Distributed Storage Fundamentals

Idit Keidar, Technion

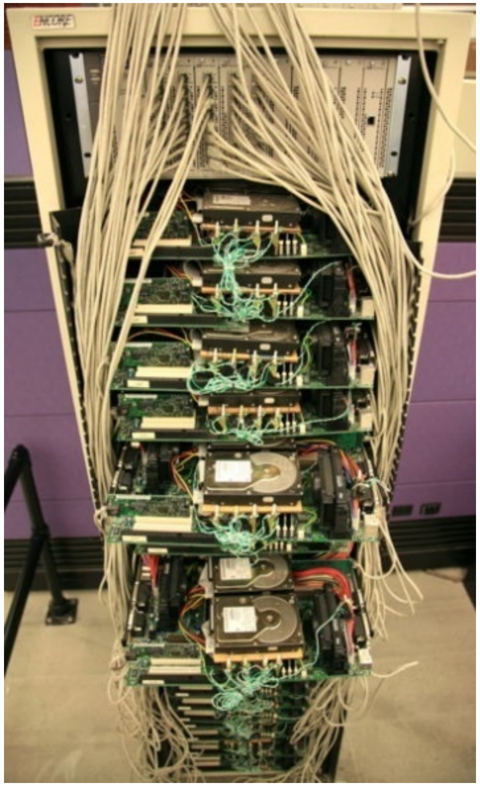
Where Are Your Files?

Where's your data?

Distributed Storage

- Scales
- Cost-effective: can even be made up of many cheap, low-reliability storage nodes
- Provides reliability via redundancy

Google's 1st server



Failures Happen

- Nodes (storage/compute) crash
  - Sometimes recover
- Processes are unresponsive (asynchronous)
  - E.g., due to GC stalls
- Networks delay/drop messages (async., lossy)
  - Buffer overflows, config errors, bad NICs
- Networks go down for periods
  - Routing loops, router failures, net maintenance

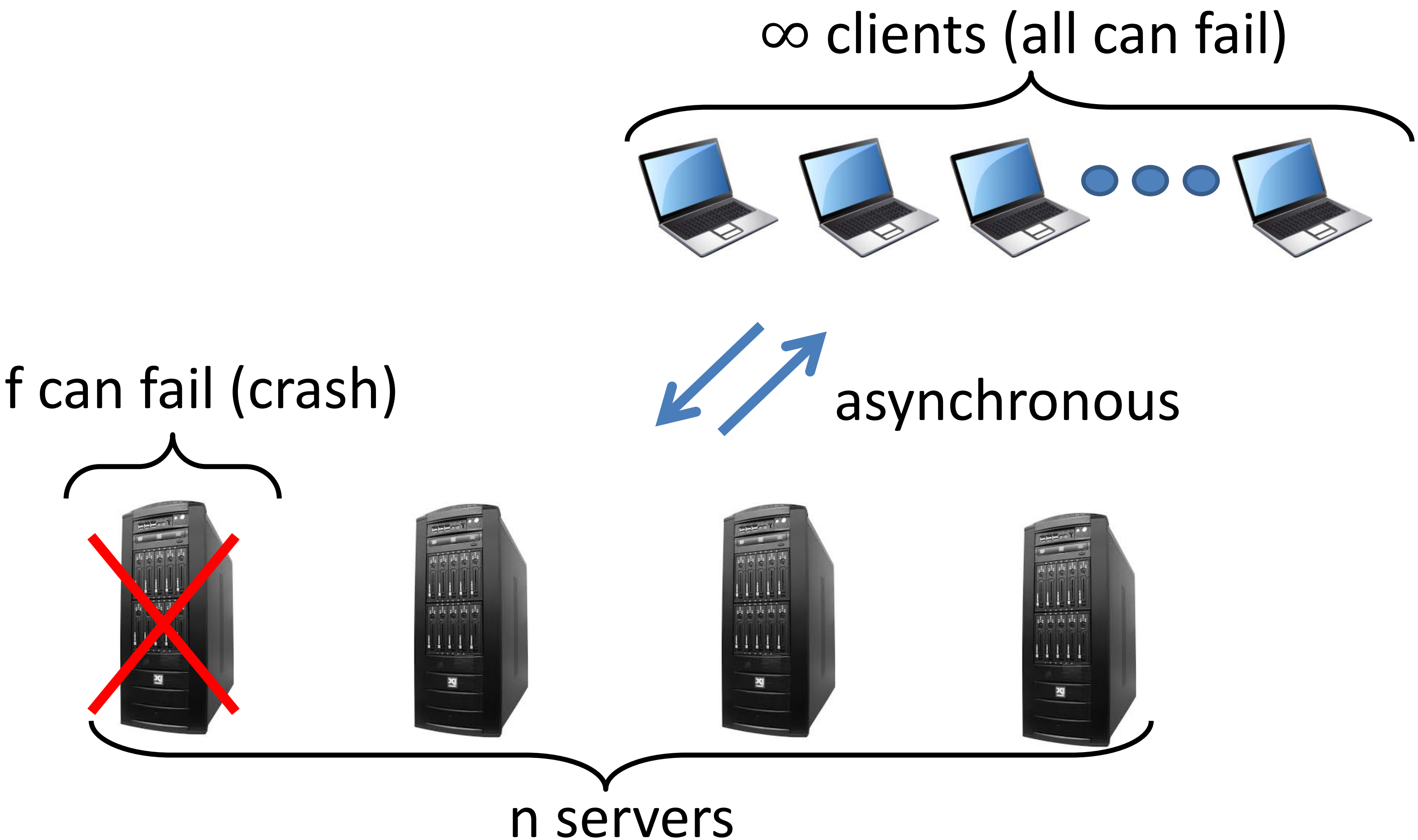
Anecdotal Evidence

- Microsoft: 40.8 link failures/day
  - 5 min to one week long
  - Path redundancy reduces loss by 43%
- Google: in cluster's 1^{st} year
  - 5 racks see 50% packet loss
  - 8 net maintenance/year, 30 min loss in 4
  - 3 router failures/year
- Companies report partition post-mortems
  - Netflix, Github, AWS, ….
  - Resulted in split brain

Asynchrony

- Unresponsive node indistinguishable from crashed one
  - Timeout without making sure it's dead
- Delays indistinguishable from drops
- Perfect failure detection impossible
  - "false suspicions" inevitable

Fault-Tolerant Distributed Storage Model



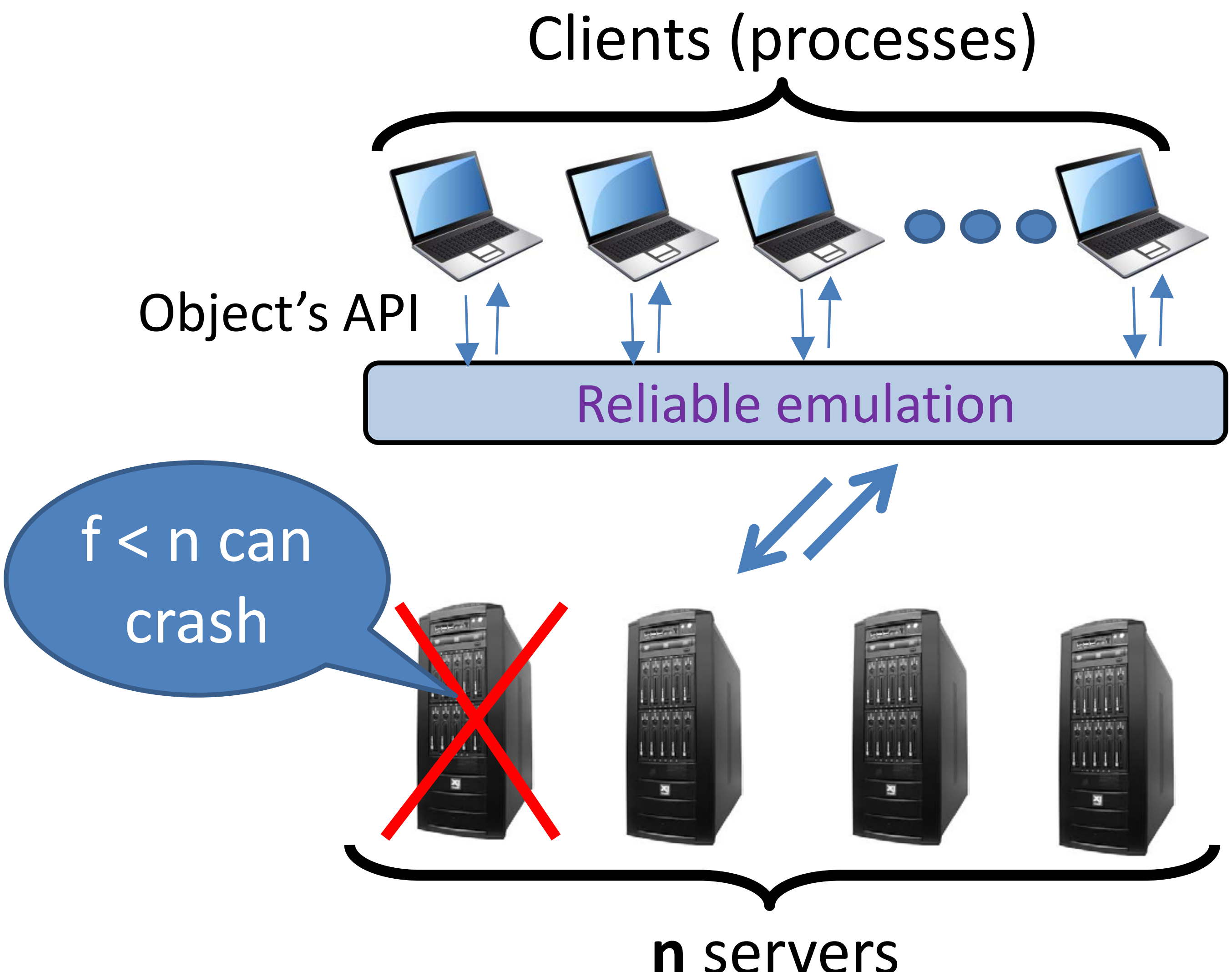
Fault-Tolerance 101

- Replication
  - Multiple copies (e.g., 3) of each data item
  - Copies on distinct storage nodes
- Disaster recovery
  - Copies geographically dispersed

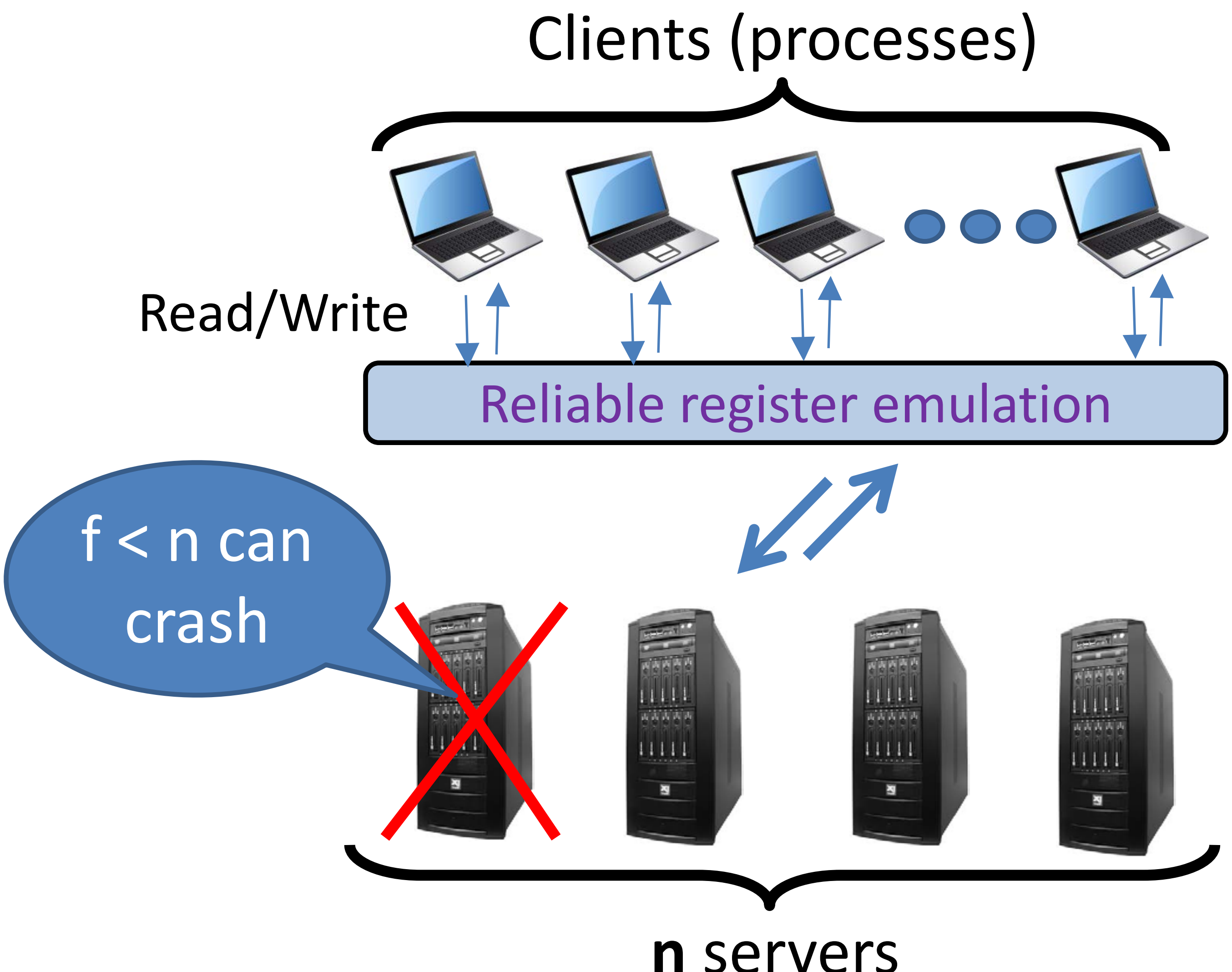
Emulating Shared Memory

- Can we provide the illusion of reliable atomic shared-memory in a message-passing system?
- In an asynchronous system?
- Where clients and servers can fail?

Shared Memory Emulation



Simple Read/Write Emulation



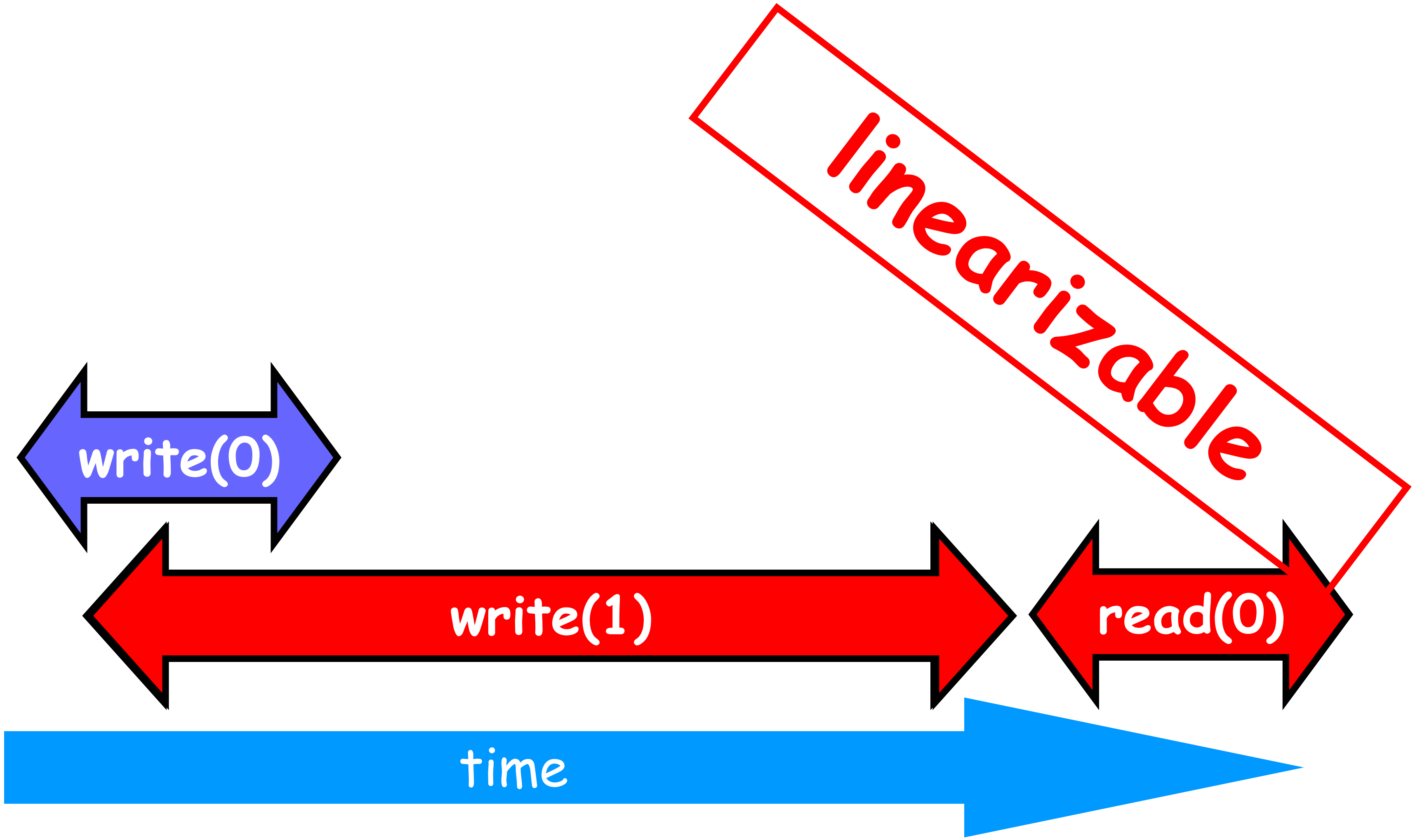
Register

- Holds a value
- Can be read
- Can be written
- Interface:
  - int read(); // returns last written value
  - void write(int v); // returns ack

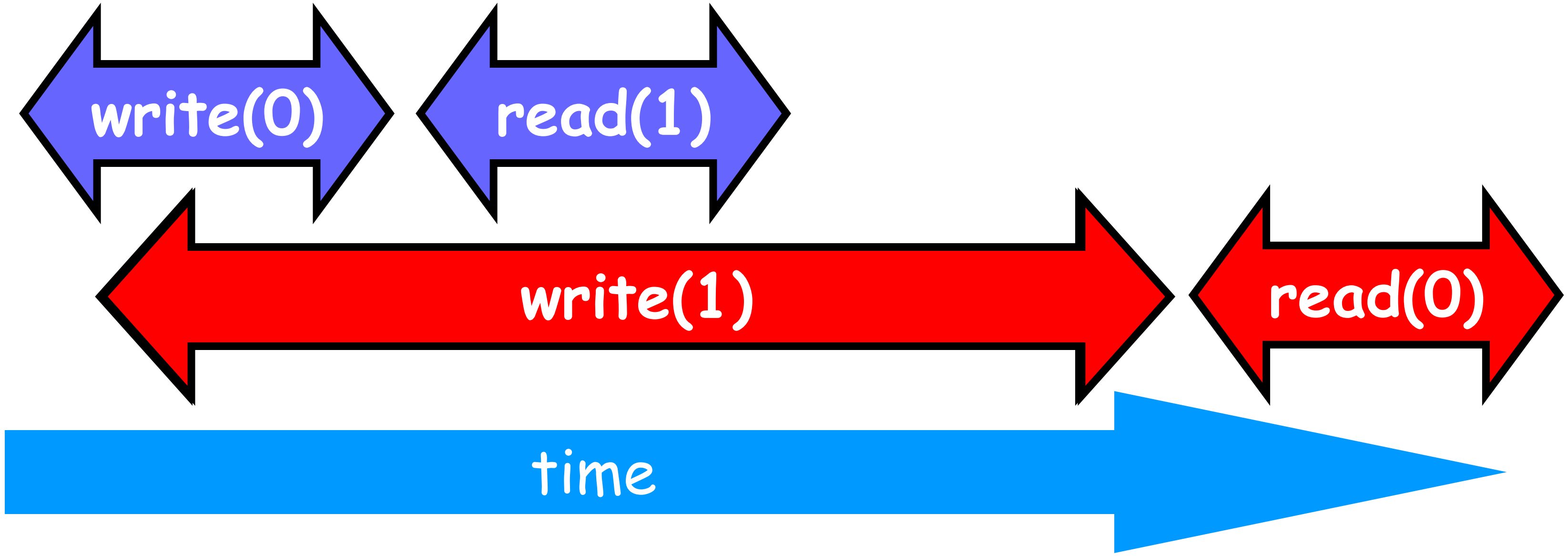
Atomic (Linearizable) Register

- Each API call should –
  - “Take effect”
    - Effect defined by the sequential specification
  - Instantaneously
    - Take 0 time
  - Between its invocation and response
    - Real-time order
    - A pending call (invocation and no response) can either occur after its invocation or not at all

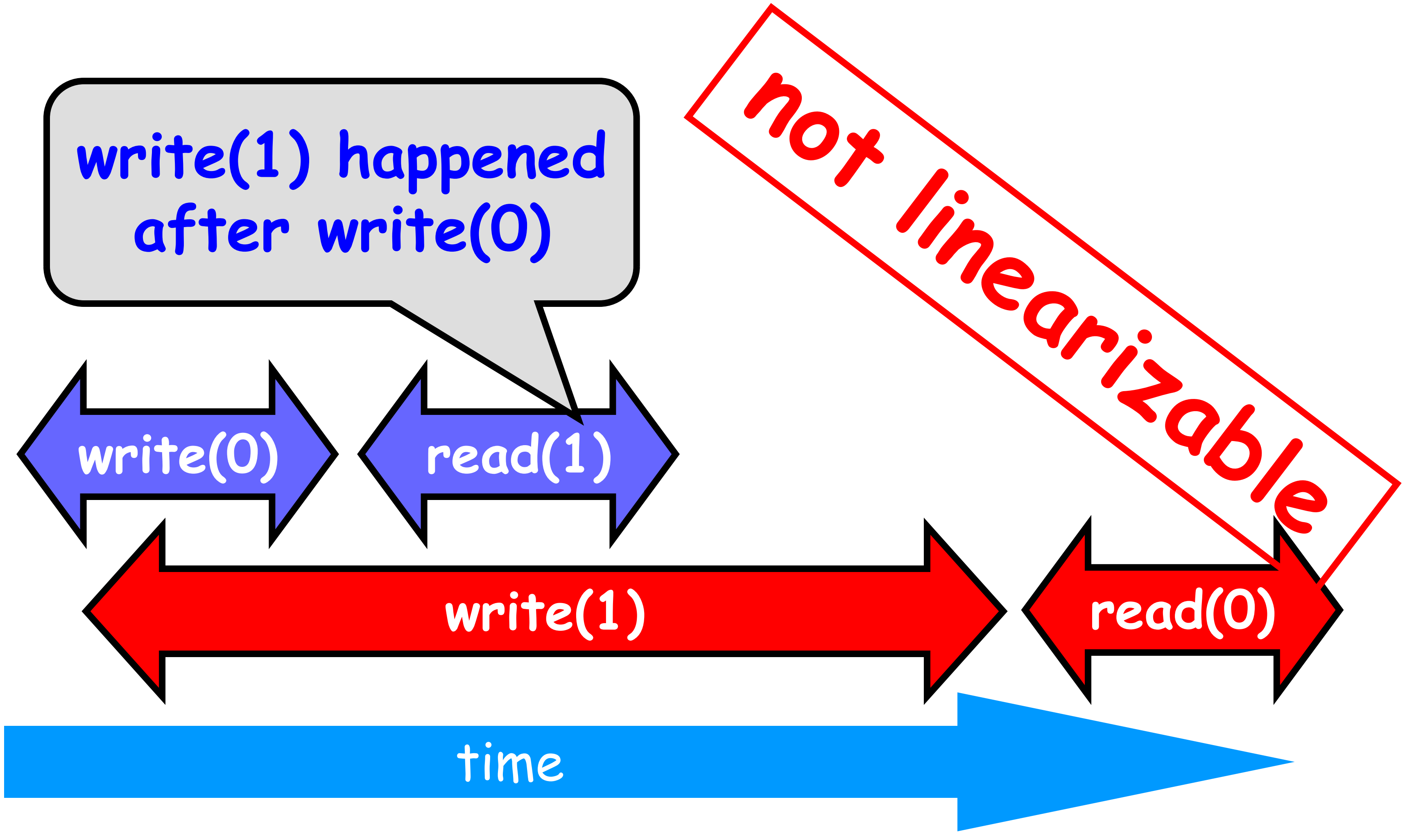
Example 1



Example 2



Example 2



write(1) happened after write(0)
not linearizable
write(0)
read(1)
write(1)
read(0)
time

Liveness: Wait-Freedom

- Wait-free
  - Every operation by a correct process *p* eventually completes
  - In a finite number of *p*'s steps
  - Regardless of steps taken/not taken by other processes

Emulating A Register

- Can we emulate a wait-free atomic shared register?
- In an asynchronous system?
- Where clients and servers can fail?

Take I: Failure-Free Case

(No server failures)

- Each server keeps a local copy of the register
- Let’s try state machine replication
- Using atomic broadcast:
  - broadcast(m)
  - deliver(m)
  - Messages are delivered in the same order at all servers

Emulation with Atomic Broadcast (Failure-Free)

- Upon client request (read/write)
  - Broadcast the request
- Upon deliver write request
  - Write to local copy of register
  - If from local client, return ack to client
- Upon deliver read request
  - If from local client, return local register value to client

linearizable

What If Processes Can Crash?

- Does the same solution work?

- FLP says: no consensus/state machine replication
  - In asynchronous network
  - With crash failures
  - But consensus with eventual synchrony/failure detectors possible (Paxos, ZooKeeper, Raft)

Take II: 1-Reader 1-Writer (SRSW)

- Single-reader – there is only one process that can read from the register
- Single-writer – there is only one process that can write to the register
- The reader and writer are just 2 processes
  - The other *n-2* processes are there to help

For simplicity, we eliminate the distinction between clients and servers for now

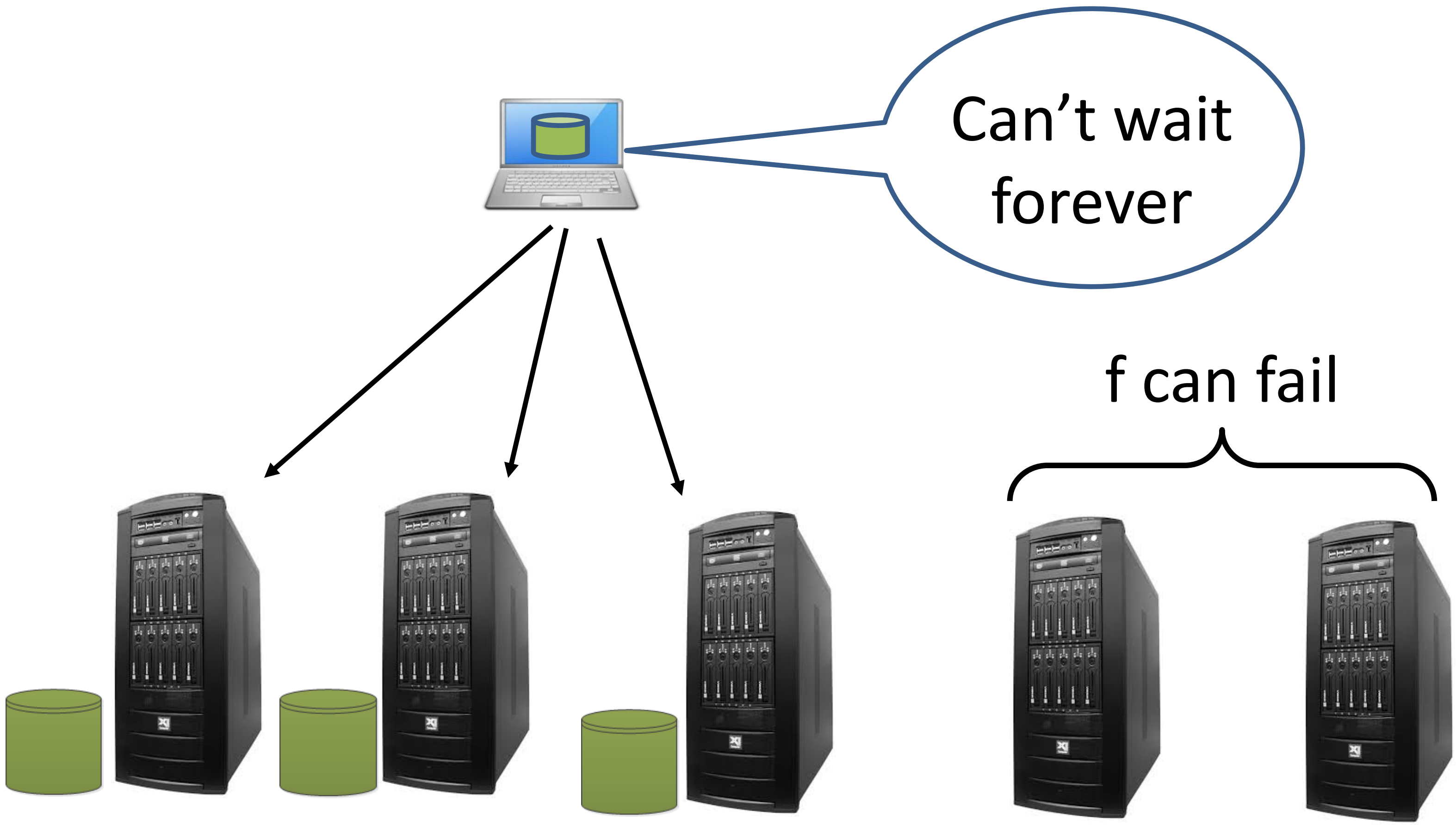
Trivial Solution?

