Wait-Free Registers

Ellis Michael

Congrats! You're Now Paxos Experts!

Drawbacks of Paxos

Drawbacks of Paxos

- Leader is a single bottleneck, processes $O(n)$ messages on every request.

Drawbacks of Paxos

- Leader is a single bottleneck, processes $O(n)$ messages on every request.
- FLP means that liveness not guaranteed.

Drawbacks of Paxos

- Leader is a single bottleneck, processes $O(n)$ messages on every request.
- FLP means that liveness not guaranteed.
- More practically, Paxos can have bad availability during failure scenarios (e.g., if a leader fails, it takes time to elect a new one).

Alternatives

Alternatives

- Allow randomness (see Ben-Or lecture).

Alternatives

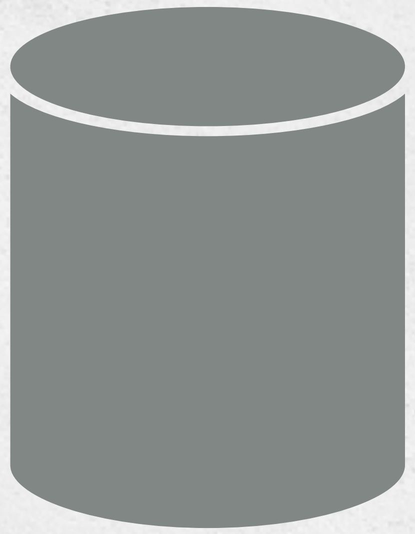
- Allow randomness (see Ben-Or lecture).
- Weaken the safety guarantees and accept weaker consistency (at your own peril).

Alternatives

- Allow randomness (see Ben-Or lecture).
- Weaken the safety guarantees and accept weaker consistency (at your own peril).
- Constrain the problem.

Registers

- Hold a single value. Want multiple values? Use multiple registers.
- Allows reads and writes only. Does not allow appends or other read-modify-write operations.
- Recall **safe**, **regular**, and **atomic**/**linearizable** semantics. We want *linearizability*.



REGISTERS

- Hold a single value. Want multiple values? Use multiple registers.
- Allows reads and writes only. Does not allow appends or other read-modify-write operations.
- Recall **safe**, **regular**, and **atomic/linearizable** semantics. We want *linearizability*.

Registers

- Hold a single value. Want multiple values? Use multiple registers.
- Allows reads and writes only. Does not allow appends or other read-modify-write operations.
- Recall **safe**, **regular**, and **atomic/linearizable** semantics. We want *linearizability*.

Registers

- Hold a single value. Want multiple values? Use multiple registers.
- Allows reads and writes only. Does not allow appends or other read-modify-write operations.
- Recall **safe**, **regular**, and **atomic/linearizable** semantics. We want *linearizability*.

Registers

- Hold a single value. Want multiple values? Use multiple registers.
- Allows reads and writes only. Does not allow appends or other read-modify-write operations.
- Recall **safe**, **regular**, and **atomic/linearizable** semantics. We want *linearizability*.

Registers

- Hold a single value. Want multiple values? Use multiple registers.
- Allows reads and writes only. Does not allow appends or other read-modify-write operations.
- Recall **safe**, **regular**, and **atomic/linearizable** semantics. We want *linearizability*.

Registers

- Hold a single value. Want multiple values? Use multiple registers.
- Allows reads and writes only. Does not allow appends or other read-modify-write operations.
- Recall **safe**, **regular**, and **atomic/linearizable** semantics. We want *linearizability*.

Why No Appends?

Simple way to implement consensus:

Why No Appends?

Simple way to implement consensus:

- All processes append their input value.

Why No Appends?

Simple way to implement consensus:

- All processes append their input value.
- All processes read the value.

Why No Appends?

Simple way to implement consensus:

- All processes append their input value.
- All processes read the value.
- They all decide the first value that was appended.

Why No Appends?

Simple way to implement consensus:

- All proce… .
- All proce…
- They all decide the first value that was appended.

If you can wait-free implement an appendable register, you can solve consensus (safety and liveness), which is impossible.

Implementing a Register

- We will use the **client**/**server** model, where servers are replicas storing the value and clients send **reads** and **writes**.
- We want linearizability of reads and writes.
- As usual, we want to tolerate up to f server *crash failures*. Clients can also fail by crashing.

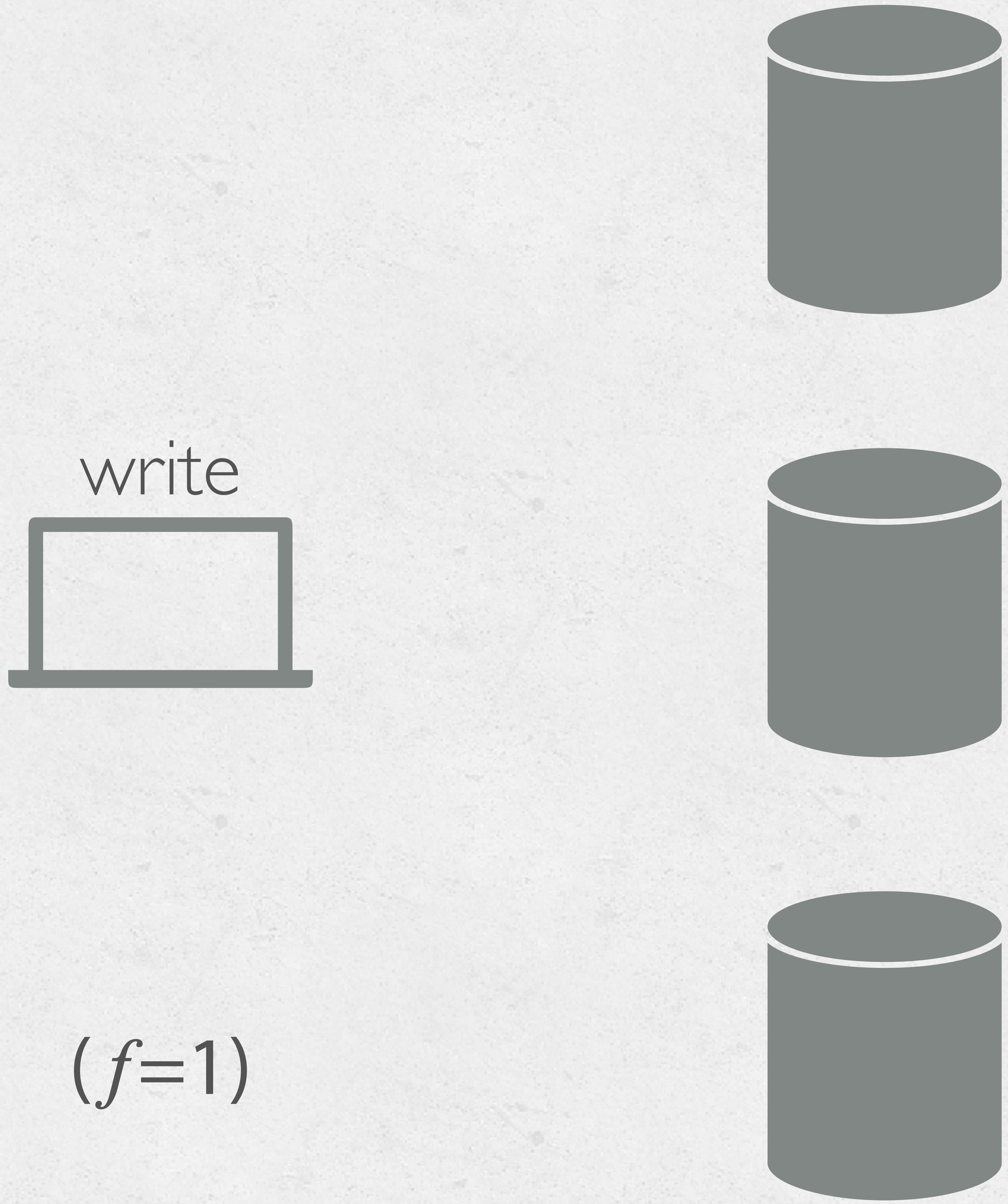
Non-Blocking Algorithms

- **Lock-free** algorithms guarantee system-wide progress.
- **Wait-free** algorithms guarantee per-client progress. That is, no matter what steps other processes take, a correct client's operations are always completed in a finite number of steps.

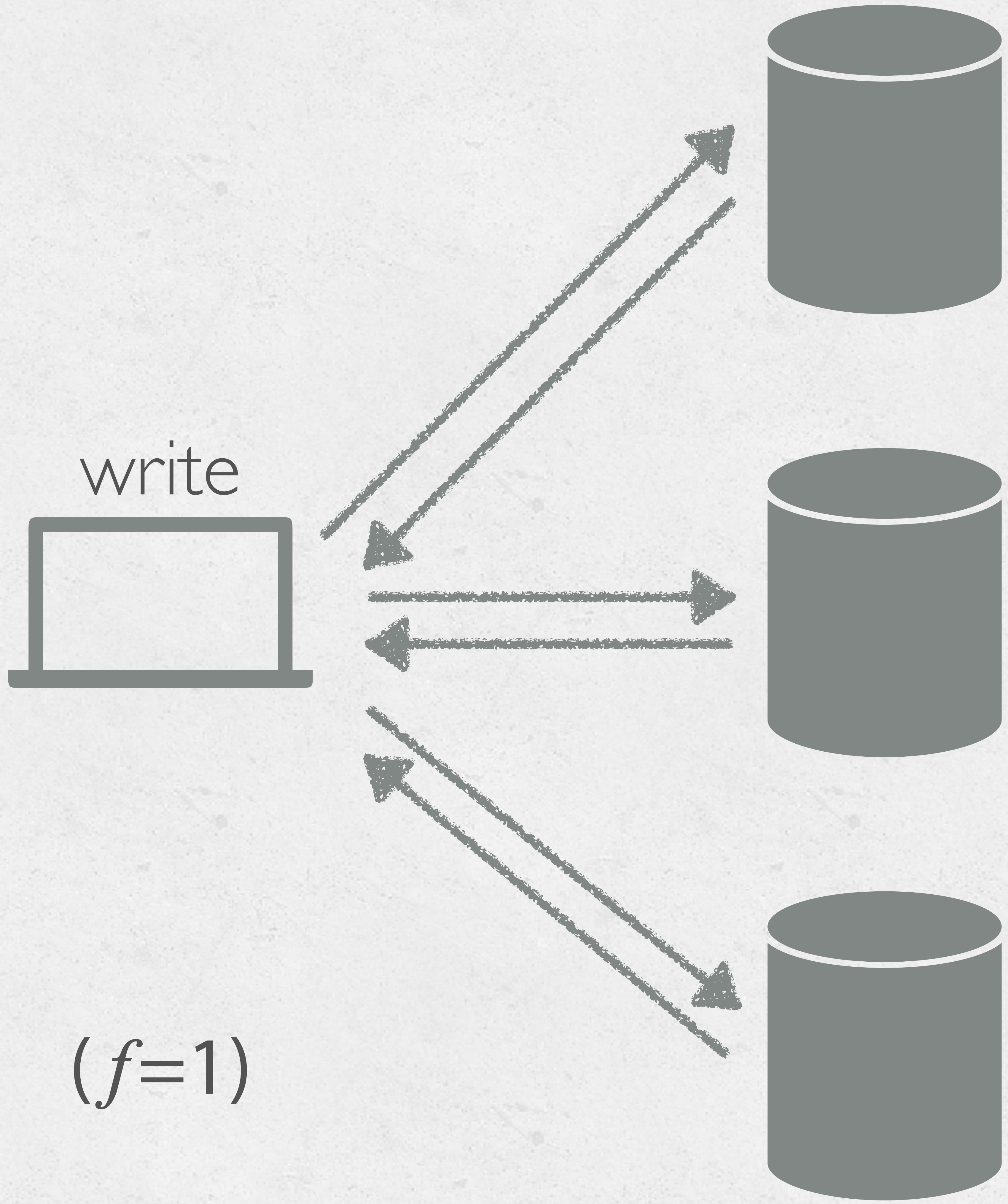
Non-Blocking Algorithms

- **Lock-free** algorithms guarantee system-wide progress.
- **Wait-free** algorithms guarantee per-client progress. That is, no matter what steps other processes take, a correct client's operations are always completed in a finite number of steps.

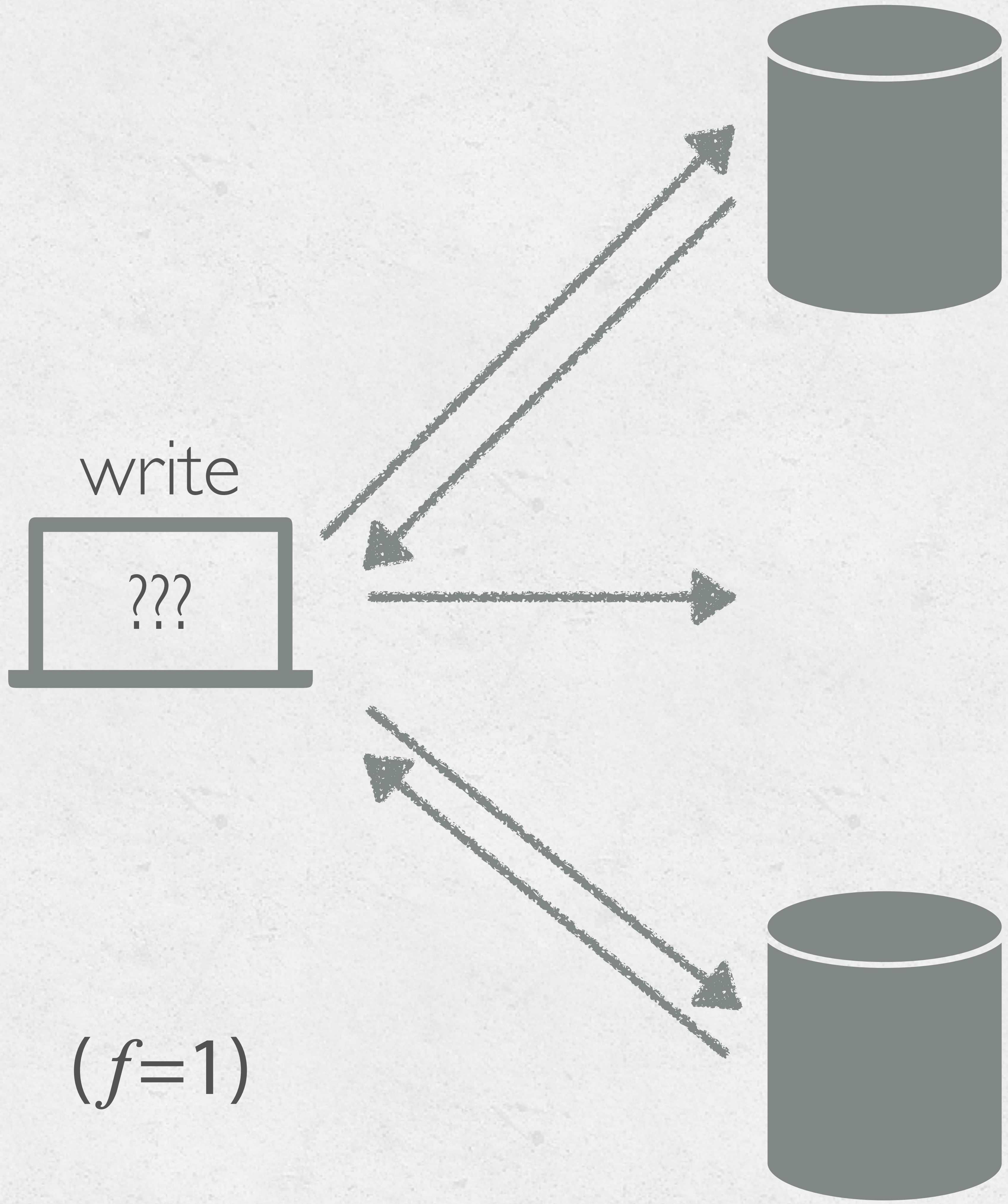
How Many Servers Do We Need?



How Many Servers Do We Need?

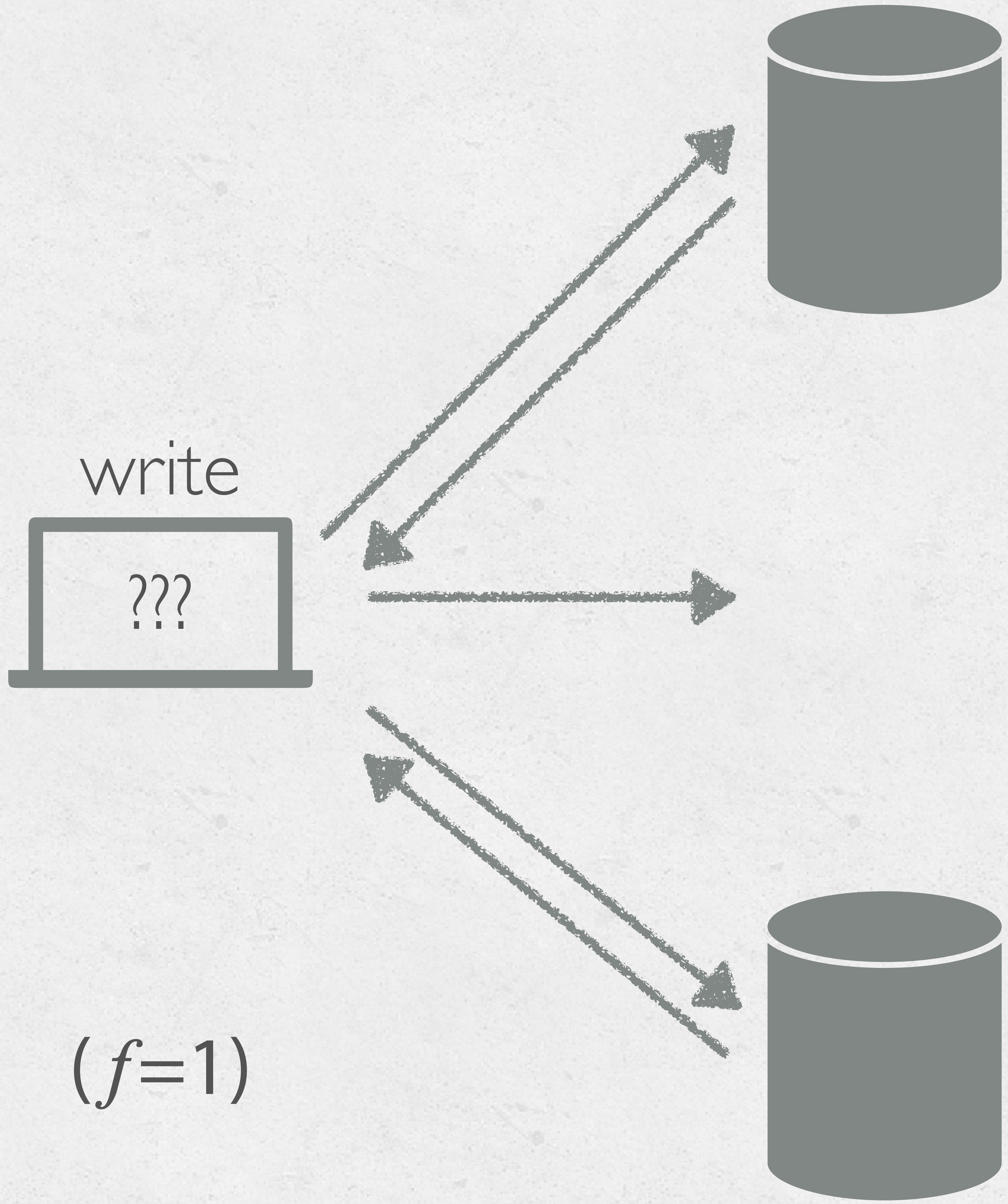


How Many Servers Do We Need?



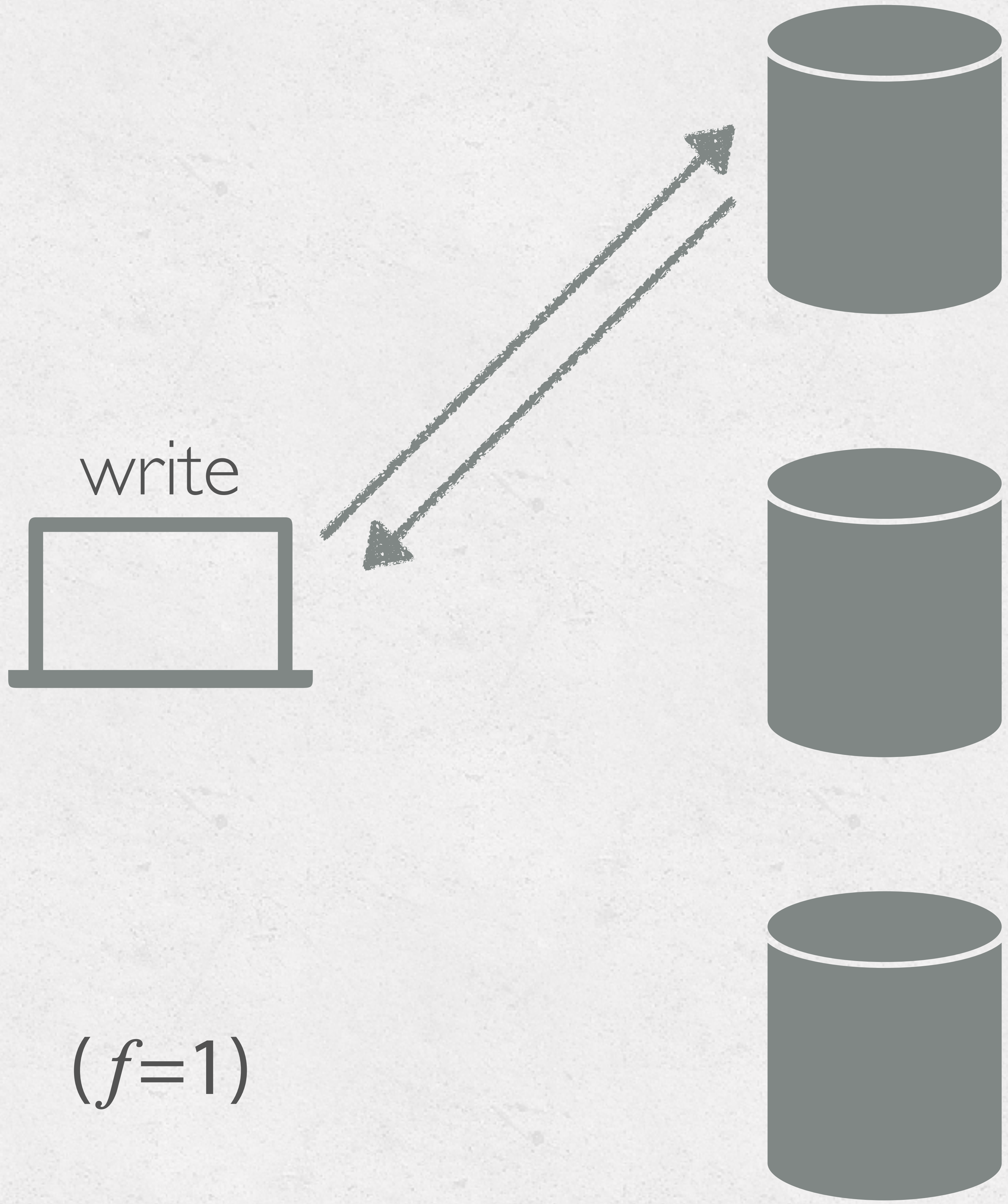
How Many Servers Do We Need?

- If we want to make progress even when f servers crash, we can wait for at most n - f responses.



