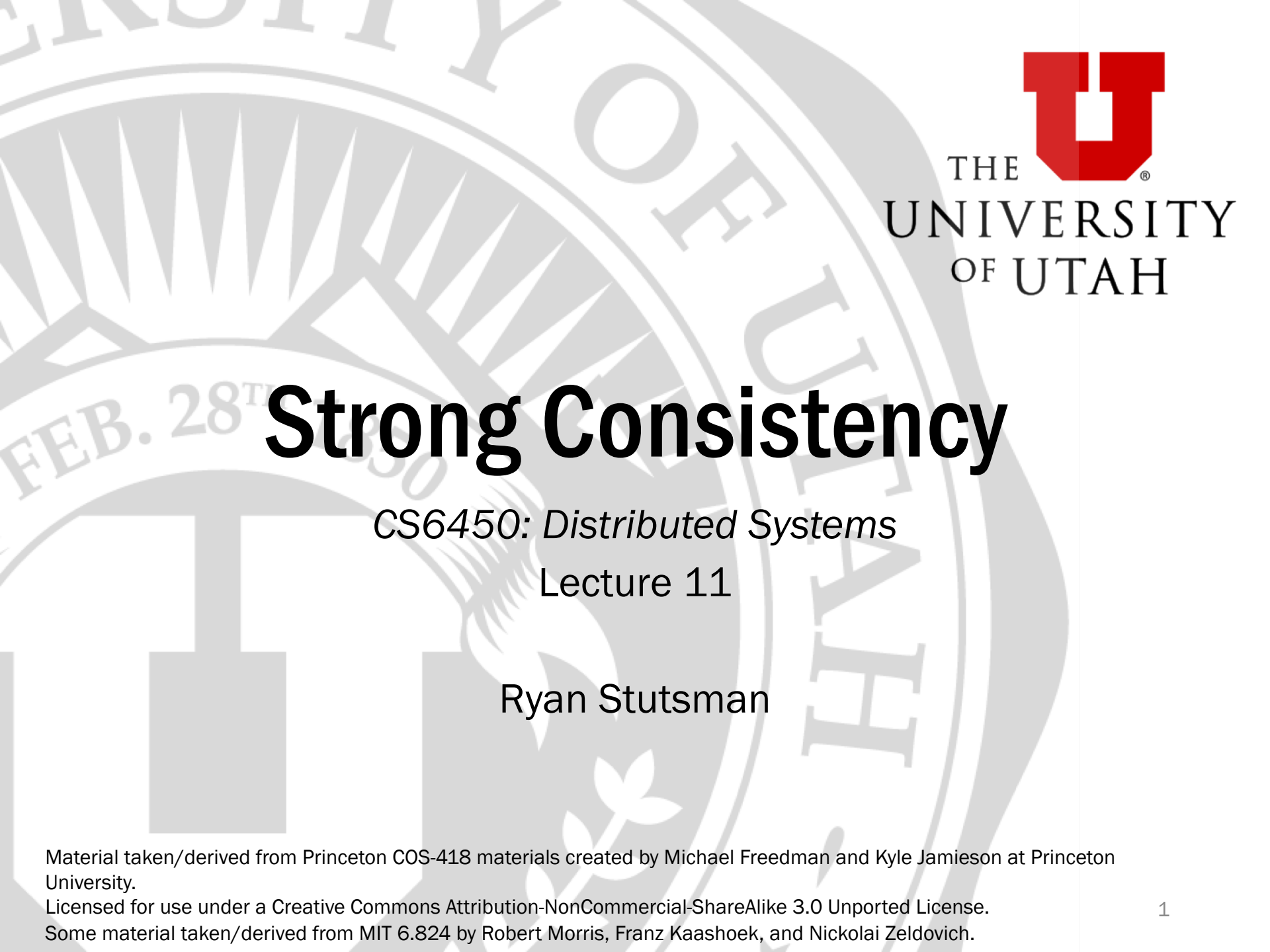


Strong Consistency

CS6450: Distributed Systems

Lecture 11

Ryan Stutsman

Takeaways

- A consistency model defines what clients can observe when interacting with different replicas of a system

- Strong models behave more like local structures
  - But, they can't make progress under (some) partitions
  - CAP theorem shows/states this
  - Linearizability, Sequential Consistency

- Weaker models admit more schedules
  - They force less order in replication
  - They can also continue operation despite partitions
  - Eventual Consistency, Causal Consistency

Consistency models

2PC / Consensus

Eventual consistency

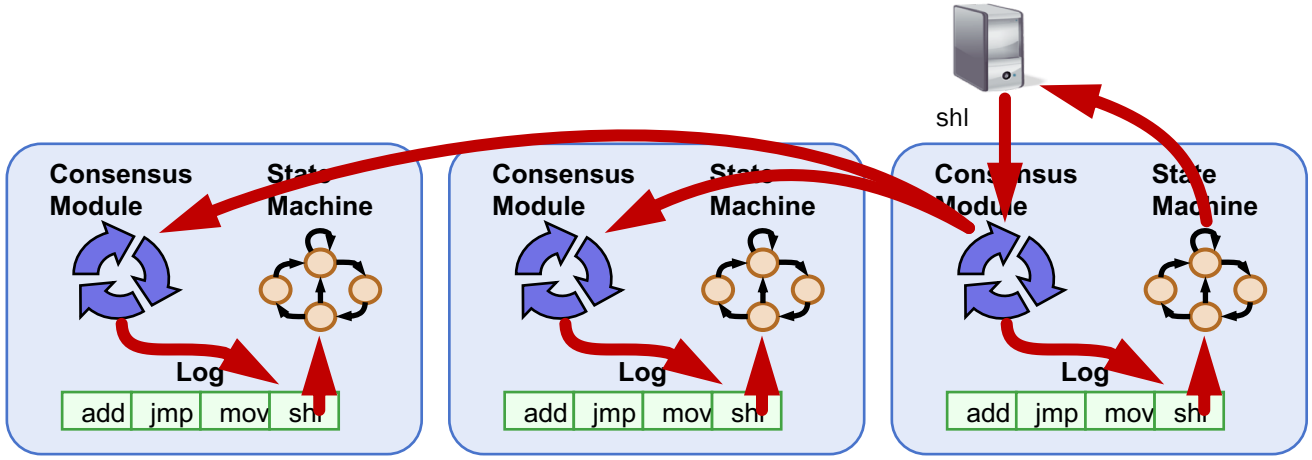
Paxos / Raft

Strong ordering,
Stops under
some
partitions

Dynamo

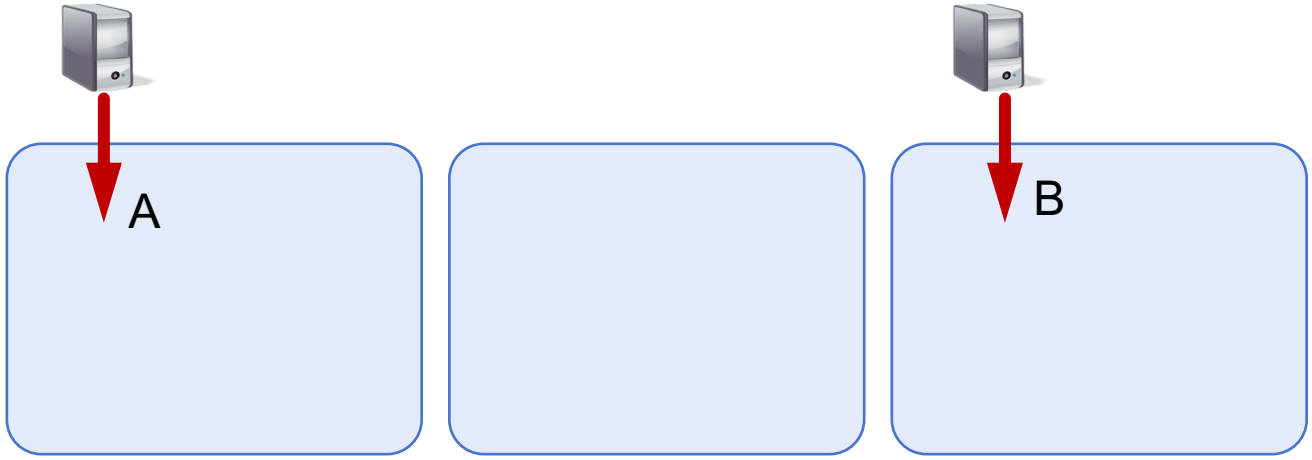
Weak ordering,
Works under
partitions

Consistency in Paxos/Raft



- Fault-tolerance / durability: Don't lose operations
- Consistency: Ordering between (visible) operations

Correct consistency model?



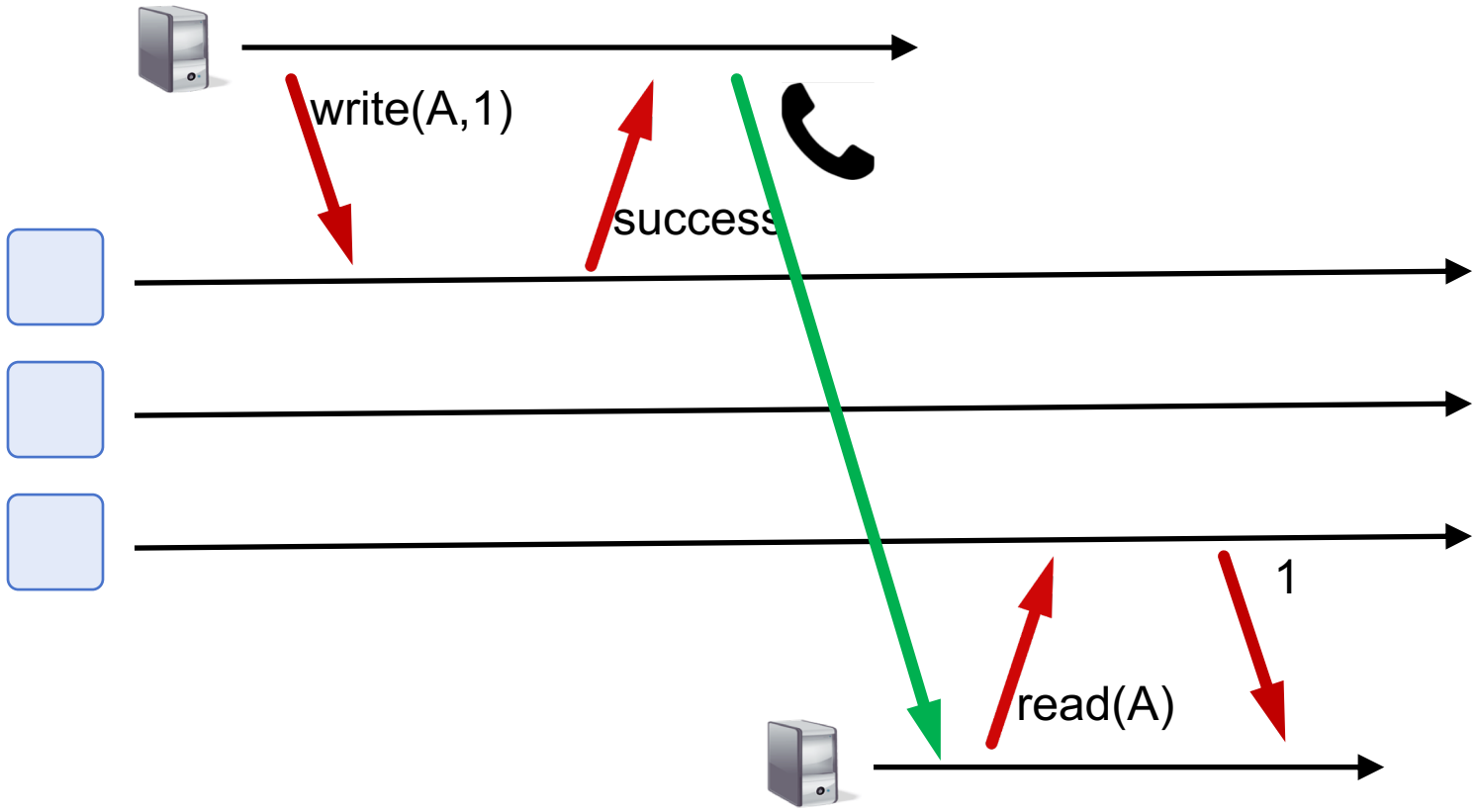
- Let's say A and B send an op.
- All readers see A → B ?
- All readers see B → A ?
- Some see A → B and others B → A ?

