CS 582: Distributed Systems

Consistency Models

Dr. Zafar Ayyub Qazi

Fall 2024

Acknowledgments: Slides use some material from John Ousterhout (Stanford University) and Ryan Stutsman (University of Utah)

Agenda

- Wrap-up elections in Raft
- Consistency Models
  - Linearizability
  - Sequential Consistency
  - Causal Consistency
  - Eventual Consistency
- CAP Theorem

Specific learning outcomes

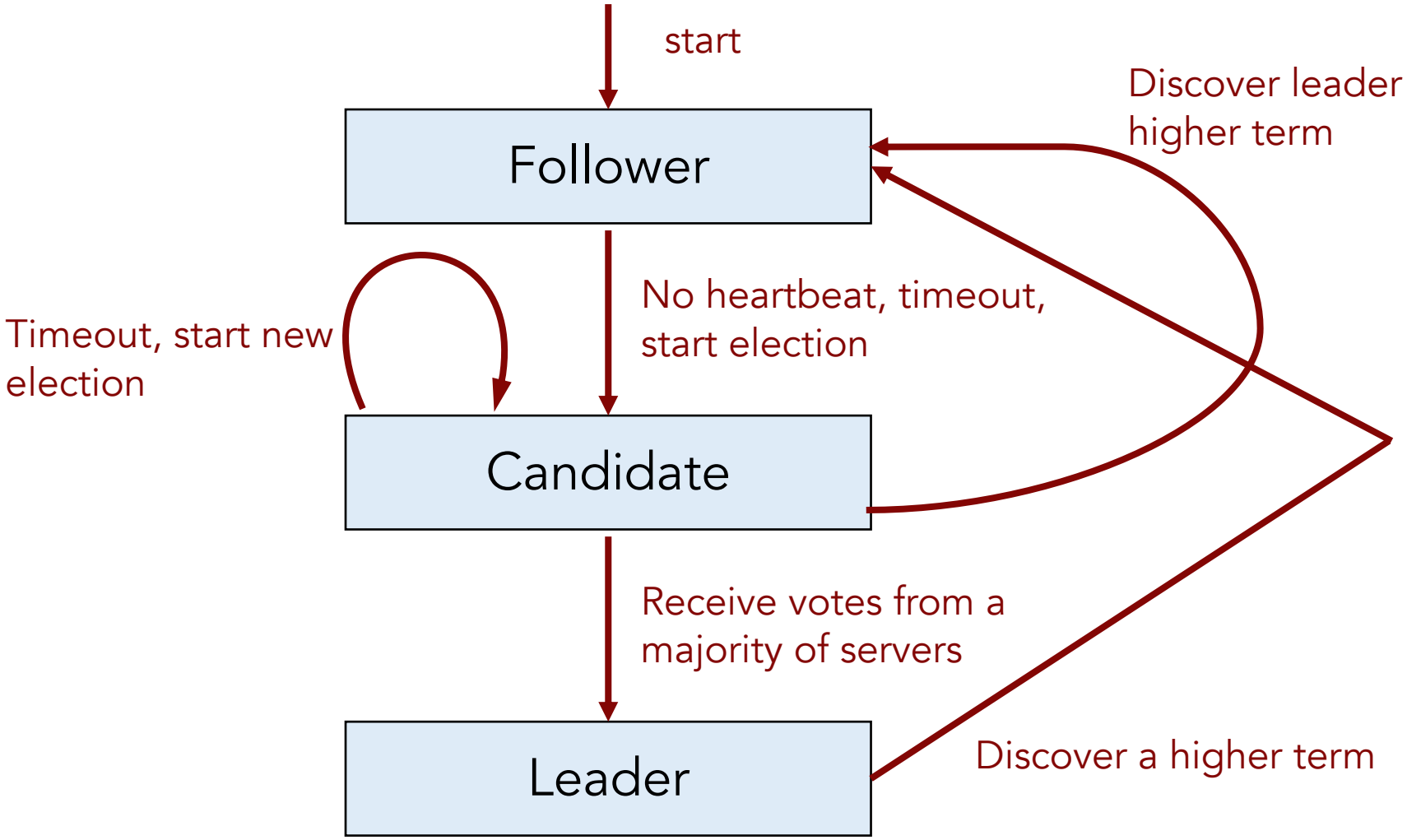
By the end of today's lecture, you should be able to:

- ❑ Explain how to set election timeouts in Raft to ensure that split votes are rare and they are resolved quickly
- ❑ Define and explain linearizability, sequential consistency, causal consistency, and eventual consistency
- ❑ Compare and contrast linearizability, sequential consistency, causal consistency, and eventual consistency, in terms of their guarantees and tradeoffs
- ❑ Given a scenario of event orderings in a distributed system, determine whether the system exhibits linearizability, sequential consistency, causal consistency, or eventual consistency
- ❑ Analyze the implications of different consistency models on the design and performance of distributed systems
- ❑ Explain the CAP theorem and its fundamental tradeoffs in distributed systems
- ❑ Evaluate the applicability of the CAP theorem to various systems distributed system designs and use case

Recap: Election Basics in Raft

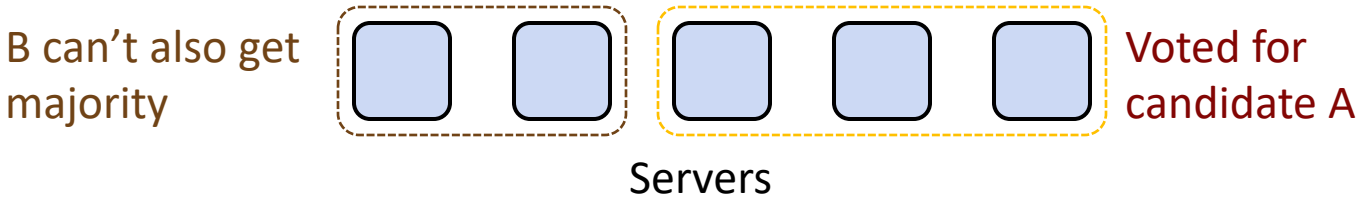
- If a server suspects a leader has failed, increment current term
- Change to Candidate state
- Vote for self
- Send RequestVote RPCs to all other servers, retry until either:
  1. Receive votes from majority of servers:
     - Become leader
     - Send AppendEntries heartbeats to all other servers
  2. Receive RPC from a valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election

Recap: Server State Transitions



Election Correctness

- Safety: allow <u>at most one winner per term</u>
  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can't accumulate majorities in same term



- Liveness: some candidate must eventually win
  - Choose election timeouts randomly in [T, 2T](e.g., 150-300 ms)
  - One server usually times out and wins election before others wake up
  - Works well if T >> broadcast time
  - Broadcast time << T << Time between failures for a single server

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

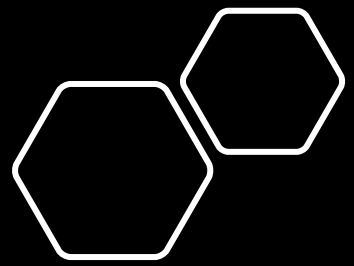
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).



Summary: Leader Election in Raft

- Three possible server states: follower, candidate, leader
- All servers start as followers
- If a follower detects a leader has failed
  - Follower can start new election
  - Increments term, and transiton to Candidate state
  - Candidate becomes a leader if a majority of processes vote for it
  - Possible no one wins election: split votes

For more on leader elections in Raft …

- Attend the next tutorial
- Check out Raft's extended paper (will be shared with the assignment 2 handout)
- Checkout the following excellent visualization of elections in Raft:
  - http://thesecretlivesofdata.com/raft/

Recap: The course so far …

- System Models
- Failure Detectors
- Remote Procedure Calls
- Time Synchronization
- Logical Time
- Leader Elections in Raft

Replication & Consistency

Replication

- Replication: creating multiple copies of data
  - Each copy is called a replica
- Replication is important in distributed systems
  - Fault tolerance
  - Performance
- However, it also introduces challenges

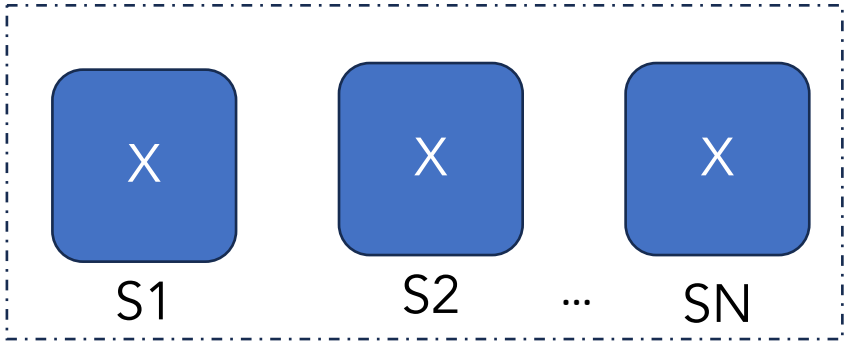
Nature of Replicated Data

- Read only data
  - Easy to replicate, we just make multiple copies
  - And we can get the performance and fault tolerance benefits
- Read-write data
  - Writes can result in different replicas
  - And we may have multiple nodes concurrently accessing replicated data
  - We need to keep replicas consistent
  - But how do we define "consistency"?