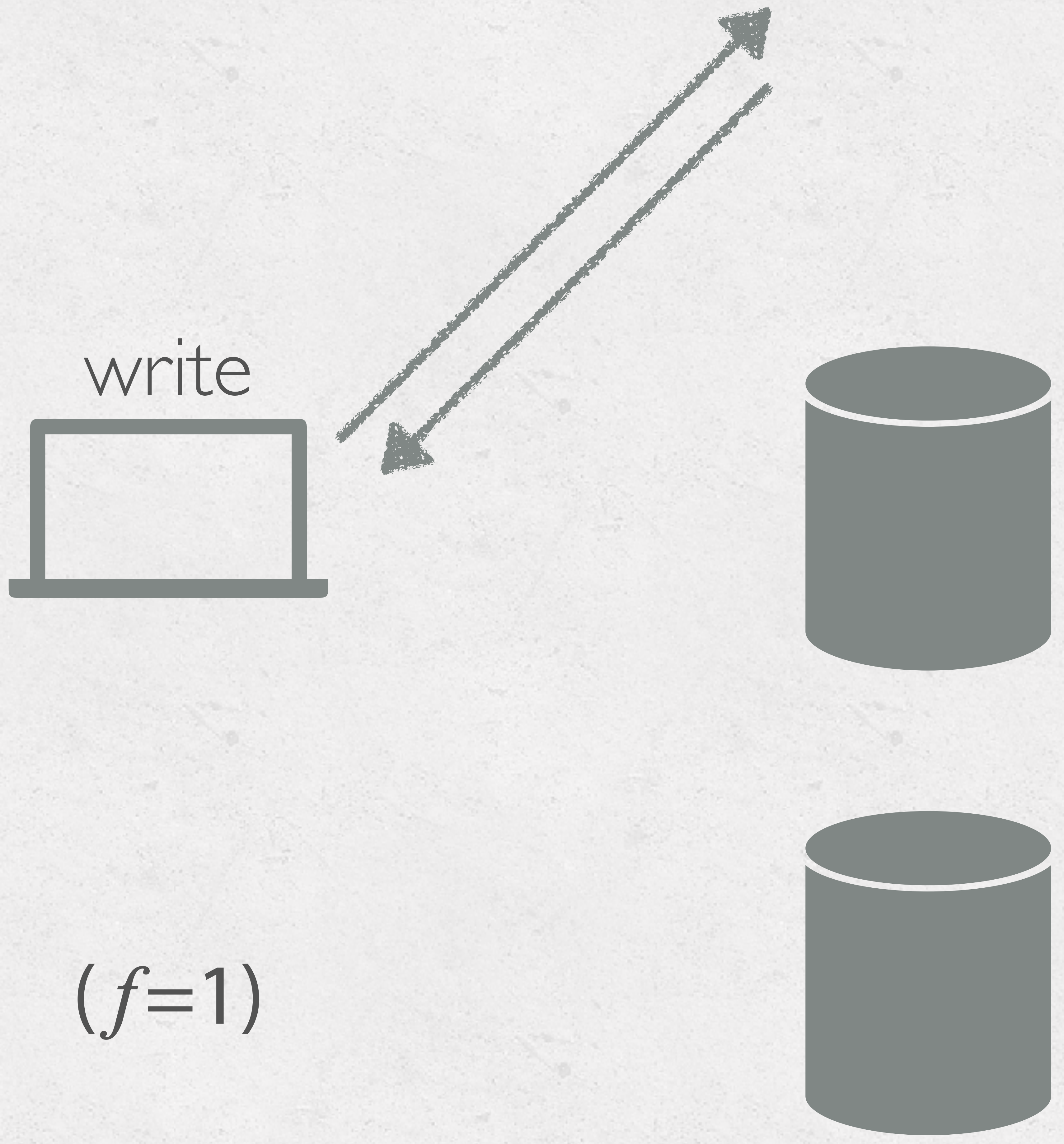
How Many Servers Do We Need?

- If we want to make progress even when f servers crash, we can wait for at most n - f responses.



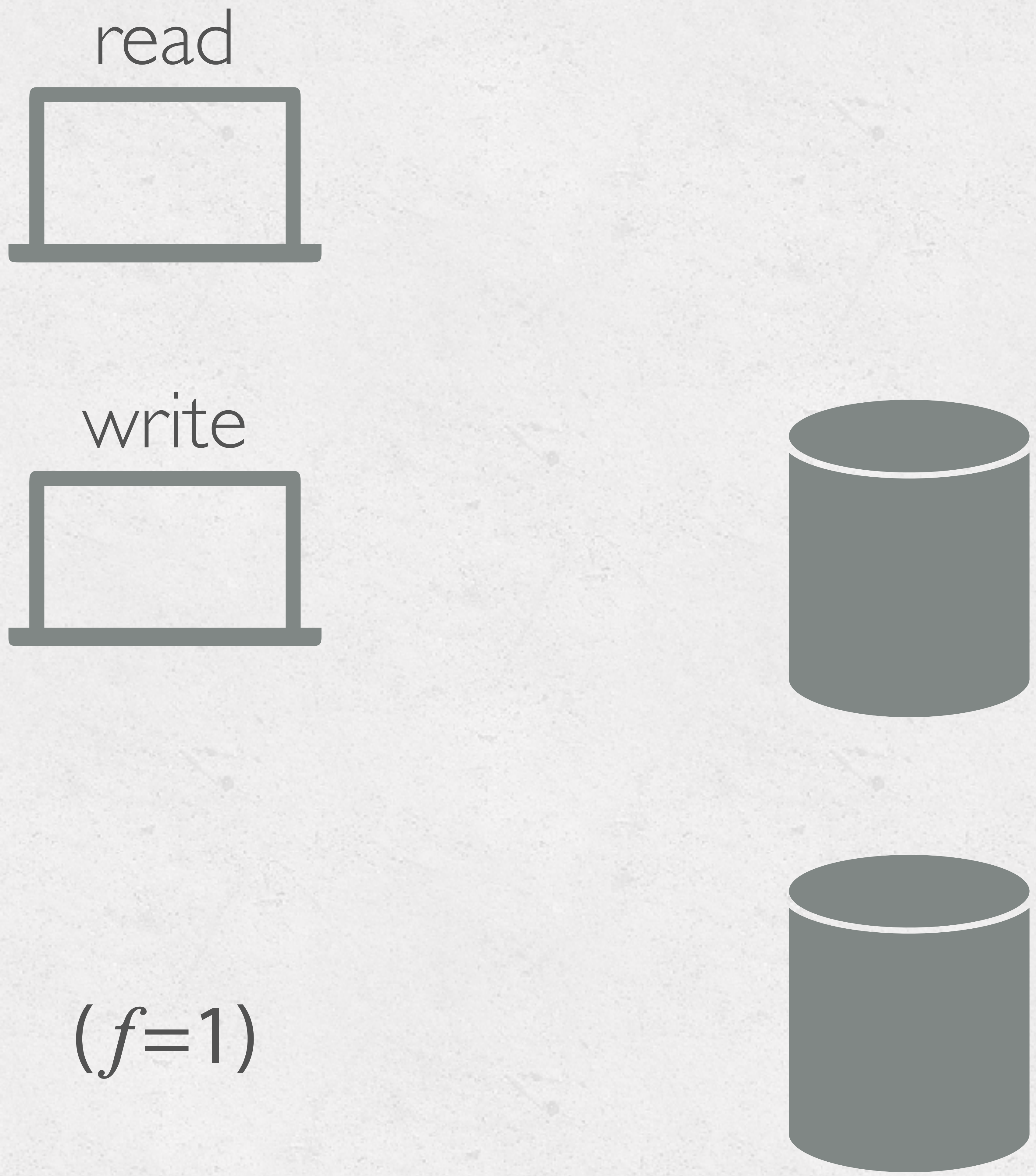
How Many Servers Do We Need?

- If we want to make progress even when f servers crash, we can wait for at most n - f responses.



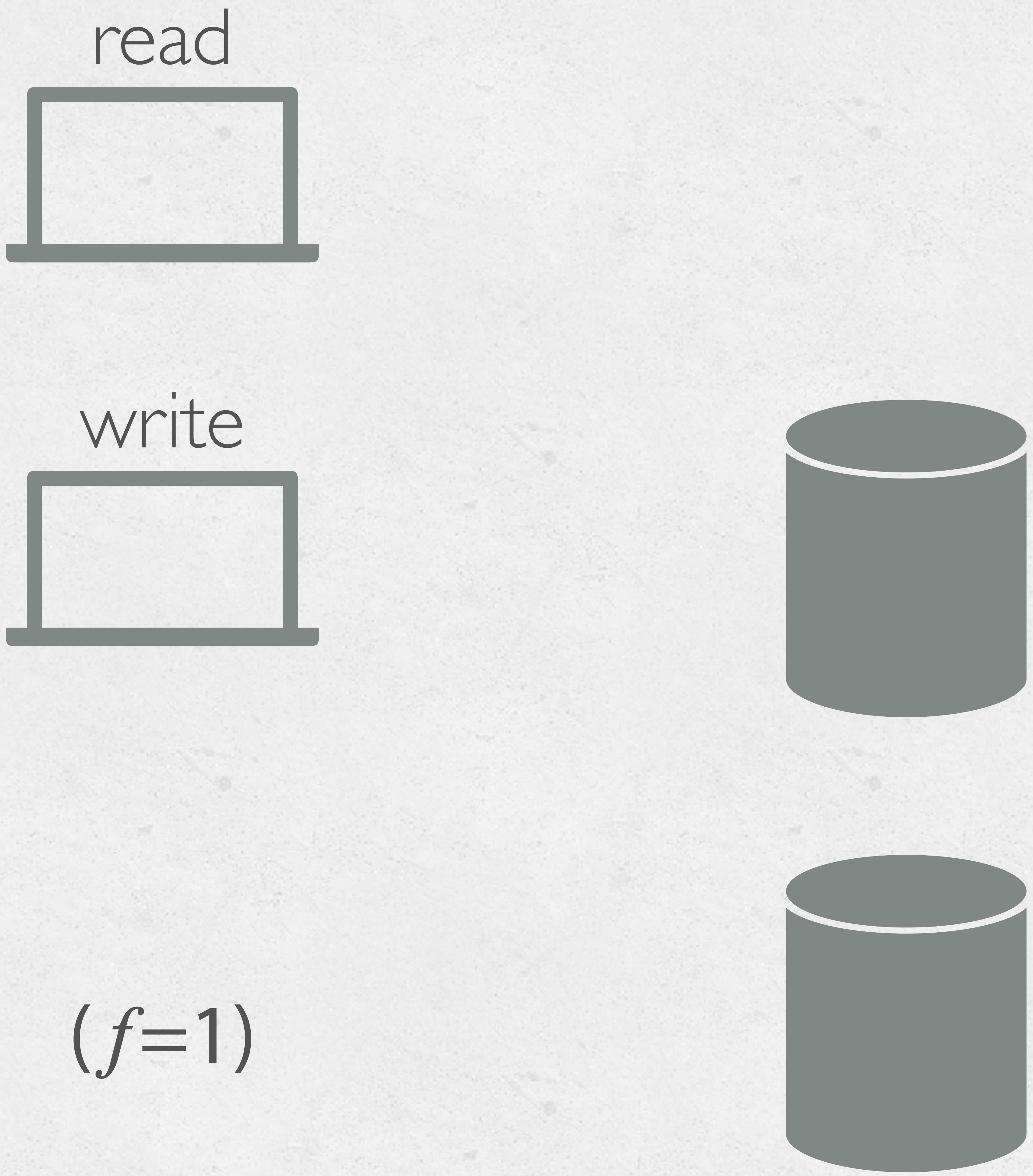
How Many Servers Do We Need?

- If we want to make progress even when f servers crash, we can wait for at most n - f responses.



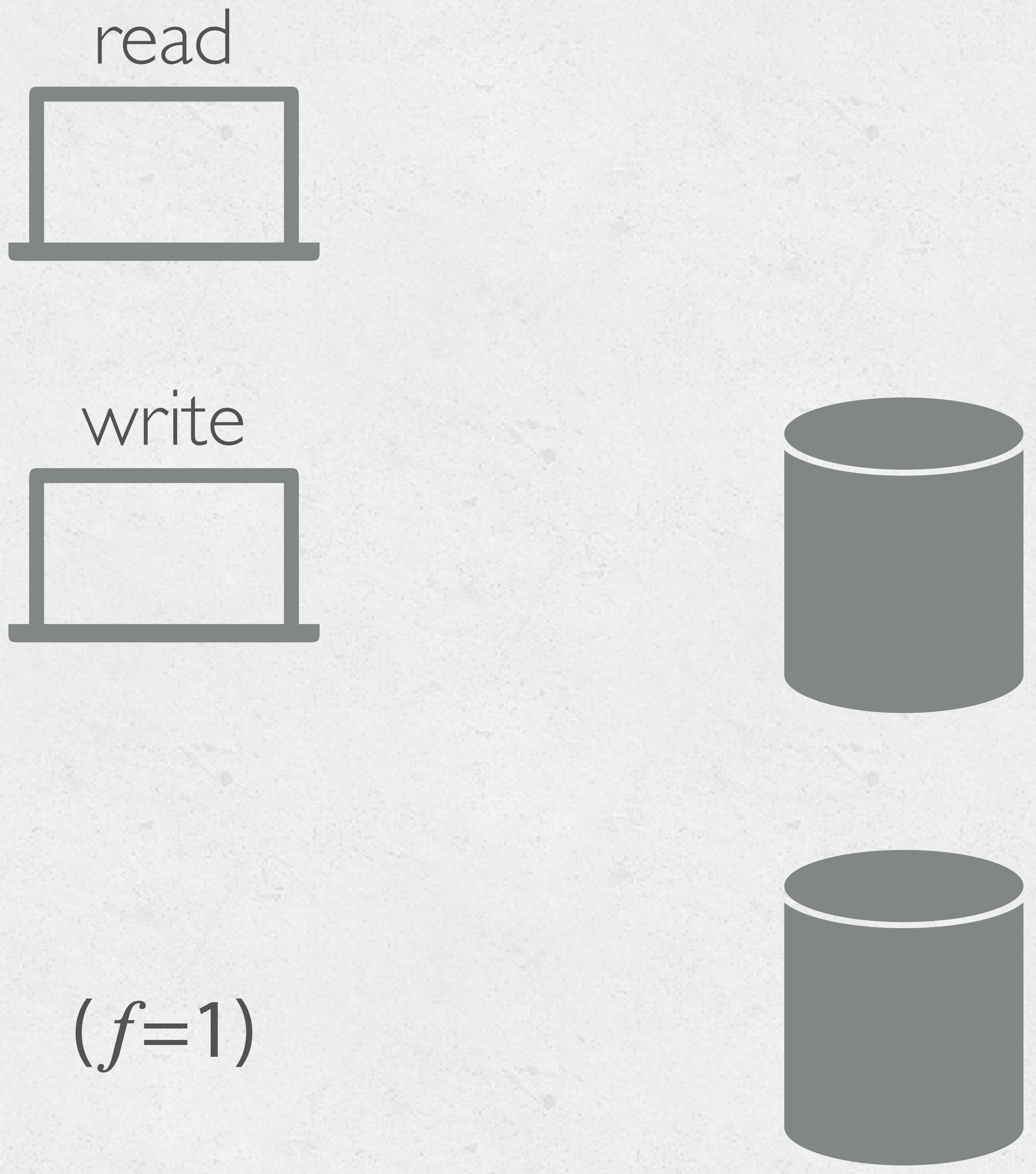
How Many Servers Do We Need?

- If we want to make progress even when f servers crash, we can wait for at most n - f responses.
- We need to send writes to $> f$ replicas, otherwise they could get lost forever.



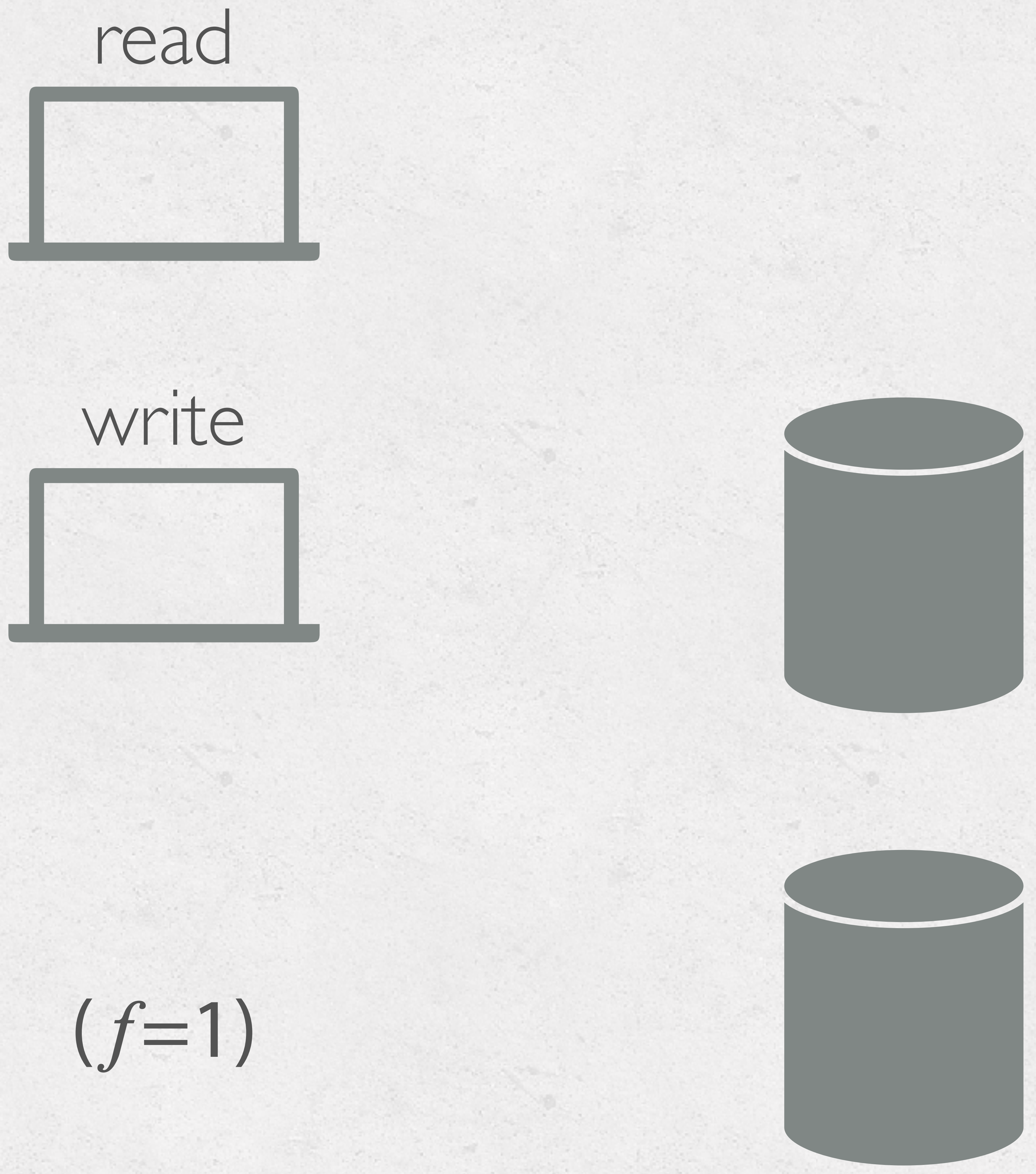
How Many Servers Do We Need?

- If we want to make progress even when f servers crash, we can wait for at most $n - f$ responses.
- We need to send writes to $> f$ replicas, otherwise they could get lost forever.
- So we need *at least* $2f+1$ servers. And, in fact, we will use $2f+1$.



How Many Servers Do We Need?

- If we want to make progress even when f servers crash, we can wait for at most n - f responses.
- We need to send writes to $> f$ replicas, otherwise they could get lost forever.
- So we need *at least* $2f+1$ servers. And, in fact, we will use $2f+1$.
- Read quorum size plus write quorum size should be greater than n (i.e., they should overlap). We'll use simple majorities.



First Step: Single Reader, Single Writer (SWSR)

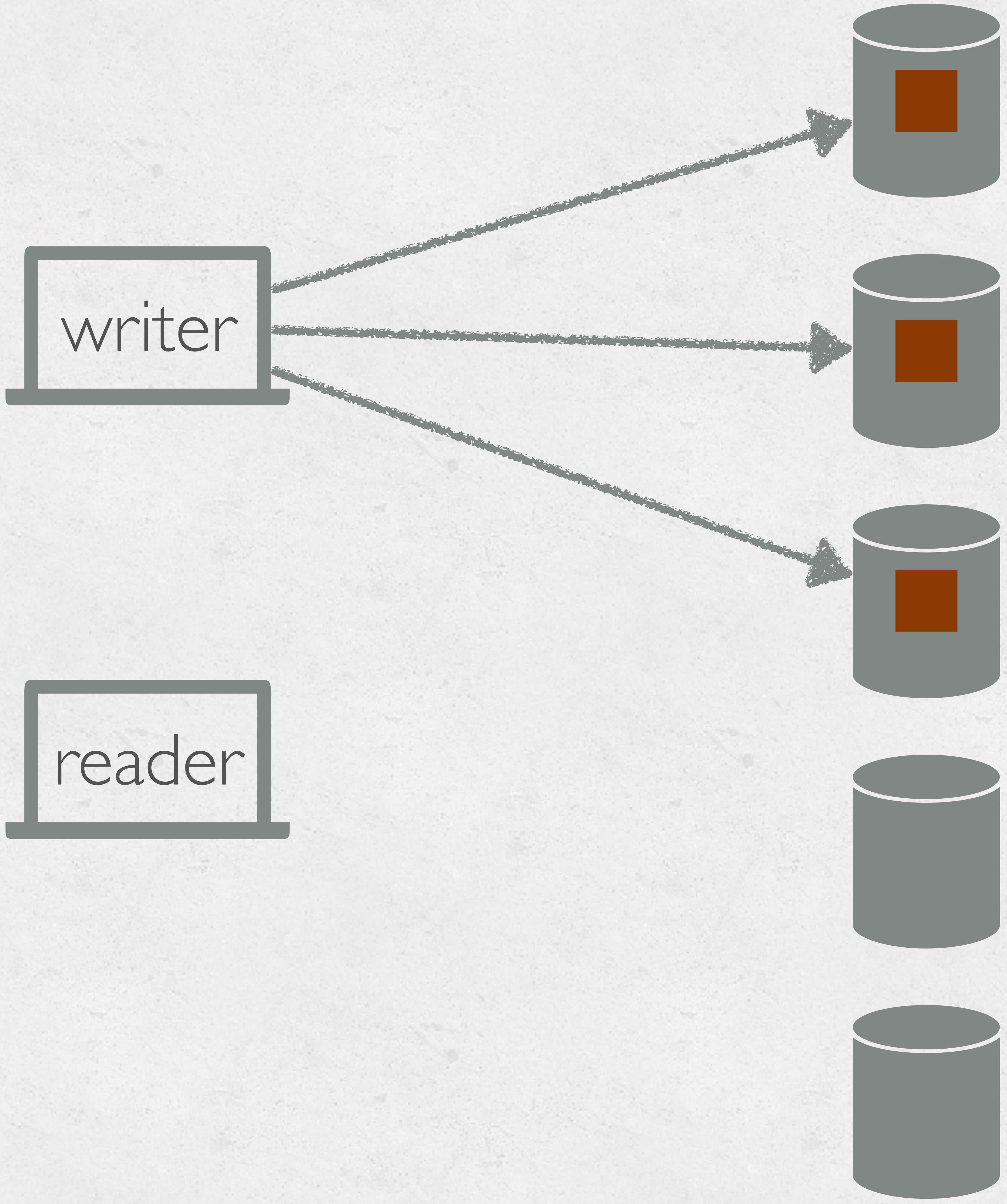
- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

Does this work?