Paxos/Raft has *strong consistency*

- Provide behavior of a single copy of object:
  - Read should return the most recent write
  - Subsequent reads should return same value, until next write

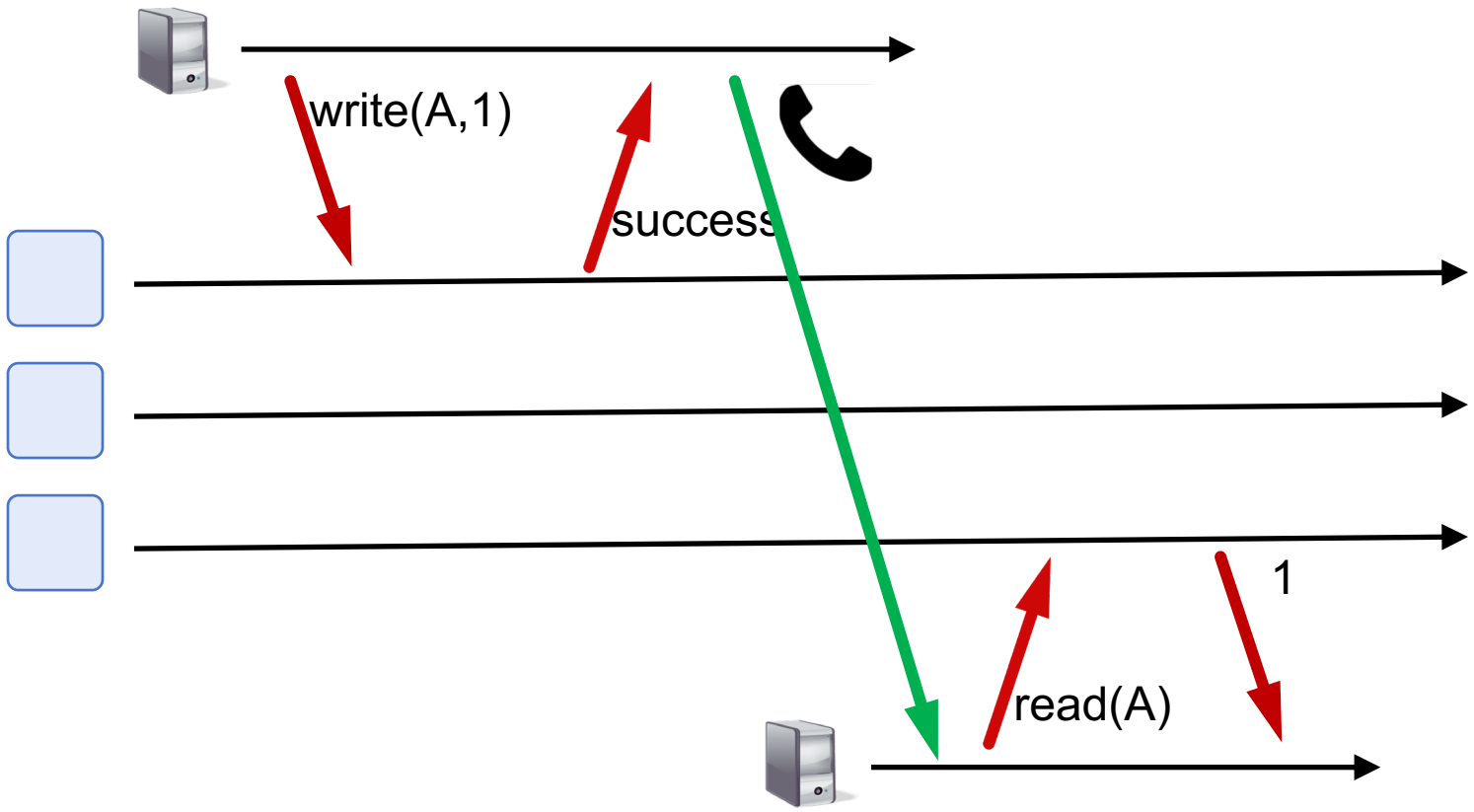
- Telephone intuition:
  1. Alice updates Facebook post
  2. Alice calls Bob on phone: “Check my Facebook post!”
  3. Bob read’s Alice’s wall, sees her post

Strong Consistency?



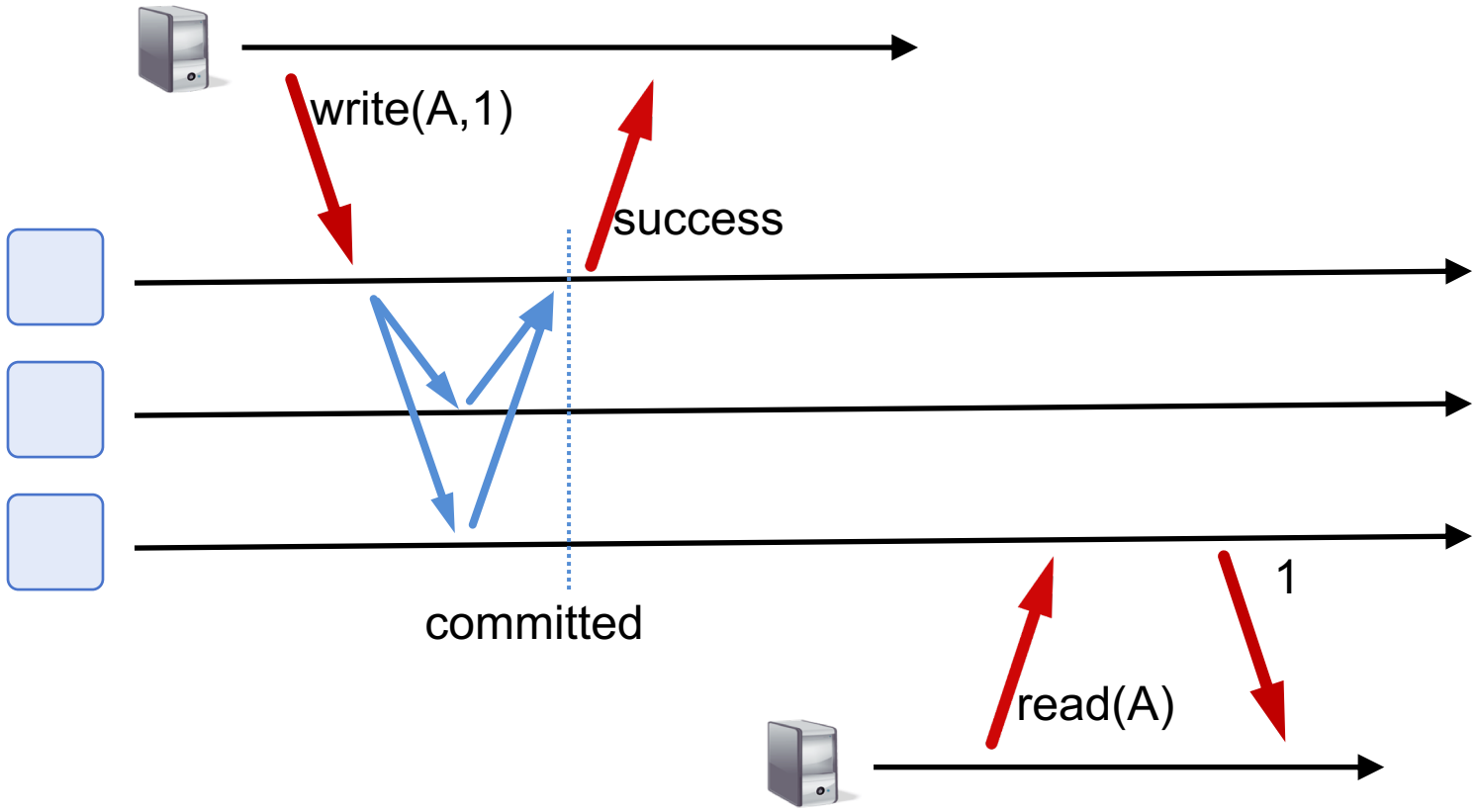
Phone call: Ensures *happens-before* relationship,
even through "out-of-band" communication

Strong Consistency?



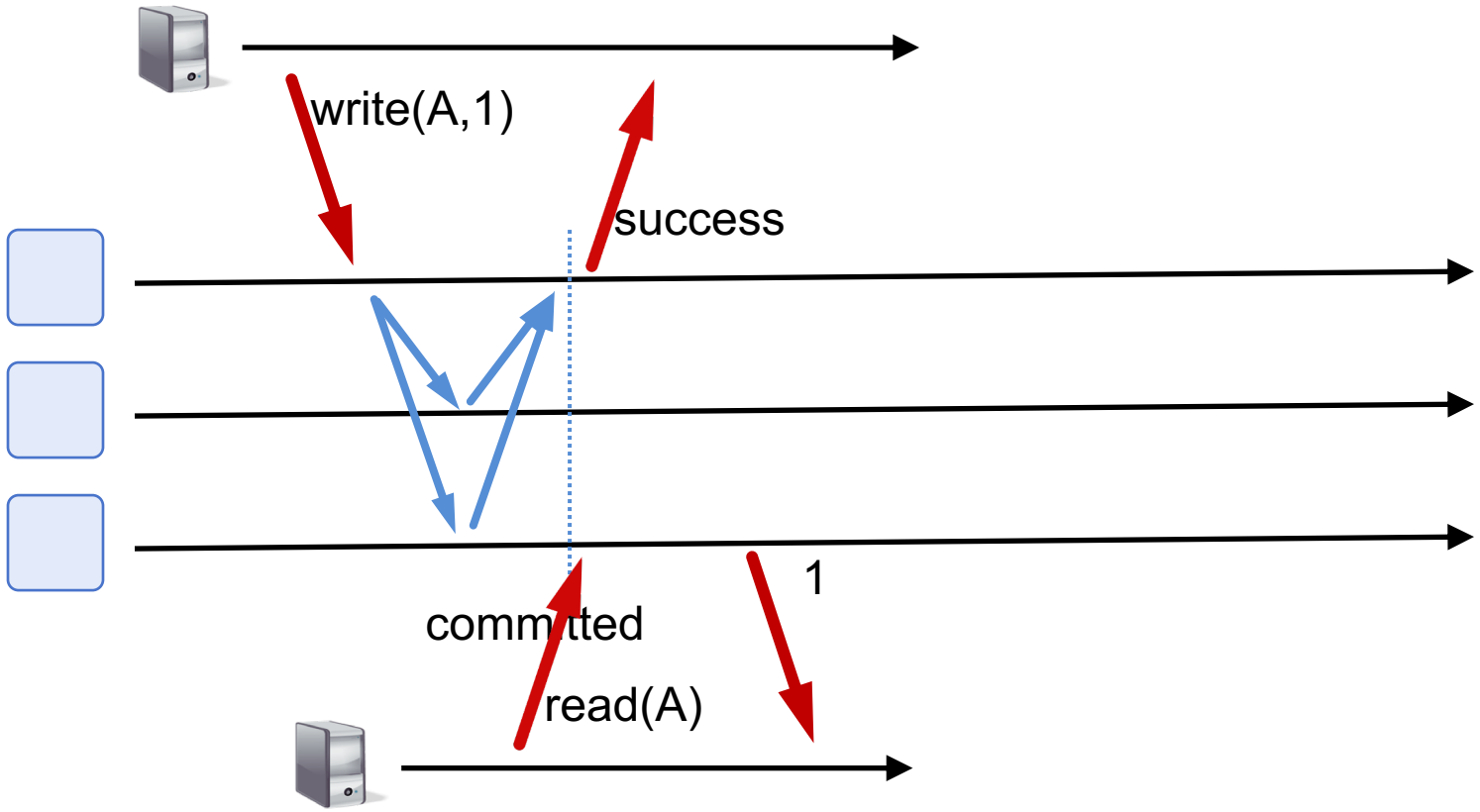
One cool trick: Delay responding to writes/ops until properly committed

Strong Consistency? This is buggy!



