Programming of Distributed Systems

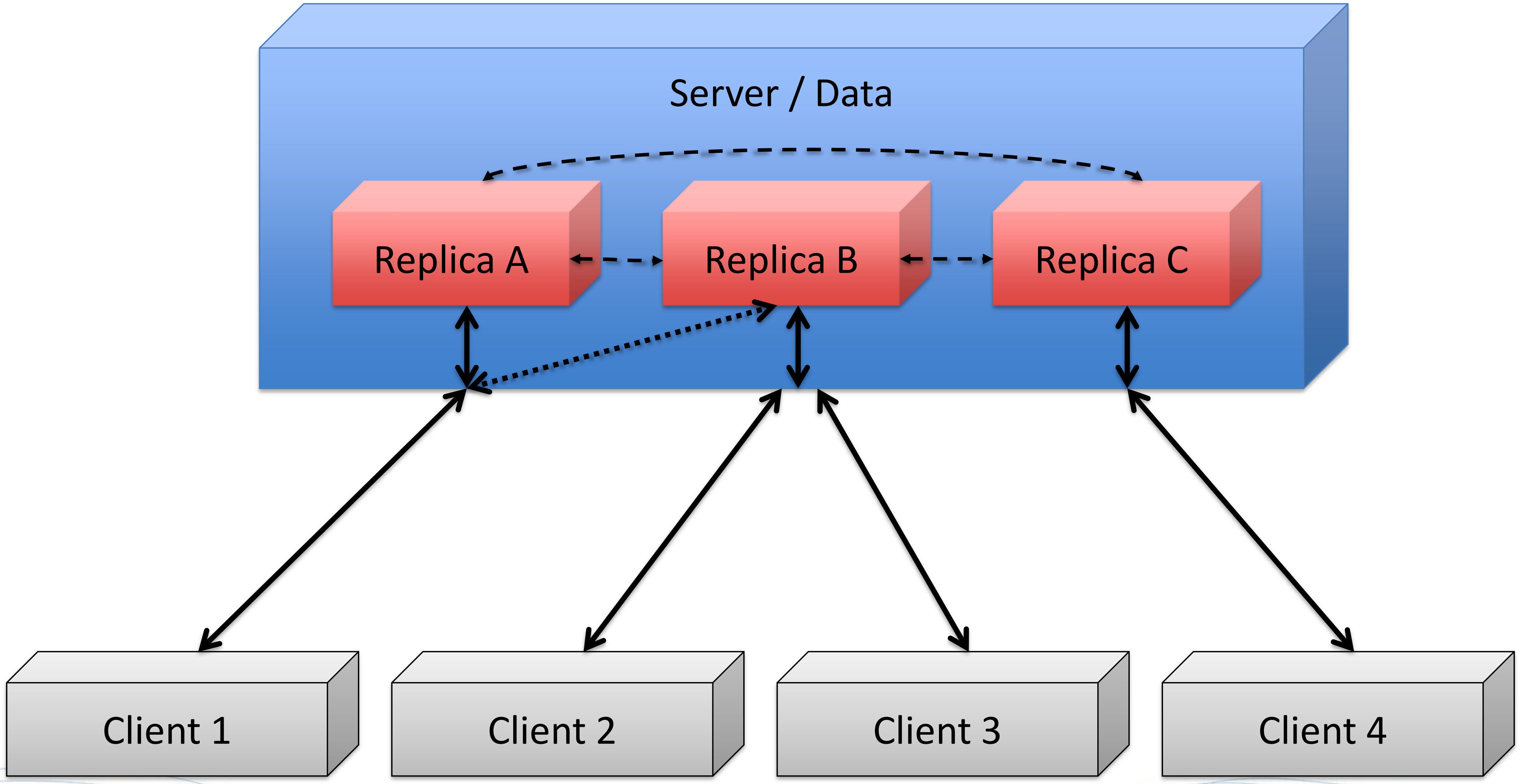
Topic VI – Replication & Consistency

Dr.-Ing. Dipl.-Inf. Erik Schaffernicht

Reading Remarks

Reading Task:
Chapter 7

Replication Transparency



Reasons to Replicate

Dependability

- availability
  - there is always a server somewhere
- reliability
  - fault tolerance regarding data corruption and faulty operations

Performance

- response time
- throughput
- scalability

Problems with Replication

Changes to one replica have to be propagated to the other replicas in order to be consistent

→ What is meant by ’consistent’?

→ When to propagate modifications?

→ How to propagate modifications?

CAP theorem

Consistency

- Data items behave as if there is only one copy
- Cave-at: Similar to ACID's *atomicity*, not ACID's *consistency*!

Availibility

- Node failures do not prevent the system from continuing to operate

Partition-tolerance

- The system continues to operate in the presence of network partitions

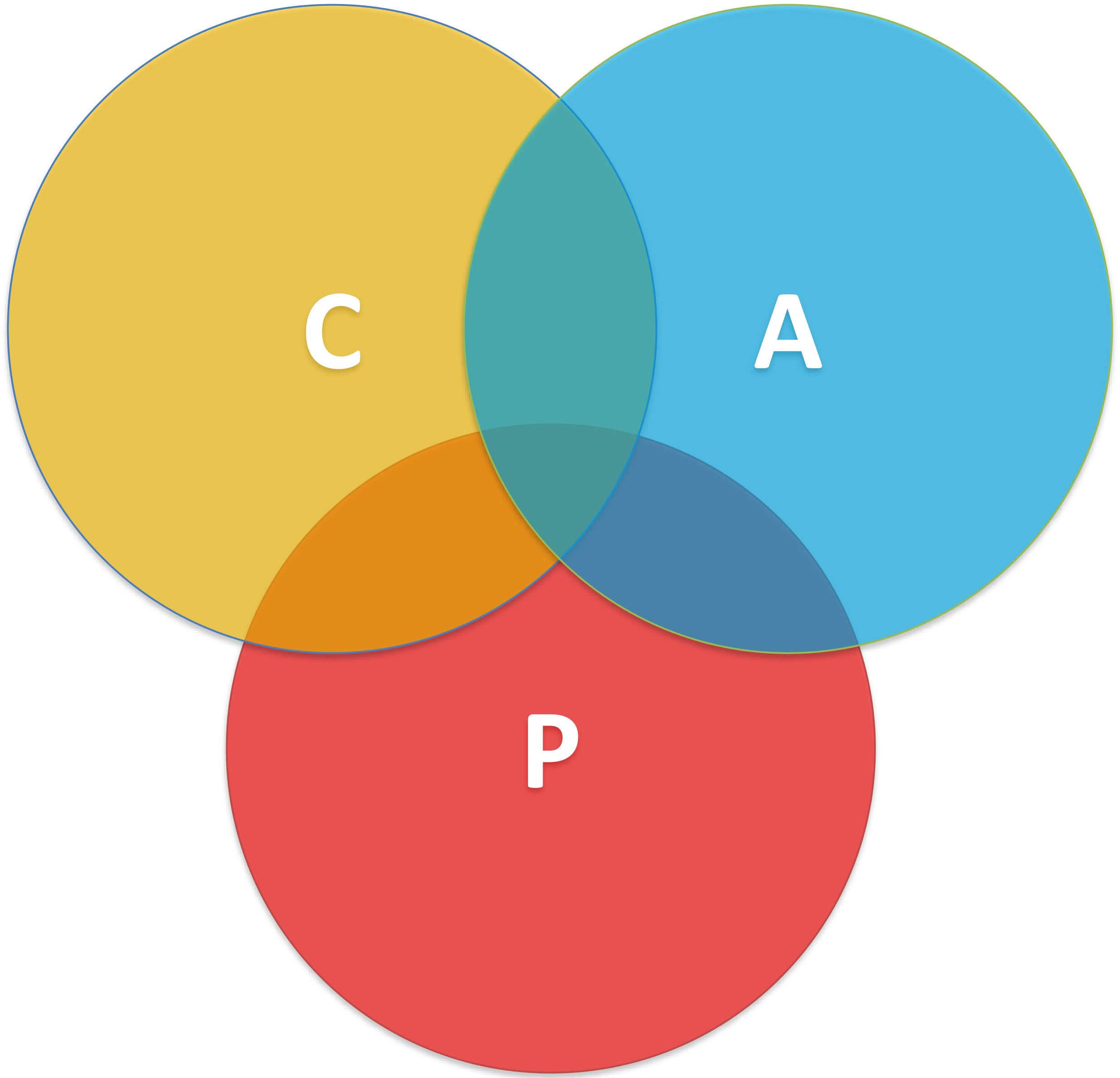
CAP theorem (2)

