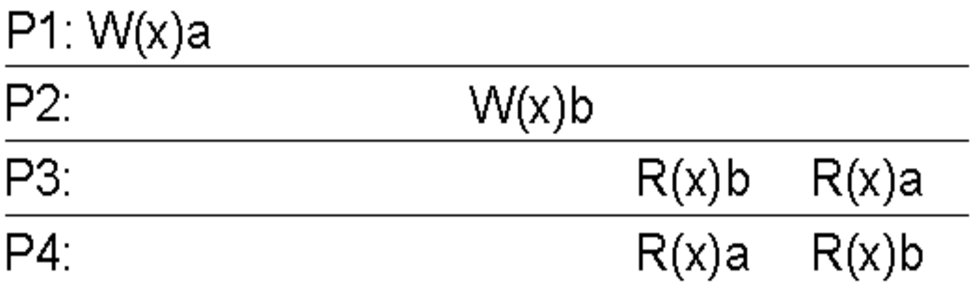
Causal Consistency (3)



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.



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

(b)

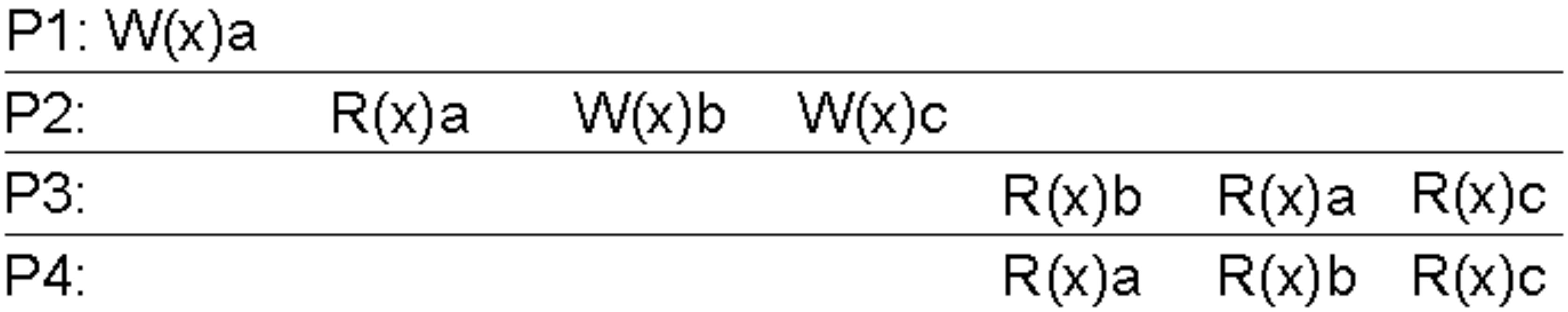
A correct sequence of events in a causally-consistent store.

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)



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

Guarantee:

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

Easy to implement!

FIFO Consistency (3)

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**
(P1)	(P2)	(P3))

Statement execution as seen by the three processes from a previous slide.
The statements in bold are the ones that generate the output shown.

FIFO Consistency (4)

Sequential consistency vs. FIFO consistency

- both: the order of execution is nondeterministic
- sequential: the processes agree what it is
- FIFO: the processes need not agree

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

assume: initially x = y = 0

possible outcomes: P1 or P2 or neither is killed

FIFO: also possible that both are killed

Less Restrictive Consistencies

- Needs
  - FIFO too restrictive: sometimes no need to see all writes
  - example: updates within a critical section *(the variables are locked => replicas need not be updated -- but the database does not know it)*
- Replicated data and consistency needs
  - single user: data-centric consistency needed at all?
    - in a distributed (single-user) application: yes!
    - but distributed single-user applications exploiting replicas are not very common …
  - shared data: mutual exclusion **and** consistency obligatory

=> **combine consistency maintenance with** the implementation of **critical regions**

Consistency of Shared Data (1)

- 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
    - exit: propagate modifications to other replicas
    - asynchronous: independent synchronization
  - control of mutual exclusion:
    - automatic, independent
  - data of concern:
    - all data, selected data

Consistency of Shared Data (2)

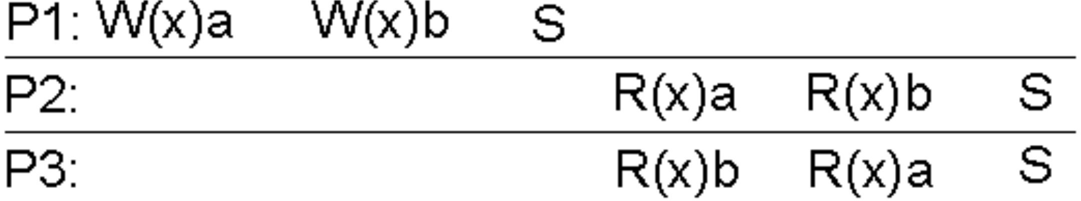
- Weaker consistency requirements
  - Weak consistency
  - Release consistency
  - Entry consistency

- Implementation method
  - control variable
    - synchronization / locking
  - operation
    - synchronize
    - lock/unlock and synchronize

Weak Consistency (1)

- Synchronization independent of “mutual exclusion”
- All data is synchronized
- Implementation
  - synchronization variable S
  - operation synchronize
  - synchronize(S):
    - all local writes by P are propagated to other copies
    - writes by other processes are brought into P’s copy

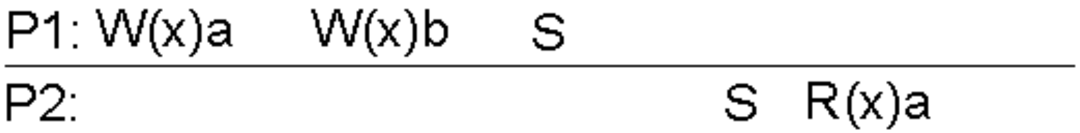
Weak Consistency (2)



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

(a)

A valid sequence of events for weak consistency.

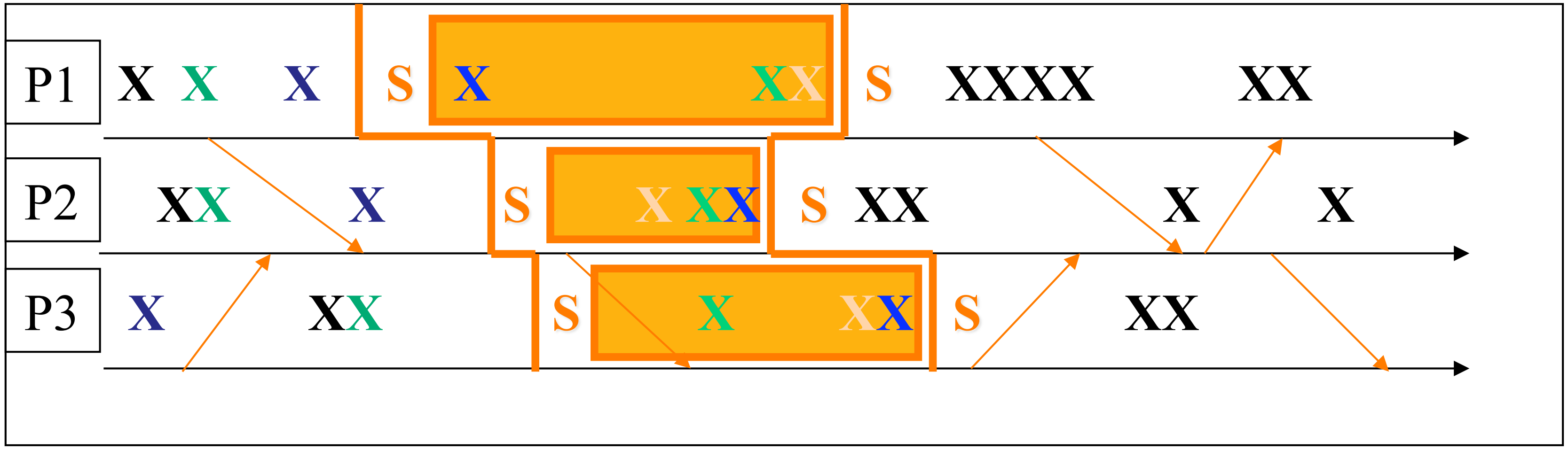


P1:	W(x)a	W(x)b	S		
P2:				S	R(x)a

(b)

An invalid sequence for weak consistency.