- Writer simply sends message to reader
  - When does it return ack?
  - What about failures?
- We want a *wait-free* solution:
  - If the reader (writer) fails, the writer (reader) should be able to continue writing (reading)

ABD: Fault-Tolerant Emulation
[Attiya, Bar-Noy, Dolev 95]

- Assumes up to $f<n/2$ processes can fail
- Main ideas:
  - Store value at majority of processes before write completes
  - read from majority
  - read intersects write, hence sees latest value

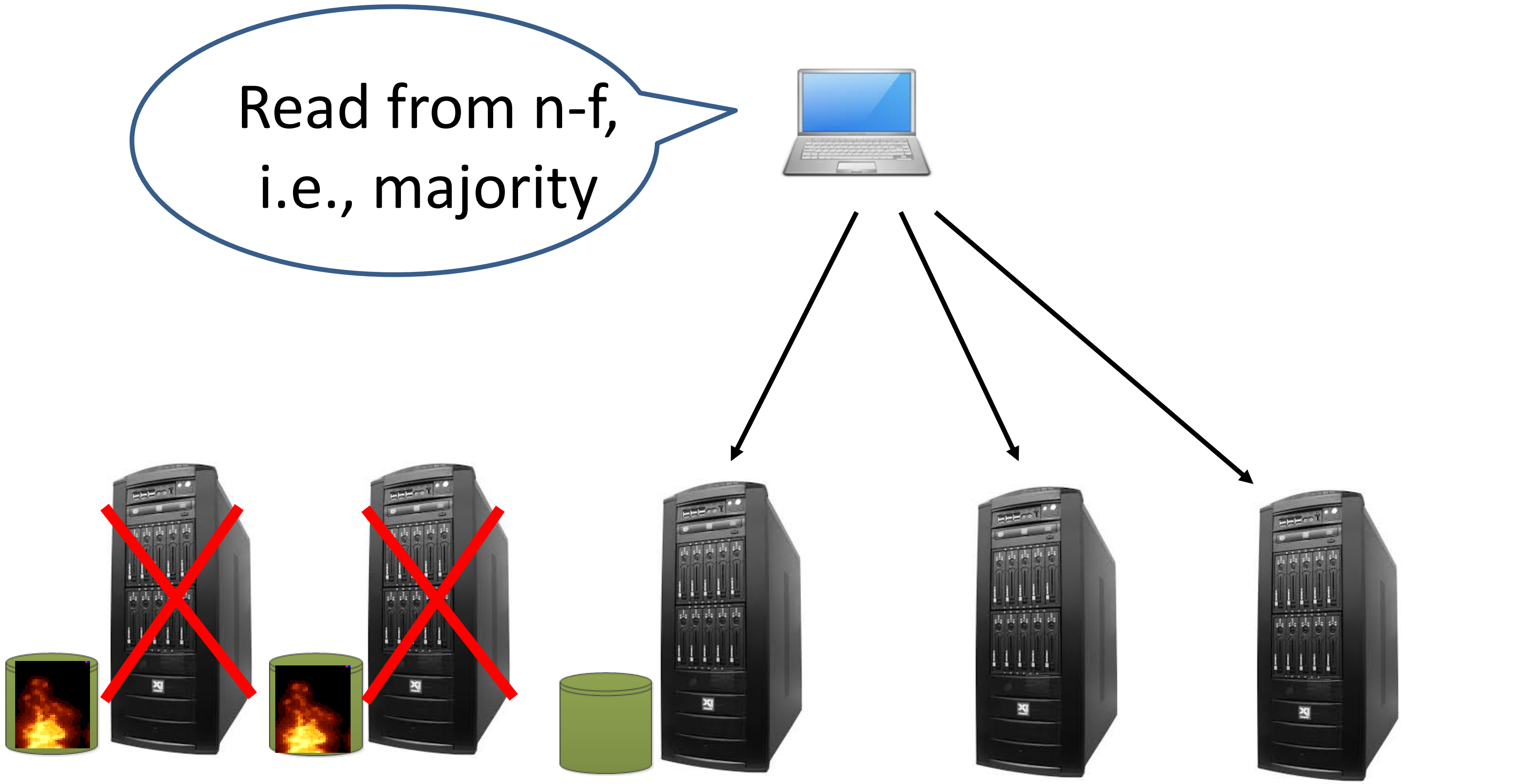
Example: Reliable Storage Emulation



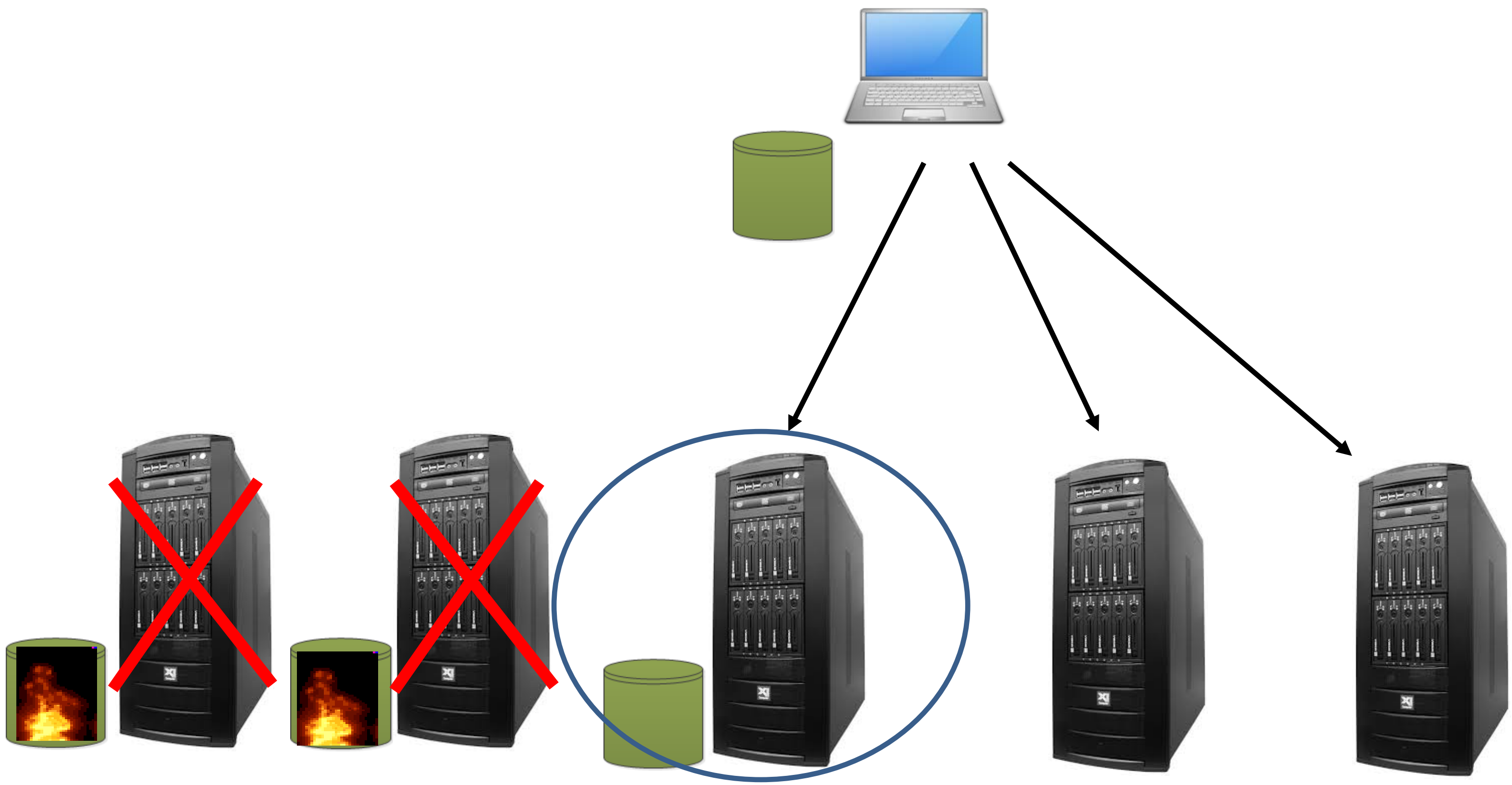
Example: Reliable Storage Emulation

Example: Reliable Storage Emulation

Example: Reliable Storage Emulation



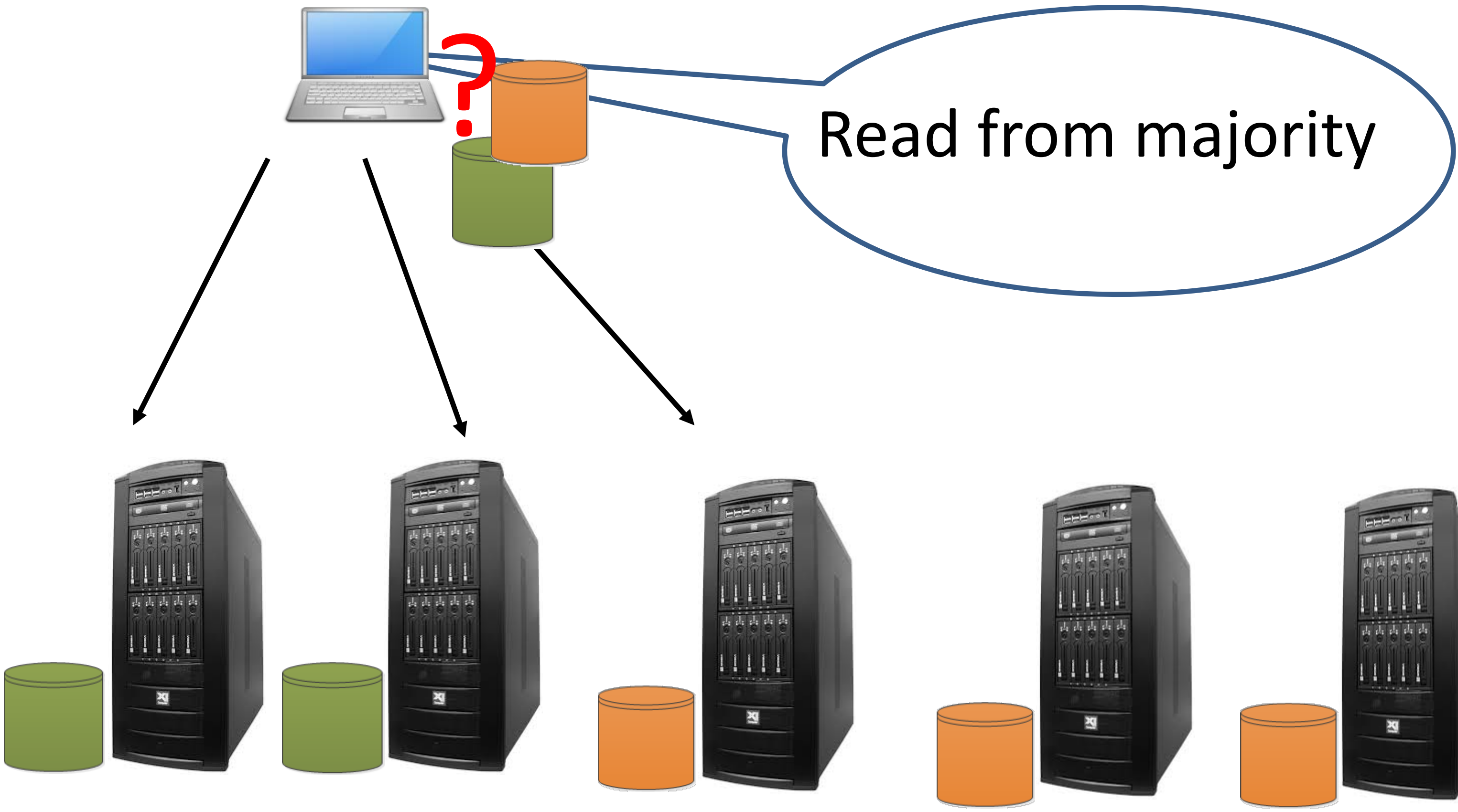
Example: Reliable Storage Emulation



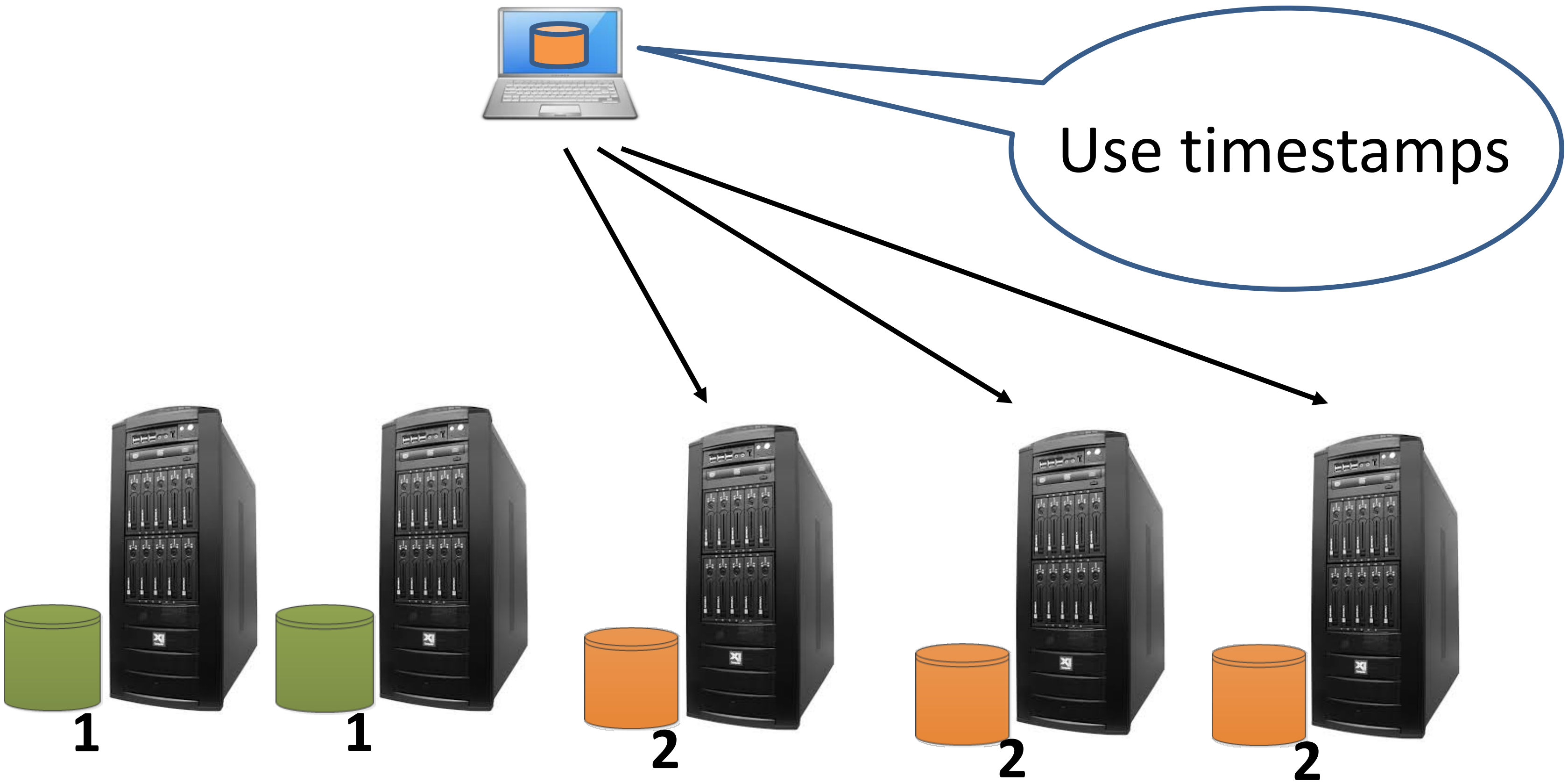
Every two majorities intersect

Example: Reliable Storage Emulation

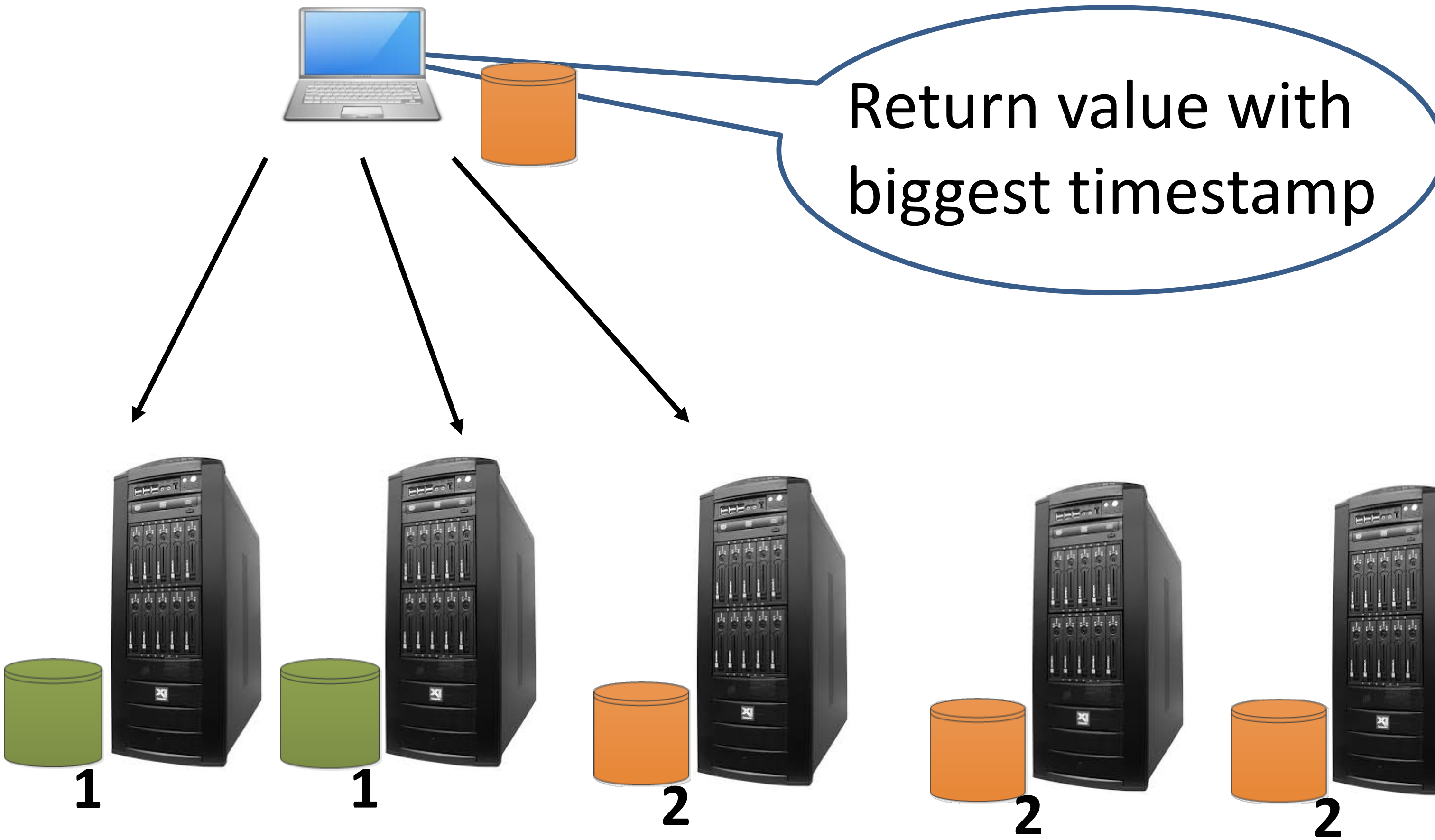
Example: Reliable Storage Emulation



Example: Reliable Storage Emulation



Example: Reliable Storage Emulation



SRSW Algorithm: Variables

- At each process:
  - *x,* a copy of the register
  - *t,* initially 0, unique tag associated with latest write

SRSW Algorithm: Write

- write(x,v)
  - choose tag > t
  - set x ← v; t ← tag
  - send (“write”, v, t) to all
- Upon receive (“write”, v, tag)
  - if (tag > t) then set x ← v; t ← tag fi
  - send (“ack”, v, tag) to writer
- When writer receives (“ack”, v, t) from majority (counting an ack from itself too)
  - return ack to client

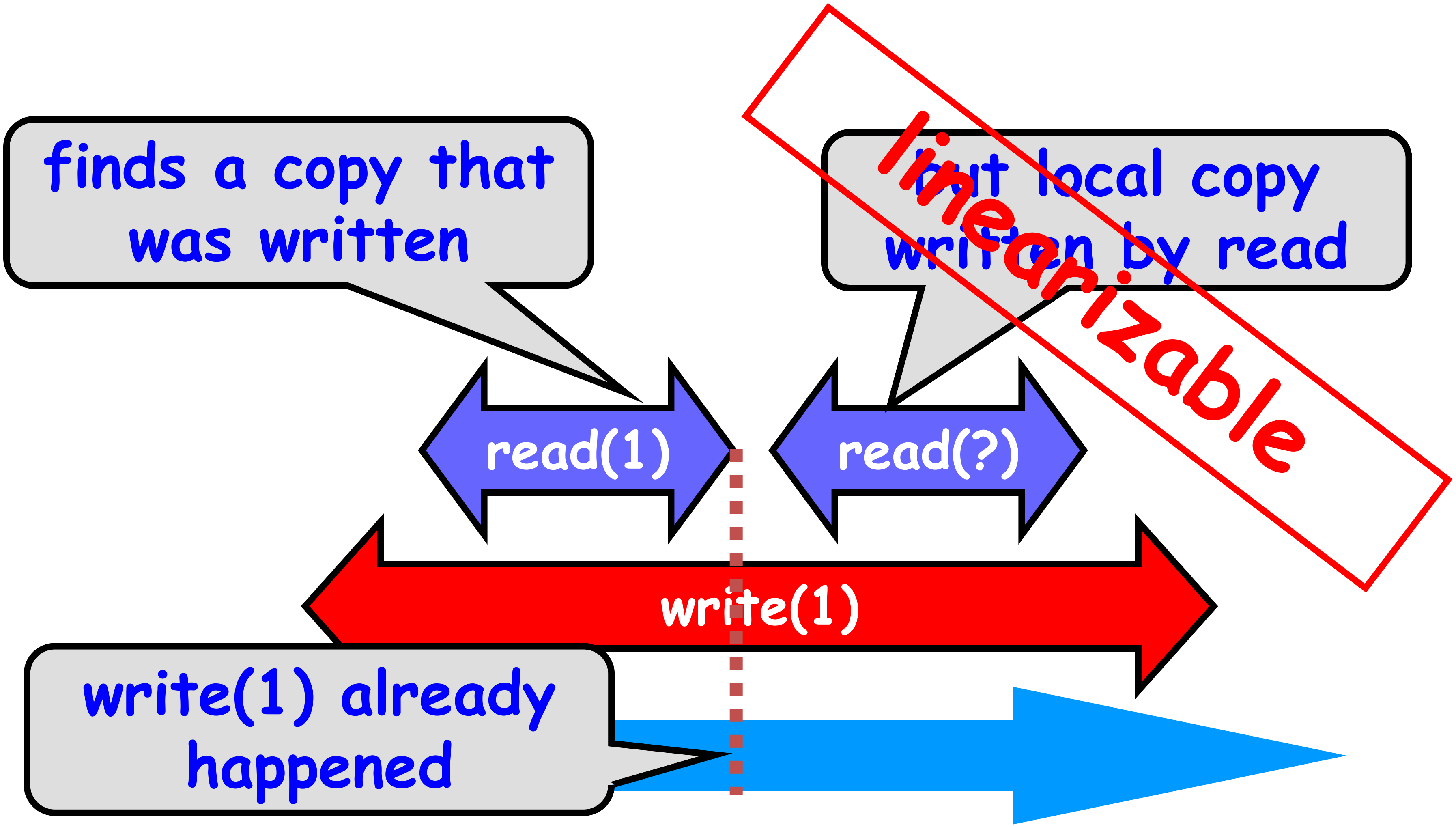
SRSW Algorithm: Read

- read(x,v)
  - send (“read”) to all
- Upon receive (“read”)
  - send (“read-ack”, x, t) to reader
- When reader receives (“read-ack”, v, tag) from majority (including local values of x and t)
  - choose value v associated with largest tag
  - store these values in x,t
  - return x

Does This Work?

- Only possible overlap is between read and write
  - why?
- When a read does not overlap any write –
  - It reads at least one copy that was written by the latest write (why?)
  - This copy has the highest *tag* (why?)
- What is the linearization order when there is overlap between read and write?
- What if 2 reads overlap the same write?

Example


finds a copy that was written
but local copy written by read
linearizable
read(1)
read(?)
write(1)
write(1) already happened

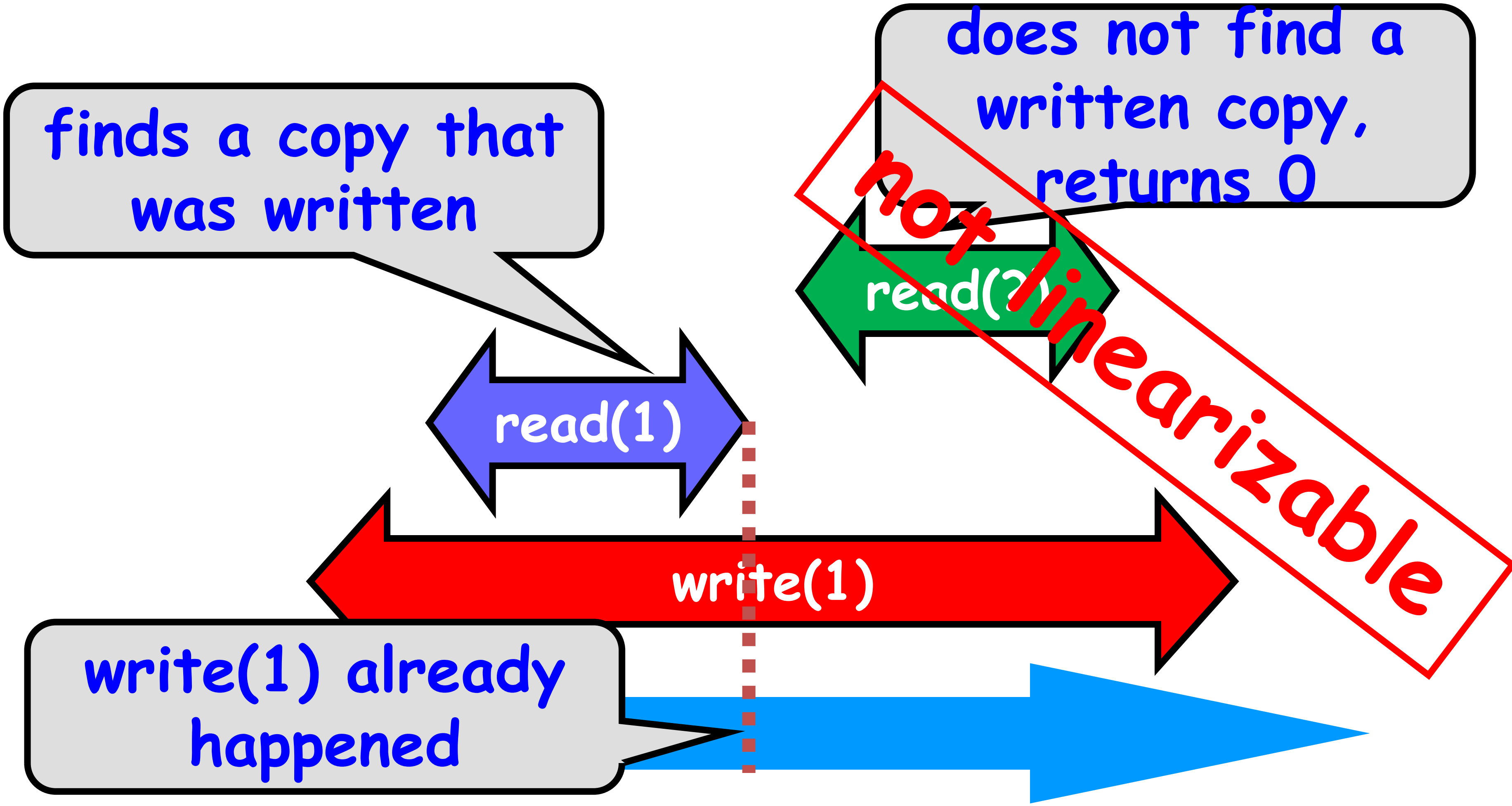
Wait-Freedom

- Only waiting is for majority of responses
- There is a correct majority
- All correct processes respond to all requests
  - Respond even if the tag is smaller

Take III: n-Reader 1-Writer (MRSW)

- n-reader – all the processes can read
- Does the previous solution work?
- What if 2 reads by *different processes* overlap the same write?

