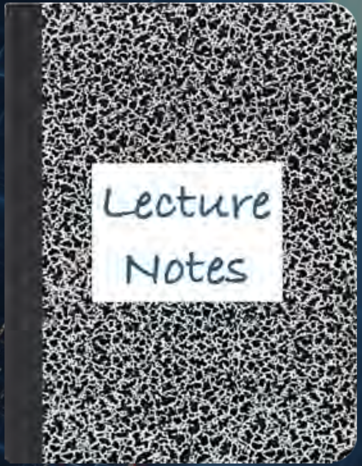
CS 417 – DISTRIBUTED SYSTEMS

Week 8: Distributed Transactions



Paul Krzyzanowski

© 2026 Paul Krzyzanowski. No part of this content may be reproduced or reposted in whole or in part in any manner without the permission of the copyright owner.

What We'll Cover

- **Concurrency control**: locking, optimistic, MVCC
- **Deadlock**: detection, prevention
- **The commit problem**: 2PC and 3PC
- **ACID** properties
- **Consistency models**: linearizability, sequential, causal, eventual
- **CAP theorem and PACELC**
- **ACID** vs. **BASE**

Transactions

What is a Transaction?

Transaction: a sequence of operations treated as a single logical unit of work

Ending a transaction

- **Commit**: make all changes permanent and visible
- **Abort** (**rollback**): undo all changes, return to prior state

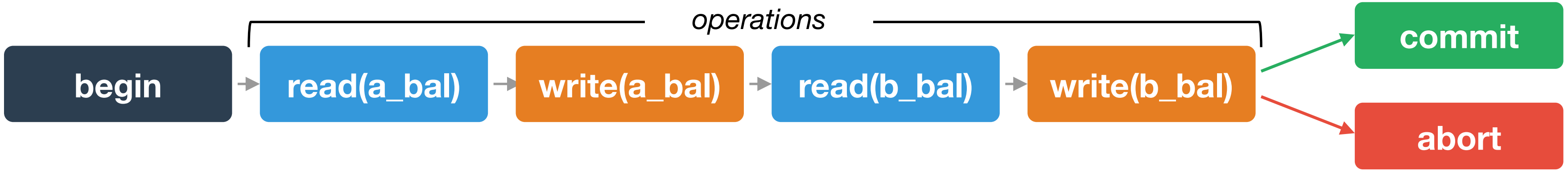
Fault tolerance -recovery

- **Write-ahead log** (**WAL**) enables recovery
  - Log changes to stable storage before applying them
  - On crash: redo committed transactions, undo incomplete ones

Distributed transactions

- Span multiple nodes
- Need protocols to guarantee all nodes commit or all abort

Transaction Lifecycle



Example: Bank transfer ($1,000 from A to B)

```
begin_transaction()
   a_bal = read(account_A)
   write(account_A, a_bal - 1000)
   b_bal = read(account_B)
   write(account_B, b_bal + 1000)
commit()
```

Write-Ahead Log (WAL)

- LOG: begin T1
- LOG: write a_bal = old_val → new_val
- LOG: write b_bal = old_val → new_val
- LOG: commit T1

Written to stable storage before data changes

Commit: all changes permanent

- Changes flushed to stable storage
- Visible to other transactions
- Survives crashes and power failures

Abort (rollback): all changes undone

- WAL replayed to undo partial writes
- System returns to pre-transaction state
- No partial state left behind

Concurrency Control

Concurrency Control

Goal: allow concurrent transactions while maintaining isolation

- **Schedule**: a sequence of reads/writes from concurrent transactions
- **Serializability**: final state equivalent to some serial execution

Two approaches:

Pessimistic

Conflicts are likely
Prevent with locks

Optimistic

Conflicts are rare
Check at commit time

Read and *Write* Locks

Exclusive locking for every access is too restrictive

- Two transactions that only read cannot create conflicts

Two lock types:

1. **Read lock (shared)** – multiple holders OK; blocks write locks
2. **Write lock (exclusive)** – single holder; blocks all other locks

Dramatically improves concurrency for read-heavy workloads

Lock Compatibility

Transaction A

Transaction B	No Lock	Read Lock	Write Lock
No Lock	✓	✓	✓
Read Lock	✓	✓	✗
Write Lock	✓	✗	✗

Multiple readers can access data simultaneously — writers need exclusive access

✓ = Compatible (can coexist) ✗ = Conflict (must wait)

Two-Phase Locking (2PL)

Protocol for serializability via locking

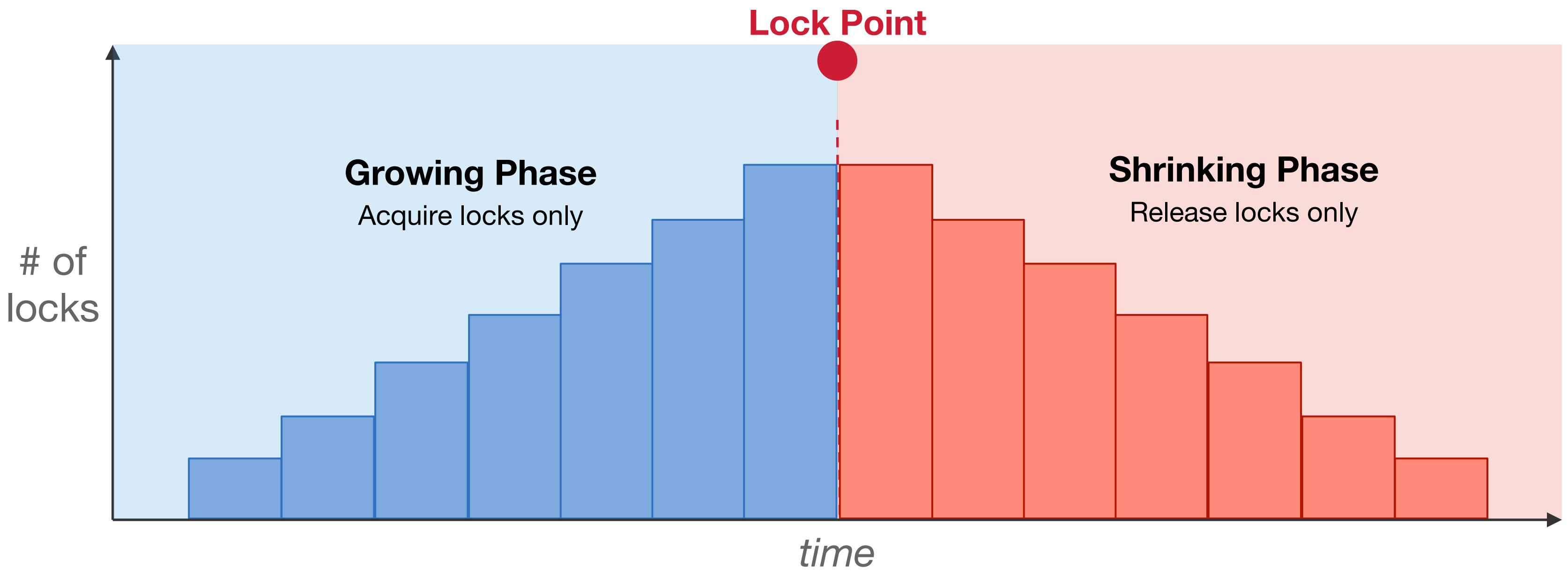
1. **Growing phase**: acquire locks – not allowed to release any
   - Eventually, the transaction gets to the ***lock point*** – when it does not need more locks
2. **Shrinking phase**: release locks – not allowed to acquire new ones