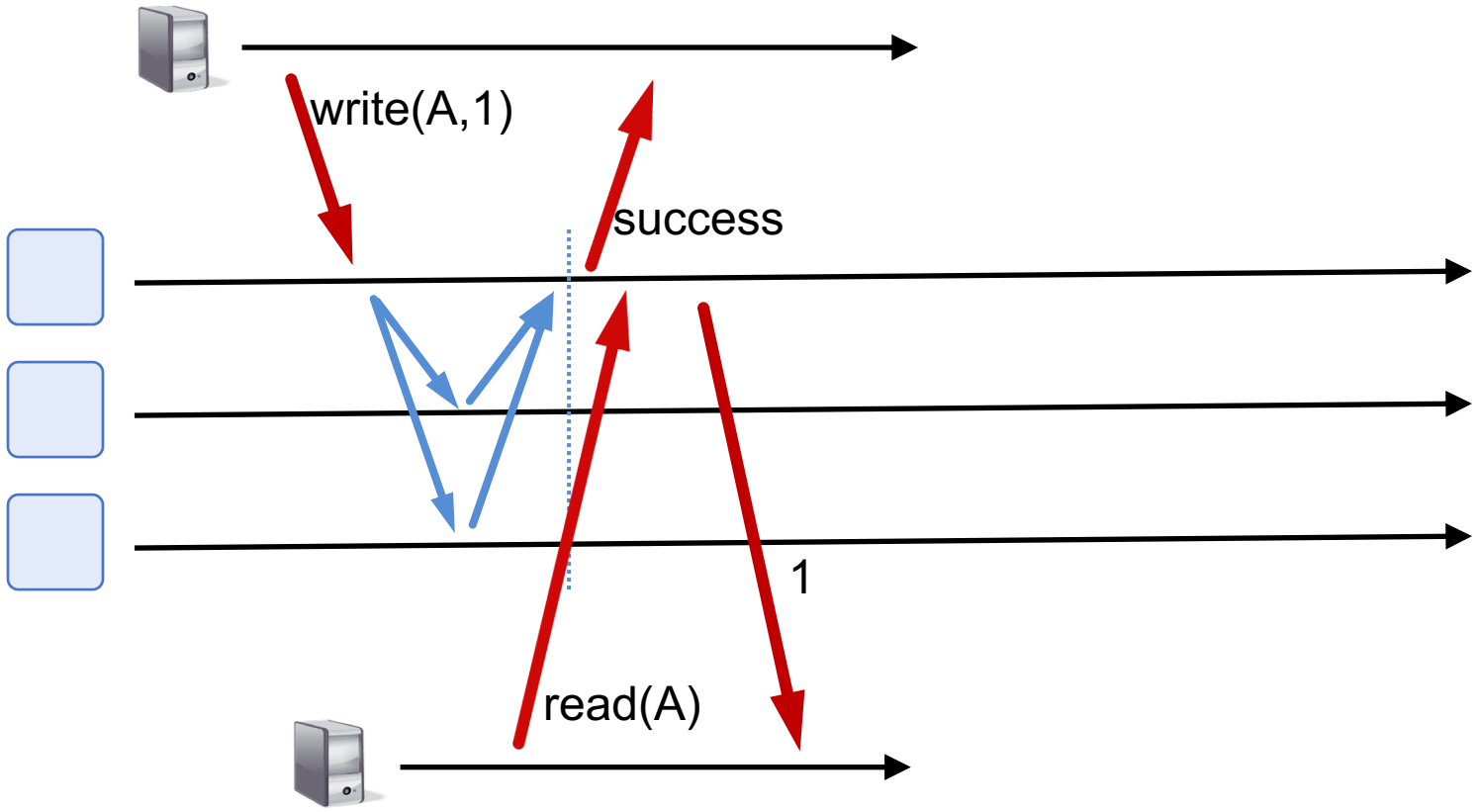
- Isn't sufficient to return value of third node: It doesn't know precisely when op is "globally" committed

Strong Consistency? This is buggy!



- Isn't sufficient to return value of third node: It doesn't know precisely when op is "globally" committed
- Instead: Need to actually *order* read operation

Strong Consistency!



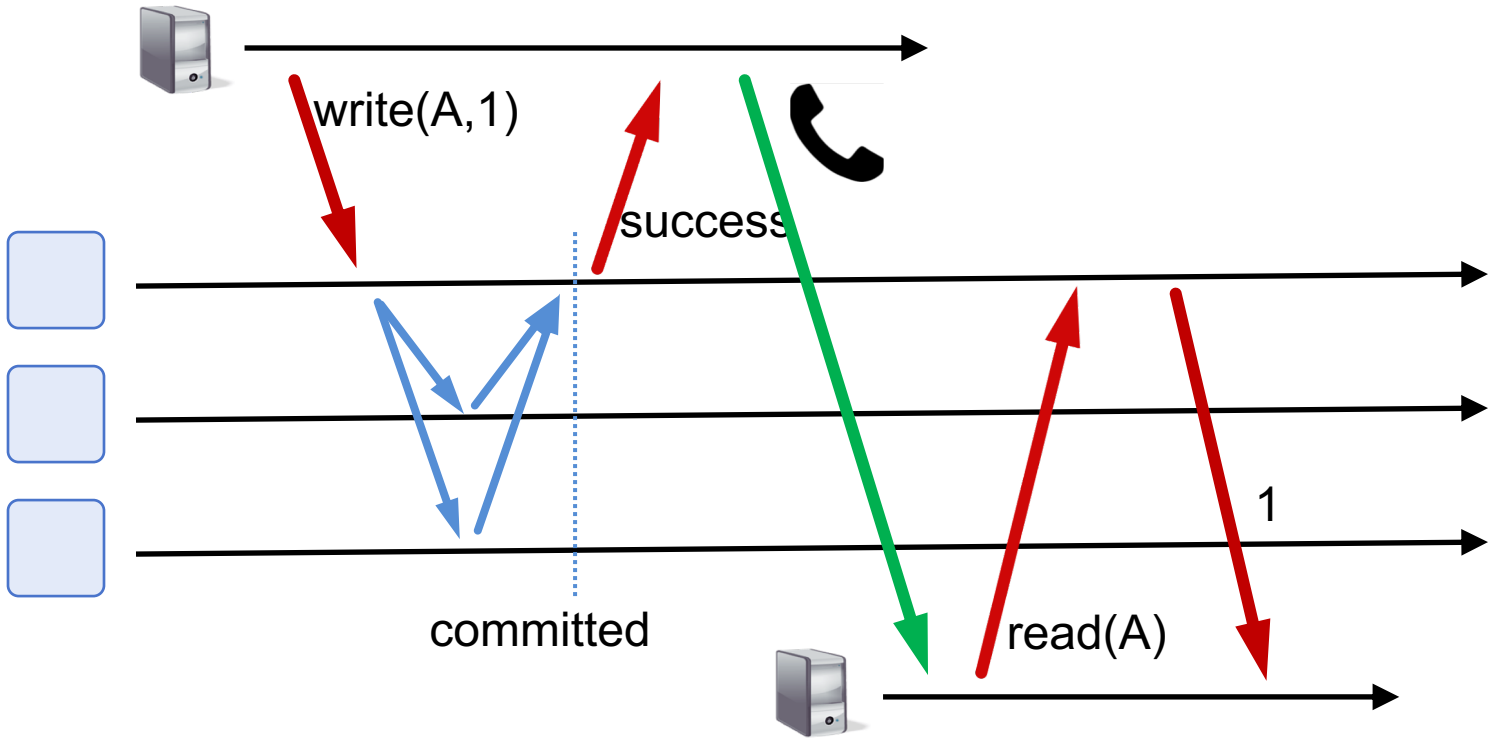
Order all operations via (1) leader, (2) consensus

Strong consistency = linearizability

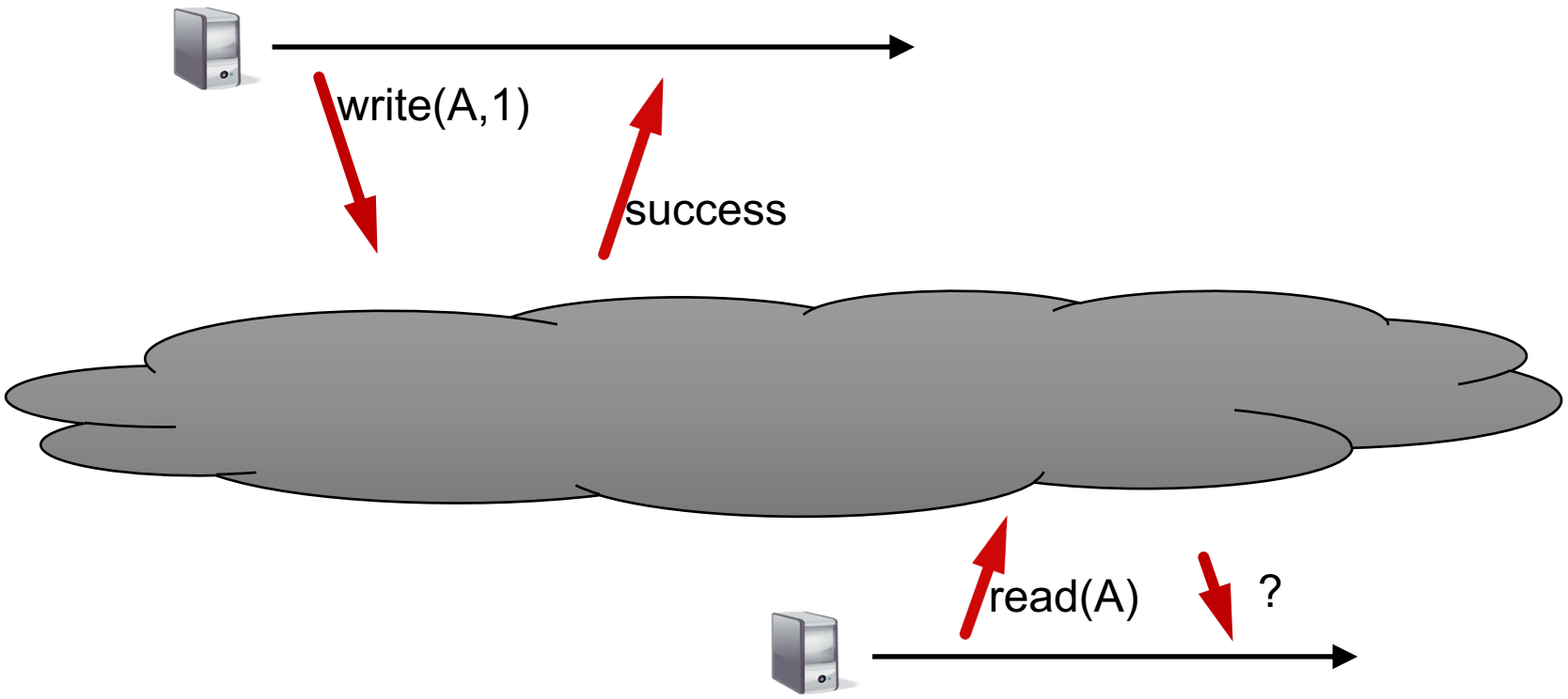
- Linearizability (Herlihy and Wang 1991)
  1. All operations on an object *appear as if* they are executed *atomically* in *some* (single, total) sequential order
  2. Global ordering preserves each client's own local ordering
  3. Global ordering consistent with invocation and response
     - Preserves real-time guarantee
     - If X completes before Y starts, then X precedes Y in sequence