Consistency Model

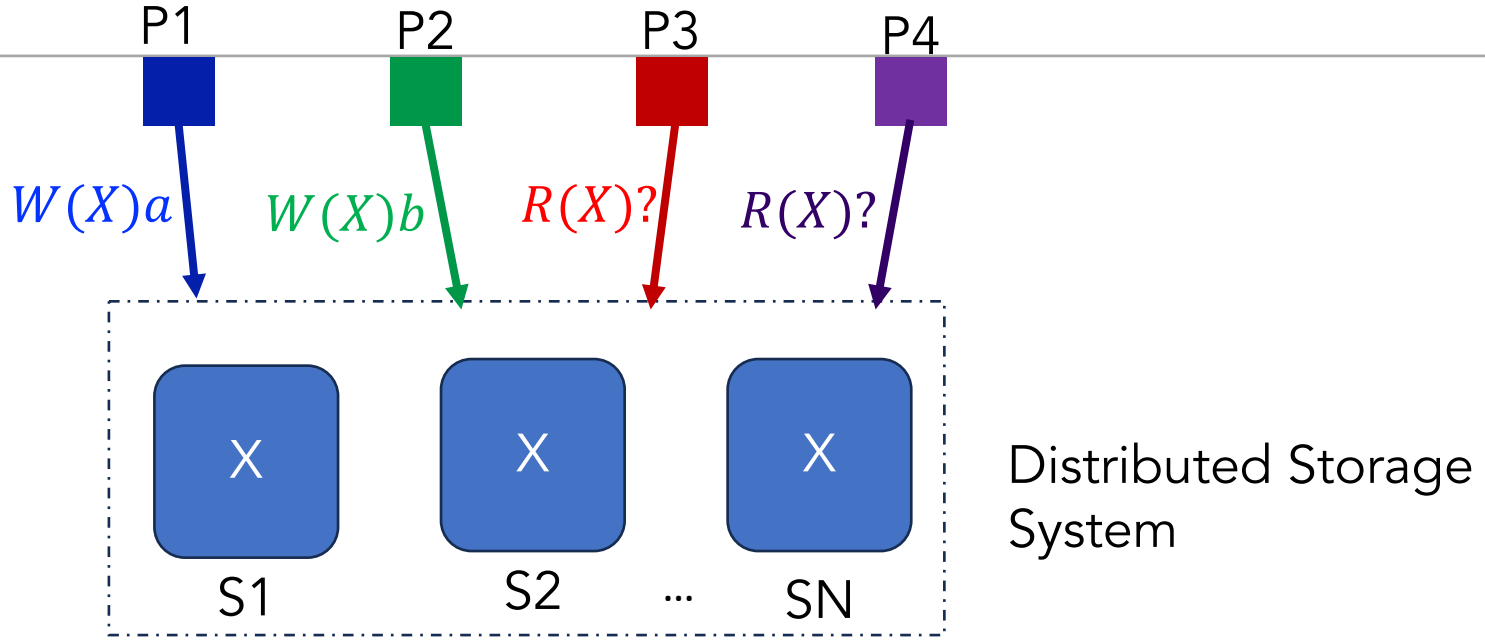
- What is consistency?
  - What processes can expect when read-write replicated data can be accessed concurrently
- What is a consistency model?
  - Contract btw a distributed system and applications that run on it
  - A consistency model is a set of guarantees made by the dist. system
  - E.g.: “If a process reads a certain piece of data, the distributed storage system pledges to return the value of the last write”
- Examples of consistency models: Linearizability, Serializability, Causal Consistency, Eventual Consistency