Example

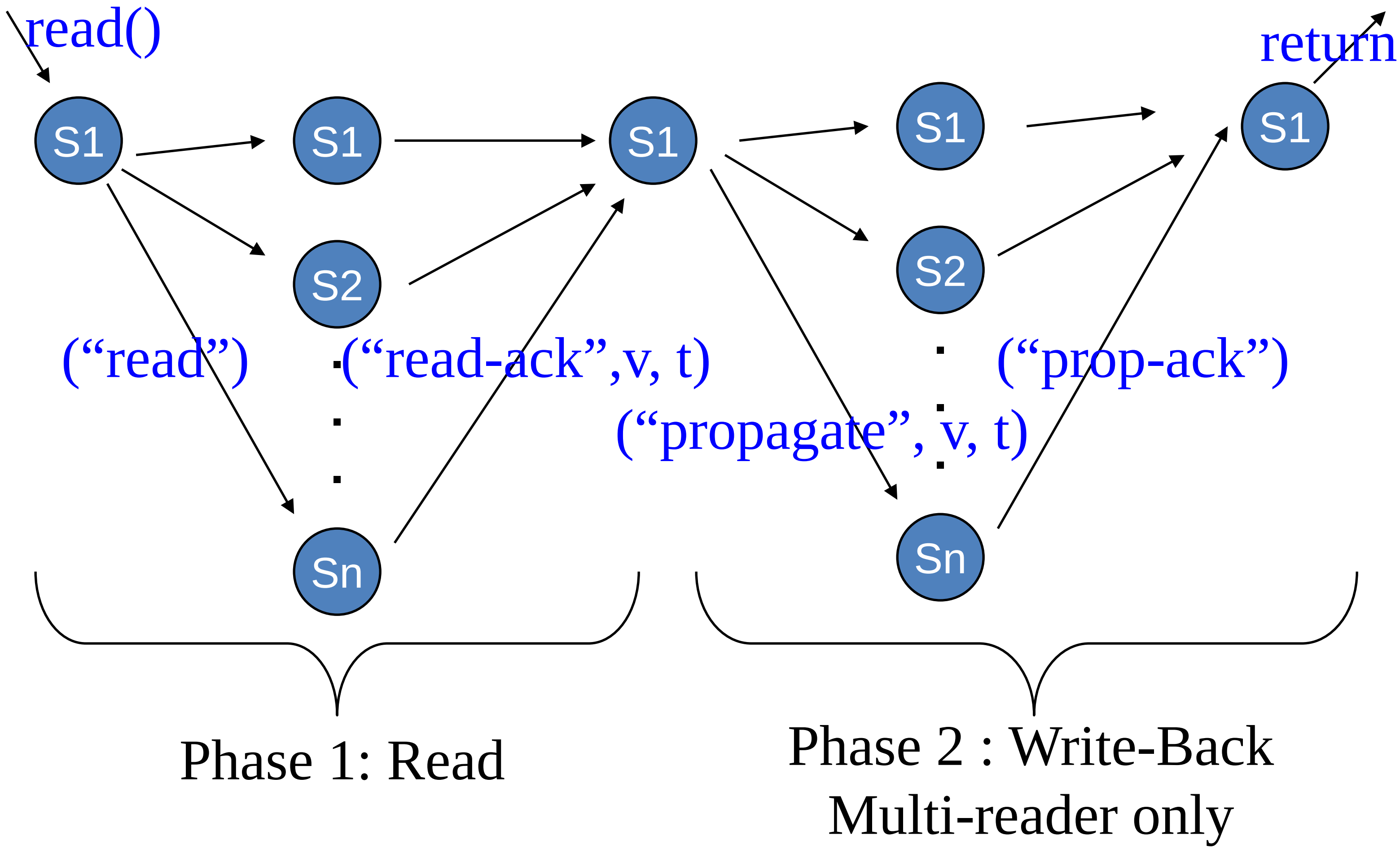


finds a copy that was written
does not find a written copy, returns 0
read(?)
read(1)
not linearizable
write(1)
write(1) already happened

MRSW Algorithm
Extending the Read

- When reader receives (“read-ack”, *v, tag*) from majority
  - choose value *v* associated with largest *tag*
  - store these values in *x,t*
  - send (“propagate”, *x, t*) to all (except writer)
- Upon receive (“propagate”, *v, tag*) from process *i*
  - **if** (*tag* > *t*) **then** set $x \leftarrow v$; $t \leftarrow tag$ **fi**
  - send (“prop-ack”, *x, t*) to process *i*
- When reader receives (“prop-ack”, *v, tag*) from majority (including itself)
  - return *x*

The Complete Read



Take IV: n-Reader n-Writer (MRMW)

- n-writer – all the processes can write to the register
- Does the previous solution work?

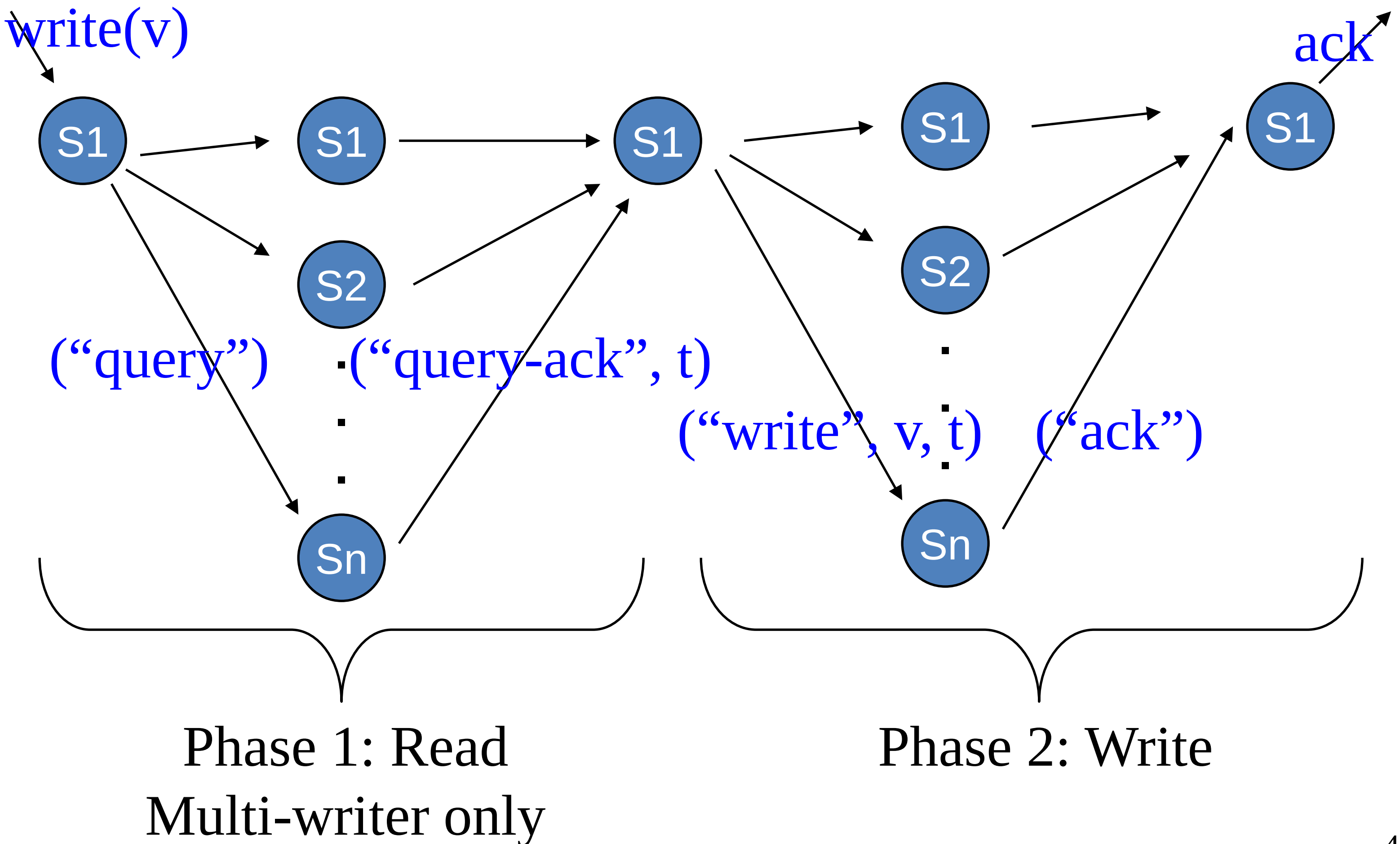
Playing Tag

- What if two writers use the same tag for writing different values?
- Need to ensure *unique* tags
  - That's easy: break ties, e.g., by process id
- What if a later write uses a smaller tag than an earlier one?
  - Must be prevented (why?)

MRMW Algorithm
Extending the Write

- To perform write(x,v)
  – send ("query") to all
- Upon receive ("query") from *i*
  – send ("query-ack", *t*) to *i*
- When writer receives ("query-ack", *tag*) from majority (counting its own tag)
  – choose unique *tag* > all received *tags*
  – continue as in 1-writer algorithm
- What if another writer chooses a higher tag before write completes?

The Complete Write



Can We Emulate *Every* Atomic Object the Same Way?

- We only emulated a read/write object
- Think of a general object type, with some data members and some methods
  - Queue, stack, counter, …

- Can we support it the same way?

R/W Registers vs. Consensus

- ABD works even if the system is completely asynchronous
- In consensus (e.g., Paxos), there is no progress when there are multiple leaders
- Here, there is always progress – multiple writers can write concurrently
  - One will prevail (which?)

Disk Paxos

Consensus in Shared Memory

- A shared object supporting a method $\text{decide}_i(v_i)$ returning a value d_i
- Satisfying:
  - Agreement: for all i and j $d_i = d_j$
  - Validity: $d_i = v_j$ for some j
  - Termination: decide returns

Solving Consensus in/with Shared Memory

- Assume asynchronous shared memory system with *atomic* R/W registers
- Can we solve consensus?
  - Consensus is *not* solvable if even one process can fail (shared-memory version of [FLP])
  - Yes, if no process can fail
  - Yes, with eventual synchrony or failure detectors

Shared Memory (SM) Paxos

- Consensus
  - In asynchronous shared memory
  - Using wait-free regular R/W registers
    - As emulated by ABD
  - And leader-election failure detector Ω
- Wait-free
  - *Any* number of processes may fail ($t < n$)
    - Unlike message-passing model

Leader Election Failure Detector

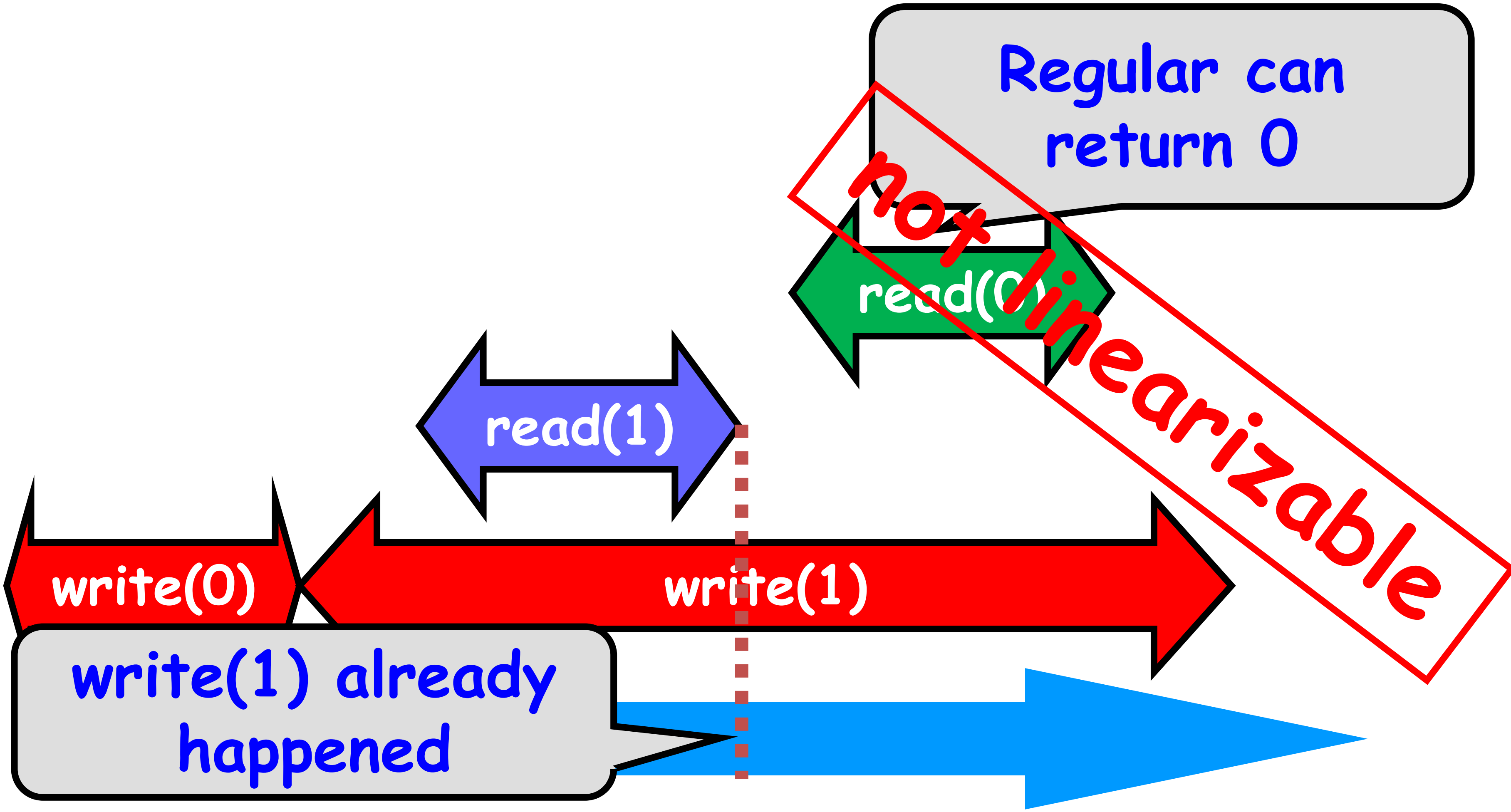
- Ω – Leader
  - Outputs one trusted process
  - *Stable* from some point on:
    All correct procs. trust the same correct proc.

- Is the weakest for consensus
  [Chandra, Hadzilacos, Toueg 96]

Regular Registers

- SM Paxos can use registers that provide weaker semantics than atomicity
- SWMR *regular register*: a read returns
  - Either a value written by an overlapping write **or**
  - The register's value before the first write that overlaps the read

Regular versus Atomic



Variables

- Paxos variables are:
  - BallotNum, AcceptVal, AcceptNum
- SM version uses *shared* regular registers:
  - $x_i = \langle$ bal, val, num, decision $\rangle_i$ for each process i
  - Initially $\langle \langle 0,0 \rangle, \perp, \langle 0,0 \rangle, \perp \rangle$
  - Writeable by i, readable by all (SWMR)
- Each process keeps *local* variables b, v, n
  - Initially $\langle \langle 0,0 \rangle, \perp, \langle 0,0 \rangle \rangle$

SM Paxos: Phase I

if leader (by Ω) **then**

b $\leftarrow$ choose new unique ballot

write $\langle$ b, v, n, $\perp$ $\rangle$ to x_i

read all x_j's

if some x_j.bal > b **then** start over

if all read x_j.val's = $\perp$ **then**

v $\leftarrow$ initial value

else v $\leftarrow$ read val with highest num

Only b changed in this phase

Write is like sending to all

Read instead of waiting for acks

No ack: someone moved on!

Phase I Summary

- Classical Paxos:
  - Leader chooses new ballot, sends to all
  - Others ack if they did not move on to a later ballot
  - If no majority, try again
  - Otherwise, move to Phase 2
- SM Paxos:
  - Leader chooses new ballot, writes its variable
  - Leader reads to check if anyone moved on to a later ballot
  - If any one moved on, try again
  - Otherwise, move to Phase 2

SM Paxos: Phase II
Leader Cont'd

$n \leftarrow b$

write $\langle b, v, n, \perp \rangle$ to x_i

Like sending "accept" to all

v,n changed in this phase

read all x_j's

if some $x_j.bal > b$ **then** start over

Read to see if all would have accepted this proposal

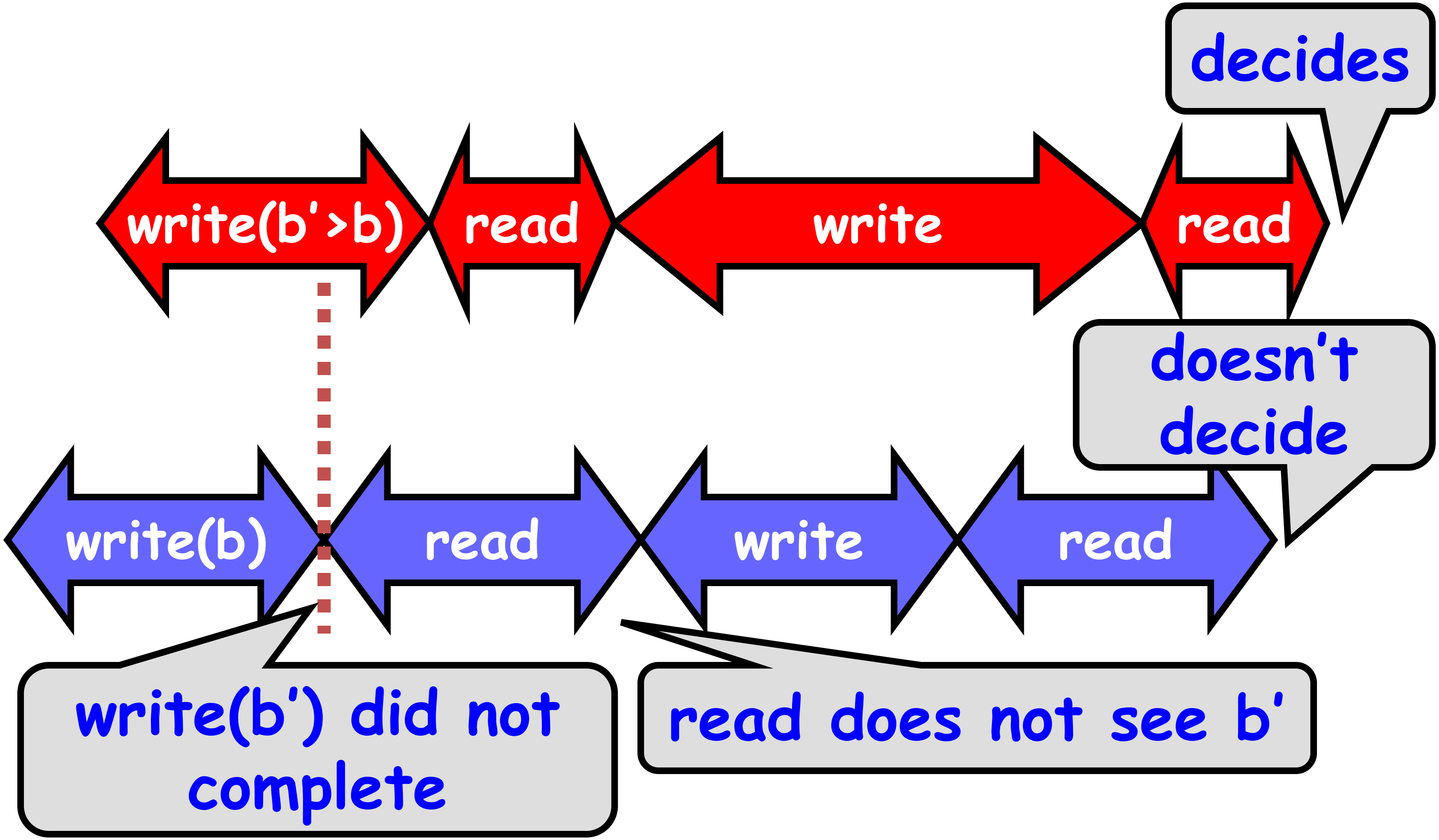
write $\langle b, v, n, v \rangle$ to x_i

return v

When don't they?

Decide

Why Read Twice?



Adding The Non-Leader Code

start over means go here

while (true)

if leader (by Ω) **then**

[leader code from previous slides]

else

read x_{ld} ,were ld is leader

if x_{ld}.decision $\neq \perp$ **then**

return x_{ld}.decision

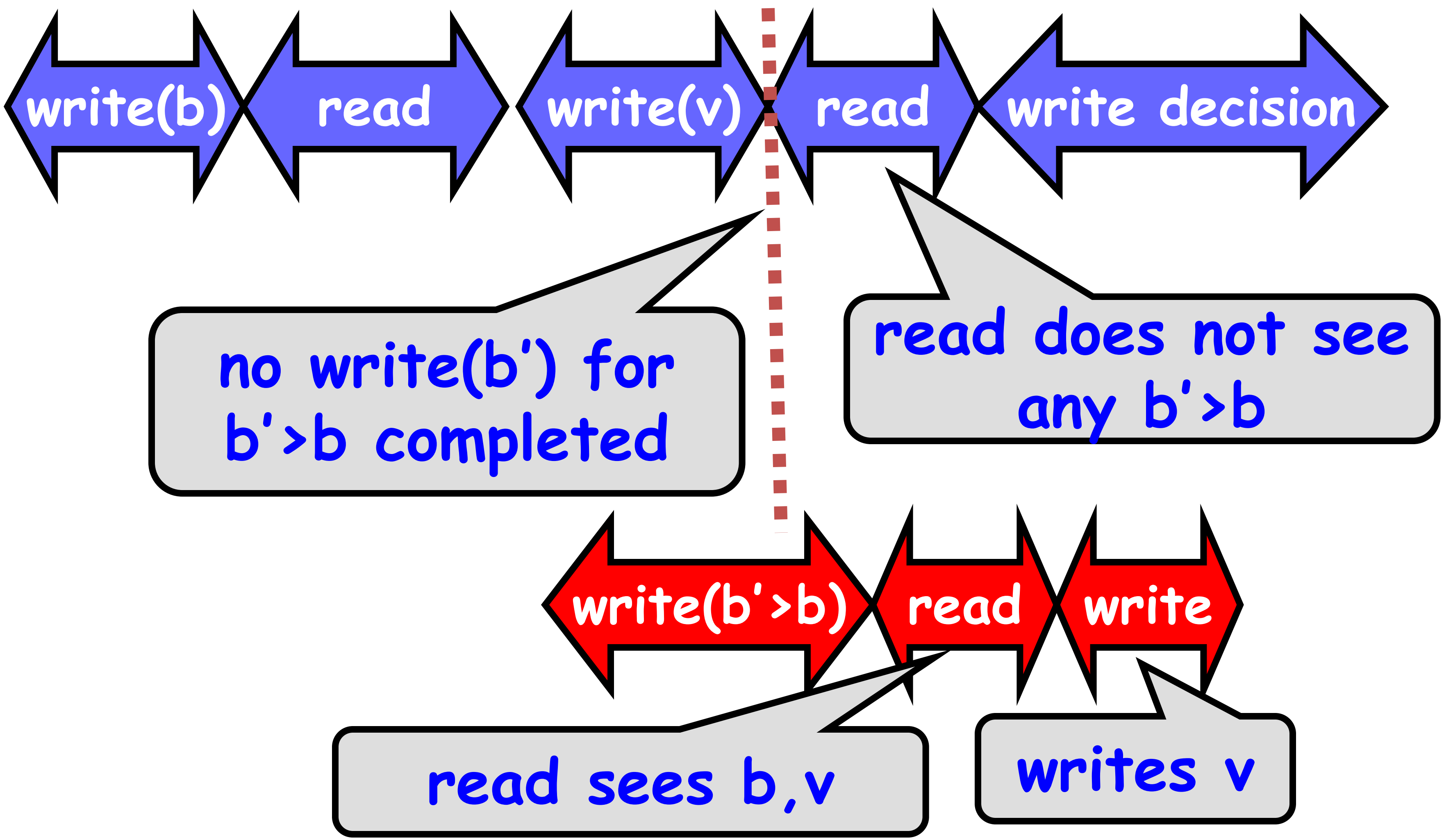
Liveness

- The shared memory is reliable
- The non-leaders don’t write
  - They don’t even need to be “around”
- The leader only fails if there is contention
  - Another leader competes with it
  - By Ω, eventually only one leader will compete
  - In shared memory systems, Ω is called a *contention manager*

Validity

- By induction
- Leader always proposes its own value or one previously proposed by an earlier leader
  – Regular registers suffice

Agreement "By Example"



Agreement Proof Idea

- Look at lowest ballot, b, in which some process decides v
- By uniqueness of b, no v’≠ v is decided with b
- Prove by induction that every decision with b’>b is v

Termination

- When one correct leader exists
  - It eventually chooses a higher b than all those written before
  - No other process writes a higher ballot
  - So it does not start over, and hence decides
- *Any number of processes can fail*

Optimization

- The first write (of b) does not write consensus values
- A leader running multiple consensus instances can perform the first write once and for all and then perform only the second write for each consensus instance

Leases

- We need eventually accurate leader (Ω)
  - But what does this mean in shared memory?
- We would like to have mutual exclusion
  - Not fault-tolerant!
- Lease: fault-tolerant, time-based mutual exclusion
  - Live but not safe in eventual synchrony model

Using Leases

- A client that has something to write tries to obtain the lease
  - Lease holder = leader
  - May fail…
- Example implementation:
  - Upon failure, *backoff* period
- Leases have limited duration, expire

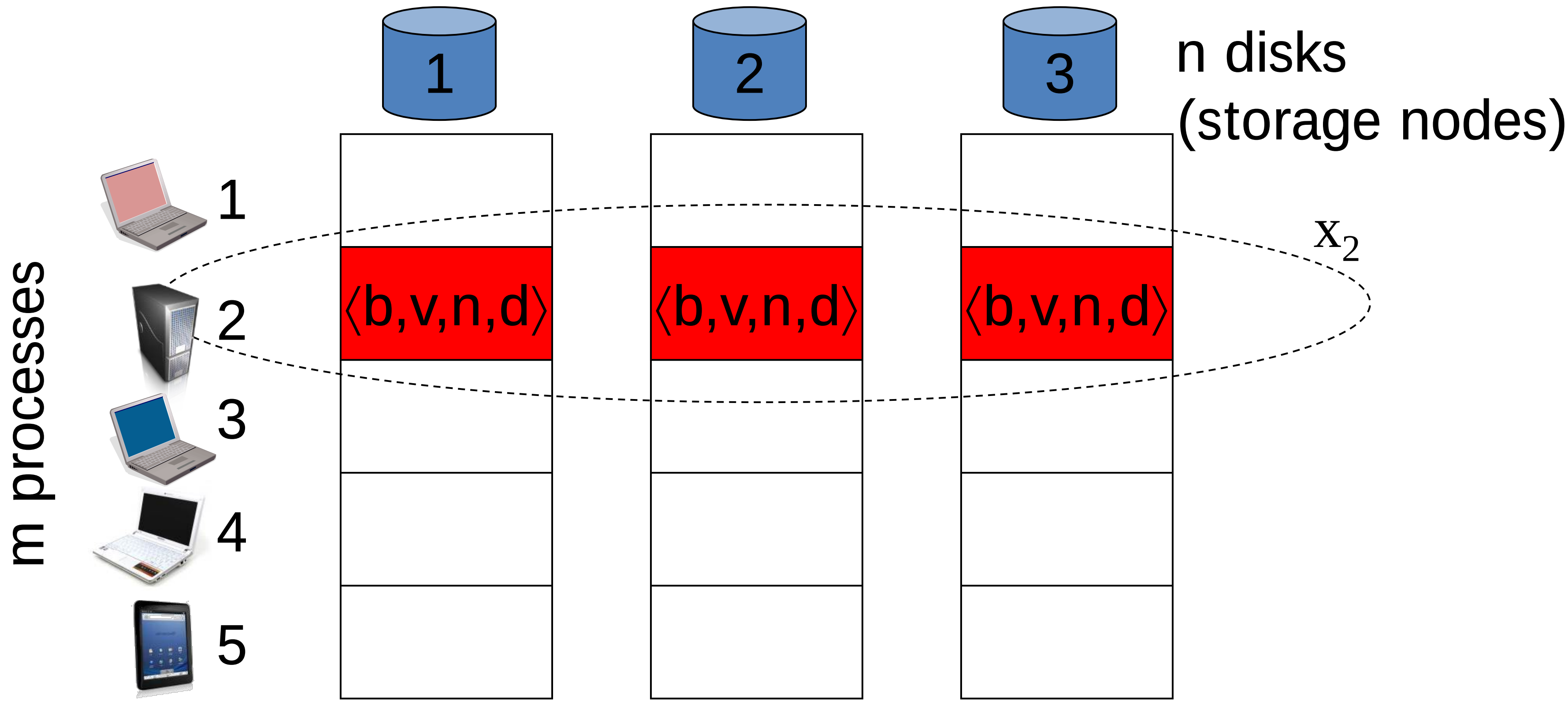
Lock versus Lease

Lock	Lease
Blocking • Using locks is not wait-free • If lock holder fails, we're in trouble	Non-blocking • Expires regardless whether holder fails
Always safe • Never two lock-holders	Unsafe • Two lease-holders possible due to asynchrony • OK for algorithms like Paxos

Disk Paxos

- Consensus using $n \geq 2t+1$ fault-prone disks
  - Disks can incur crash failures
- Solution combines:
  - *m*-process shared memory Paxos and
  - ABD-like emulation of shared registers from fault-prone ones

Disk Paxos Data Structures

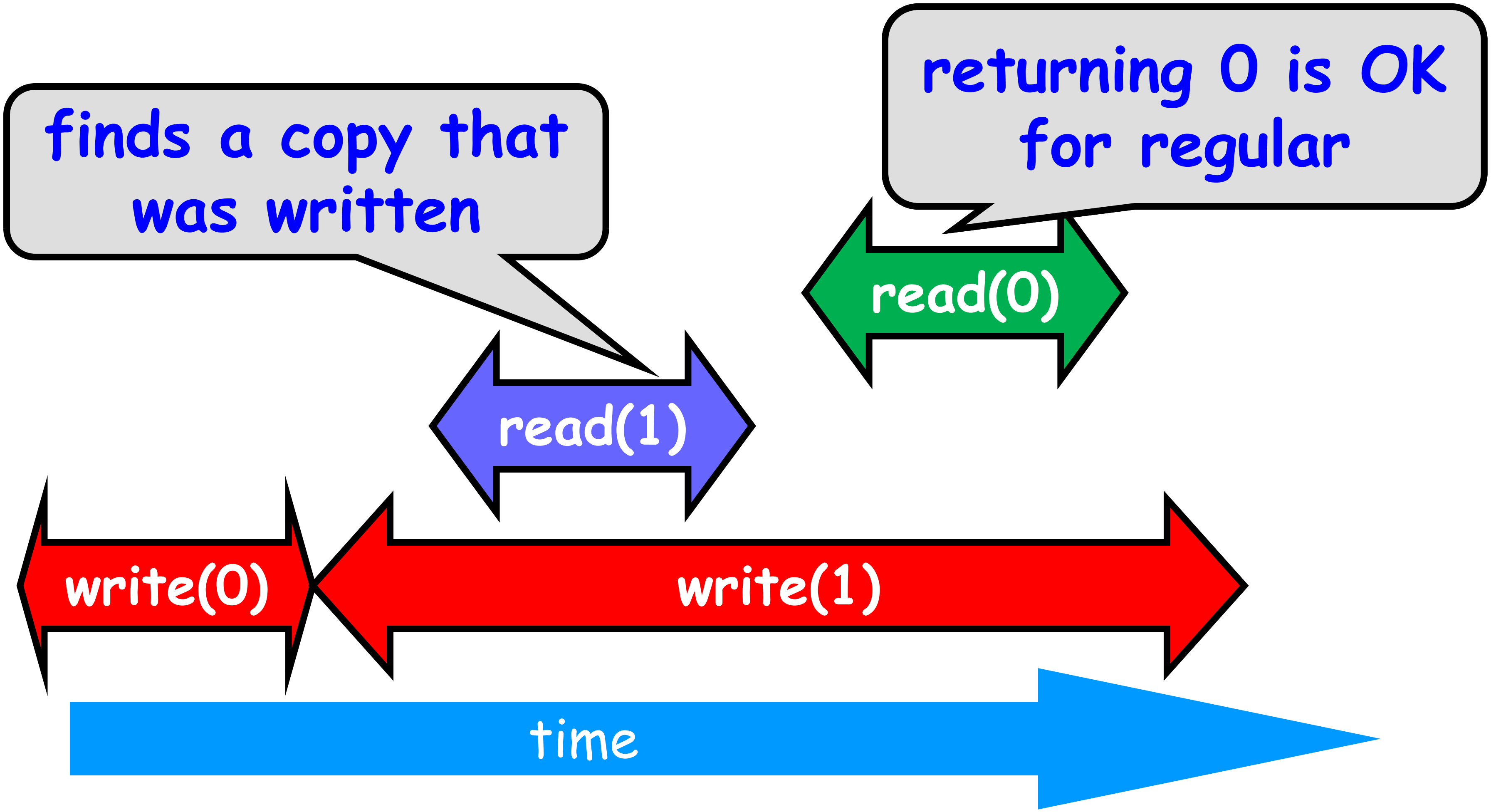


Process i can write block[i][j] in each disk j, can read all blocks

Read Emulation

- In order to read x_i
  - Issue read block[i][j] for each disk j
  - Wait for majority of disks to respond
  - Choose block with largest b,n
- Is this enough?
- How did ABD's read emulation work?