Simple (mis-)interpretation

- no system can have all 3 properties (in a very strict sense)

Somewhat better

In the presence of network partitions, one has to give up on either consistency (AP system) or availability (CP system)



AP – Best Effort Consistency

AP systems relax consistency in favor of availability, but are not totally inconsistent

Examples

- Caches
- Content Distribution Networks (CDN)
- Domain Name System (DNS)
- Conflict-free replicated data type (CRDT)

CP – Best Effort Availibility

CP systems sacrifice availability for consistency,
but are not unavailable

Examples

- Majority protocols (Paxos, Raft, see end of lecture)
- Distributed locking (Chubby lock service)

Consistency

Intuitive definition

> A set of replicas is **consistent** when all the replicas are always the same.

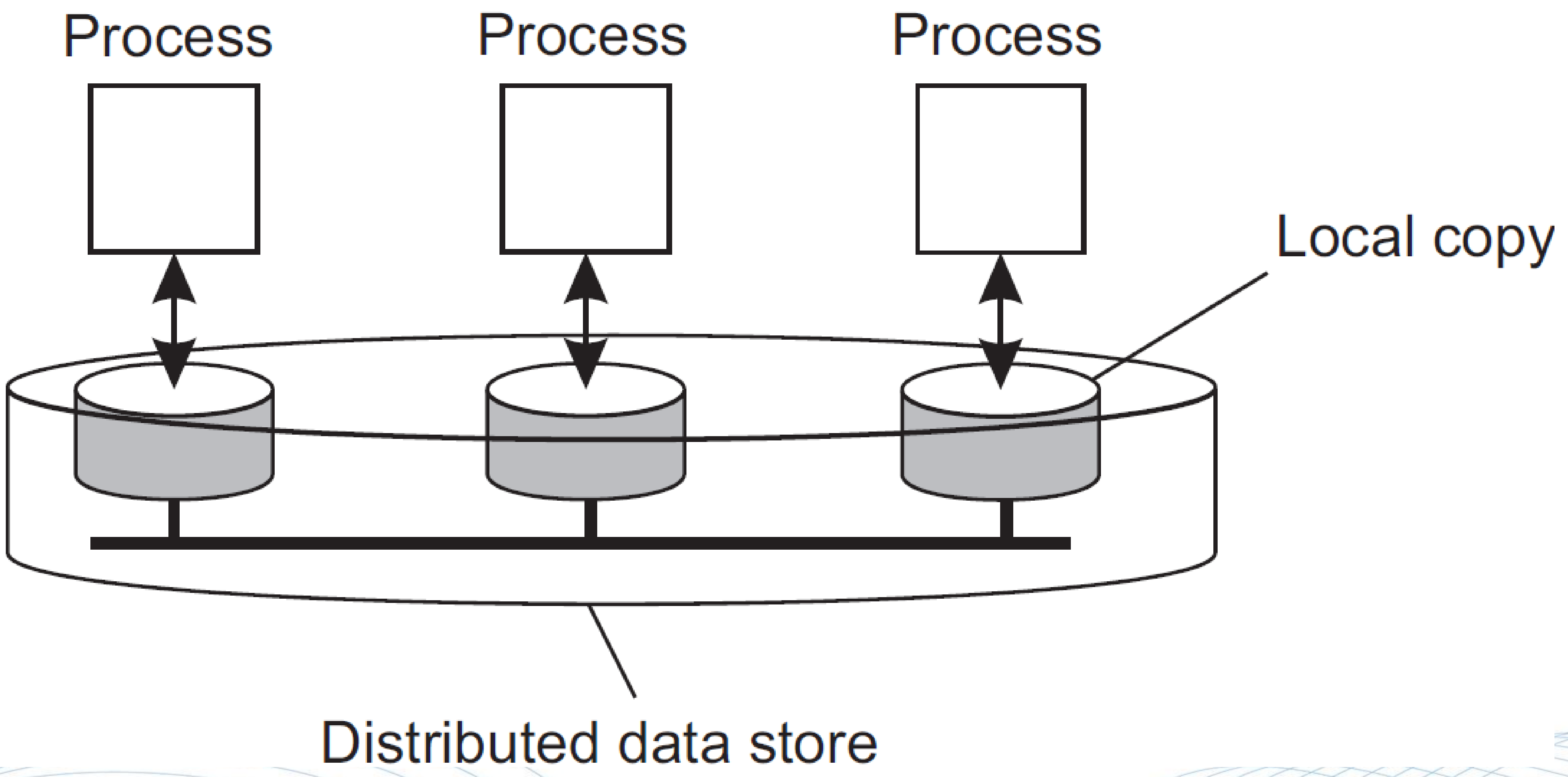
➔ all conflicting operations are done in the same order everywhere (*global synchronization*)

Danger of **disimprovements**! Performance improvements by introducing additional costs for replica management?

➔ Loose up the requirements to avoid global sychronous updates

A data-centric view

A **data store** is a physically distributed collection of storages that are replicated over multiple processes



Any operation that changes the data is considered a `write` operation.

Any other operation is a `read` operation.