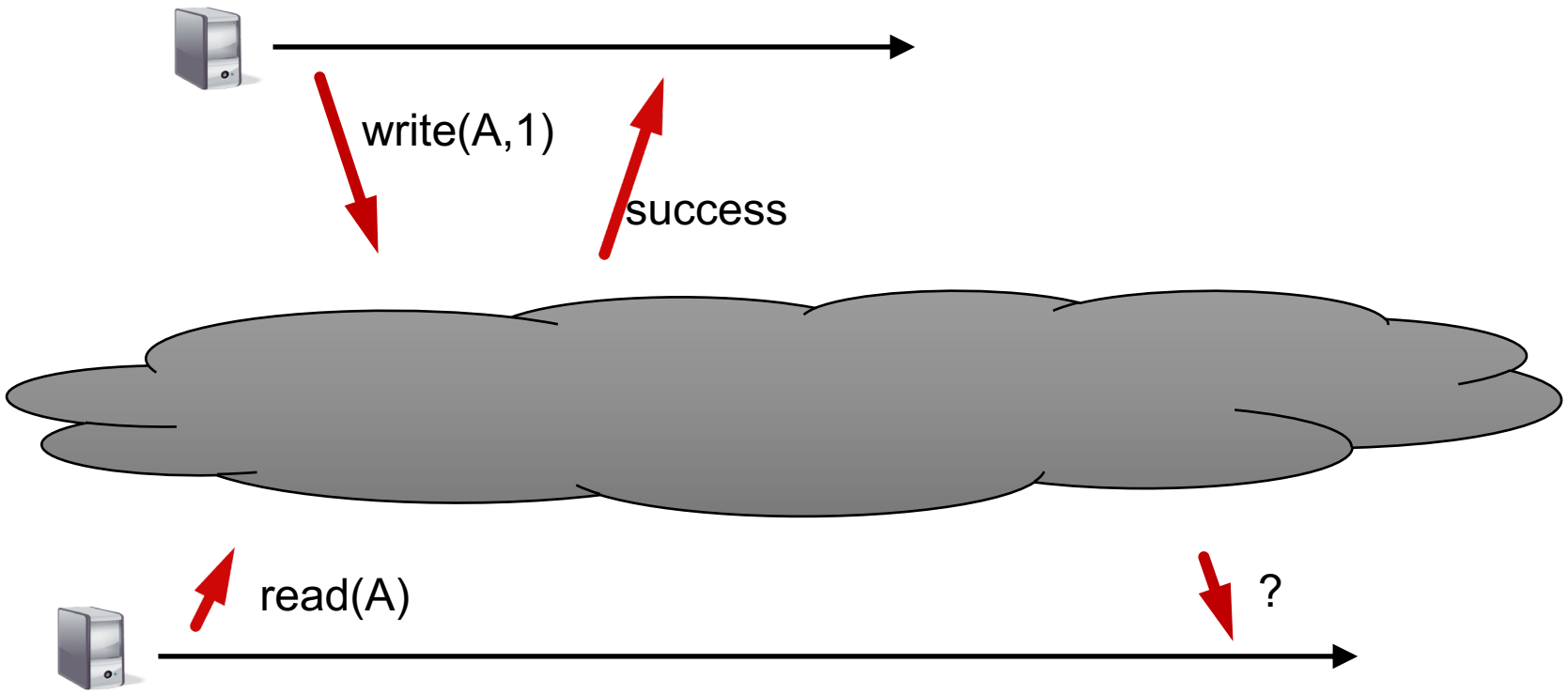
- Once write completes, all later reads (by wall-clock start time) should return value of that write or value of later write.
- Once read returns particular value, all later reads should return that value or value of later write.

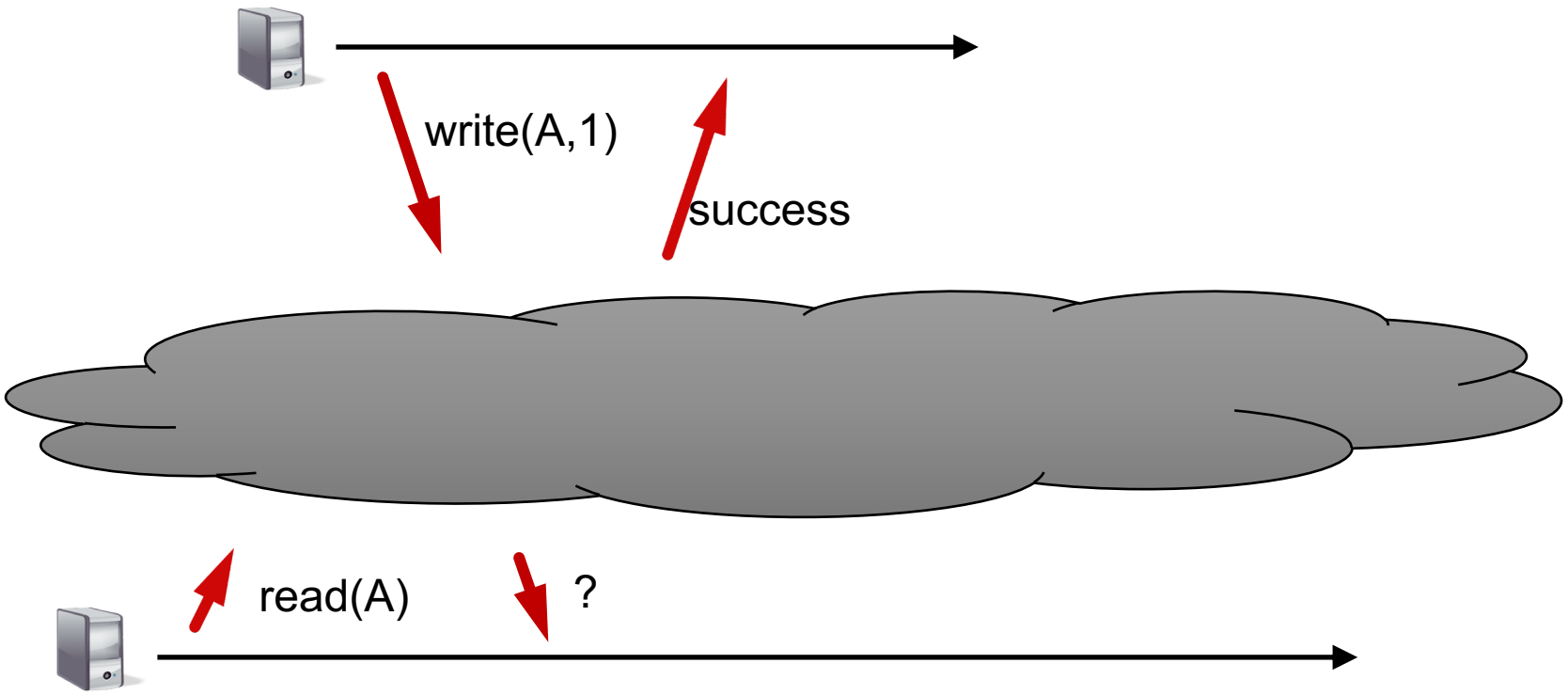
Intuition: Real-time ordering

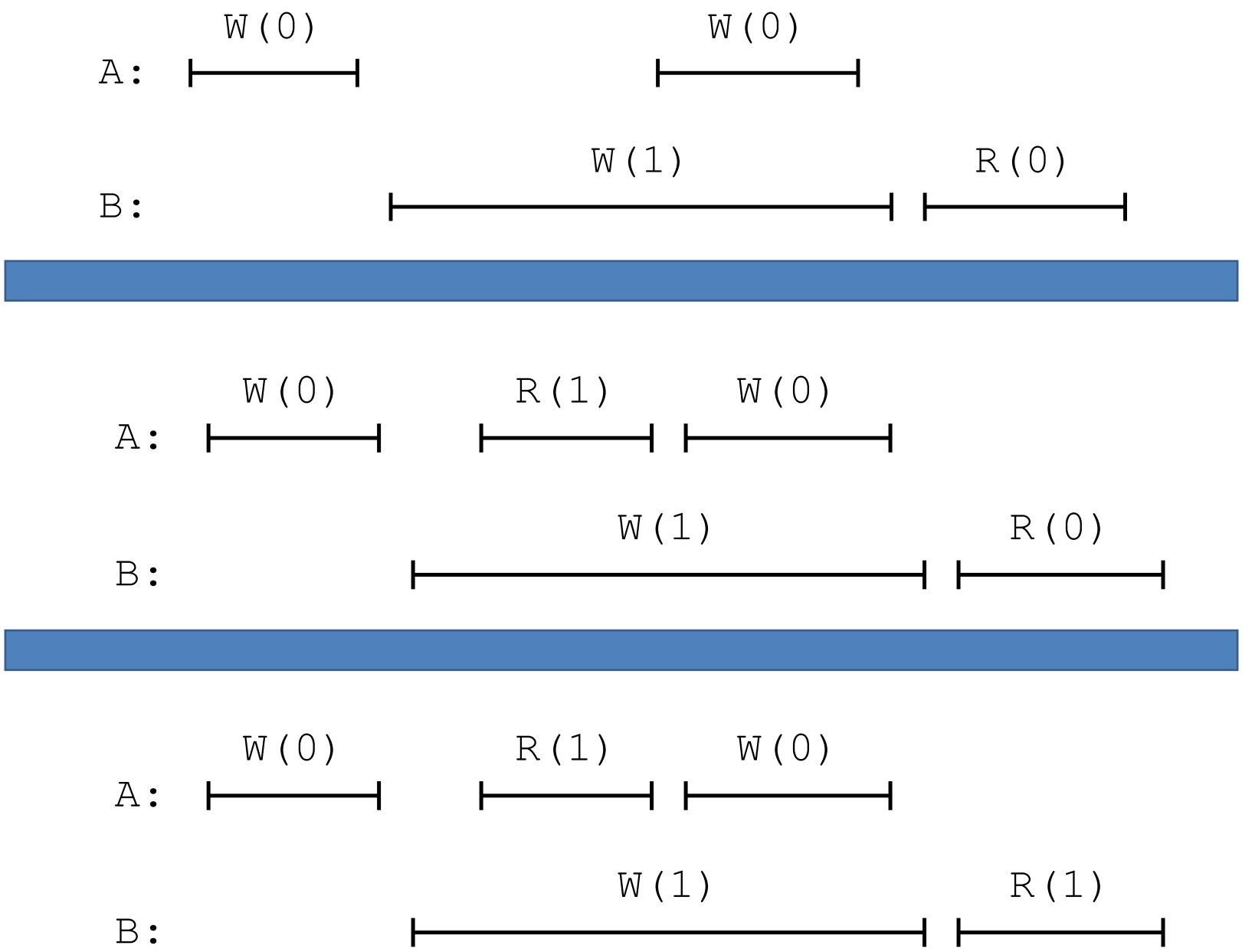


- Once write completes, all later reads (by wall-clock start time) should return value of that write or value of later write.
- Once read returns particular value, all later reads should return that value or value of later write.


write(A,1)
success
read(A)
?


write(A,1)
success
read(A)
?


write(A,1)
success
read(A)
?