One Read Round Enough for Regular



Write Emulation

- In order to write x_i
  - Issue write block[i][j], for each disk j
  - Wait for majority of disks to respond
- Is this enough?

Summary

- ABD: Emulate reliable shared memory
  - In asynchronous system
  - Using fault-prone storage nodes (minority)
- SM Paxos: Solve consensus
  - In asynchronous reliable shared memory
  - Using leader-election failure detector
  - Tolerate any number of client failures
- Disk Paxos: Combine the two

Additional Challenges & New Results

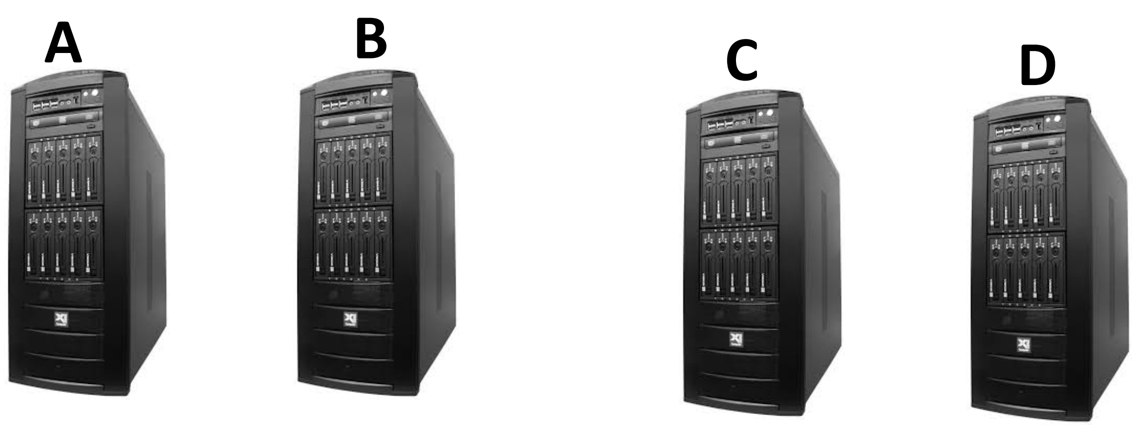
Reconfiguration

Codes to Mitigate Storage Blow Up

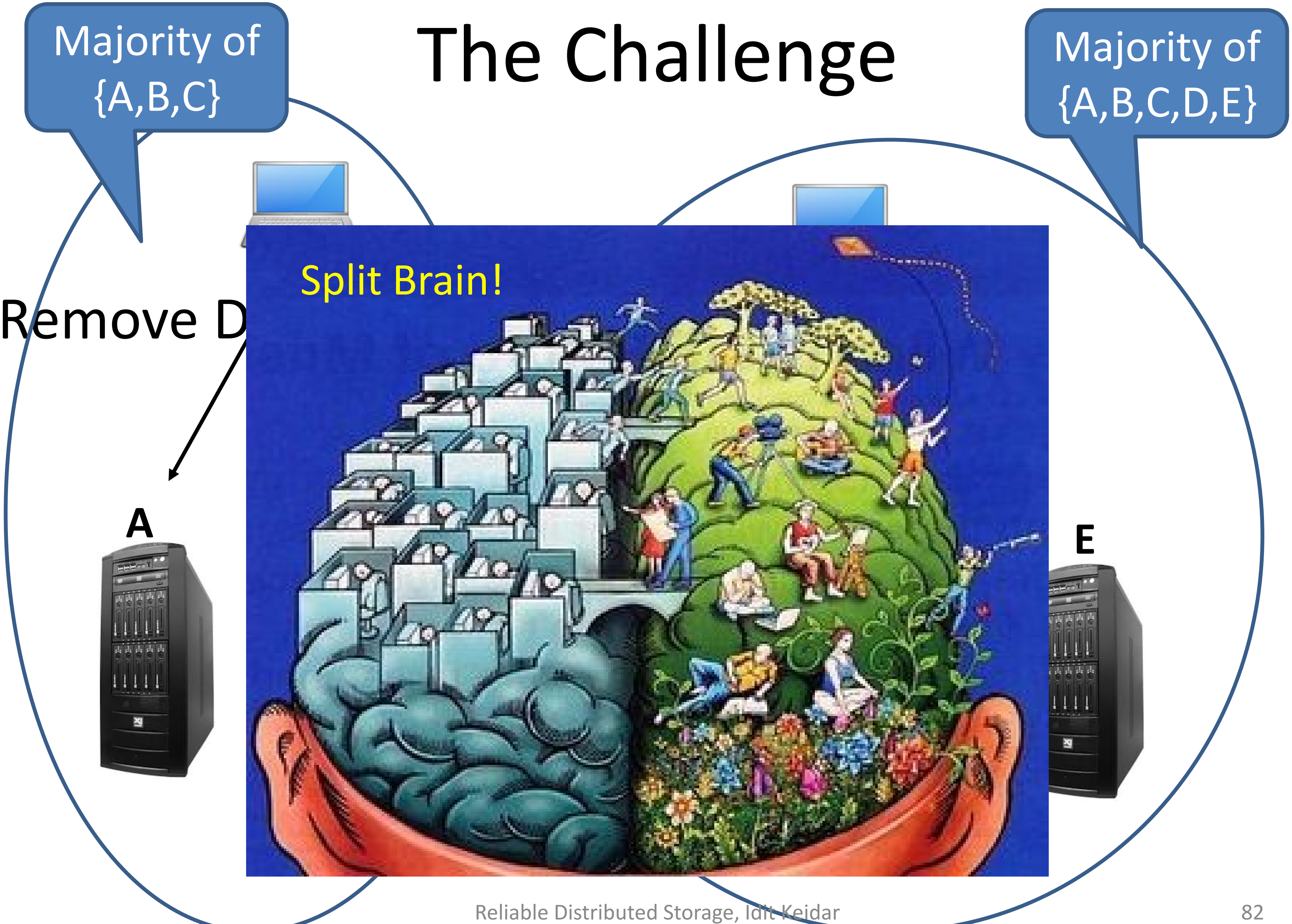
Reconfiguration

- Limited availability: always need C, D, and E
- After removing A, B, need two of {C,D,E}

The Challenge



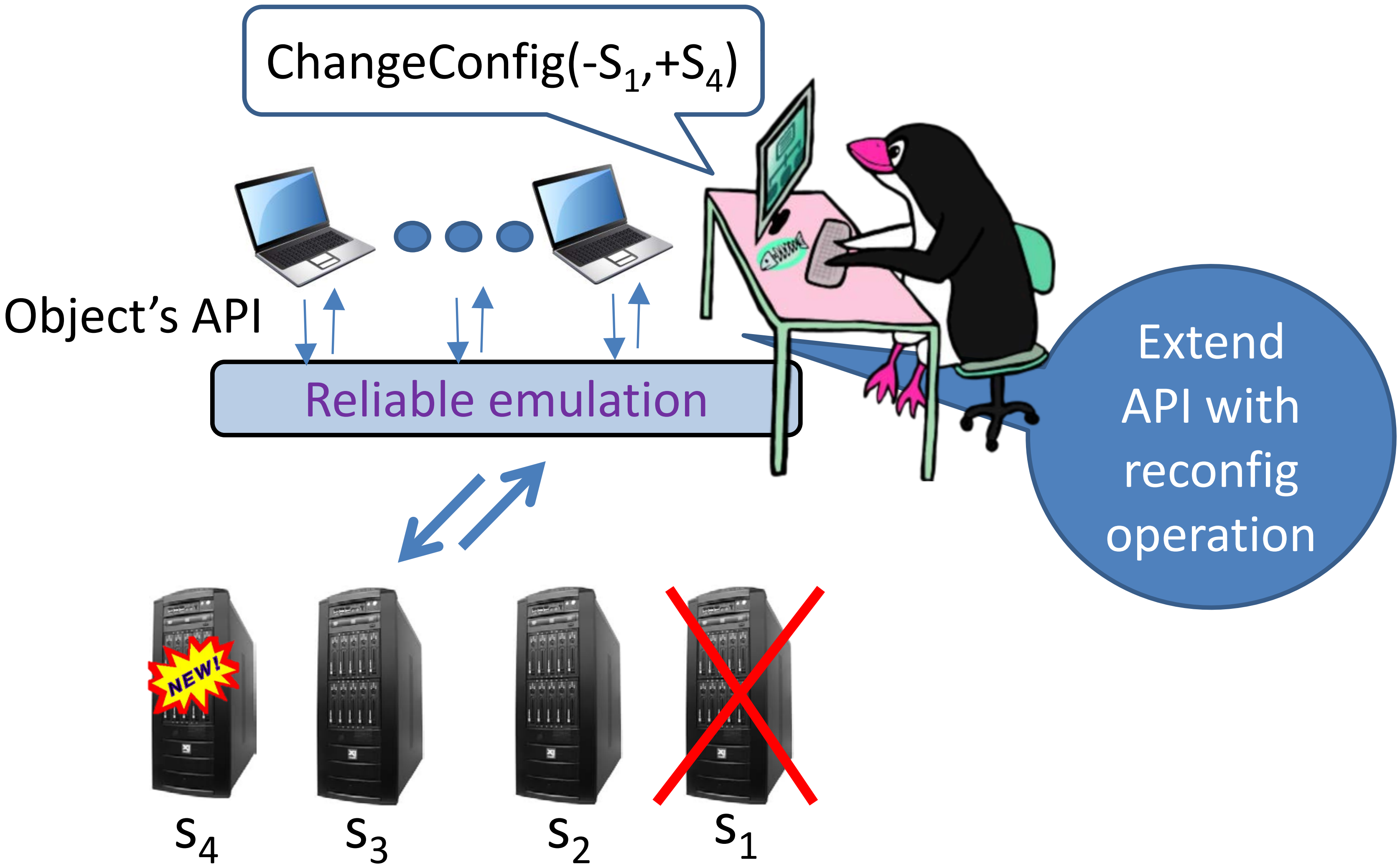
The Challenge



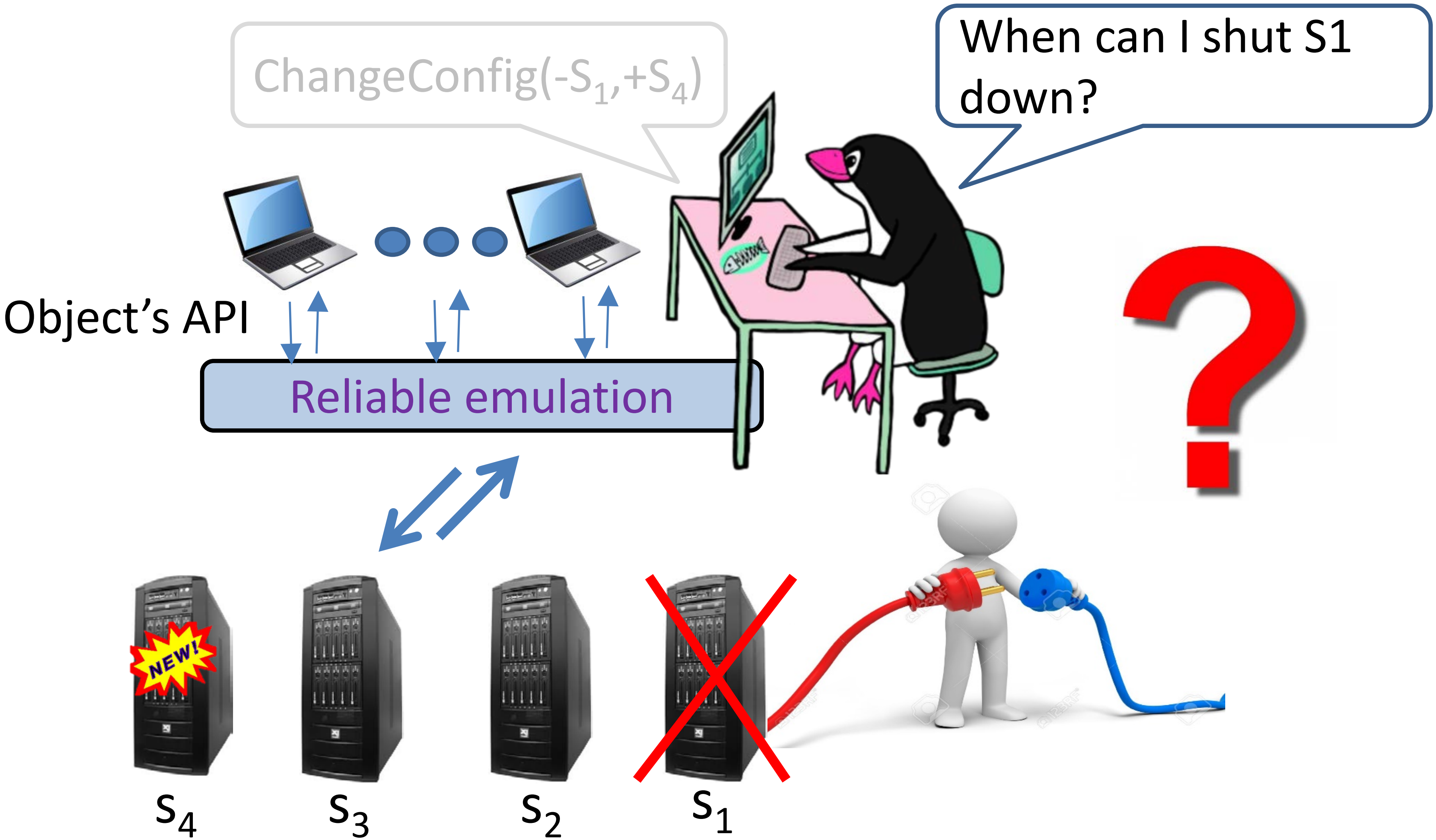
Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution

A. Spiegelman, I. Keidar, and D. Malkhi, DISC 2017

Dynamic Reliable Objects



What Now?

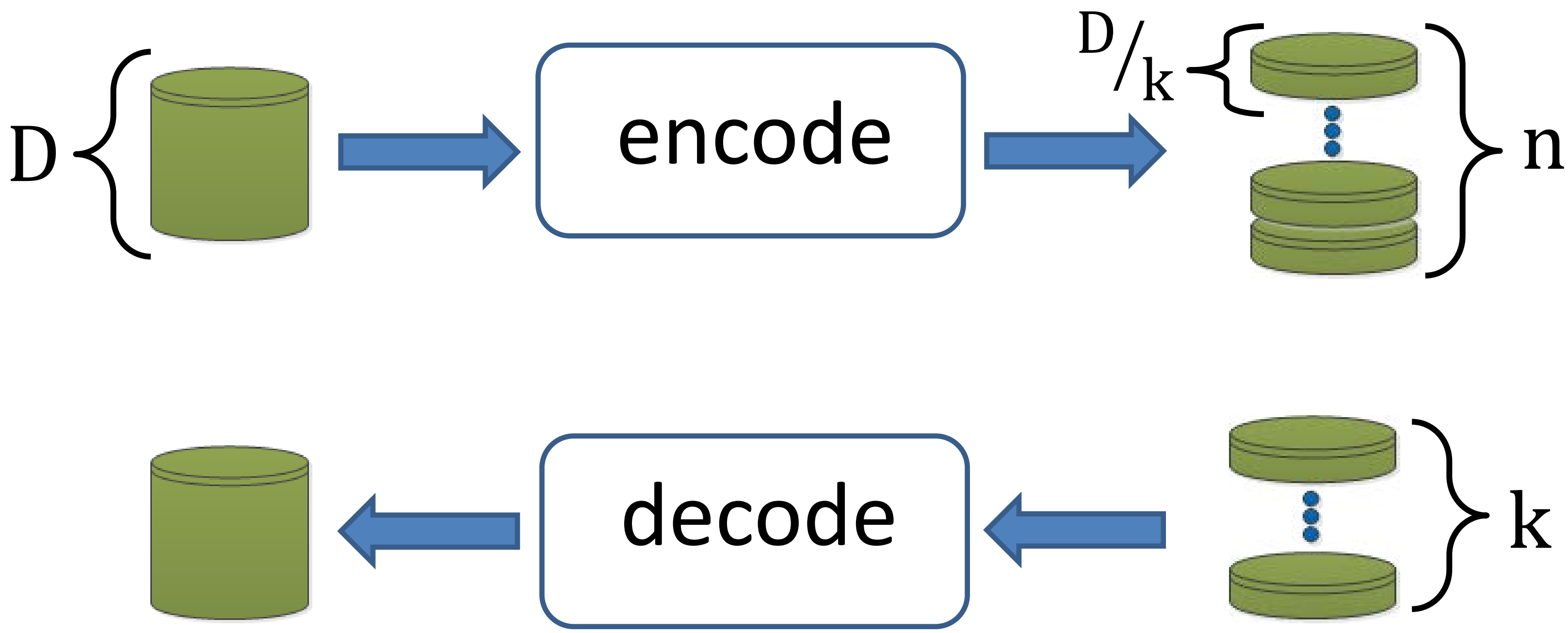

ChangeConfig(-S_1,+S_4)
When can I shut S1 down?
Object's API
Reliable emulation
NEW!
S_4
S_3
S_2
S_1

Contributions

- Clean model for dynamic objects
  - API, failure condition, complexity metrics
- General abstraction for reconfiguration
- Optimal asynchronous register emulation
  - see paper

Storage Blow Up

k-of-n Erasure Codes



Why Codes?

