Wait-free registers
Randomized Consensus

Arvind Krishnamurthy

University of Washington

(Slides from: Ellis Michael)

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

- Weaken the safety guarantees and accept weaker consistency (at your own peril).
- Constrain the problem (wait-free registers)
- Allow randomness (randomized algorithms)

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.
- Provides **safe**, **regular**, and **atomic/linearizable** semantics

- **safe:** a read not concurrent with any write obtains the previously written value
- **regular:** safe + a read that overlaps a write obtains either the old or new value
- **atomic:** safe + reads and writes behave as if they occur in some definite order

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, no limit on number of client crashes.

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.

Why No Appends?

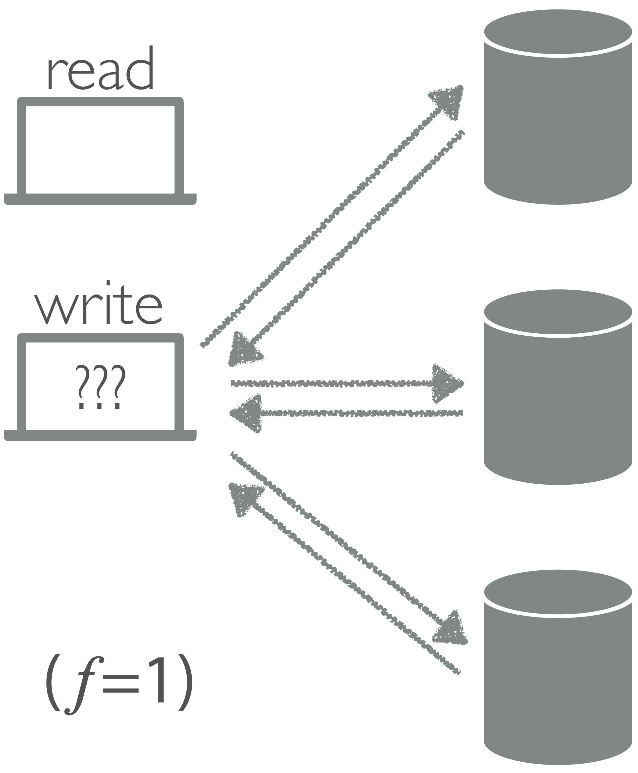
Simple way to implement consensus:

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

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

How Many Servers Do We Need?

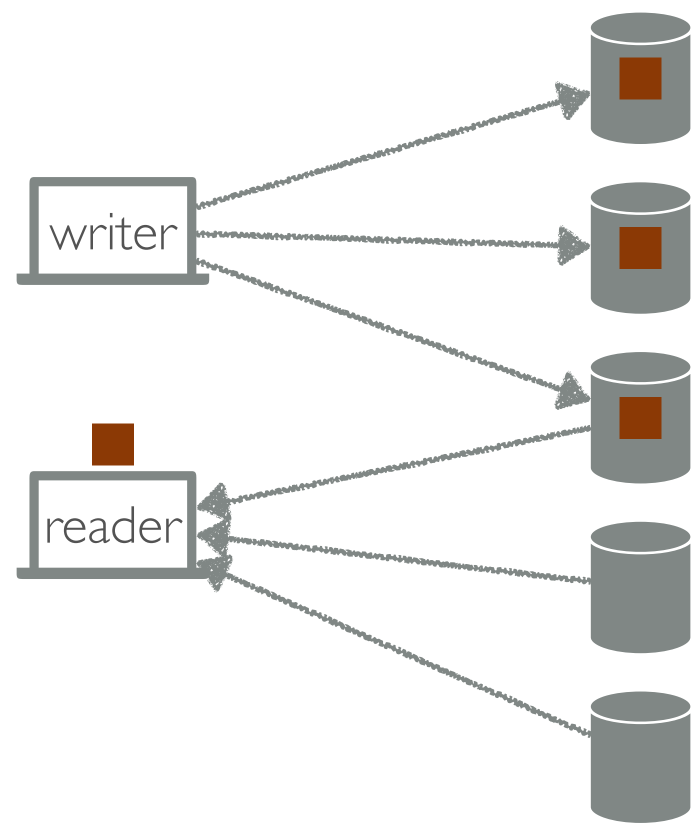
- If we want to make progress even when f servers crash, we can wait for at most n - f responses.
- We need to send writes to > f replicas, otherwise they could get lost forever.
- So we need *at least* 2 f+1 servers. And, in fact, we will use 2 f+1.
- Read quorum size plus write quorum size should be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