First Step: Single Reader, Single Writer (SWSR)

- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

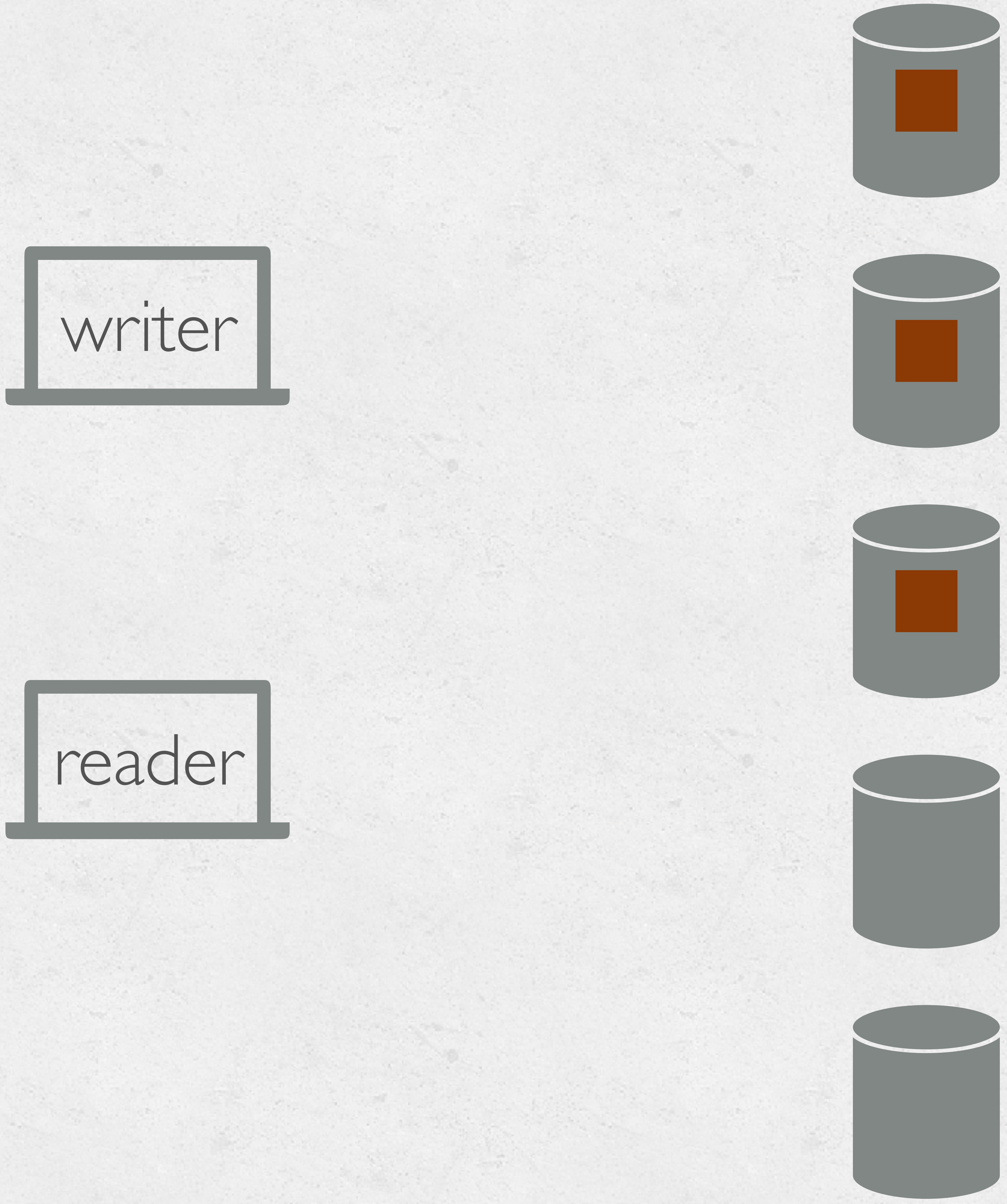
Does this work?



First Step: Single Reader, Single Writer (SWSR)

- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

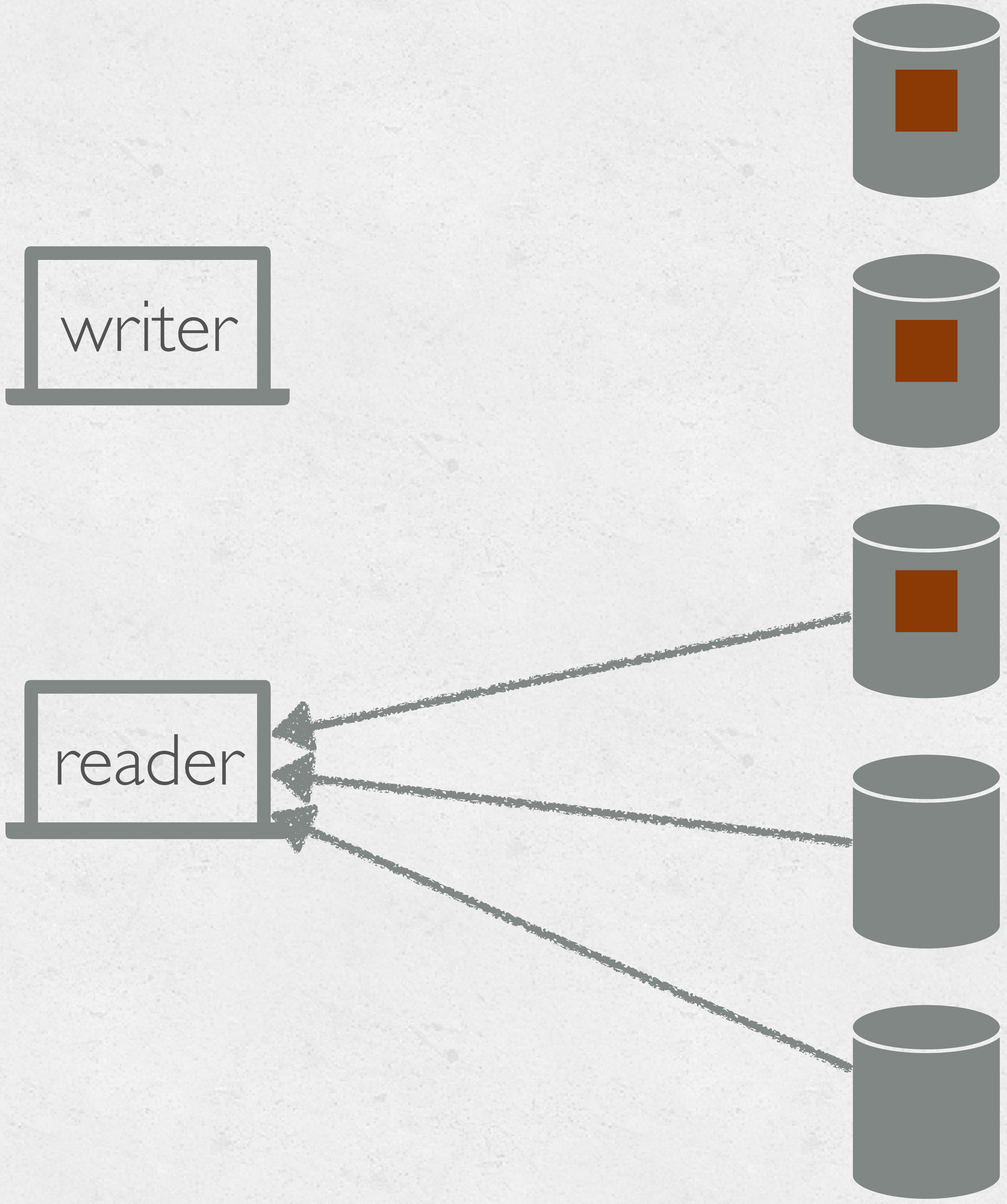
Does this work?



First Step: Single Reader, Single Writer (SWSR)

- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

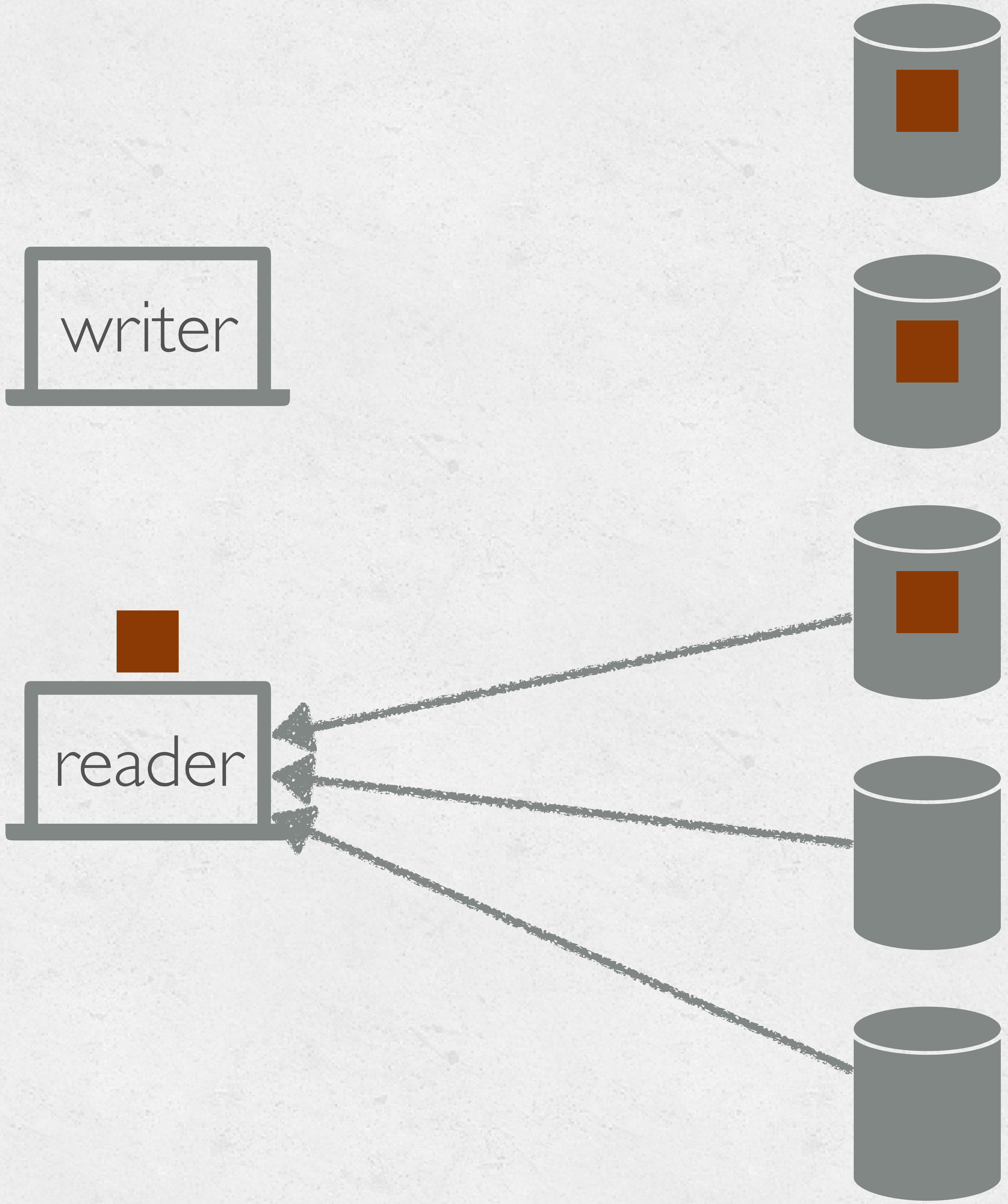
Does this work?



First Step: Single Reader, Single Writer (SWSR)

- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

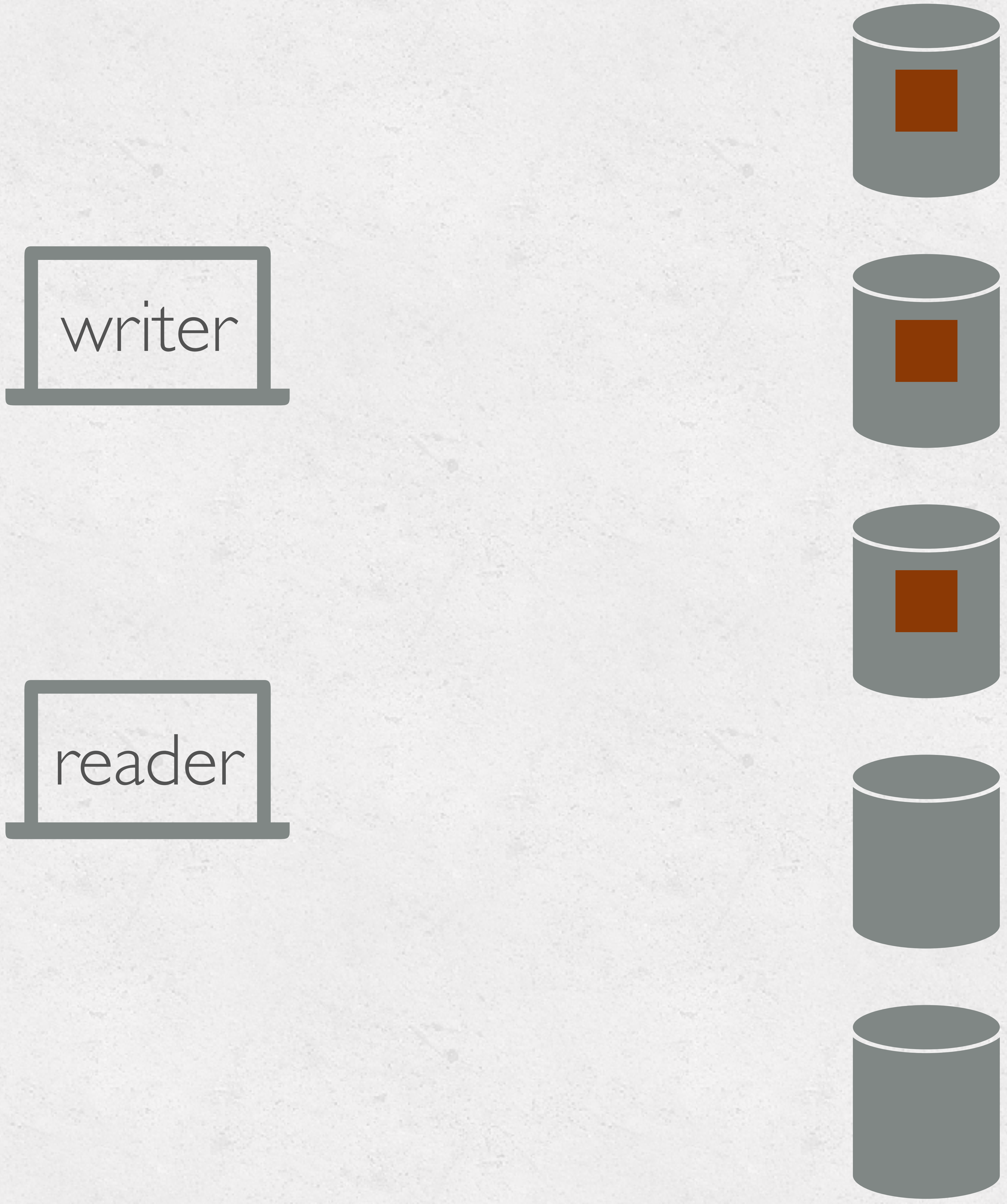
Does this work?



First Step: Single Reader, Single Writer (SWSR)

- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

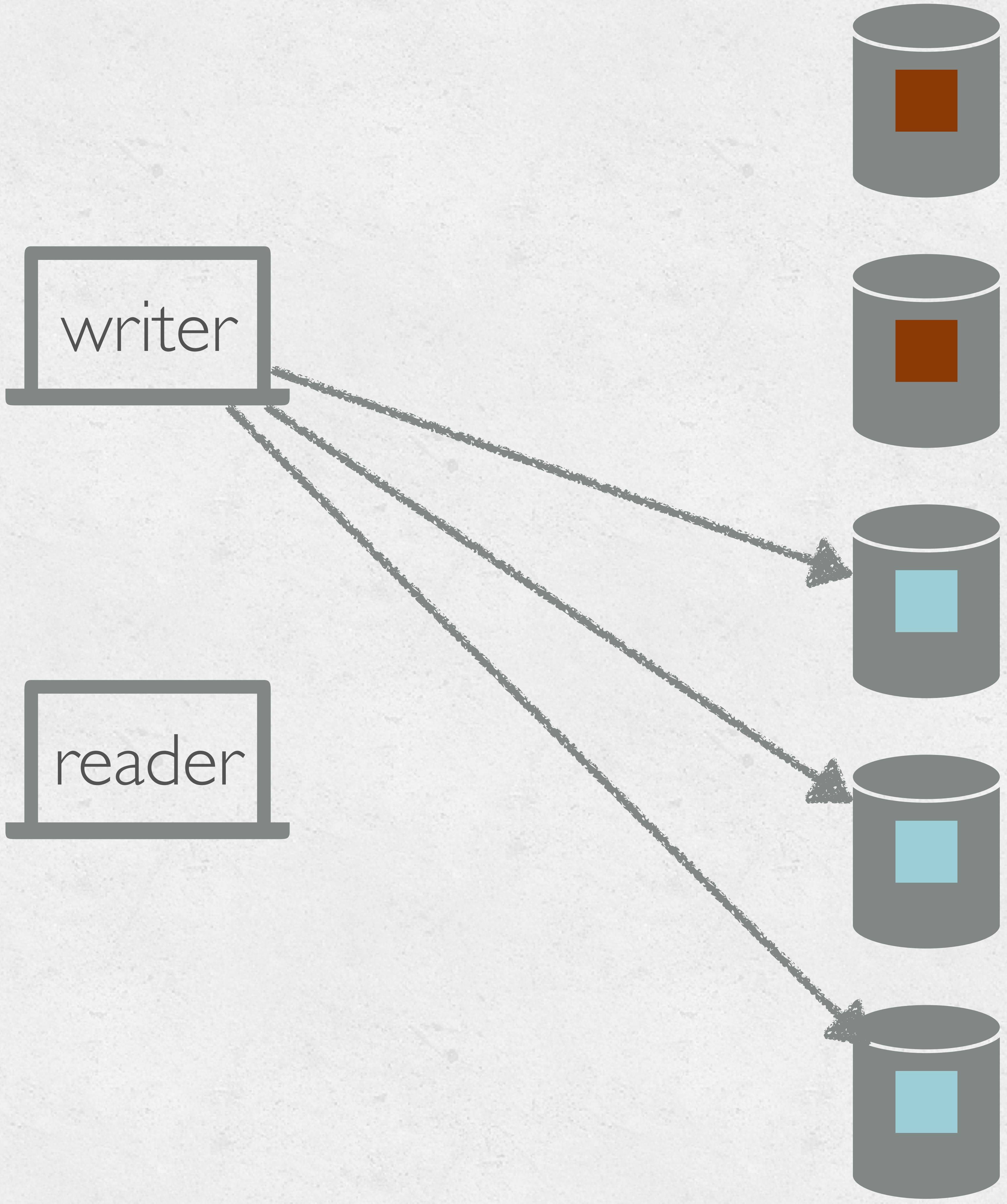
Does this work?



First Step: Single Reader, Single Writer (SWSR)

- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

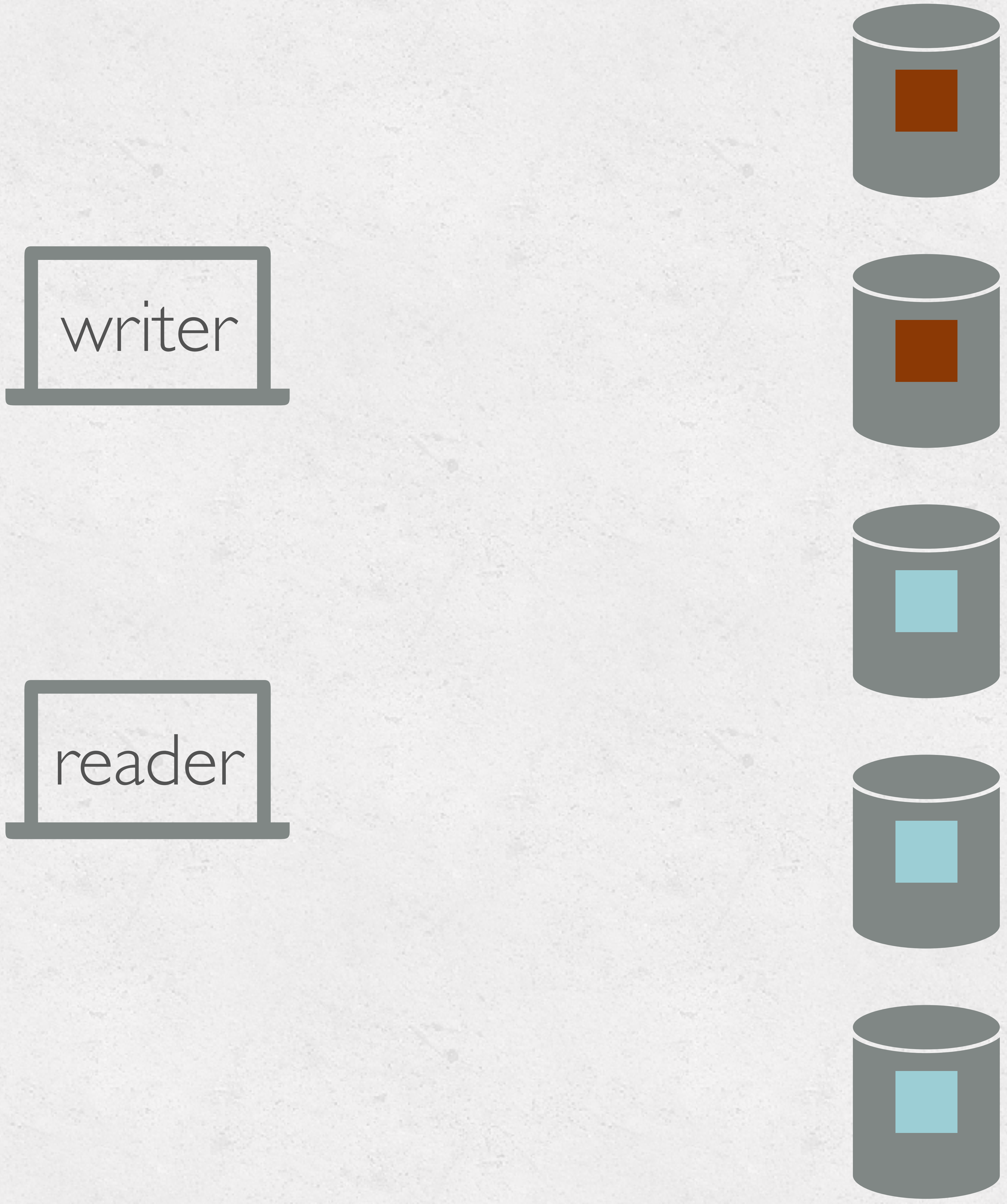
Does this work?