Data-centric consistency models

Two types of conflicts

- `read-write`: concurrent read and write operations
- `write-write`: two concurrent write operations

→ Consistency means conflicting operations are done in the same order everywhere

Consistency models

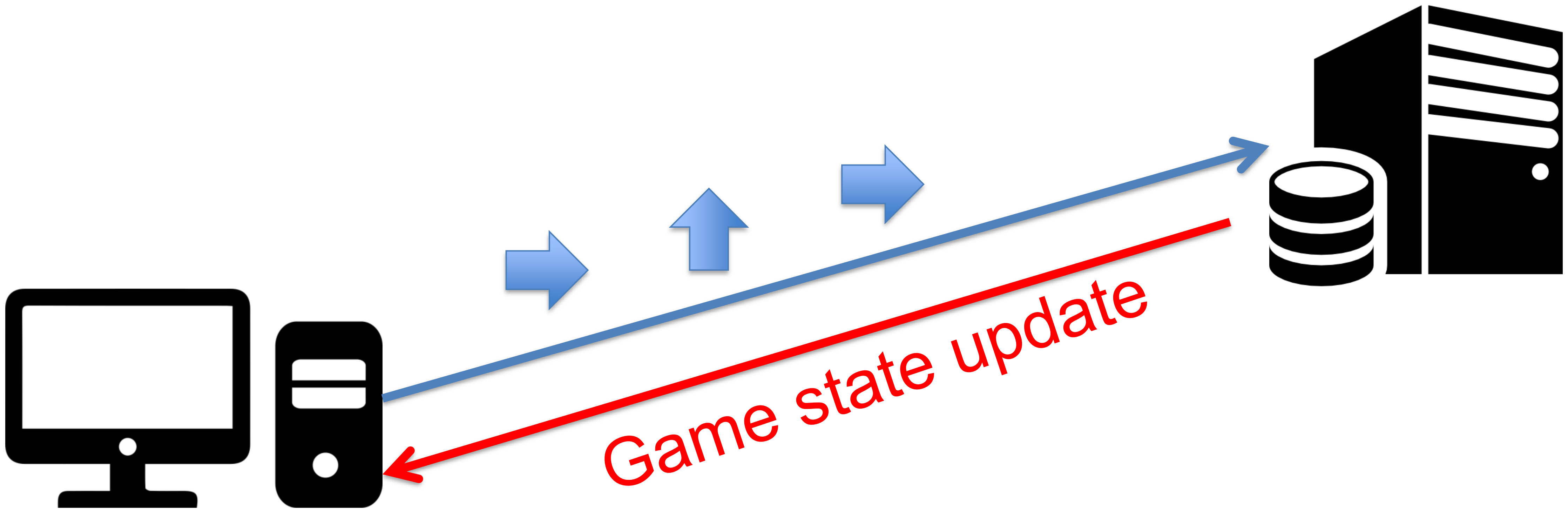
What is the guaranteed result of concurrent operations?

Degrees of consistency

Three different aspects of *loose* consistency

- replicas may differ in their (numerical) **values**
- replicas may differ in their **staleness**
- replicas may differ in their **order** of performed updates

Example: Consistency in an Online Game



Game Server:

- state of the game (position of players on the map, inventory, etc)
- conflict resolution

Client:

- user interface
- Keyboard/mouse inputs
- partial game state

Example: Consistency in an Online Game

Local Game Server:

- state of the game

Two replicas of the game state that need to be consistent

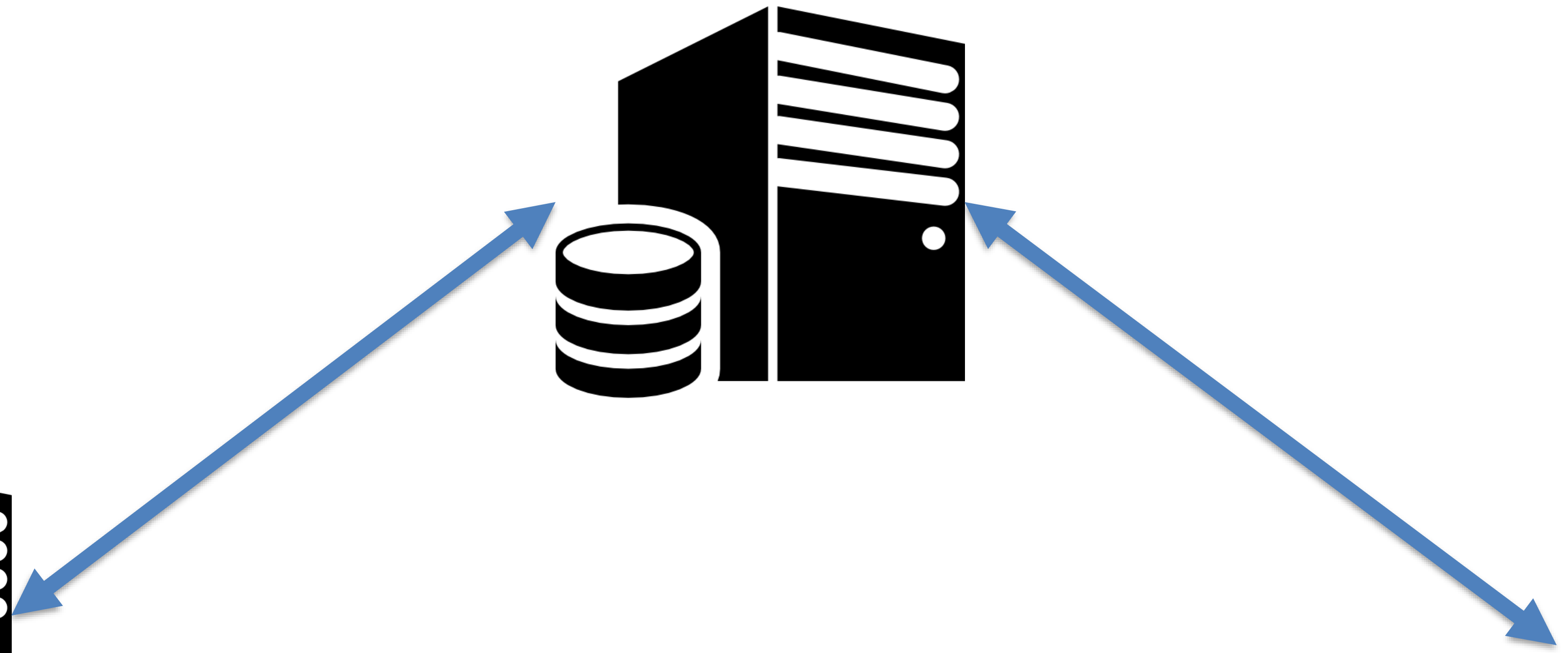
Remote Game Server:

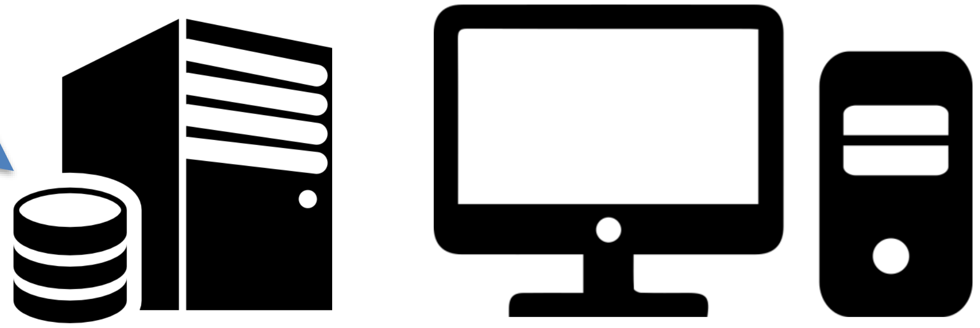
- state of the game (position of players on the map, inventory, etc)
- conflict resolution

Client:

- user interface
- Keyboard/mouse inputs
- partial game state

Example: Online 3D Shooter





Multiple Game States:

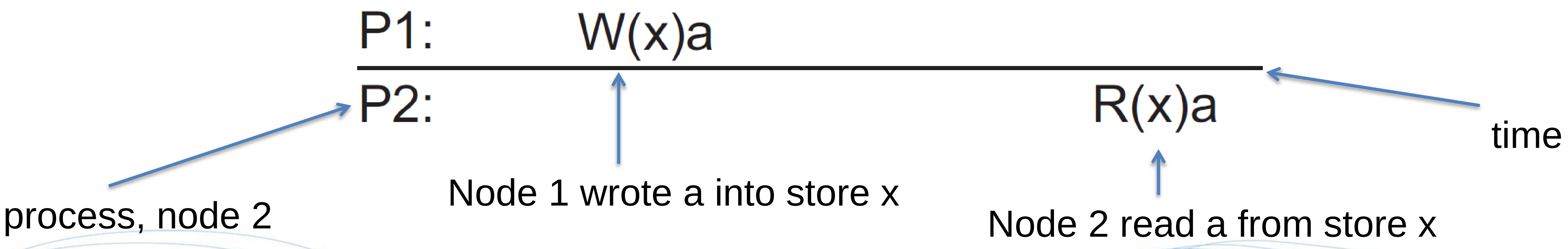
- Each one potentially different

➔ Inconsistencies can appear puzzling to players

Strict consistency

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

Notation from the book:



Sequential consistency

The **order** of operations applied on each replica is the **same**. Operations from the same node follow the order given by its program.

Comments

- any order of reads and writes of different machines is acceptable, as long as they are the same for each replica (linearization of the concurrent processes)
- no notion of time (*most recent*)

Sequential consistency

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

sequentially consistent

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

sequentially not consistent

Example

Three variables `x,y,z` initialized with 0

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

Node 2
```
y = 1;
print(x,z)
```

Node 3
```
z = 1;
print(x,y)
```

→ 90 different valid execution sequences

Consistency model for sequential consistency allows any of those 90 sequences as correct results!

Sequence 27
```
z = 1;
x = 1;
y = 1;
print(x,y)
print(y,z)
print(x,z)
```

Output
```
111111
```

Sequence 13
```
z = 1;
x = 1;
print(y,z)
print(x,y)
y = 1;
print(x,z)
```

Output
```
011011
```

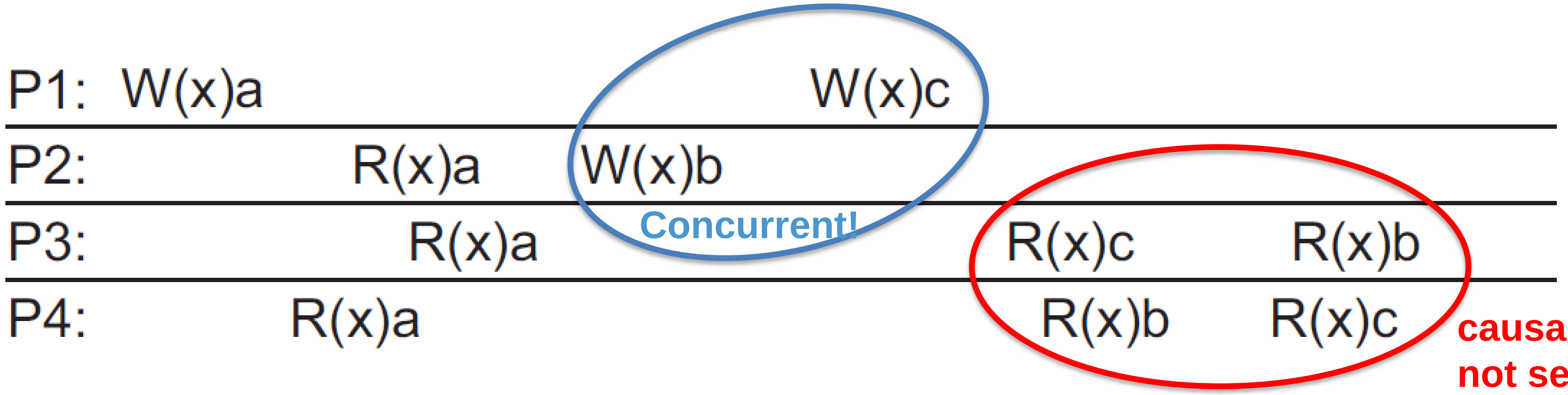
Causal consistency

The order of **potentially causally related** write operations applied on each replica is the same. **Concurrent** write operations can have different order in each replica.

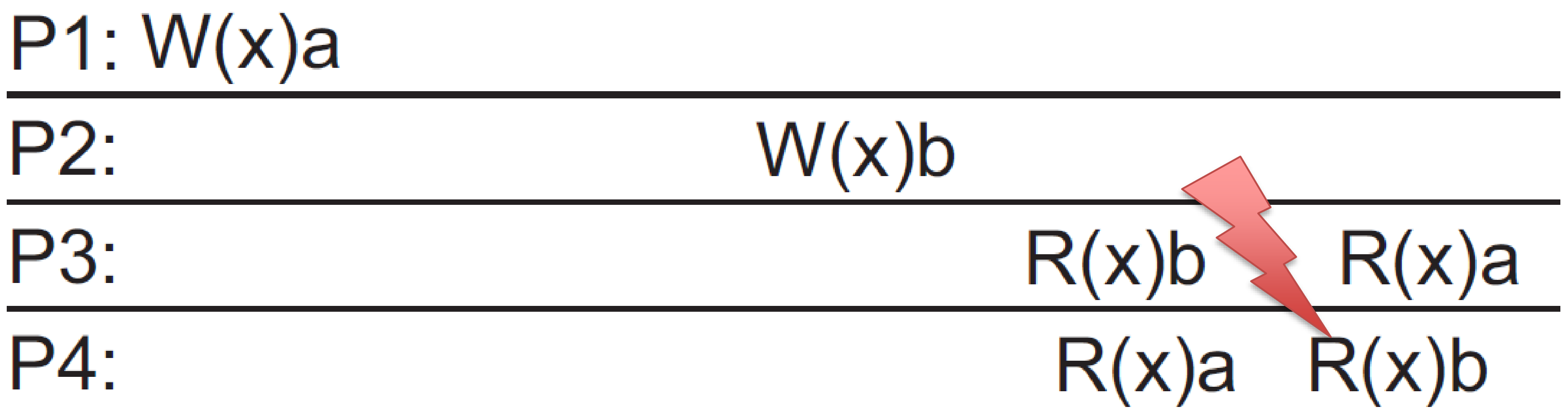
Comments

- weaker requirements than sequential consistency
- concurrent = not (potentially causally related)
- causal dependencies modelled with a graph → not trivial

Causal consistency



causally consistent
not sequential consistent
not strict consistent



ÖREBRO UNIVERSITY

FIFO consistency

> Write operations from a single node are applied to each replica in the correct order, but writes from different nodes may be applied to each replica in a different order.