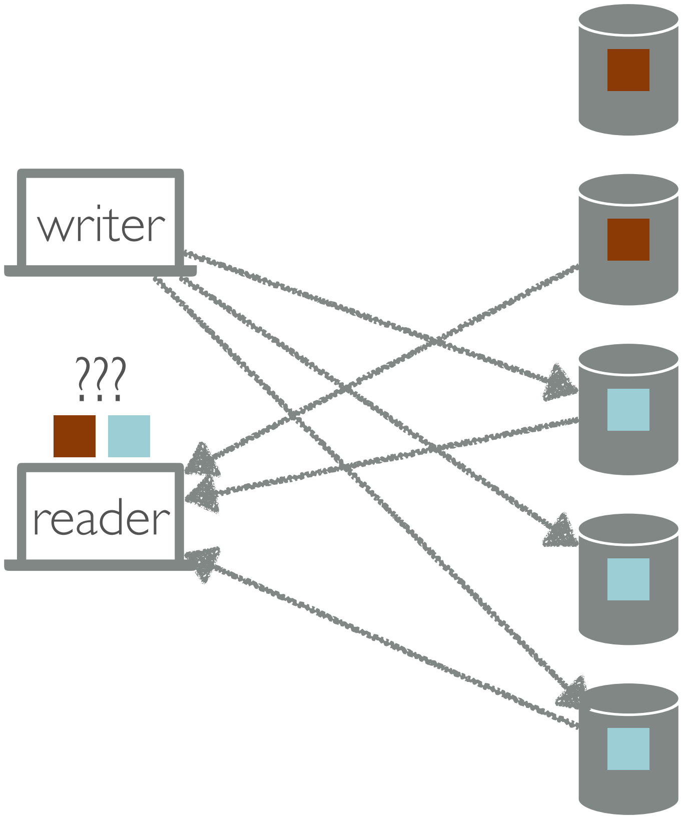
Does this work?



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.

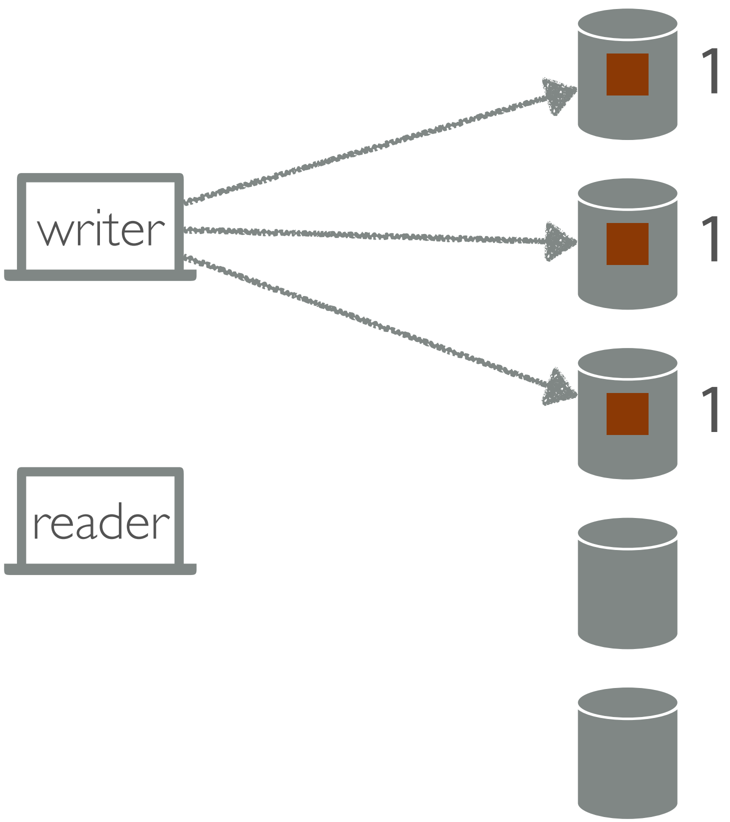
Does this work?



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 latest written value.

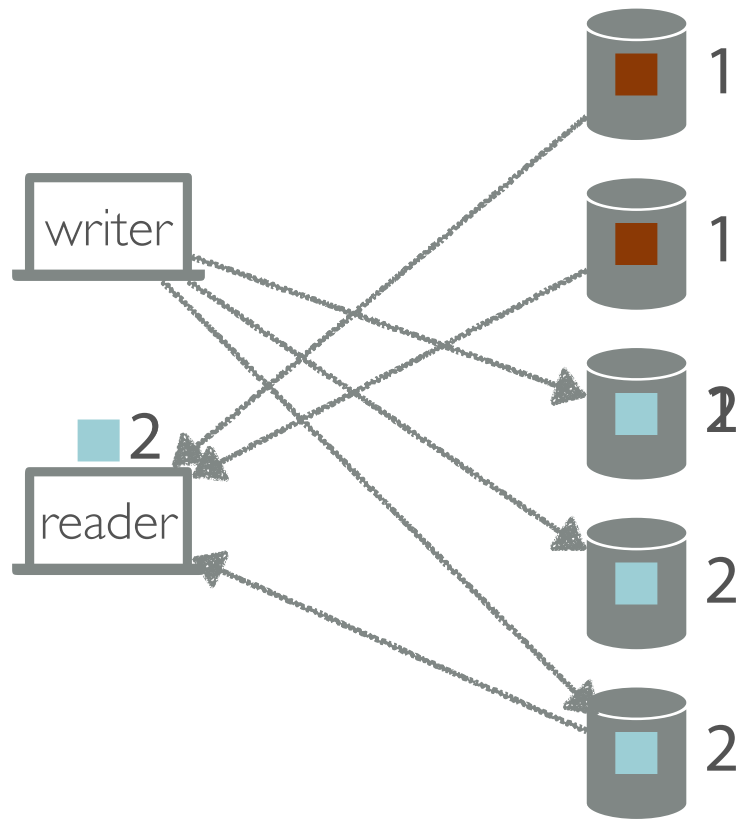
Does this work?