First Step: Single Reader, Single Writer (SWSR)

- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

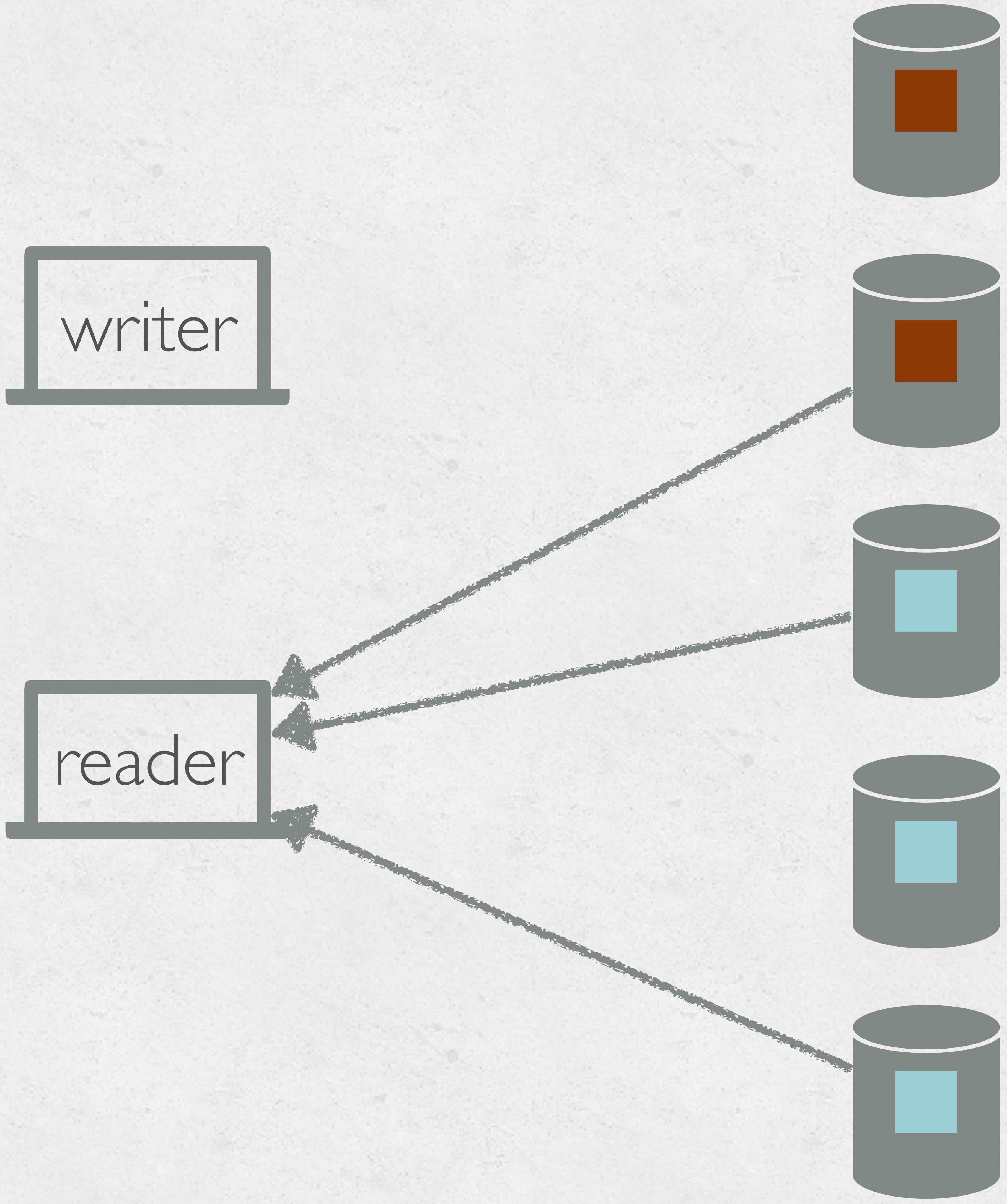
Does this work?



First Step: Single Reader, Single Writer (SWSR)

- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

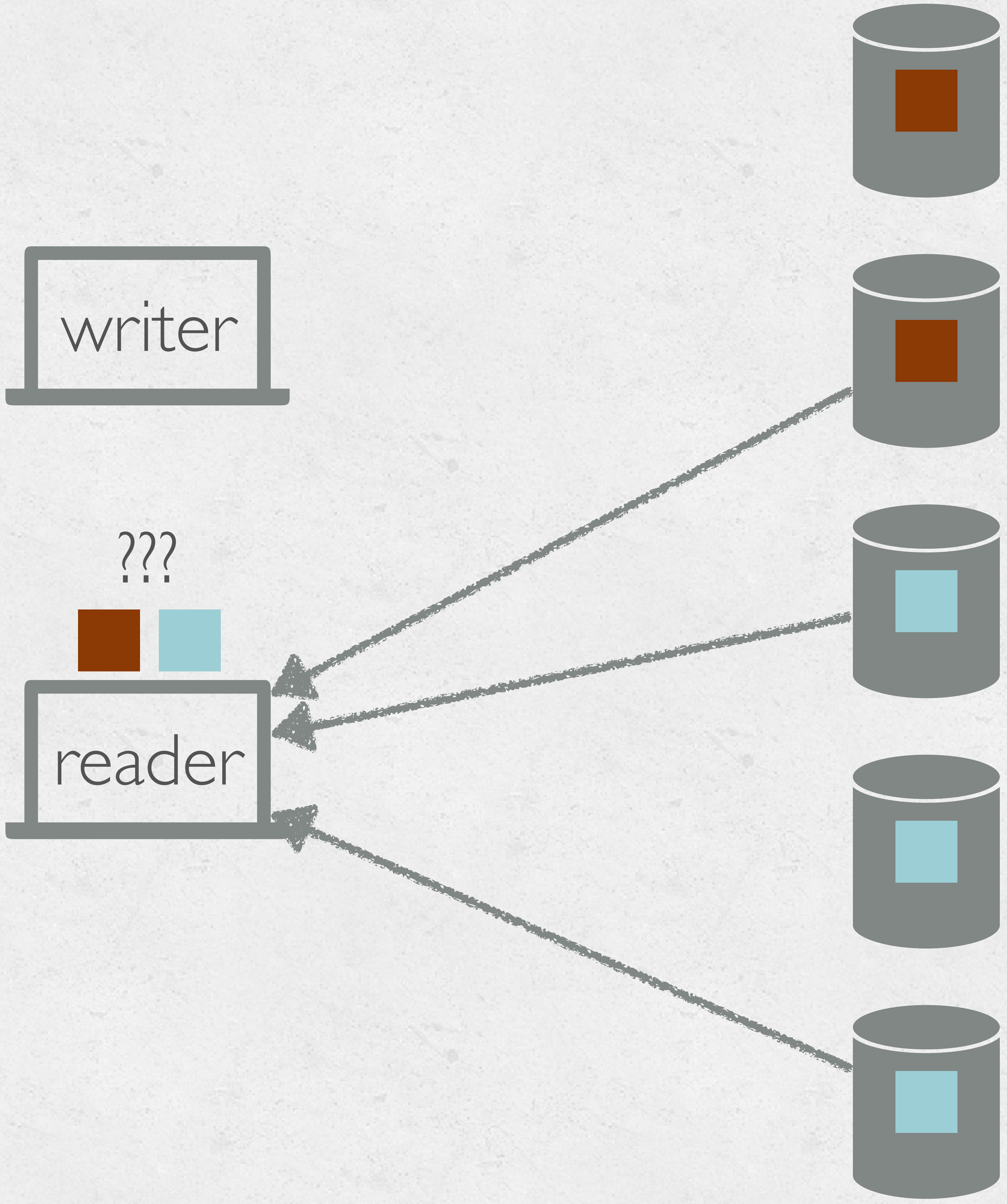
Does this work?



First Step: Single Reader, Single Writer (SWSR)

- Writer sends value to a majority.
- Reader reads value from a majority.
- Since majorities intersect, reader reads writer's value.

Does this work?



SWSR II

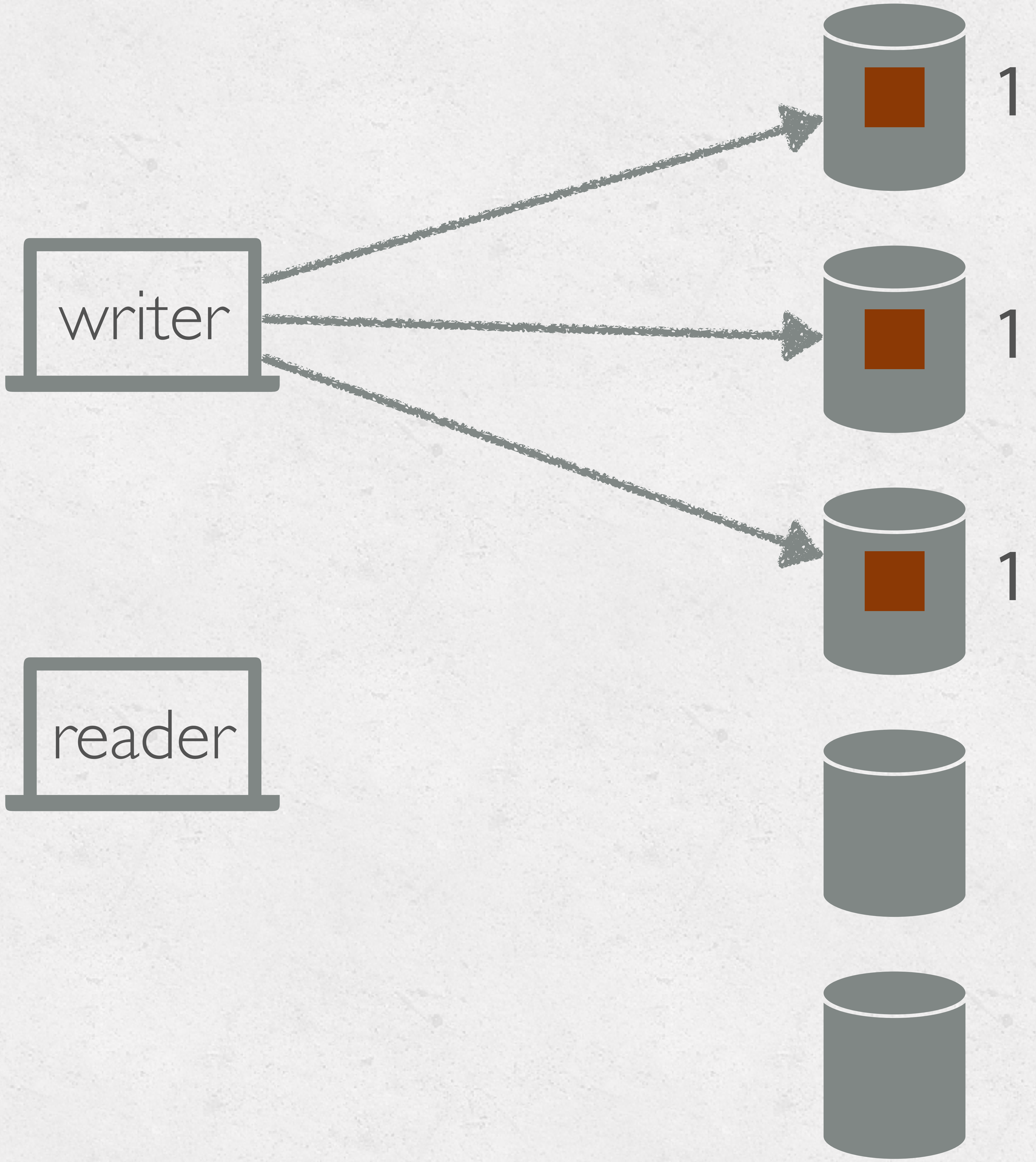
- Writer sends a **timestamped** value to a majority.
- Reader reads value from a majority, takes the one with the **highest timestamp**.
- Since majorities intersect, reader reads writer's value.

Does this work?

SWSR II

- Writer sends a **timestamped** value to a majority.
- Reader reads value from a majority, takes the one with the **highest timestamp**.
- Since majorities intersect, reader reads writer's value.

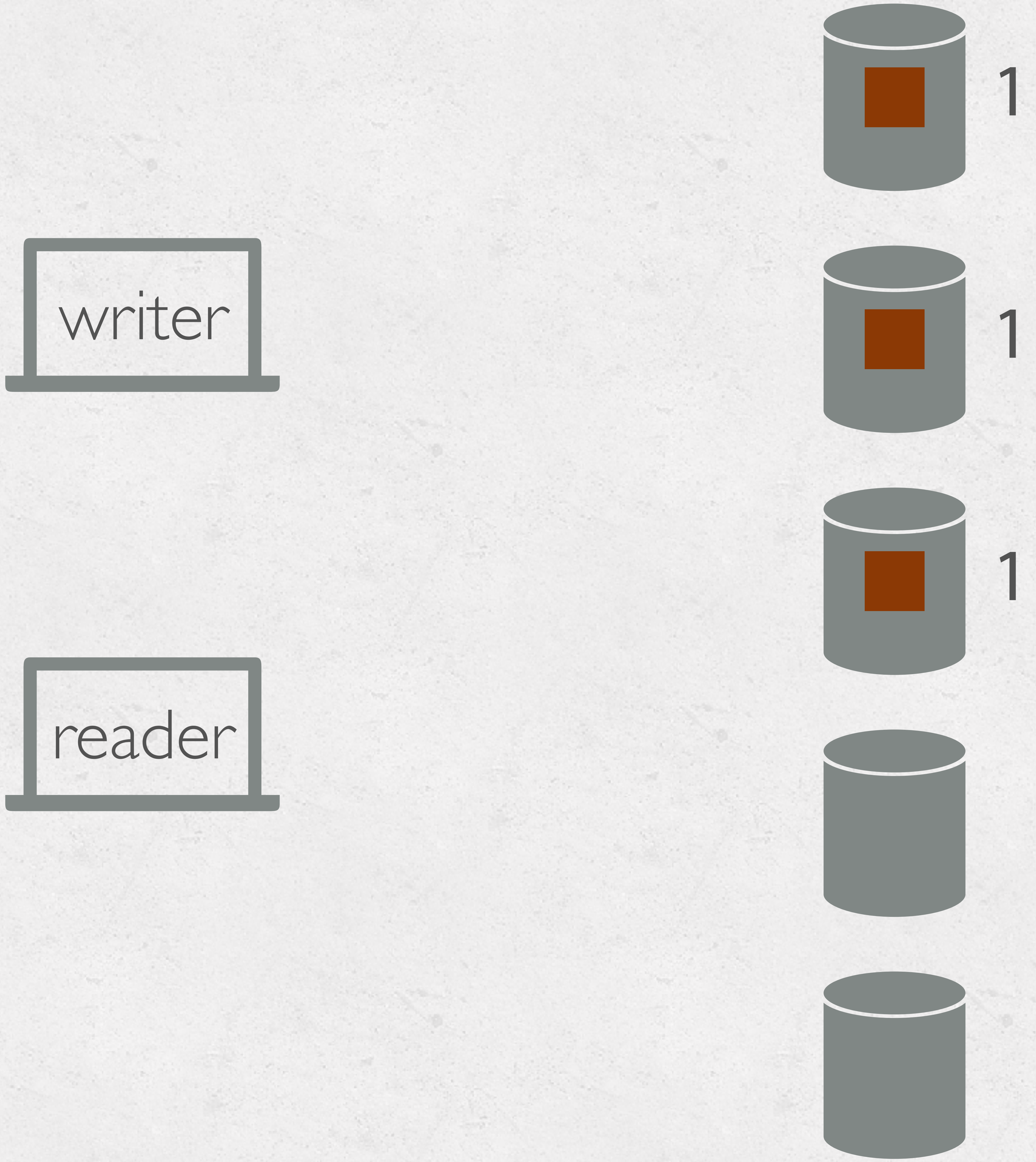
Does this work?



SWSR II

- Writer sends a **timestamped** value to a majority.
- Reader reads value from a majority, takes the one with the **highest timestamp**.
- Since majorities intersect, reader reads writer's value.

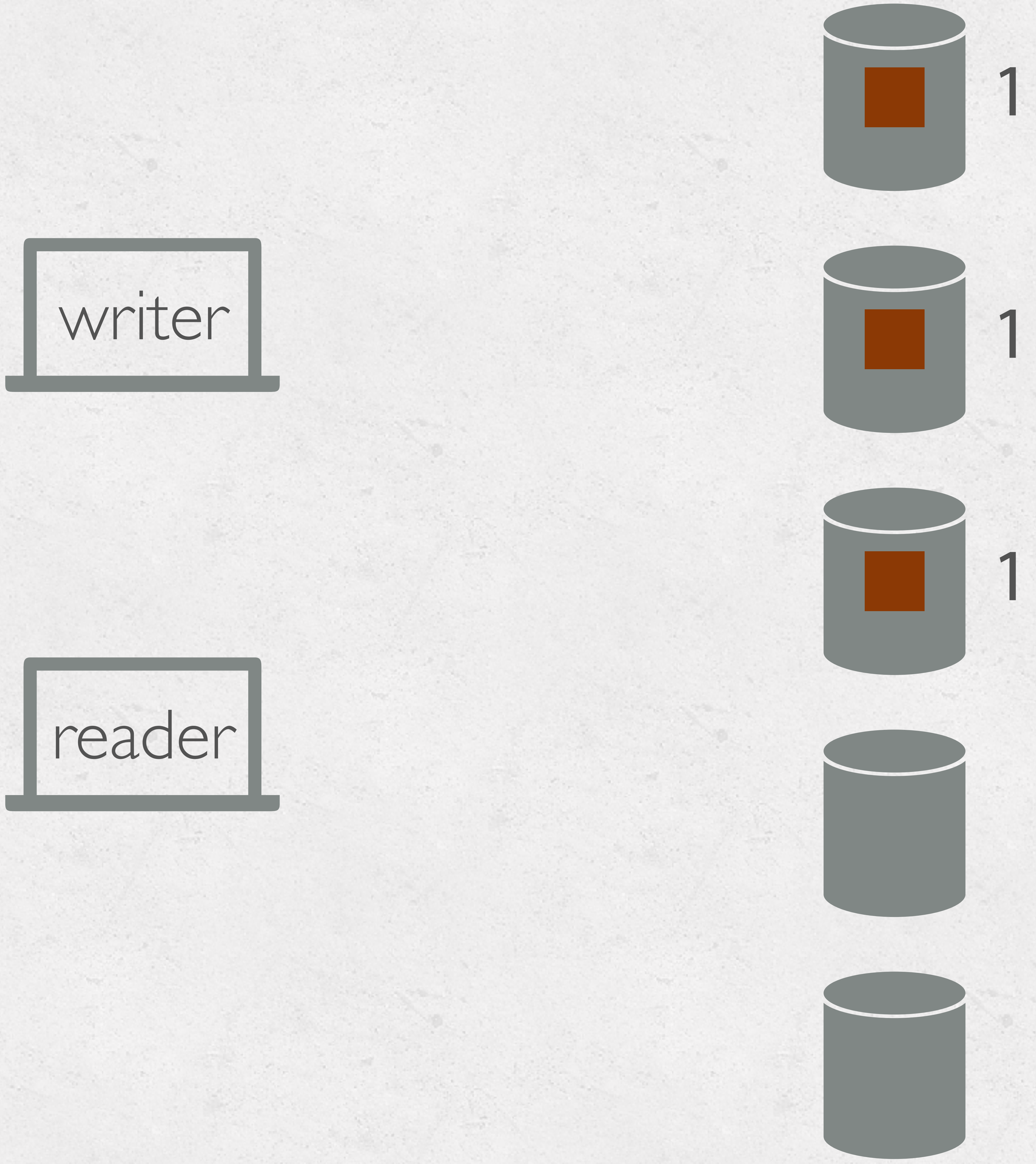
Does this work?



SWSR II

- Writer sends a **timestamped** value to a majority.
- Reader reads value from a majority, takes the one with the highest **highest timestamp**.
- Since majorities intersect, reader reads writer's value.

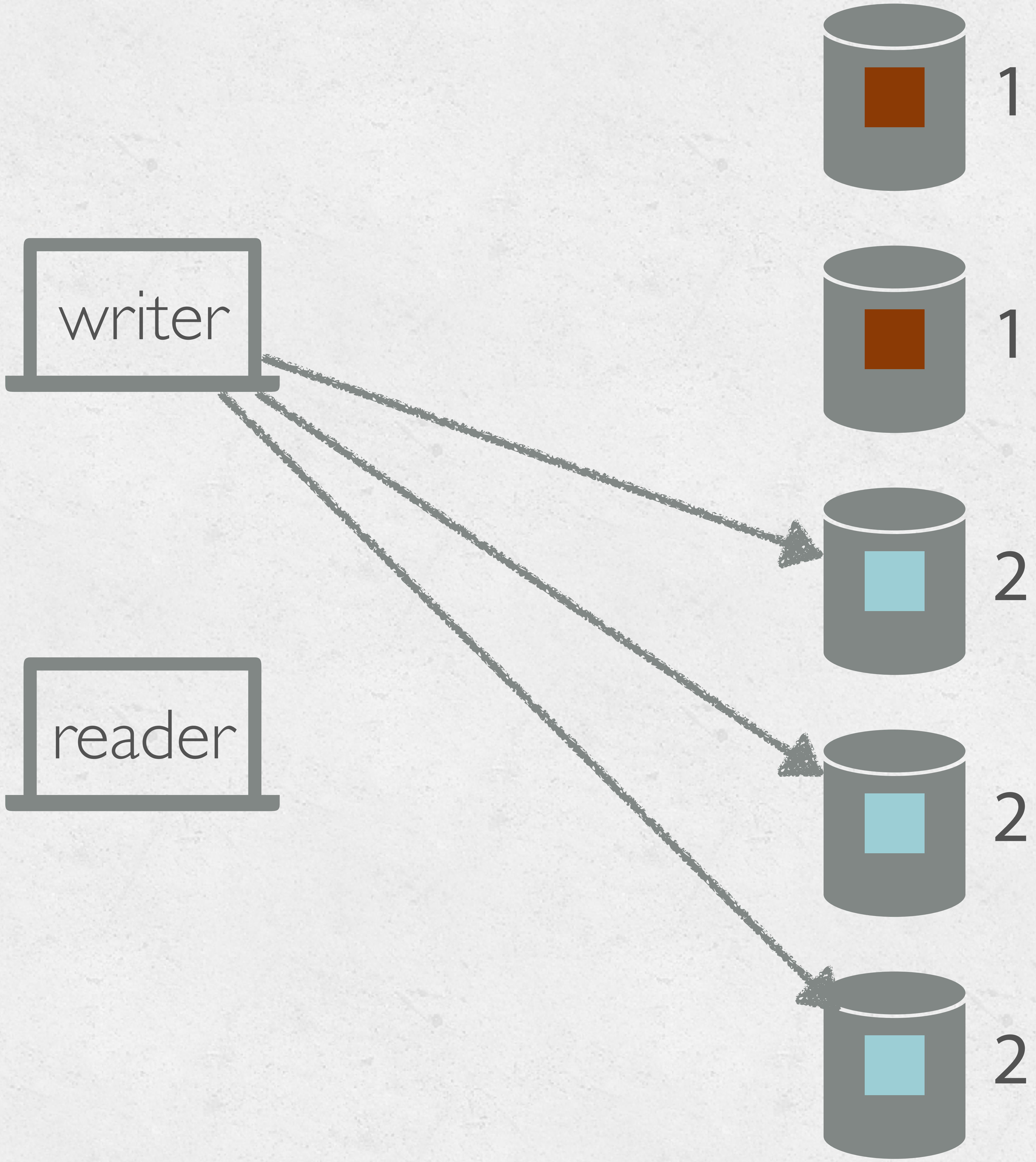
Does this work?



SWSR II

- Writer sends a **timestamped** value to a majority.
- Reader reads value from a majority, takes the one with the **highest timestamp**.
- Since majorities intersect, reader reads writer's value.