Problem: cascading aborts if released data is read by other transactions

Solutions

- **Strict 2PL**: hold *write* locks until commit/abort
- **Strong Strict 2PL (SS2PL)**: hold *ALL* locks until commit/abort
  (simpler implementation – many databases use this)

Two-Phase Locking: Lock Timeline



Basic 2PL

Release after lock point

Strict 2PL

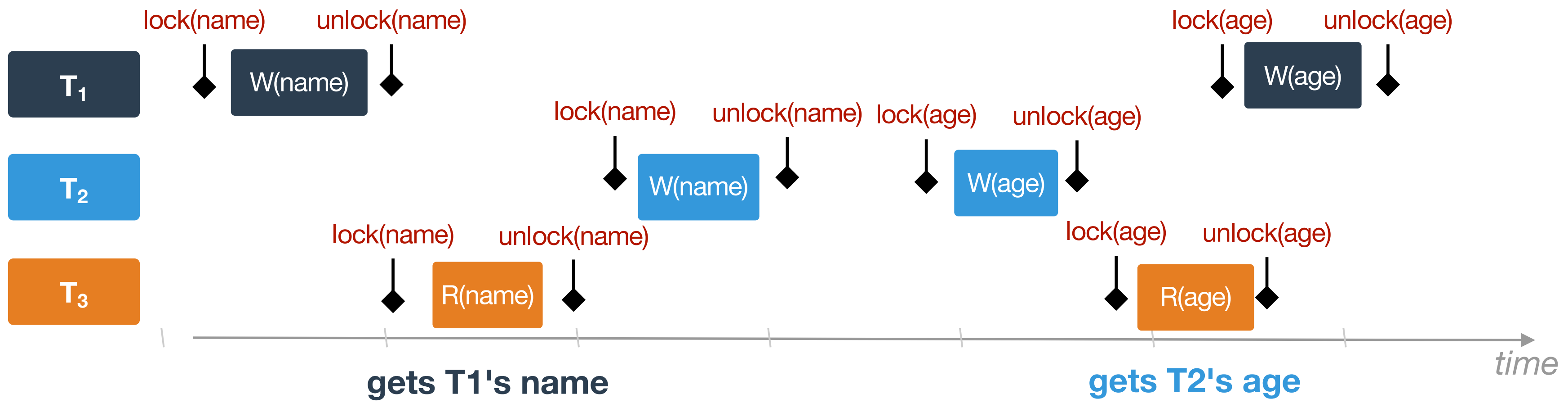
Hold write locks until commit

SS2PL

Hold ALL locks until commit

Without 2PL: Inconsistent Reads

Without 2PL: T_1 and T_2 both update *name* and *age*. T_3 reads both fields.



What T_3 reads:

name = T_1's age = T_2's

Valid states:

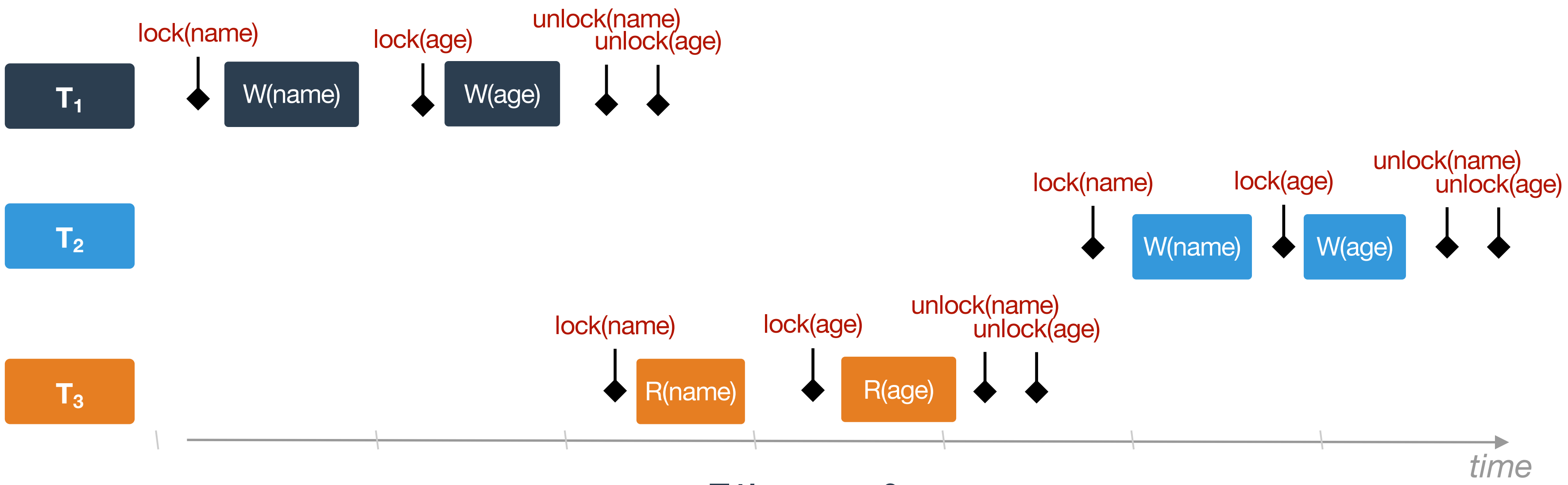
name = T_1's age = T_1's

name = T_2's age = T_2's

T_3 sees a mix of T_1 and T_2 — a state that never existed! 2PL prevents this.