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 latest written value.

Does this work?

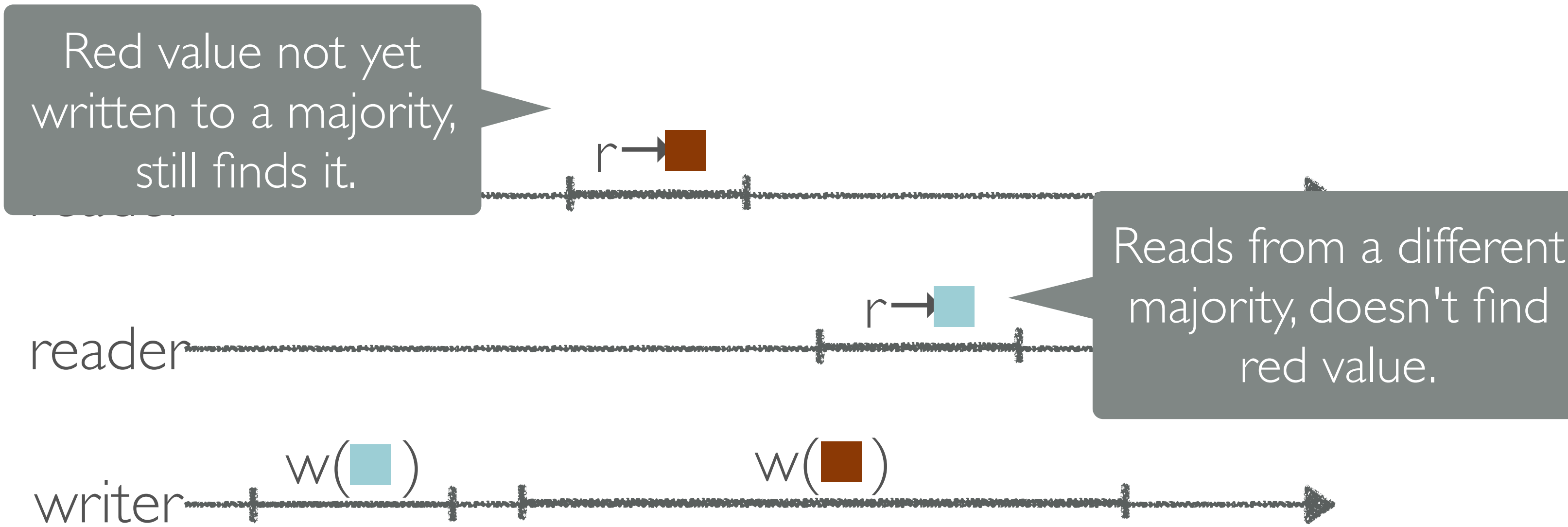


Multiple Readers, Single Writer (MRSW)

Does this previous solution just work?

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

MRSW: Inconsistent Reads

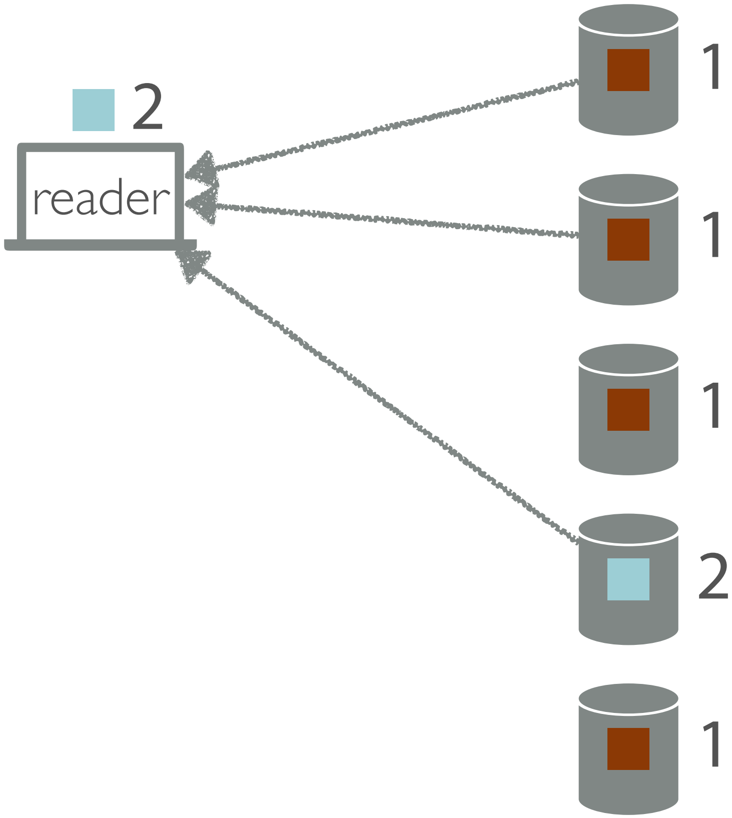


Not linearizable!