Weak Consistency (4)



- Weak consistency enforces consistency of a group of operations, not on individual reads and writes

- Sequential consistency is enforced between **groups** of operations

- Compare with: distributed snapshot

Weak Consistency (3)

Properties:

1. **Accesses to synchronization variables** associated with a data store are **sequentially consistent** *(synchronizations are seen in the same order)*
2. **No operation on a synchronization variable** is allowed to be performed **until all previous writes have been completed** everywhere
3. **No read or write operation** on data items are allowed to be performed **until all previous operations to synchronization variables** have been performed.

Release Consistency (1)

- Consistency synchronized with “mutual exclusion”

=> fewer consistency requirements needed

  - enter: only local data must be up-to-date
  - exit: writes need not be propagated until at exit
  - only protected data is made consistent

- Implementation
  - “lock” variables associated with data items
  - operations acquire(Lock) and release(Lock)
  - implementation of acq/rel application dependent:
    lock <=> data associations are application specific
    (this functionality could be supported by middleware)

Release Consistency (2)

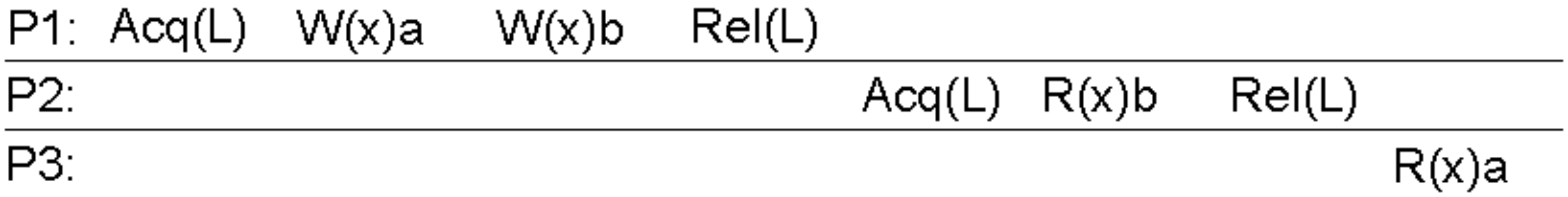
Synchronization: enter or exit a critical section

- enter => bring all local copies up to date

  (but even previous local changes can be sent later to others)

- exit => propagate changes to others

  (but changes in other copies can be imported later)



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.

Release Consistency (3)

Rules:

- Synchronization (mutual ordering) of
  - acquire/release operations

  wrt.
  - read/write operations

  see: weak consistency
- Accesses to synchronization variables are FIFO consistent

  (sequential consistency is not required).

The **lazy** version

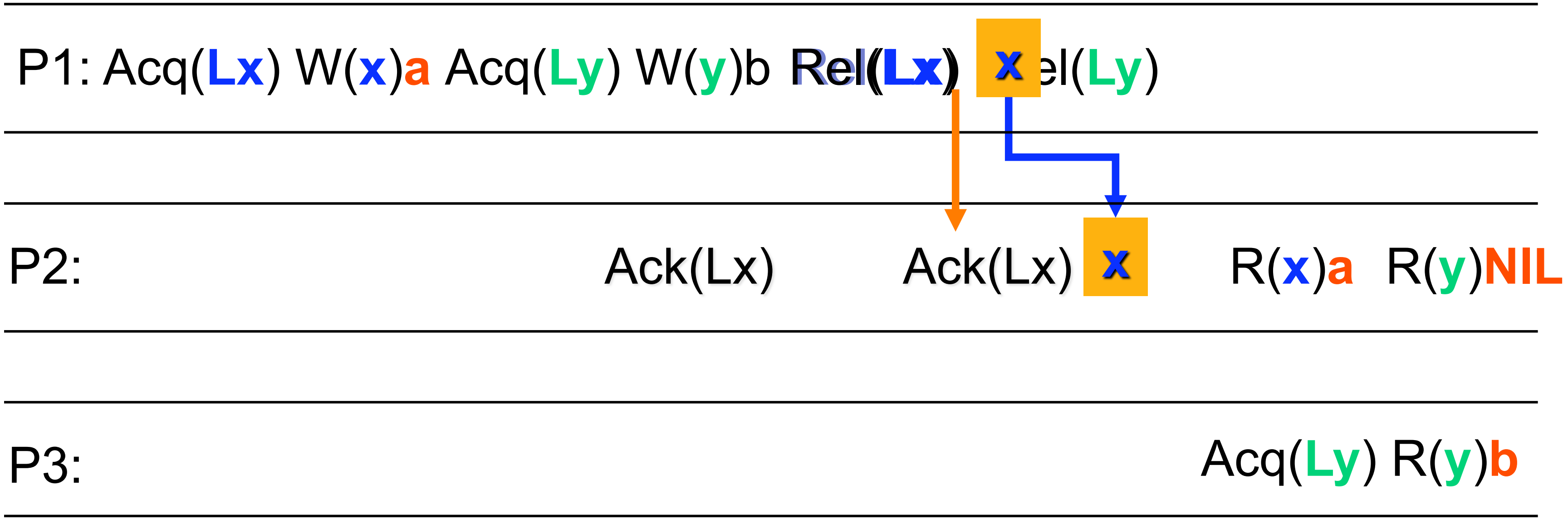
- release: nothing is sent
- acquire: get the most recent values

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: first 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: Ack(Lx) Ack(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 (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 used. Writes from different processes may not always be seen in that order.

Consistency models not using synchronization operations.

Summary of Consistency Models (2)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	All shared data are made consistent after the exit out of the critical section
Entry	Shared data associated with a synchronization variable are made consistent when a critical section is entered.

Models with synchronization operations.