2PL Ensures Isolation

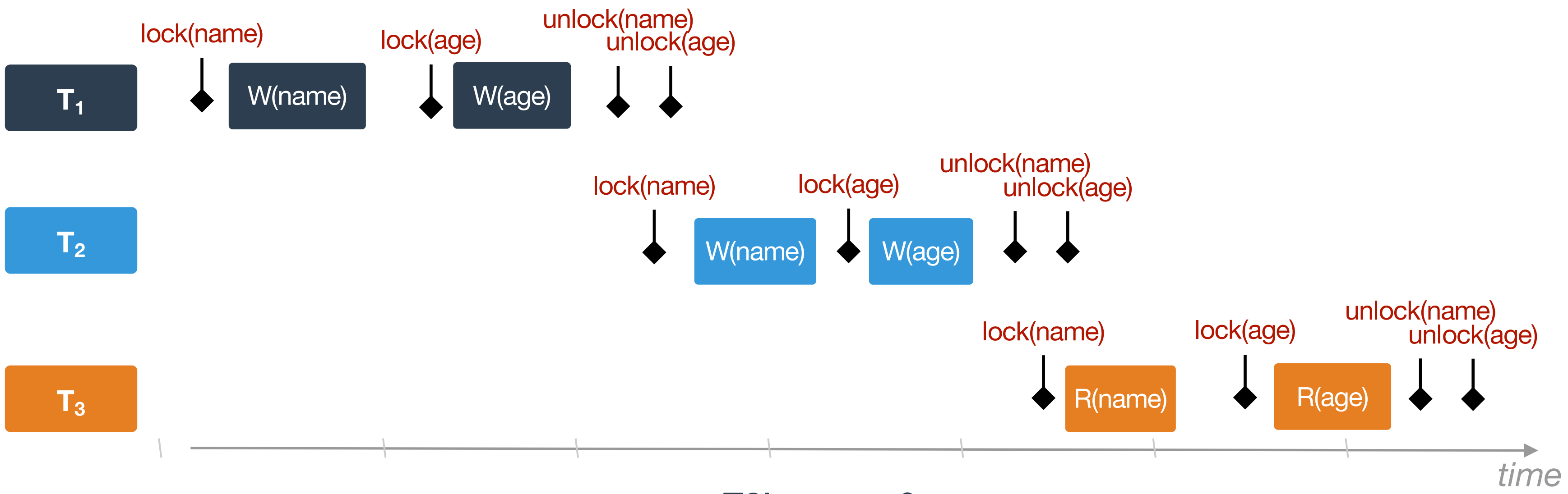
2PL: doesn't allow taking locks if any were released



gets T1's name & age

2PL Ensures Isolation: Another Valid Schedule

2PL: doesn't allow taking locks if any were released



gets T2's name & age

Deadlock

Deadlock

Transactions waiting for locks held by each other may result in a **deadlock**:

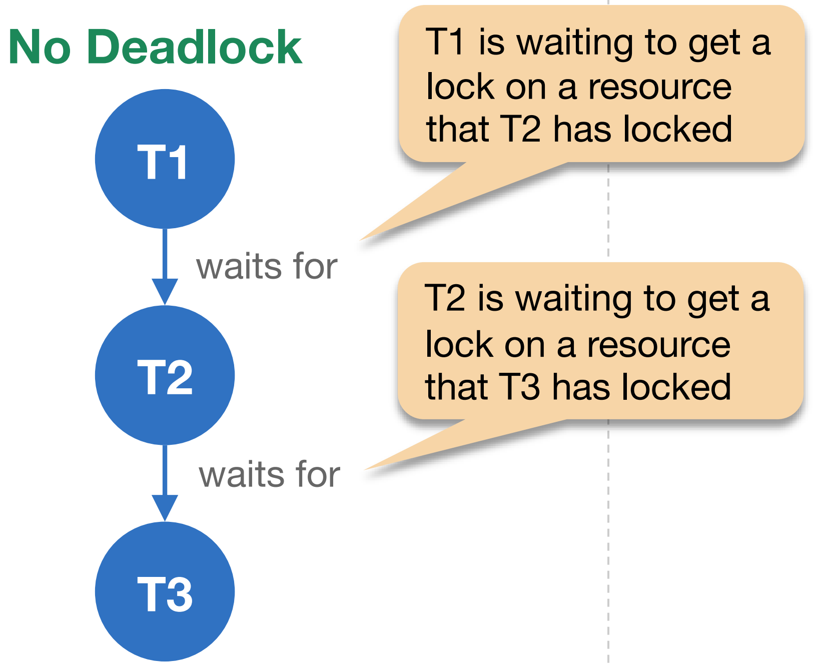
*A transaction cannot make progress because it needs a lock,
but it isn't ready to release a lock that another transaction needs*

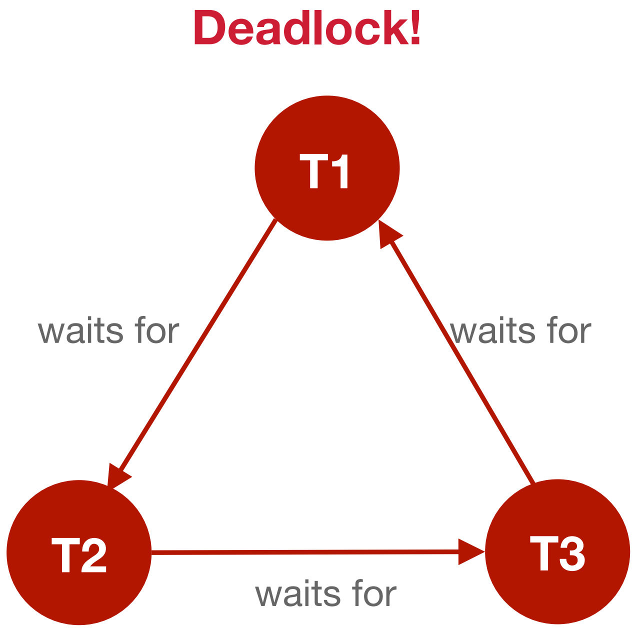
Four necessary conditions:

1. **Mutual exclusion** – resource held by at most one transaction
2. **Hold and wait** – holding locks while requesting more
3. **Non-preemption** – locks cannot be forcibly taken away
4. **Circular wait** – cycle of transactions waiting for each other

Wait-For Graph: Deadlock Example

Wait-for graph (WFG): nodes = transactions, edge **T1→T2** = ***T1 waits for T2***