MRSW II

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

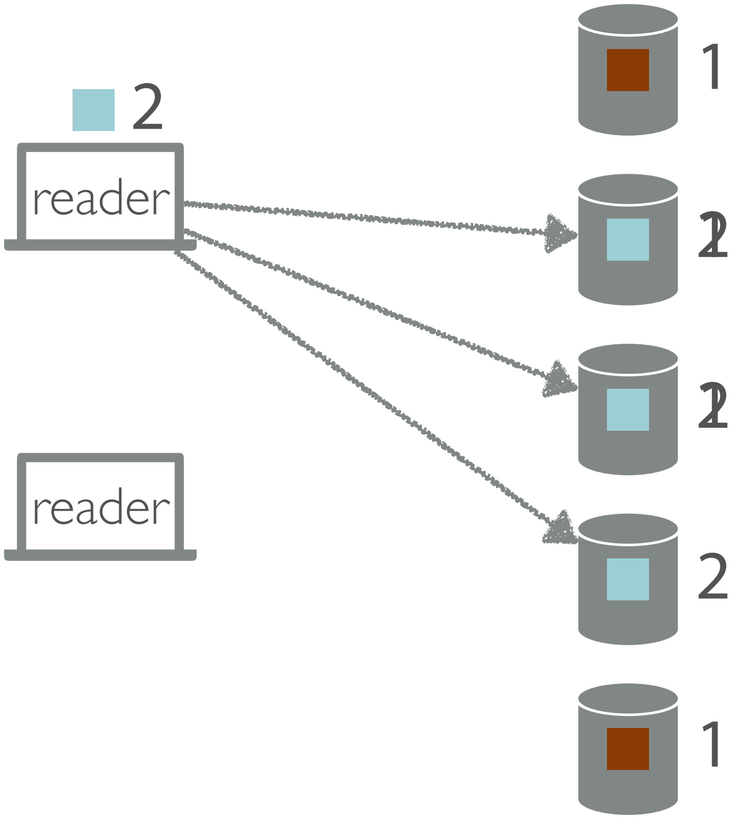
- Reader reads value from a majority, takes the one with the highest timestamp.
- Reader then performs a **write-back**, writing the value to a majority (not necessarily the same one). Only returns from read after write-back is complete.
- Later readers are guaranteed to read a value **at least as new**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



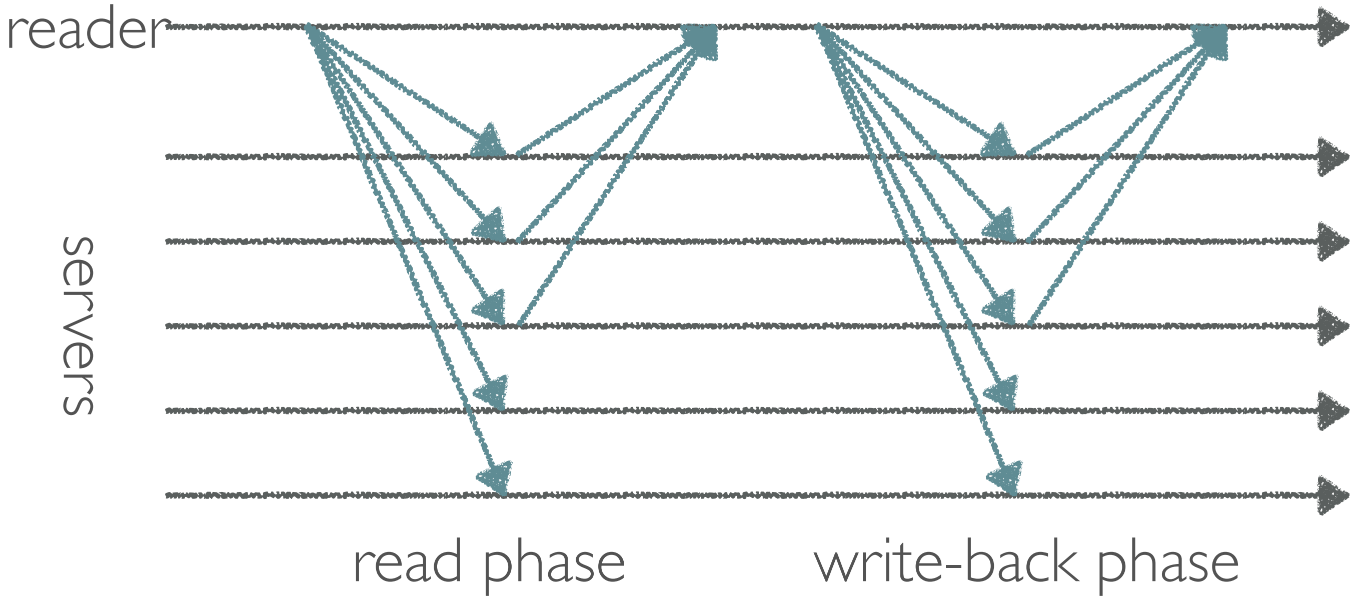
MRSW II

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?