Example

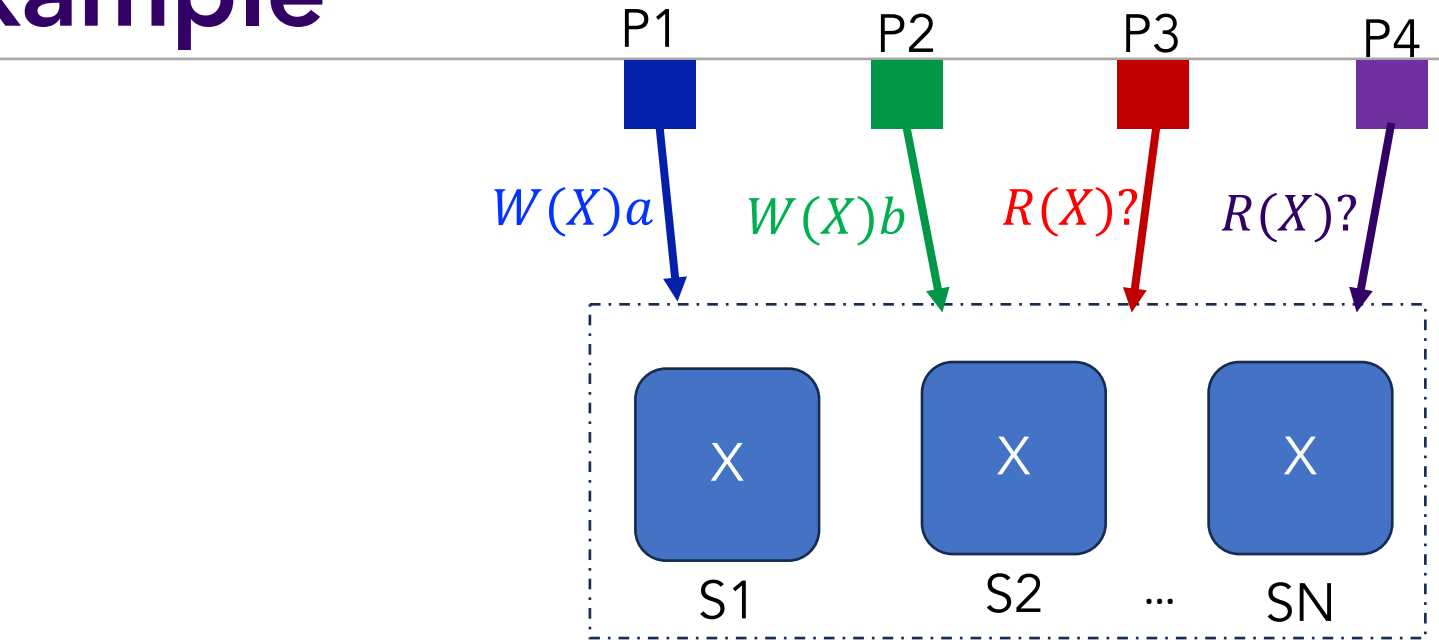


Distributed Storage System

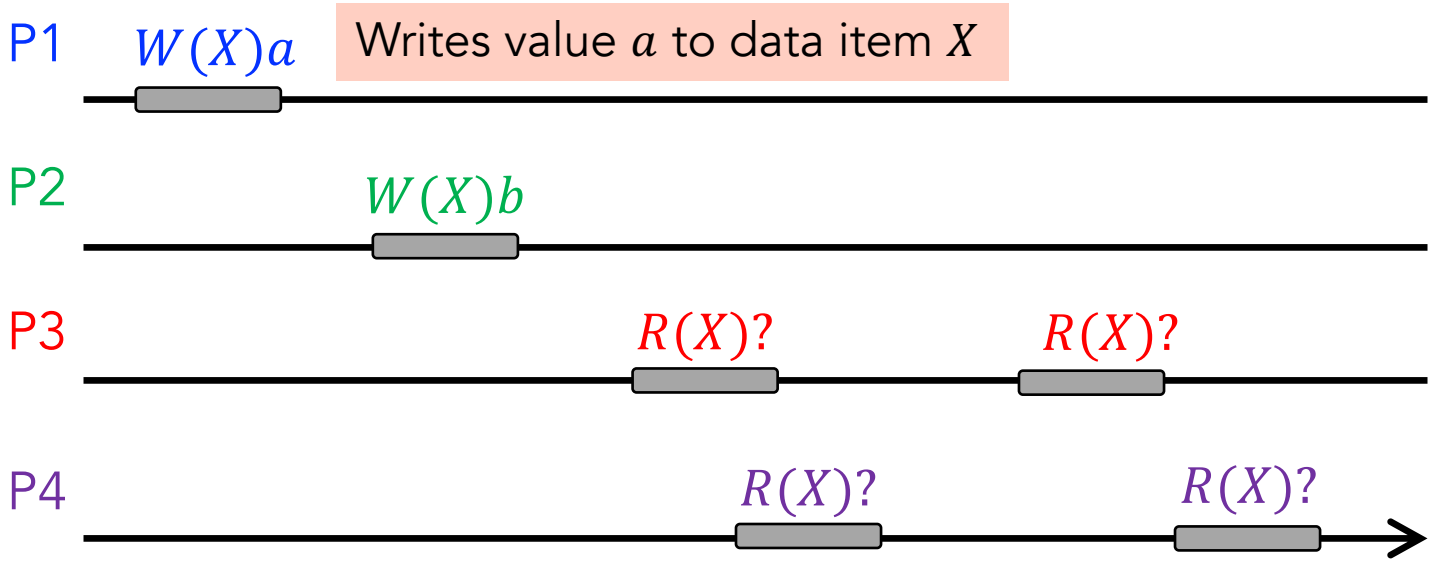
Example


P1
P2
P3
P4
$W(X)a$
$W(X)b$
$R(X)?$
$R(X)?$
X
X
X
S1
S2
...
SN
Distributed Storage System

Example



Distributed Storage System



global physical time

Stronger vs. Weak Consistency

- Stronger consistency models
  - \+ Easier to write an application
  - \- More guarantees for the system to ensure
    Results in performance tradeoffs

- Weaker consistency models
  - \- Harder to write applications
  - \+ Fewer guarantees for the system to ensure
    Generally, results in better performance

Which consistency models are used where?

- Linearizability
  - Banking applications
  - Google's Spanner uses sth. similar (they call it "external consistency")
    - Used for Google's Advertisement system and some Cloud applications

- Sequential consistency
  - A number of both academic and industrial systems provide (at least) sequential consistency
  - Examples: Microsoft's Niobe DFS, Cornell's chain replication, …

Which consistency models are used where?

- Causal consistency
  - Bayou supports a variation of causal consistency
  - MongoDB, Antidote DB provide causal consistency

- Eventual consistency
  - Very popular both in industry and in academia
  - Examples: Amazon's Dynamo, git, IPhone sync, Dropbox, Calendar apps

Linearizability [Herlihy and Wing 1990]

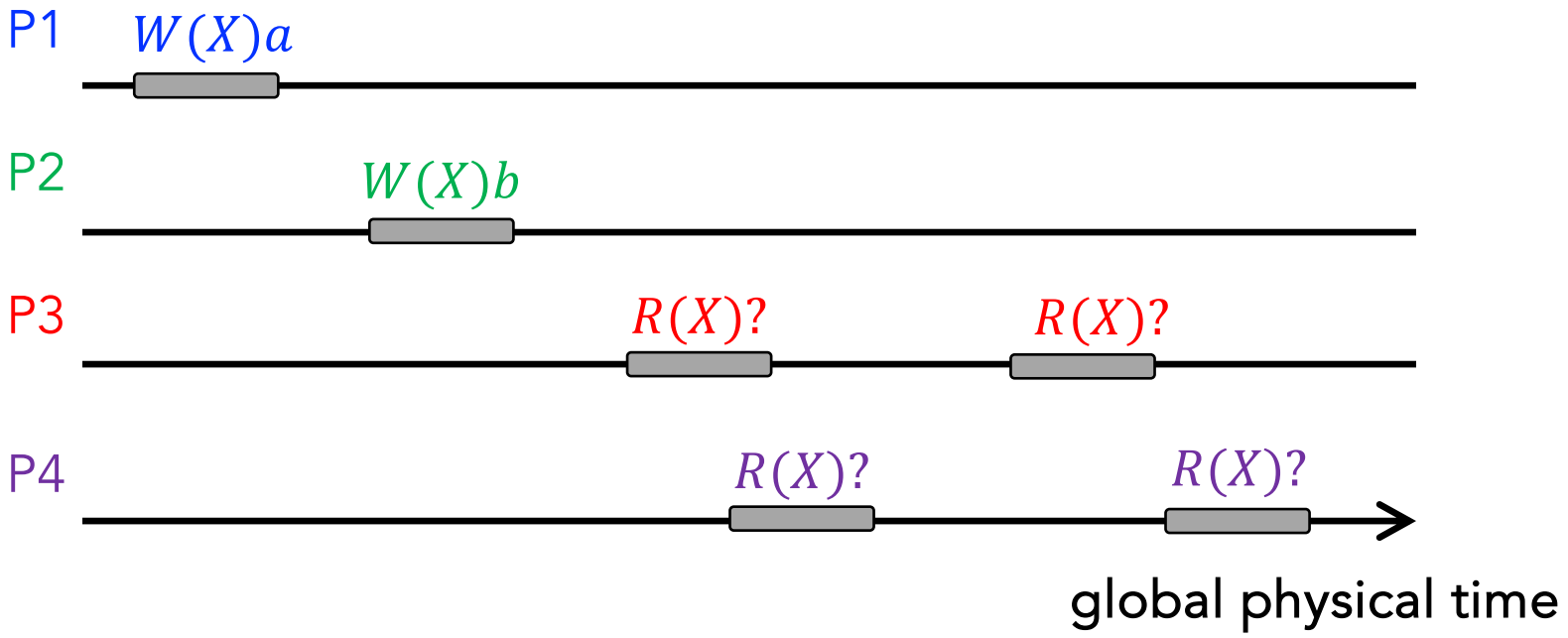
Linearizability == "Appears to be a Single Machine"

- All operations behave as if executed on a single copy of data (even if there are in fact multiple replicas)
- Consequence: every operation returns an "up-to-date" value
- Often referred to as strong consistency