No Deadlock
T1
waits for
T2
waits for
T3
T1 is waiting to get a lock on a resource that T2 has locked
T2 is waiting to get a lock on a resource that T3 has locked



Cycle in wait-for graph = deadlock detected

Deadlock Detection

Centralized detection

- Single coordinator collects local WFGs, merges into global graph
- Cycle in graph = deadlock detected
- Risk of **phantom deadlocks**: false positives from stale/out-of-order messages

Chandy-Misra-Haas (distributed via edge chasing)

- T_0 blocks: sends **probe** message (*originator, sender, receiver*) to holder
- Each blocked node forwards the *probe* to its own blockers
- If the *probe* returns to T_0 → a cycle exists → deadlock confirmed

Deadlock Prevention

Make cycles structurally impossible — no detection needed

- Each transaction gets a unique timestamp at the start

Wait-Die

- Older transaction waits for a younger one
- A younger that needs a resource held by an older one ***aborts*** *and restarts*
- Edges always flow *old* → *young* : no cycles are possible

Wound-Wait

- A younger transaction waits for an older one
- An older preempts a younger one by **aborting** it
- Edges always flow *young* → *old* : no cycles are possible

Optimistic Concurrency Control (OCC)

Assume conflicts are rare

No locks — proceed freely, and check for conflicts at commit time

Three phases:

1. **Working phase** – read/write to a private workspace, no locks held
2. **Validation phase** – check if data was modified by a committed transaction
3. **Update phase** – if the data wasn't modified, make changes permanent

Trade-offs:

- **Deadlock-free**, **maximum parallelism** during the working phase
- **But**: wasted work if aborted after validation fails (bad for high contention)

Optimistic Concurrency Control: Three Phases

1. Working → **2. Validation** → **3. Update**

1. Working

Read from database
Write to private workspace (***tentative***)

No locks needed!

2. Validation

Check if any other transaction modified data we read

Conflict? Abort!

3. Update

Make changes permanent in the real database

Now visible to all

Most transactions succeed without contention → minimal overhead

Multi-Version Concurrency Control

- Each transaction gets a **timestamp**
  - *Writes* create new versions of data
- **Snapshot isolation**:
  - *reads* see a consistent snapshot from the start of the transaction
- **Reads never block** – always read from snapshot, no waiting for writers
- *Write-write* conflicts: **first-committer-wins** rule at commit time
- Old versions of data need garbage collection

Used by PostgreSQL, Oracle, MySQL/InnoDB

Leases

Problem: if a lock holder crashes, the lock is never released

- **Lease** = lock with a time limit

Duration trade-off:

- **Short leases**: frequent renewals, may expire on slow-but-alive nodes
- **Long leases**: longer wait when holder actually fails

The Commit Problem

Why Distributed Commit is Hard

Easy with a single node: *write to WAL – flush data to disk – you're done*

Challenging with multiple nodes

Example: bank transfer

Database A (New York) debits, Database B (London) credits

- If A commits, but B crashes → *money destroyed*
- If B commits, but A rolls back → *money created from nothing*

No single node has full knowledge of all others' state

Two-Phase Commit (2PC) Protocol

One node acts as the **coordinator**; others are **participants**

Phase 1 – Prepare (Voting)

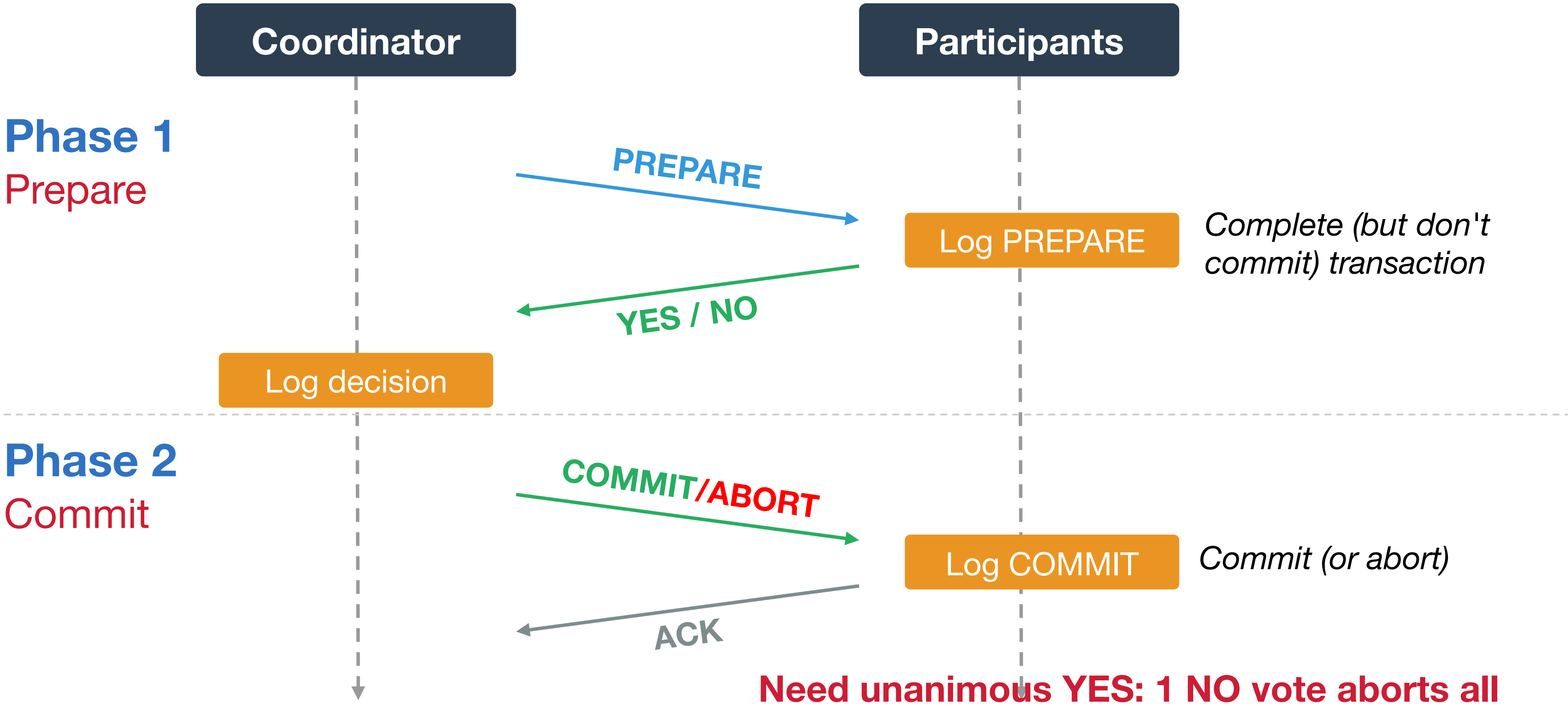
- Coordinator sends **PREPARE**; participants vote YES or NO
- YES: durable promise (flushed to stable storage) – cannot change the decision