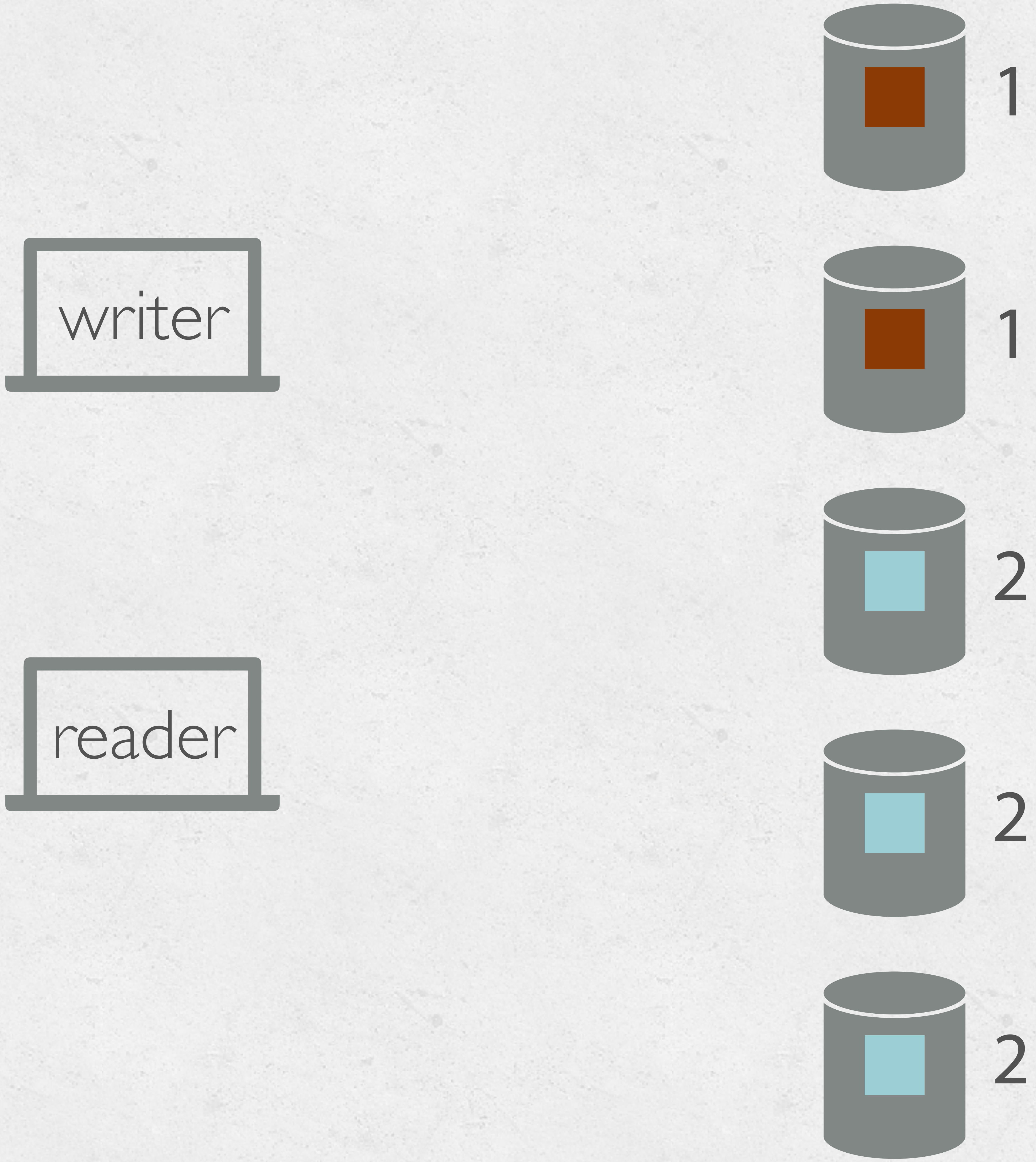
Does this work?



SWSR II

- Writer sends a **timestamped** value to a majority.
- Reader reads value from a majority, takes the one with the **highest timestamp**.
- Since majorities intersect, reader reads writer's value.

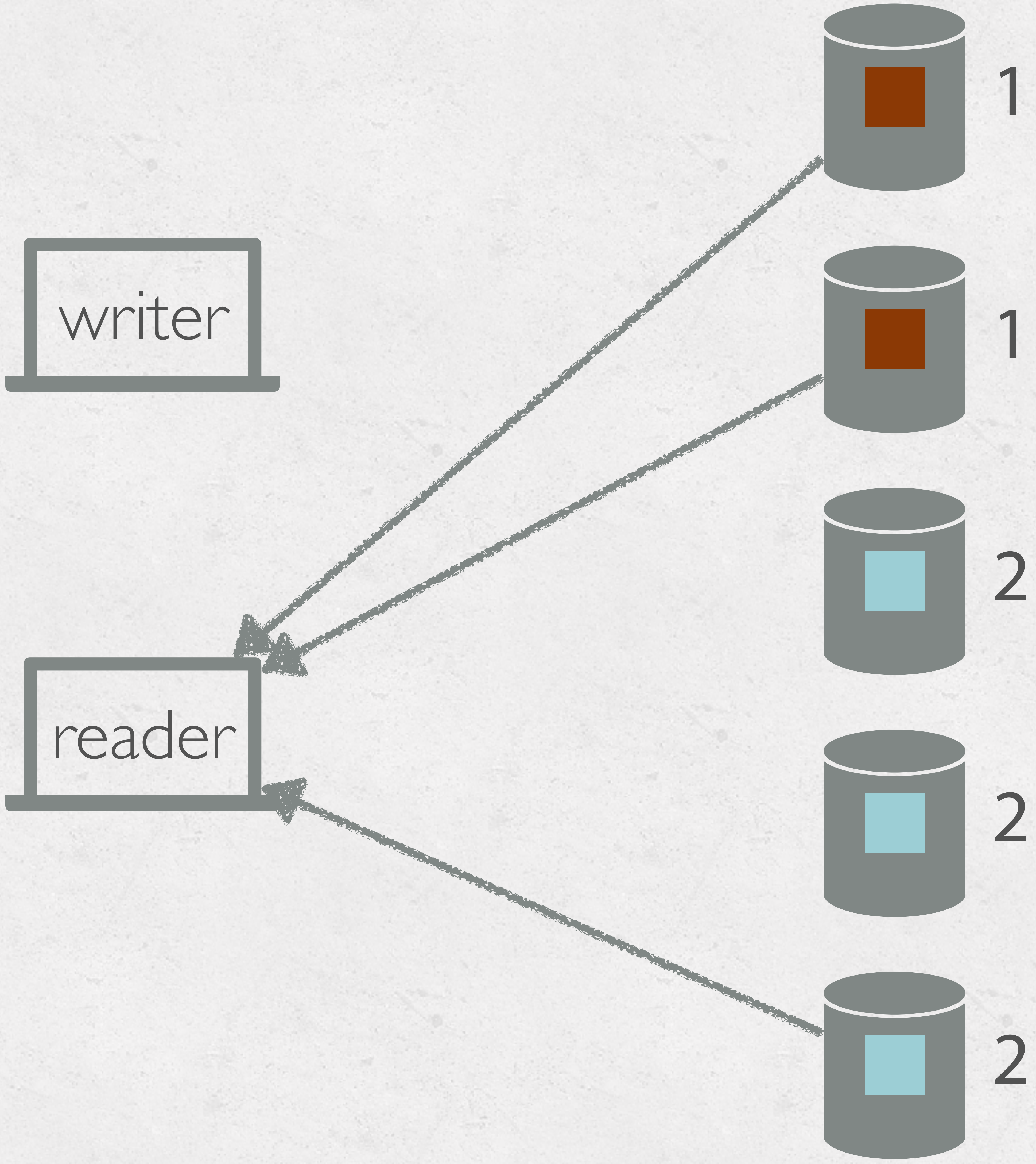
Does this work?



SWSR II

- Writer sends a **timestamped** value to a majority.
- Reader reads value from a majority, takes the one with the **highest timestamp**.
- Since majorities intersect, reader reads writer's value.

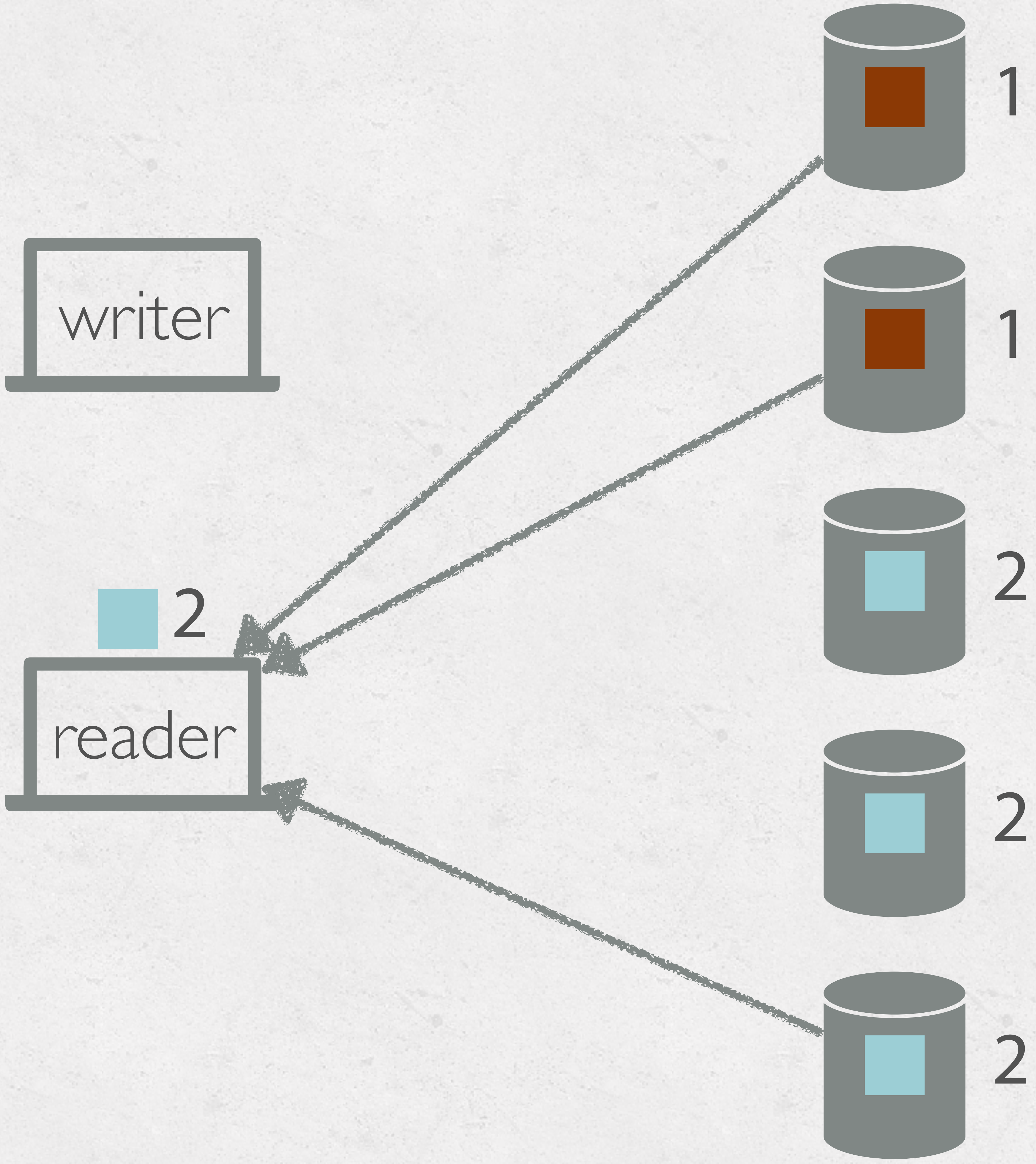
Does this work?



SWSR II

- Writer sends a **timestamped** value to a majority.
- Reader reads value from a majority, takes the one with the **highest timestamp**.
- Since majorities intersect, reader reads writer's value.

Does this work?



SWSR III

Servers' algorithm:

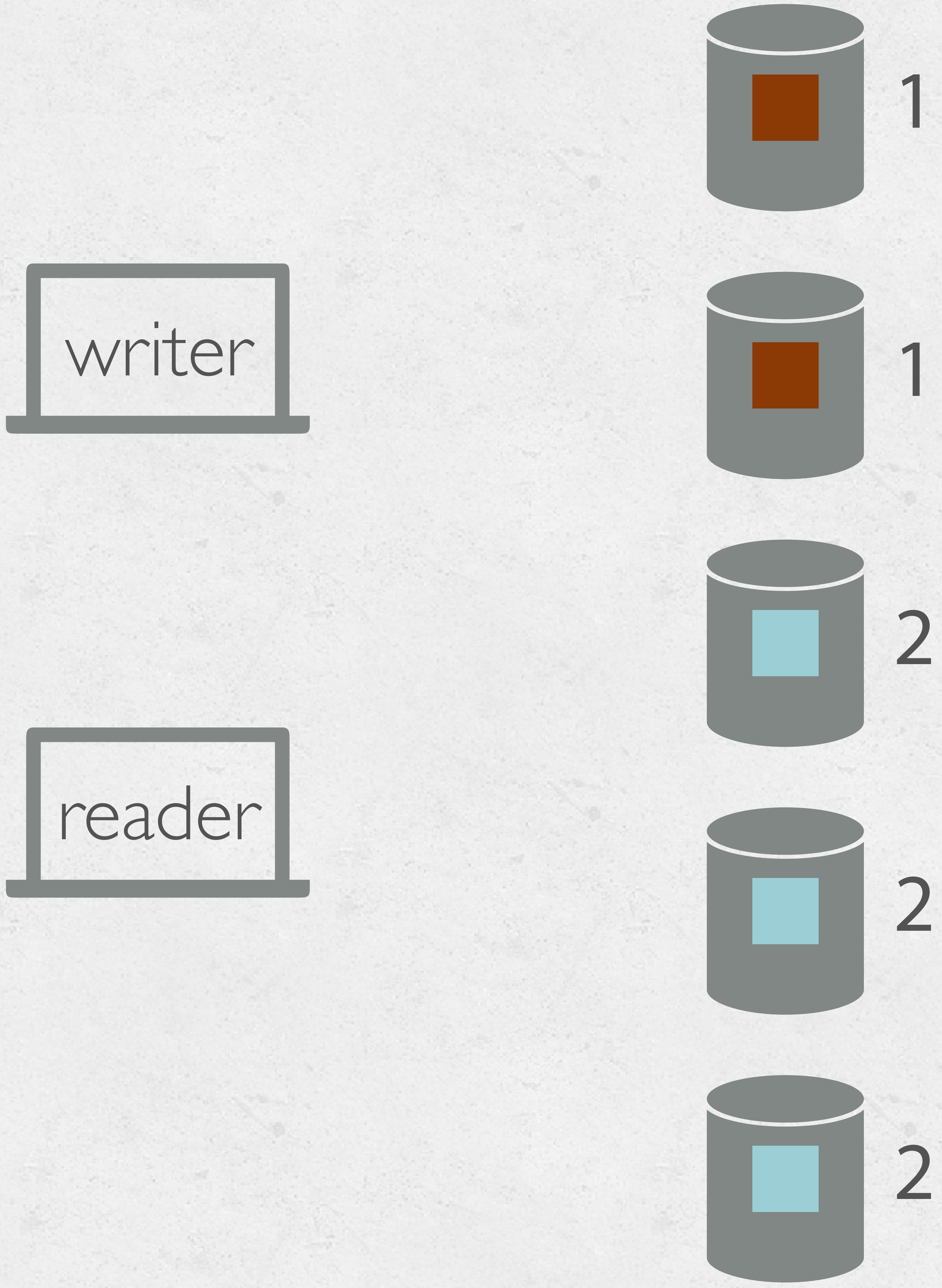
- Upon receiving a write, update local timestamp and value if write's timestamp is greater; send ack.
- Respond to reads with local timestamp and value.

Writer's algorithm:

- When writing, increment local timestamp, send timestamp and value to all.
- Wait for acks from a majority.

Reader's algorithm:

- Read from a majority, take value with highest timestamp.
- Maintain local value, return local value if servers' timestamps smaller.



SWSR III

Servers' algorithm:

- Upon value if write'
- Respo

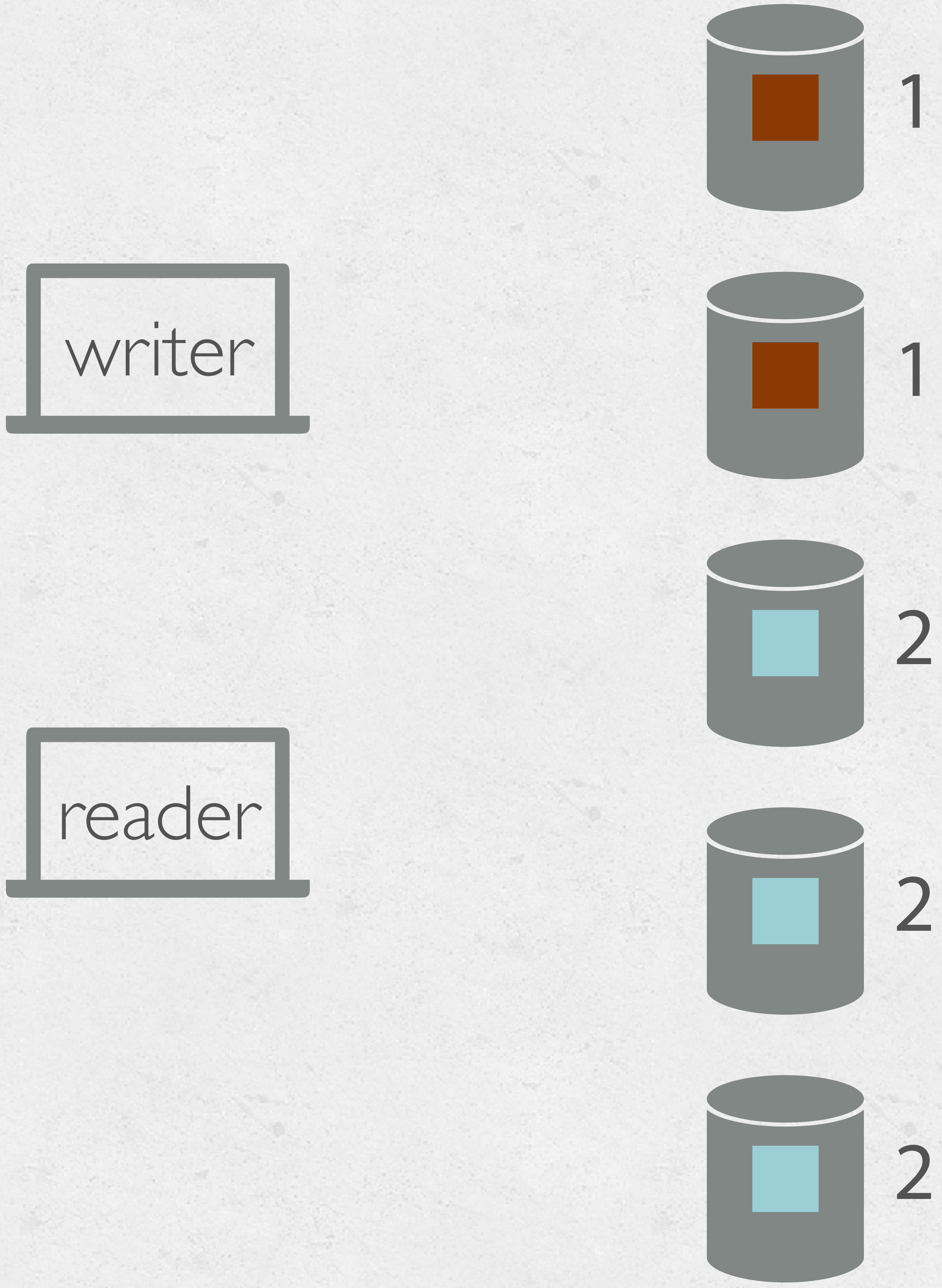
Writer's

- When writi increment local timestamp, send timestamp and value to all.
- Wait for acks from a majority.

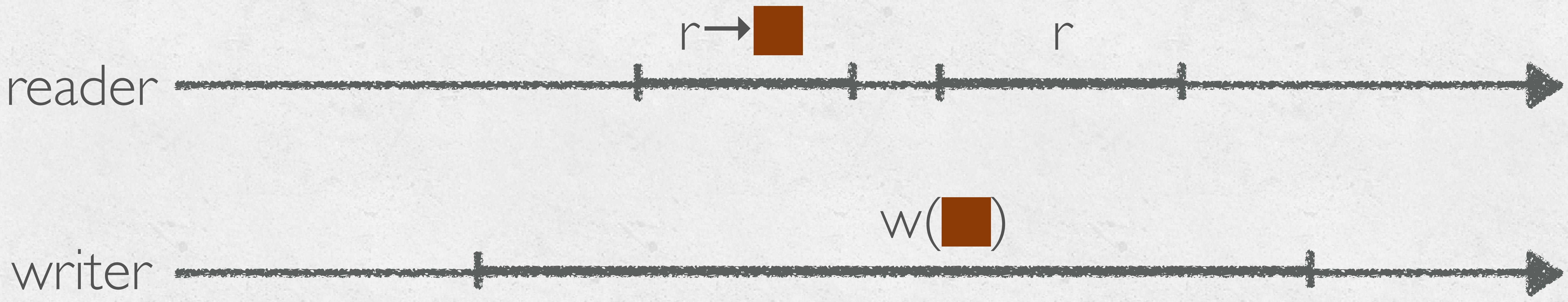
Reader's algorithm:

- Read from a majority, take value with highest timestamp.
- Maintain local value, return local value if servers' timestamps smaller.

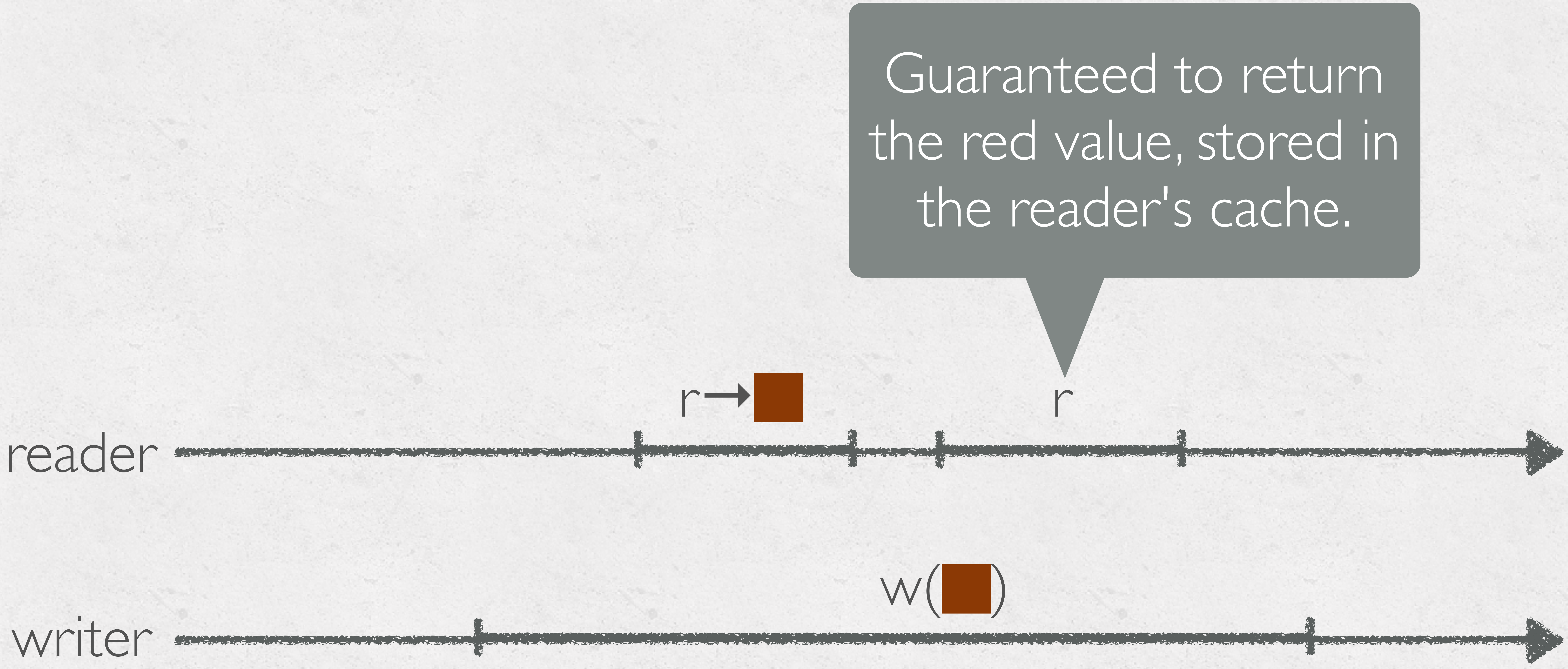
Assume clients can associate requests with responses (i.e., ignore responses from old requests)



SRSW: What About Multiple Reads?