Recap: Linearizability

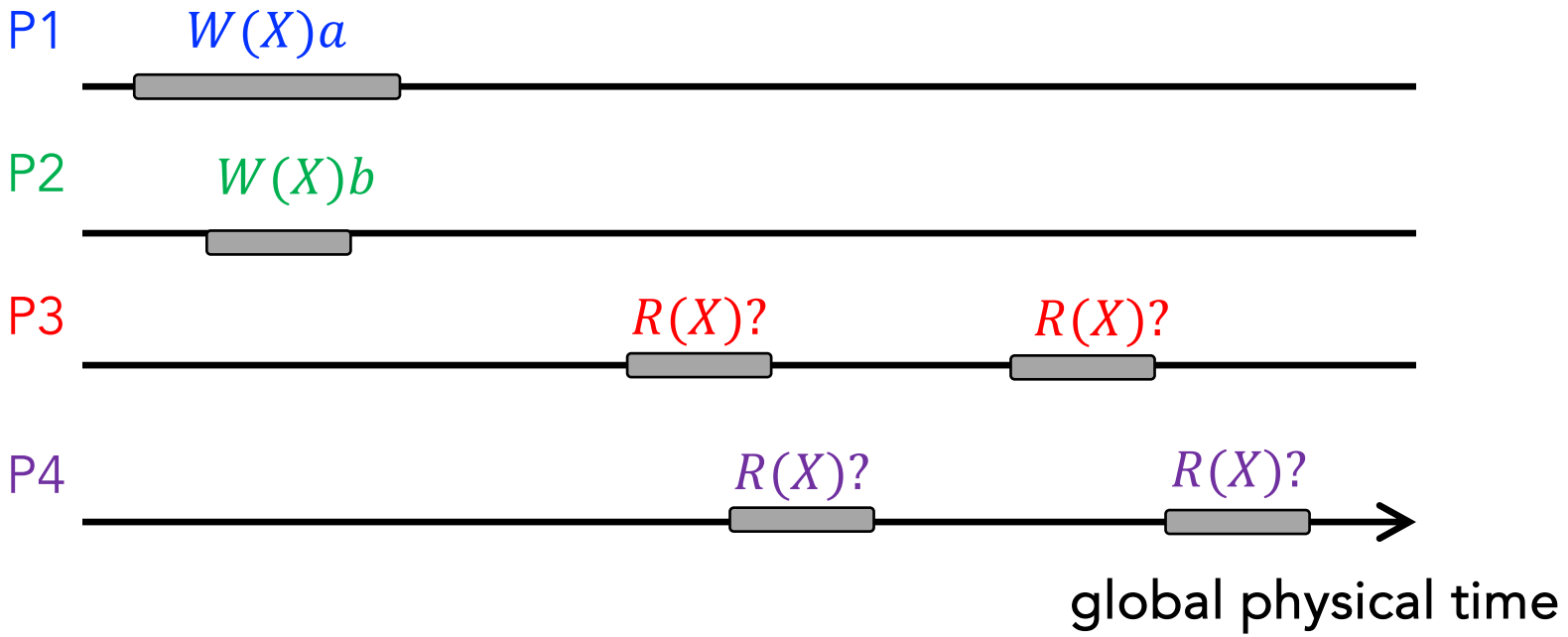
- All replicas execute operations in some total order
- That total order preserves the real-time (physical-time) ordering between operations
  - If operation A completes before operation B begins, then A is ordered before B in real-time
  - If neither A nor B completes before the other begins, then there is no real-time order
    - (But there must be *some* total order)

Linearizability: (Counter) Examples



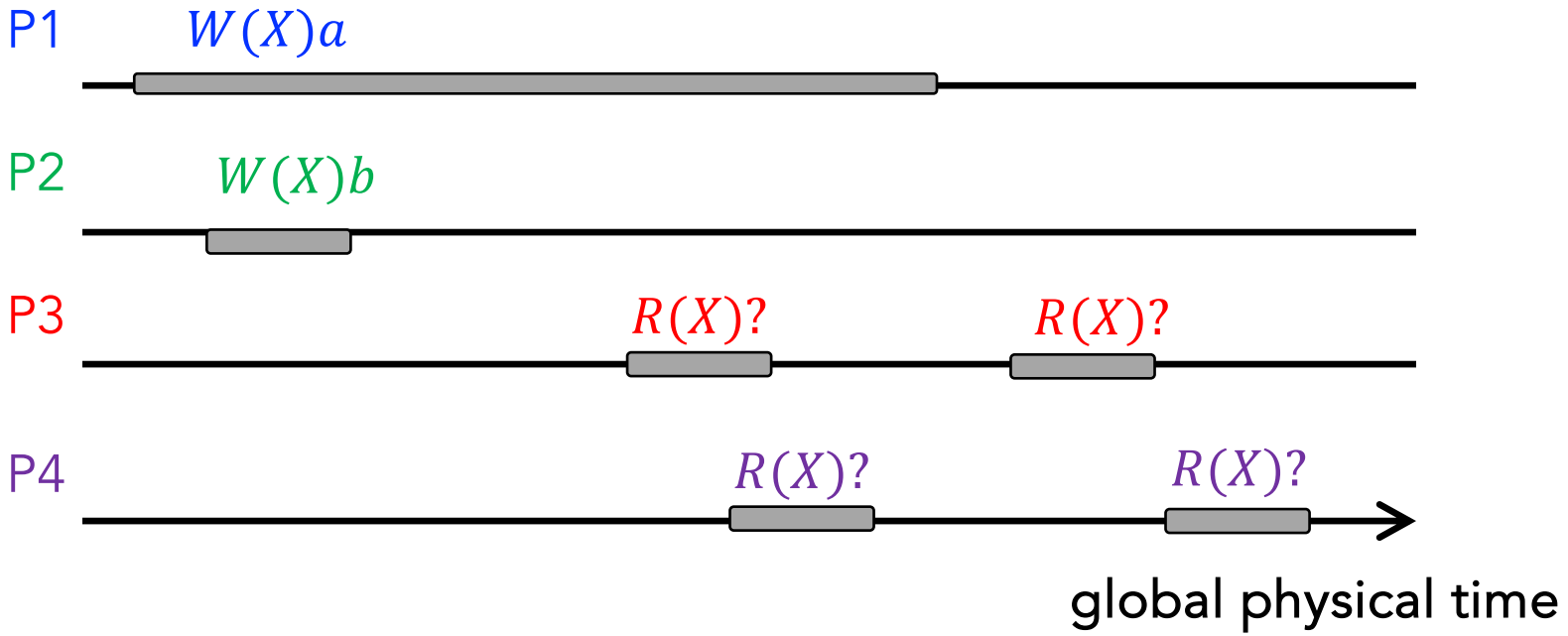
If the storage system is linearizable, what can these reads return?

Linearizability: (Counter) Examples



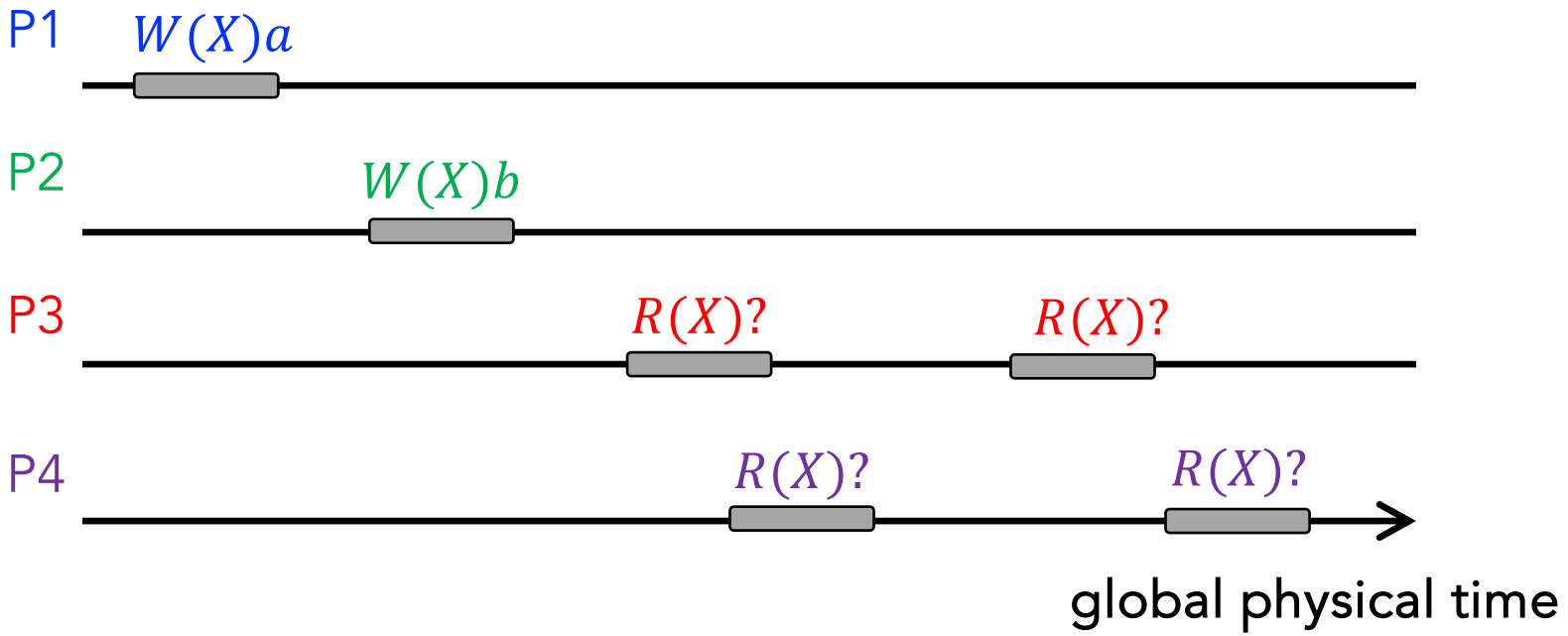
If underlying storage system is linearizable, what can these reads return?