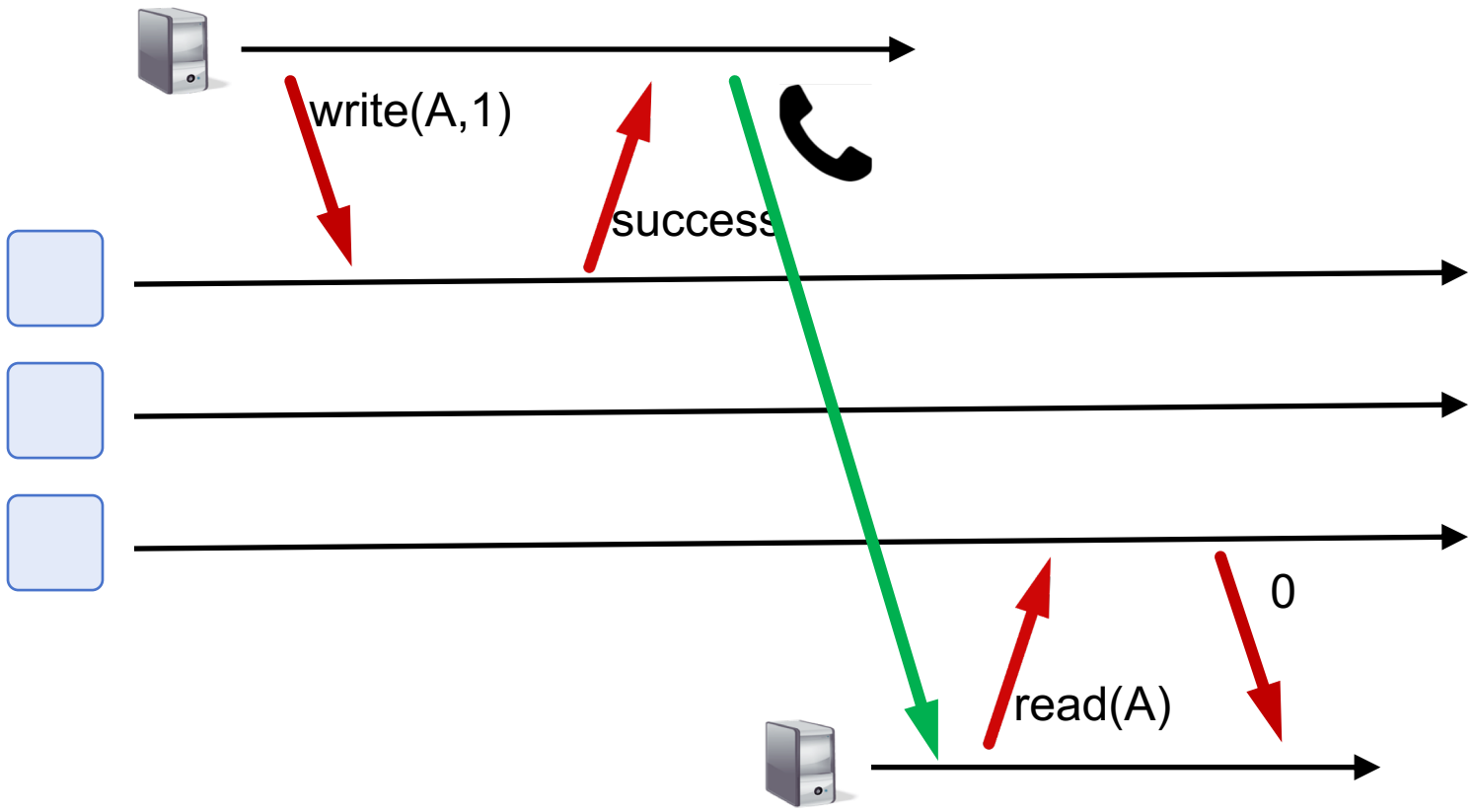

W(0)
W(0)
A:
W(1)
R(0)
B:
W(0)
R(1)
W(0)
A:
W(1)
R(0)
B:
W(0)
R(1)
W(0)
A:
W(1)
R(1)
B:

Weaker: Sequential consistency

- Sequential = Linearizability – real-time ordering
  1. All servers appear to execute all ops in *some* identical sequential order
  2. Global ordering preserves each client's own local ordering

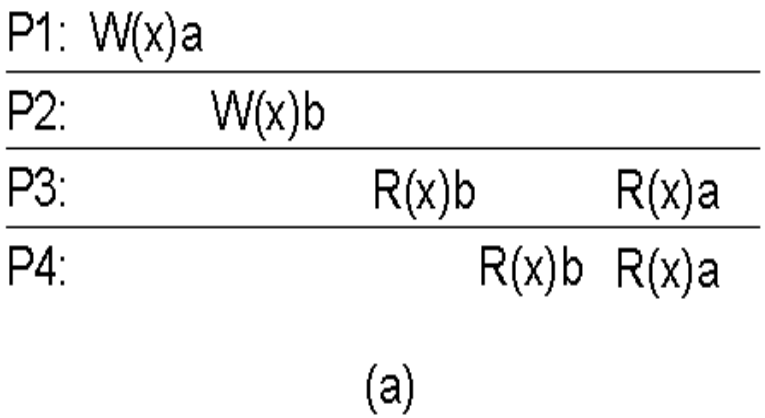
- With concurrent ops, "reordering" of ops (w.r.t. real-time ordering) acceptable, but all servers must see same order
  - e.g., linearizability cares about time
    sequential consistency cares only about program order

Sequential Consistency

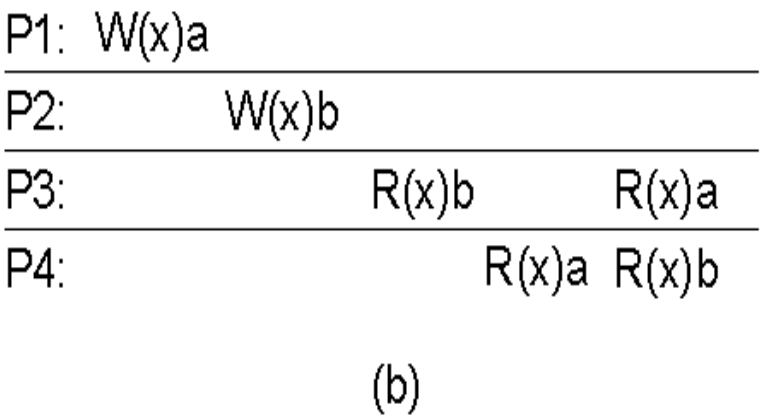


In example, system orders read(A) before write(A,1)

Valid Sequential Consistency?

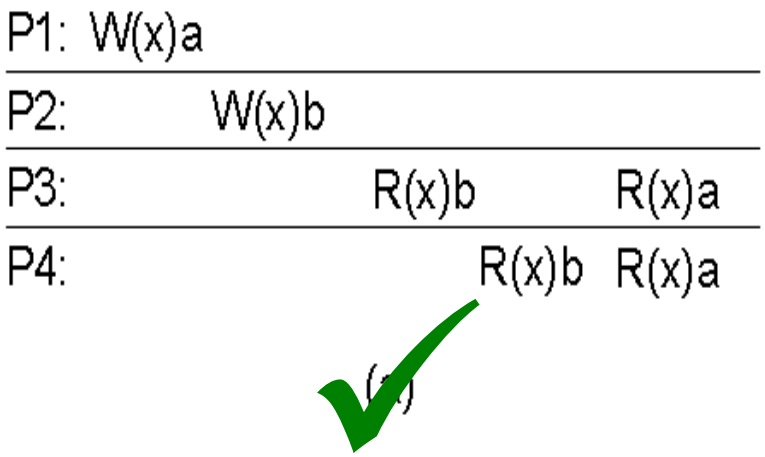


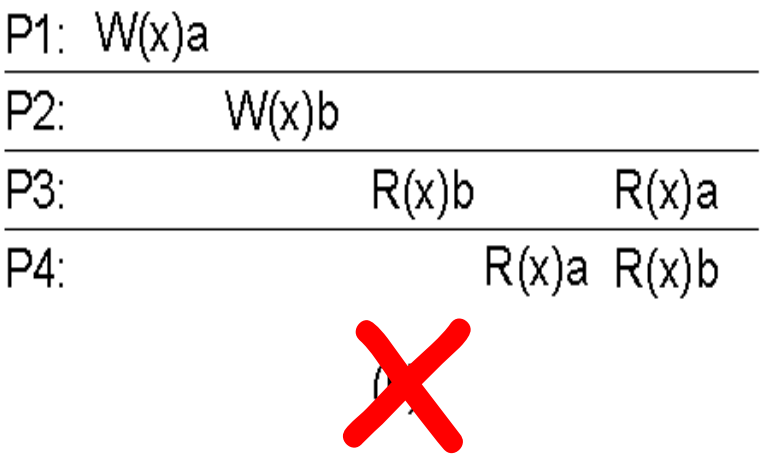
(a)



(b)

Valid Sequential Consistency?





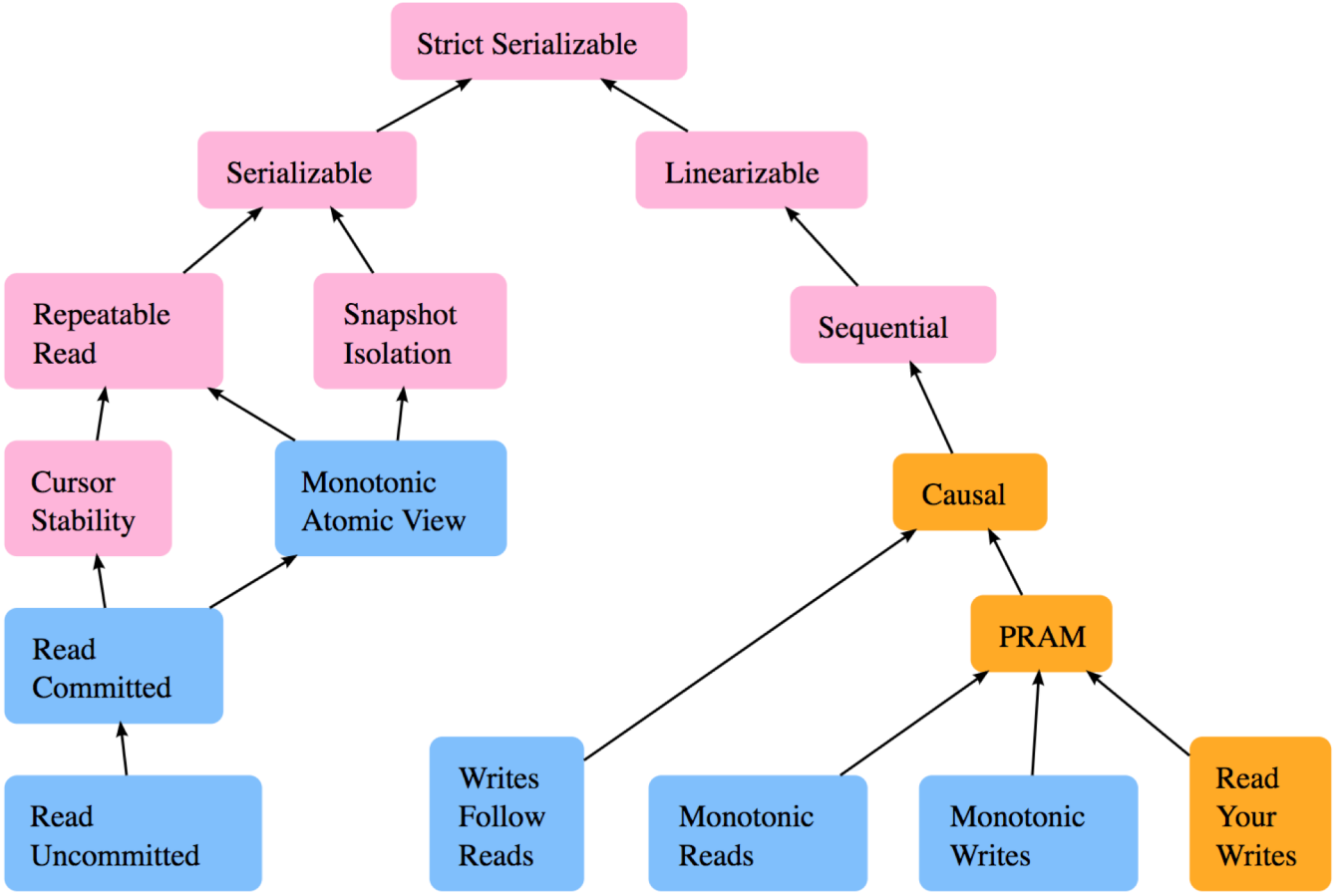
- Why? Because P3 and P4 don't agree on order of ops. Doesn't matter when events took place on diff machine, as long as proc's AGREE on order.
- What if P1 did both W(x)a and W(x)b?
  - Neither valid, as (a) doesn't preserve local ordering

Why Use SC?

- If processes only communicate through the store, then this is good enough
- Allows replicas to lag/cache and write back to replicas asynchronously
- Still: requires coordination to ensure total order of updates/visibility

Tradeoffs are fundamental?