Comments

- weaker requirements than causal consistency
- rather easy to implement

FIFO Consistency

Valid sequence of FIFO consistency

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

Consistency & mutual exclusion

Idea

- considering sequences of operations as critical sections with access control provided by locks

➔ turning the sequence into a single atomic operation, where intermediate results do not matter
(→ Lecture 2, ACID properties of transactions)

Explicit sychronization before entering a CS (entry consistency) or after leaving a CS (release consistency)

Eventual consistency

- many systems have considerable more read than write operations and/or limited nodes that are allowed to perform write operations

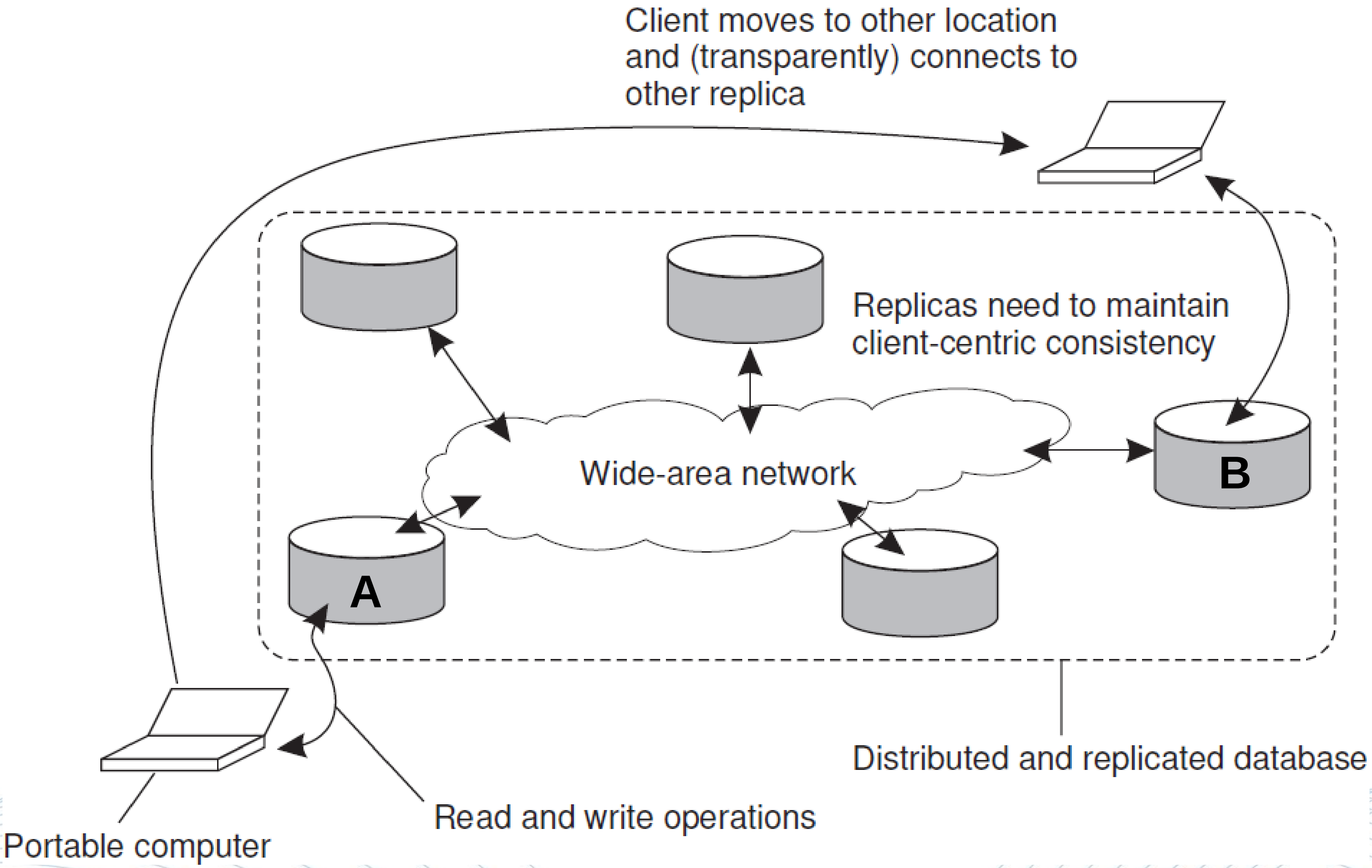
→ low chance of write-write conflicts

e.g. Content distribution networks, DNS, web pages (read-write inconsistencies easy to tolerate)

➔ lazy updates, but after a long enough time with out write operations consistency will **eventually** be achieved

Client-centric consistency

- global view vs. local view on consistency



Only the states in A & B need to match, the state of the overall store is irrelevant for the user impression of consistency.

Replica & Content Placement

Goal

Find *k* ’good’ servers to place items choosing from *n* options

Optimization criteria

- minimizing average latency between clients and replicas
- minimizing difference of bandwith utilization of replicas

Replica & Content Placement

Optimal solution

- NP hard → not feasible

’Good’ approximate solutions using cluster analysis

- iterative/recursive procedures to form groups
- still to expensive / slow → > $O(n^2)$

Practical solutions based on heuristics

- allow real-time placement of replicas

Replica & Content Placement

Three different types of replicas

Permanent replicas

- node always having a replica

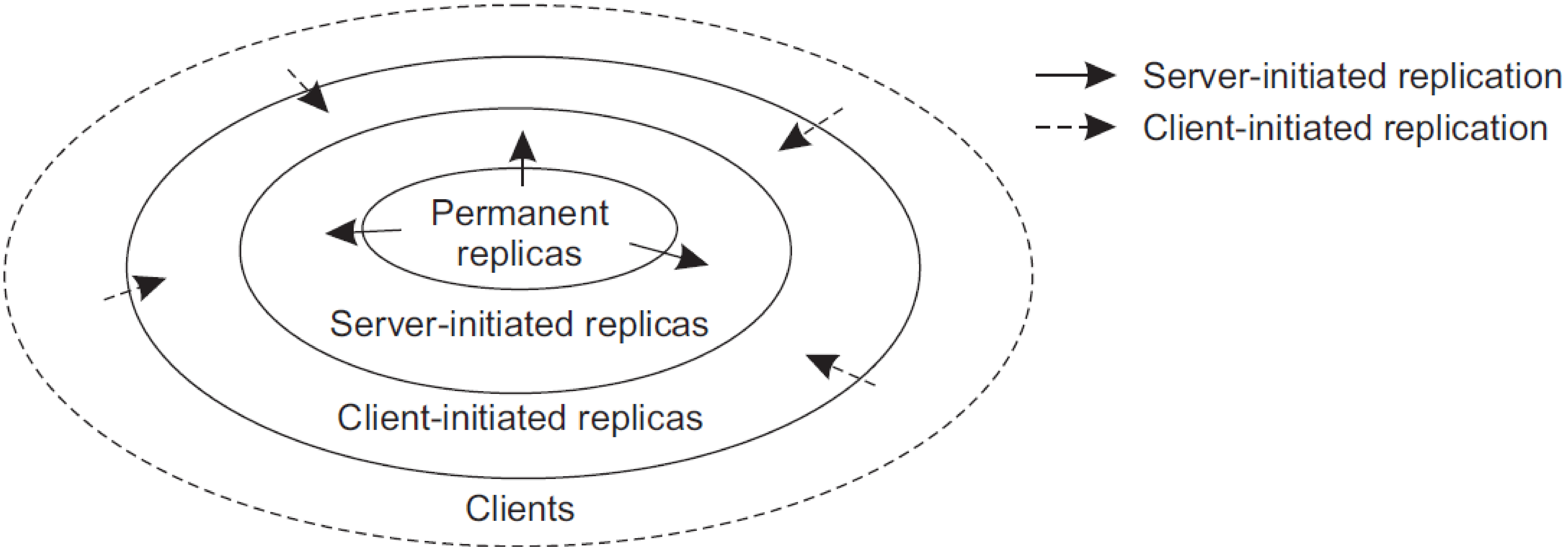
Server-initiated replica

- node that can dynamically host a replica on request of another server in the data store