SRSW: What About Multiple Reads?



If there's only one writer and one reader, why do we need the servers at all? Couldn't the writer just send its value to the reader directly?

Multiple Readers, Single Writer (MRSW)

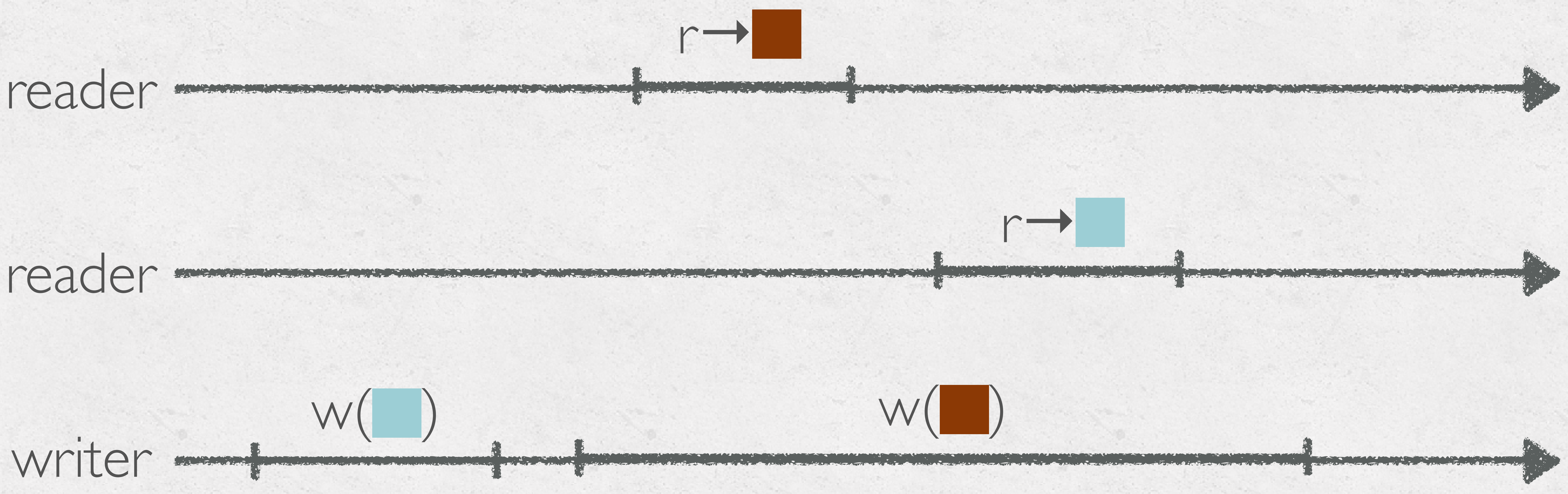
Does this previous solution just work?

Multiple Readers, Single Writer (MRSW)

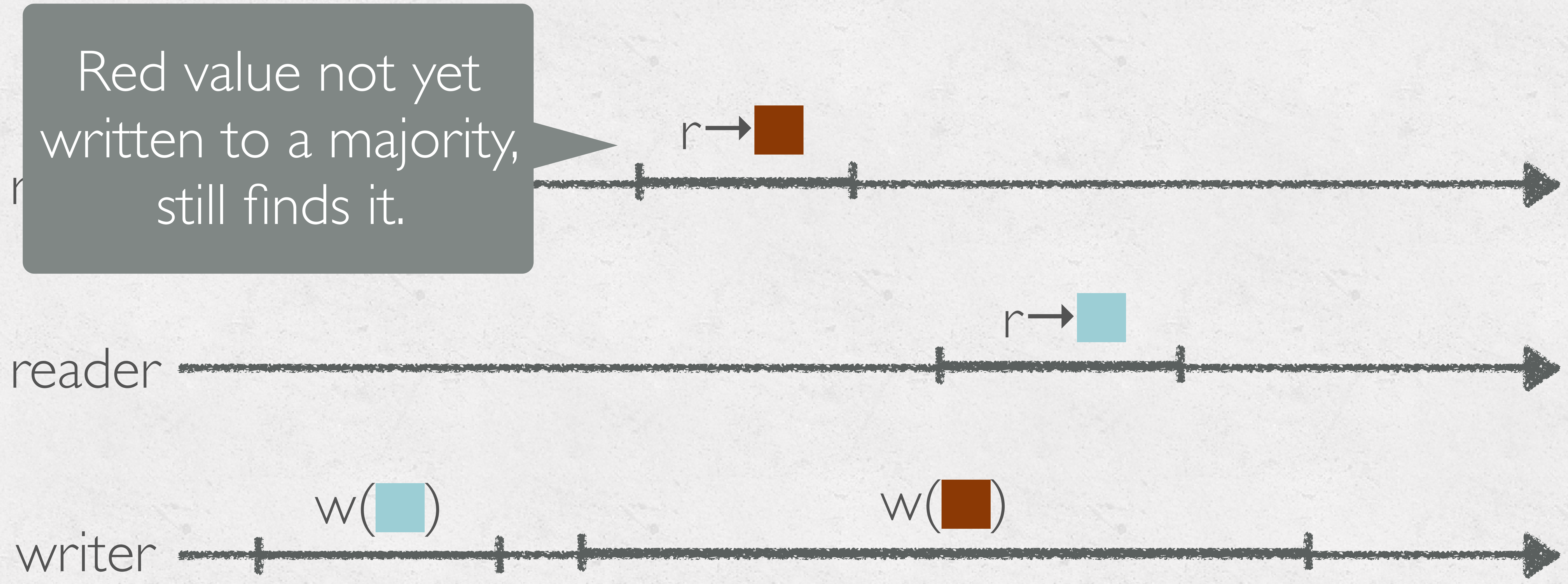
Does this previous solution just work?

What happens if there are multiple reads by **different processes** overlapping the same write?

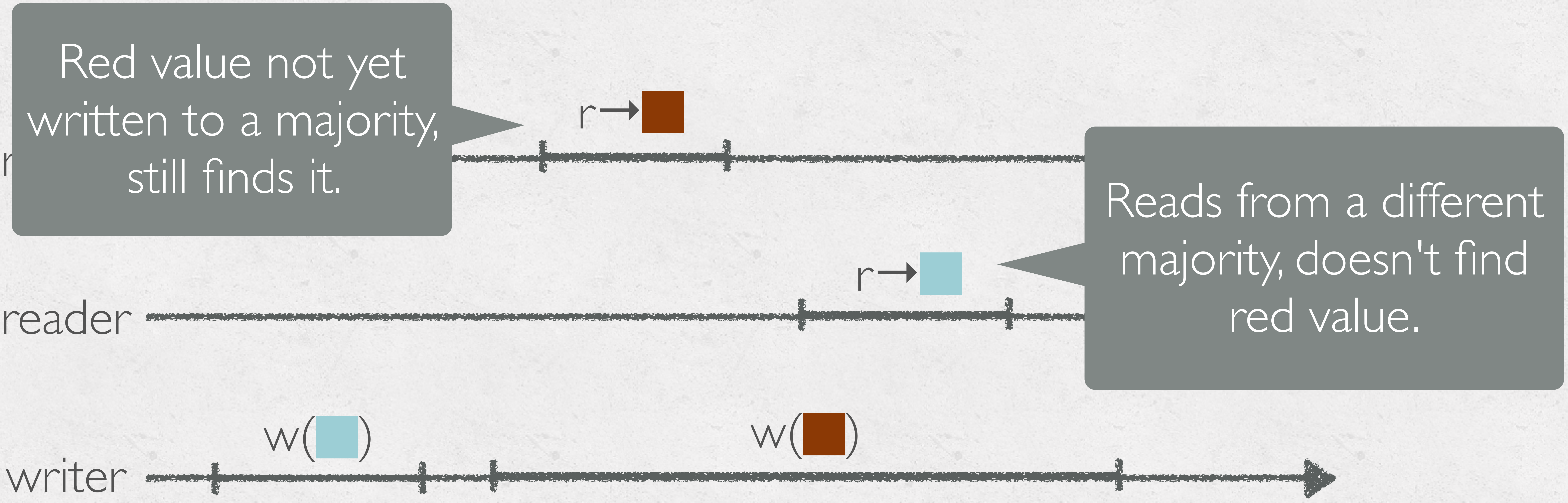
MRSW: Revenge of the Reads



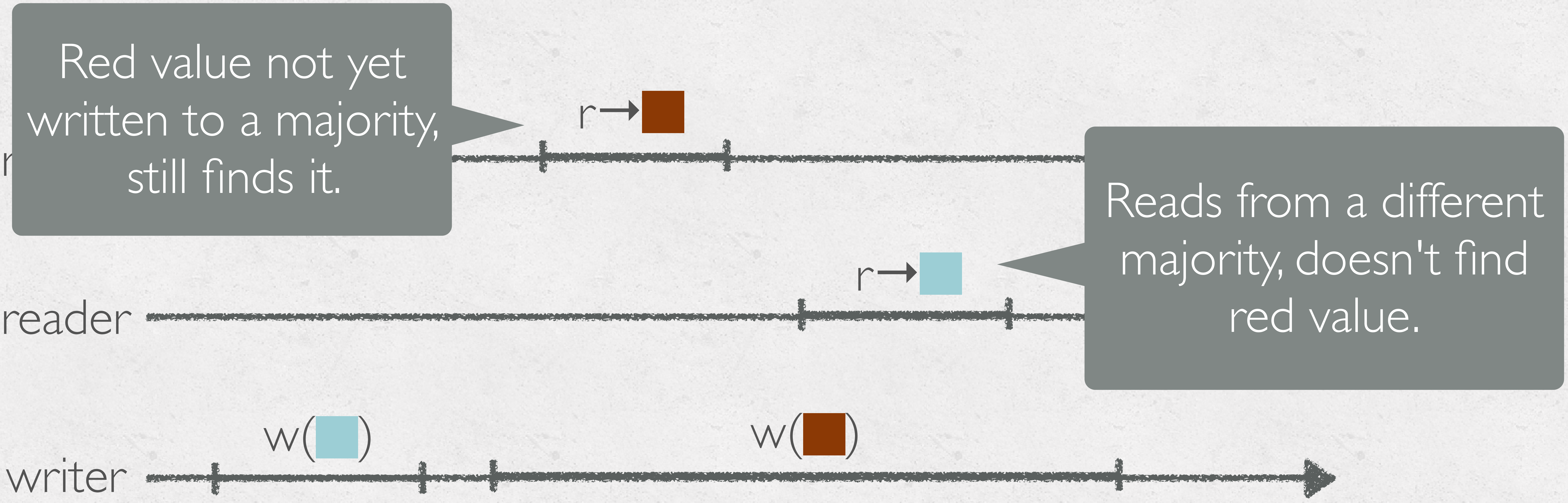
MRSW: Revenge of the Reads



MRSW: Revenge of the Reads



MRSW: Revenge of the Reads

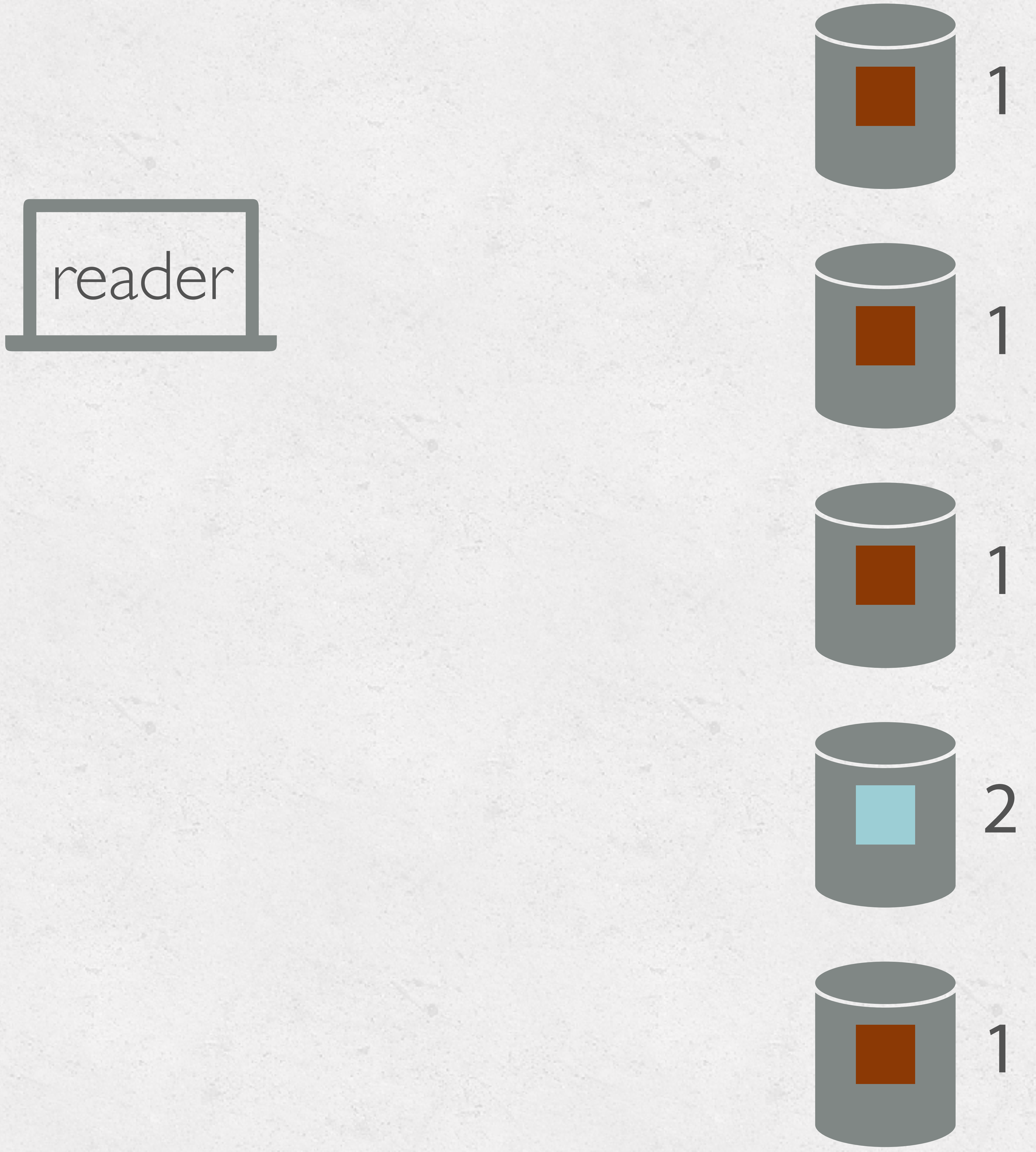


Not linearizable!

MRSW II

Suppose a write is ongoing (or the writer died).

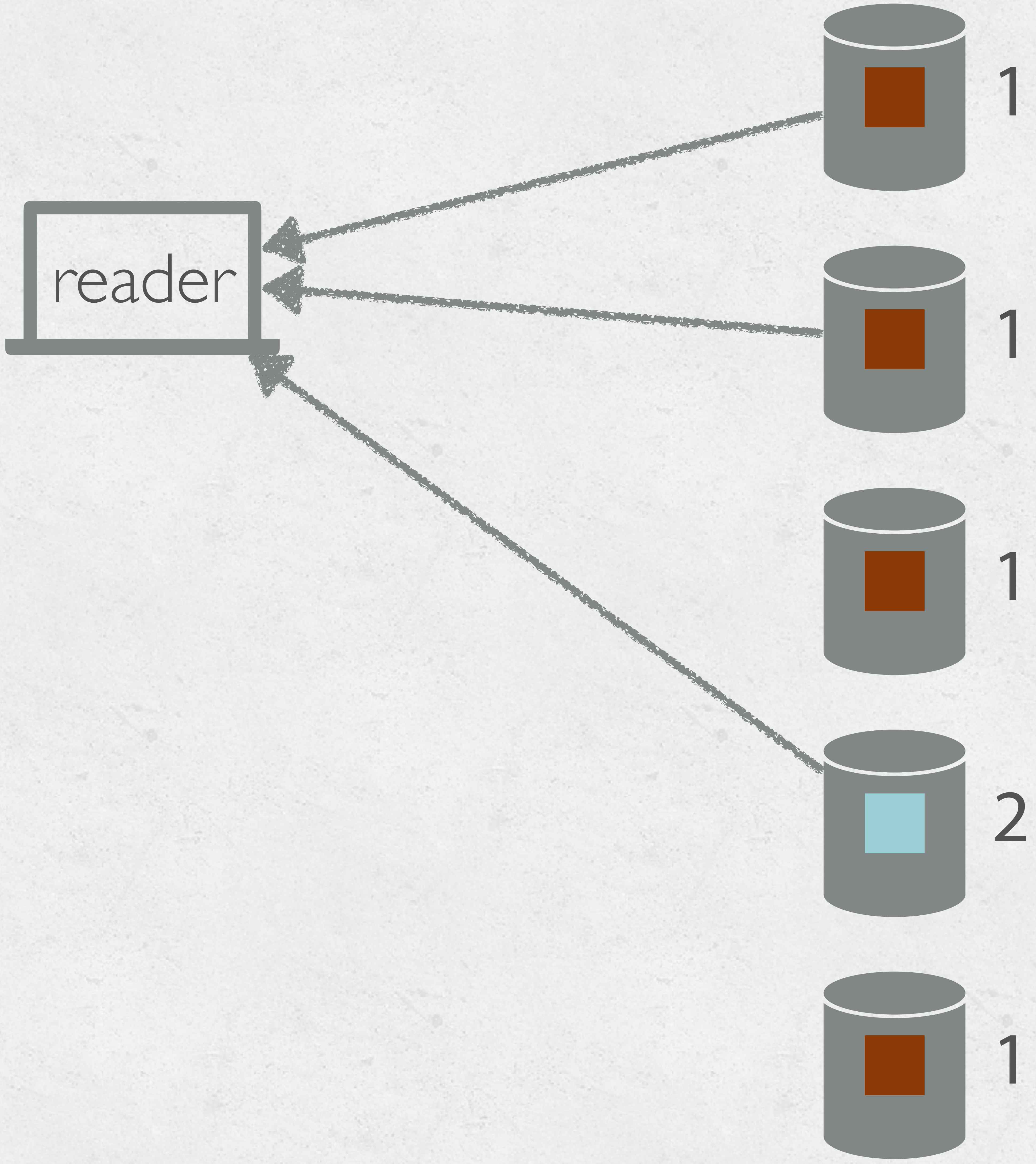
- Reader reads value from a majority, takes the one with the highest timestamp.
- Reader then performs a **write-back**, writing the value to a majority (not necessarily the same one). Only returns from read after write-back is complete.
- Later readers are guaranteed to read a value **at least as new** as the previously returned one.



MRSW II

Suppose a write is ongoing (or the writer died).

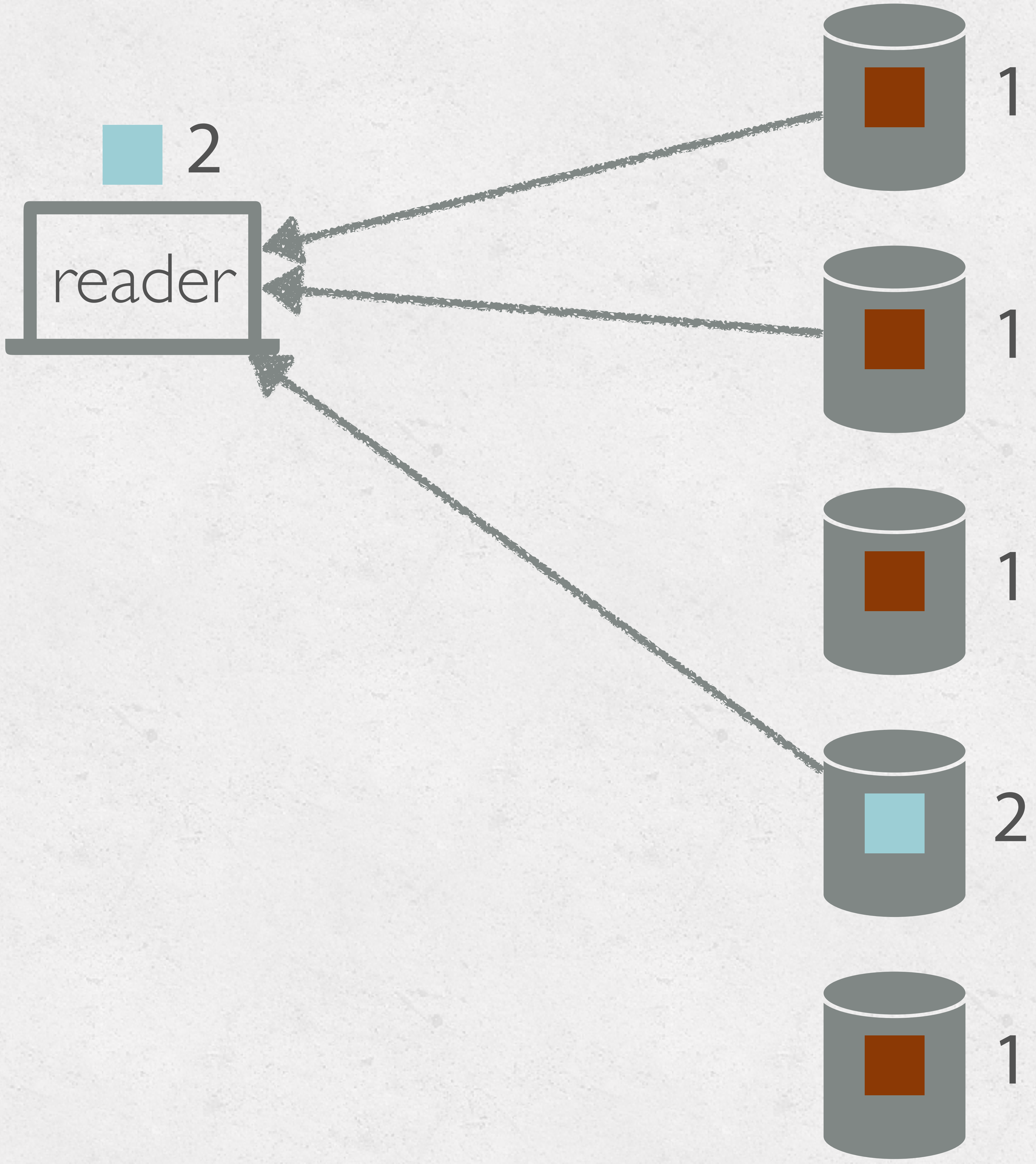
- Reader reads value from a majority, takes the one with the highest timestamp.
- Reader then performs a **write-back**, writing the value to a majority (not necessarily the same one). Only returns from read after write-back is complete.
- Later readers are guaranteed to read a value **at least as new** as the previously returned one.



MRSW II

Suppose a write is ongoing (or the writer died).