Consistency Models

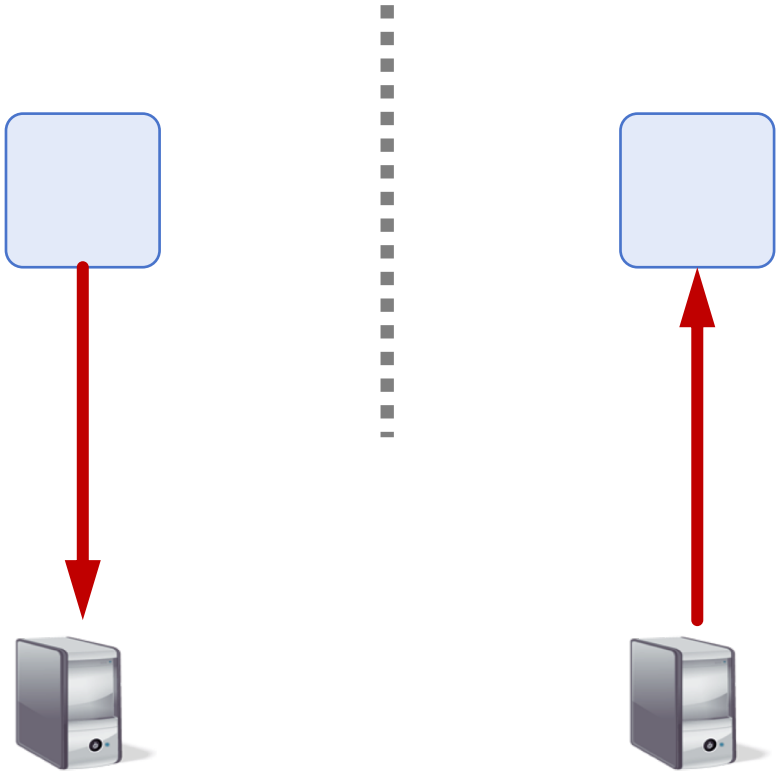


jepsen.io/consistency

"CAP" Conjecture for Distributed Systems

- From keynote lecture by Eric Brewer (2000)
  - History: Eric started Inktomi, early Internet search site based around "commodity" clusters of computers
  - Using CAP to justify "BASE" model: Basically Available, Soft-state services with Eventual consistency
- Popular interpretation: 2-out-of-3
  - Consistency (Linearizability)
  - Availability (Can reply to **all** requests without blocking)
  - Partition Tolerance: Arbitrary crash/network failures

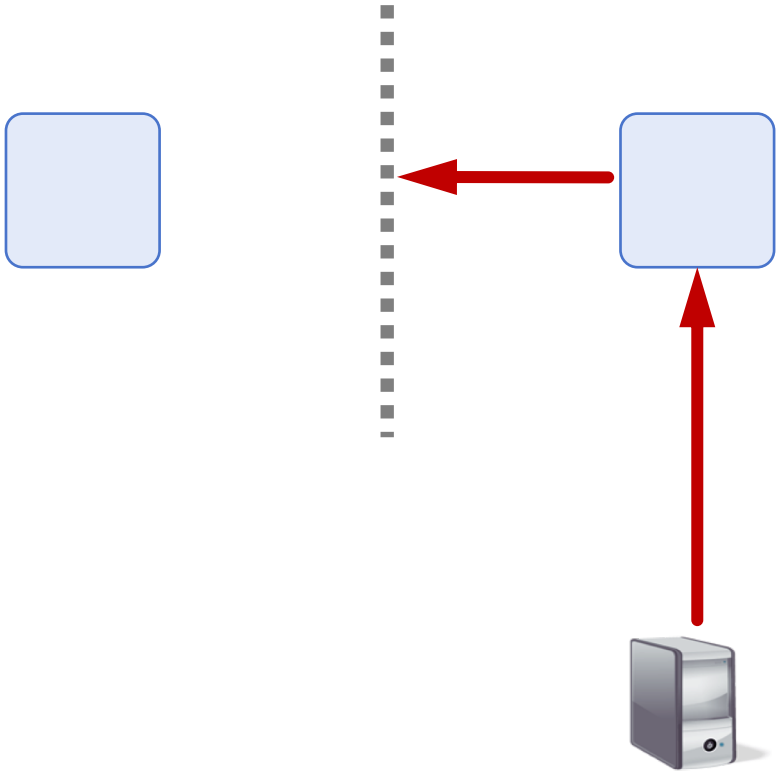
CAP Theorem: Proof



AP, but not consistent

Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.

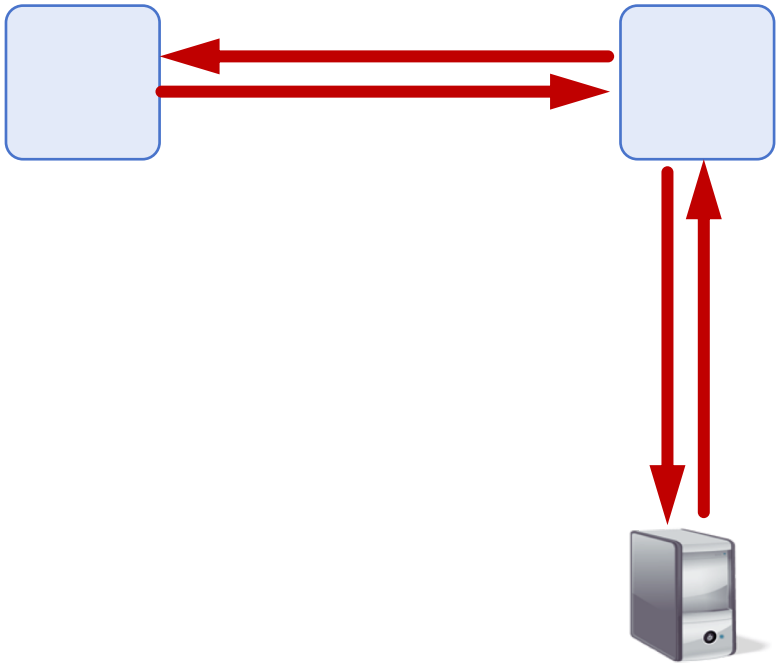
CAP Theorem: Proof



CP, but not available

Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.

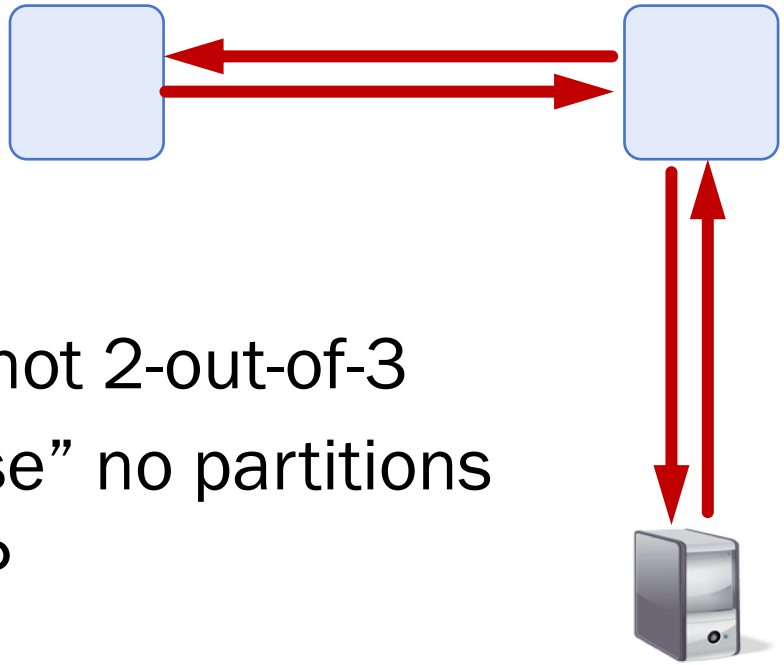
CAP Theorem: Proof



Not partition tolerant

Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.

CAP Theorem: AP or CP

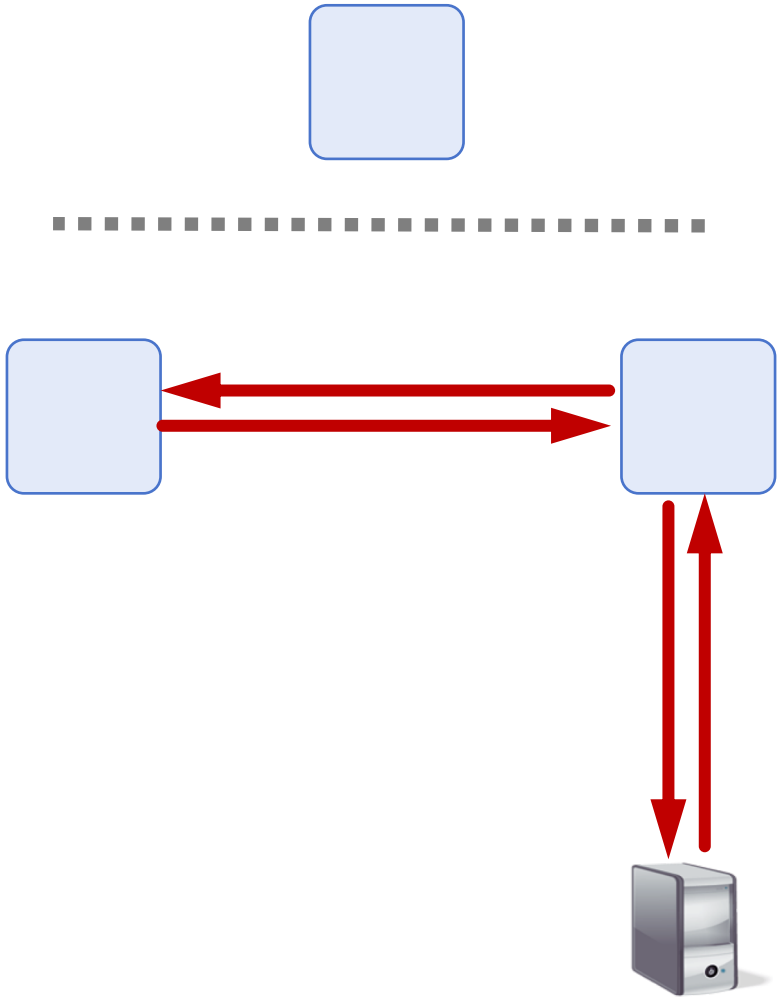


Criticism: It's not 2-out-of-3

- Can't "choose" no partitions
- So: AP or CP

CA not partition tolerant?

Raft?



Another Tradeoff: Low-latency vs. Consistency

- Low-latency: Speak to fewer than quorum of nodes?
  - 2PC: write N, read 1
  - Raft: write $\lfloor N/2 \rfloor + 1$, read $\lfloor N/2 \rfloor + 1$
  - General: $|W| + |R| > N$

- Low-latency and consistency are fundamentally at odds
  - "C" = linearizability, sequential, serializability

PACELC

- If there is a partition (P):
  - How does system tradeoff A(vailability) and C(onsistency)?
- Else (no partition)
  - How does system tradeoff L(atency) and C(consistency)?

- Is there a useful system that switches?
  - Dynamo: PA/EL
  - "ACID" dbs: PC/EC
  - PNUTS: PC/EL, SC for local reads, keep SC on partition
  - PA/EC: PA means some divergence, why force extra order in normal case if system is already built to handle it?

http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

Availability versus consistency

- NFS and 2PC all had single points of failure
  - Not available under failures
  - Raft/Paxos help, but unavailable sometimes

- Distributed consensus algorithms allow view-change to elect primary
  - Strong consistency model
  - Strong reachability requirements

If the network fails, can we provide any consistency when we replicate?