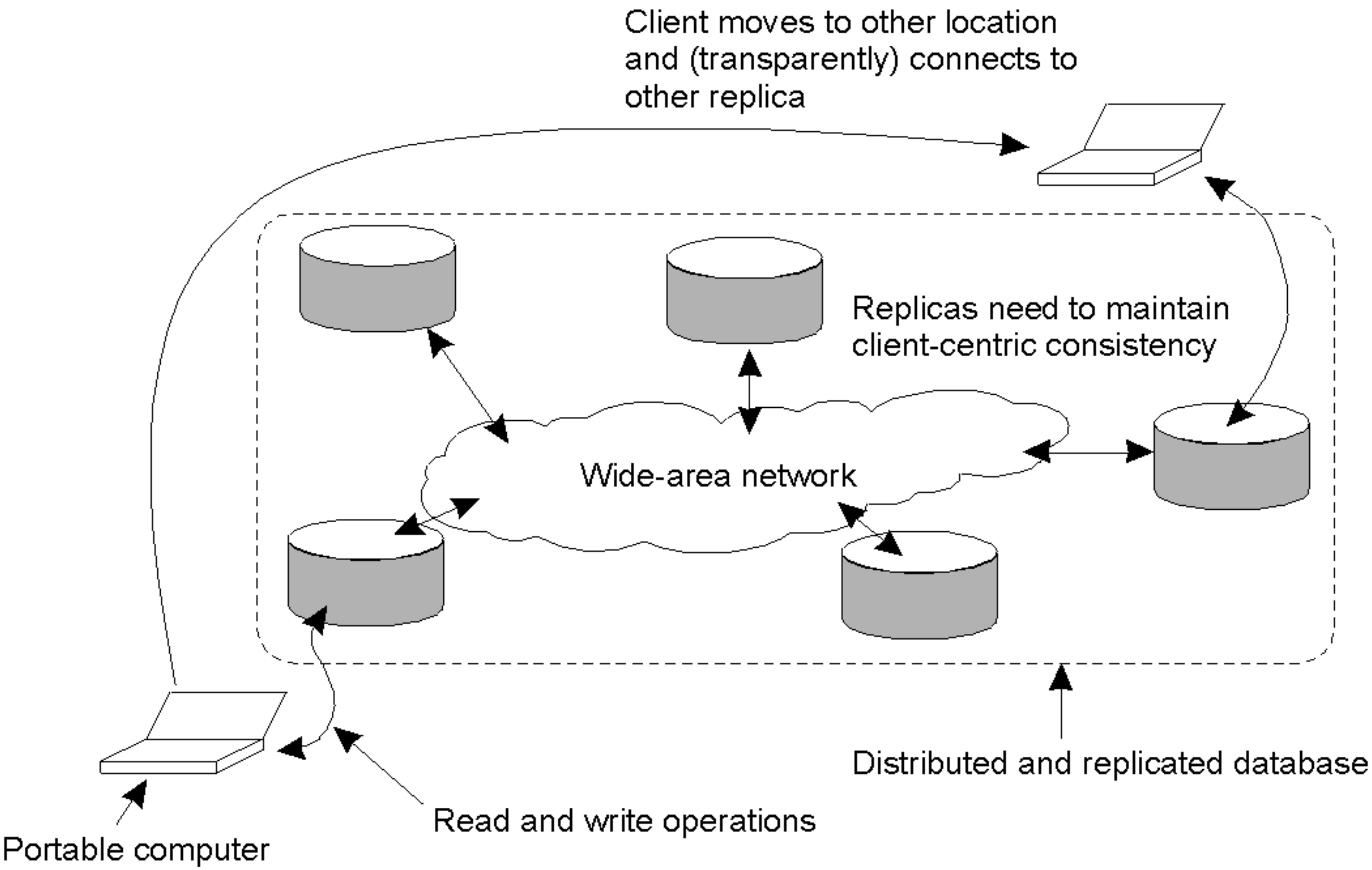
Client-Centric Models

- Environment
  - most operations: “read”
  - “no” simultaneous updates
  - a relatively high degree of inconsistency tolerated

    (examples: DNS, WWW pages)

- Wanted
  - eventual consistency
  - consistency seen by one single client

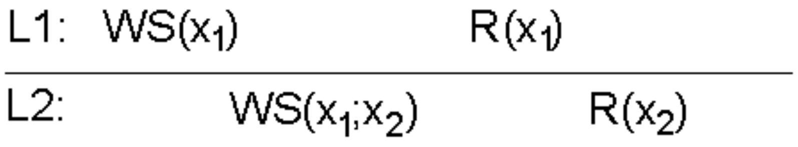
Eventual Consistency


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

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)

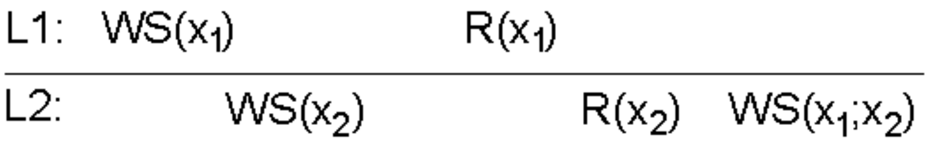


L1: WS(x_1) R(x_1)

L2: WS(x_1;x_2) R(x_2)

(a)

A monotonic-read consistent data store



L1: WS(x_1) R(x_1)

L2: WS(x_2) R(x_2) WS(x_1;x_2)

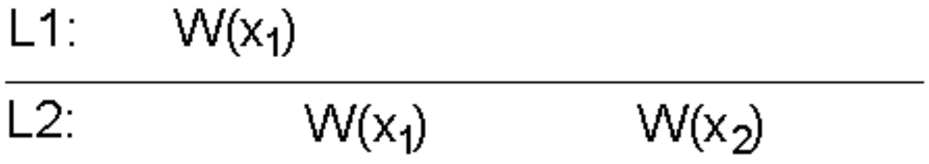
(b)

A data store that does not provide monotonic reads.

WS(x_i): write set = sequence of operations on x at node L_i

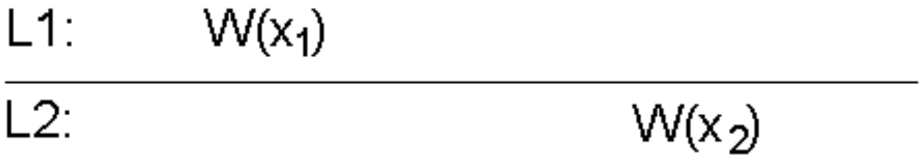
Monotonic Writes

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)

A monotonic-write consistent data store.

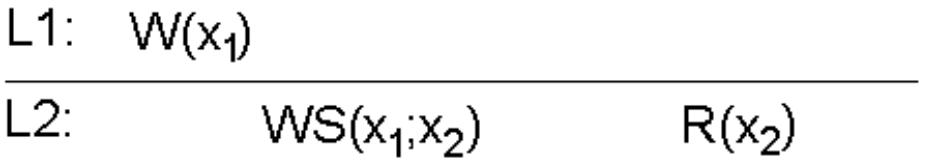


(b)

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: edit www-page)

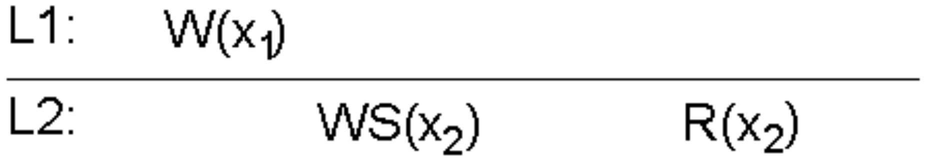


L1: $W(x_1)$

L2: $WS(x_1;x_2)$ $R(x_2)$

(a)

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



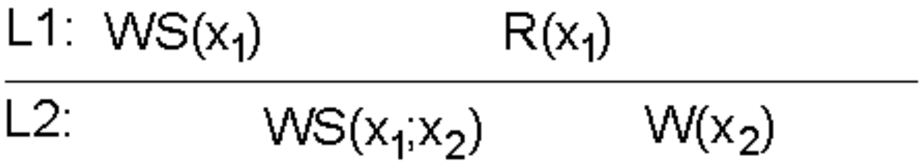
L1: $W(x_1)$

L2: $WS(x_2)$ $R(x_2)$

(b)

A data store that does not.