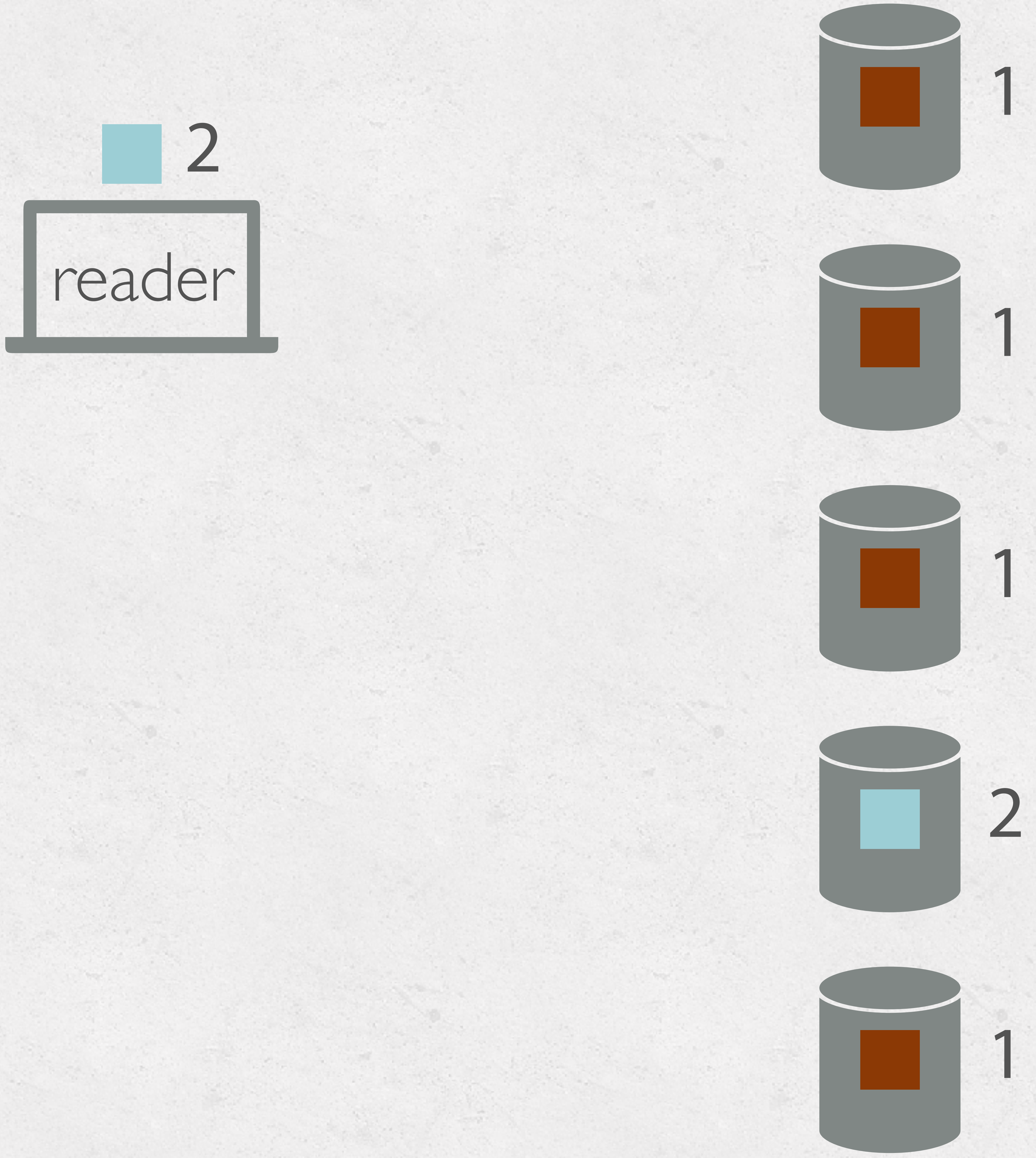
- Reader reads value from a majority, takes the one with the highest timestamp.
- Reader then performs a **write-back**, writing the value to a majority (not necessarily the same one). Only returns from read after write-back is complete.
- Later readers are guaranteed to read a value **at least as new** as the previously returned one.



MRSW II

Suppose a write is ongoing (or the writer died).

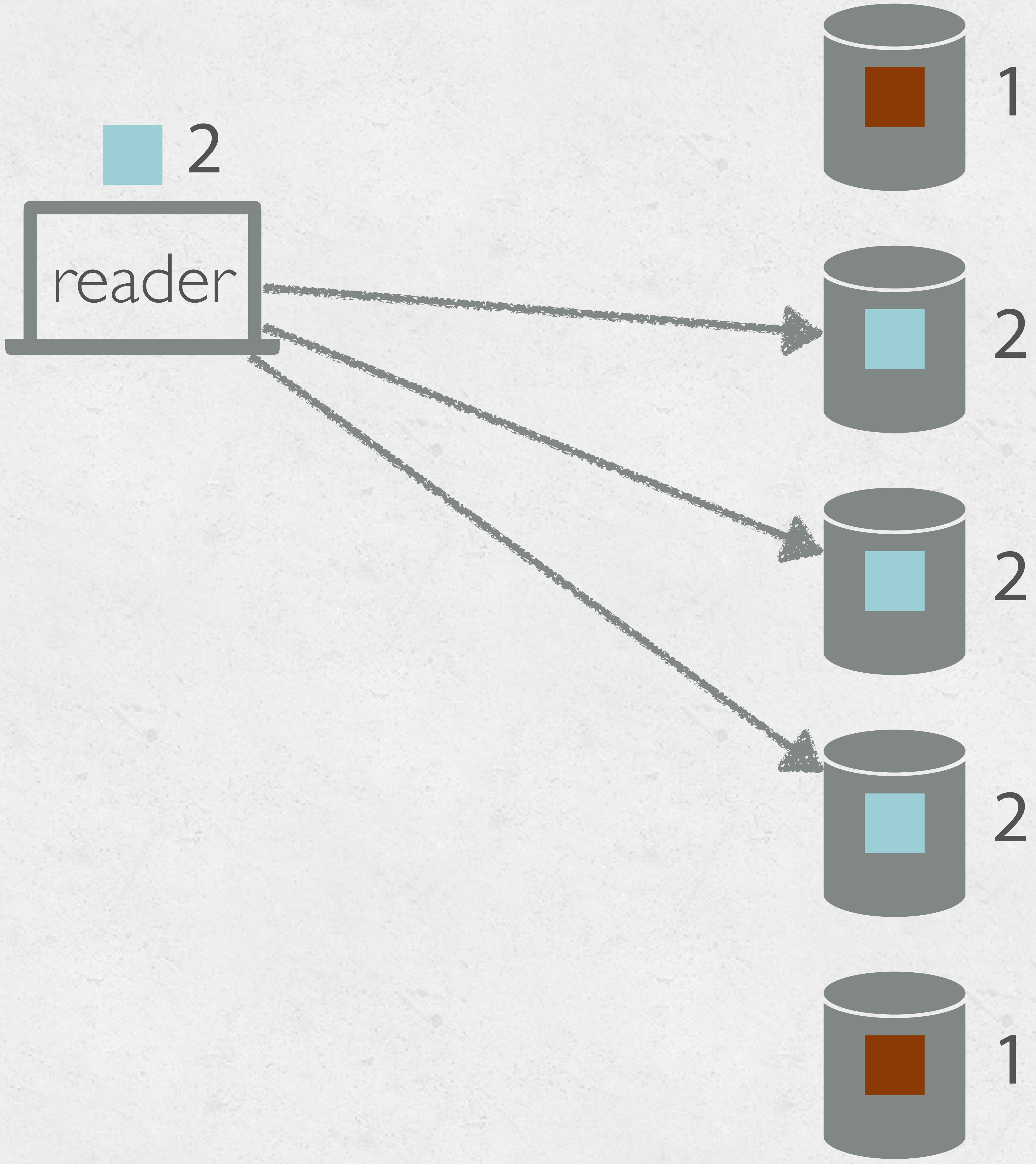
- Reader reads value from a majority, takes the one with the highest timestamp.
- Reader then performs a **write-back**, writing the value to a majority (not necessarily the same one). Only returns from read after write-back is complete.
- Later readers are guaranteed to read a value **at least as new** as the previously returned one.



MRSW II

Suppose a write is ongoing (or the writer died).

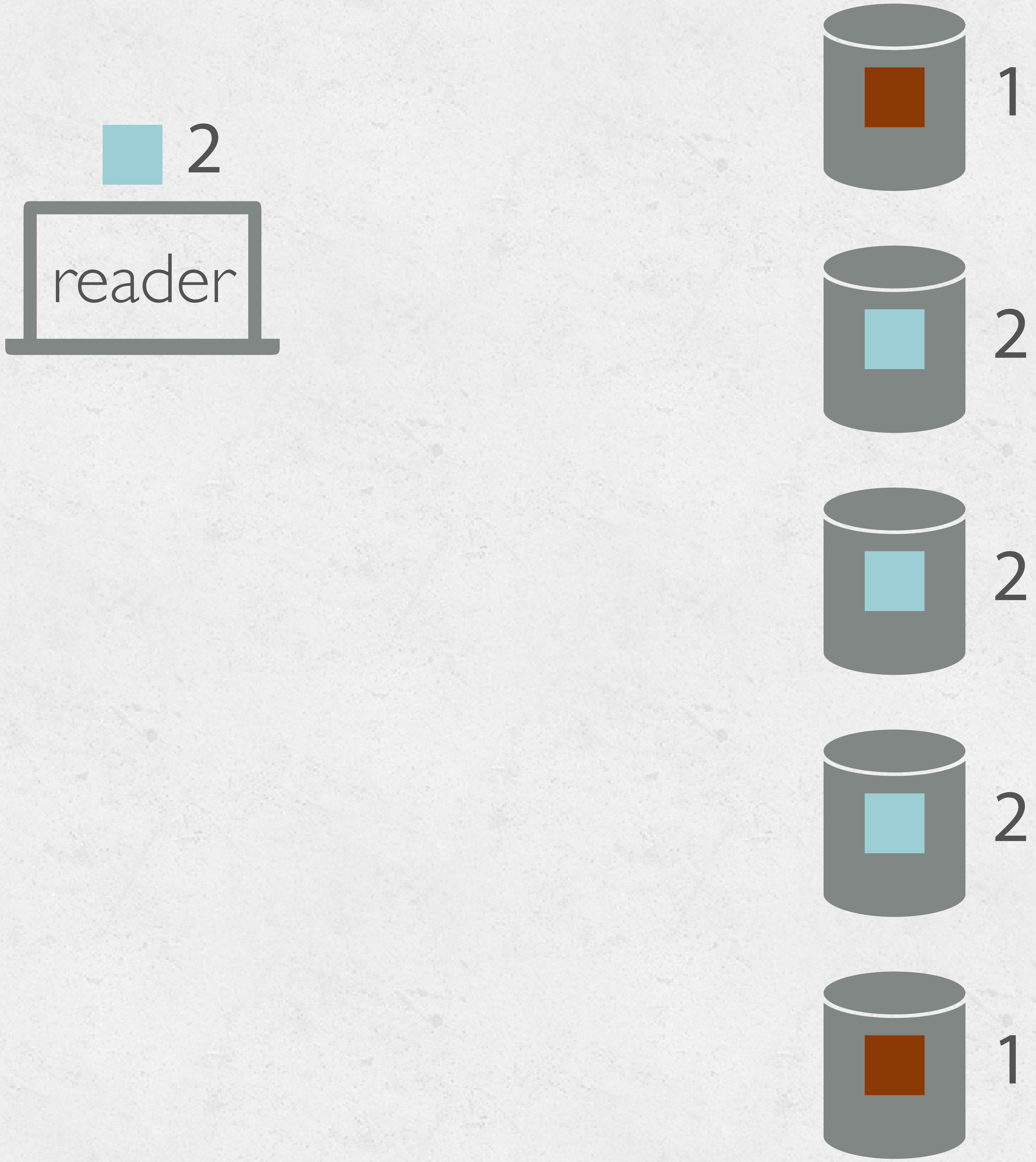
- Reader reads value from a majority, takes the one with the highest timestamp.
- Reader then performs a **write-back**, writing the value to a majority (not necessarily the same one). Only returns from read after write-back is complete.
- Later readers are guaranteed to read a value **at least as new** as the previously returned one.



MRSW II

Suppose a write is ongoing (or the writer died).

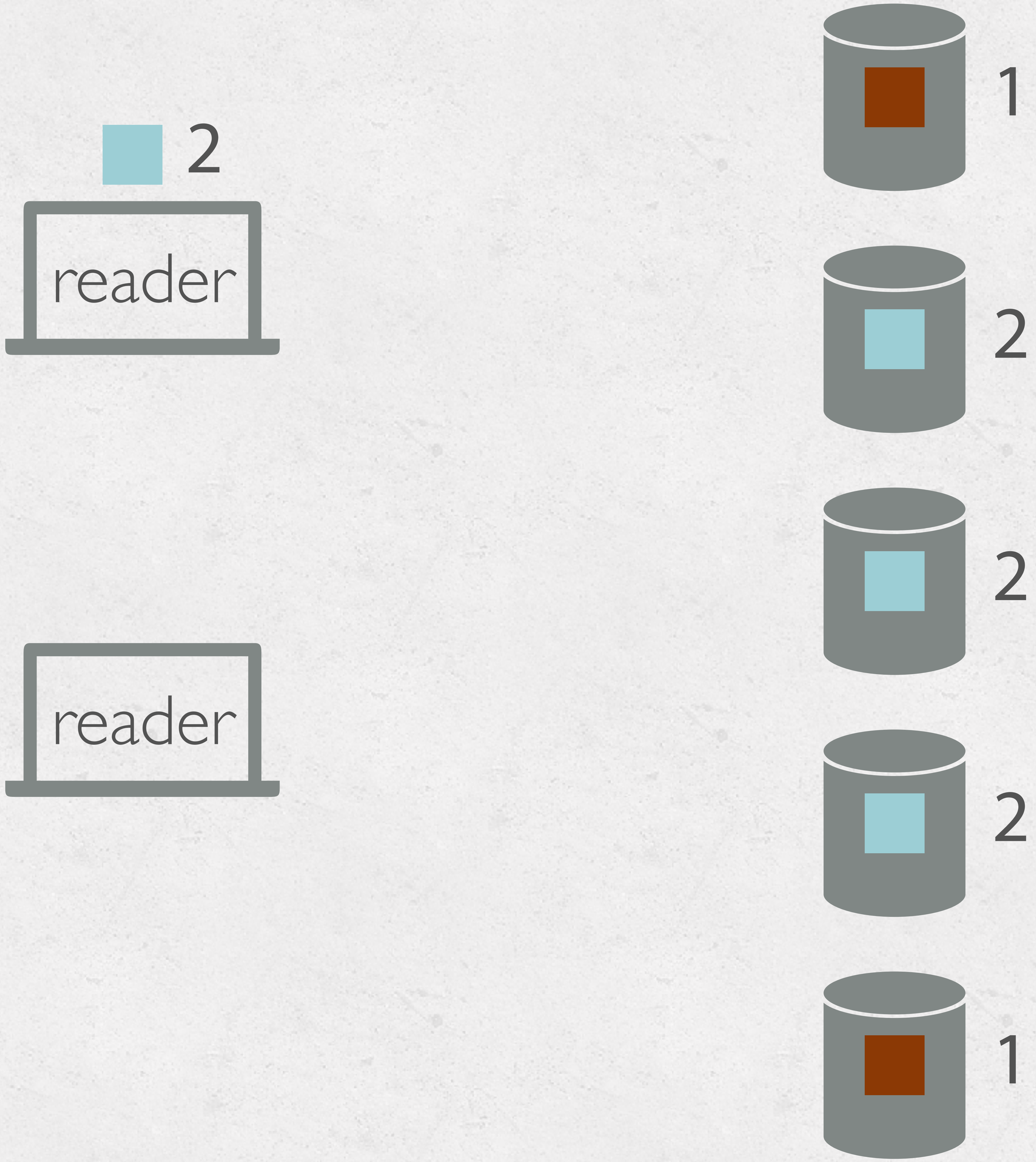
- Reader reads value from a majority, takes the one with the highest timestamp.
- Reader then performs a **write-back**, writing the value to a majority (not necessarily the same one). Only returns from read after write-back is complete.
- Later readers are guaranteed to read a value **at least as new** as the previously returned one.



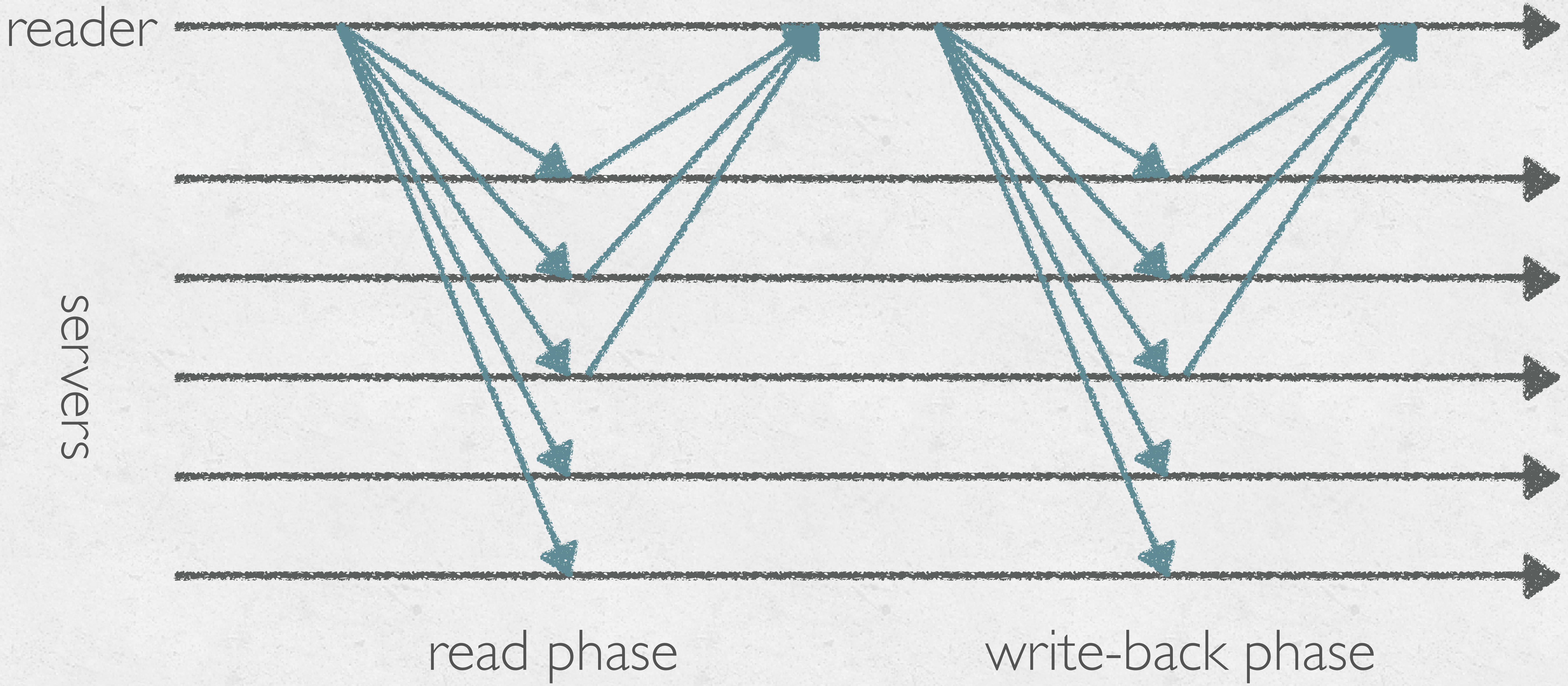
MRSW II

Suppose a write is ongoing (or the writer died).

- Reader reads value from a majority, takes the one with the highest timestamp.
- Reader then performs a **write-back**, writing the value to a majority (not necessarily the same one). Only returns from read after write-back is complete.
- Later readers are guaranteed to read a value **at least as new** as the previously returned one.



MRSW III



Do we always need to execute the write-back phase?

What if we only care about sequential consistency?
Do we care about the write-back phase at all?

Putting It All Together: MRMW

Does the previous solution just work?

Putting It All Together: MRMW

Does the previous solution just work?

What if writers use the **same timestamp**?

Putting It All Together: MRMW

Does the previous solution just work?

What if writers use the **same timestamp**?

Prevented by breaking ties using writers ID, same as PMMC.

Putting It All Together: MRMW

Does the previous solution just work?

What if writers use the **same timestamp**?

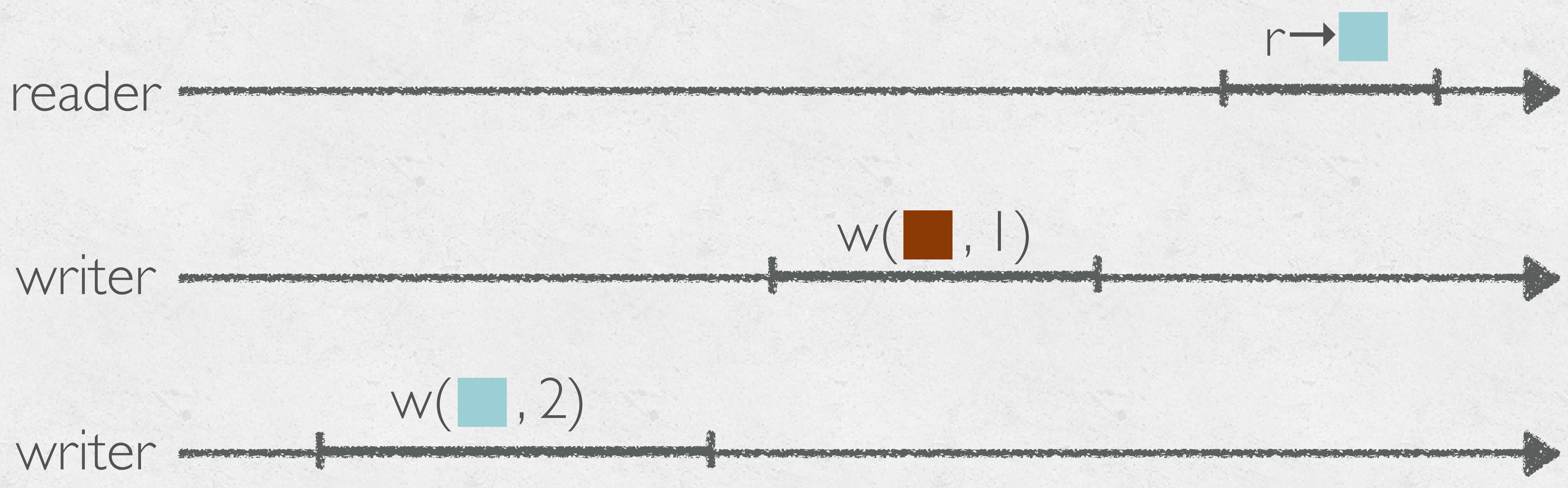
Prevented by breaking ties using writers ID, same as PMMC.

What if a write that starts after a previous write ended uses a **smaller timestamp**?

MRMW: Untimely Timestamps

MRMW: Untimely Timestamps

Reads from a majority, sees blue value has the highest timestamp.