Linearizability: (Counter) Examples



If underlying storage system is linearizable, what can these reads return?

Linearizability: (Counter) Examples



If the storage system is linearizable, what can these reads return?

Linearizability: Implications

- Once a write completes, all later reads (by physical time) should return the value of that write or the value of a later write
- Once a read returns a particular value, all later reads should return that value or the value of a later write

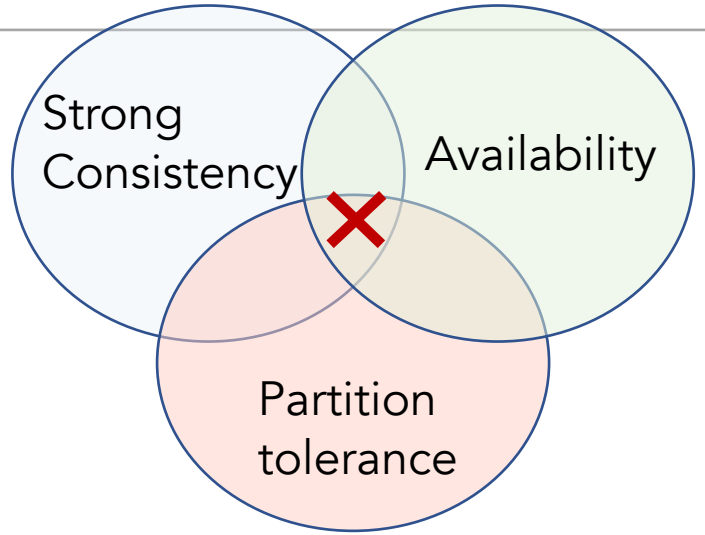
Linearizability Tradeoffs

- Hides the complexity of the underlying distributed system from applications!
  - Easier to write applications
  - Easier to write correct applications

- But, performance trade-offs

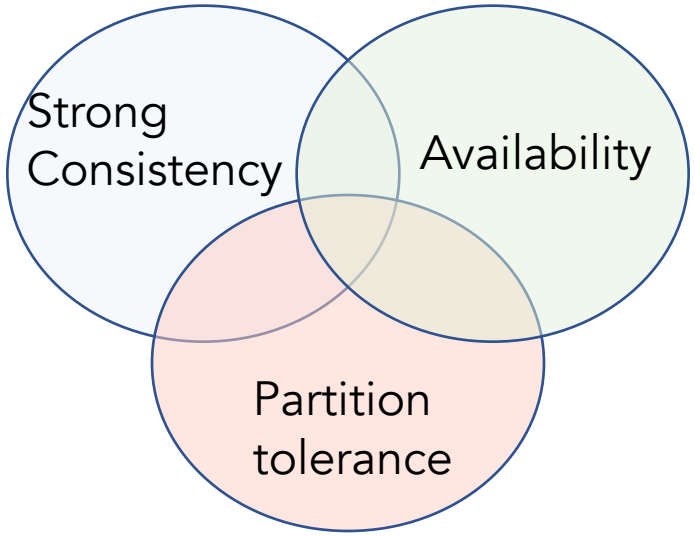
CAP Theorem

- We cannot achieve all three of:
  1. Strong Consistency
  2. Availability
  3. Partition-Tolerance



- Partition Tolerance => Network Partitions Can Happen
- Availability => All Sides of Partition Continue
- Strong Consistency => Replicas Act Like Single Machine
  - Specifically, Linearizability

Visualizing CAP



Three possible systems

1. CA (strong consistency + availability)
2. CP (strong consistency + partition tolerance)
3. AP (availability + partition tolerance)

Four potential conclusions from CAP Theoreom

Conclusion #1

- Many system designs used in early distributed relational database systems did not take into account partition tolerance (e.g., they were CA designs)

- Partition tolerance is an important property for modern systems, since network partitions become much more likely if the system is geographically distributed (as many large systems are)

Conclusion #2

- There is a tension between strong consistency and high availability during network partitions

- The CAP theorem is an illustration of the tradeoffs that occur between strong consistency guarantees and distributed computation

Conclusion #3

- There is a *tension between strong consistency and performance in normal operation*. Strong consistency/linearizability requires that nodes communicate and agree on every operation. This results in high latency during normal operation

Conclusion #4

- Somewhat indirect - that *if we do not want to give up availability during a network partition*, *then we need to explore whether consistency models other than strong consistency are workable for our purposes*

Next …

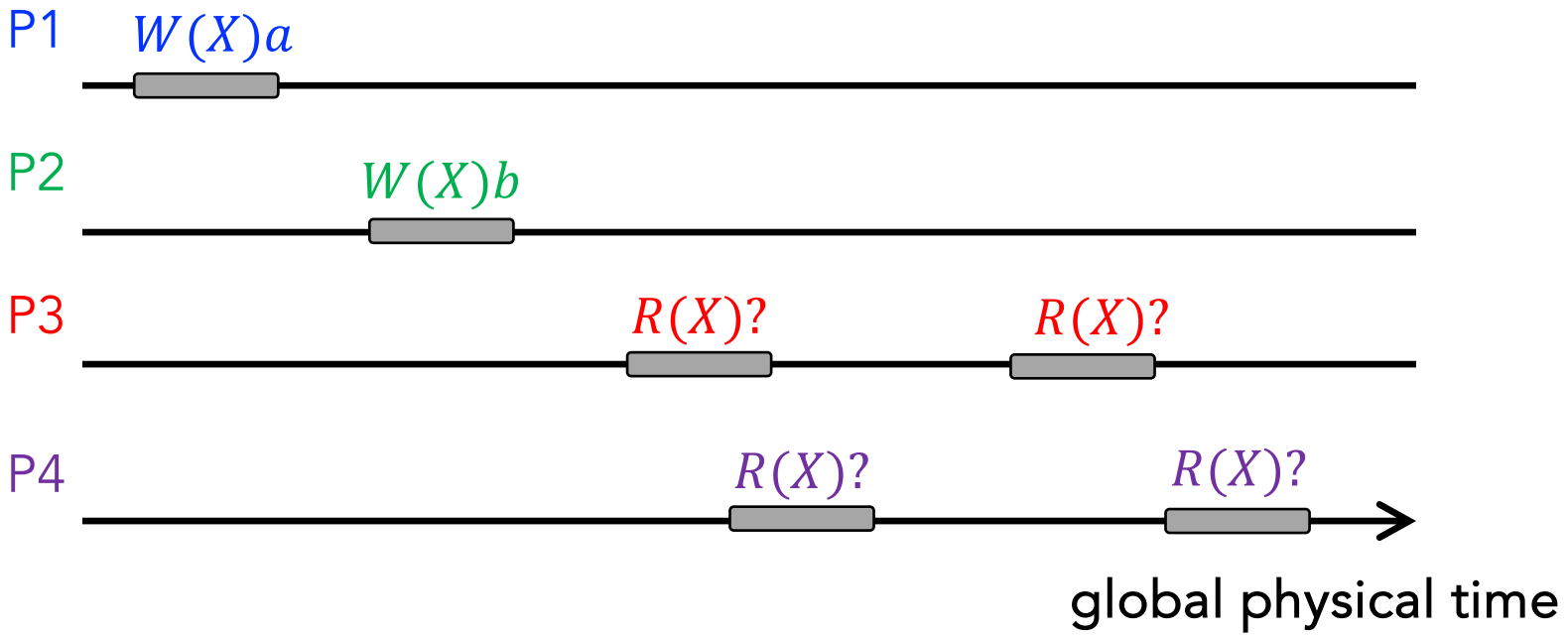
- Other consistency models

Sequential Consistency [Lamport 1979]

Sequential Consistency [Lamport 1979]