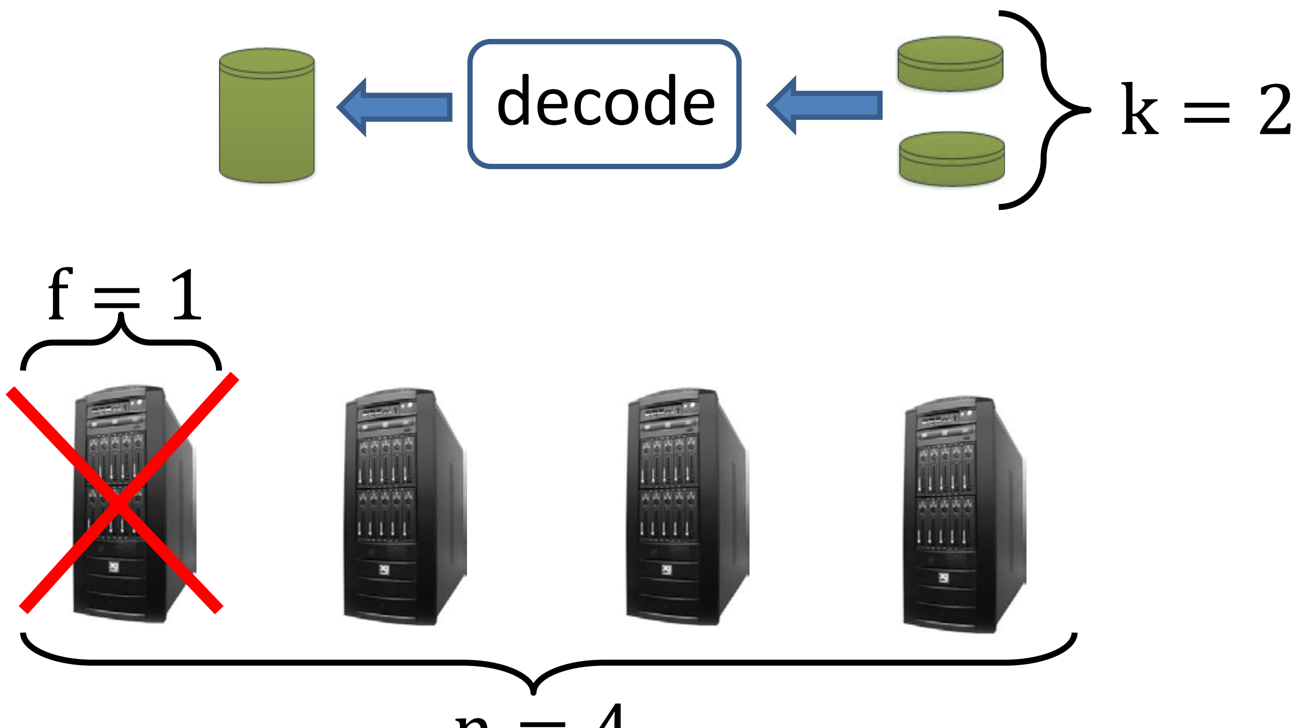
To tolerate **one** failure

- With replication

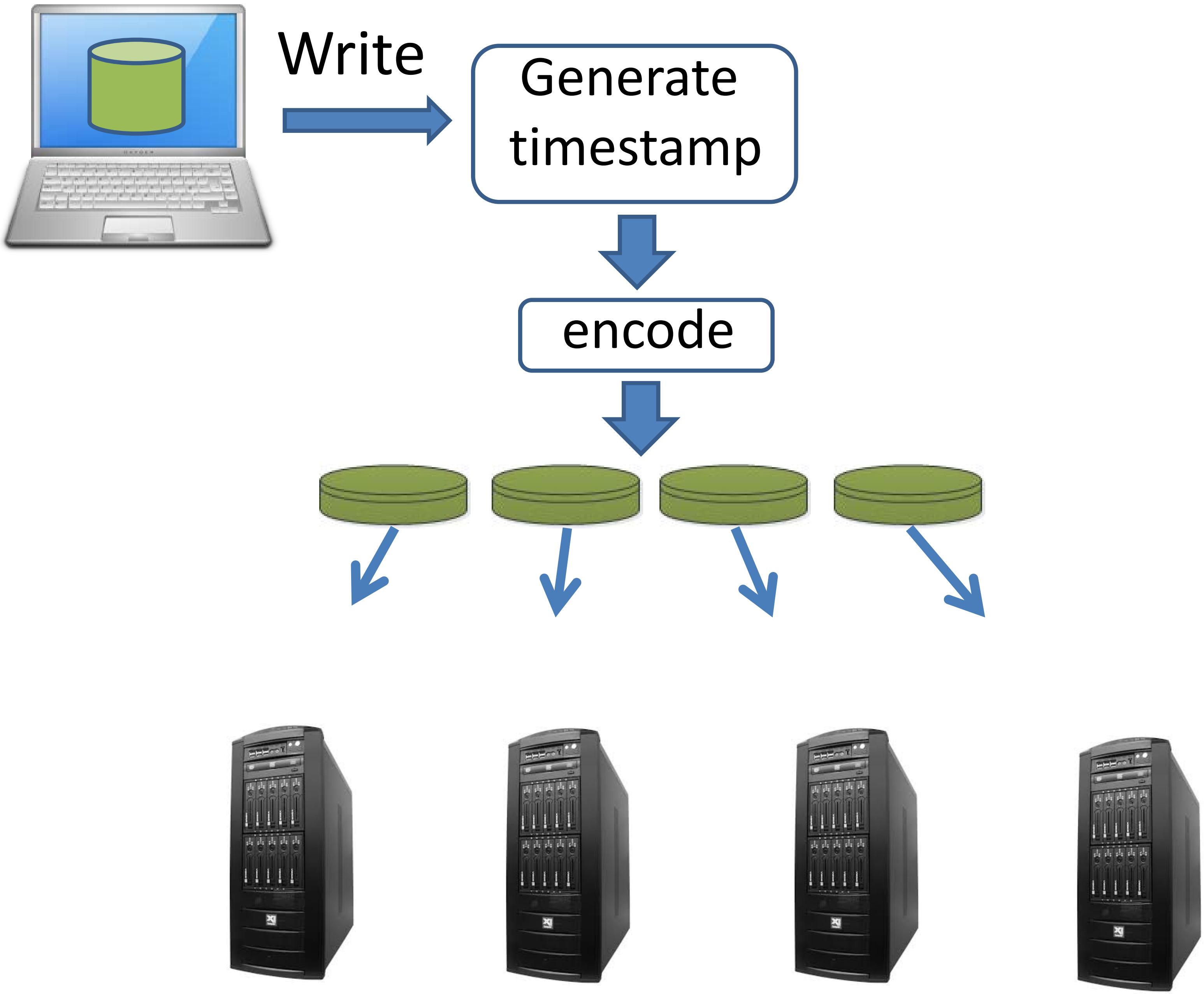
- With erasure codes

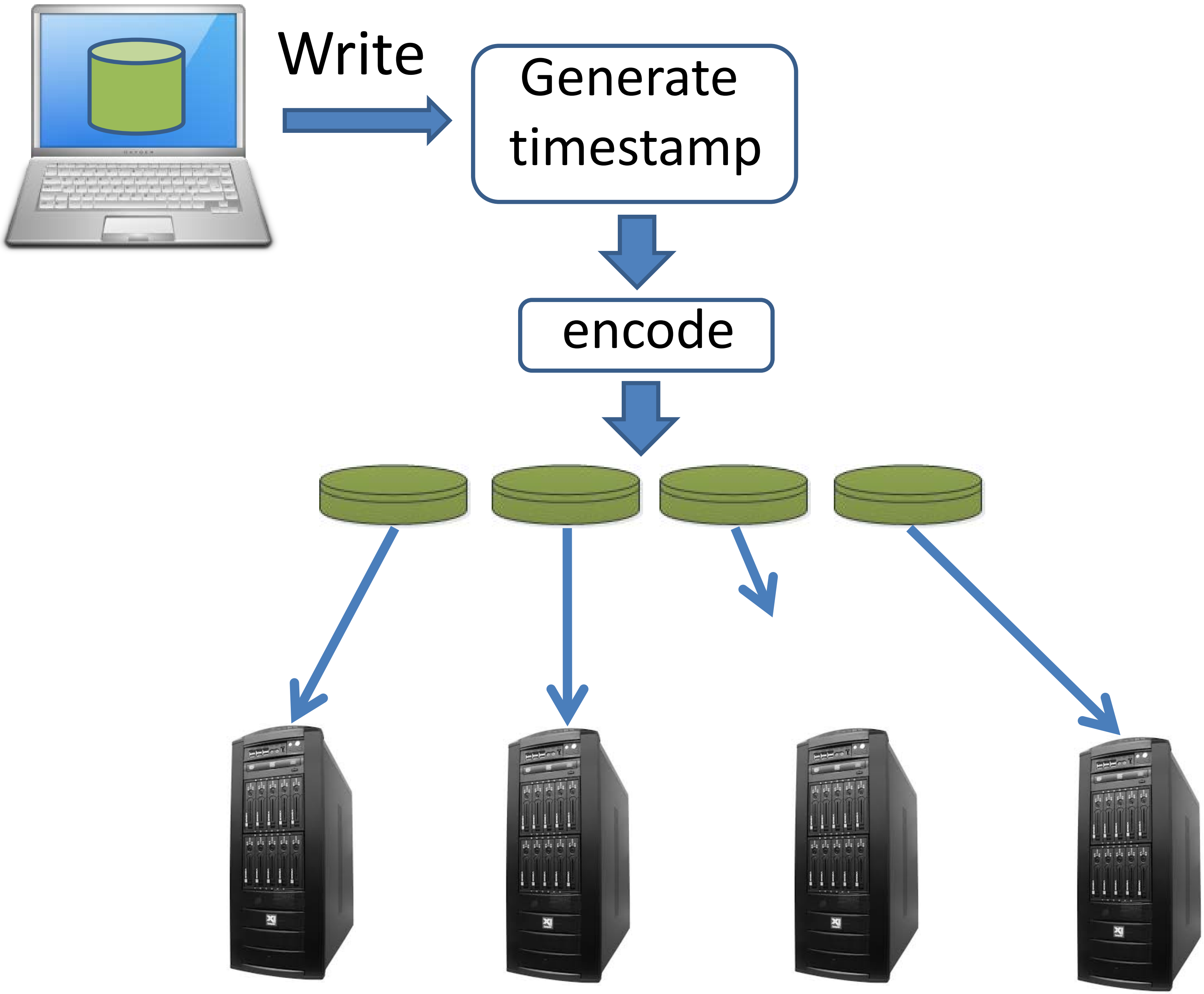
Reliable Storage Example

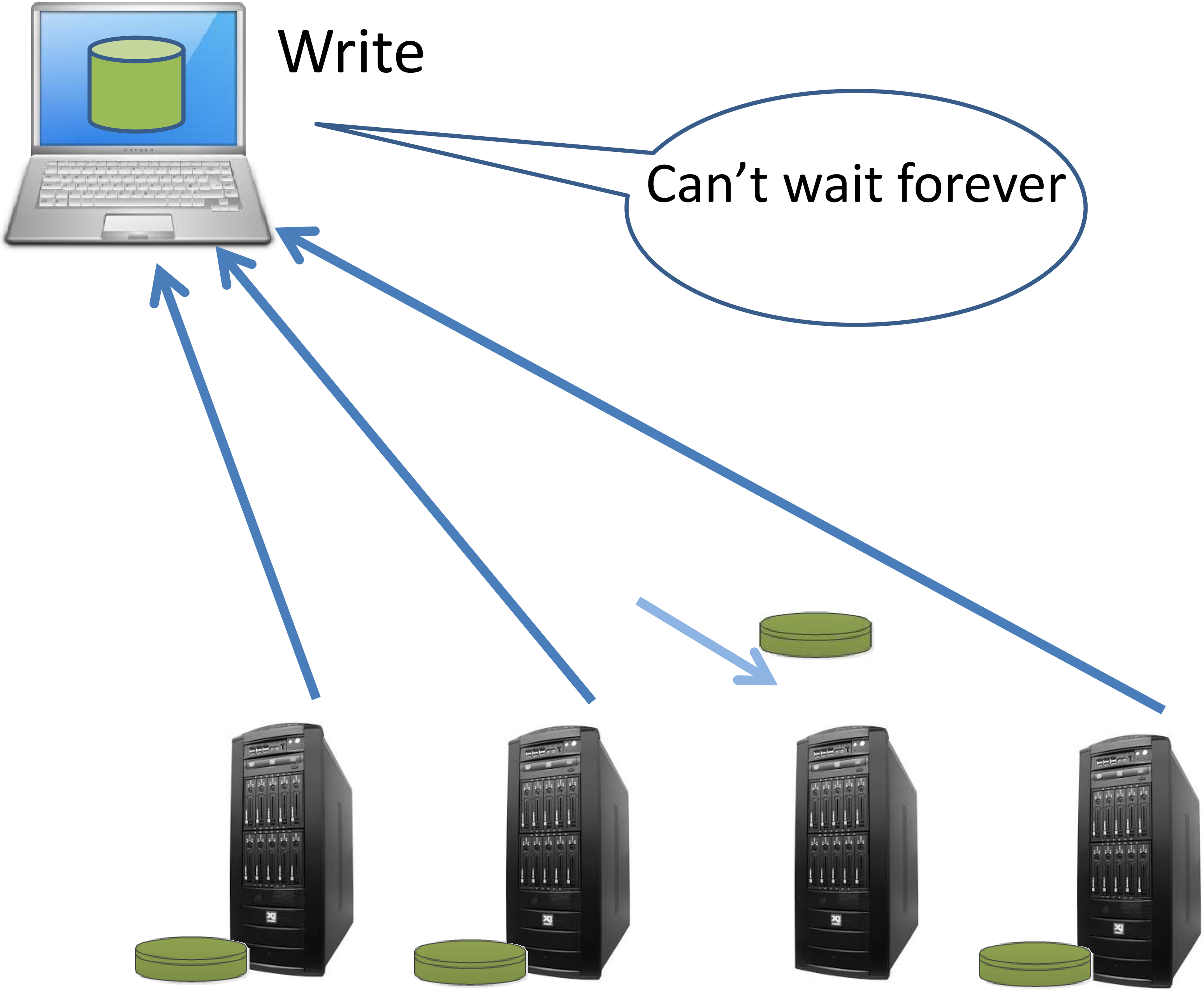
- $\mathbf{n = 2f + k}$

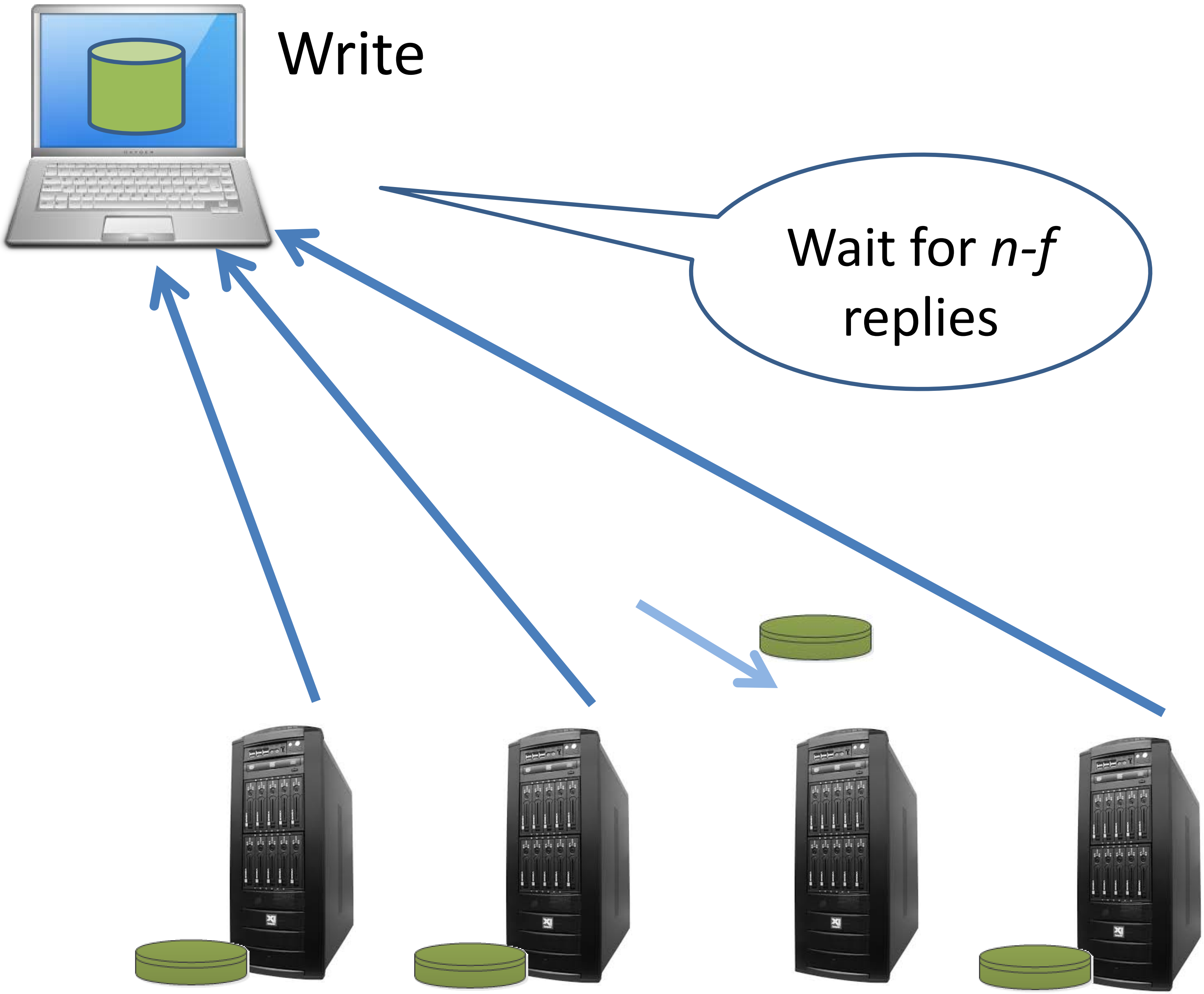


Write

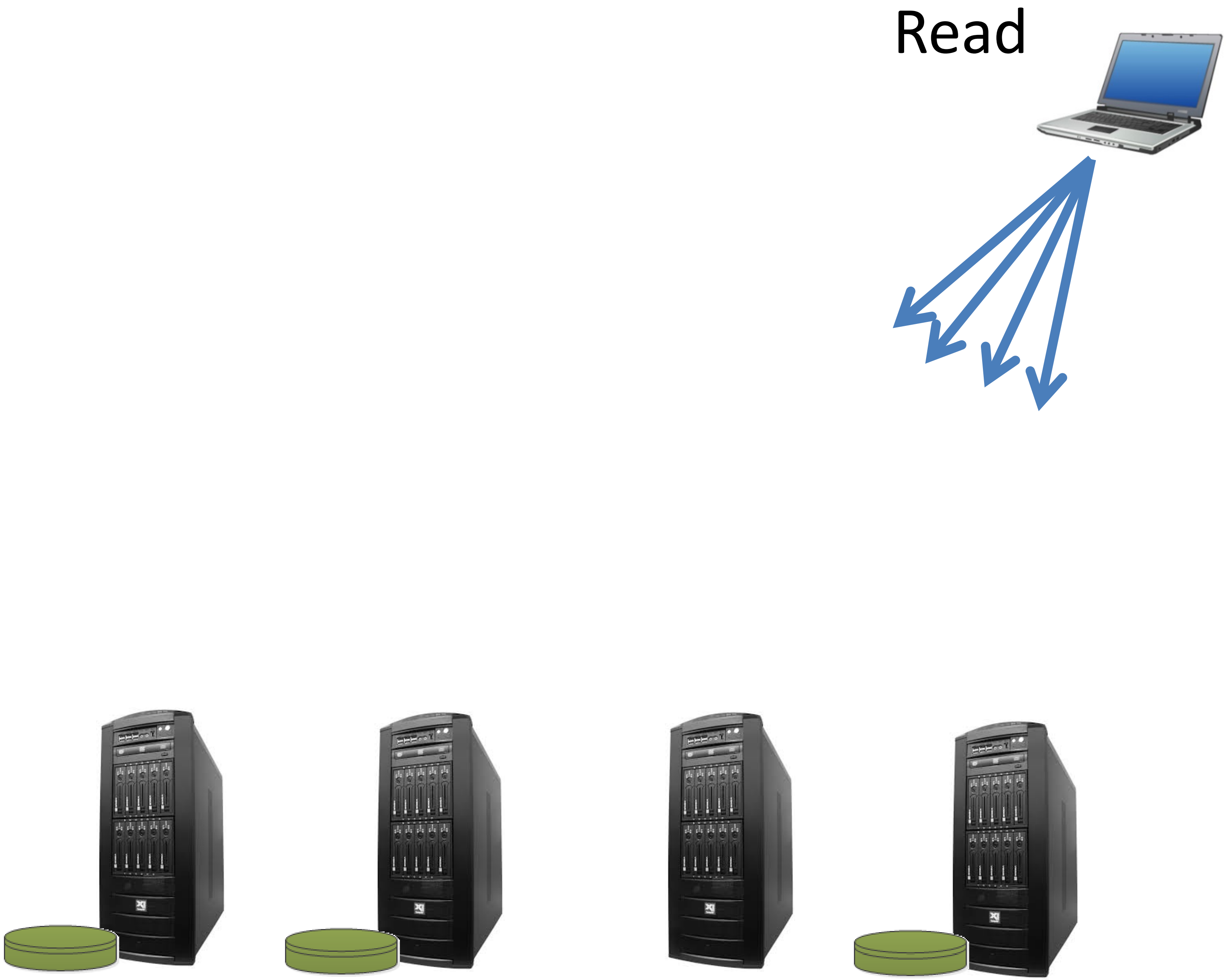

Write
Generate timestamp
encode

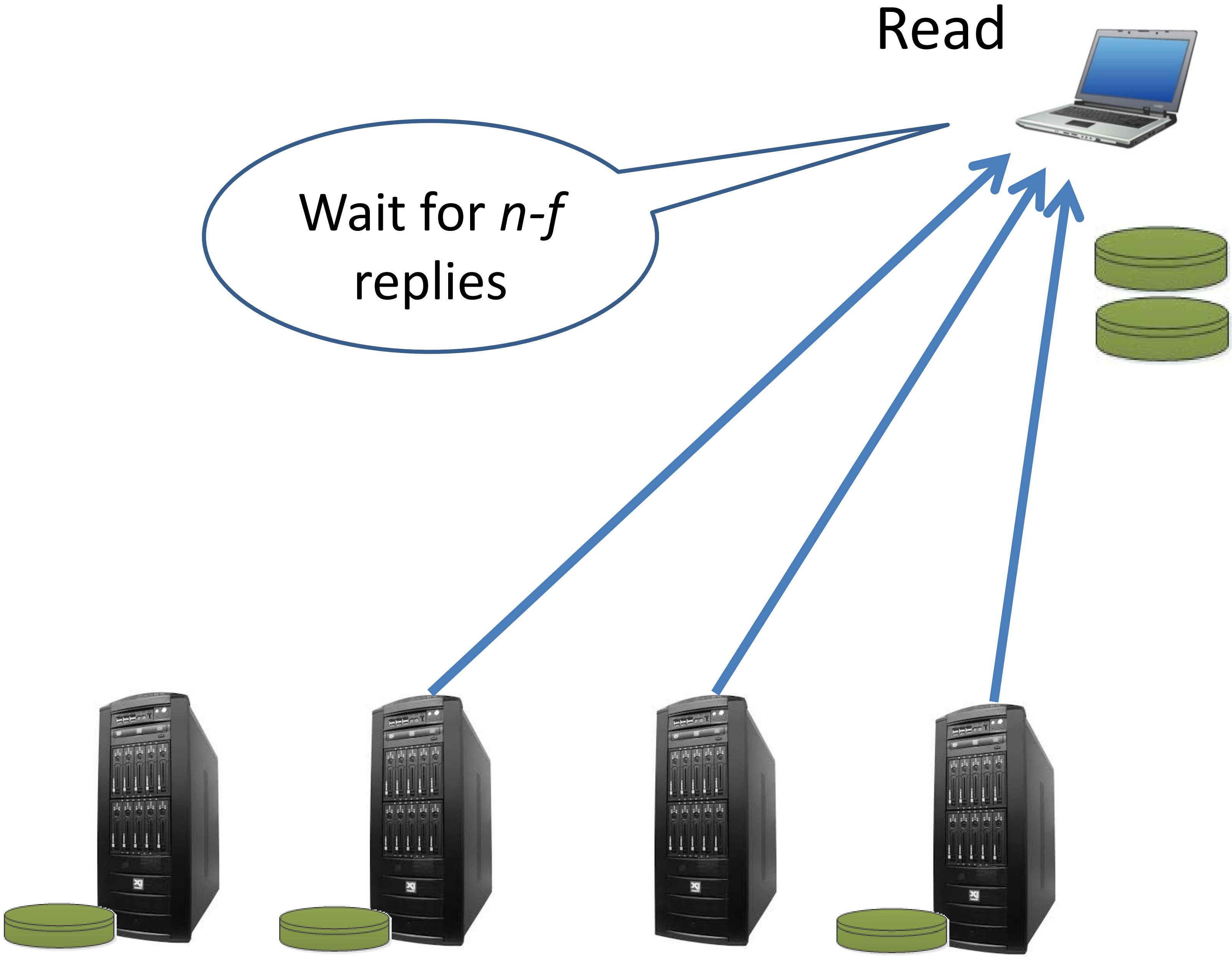

Write
Generate timestamp
encode

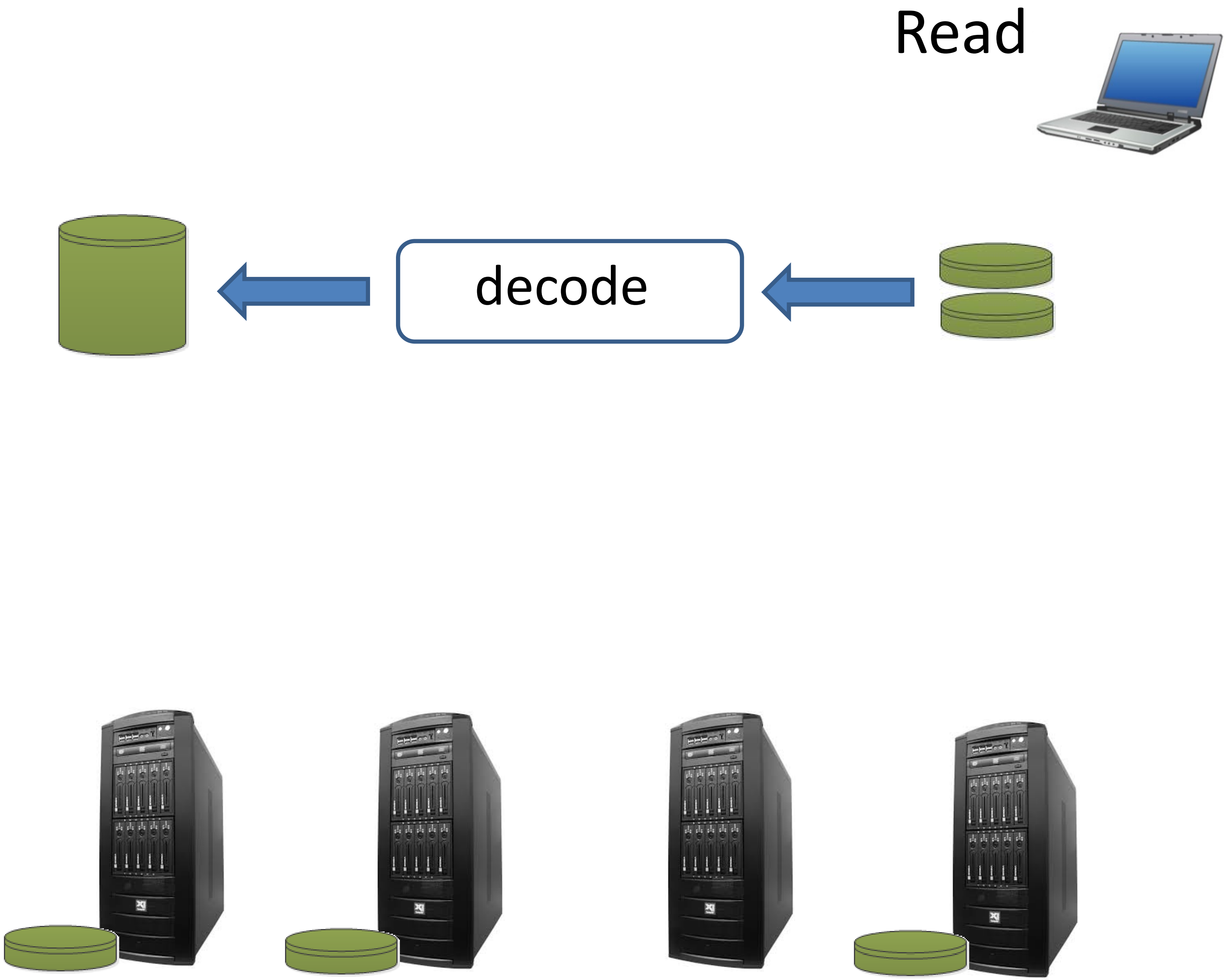

Write
Can't wait forever


Write
Wait for *n-f* replies

Read


Read

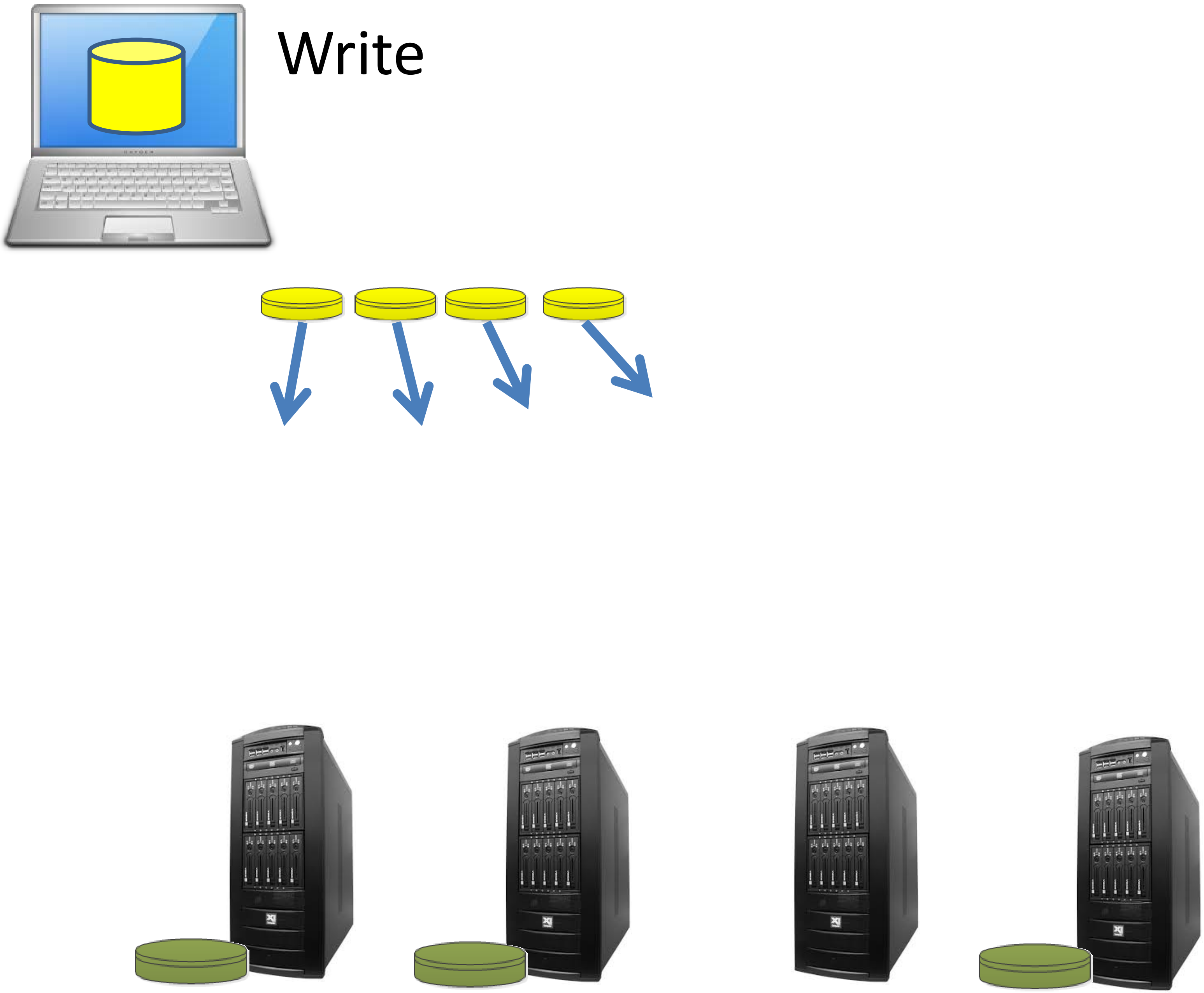

Read
Wait for *n-f* replies

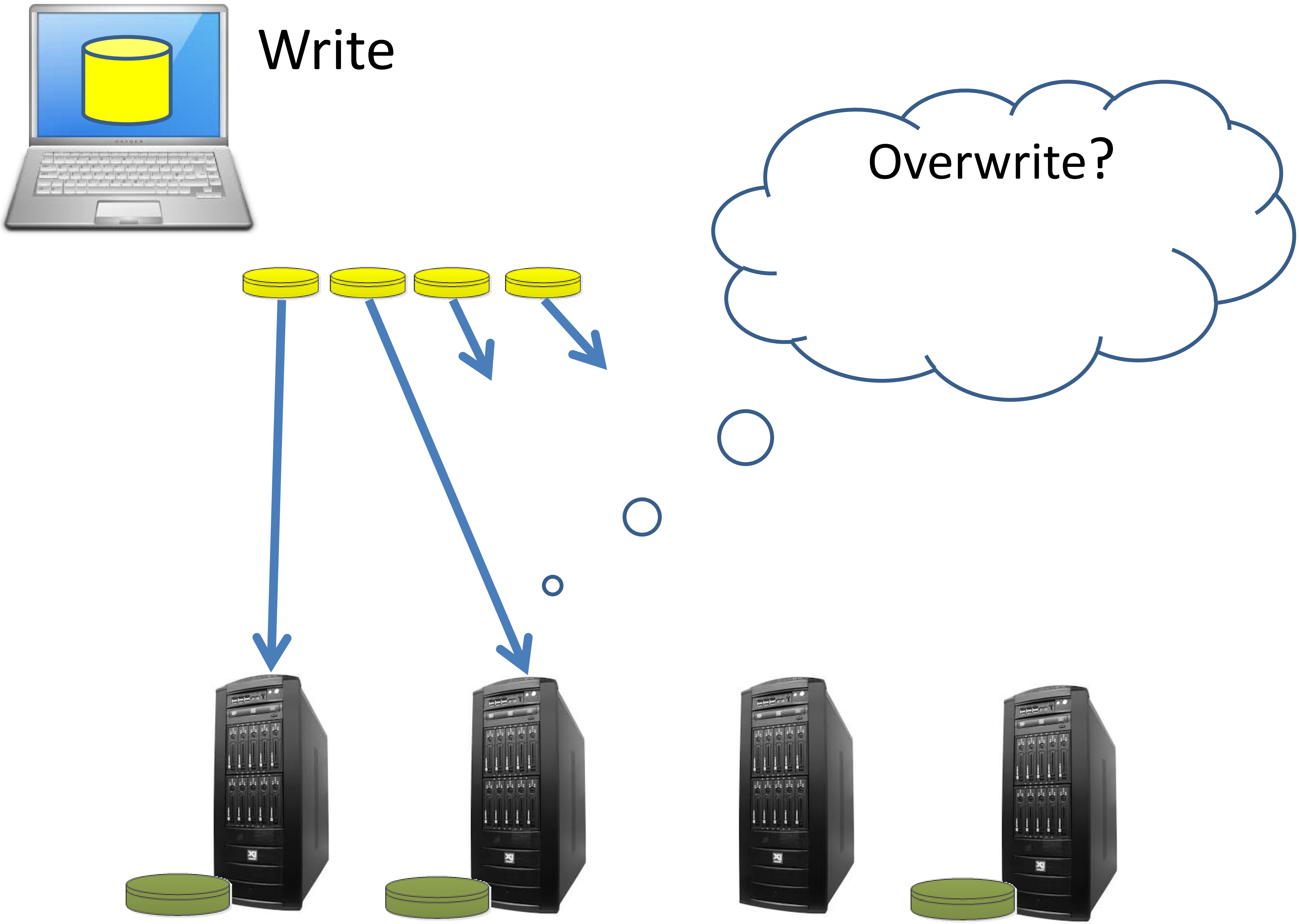

Read
decode

What About Concurrency?