Putting It All Together: MRMW

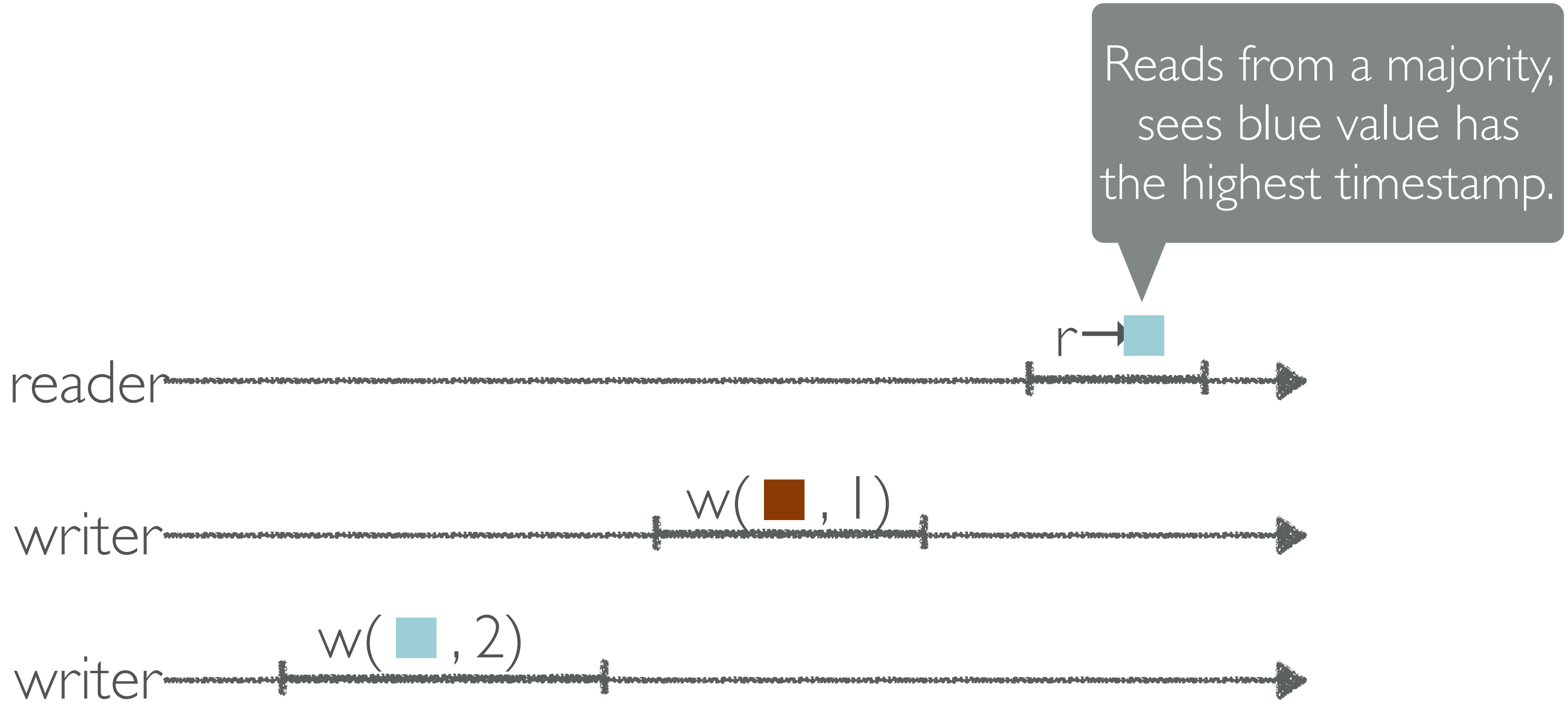
Does the previous solution just work?

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**?

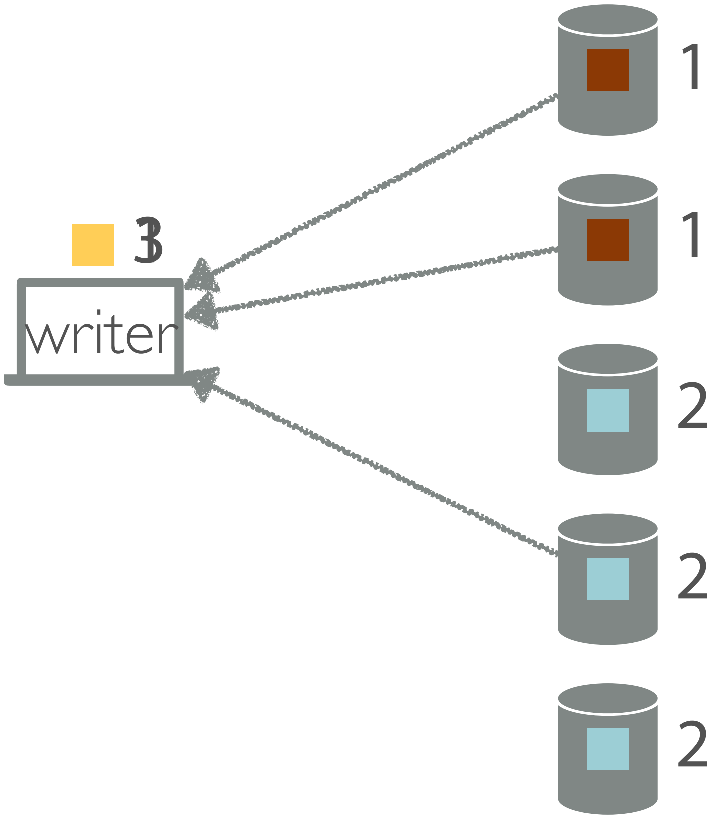
MRMW: Untimely Timestamps



Not linearizable!