Writes Follow Reads

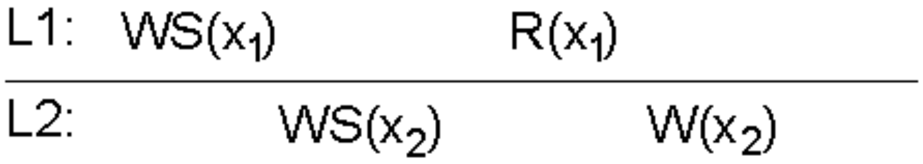


L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_1;x_2)$ $W(x_2)$

(a)

A writes-follow-reads consistent data store



L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_2)$ $W(x_2)$

(b)

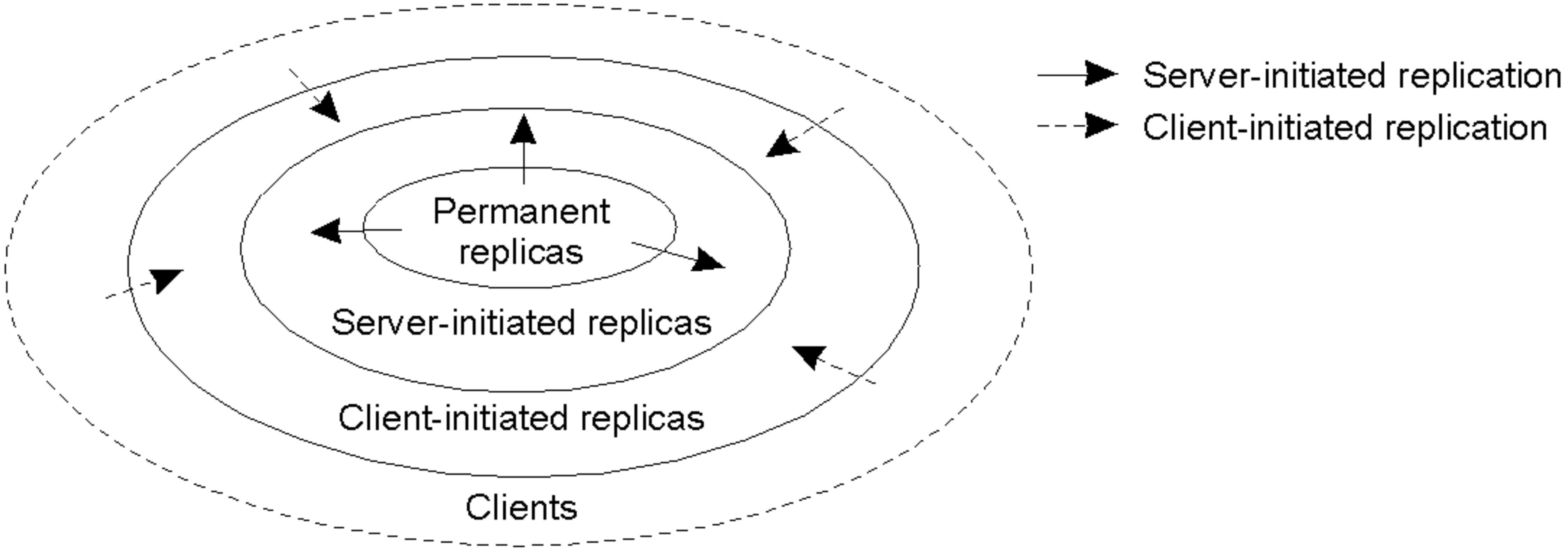
A data store that does not provide writes-follow-reads consistency

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)

Distribution Protocols

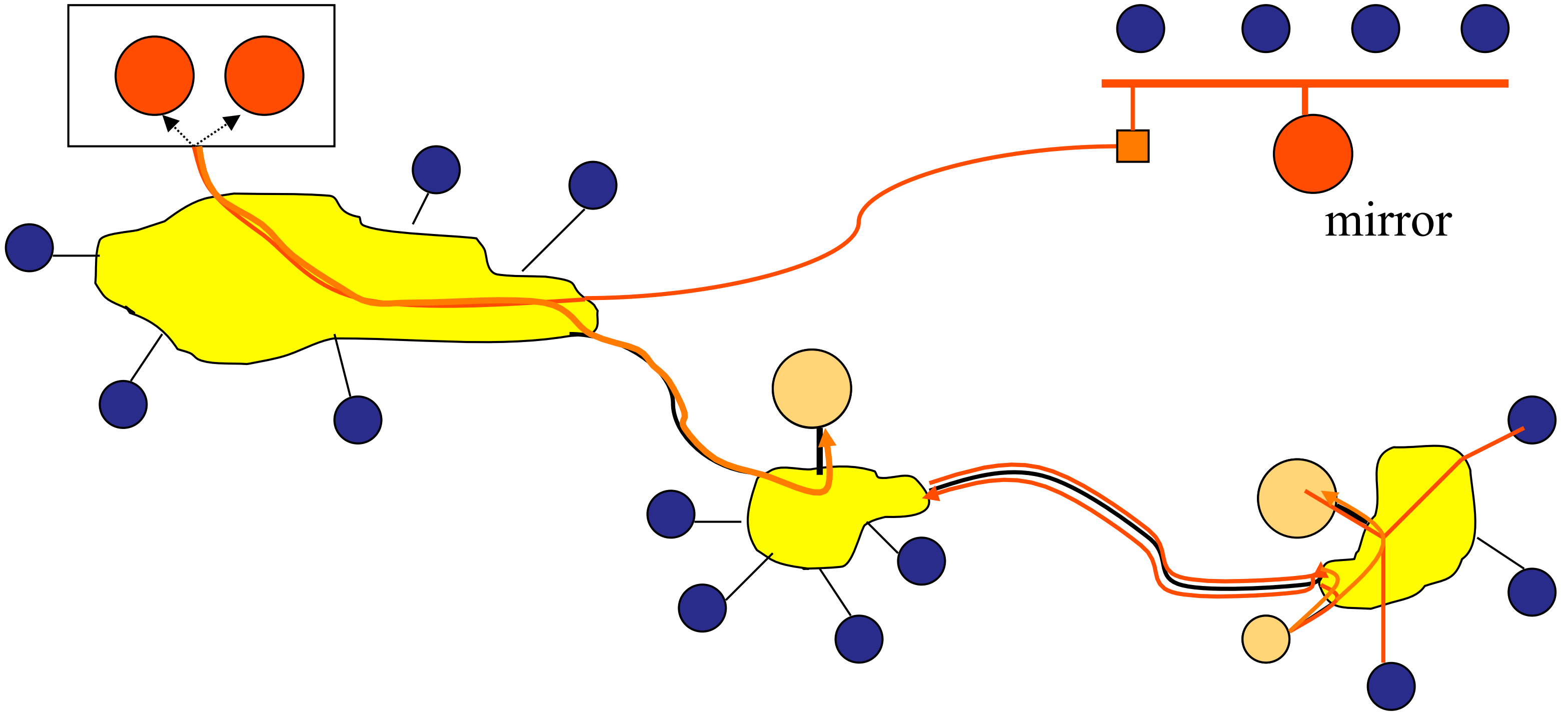
- Replica placement
- Update propagation
- Epidemic protocols

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
- Called as **push caches**
- *Example: www hosting services*
    - *a collection of servers*
    - *provide access to www files belonging to third parties*
    - *replicate files “close to demanding clients”*

Server-Initiated Replicas (2)