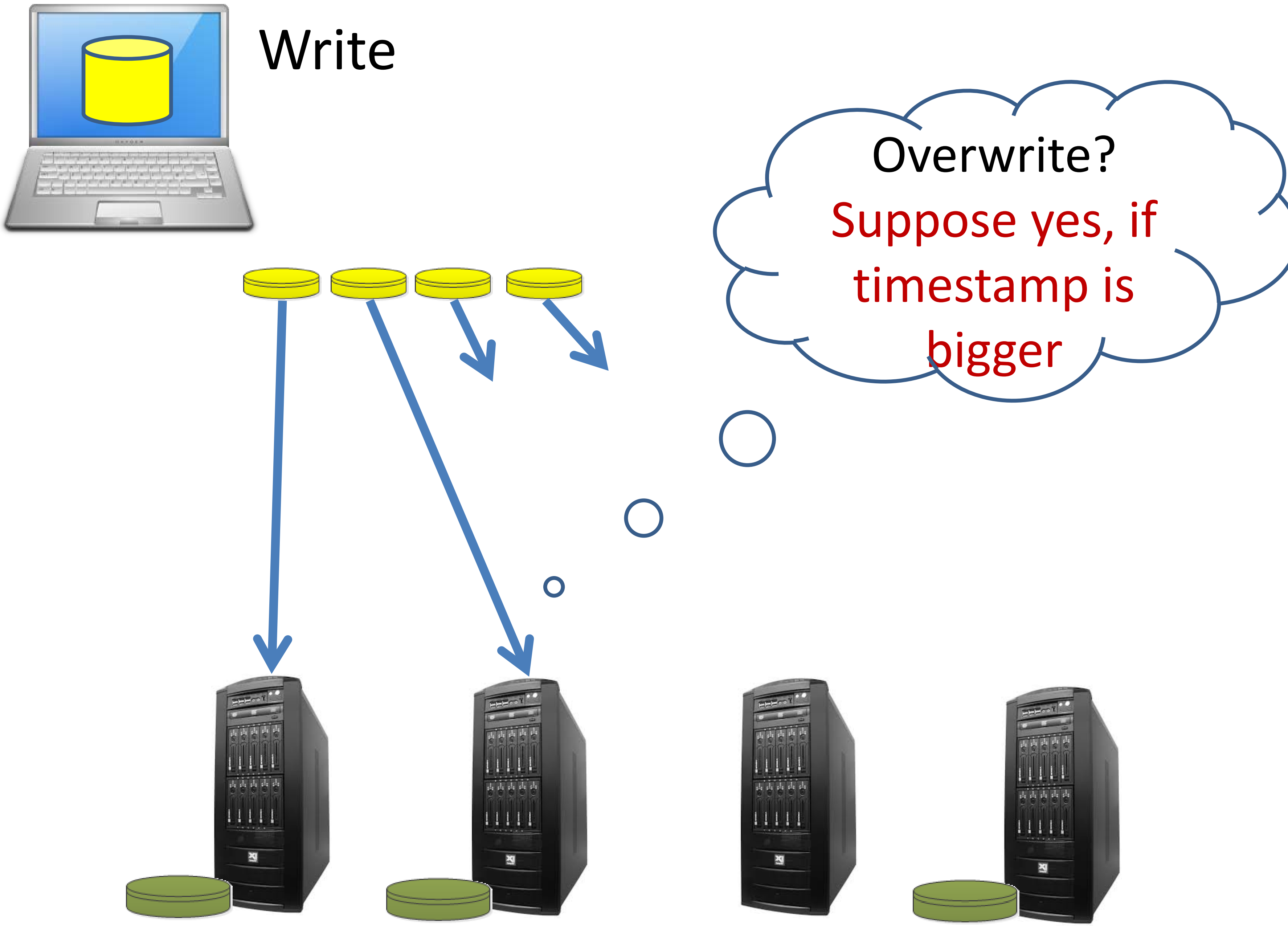
Write

Write

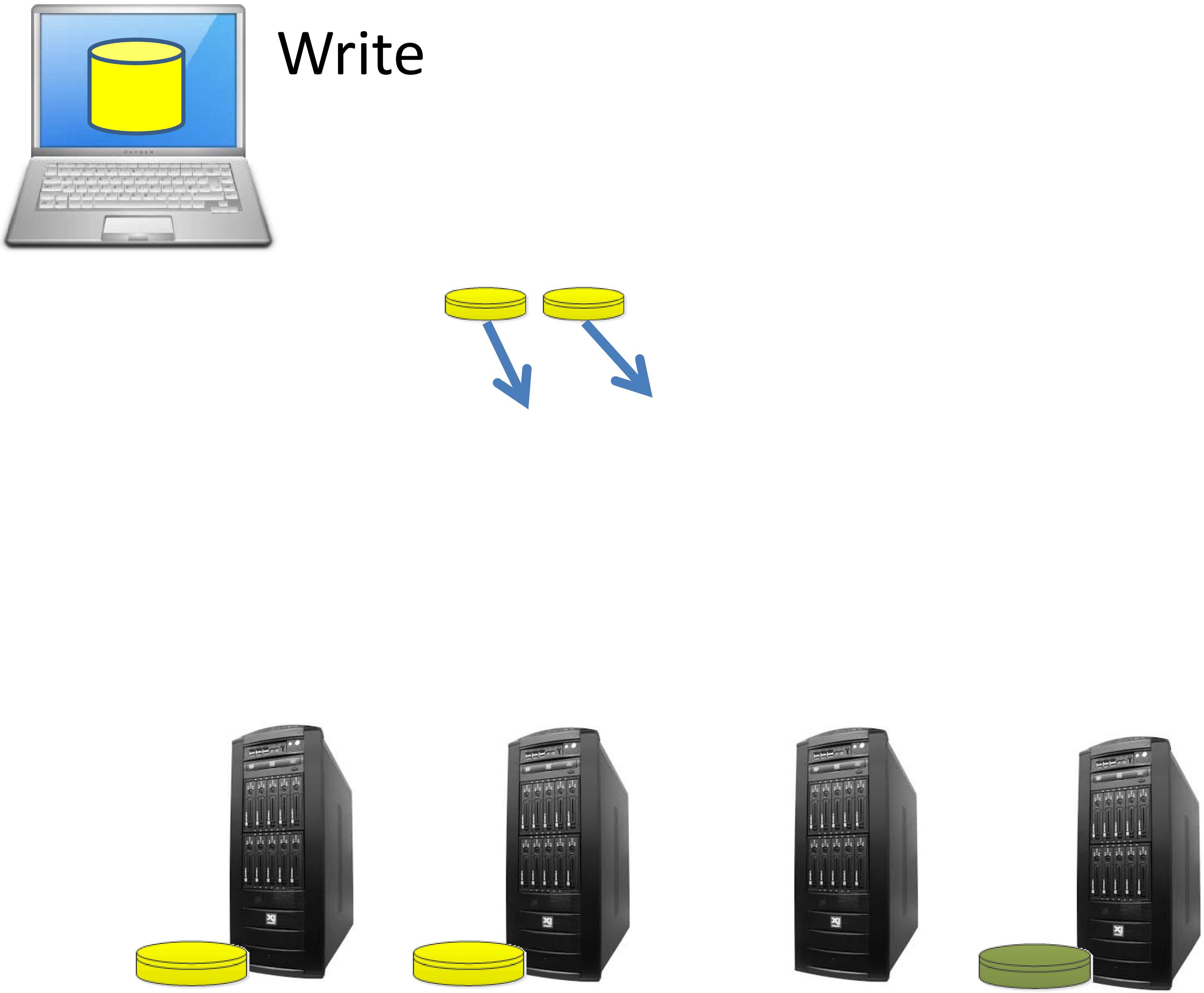


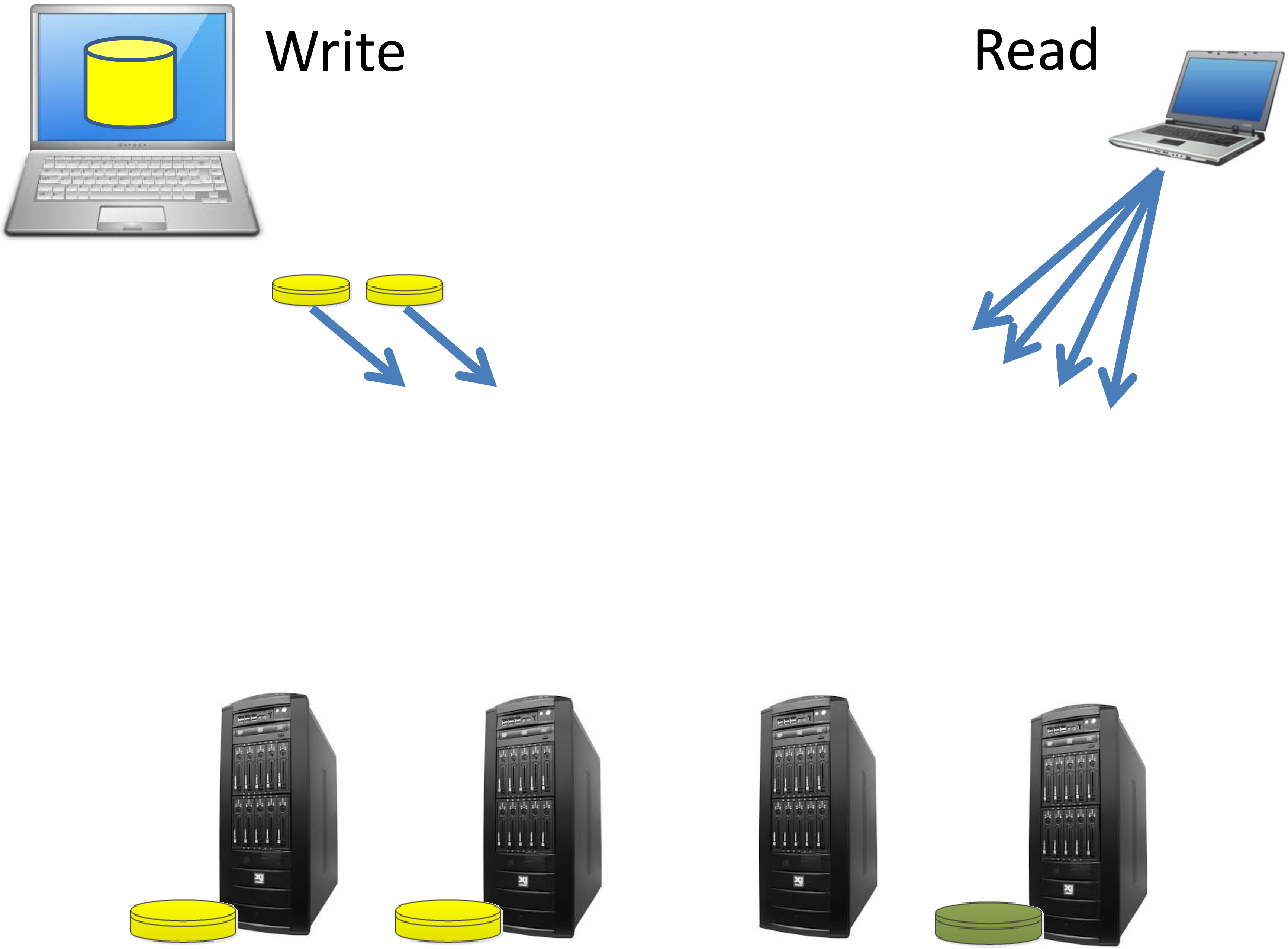

Write
Overwrite?

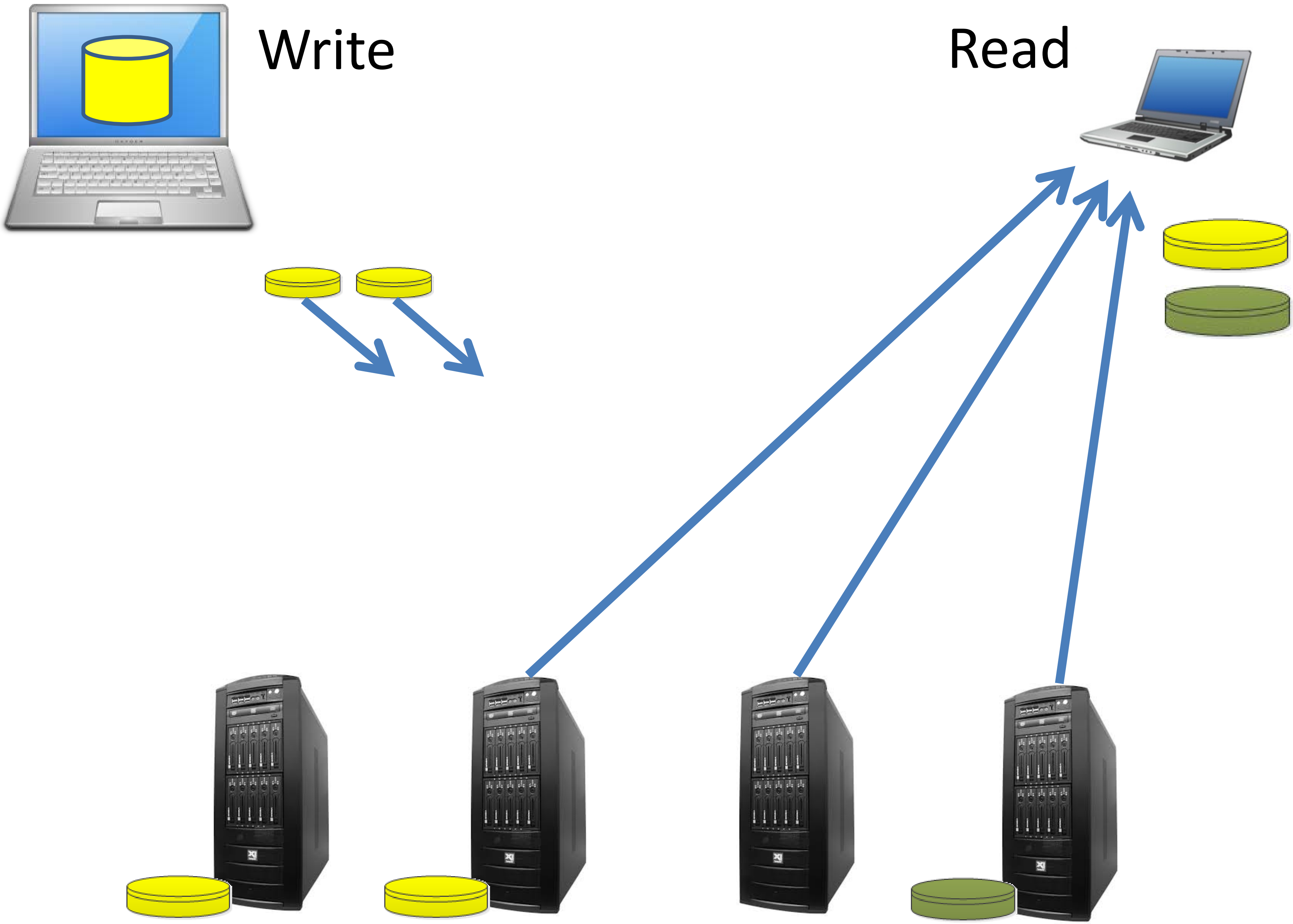
Write

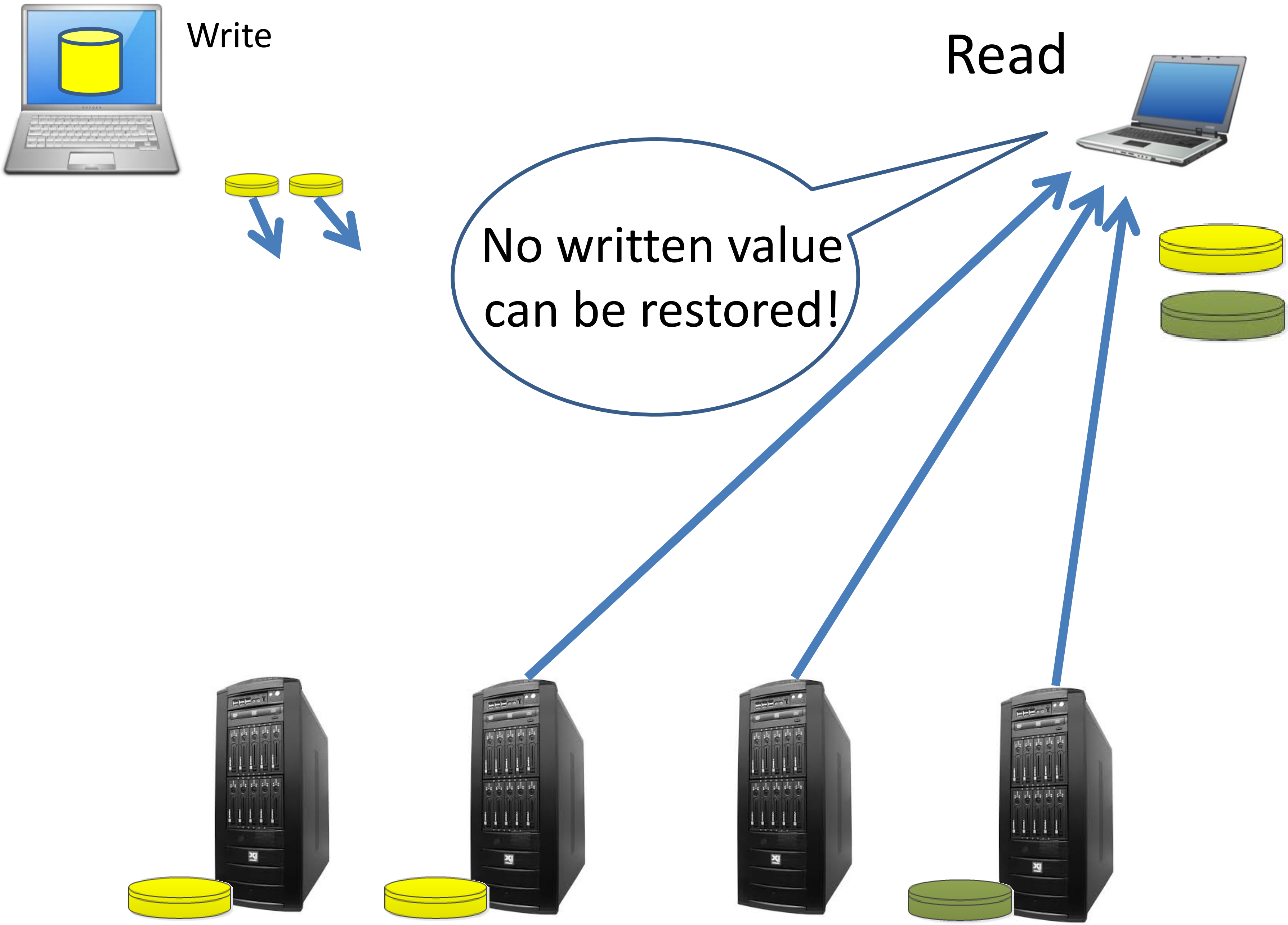


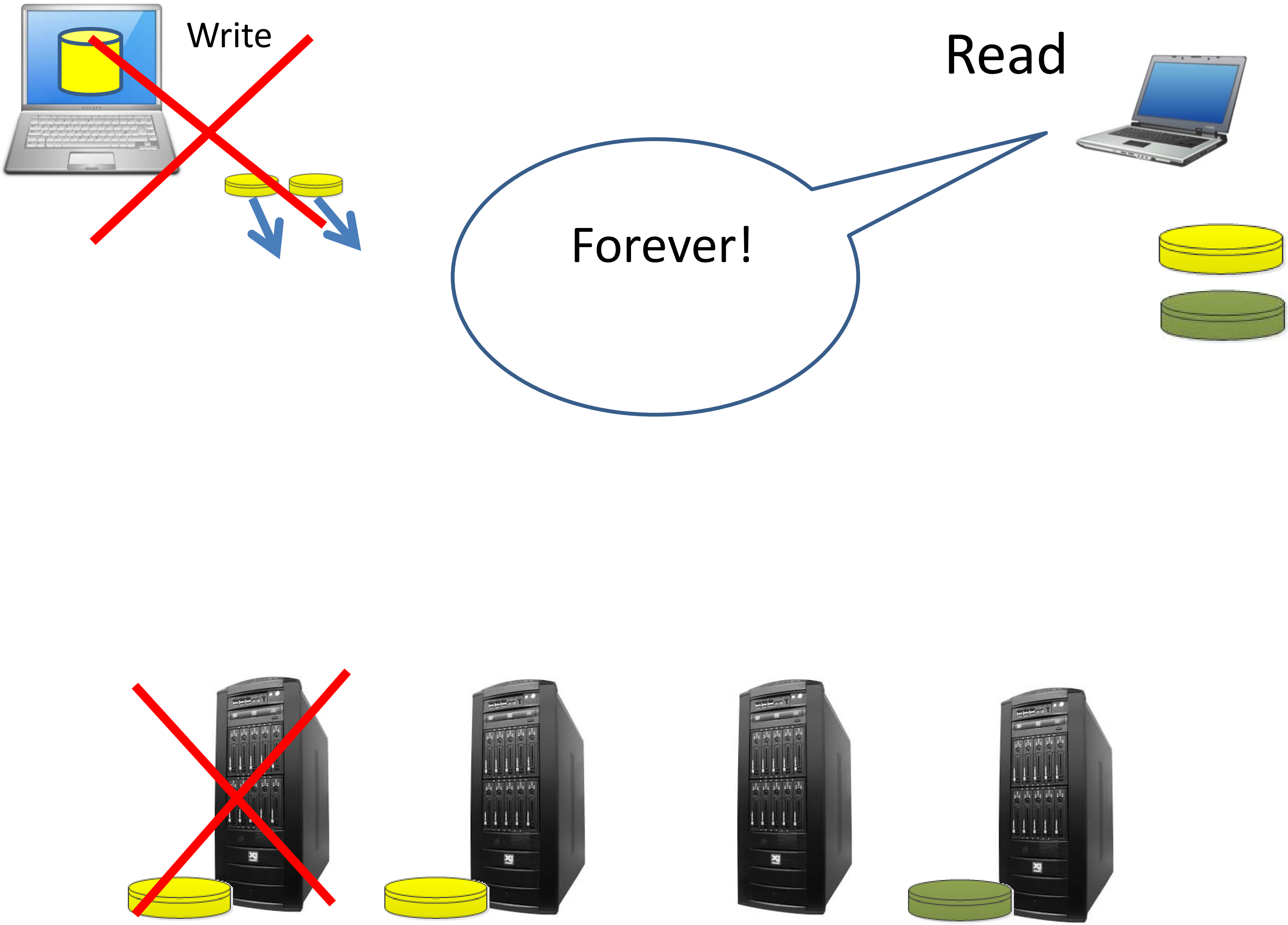
Write



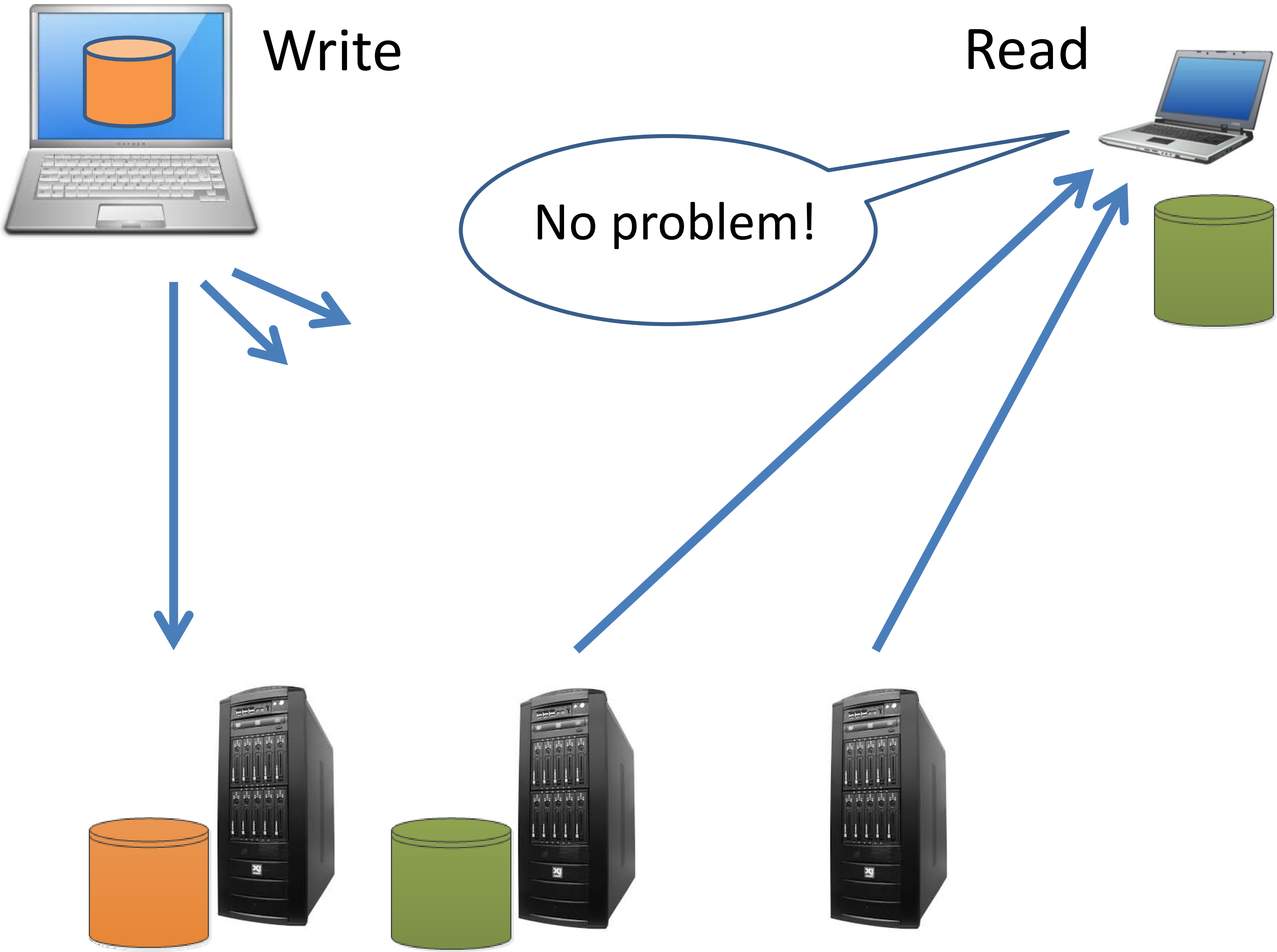

Write
Read

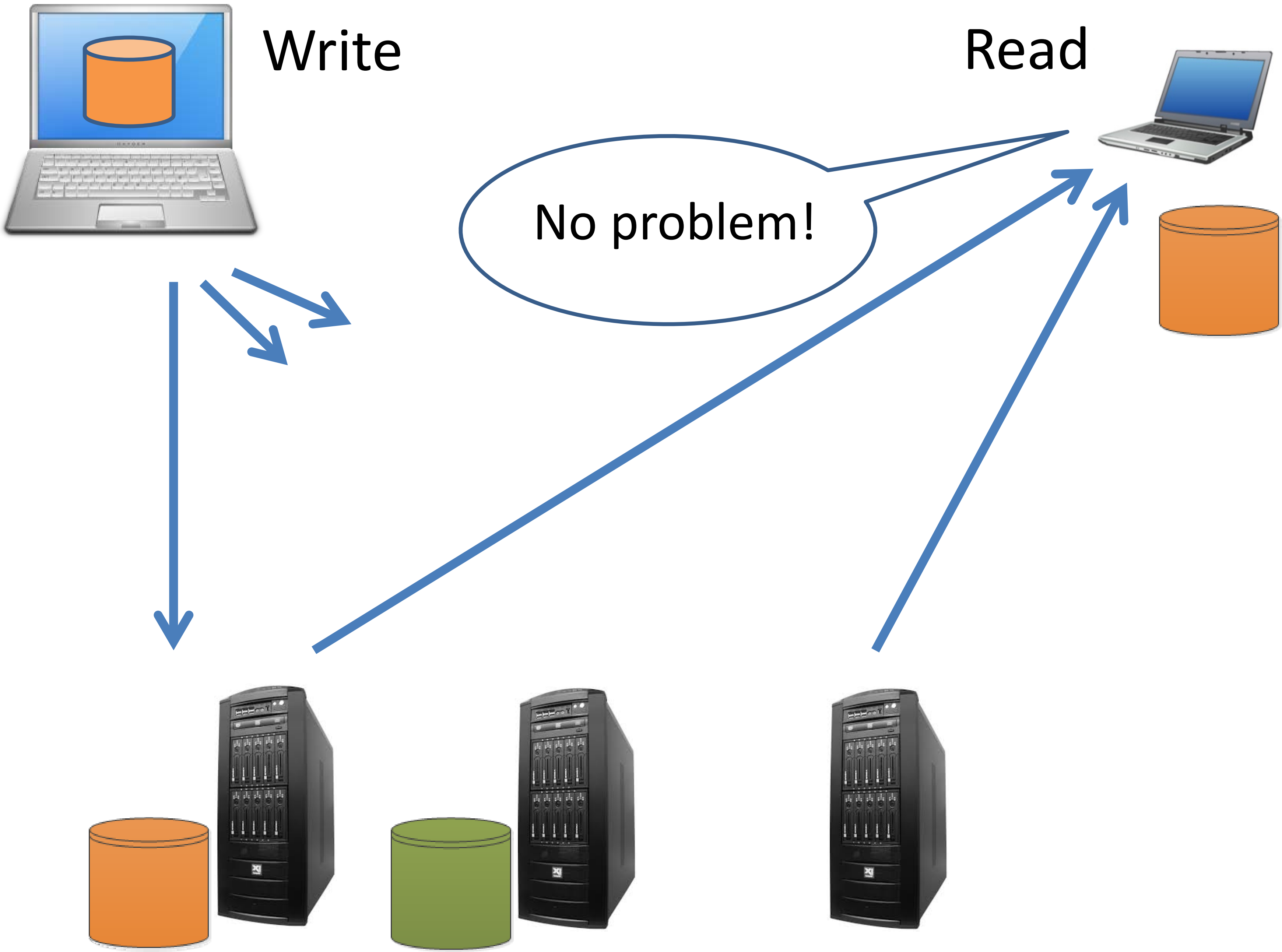

Write
Read


Write
Read
No written value
can be restored!

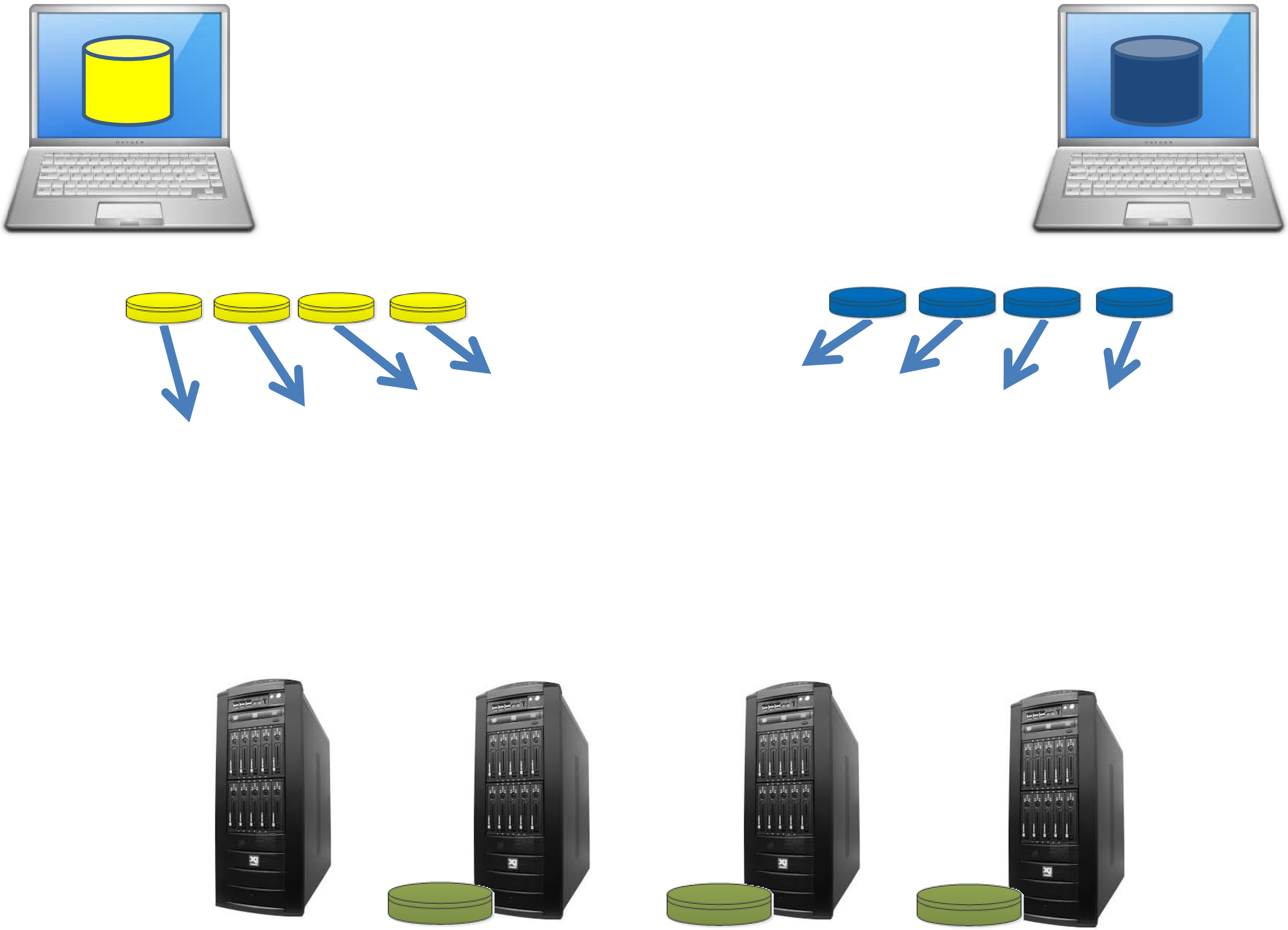

Write
Read
Forever!