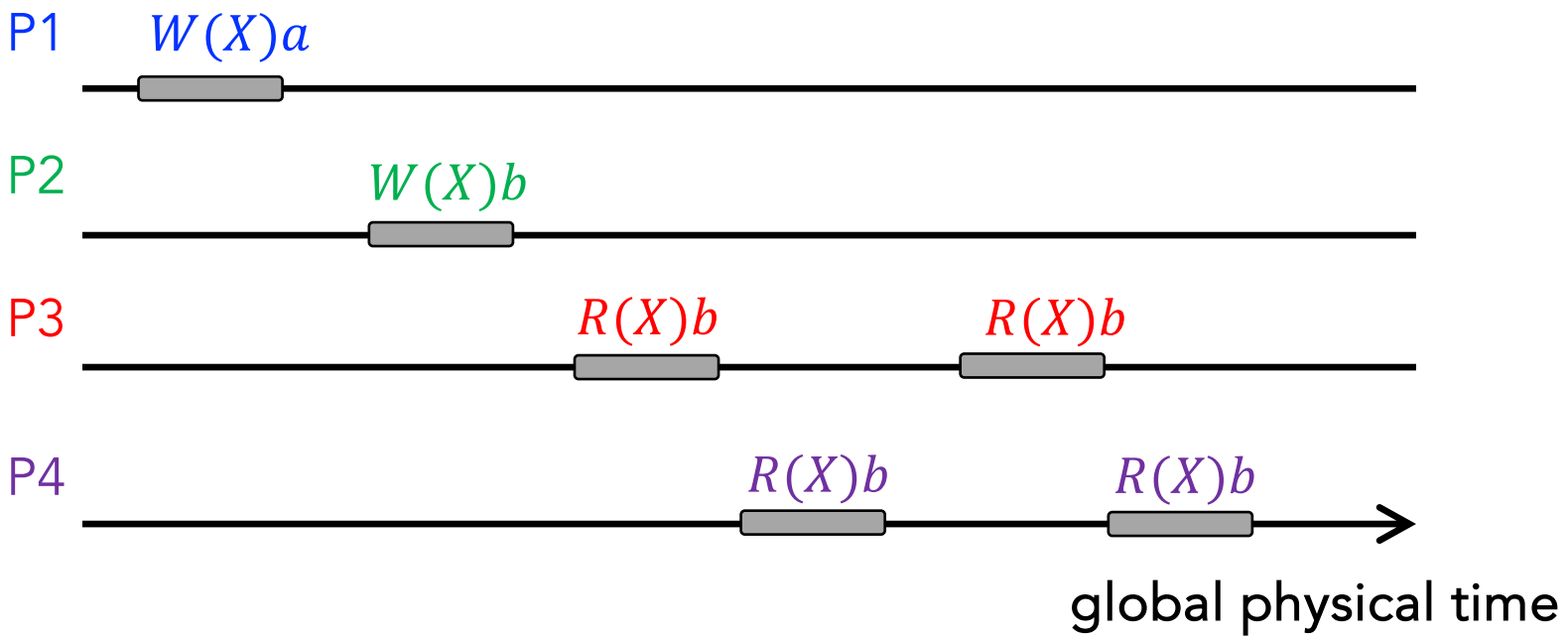
- Implies that operations appear to take place in (1) some total order, and that order is (2) consistent with the order of operations on each individual client process
- Therefore:
  - Reads may be stale in terms of real time, but not in logical time
  - Writes are totally ordered according to logical time across all replicas
- Key difference from linearizability
  - May not preserve real time ordering

Sequential Consistency: (Counter) Examples



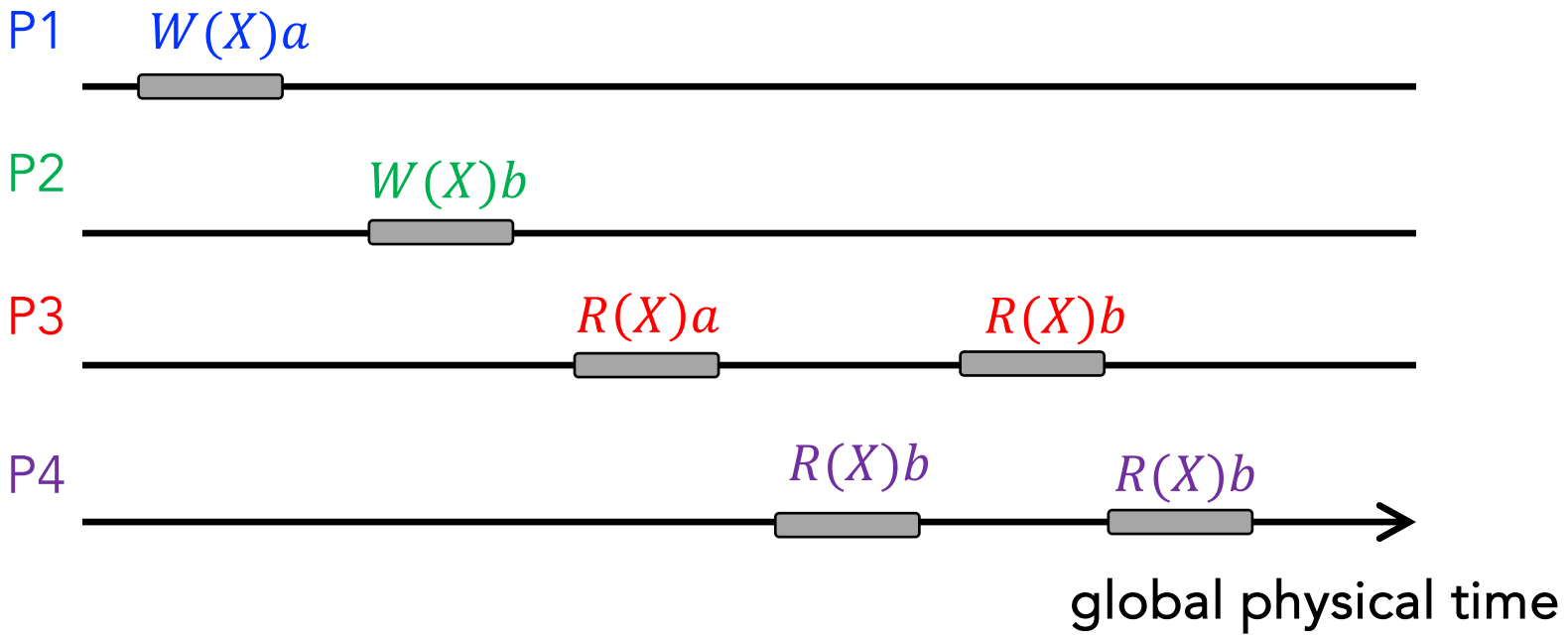
If underlying storage system is sequentially consistent, what can these reads return?