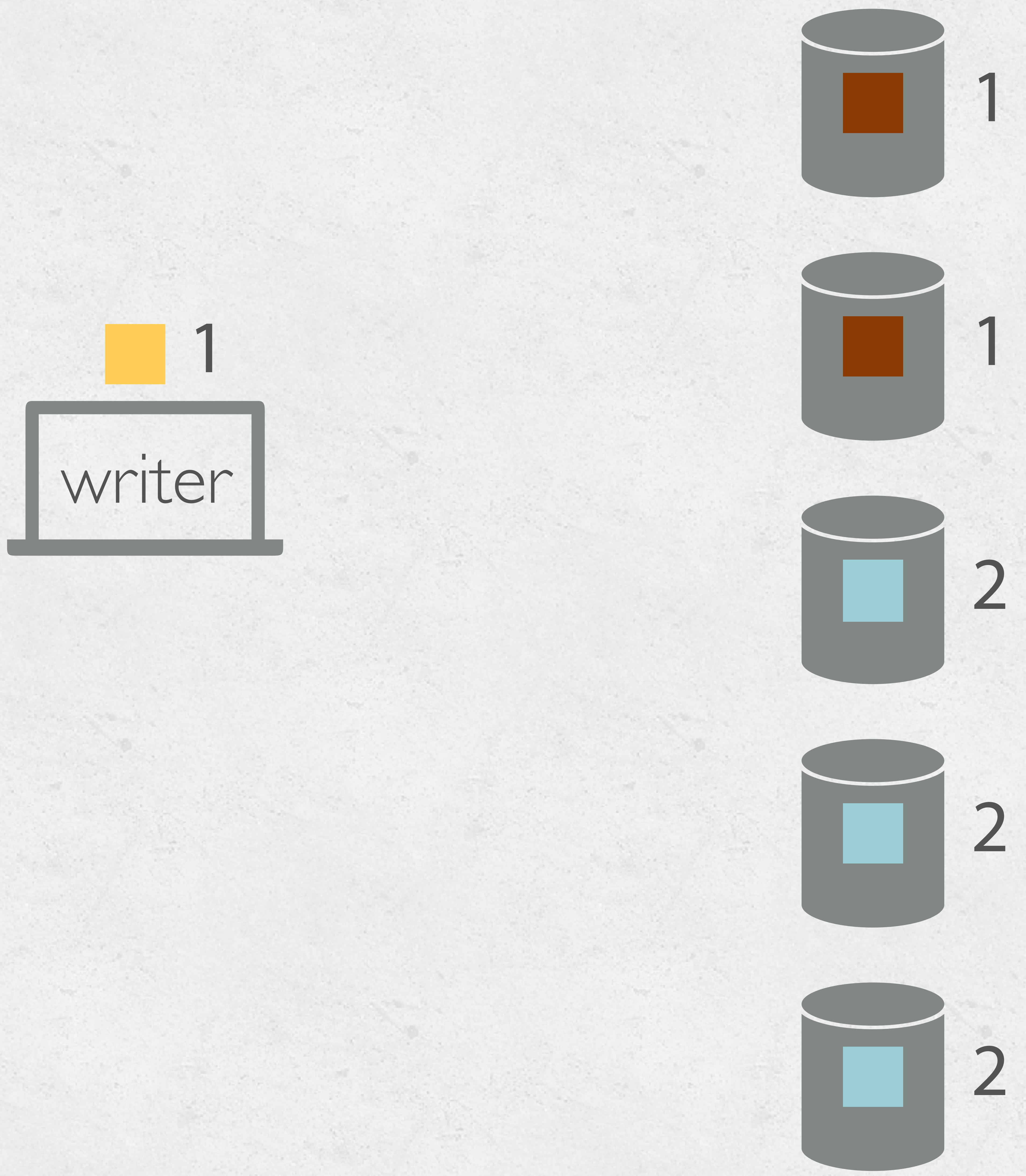
MRMW: Untimely Timestamps

Reads from a majority, sees blue value has the highest timestamp.

Not linearizable!

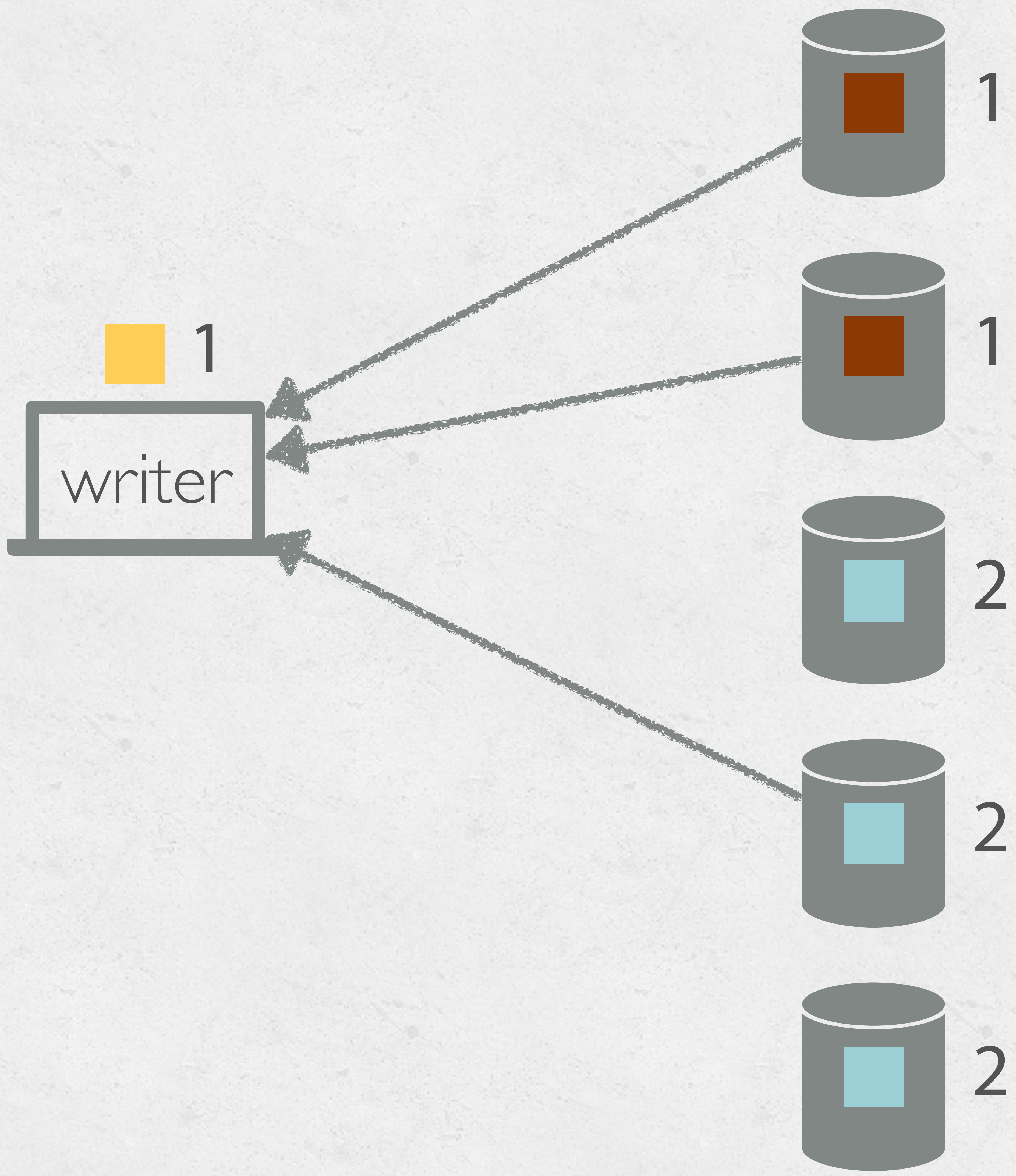
MRMW II: Ensuring Timestamp Ordering

- Writer first **queries** a majority, updates its timestamp to be larger than largest timestamp found.
- Writer then writes value to majority as usual.
- Written value guaranteed to have a timestamp larger than previously written values, readers will read latest value (again, writer IDs break timestamp ties).



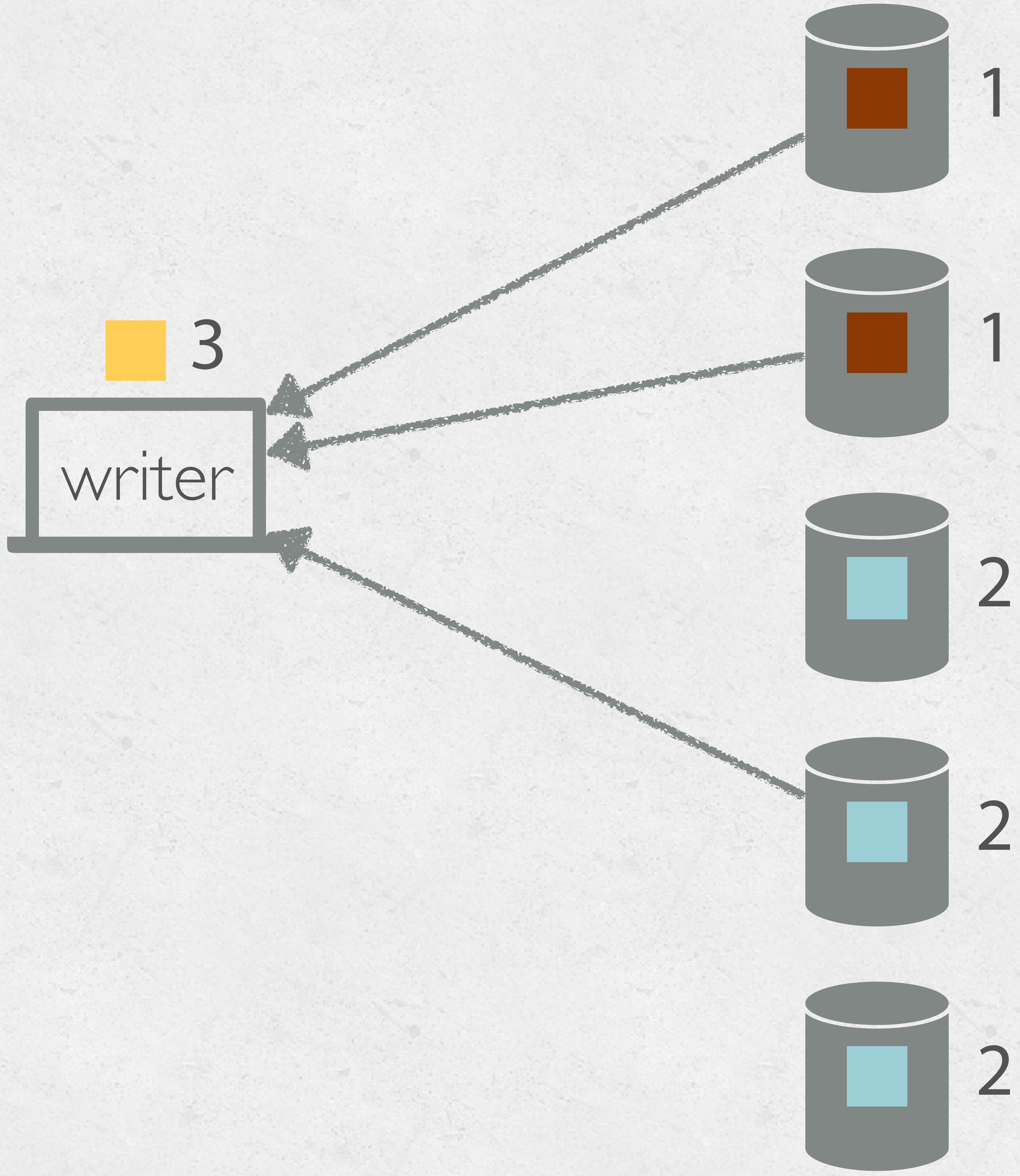
MRMW II: Ensuring Timestamp Ordering

- Writer first **queries** a majority, updates its timestamp to be larger than largest timestamp found.
- Writer then writes value to majority as usual.
- Written value guaranteed to have a timestamp larger than previously written values, readers will read latest value (again, writer IDs break timestamp ties).



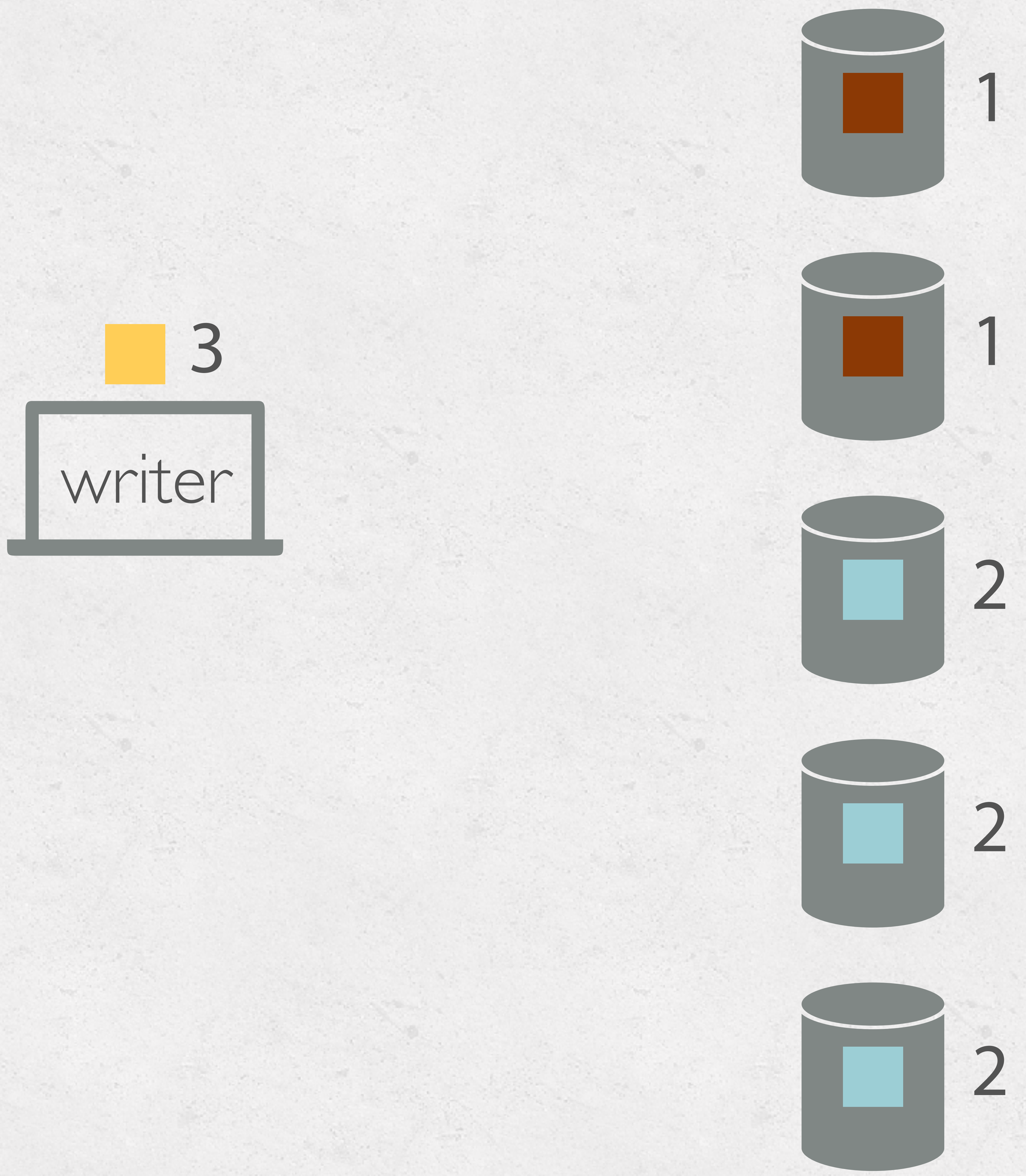
MRMW II: Ensuring Timestamp Ordering

- Writer first **queries** a majority, updates its timestamp to be larger than largest timestamp found.
- Writer then writes value to majority as usual.
- Written value guaranteed to have a timestamp larger than previously written values, readers will read latest value (again, writer IDs break timestamp ties).



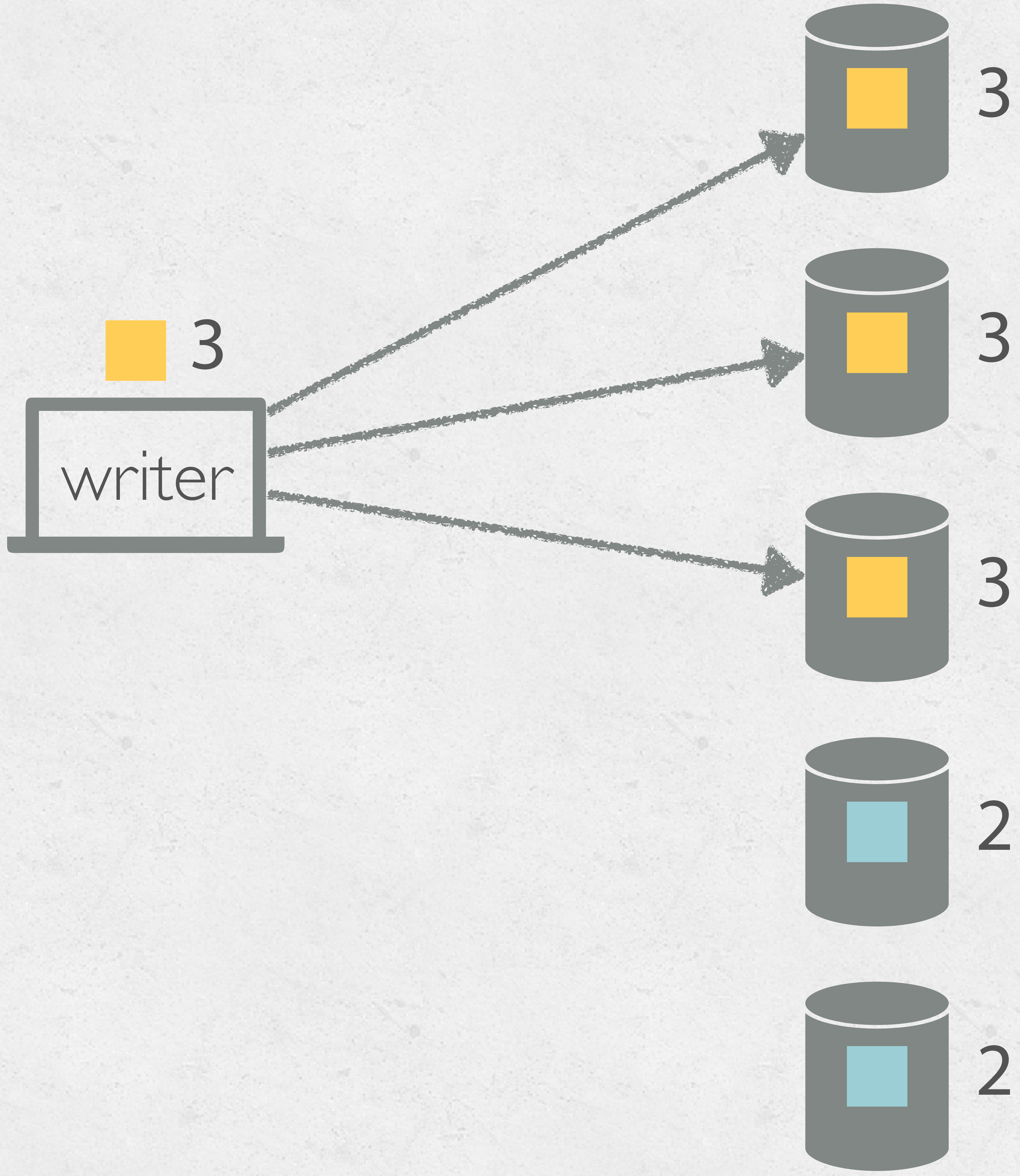
MRMW II: Ensuring Timestamp Ordering

- Writer first **queries** a majority, updates its timestamp to be larger than largest timestamp found.
- Writer then writes value to majority as usual.
- Written value guaranteed to have a timestamp larger than previously written values, readers will read latest value (again, writer IDs break timestamp ties).

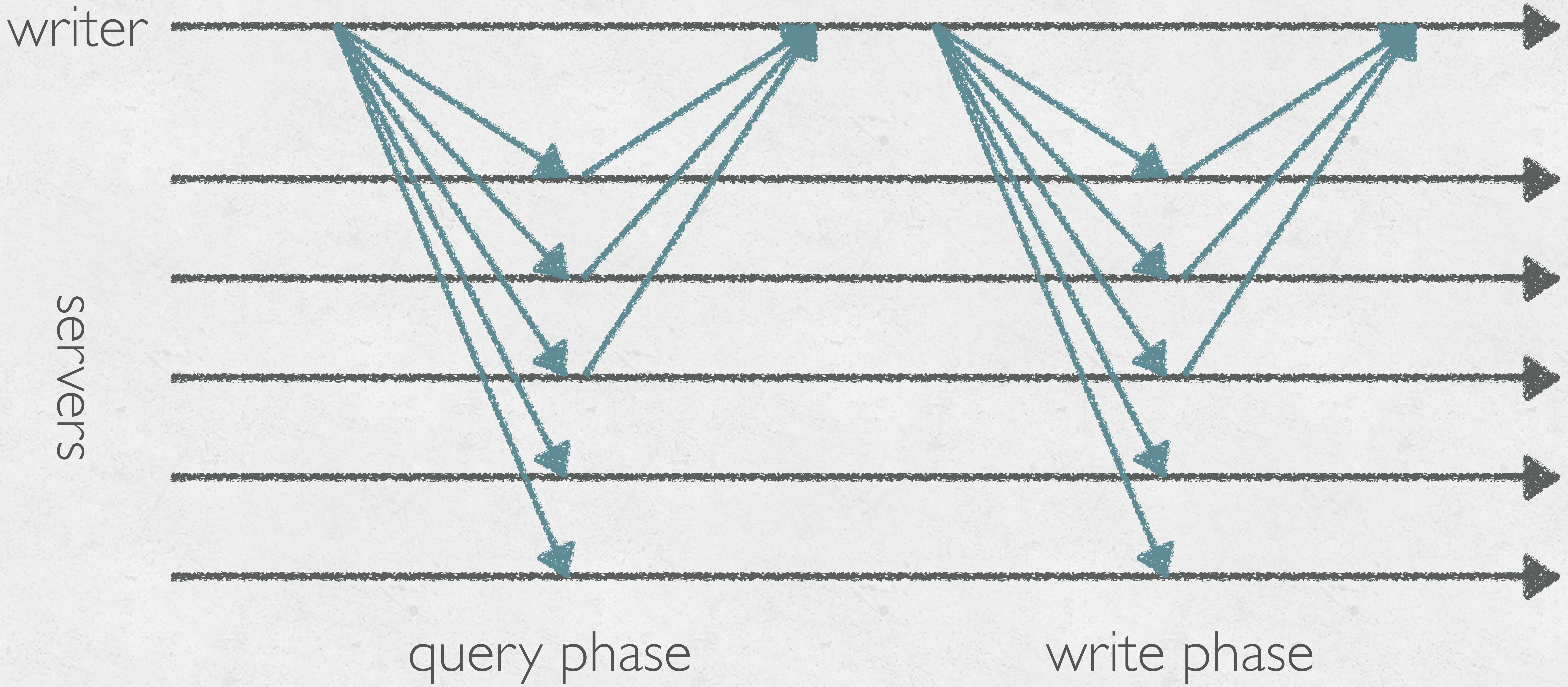


MRMW II: Ensuring Timestamp Ordering

- Writer first **queries** a majority, updates its timestamp to be larger than largest timestamp found.
- Writer then writes value to majority as usual.
- Written value guaranteed to have a timestamp larger than previously written values, readers will read latest value (again, writer IDs break timestamp ties).



MRMW III



Wait A Second!

- The methods for reading and writing are now the exact same.
- The only difference is that a read writes and returns the value that was read, but a write writes the value to be written.
- Also, for the record, there's no reason that processes can't be both readers and writers.

Attiya, **B**ar-Noy, **D**olev 1995 **ABD Algorithm**

ABD vs. Paxos

- Paxos doesn't guarantee liveness when the network is asynchrony. ABD guarantees wait-freedom, even when there are multiple writers.

- Paxos-based state-machine replication (SMR) can support arbitrary state machines. The ABD algorithm only allows a read/write interface.

- ABD removes the leader bottleneck, has the same latency cost as leader-based Paxos.

What Can We Do With Registers?

- Implement a read/write key-value store.
- Emulate shared memory.

Consensus isn't always the right problem! Don't solve it if you don't have to!