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

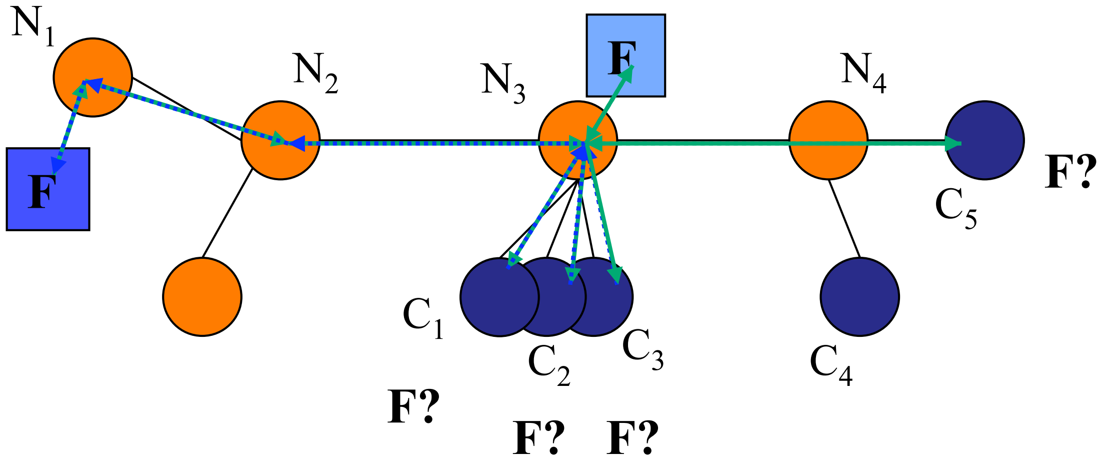
⇒ bring files to servers placed in the proximity of clients

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

Client-Initiated Replicas

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

Example: Shared Cache in Mobile Ad Hoc Networks



● client

● server

1. C_1 : Read F => N_1 returns F
2. N_3 : several clients need F => Cache F
3. C_5 : Read F => N_3 returns F

Source: Cao et al, Cooperative Cache-Based Data Access …; Computer, Febr. 2004

Update Propagation: State vs. Operations

- Update route: client => copy => {other copies}
- 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 (useful if high read-to-write ratio)
  - propagate the update operation (**active replication**)

Pull versus Push (1)

- Push
  - a server sends 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}}

Pull versus Push Protocols (2)

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.

Pull vs. Push: Environmental Factors

- Read-to-update ratio
  - high => push (one transfer – many reads)
  - low => pull (when needed – check)
- Cost-QoS ratio
  - factors:
    - update rate, number of replicas => maintenance workload
    - need of consistency (guaranteed vs. probably_ok)
  - examples
    - (popular) web pages
    - arriving flights at the airport
- Failure prone data communication
  - lost push messages => unsuspected use of stale data
  - pull: failure of validation => known risk of usage
  - high reqs => combine push (data) and pull

Leases

- Combined push and pull
- A “server promise”: push updates for a certain time
- A lease expires

  => the client

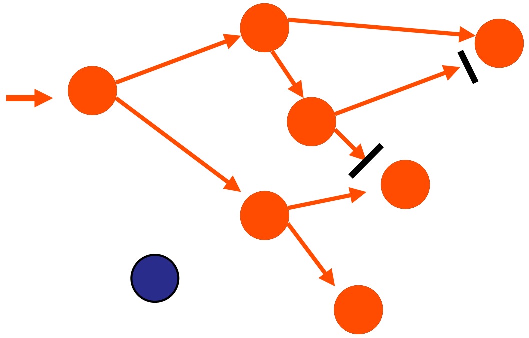
  - polls the server for new updates or
  - requests a new lease
- Different types of leases
  - age based: {time to last modification}
  - renewal-frequency based: long-lasting leases to active users
  - state-space overhead: increasing utilization of a server => lower expiration times for new leases

Propagation Methods

- Data communication
  - LAN: push & multicasting, pull & unicasting
  - wide-area network: unicasting
- Information propagation: epidemic protocols
  - a node with an update: **infective**
  - a node not yet updated: **susceptible**
  - a node not willing to spread the update: **removed**
  - **propagation**: **anti-entropy**
    - **P** picks randomly **Q**
    - three information exchange alternatives:

      P => **Q** or **P** <= **Q** or **P**⇔**Q**
  - **propagation**: **gossiping**

Gossiping (1)



P starts a gossip round (with a fixed k)

1. P selects randomly $\{Q_1,..,Q_k\}$
2. P sends the update to $\{Q_i\}$
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 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: 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 =>

    P{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)

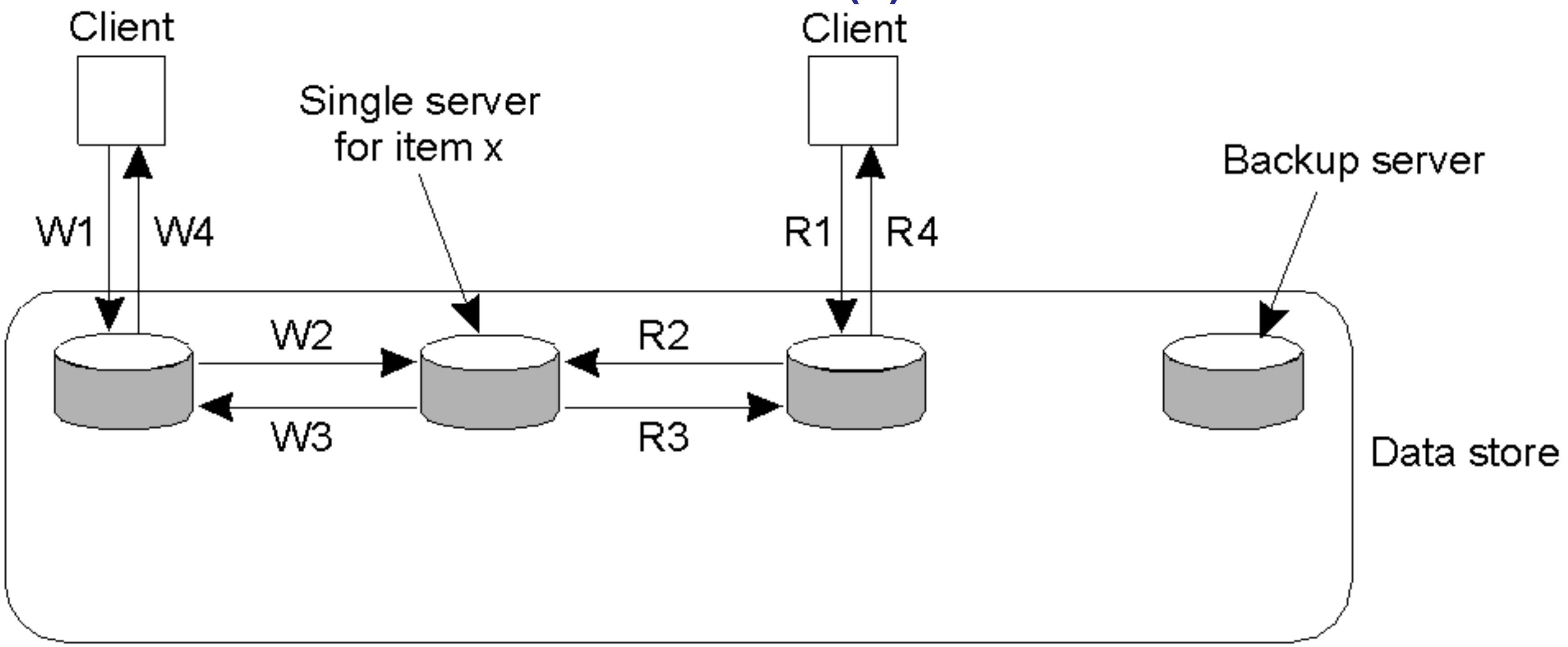
A solution: deletion is a special update *(death certificate)*

- allows normal update communication
- a new problem: cleaning up of death certificates
- solution: time-to-live 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)