Phase 2 – Commit or Abort

- All participants voted YES → coordinator logs **COMMIT**, broadcasts **COMMIT**
- Any participant votes NO → coordinator broadcasts **ABORT**

Requires unanimous agreement, not a majority (unlike Raft/Paxos)

2PC Protocol Flow



Need unanimous YES: 1 NO vote aborts all

2PC Failure Scenarios

Failure	Action
Participant fails before voting	Coordinator waits for recovery (**fail-recover** model)
Participant fails after YES, before decision	Enters **uncertain** state; must contact coordinator on recovery
Coordinator fails before decision	All uncertain, nobody knows outcome ⇒ ***2PC BLOCKS***
Coordinator fails after partial delivery	Recoverable by querying participants

Fundamental limitation:

- A participant that voted YES cannot unilaterally decide
- Locks are held indefinitely until the coordinator recovers

Three-Phase Commit (3PC)

Goal: eliminate the blocking behavior of 2PC

- **Phase 1 – CanCommit**: same as 2PC voting
- **Phase 2 – PreCommit**: coordinator signals intent to commit; participants ACK
- **Phase 3 – DoCommit**: coordinator sends final COMMIT

PreCommit removes the fatal uncertainty: *a new coordinator can query the state*

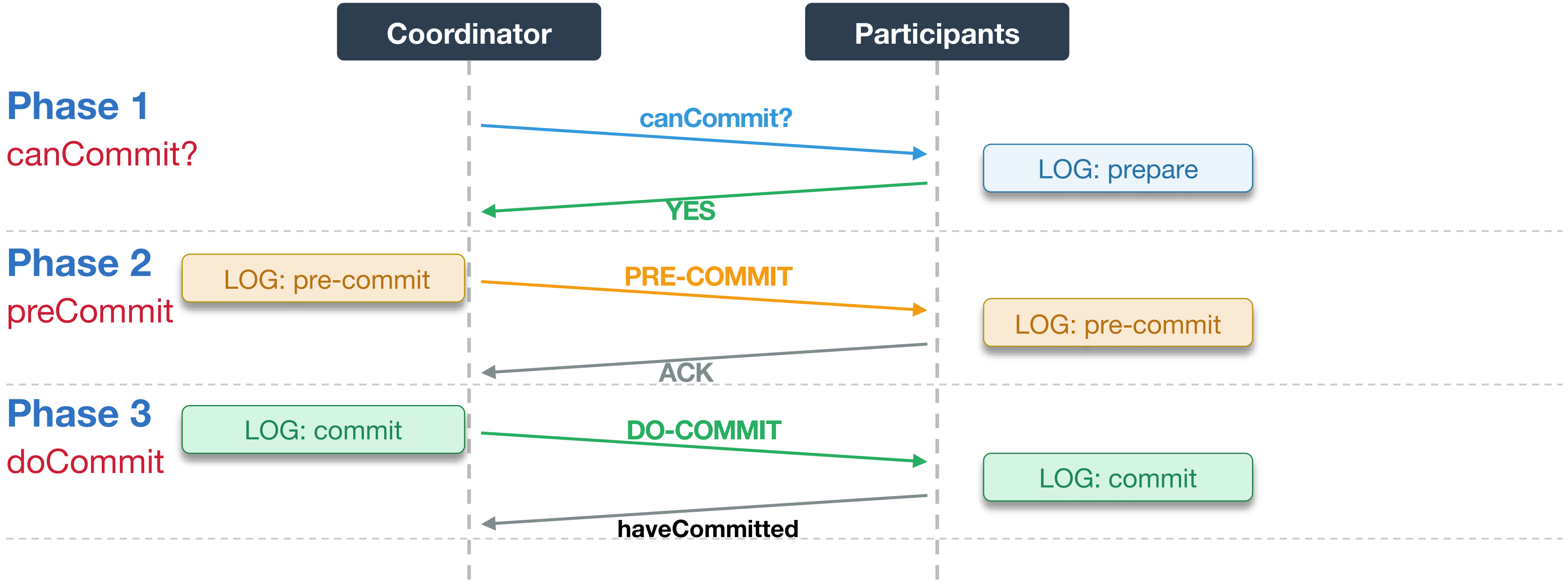
The catch: assumes a synchronous network

- Assumes bounded message delay & reliable failure detection — *rarely holds in practice*
- A partition in the PreCommit phase can cause both sides to reach different decisons