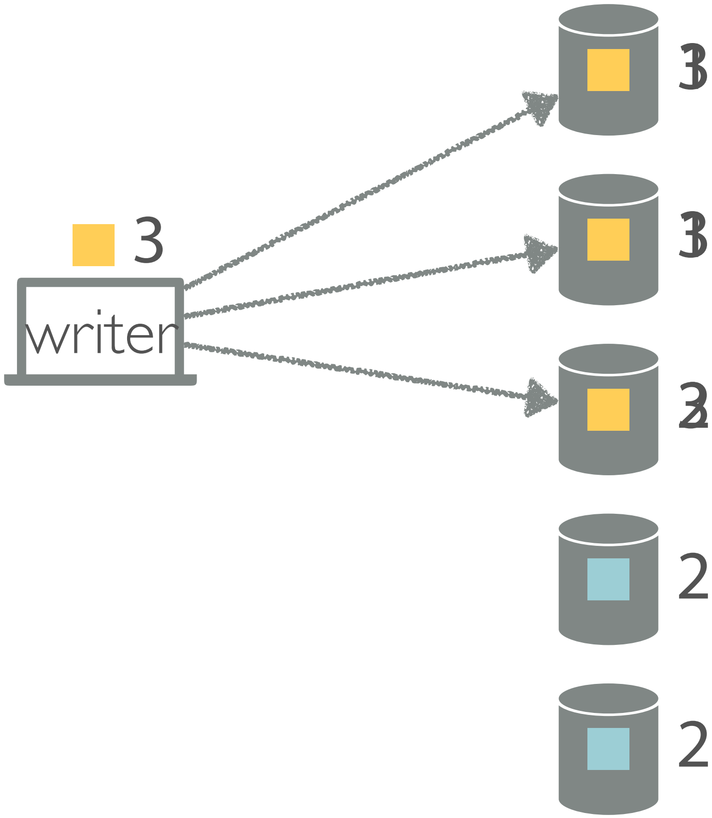
MRMW II: Ensuring Timestamp Ordering

- 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).

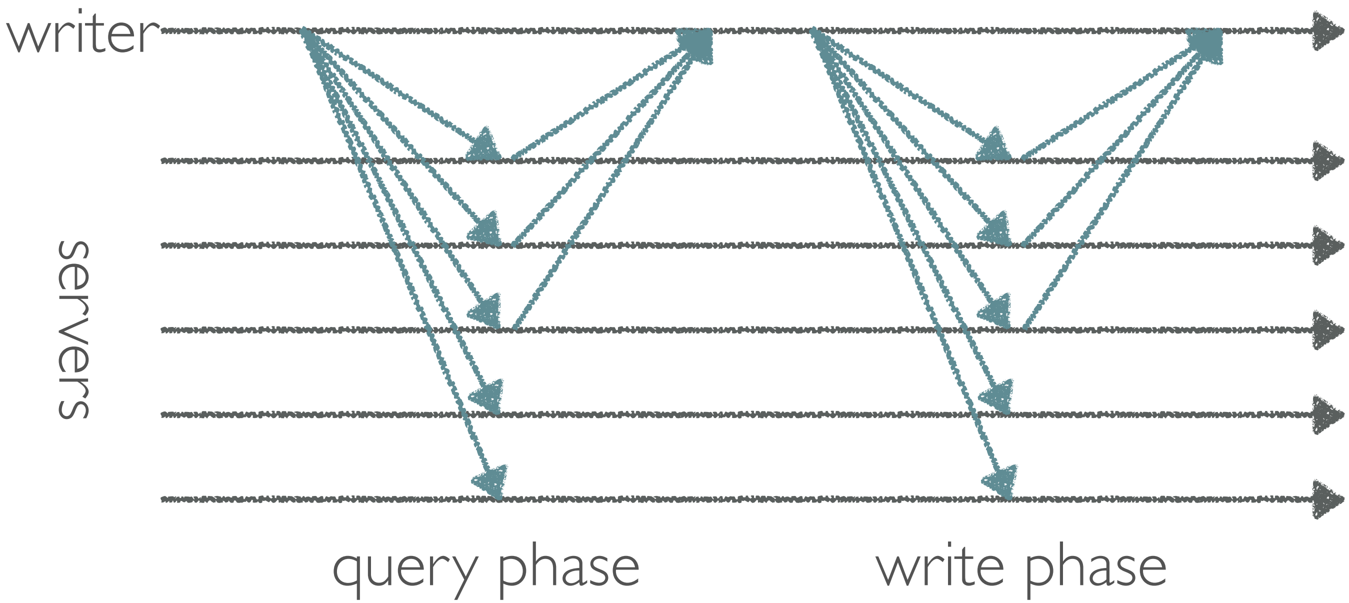


MRMW II: Ensuring Timestamp Ordering

- 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 III



Read/writes almost the same!