Sequential Consistency: (Counter) Examples



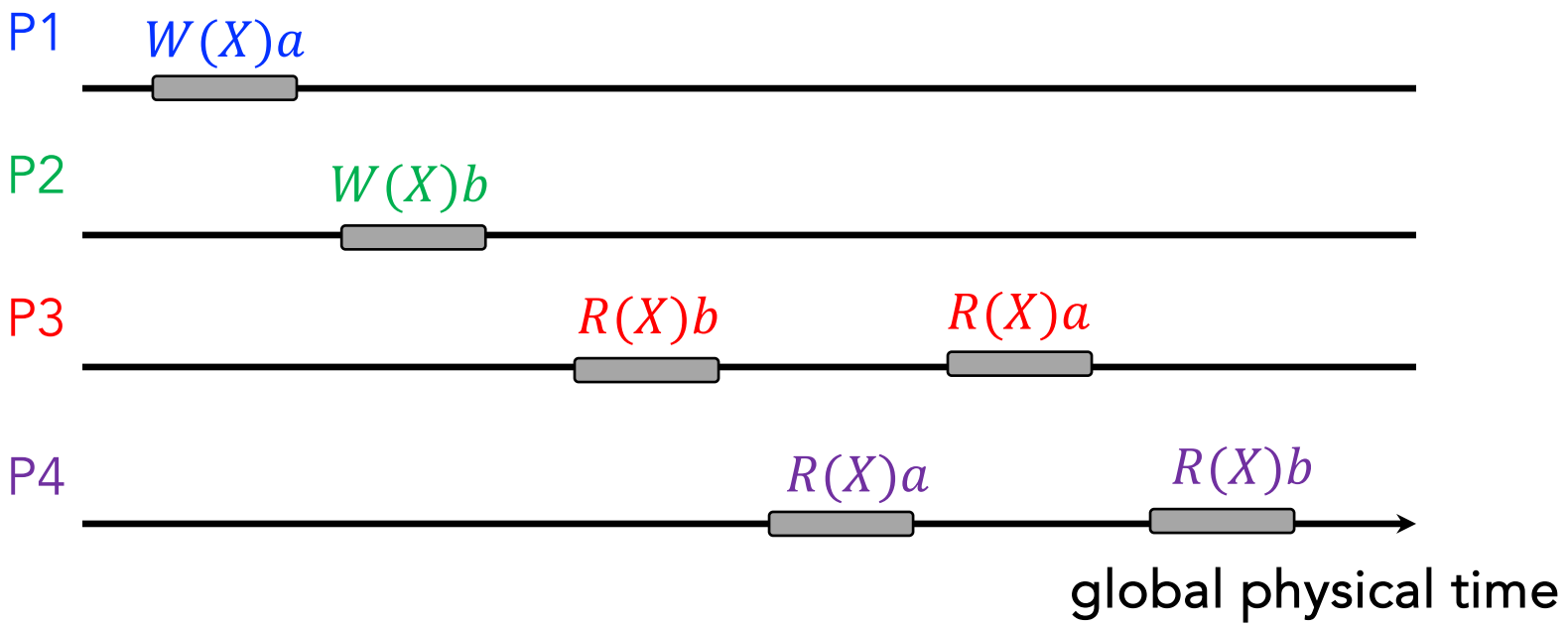
Can these reads be returned by a sequentially consistent system?

Sequential Consistency: (Counter) Examples



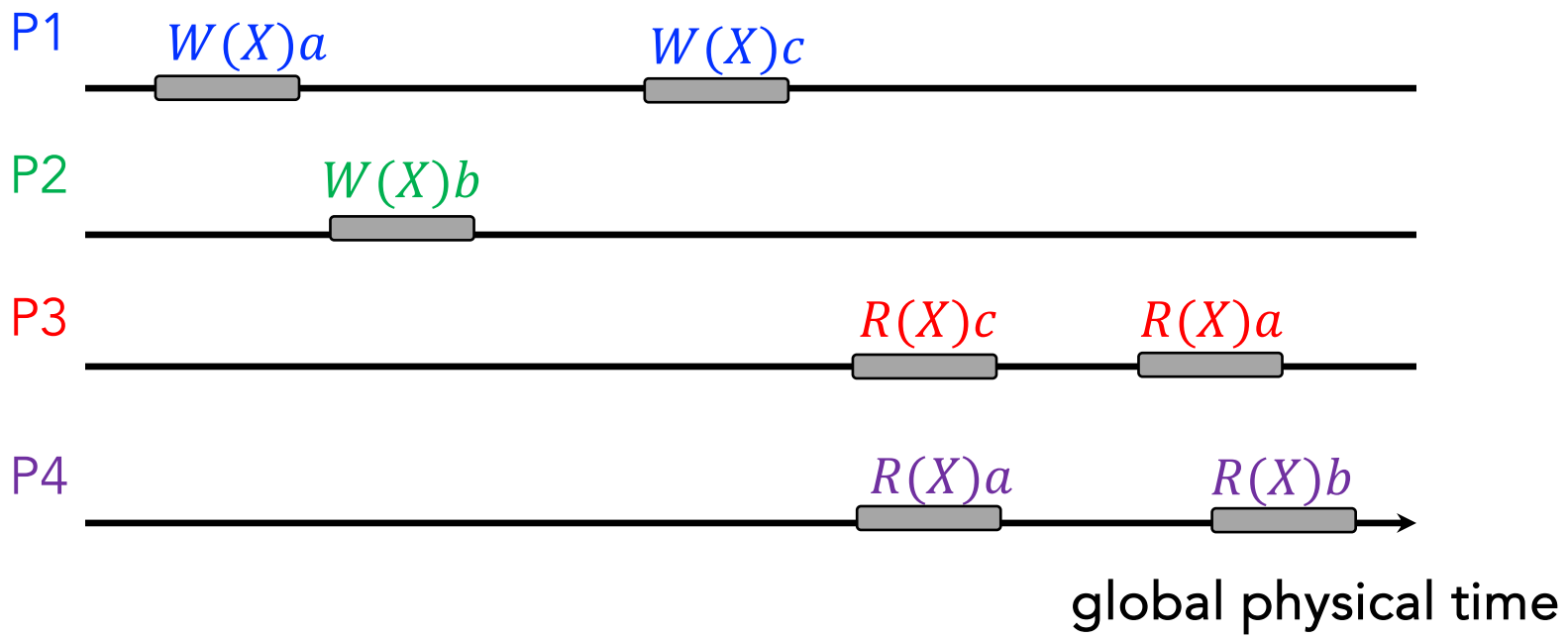
Can these reads be returned by a sequentially consistent system?

Sequential Consistency: (Counter) Examples



Can these reads be returned by a sequentially consistent system?

Sequential Consistency: (Counter) Examples



Can these reads be returned by a sequentially consistent system?

Causal Consistency [Hutto and Ahamad, 1990]

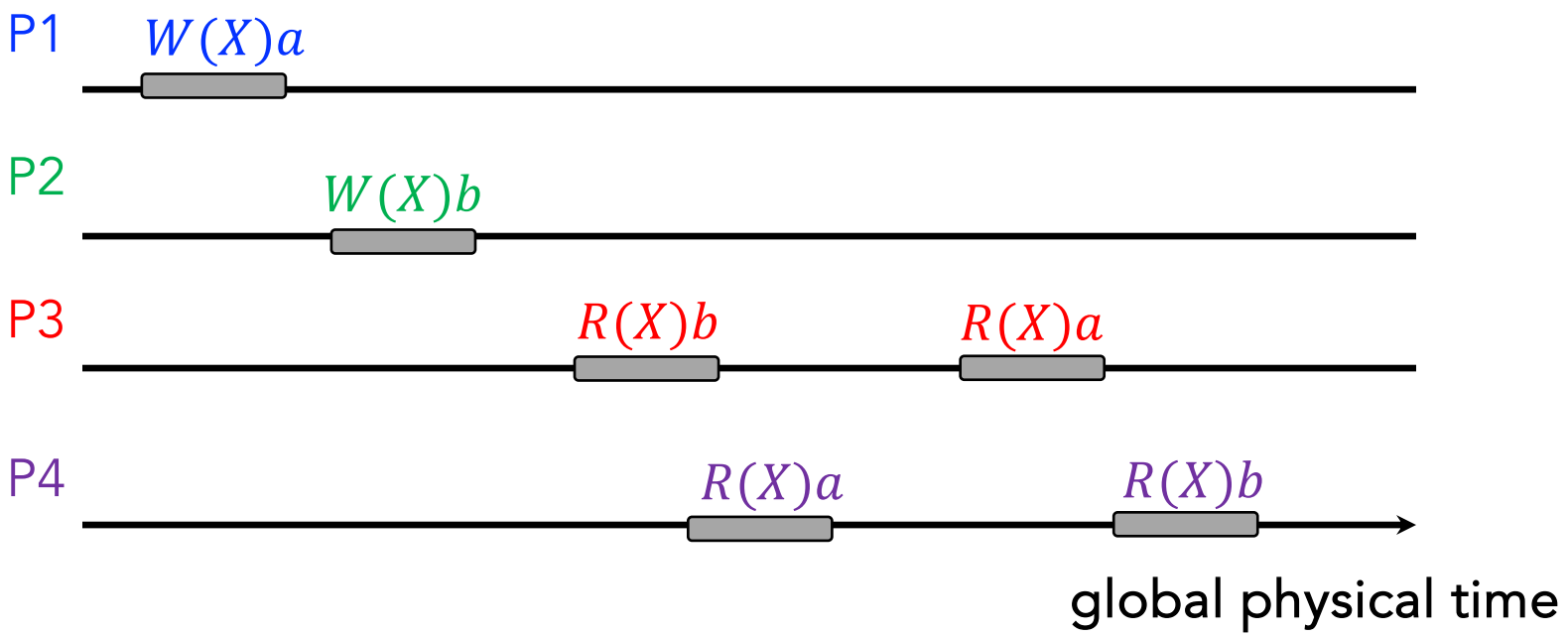
- Recall causality notion from Lamport clocks
  - Remember happens-before implies potential causality
  - That's what causal consistency enforces
- Causal consistency: potentially causally related operations must be seen by all processes in the same order
  - In other words, if $a \rightarrow b$, then a must execute before b on all replicas
  - All concurrent ops may be seen in different orders
- Key differences from sequential consistency
  - Does not require a total order