3PC is rarely implemented

- More reliable approach: run 2PC on a Raft/Paxos-replicated coordinator group

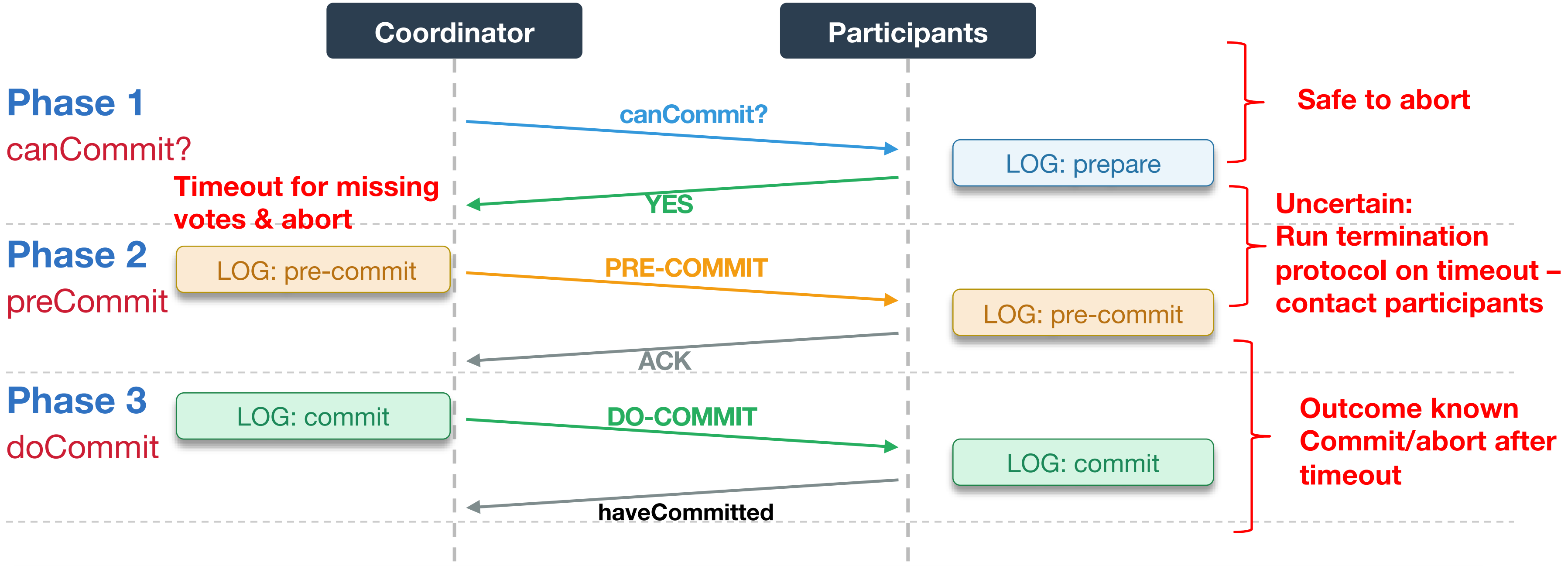
3PC Protocol Flow



Why preCommit is important for recovery:

If a new coordinator takes over: if **any** participant has a *pre-commit* in log → decision was *commit*. If **no** participant has pre-commit → safe to abort. This removes the 2PC blocking window.

3PC Protocol Flow: Timeout/Abort Opportunities



Why preCommit is important for recovery:

If a new coordinator takes over: if **any** participant has a *pre-commit* in log → decision was *commit*. If **no** participant has pre-commit → safe to abort. This removes the 2PC blocking window.

2PC vs. Consensus Protocols

	2PC	Raft / Paxos
Goal	Atomic commit across nodes	Agree on a single value/log
Agreement	Unanimous (every node)	Majority (F+1 of 2F+1)
Veto Power	Any participant can say NO	No veto — majority rules
Blocking	Blocks if coordinator fails	Non-blocking with quorum
Cost	2 stable-storage flushes per node	Log replication + quorum ACK

Hybrid approach: run 2PC coordinator as a Raft/Paxos group → fixes coordinator blocking

2PC: Availability Cost

Availability cost (serial system)

- 2 databases at 99.9% each → 99.8% combined (17.5 hrs downtime/yr)
- 5 databases → 99.5% (~2 full days downtime/yr)

Drives the push to minimize cross-database transactions

ACID

ACID Properties

Atomicity	All or nothing; no partial execution (2PC)
Consistency	Valid state to valid state; integrity constraints hold Different from distributed systems "consistency" (replica agreement)
Isolation	Concurrent transactions don't interfere (locking, OCC, MVCC)
Durability	committed = permanent, survives crashes (stable storage, write-ahead log)

Easy on one machine; every property
becomes harder and more expensive across multiple machines

Consistency Models

Consistency Model Spectrum

STRONGEST → **WEAKEST**

Linearizability	Sequential	Causal	Eventual	No guarantee
Real-time order for non-overlapping operations. All ops appear instantaneous.	Global total order. No real-time guarantees.	Preserves cause-effect. Concurrent ops unordered.	All replicas converge... eventually.	No ordering promises. Best effort only.

Performance
Low → High

Availability
Low → High

Key tradeoff: stronger consistency = lower performance + lower availability

Linearizability

Strongest practical consistency model

1. Every operation appears to happen **instantaneously** between invocation and completion
2. Order must be **consistent with real time**
3. System behaves as if there is a single copy of the data

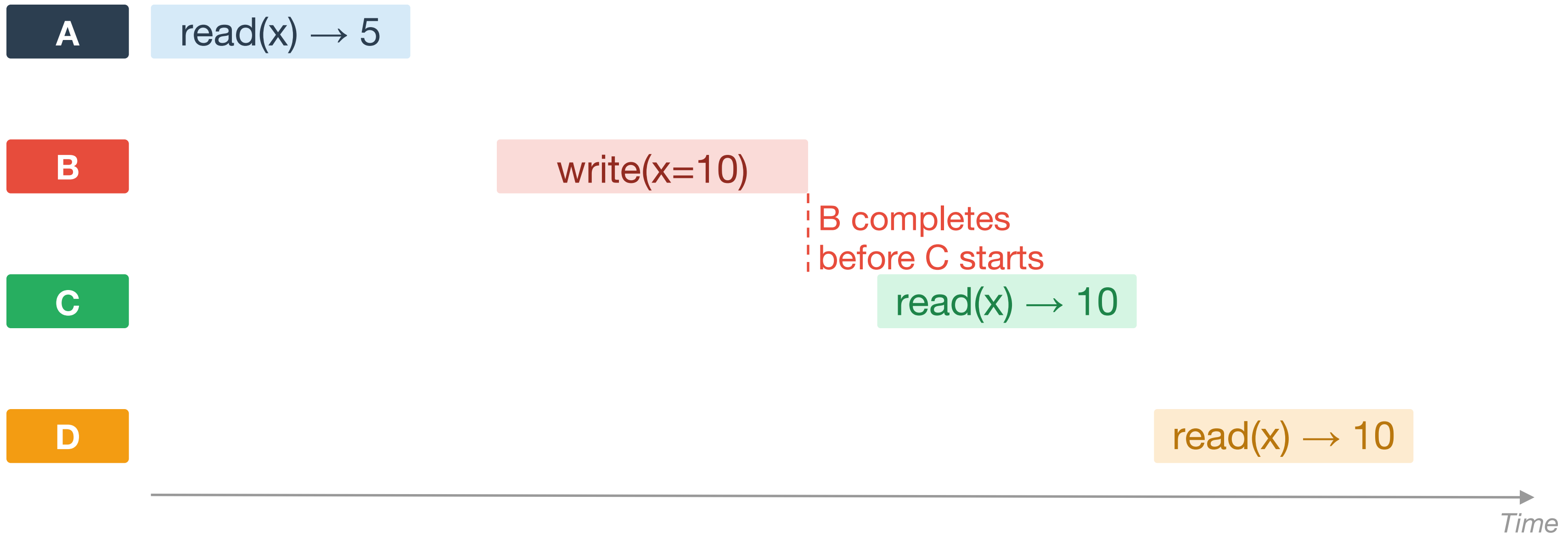
Implementations:

– Central coordinator, leader-based replication, Raft/Paxos
– Google Spanner: TrueTime (hardware-assisted clock bounds)

Cost: coordination on every read/write → higher latency

Example: Linearizability

x was last written as 5. Real-time order matters.