- The methods for reading and writing are now the 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 clients can't be both readers and writers.

ABD vs. Paxos

- Paxos doesn't guarantee liveness when the network is asynchronous. 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.
- How does its cost compare to 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!

Randomized Consensus

FLP Impossibility

Theorem: *In an asynchronous environment in which a single process can fail by crashing, there does not exist a protocol which solves binary consensus.*

Paxos doesn't save us. It doesn't guarantee liveness.

Result assumed a **deterministic** computation model.

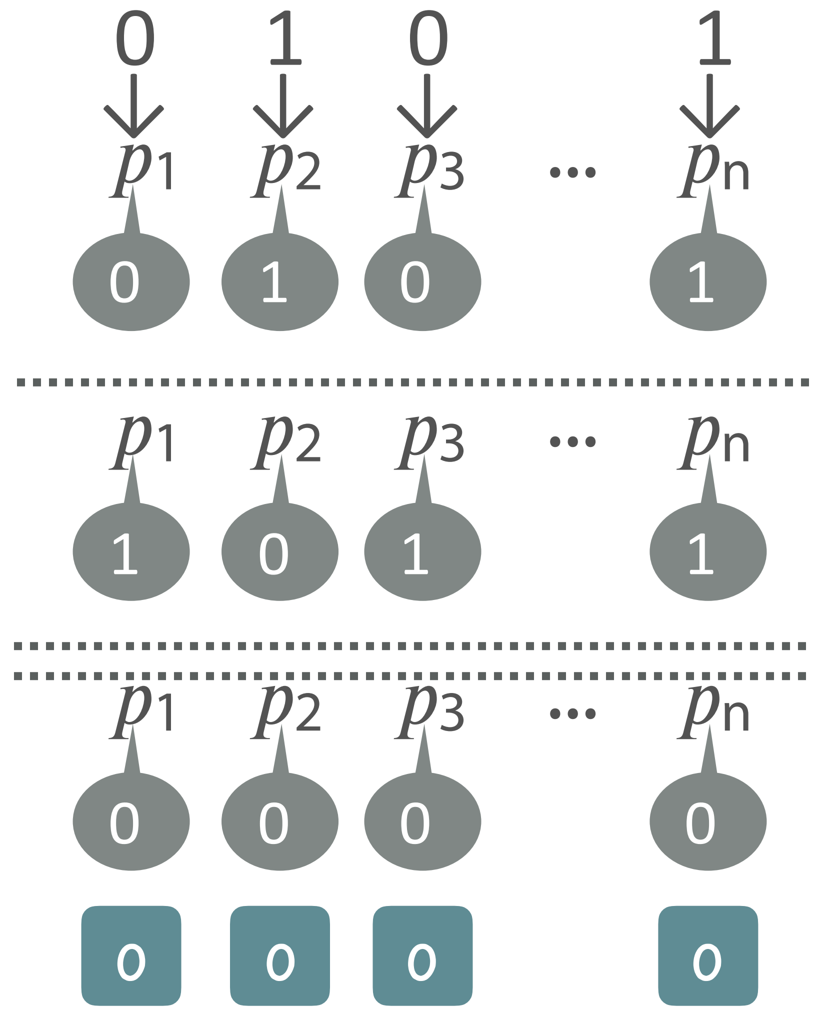
Let's go random!

Ben-Or's algorithm uses randomization to ***guarantee consensus*** for crash failures when $f < n/2$.

A variant even works for Byzantine faults!

Intuition

- At first every process proposes their input value.
- After that, they propose random values.
- When enough processes propose the same value, the value is chosen.
- Eventually, that will happen!



Setup

- Again, we're considering binary consensus.
- Protocol proceeds in **asynchronous rounds**, where each round has two phases.
- For each phase, processes broadcast their input values and wait for $n - f$ messages from the other processes.
- Each message is tagged with the round and phase number. (And messages can be resent to deal with a lossy network. But once a message is sent, that value is locked in for that process for that phase/round.)

Ben-Or Algorithm

Processes send proposals for each phase and then block and wait for the requisite $n - f$ messages (including their own).

During the first phase, processes make a preliminary proposal.

If they receive matching responses from a majority in the first phase, they propose that value in the second phase. Otherwise, they propose ⊥ (a special null value).

If they get enough non-⊥ responses from the second phase, they decide.

```
a←input
loop:
    send_phase1(a)
    A←receive_phase1()
    if (∃a′∈ A : |A_a′| > n/2):
        b←a′
    else:
        b←⊥

    send_phase2(b)
    B←receive_phase2()
    if (∃b′∈ B : b′≠⊥ ∧ |B_b′| > f):
        decide(b′)
    if (∃b′∈ B : b′≠⊥):
        a←b′
    else:
        a←choose_random({0,1})
```

Do We Have Consensus?