Causal Consistency: Implications

- Reads are fresh only w.r.t. the writes that they are (potentially) causally dependent on

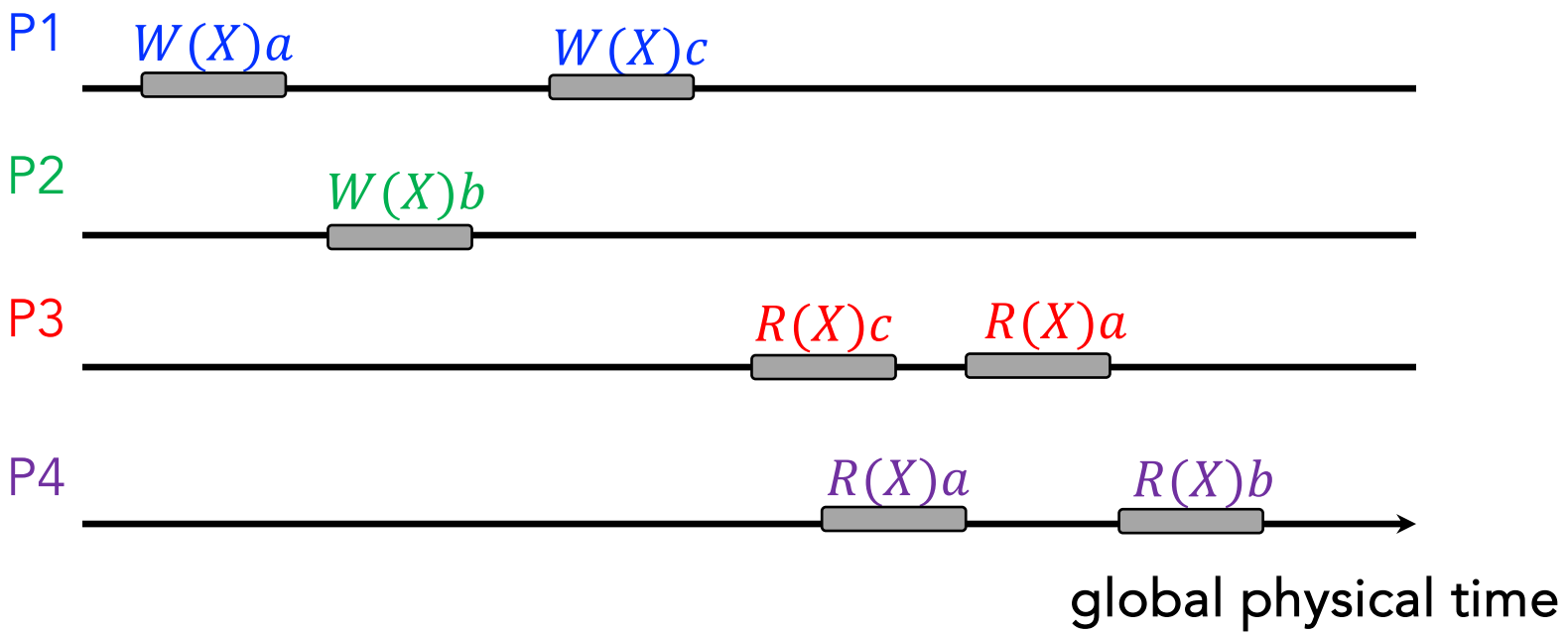
- Only (potentially) causally-related writes are ordered by all replicas in the same way
  - But concurrent writes may be committed in different orders

Causal Consistency: (Counter) Examples



Can these reads be returned by a causally consistent system?

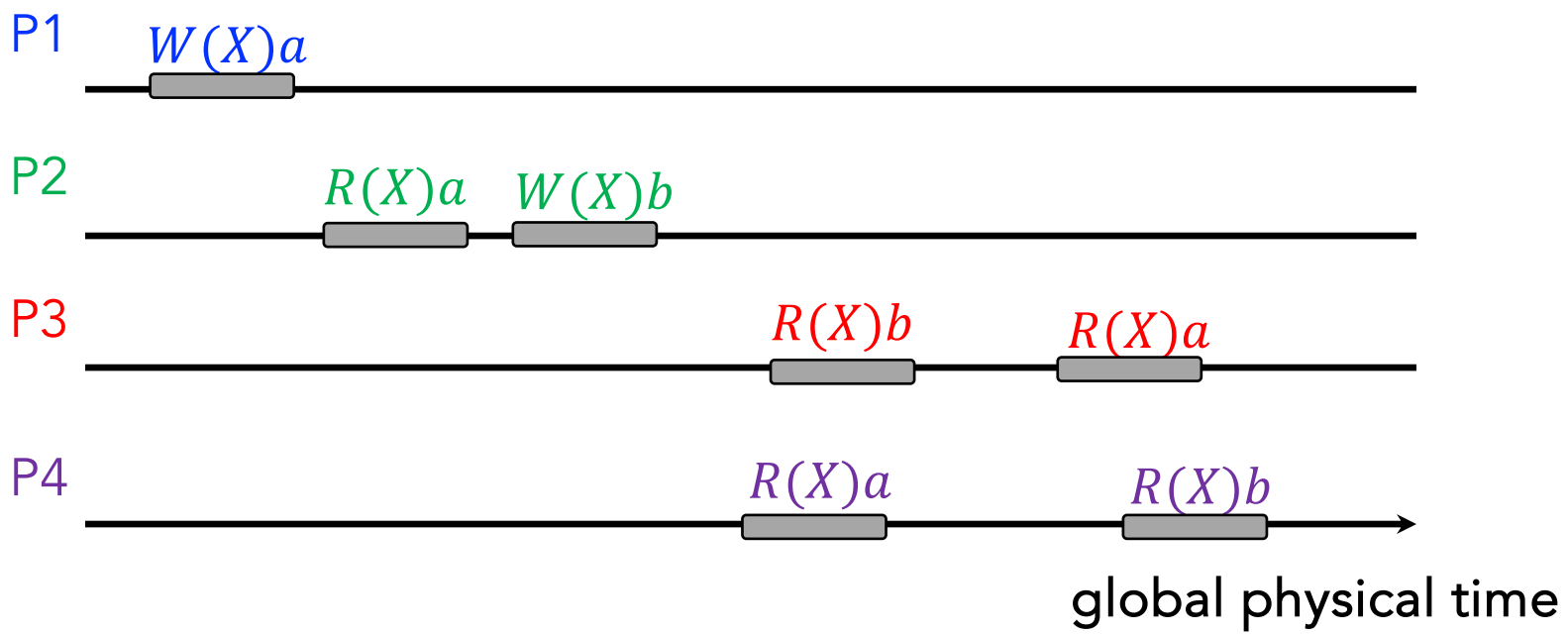
Causal Consistency: (Counter) Examples



Can these reads be returned by a causally consistent system?

Having read c, $R(X)c$, P3 must continue to read c or some newer value (perhaps b), but can't go back to a, because $W(X)c$ was conditional upon $W(X)a$ having finished

Causal Consistency: (Counter) Examples



Can these reads be returned by a causally consistent system?

Why Causal Consistency?

- Causal consistency is strictly weaker than sequential consistency although can give strange results, as you have seen
  - If system is sequentially consistent → it is also causally consistent
- BUT: it also offers more possibilities for concurrency
  - Concurrent operations (which are not causally-dependent) can be executed in different orders by different people
  - In contrast to sequential consistency, we do not need to enforce a global ordering of all operations
  - Hence, one can get better performance than sequential

Next Lecture

- Eventual Consistency
- Consensus in Distributed Systems