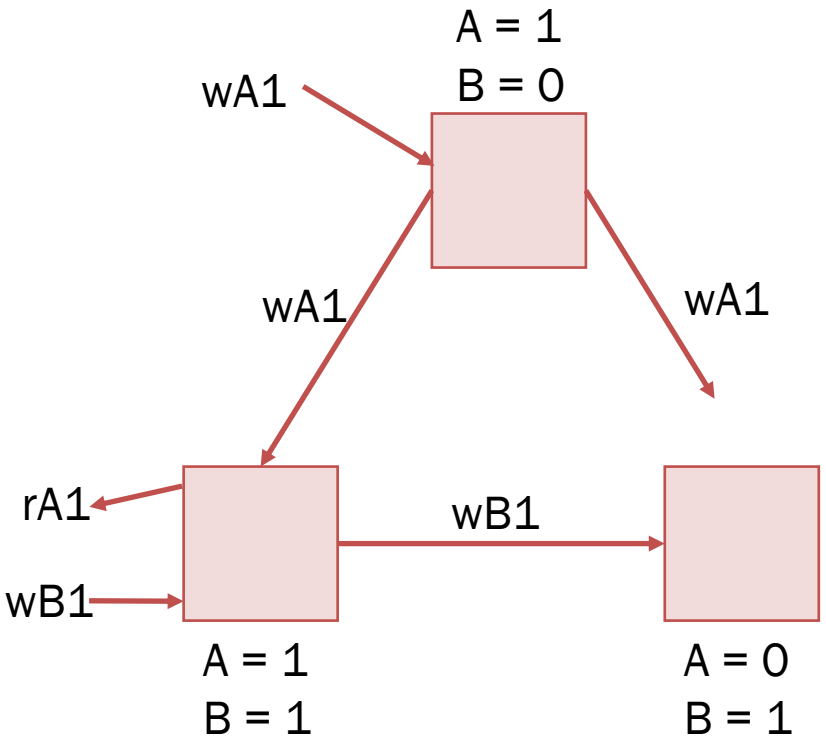
Eventual consistency

- *Eventual consistency:* If no new updates to the object, eventually all accesses will return the last updated value
- **Common:** git, iPhone sync, Dropbox, Amazon Dynamo

- Why do people like eventual consistency?
  - Fast read/write of **local** copy (no primary, no Paxos)
  - Disconnected operation

Issue: Conflicting writes to different copies
How do we reconcile them when discovered?

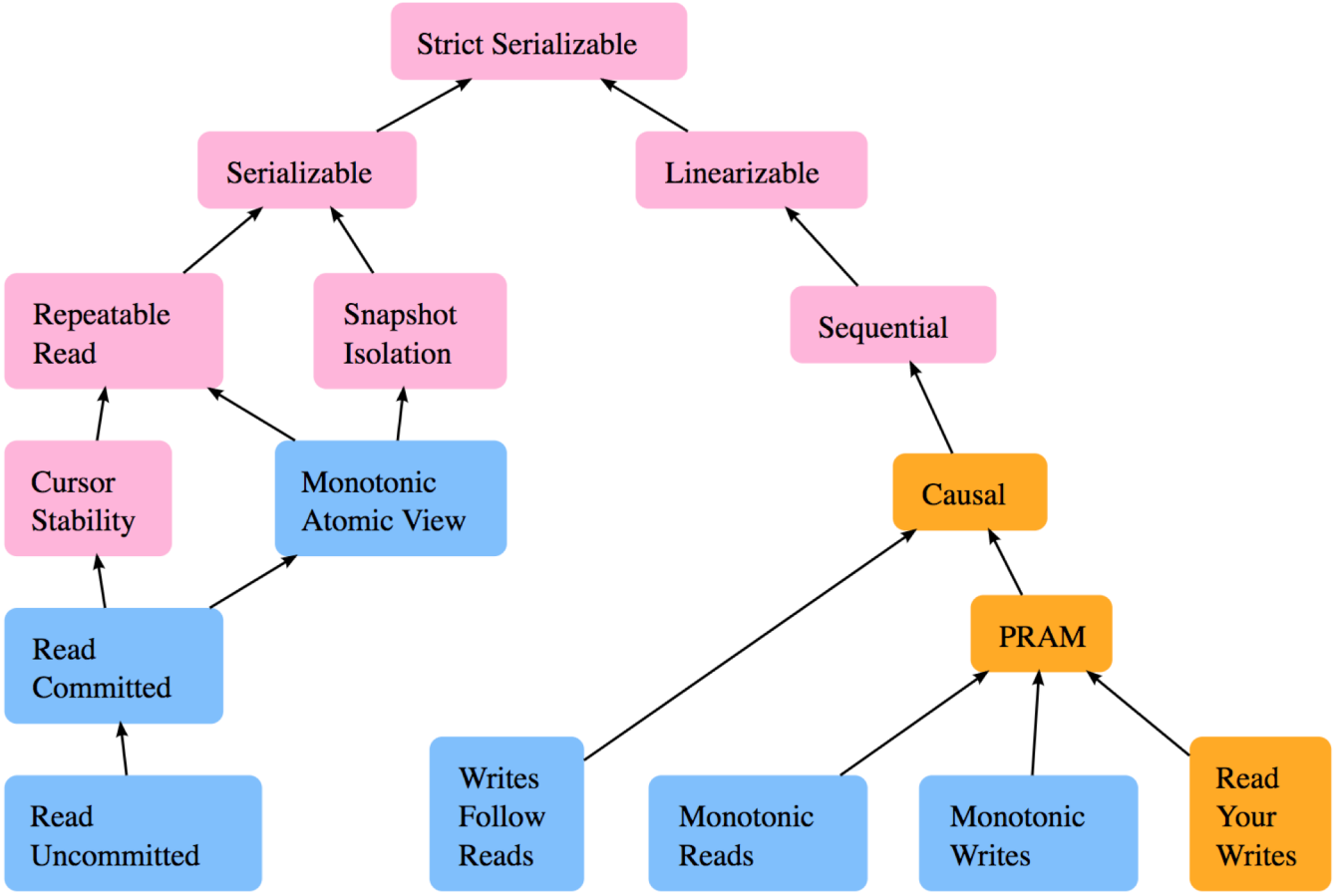
Convergence and Causality



Convergent?

Causally Consistent?

Consistency Models



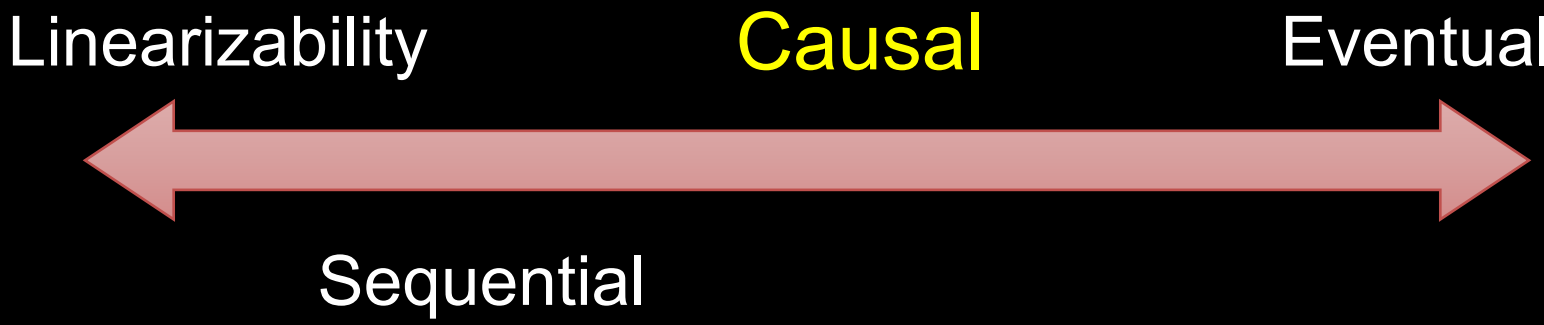
jepsen.io/consistency

Can we do better without total order?

- Eventual consistency doesn't say much about the ongoing states that clients can observe

- Is there a way to replicate
  - without total order (thus partition available)
  - without losing reasoning over what a client can observe?

Consistency models



Recall use of logical clocks

- Lamport clocks: C(a) < C(z) Conclusion: None
- Vector clocks: V(a) < V(z) Conclusion: a → … → z

- Distributed bulletin board application
  - Each post gets sent to all other users
  - Consistency goal: No user to see reply before the corresponding original message post
  - Conclusion: Deliver message only **after** all messages that **causally precede** it have been delivered

Causal Consistency

1. Writes that are ***potentially*** causally related must be seen by all machines in same order.
2. Concurrent writes may be seen in a different order on different machines.

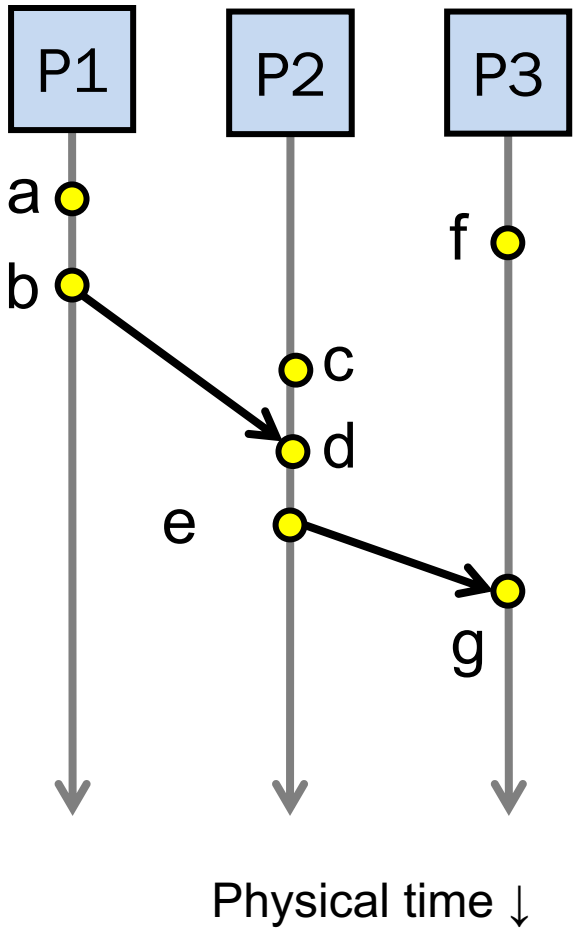
- Concurrent: Ops not causally related

Causal Consistency

1. Writes that are ***potentially*** causally related must be seen by all machines in same order.

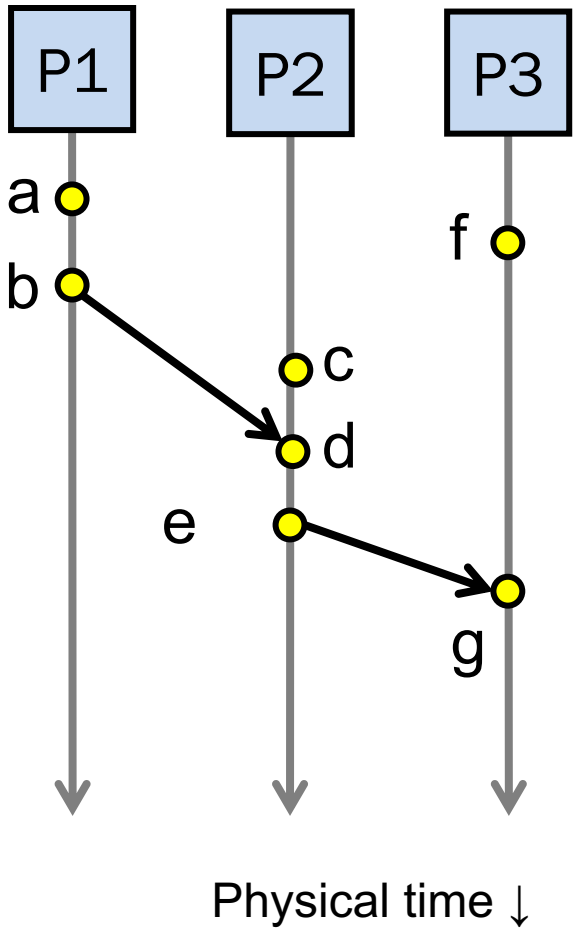
2. Concurrent writes may be seen in a different order on different machines.