Remote-Write Protocols (1)

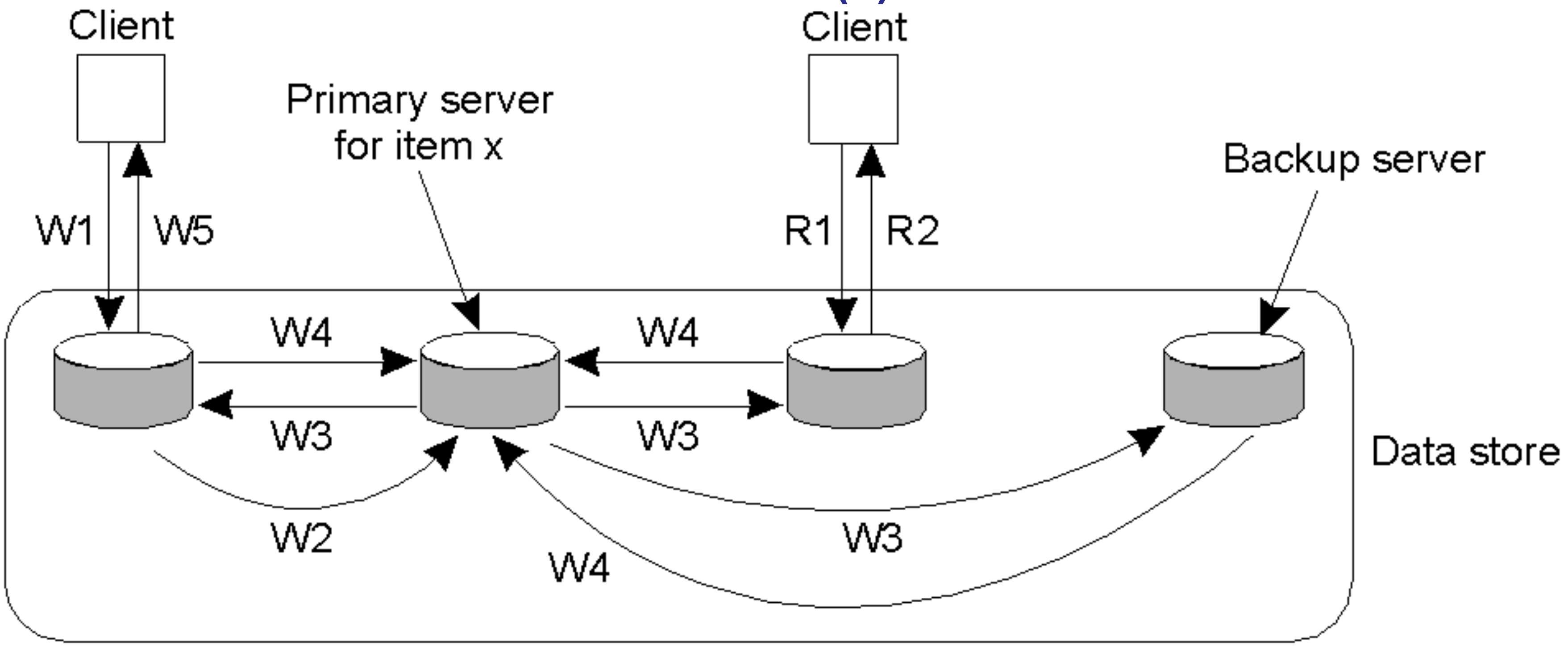


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

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

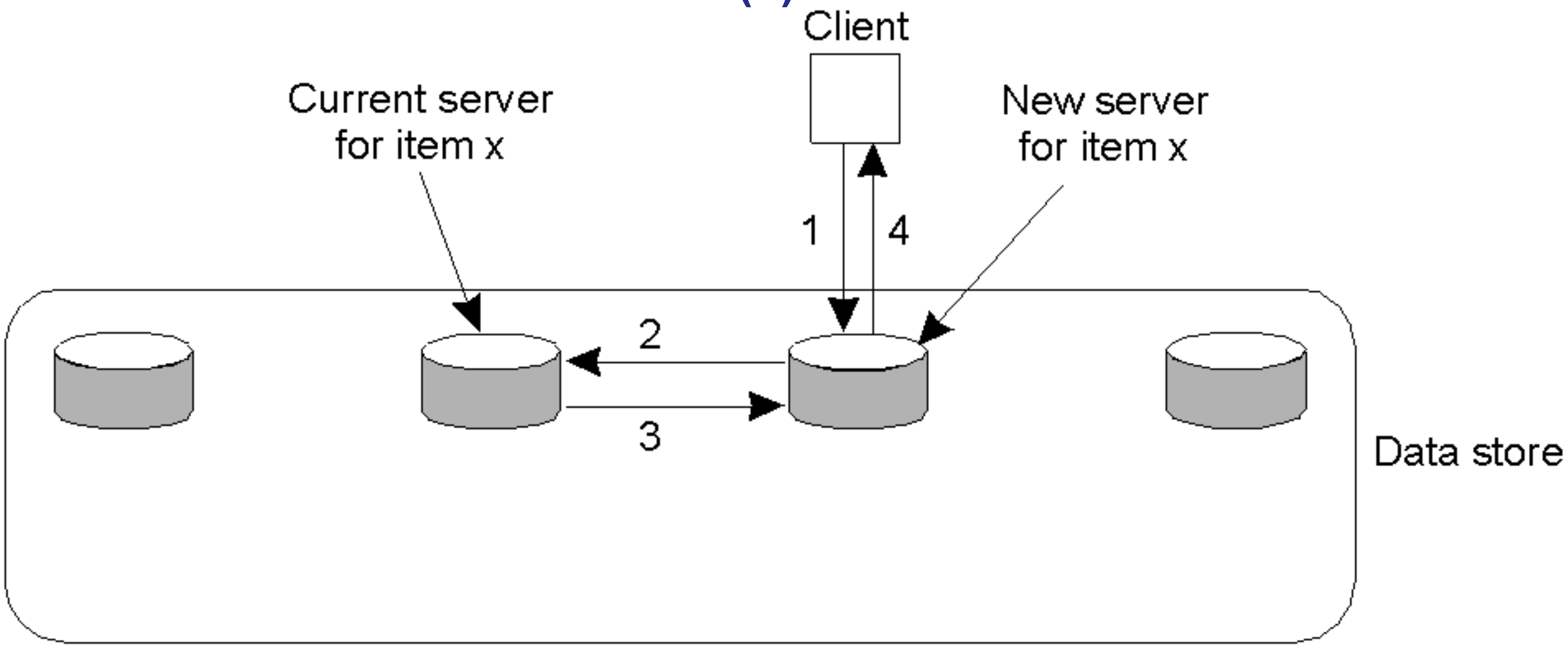
R1. Read request
R2. Response to read

Sequential consistency

Read Your Writes

The principle of primary-backup protocol.