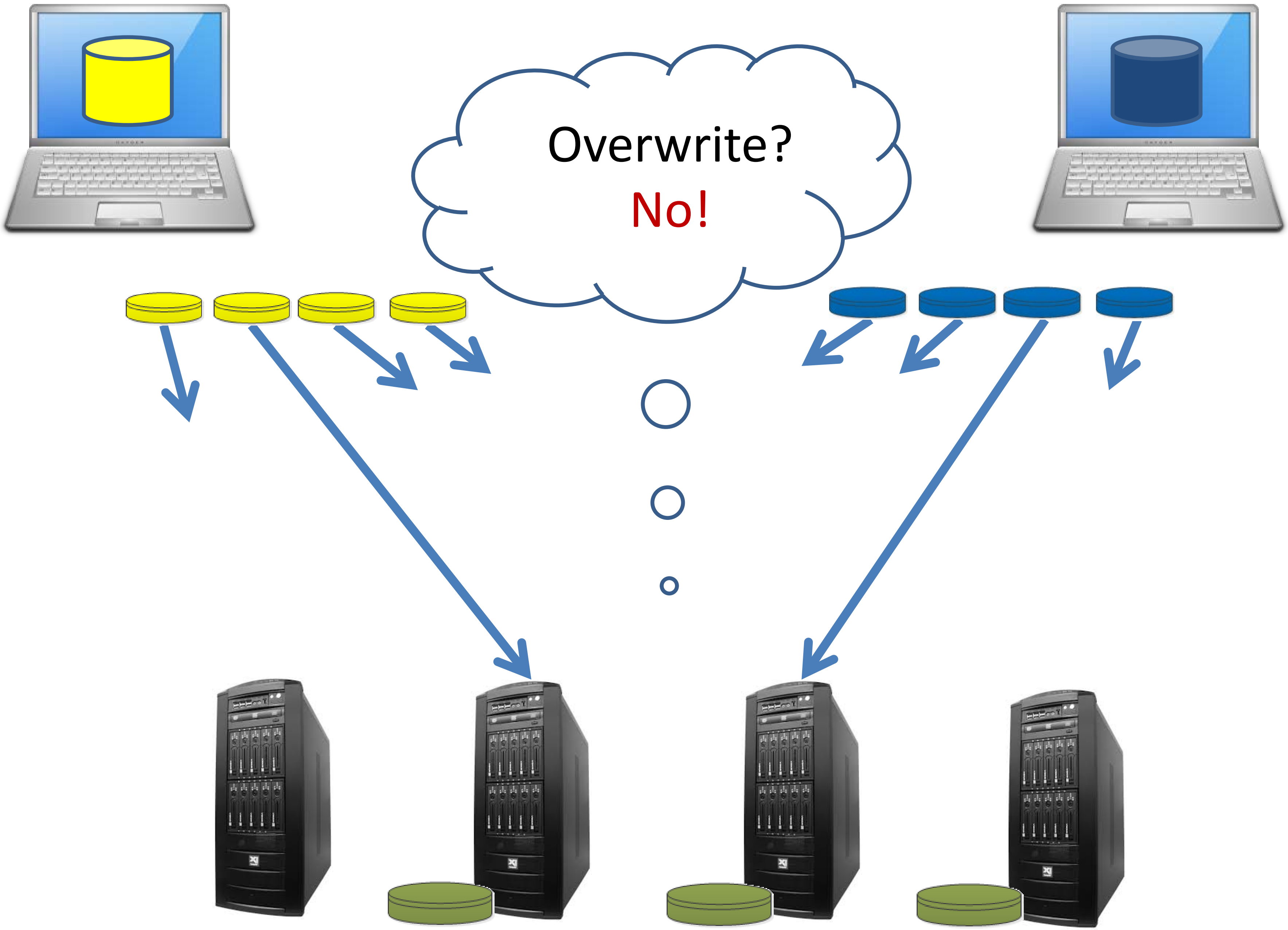
What About Replication?

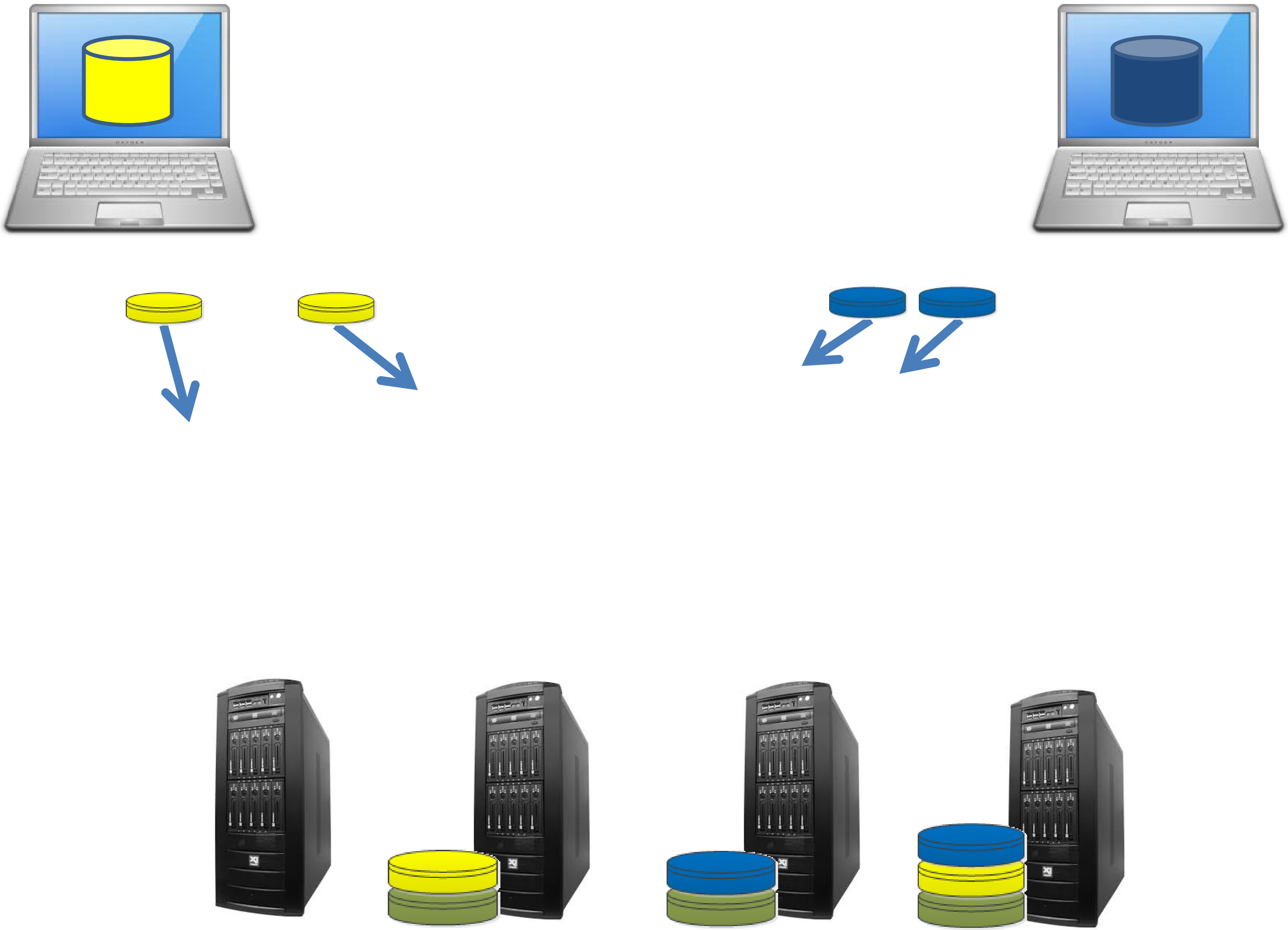

Write
Read
No problem!

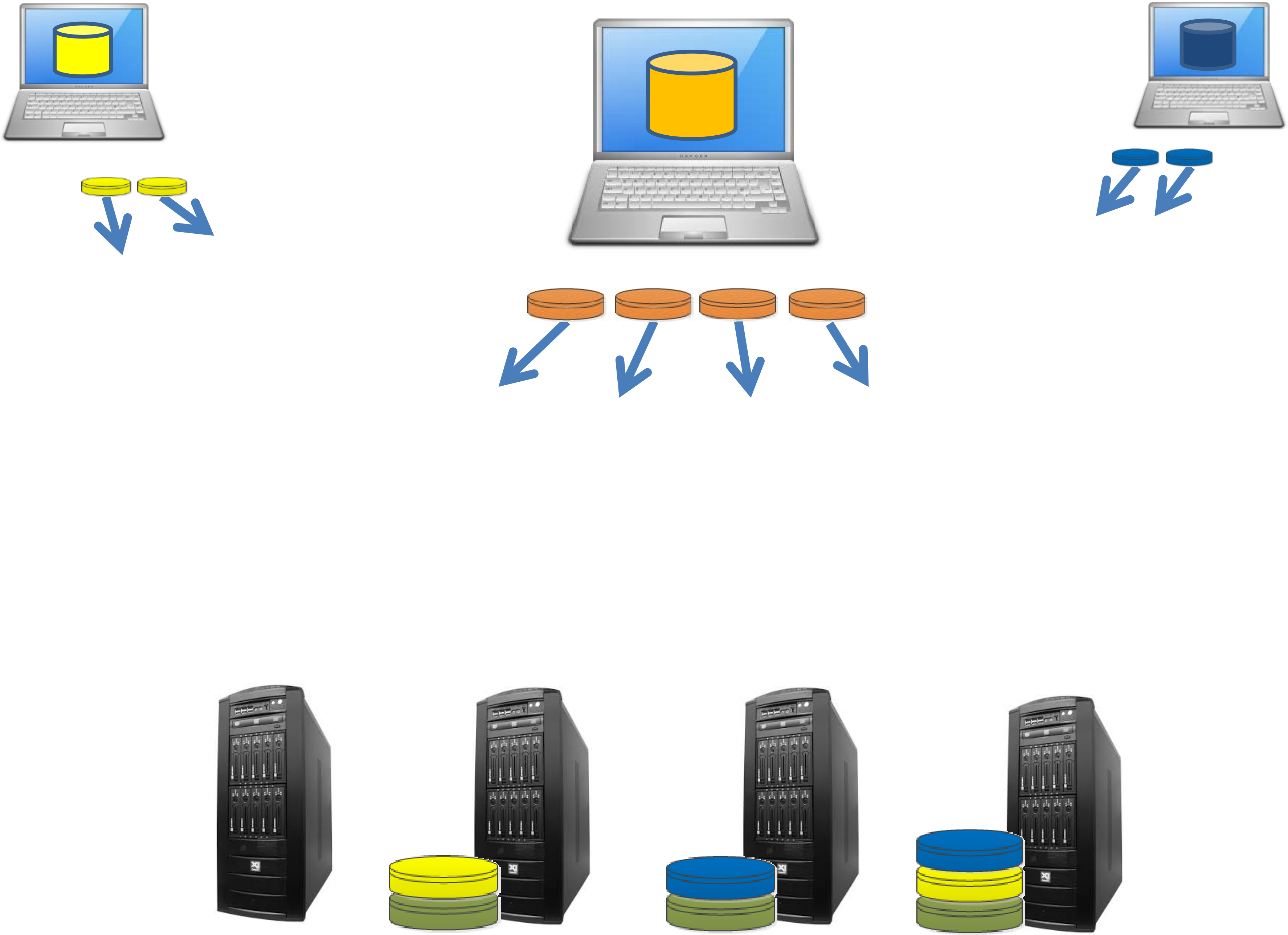

Write
Read
No problem!

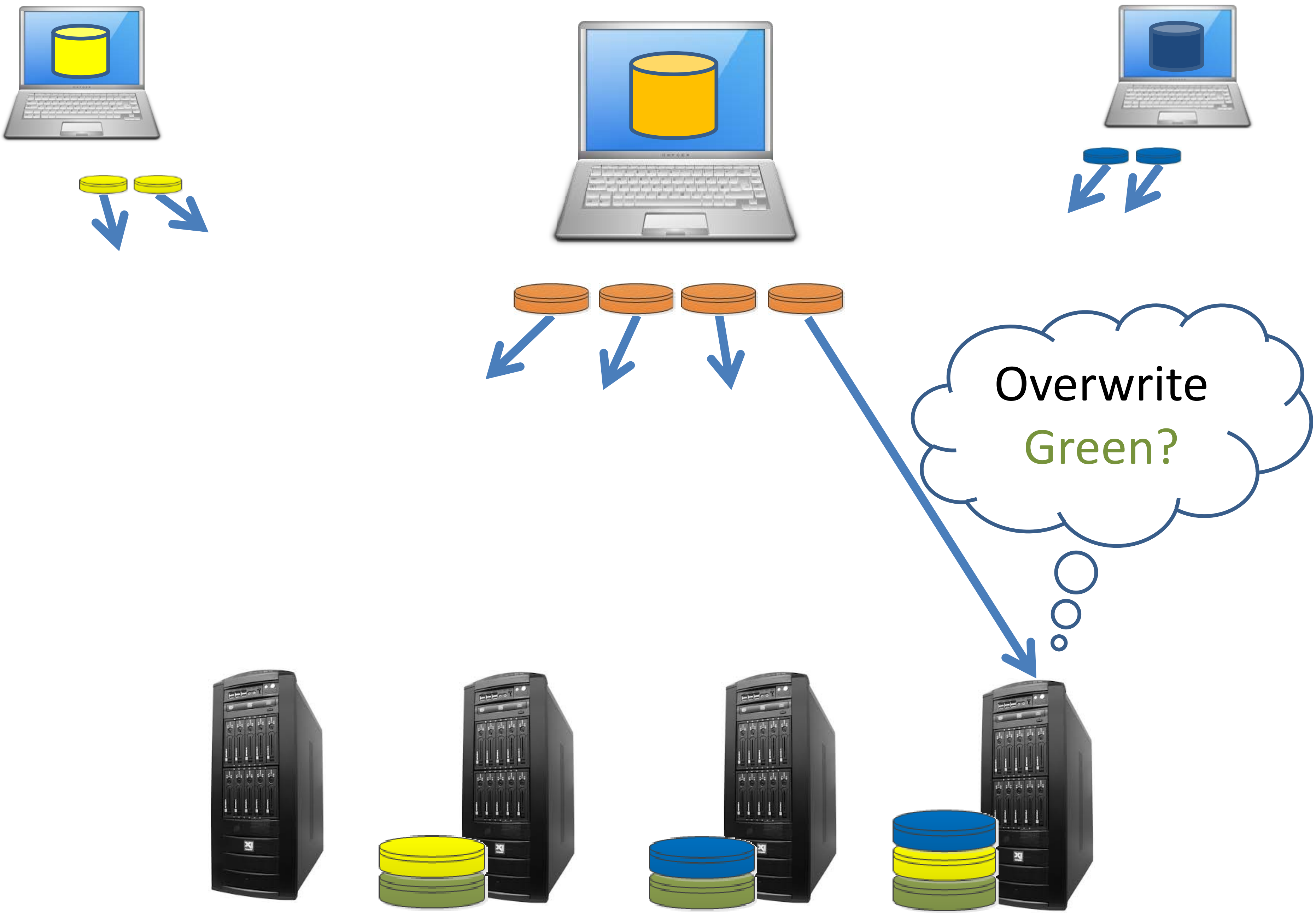
Back to Coding …

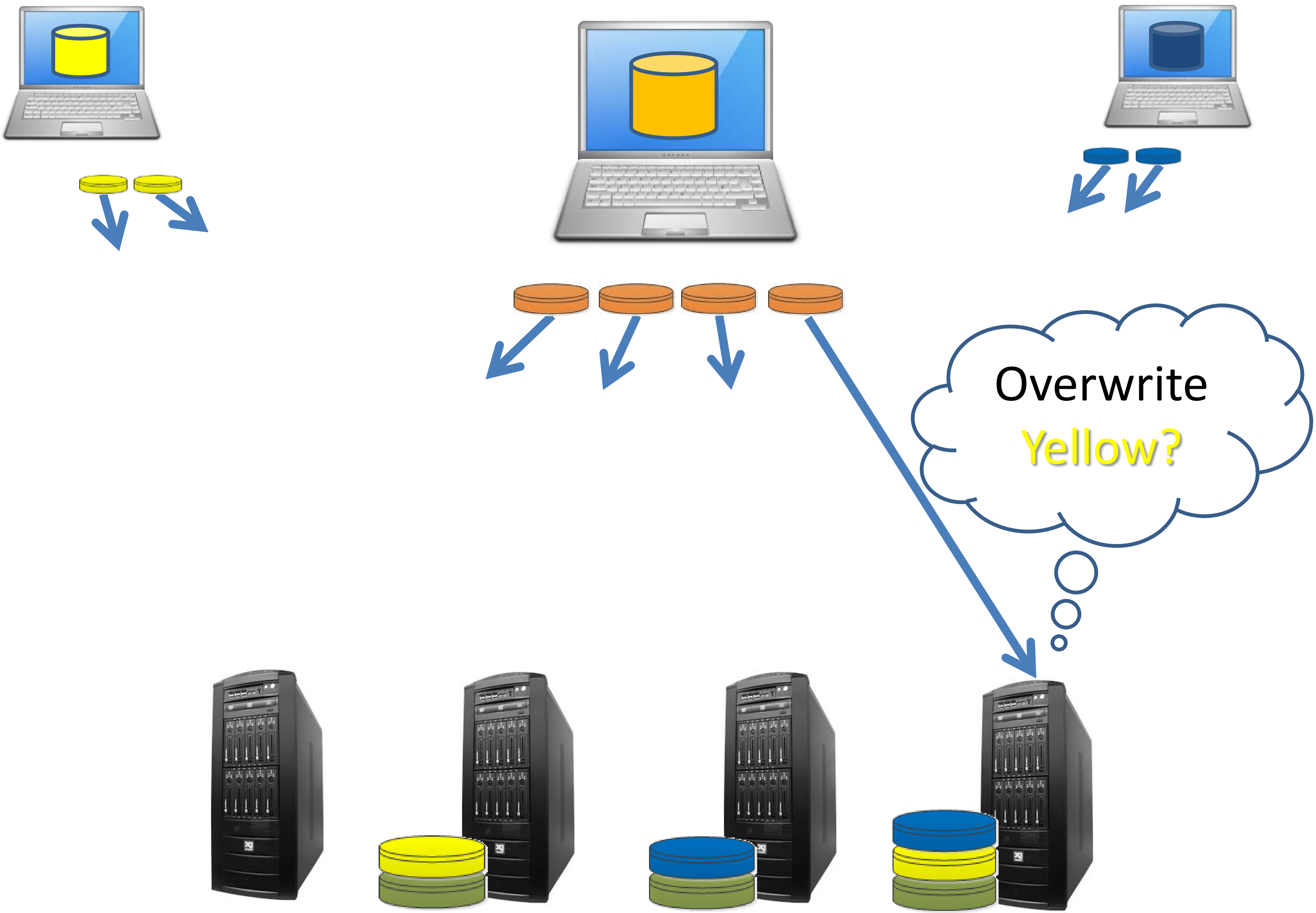


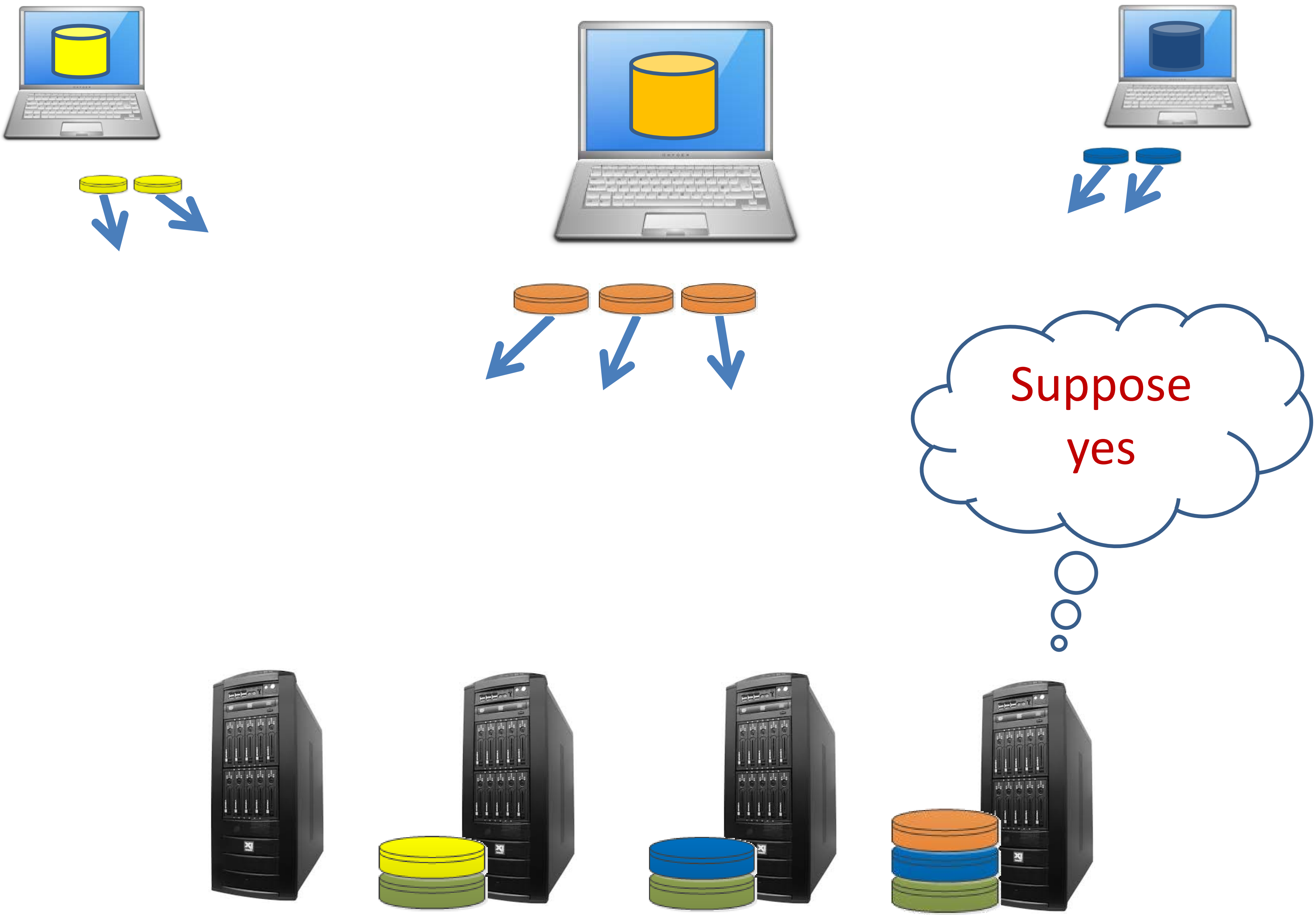

Overwrite?
No!

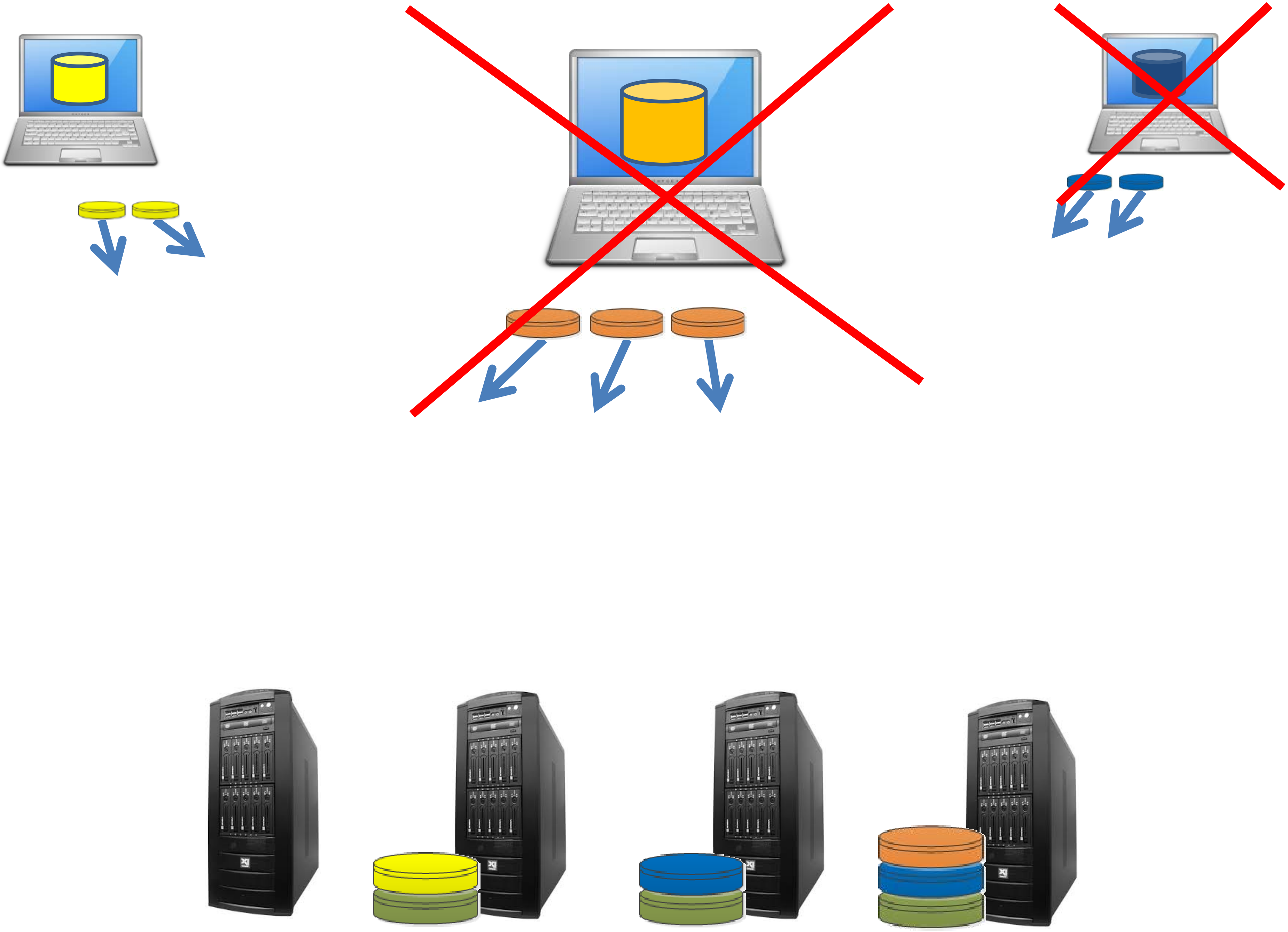


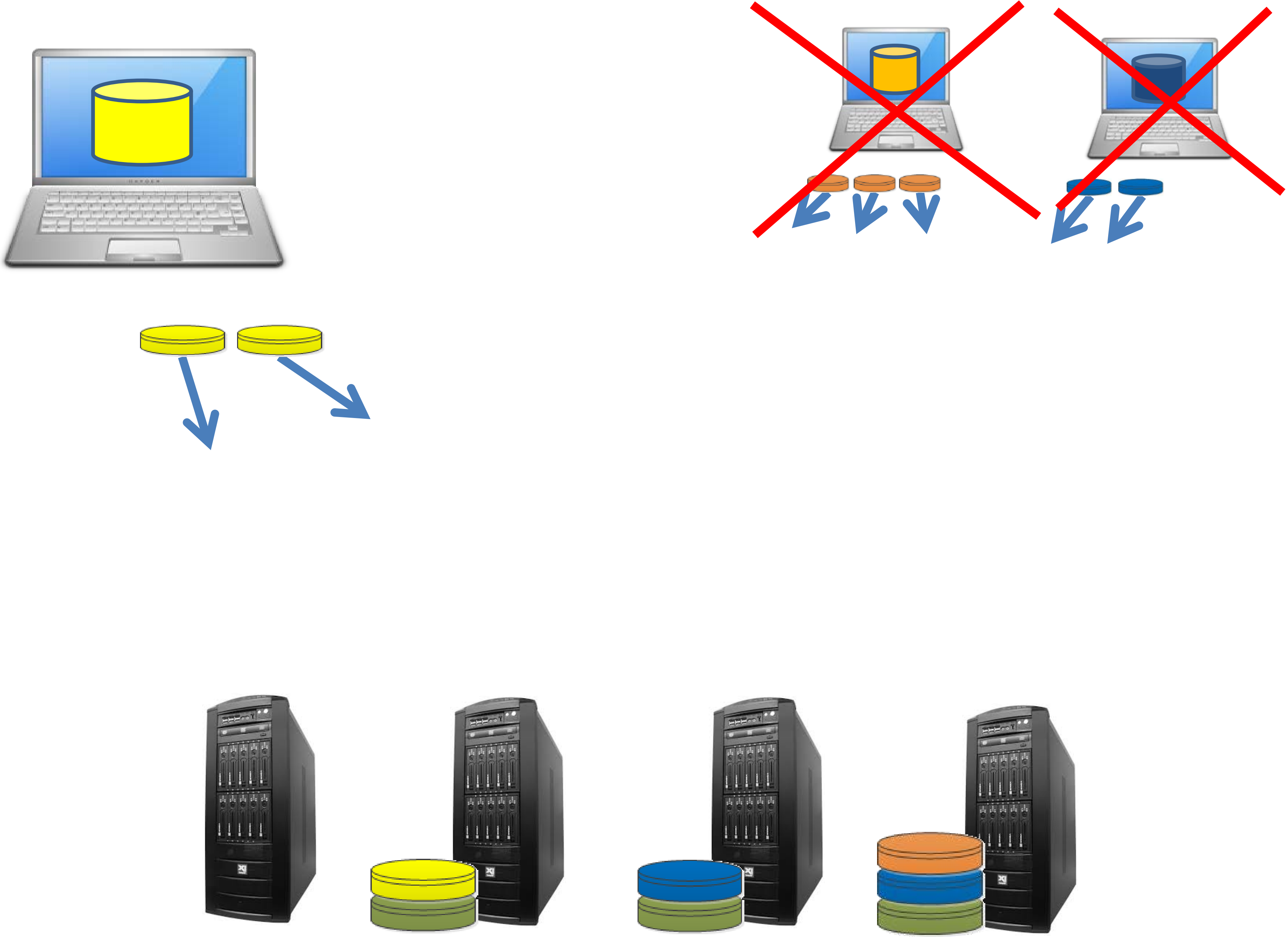


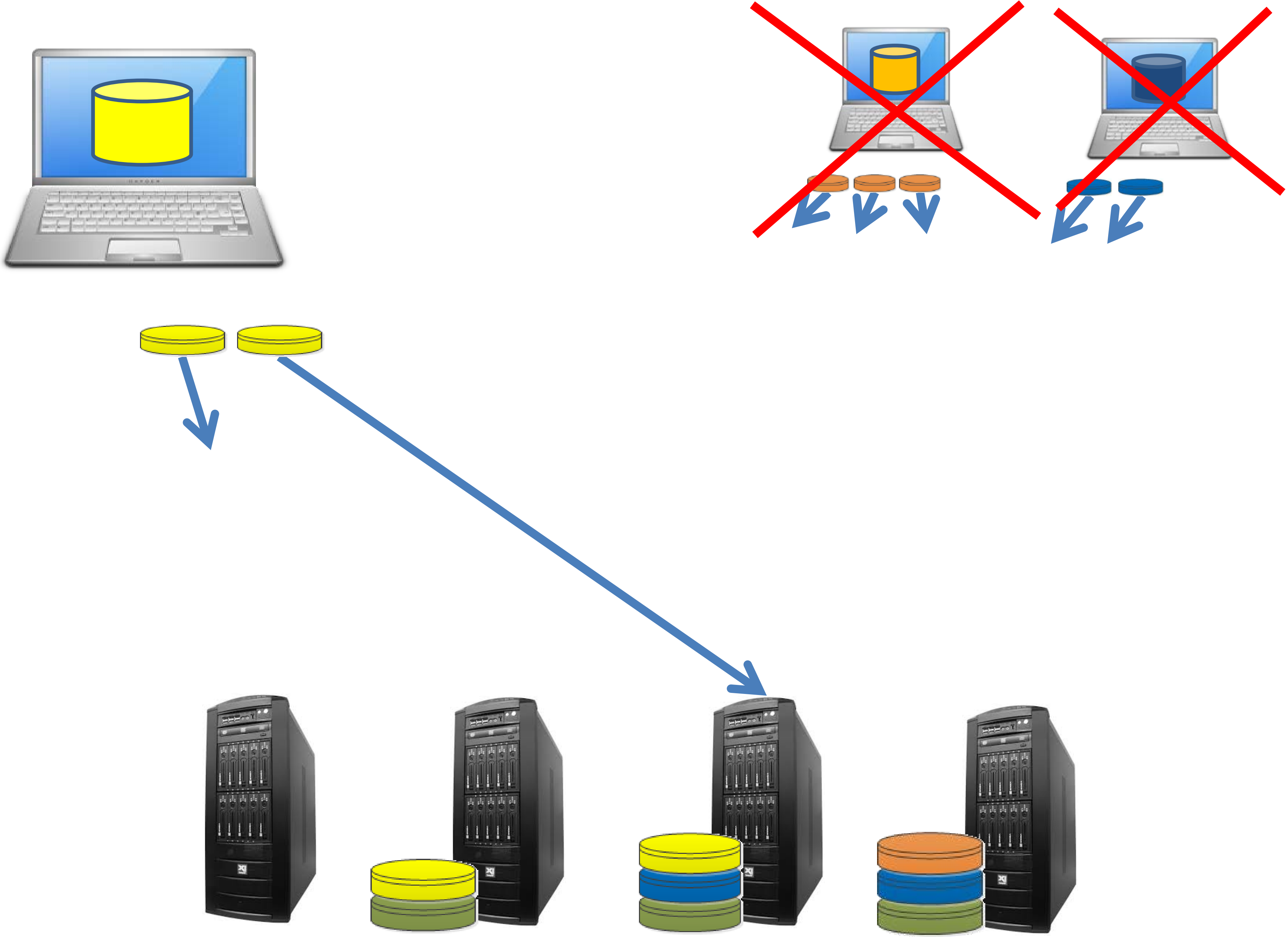

Overwrite
Green?