- Concurrent: Ops not causally related



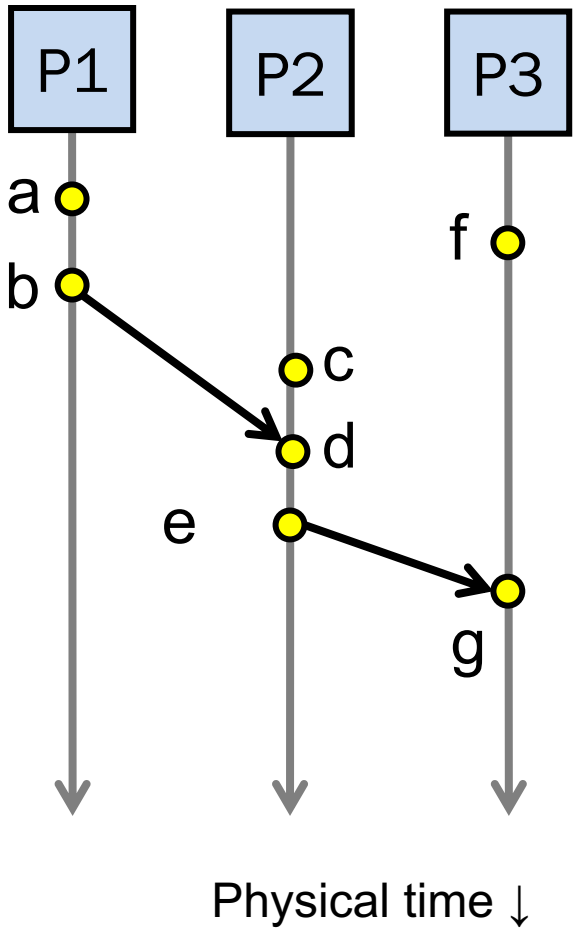
Causal Consistency

Operations	Concurrent?
a, b	
b, f	
c, f	
e, f	
e, g	
a, c	
a, e	



Causal Consistency

Operations	Concurrent?
a, b	N
b, f	Y
c, f	Y
e, f	Y
e, g	N
a, c	Y
a, e	N



Causal Consistency: Quiz

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

Causal Consistency: Quiz

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

- Valid under causal consistency
- **Why?** *W(x)b* and *W(x)c* are concurrent
  - So all processes don't (need to) see them in same order
- P3 and P4 read the values 'a' and 'b' in order as potentially causally related. No 'causality' for 'c'.

Sequential Consistency: Quiz

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

Sequential Consistency: Quiz

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

- Invalid under sequential consistency
- **Why?** P3 and P4 see b and c in different order
- But fine for causal consistency
  - B and C are not causually dependent
  - Write after write has no dep's, write after read does

Causal Consistency

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)

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

(b)

Causal Consistency

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)

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

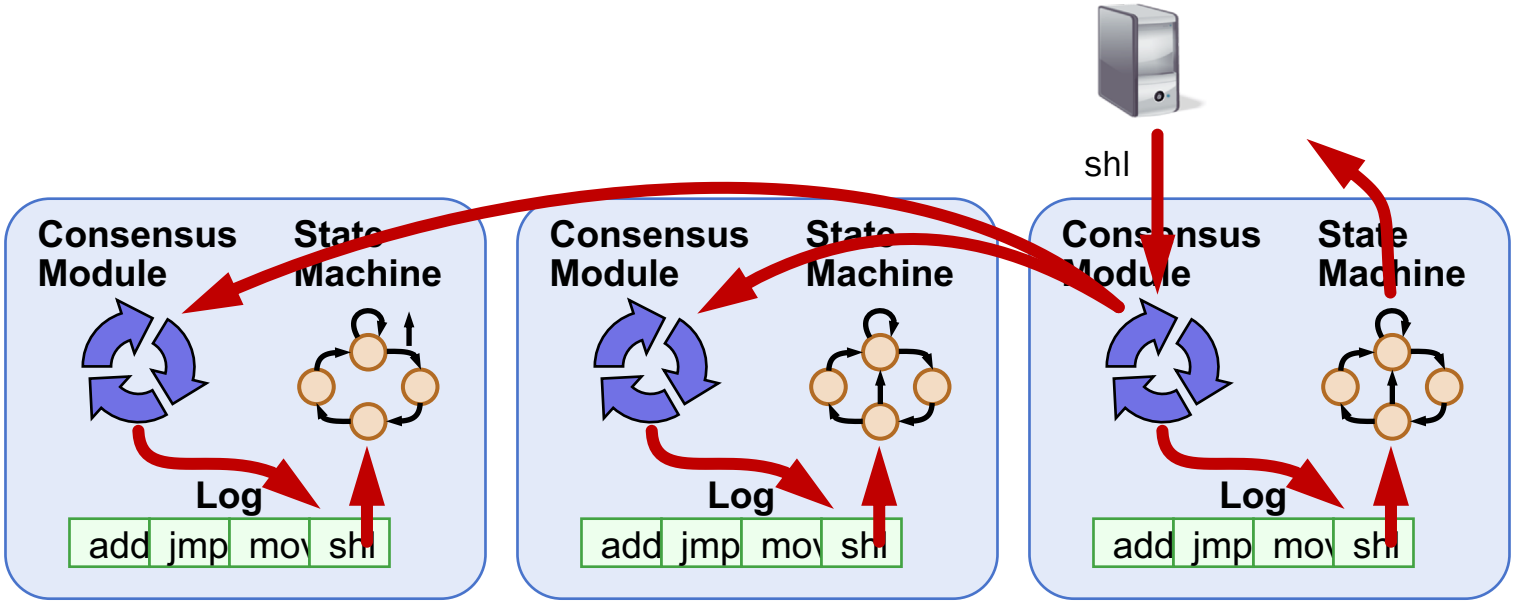
(b)

A: Violation: W(x)b is potentially dep on W(x)a

B: Correct. P2 doesn't read value of a before W

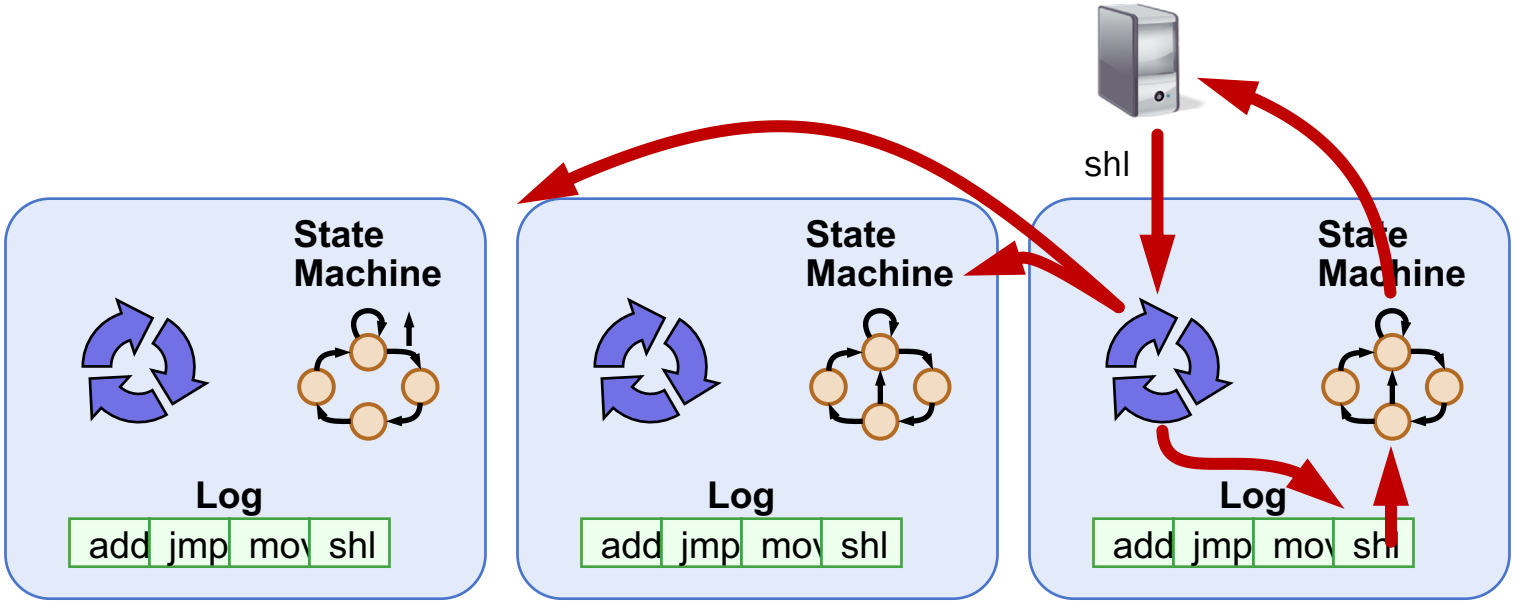
Causal consistency within replication systems

Implications of laziness on consistency



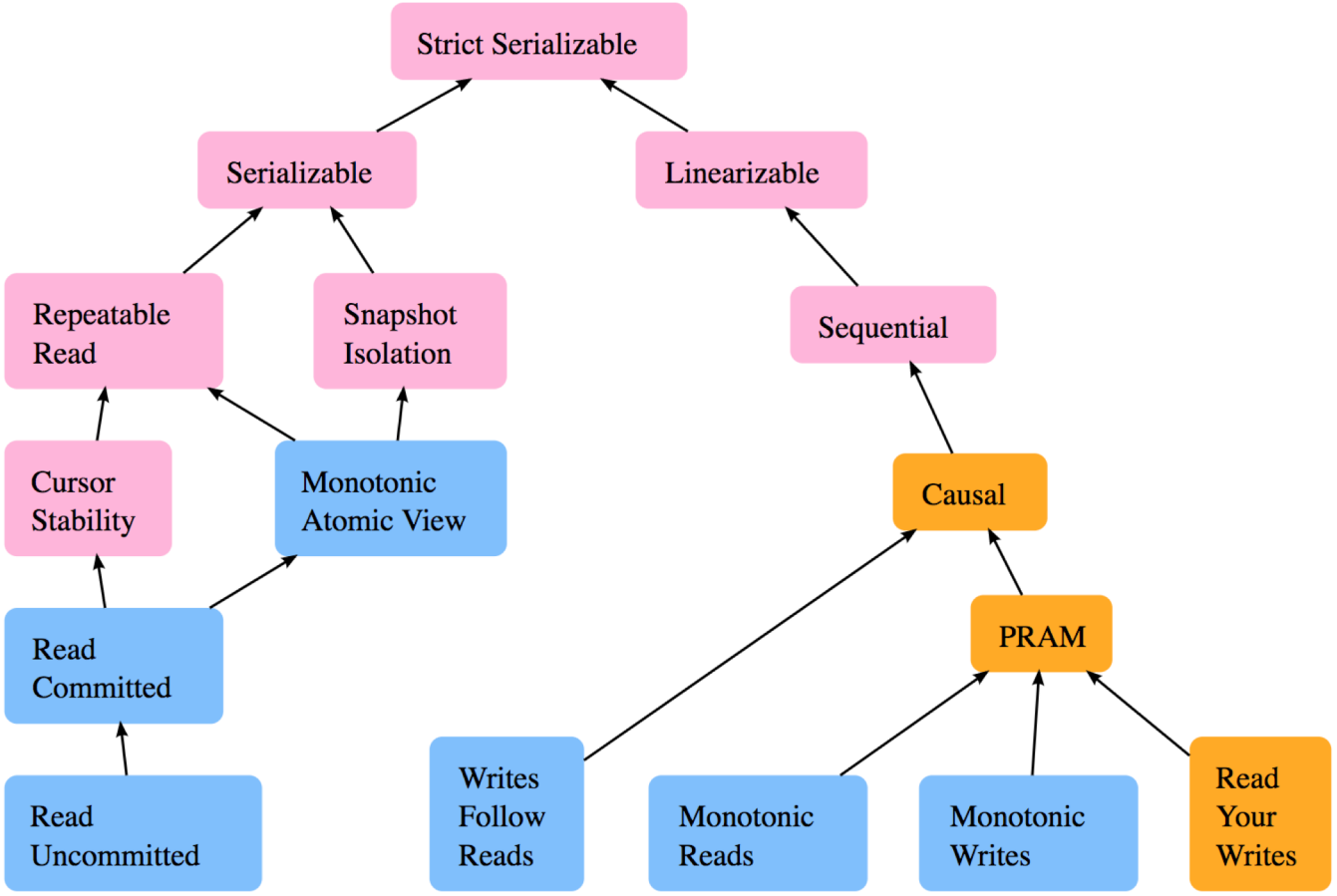
- Linearizability / sequential: Eager replication
- Trades off low-latency for consistency

Implications of laziness on consistency



- Causal consistency: Lazy replication
- Trades off consistency for low-latency
- Maintain local ordering when replicating
- Operations may be lost if failure before replication

Consistency Models



jepsen.io/consistency