Allowed:
C returns 10 (B finished before C started)
D returns 10 (after C, must see same or newer)

Forbidden:
C returns 5 (B already completed)
D returns 5 (would violate real-time order)

Sequential & Causal Consistency

Sequential consistency

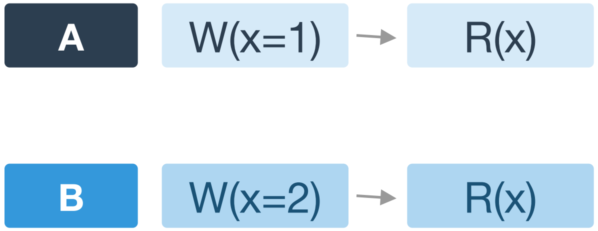
- All processes agree on some total order of operations
- Each process's ops appear in program order within that total order
- Drops real-time requirement — weaker than linearizability

Causal consistency

- Only causally related operations must be seen in same order
- Independent operations can be seen in different orders by different nodes
- Implementable with vector clocks; lower latency than sequential

Example: Sequential Consistency

A writes x=1 then reads x.   B writes x=2 then reads x.



Valid global order 1:   A:W(1) → B:W(2) → A:R → B:R   **A reads 2, B reads 2 ✓**

Valid global order 2:   B:W(2) → A:W(1) → B:R → A:R   **B reads 1, A reads 1 ✓**

Valid global order 3:   A:W(1) → A:R → B:W(2) → B:R   **A reads 1, B reads 2 ✓**

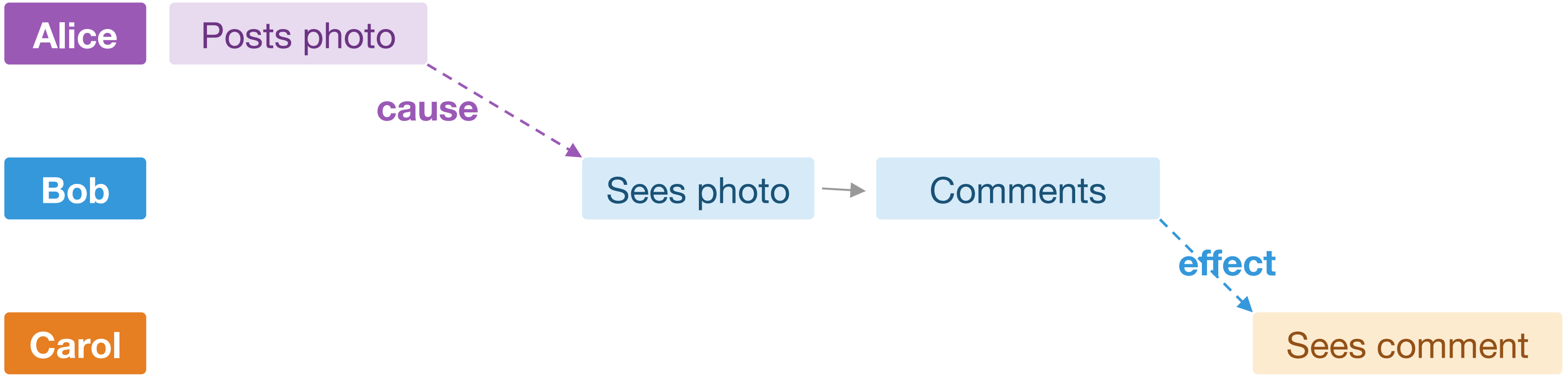
Invalid: A reads 2, B reads 1 simultaneously   **No total order produces this ✗**

Key: unlike linearizability, real-time order doesn't matter

A could see B's write before B sees A's write — as long as one consistent total order exists that respects each process's local program order.

Example: Causal Consistency

Social network: Alice posts a photo, Bob sees it and comments.



Causal consistency requires:

Carol sees the comment
→ **Carol MUST also see the photo (the cause)**

Causally inconsistent:

Carol sees Bob's comment
→ **but does NOT see Alice's photo** ✗

Only causally related events must be ordered.
Independent events (e.g., Dave posts an unrelated status) can appear in any order.

Eventual Consistency

Weakest useful guarantee

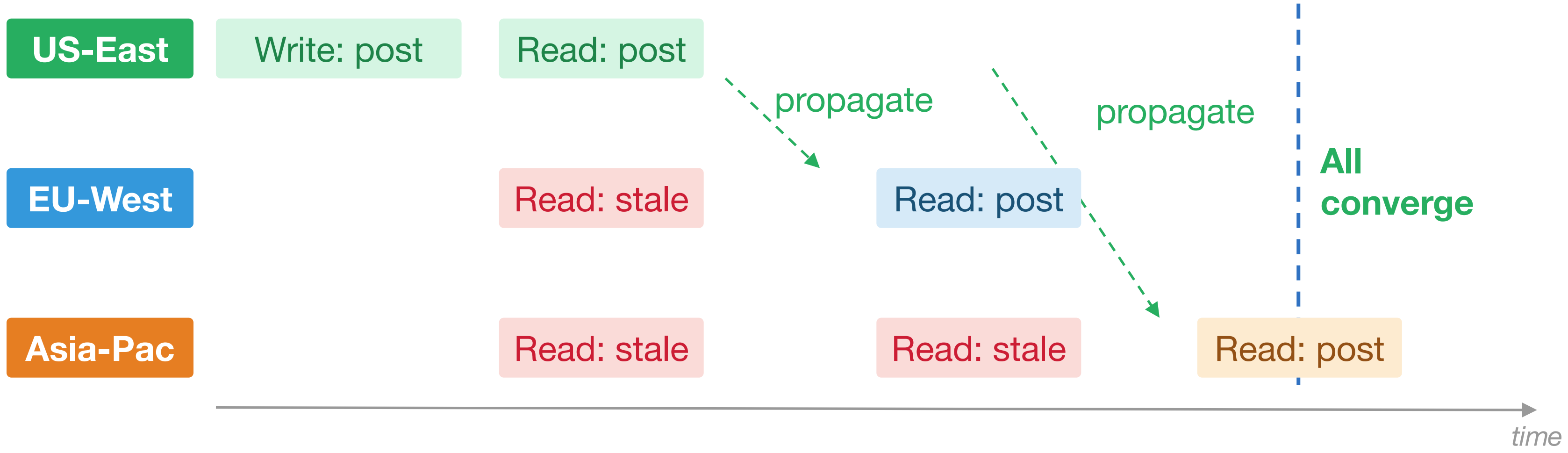
- If no new updates, all replicas **eventually converge** to the same value
- No guarantee on **when**; stale reads are possible in the interim
- Examples: DNS propagation, social media feeds

Benefits:

- High availability, low latency — writes ACKed from single local replica
- Application must tolerate stale reads and handle write conflicts

Example: Eventual Consistency

You post a status update. Your nearby replica has it immediately.