Client-initiated replica

- node that can dynamically host a replica on request of a client

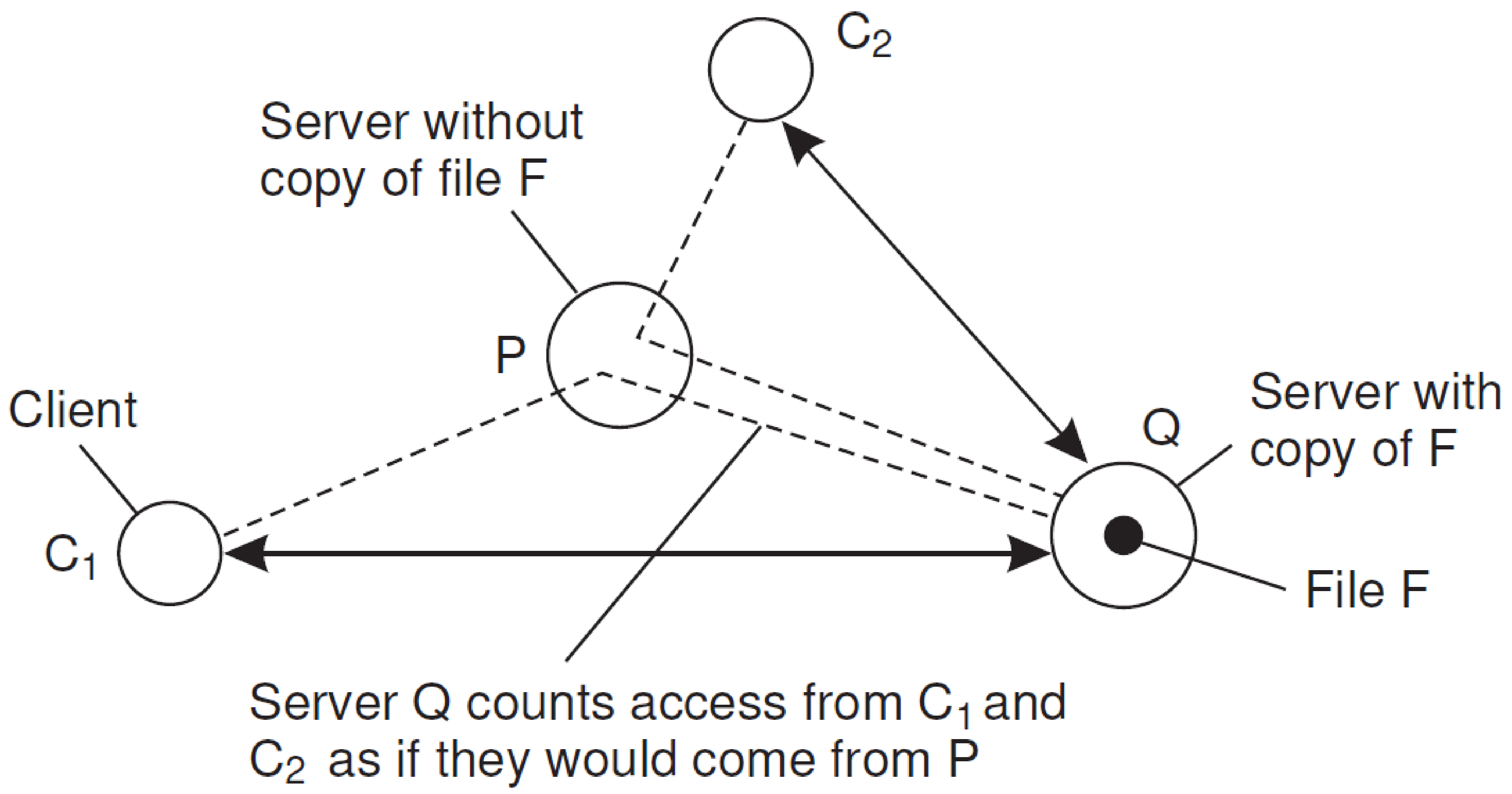
Replica & Content Placement



Server-initiated replica - example

Access counter at temporary replicas & thresholds

- Very low number of access operations → *drop data*
- Very high number of access operations → *replicate data*
- With known topology and requests coming from certain areas only → *migrate data*



Update propagation

Information

- update notifications
- updated data (passive replication)
- update operations (active replication)

Responsibility

- push → server propagates update unasked
- pull → client requests to be updated

Push vs. Pull protocols

Read-Write-ratio

- High → push
- Low → pull

Failures

- Push → use of stale (outdated) data
- Pull → known risk of using stale data
- Highly reliable systems → push + pull

Push vs. Pull protocols

Consistency model

- Strict(er) → push
- Loose(r) → pull

Cost vs. Quality-of-Service factors

- update rate & number of replicas ➔ maintainance workload
- bookkeeping for push servers
- response times
- traffic → updates vs. poll + maybe updates

Leases

Combining push and pull

- client pulls for a lease
- time interval in which the server pushes updates

Lease expiration

- fixed time
- age-based: the longer data is unchanged, the longer the lease
- renewal-frequency: the more often a clients needs data, the longer the lease
- server state based: longer leases, if server is idle

Propagation methods

Communication

- LAN: push & multicast, pull & unicast
- WAN: unicast

Algorithm & Information flow (see also Lecture 3: Naming)

- Overlay network (e.g. tree)
- Flooding (e.g. structured P2P architectures)
- Epidemic protocols

Epidemic protocols

Assumption

- no write-write conflicts → single server introduces changes
- eventual consistency model (lazy updates)
- replica passes updates only to a few neighbours