Local-Write Protocols (1)

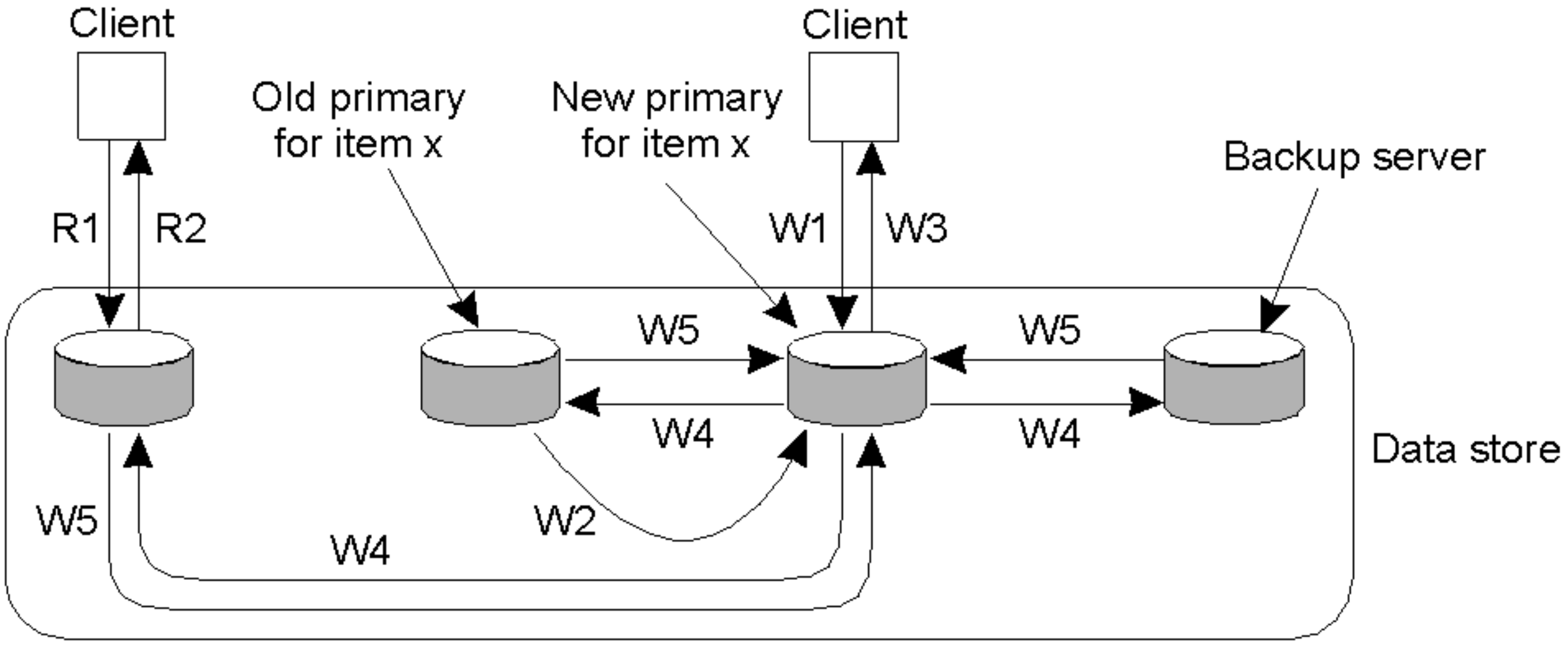


1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

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.

Active replication (1)

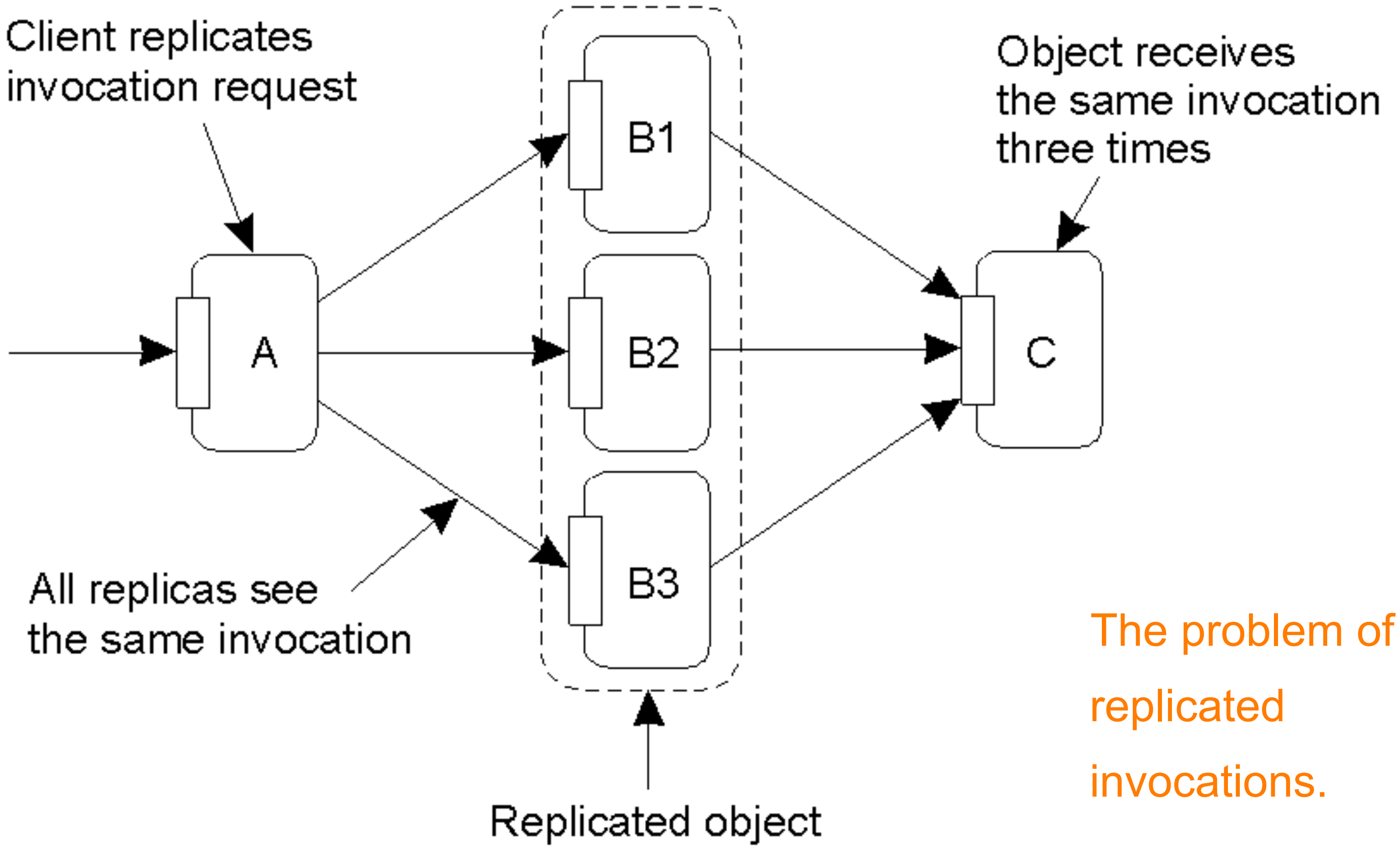
- Each replica:

  an associated process carries out update operations

- Problems
  - replicated updates: total order required
  - replicated invocations