The guarantee:

If no new updates are made, all replicas will eventually converge to the same value.

Real-world examples:

DNS propagation (minutes to hours)
Social media feeds, shopping carts

Tradeoff: low latency + high availability, but stale reads during propagation

Serializability vs. Linearizability

Serializability

- Property of **transactions** (multi-step, multi-object)
- Result equivalent to **some** serial order — says nothing about real time

Linearizability

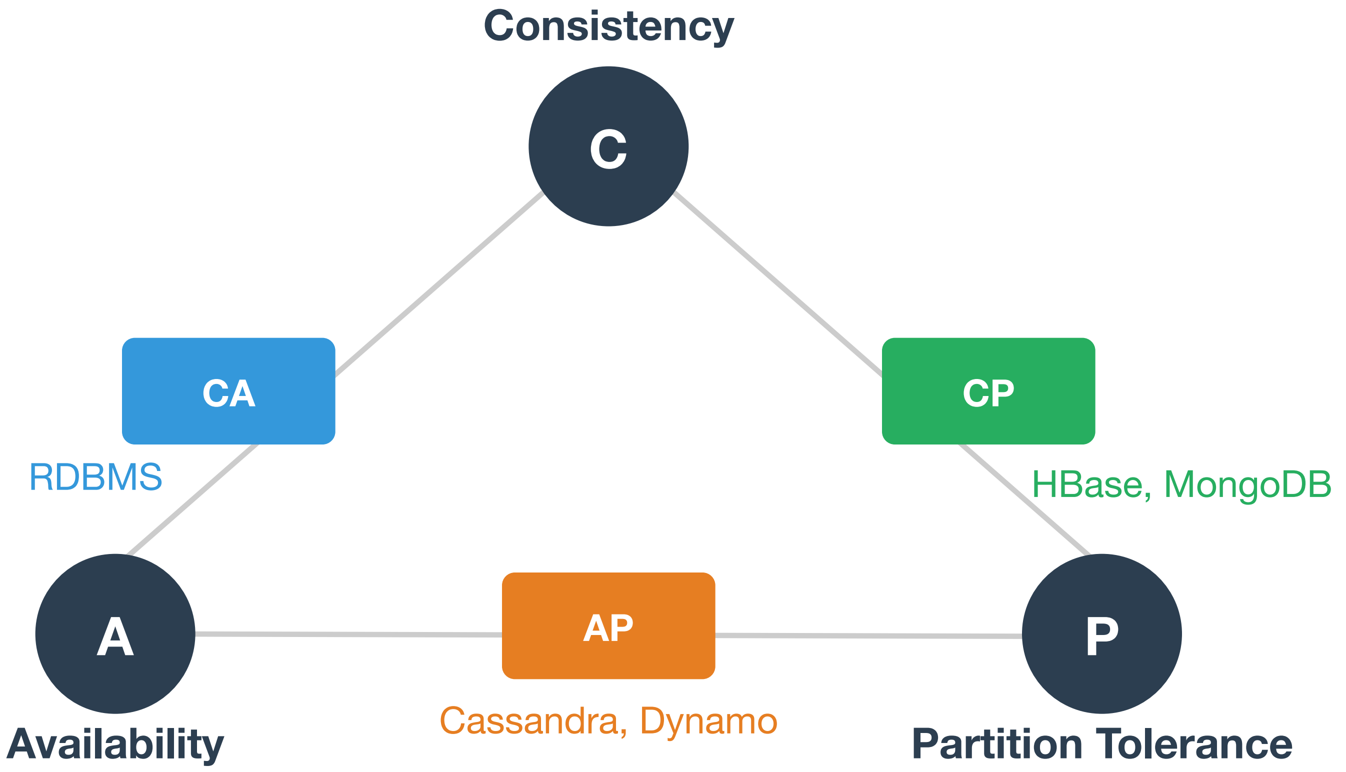
- Property of **individual operations** on a single object
- The order must respect **real time**

They are independent — you can have one without the other

Strong serializability = both combined (e.g., Google Spanner)

The CAP Theorem

CAP Theorem: Pick Two



During a network partition, you must choose: *consistency* OR *availability*

CAP Theorem

C	**Consistency**	Every read gets the most recent write (linearizability)
A	**Availability**	Every request to a non-failing node gets a response
P	**Partition Tolerance**	System operates despite network splits

During a partition, you must choose C or A

- **CP**: reject requests during partition (banks, financial systems)
- **AP**: accept reads/writes, reconcile later (DNS, shopping carts)

PACELC

CAP says nothing about normal operation

Partitions are rare events (but must be addressed)

Latency–consistency tradeoff matters more in normal operation

if Partition → trade A vs. C
else → trade L vs. C

System	Partition (P)	Normal (E)	Examples
PA/EL	Favor Availability	Favor Low Latency	Dynamo, Cassandra, Riak
PA/EC	Favor Availability	Favor Consistency	MongoDB (some configs)
PC/EC	Favor Consistency	Favor Consistency	HBase, Spanner, VoltDB

BASE

Basically **A**vailable, **S**oft state, **E**ventually consistent

Basically Available	Prioritize responding over consistency – serve stale data if needed
Soft State	The state may change over time as updates propagate
Eventually Consistent	Replicas converge given time without new updates

- Not a protocol — a **design philosophy**; shifts inconsistency handling to application
- Chemistry pun: acid and base are chemical opposites

ACID vs. BASE

Use ACID when:

- Financial transfers, medical records, any partial update causes harm
- Worth the cost in latency and throughput

Use BASE when:

- Social feeds, recommendations, shopping carts — stale reads tolerable
- Need dramatically better scalability and availability

Most environments are hybrid

- ACID for payments; eventually consistent for transaction history display
- *The art: identify which parts need strong consistency and which don't*

The End