Two variants

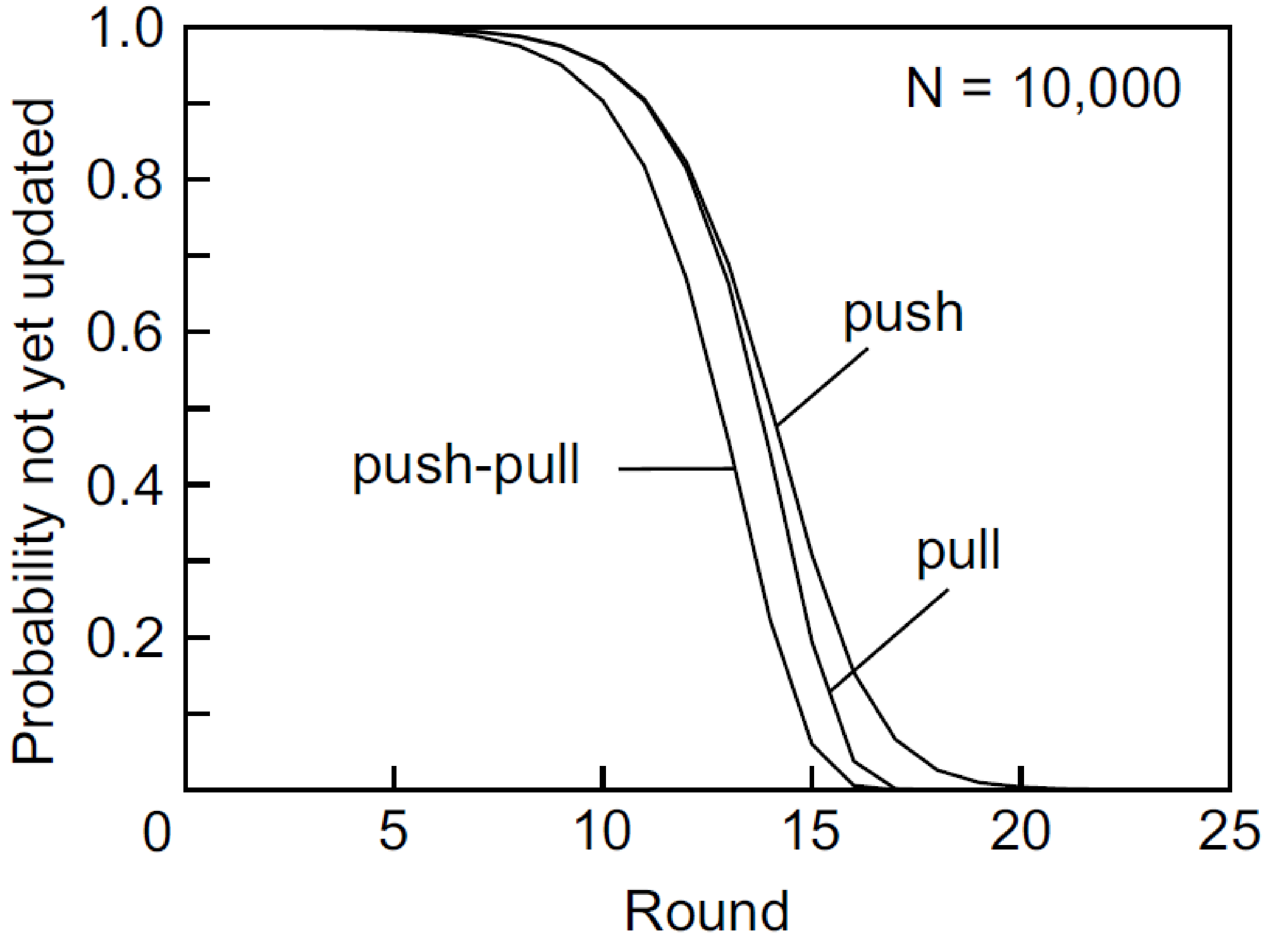
- anti-entropy
- rumor-spreading / gossiping

Epidemic protocols: Anti-entropy

Idea

Per round each replica randomly chooses another replica and either

- pulls updates from the contacted replica
- pushes updates to the contacted replica
- push+pull: both replicas update each other (consistency models!)



Epidemic protocols: Gossiping

Idea

In each round each updated (infected) replica contacts *k* replicas

- a replica stops participating (removed) with a probability of *s*/*k*, where *s* is the number of contacted replicas that are already updated, other replica are being updated (infected)
- large *k*: good coverage, large overhead
- small *k*: gossip dies out rather soon

➔ does not ensure eventual consistency

ÖREBRO UNIVERSITY

Consistency protocols

A consistency protocols describes the implementation of a consistency model.