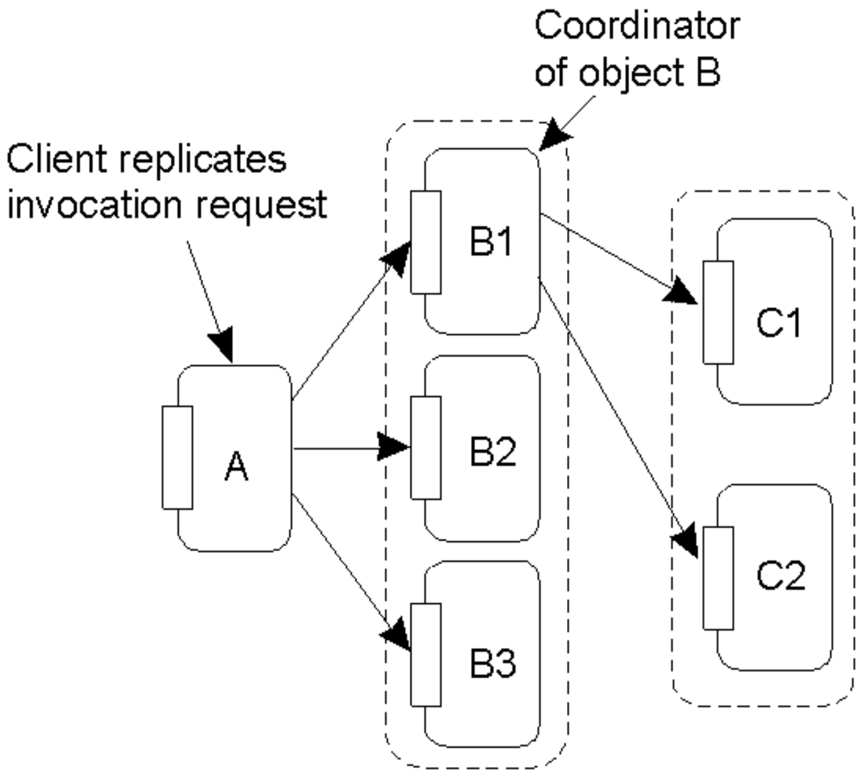
- Total order:
  - sequencer service
  - distributed algorithms

Active Replication (2)

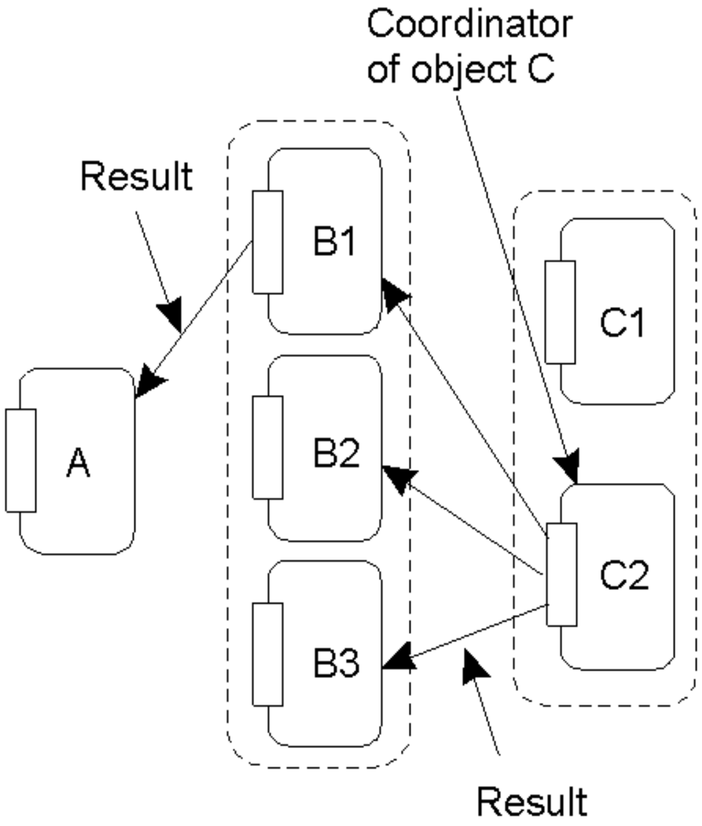

Client replicates invocation request
B1
B2
B3
A
C
Object receives the same invocation three times
All replicas see the same invocation
Replicated object

The problem of replicated invocations.

Active Replication (3)



Forwarding an invocation request from a replicated object



Returning a reply to a replicated object.

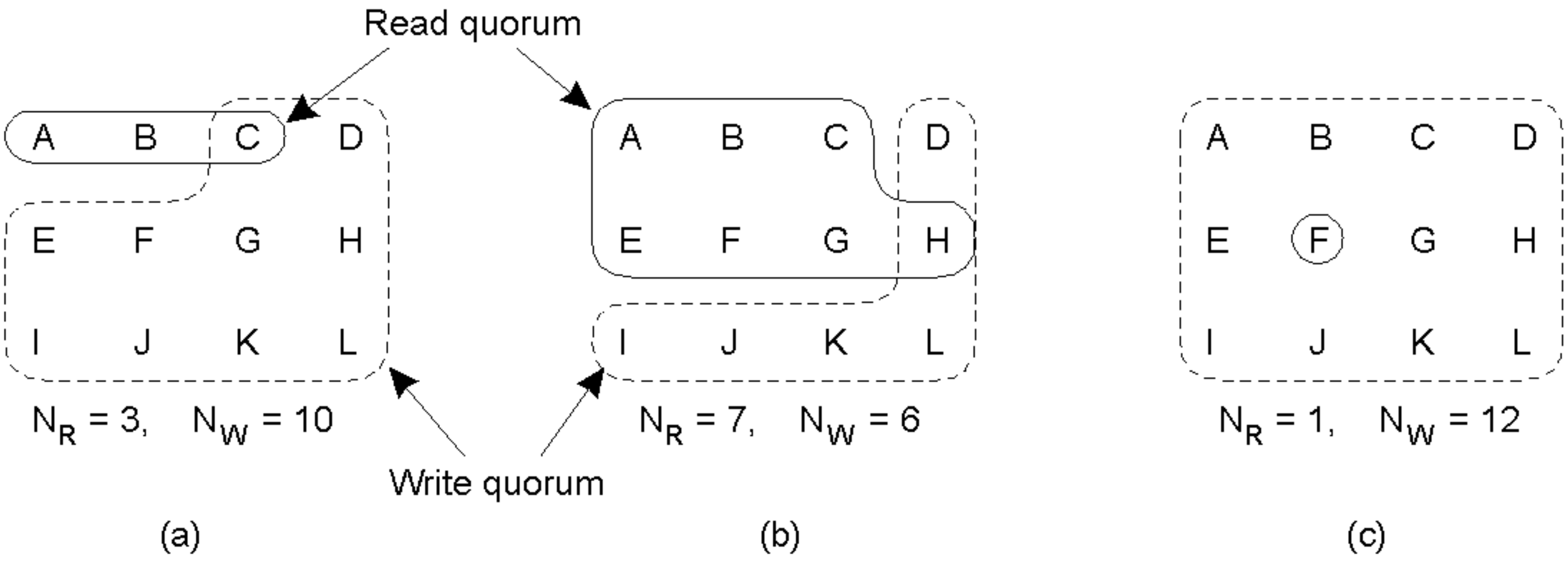
Quorum-Based Protocols

- Consistence-guaranteeing update of replicas:

  an update is carried out as a transaction
- Problems
  - Performance?
  - Sensitivity for availability (all or *nothing*) ?
- Solution:
  - a **subgroup of available** replicas **is allowed** to update data
- Problem in a partitioned network:
  - the groups cannot communicate =>

    each group must decide independently whether it is allowed

    to carry out operations.
- A **quorum** is a group which is large enough for the operation.



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 Consensus: Examples

		Example 1	Example 2	Example 3
Latency	Replica 1	75	75	75
(msec)	Replica 2	65	100	750
	Replica 3	65	750	750
Voting configuration	Replica 1	1	2	1
	Replica 2	0	1	1
	Replica 3	0	1	1
Quorum sizes	R	1	2	1
	W	1	3	3
Derived performance of file suite:				
Read	Latency	65	75	75
	Blocking probability	0.01	0.0002	0.000001
Write	Latency	75	100	750
	Blocking probability	0.01^{-}	0.0101	0.03

CoDoKi, p. 600

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 *(WHY?)*
- Update all replicas belonging to the write quorum.

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

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)