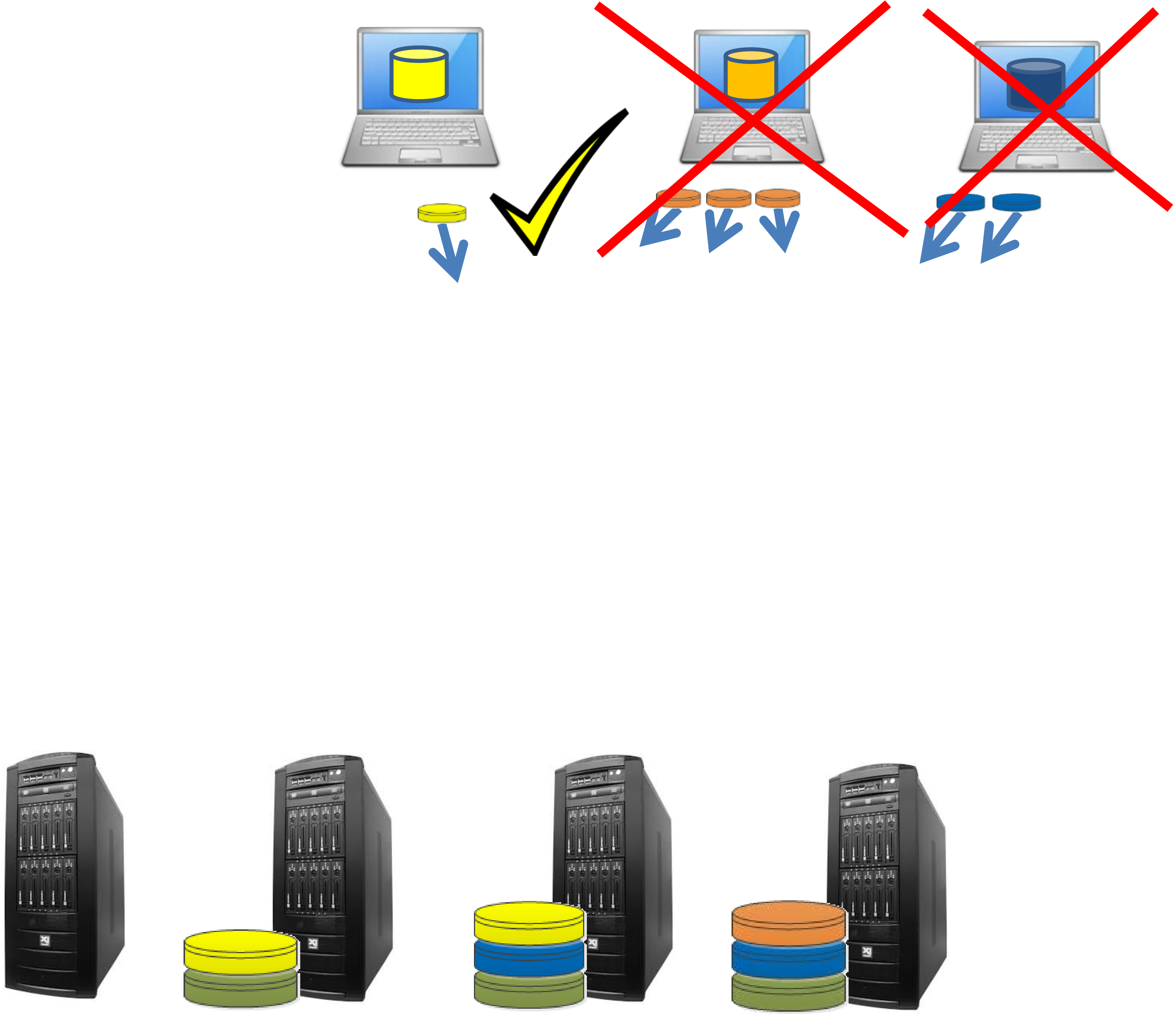

Overwrite
Yellow?

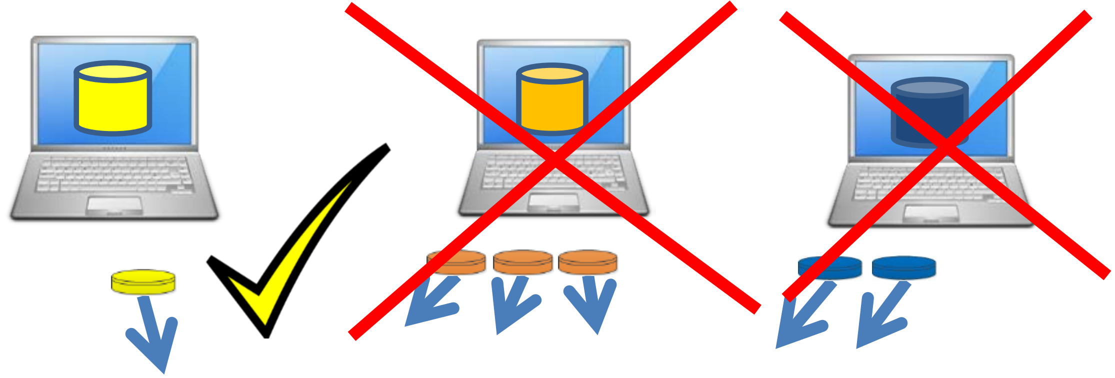

Suppose
yes

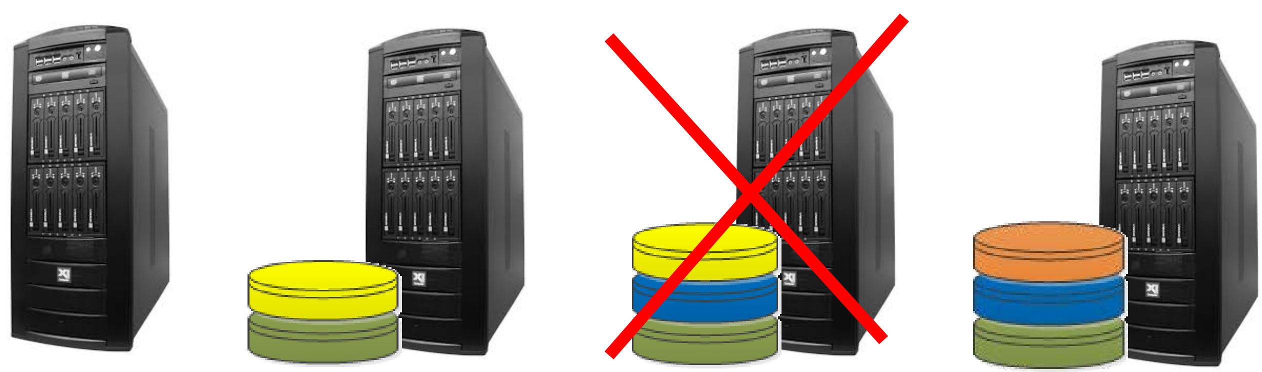


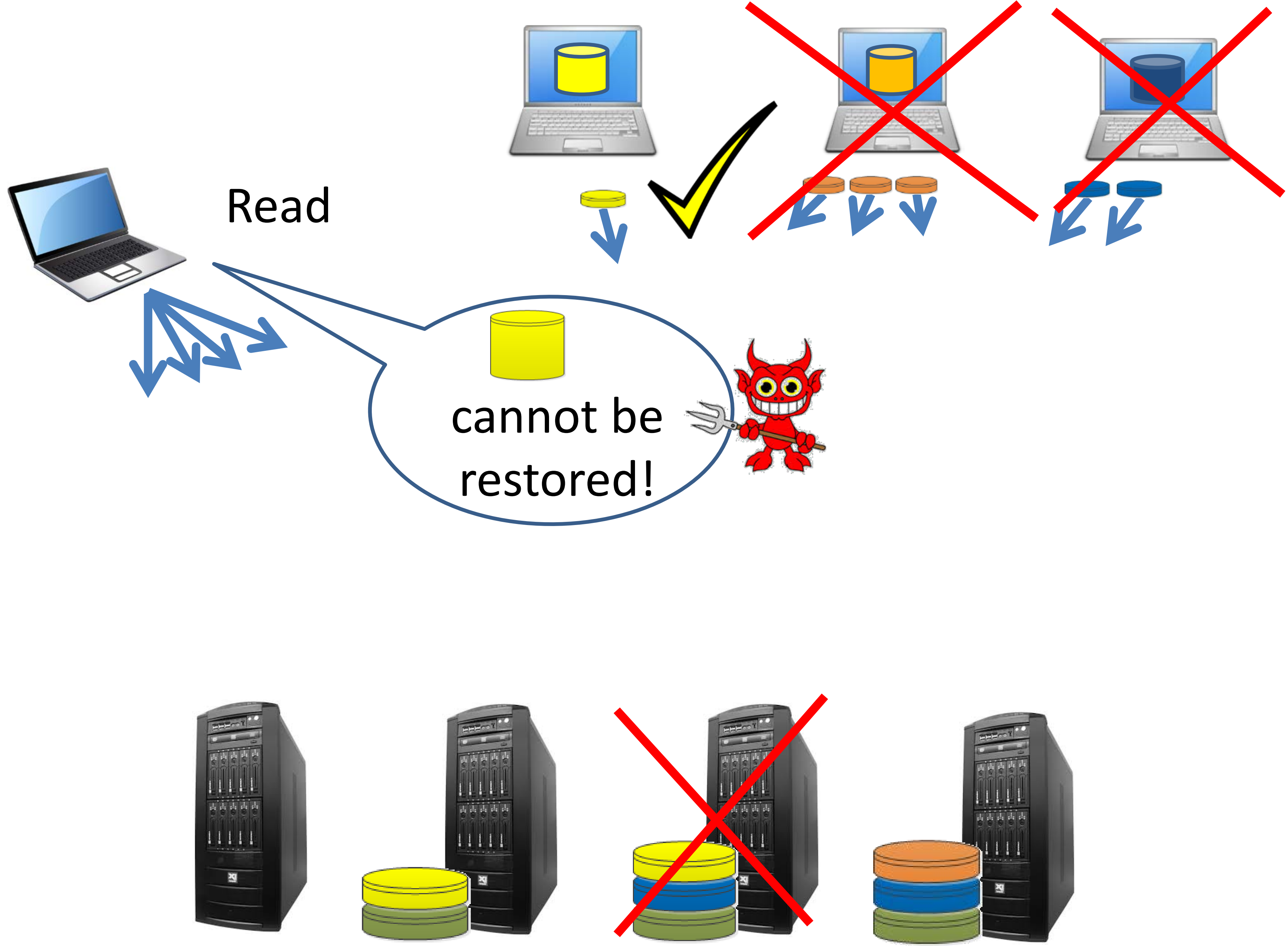


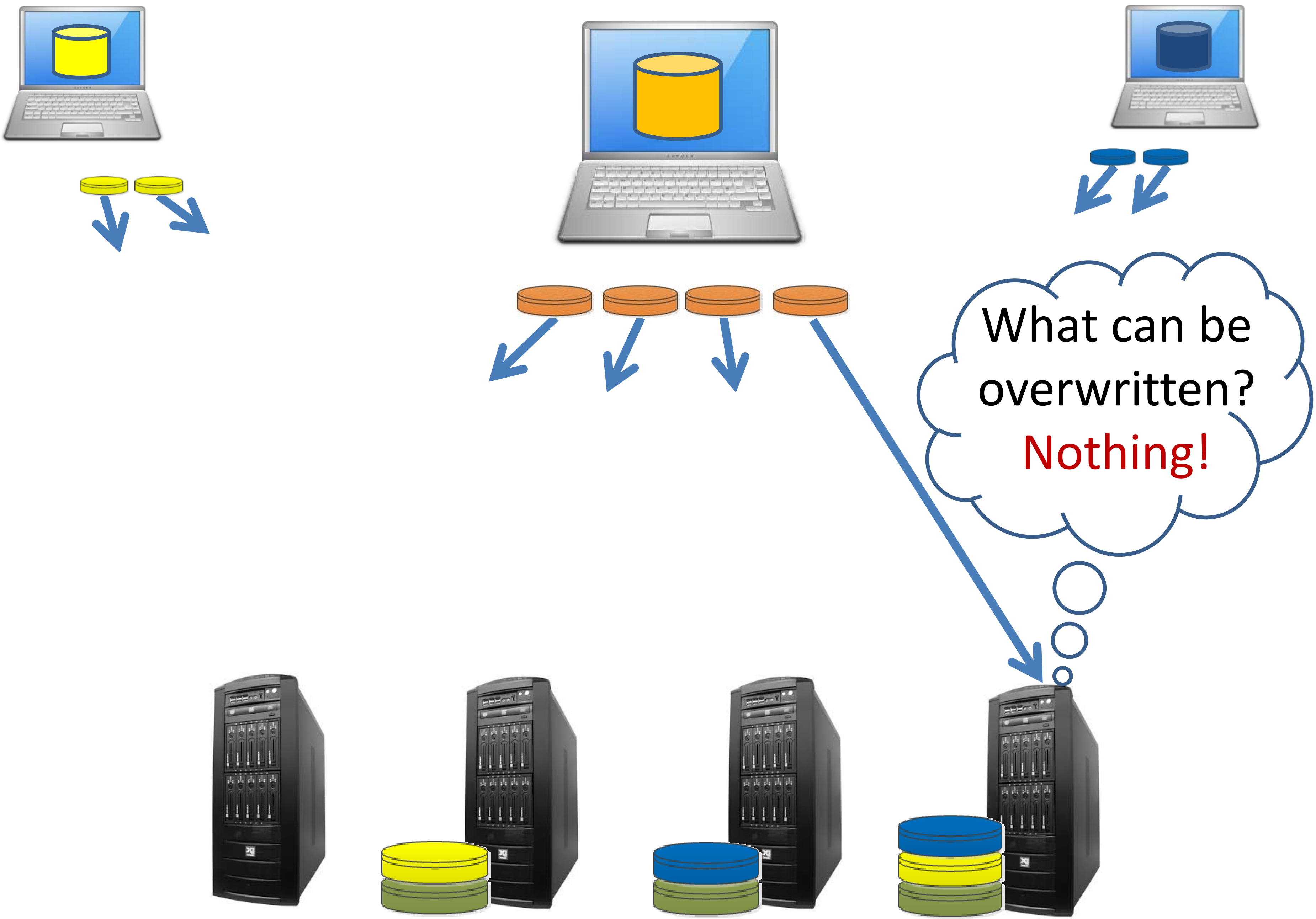


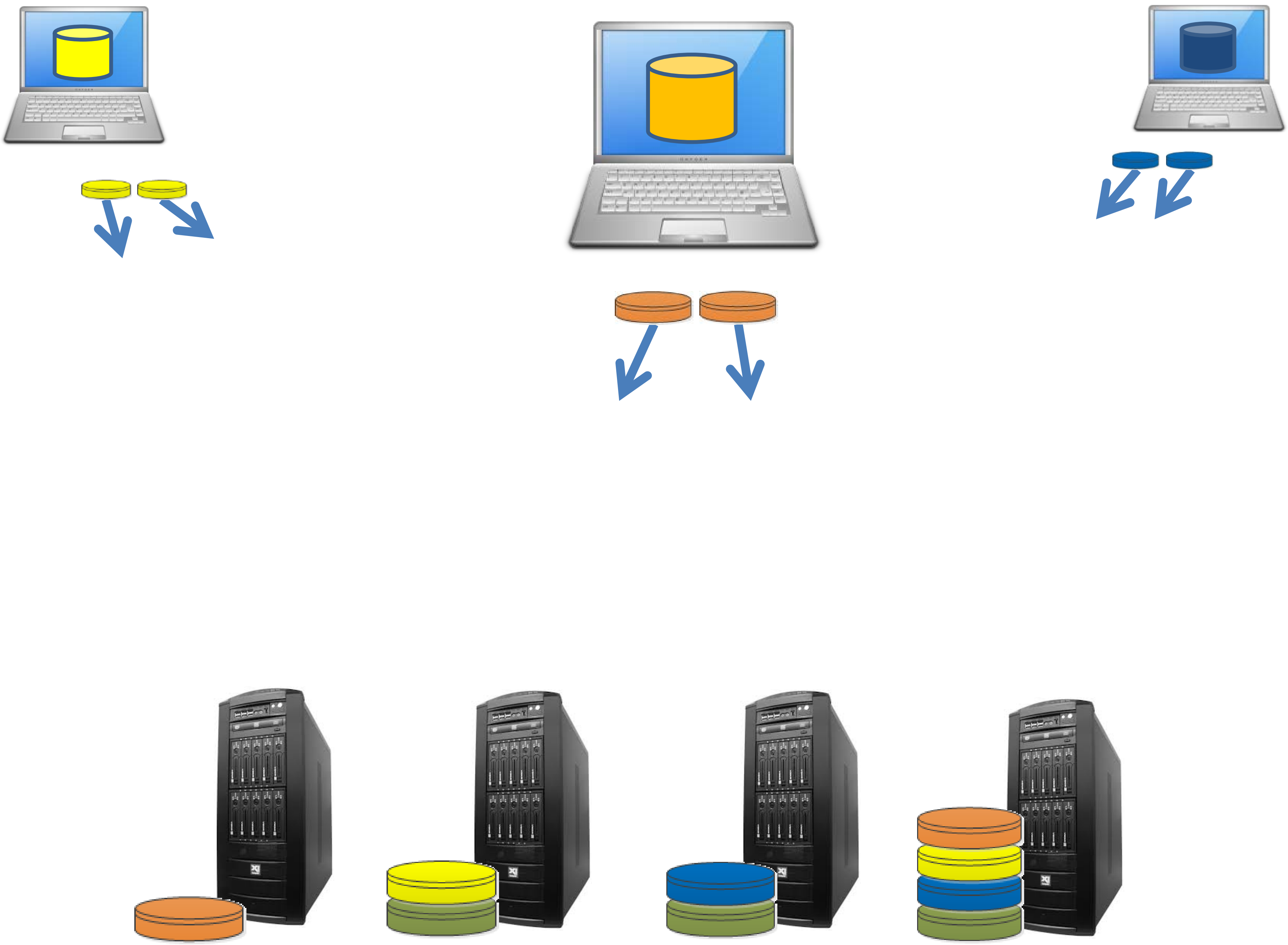


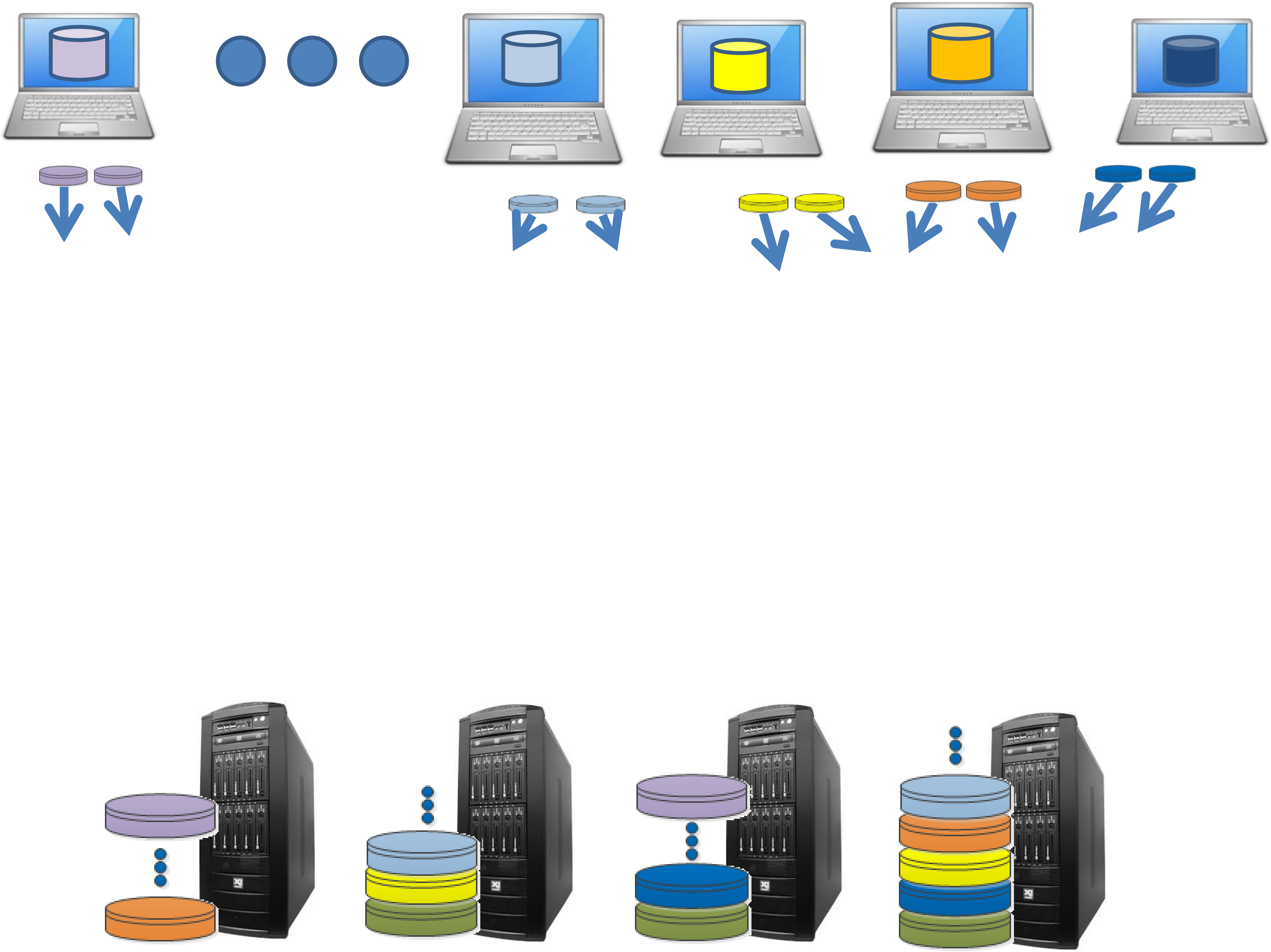


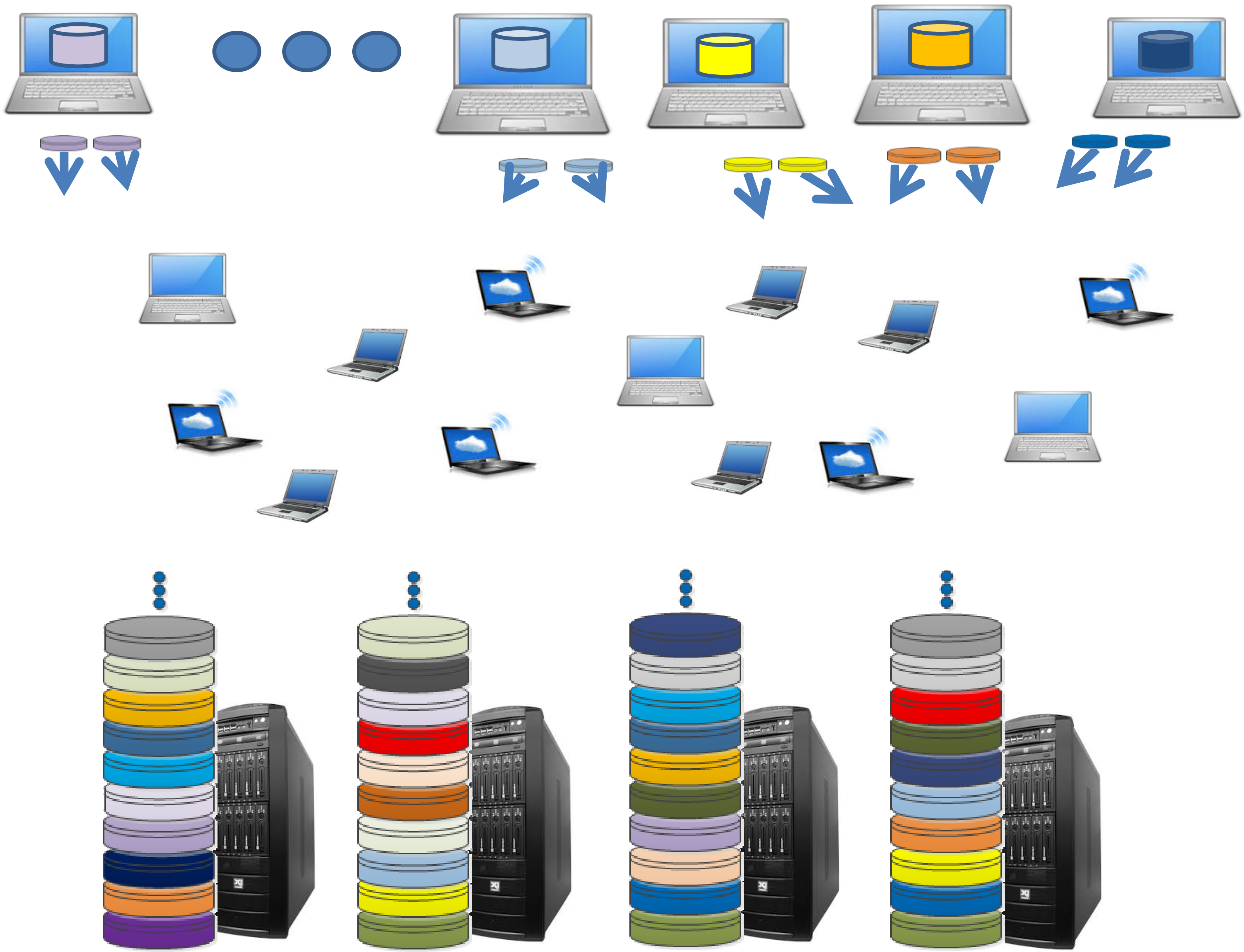



Read
cannot be
restored!


What can be overwritten?
Nothing!







Distributed Storage: Space Bounds

- Spiegelman et al. PODC 2016: $\Omega(D \cdot \min(f,c))$
  - Lock-free multi-writer
  - f failures,
  - c concurrent writes
  - Value size D
- Berger et al. DISC 2018: $\Omega(k \cdot \min(2^D, R))$
  - k-out-of-n coding
  - R visible readers; R infinite with invisible readers
  - Value size