Approaches that are often relevant

- sequential consistency
- eventual consistency

Primary-based protocols

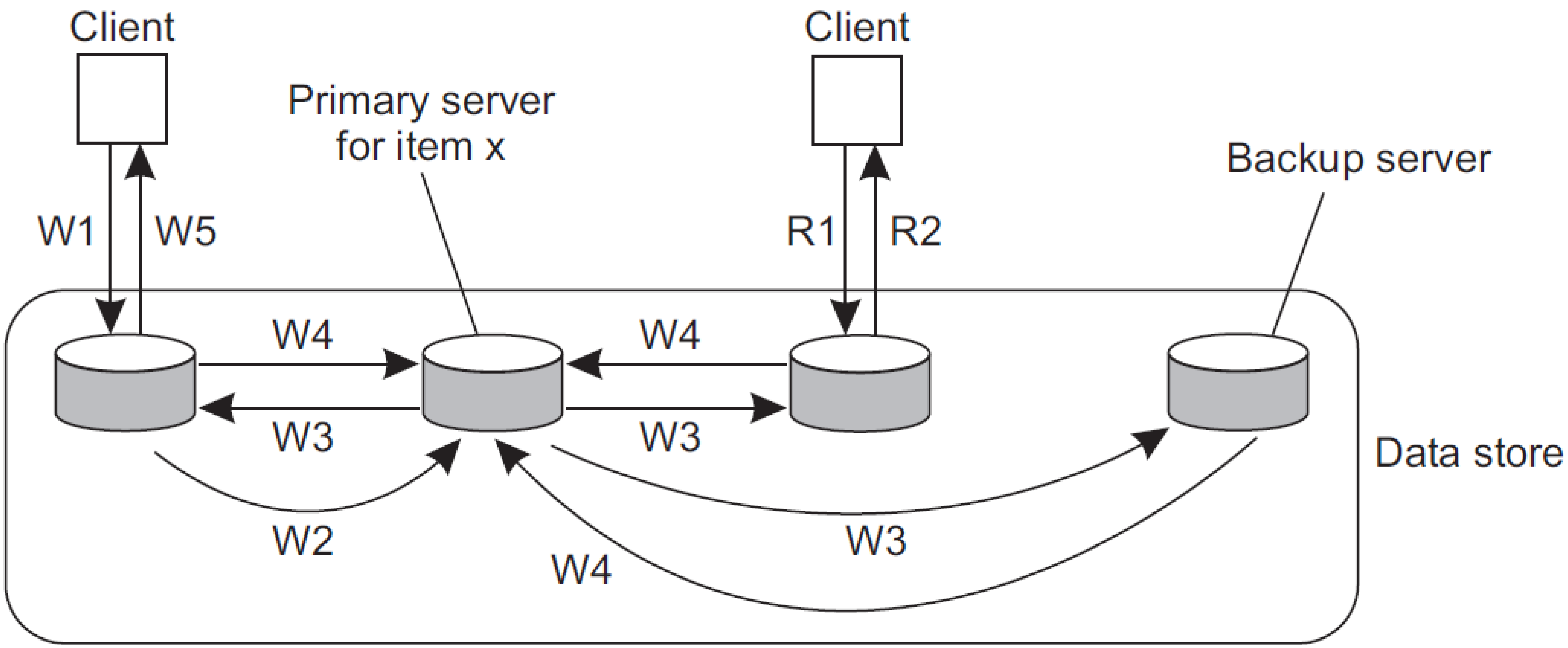
Purpose

Implementing sequential consistency

Idea

One replica acts as coordinator (primary) for all updates to a certain data item

Primary-based protocols



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

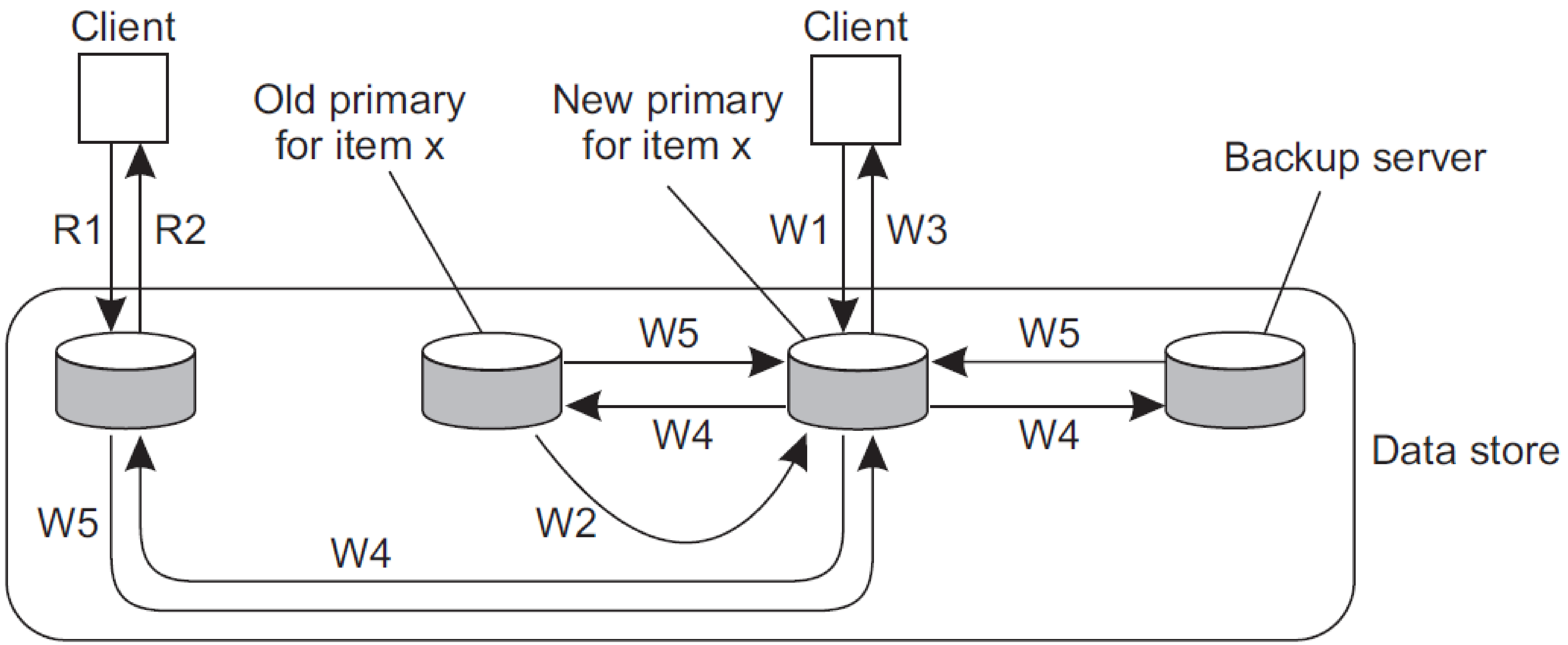
R1. Read request
R2. Response to read

Remote-write

- primary is fixed
- primary enforces global order
- mostly blocking, but non-blocking variants possible

→ Also read-your-writes consistent

Primary-based 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

Local-write

- primary is migrating to the replica that initiated to last write
- non-blocking variant allows a sequence of local writes that are then propagated as a batch

Replicated-write protocols

Idea

Each read or write operation requires permission by a number (quorum) of replicas before execution, subject to the following constraints:

- $N_R + N_W > N$
- $N_W > N/2$

N: number of nodes / replicas

N_R: number of nodes necessary to contact for `read`

N_W: number of nodes necessary to contact for `write`

Replicated-write protocols

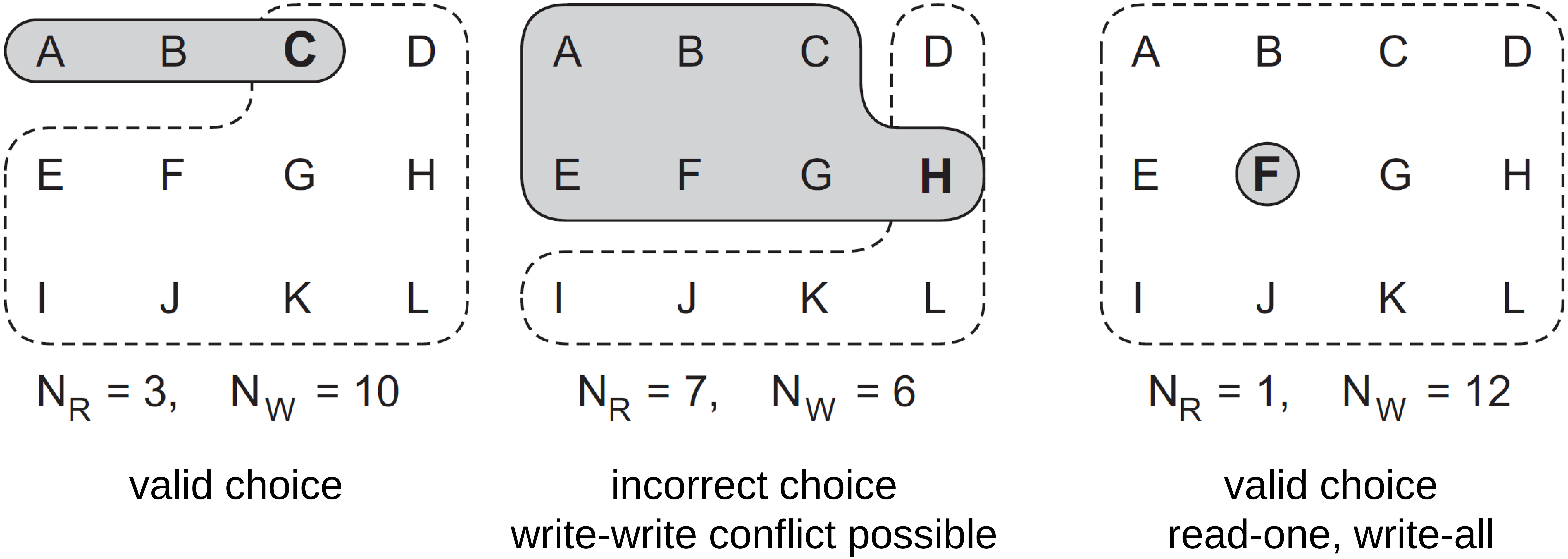
Read

- collect the read quorum
- read from any up-to-date replica (latest time stamp)

Write

- collect the write quorum
- update any out-of-date replicas in the quorum before write
- write on all replicas belonging to the quorum

Replicated-write protocols



Replicated-write protocols

Allows different levels of strictness

- Guaranteed-up-to-date: full quorum
- Limited guarantee: read does not require the full quorum
- Best effort: read/write without a quorum (requires another form of consistency checks)

Possibility to combine quorum-based methods with locks to implement a sequence/transaction mechanism

ÖREBRO UNIVERSITY

Cache-coherence protocols

Combination of previously discussed protocols (often primary-based) and results from computer architectures dealing with

- coherence detection
- coherence enforcement

Summary

- Replication
- Consistency models and CAP theorem
- Distribution protocols
- Consistency protocols