- **Agreement:** No two processes decide different values.
- **Integrity:** Every process decides at most one value, and if a process decides a value, some process had it as its input.
- **Termination:** Every correct process eventually decides a value.

$a \leftarrow$ *input*

loop:

    send_phase1(a)

    $A \leftarrow$ *receive_phase1*()

    if ($\exists a' \in A : |A_{a'}| > n/2$):

        $b \leftarrow a'$

    else:

        $b \leftarrow \perp$

    send_phase2(b)

    $B \leftarrow$ *receive_phase2*()

    if ($\exists b' \in B : b' \neq \perp \wedge |B_{b'}| > f$):

        decide(b')

    if ($\exists b' \in B : b' \neq \perp$):

        $a \leftarrow b'$

    else:

        $a \leftarrow$ *choose_random({0,1})*

Integrity I

If both 0 and 1 are input values to processes, integrity is trivially satisfied.

Suppose all processes have the same input value.

- Then, they all send the same phase 1 value in round 1.
- So they all send that same value in phase 2.
- So they all decide that value at the end of round 1.

```
a←input
loop:
    send_phase1(a)
    A←receive_phase1()
    if (∃a′∈ A : |A_a′| > n/2):
        b←a′
    else:
        b←⊥
    send_phase2(b)
    B←receive_phase2()
    if (∃b′∈ B : b′≠⊥ ∧ |B_b′| > f):
        decide(b′)
    if (∃b′∈ B : b′≠⊥):
        a←b′
    else:
        a←choose_random({0,1})
```

Fun Fact

Lemma: *No two processes receive different non-⊥ phase 2 values in the same round.*

Suppose they did. That means that one process received 0s from a majority in phase 1 and another received 1s.

But majorities intersect!

```
a←input
loop:
    send_phase1(a)
    A←receive_phase1()
    if (∃a′∈ A : |A_a′| > n/2):
        b←a′
    else:
        b←⊥

    send_phase2(b)
    B←receive_phase2()
    if (∃b′∈ B : b′≠⊥ ∧ |B_b′| > f):
        decide(b′)
    if (∃b′∈ B : b′≠⊥):
        a←b′
    else:
        a←choose_random({0,1})
```

Agrement + Integrity II

Let round r be the first round any process decides a value, 0 w.l.o.g.

If a process decided a value, it must have received $> f$ 0s in phase 2.

Which means that every process received at least one 0 because they all wait for $n - f$ messages. No process received a 1 by the previous lemma.

Therefore, on round $r + 1$ (and all subsequent rounds), all processes propose 0 and all processes decide 0.

```
a←input
loop:
    send_phase1(a)
    A←receive_phase1()
    if (∃a′∈ A : |A_a′| > n/2):
        b←a′
    else:
        b←⊥

    send_phase2(b)
    B←receive_phase2()
    if (∃b′∈ B : b′≠⊥ ∧ |B_b′| > f):
        decide(b′)
    if (∃b′∈ B : b′≠⊥):
        a←b′
    else:
        a←choose_random({0,1})
```

Termination

We know that if all processes propose the same value for a round, they all decide that value that round.

At worst, the probability of this happening on any particular round is $1/2^n$.

Why? By the previous lemma, all the non-random values are identical.

Over time, the probability of this happening on **at least one round** converges to 1.

$a \leftarrow$ *input*

loop:

    send_phase1(a)

    $A \leftarrow$ *receive_phase1*()

    if ($\exists a' \in A : |A_{a'}| > n/2$):

        $b \leftarrow a'$

    else:

        $b \leftarrow \perp$

    send_phase2(b)

    $B \leftarrow$ *receive_phase2*()

    if ($\exists b' \in B : b' \neq \perp \wedge |B_{b'}| > f$):

        decide(b')

    if ($\exists b' \in B : b' \neq \perp$):

        $a \leftarrow b'$

    else:

        $a \leftarrow$ *choose_random({0,1})*

Other Values?

Binary consensus is conceptually simple but not as useful. However, the algorithm can be to support larger domains, even when the processes don't know the domains *a priori* and even when some processes don't receive input values.

- Processes without input values start by proposing ⊥.
- Instead of randomly choosing from {0,1}, processes randomly choose from all non-⊥ values they've seen so far (in any message). Only choose ⊥ as a last resort.

$a \leftarrow$ *input*

loop:

  send_phase1(a)

  $A \leftarrow$ *receive_phase1*()

  if ($\exists a' \in A : |A_{a'}| > n/2$):

    $b \leftarrow a'$

  else:

    $b \leftarrow \perp$

  send_phase2(b)

  $B \leftarrow$ *receive_phase2*()

  if ($\exists b' \in B : b' \neq \perp \wedge |B_{b'}| > f$):

    decide(b')

  if ($\exists b' \in B : b' \neq \perp$):

    $a \leftarrow b'$

  else:

    $a \leftarrow$ *choose_random({0,1})*

Takeaways

- Randomization can actually solve consensus*
- You can structure an asynchronous protocol using rounds. It's potentially useful and certainly an interesting way to think about asynchronous computation.