Eventual Consistency & Bayou

COS 418: Distributed Systems

Lecture 8

Wyatt Lloyd

Availability versus Consistency

- Later topic: Distributed consensus algorithms
  - Strong consistency (ops in same order everywhere)
  - But, strong reachability/availability requirements

If the network fails (common case), can we provide any consistency when we replicate?

Eventual consistency

- Eventual consistency: If no new updates to the object, eventually all reads 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 of data
  - Disconnected operation

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

Bayou:
A Weakly Connected Replicated Storage System

- Meeting room calendar application as case study in ordering and conflicts in a distributed system with poor connectivity

- Each calendar entry = room, time, set of participants

- Want everyone to see the same set of entries, eventually
  - Else users may double-book room
    - or avoid using an empty room

Paper context

- Early ’90s: Dawn of PDAs, laptops
  - H/W clunky but showing clear potential
  - Commercial devices did not have wireless.
- This problem has not gone away!
  - Devices might be off, not have network access
    - Mainly outside the context of datacenters
  - Local write/reads still really fast
    - Even in datacenters when replicas are far away (geo-replicated)

Why not just a central server?

- Want my calendar on a disconnected mobile phone
  - i.e., each user wants database replicated on their device
  - Not just a single copy
- But phone has only intermittent connectivity
  - Mobile data expensive, Wi-Fi not everywhere, all the time
  - Bluetooth useful for direct contact with other calendar users' devices, but very short range

Swap complete databases?

- Suppose two users are in Bluetooth range
  - Each sends entire calendar database to other
  - Possibly expend lots of network bandwidth

- What if the calendars conflict, e.g., the two calendars have concurrent meetings in a room?
  - iPhone sync keeps both meetings
  - Want to do better: automatic conflict resolution

Automatic conflict resolution: Granularity of "conflicts"

- Can't just view the calendar database as abstract bits:
  - Too little information to resolve conflicts:

1. "Both files have changed" can falsely conclude calendar conflict
   - e.g., Monday 10am meeting in room 3 and Tuesday 11am meeting in room 4

2. "Distinct record in each db changed" can falsely conclude no conflict
   - e.g., Monday 10–11am meeting in room 3 Doug attending, Monday 10-11am meeting in room 4 Doug attending, …

Application-specific conflict resolution

- Intelligence that can identify and resolve conflicts
  - More like users' updates: read database, think, change request to eliminate conflict
  - Must ensure all nodes resolve conflicts in the same way to keep replicas consistent

Application-specific update functions

- Suppose calendar write takes form:
  - “10 AM meeting, Room=302, COS-418 staff”
  - How would this handle conflicts?

- Better: write is an update function for the app
  - “1-hour meeting at 10 AM if room is free, else 11 AM, Room=302, COS-418 staff”

Potential Problem:
Permanently inconsistent replicas

- Node **A** asks for meeting **M1** at 10 AM, else 11 AM
- Node **B** asks for meeting **M2** at 10 AM, else 11 AM

- Node **X** syncs with **A,** then **B**
- Node **Y** syncs with **B,** then **A**

- **X** will put meeting **M1** at **10:00**
- **Y** will put meeting **M1** at **11:00**

Can’t just apply update functions when replicas sync

Totally Order the Updates!

- Maintain an ordered list of updates at each node

  Write log

  - Make sure every node holds same updates
    - And applies updates in the same order
  - Make sure updates are a deterministic function of db contents
- If we obey above, “sync” is simple merge of two ordered lists

Agreeing on the update order

- Timestamp: ⟨local timestamp **T,** originating node **ID**⟩
- Ordering updates a and b:
  - a < b **if** a.T < b.T or (a.T = b.T and a.ID < b.ID)

Write log example

- ⟨701, A⟩: A asks for meeting **M1** at 10 AM, else 11 AM
- ⟨770, B⟩: B asks for meeting **M2** at 10 AM, else 11 AM

Timestamp

- Pre-sync database state:
  - A has M1 at 10 AM
  - B has M2 at 10 AM
- What's the correct eventual outcome?
  - The result of executing update functions in timestamp order: **M1** at 10 AM, **M2** at 11 AM

Write log example: Sync problem

- ⟨701, A⟩: A asks for meeting **M1** at 10 AM, else 11 AM
- ⟨770, B⟩: B asks for meeting **M2** at 10 AM, else 11 AM

- Now A and B sync with each other. Then:
  - Each sorts new entries into its own log
    - Ordering by timestamp
  - Both now know the full set of updates

- A can just run B's update function
- But B has already run B's operation, too soon!

Solution: Roll back and replay

- B needs to "roll back" the DB, and re-run both ops in the correct order

- Bayou User Interface: Displayed meeting room calendar entries are "Tentative" at first
  - B's user saw M2 at 10 AM, then it moved to 11 AM

Big point: The log at each node holds the truth; the DB is just an optimization

Does update order respect causality?

- ⟨701, A⟩: A asks for meeting M1 at 10 AM, else 11 AM
- ⟨700, B⟩: Delete update ⟨701, A⟩
  - Possible if B's clock is slow, and using real-time timestamps

- Result: delete will be ordered before add
  - (Delete never has an effect.)

- Q: How can we assign timestamp to respect causality?

Lamport clocks respect causality

- Want event timestamps so that if a node observes E1 then generates E2, then TS(E1) < TS(E2)

- Use lamport clocks!
  - If E1 → E2 then TS(E1) < TS(E2)

Lamport clocks respect causality

- ⟨701, A⟩: A asks for meeting M1 at 10 AM, else 11 AM
- ~~⟨700, B⟩: Delete update ⟨701, A⟩~~
- ⟨706, B⟩: Delete update ⟨701, A⟩

- With Lamport clocks:
  - When A sends ⟨701, A⟩, it includes its clock, T (> 701)
  - When B receives ⟨701, A⟩, it updates its clock to T’ > T
  - When B creates the delete, it timestamps it with its clock, T” > T’
  - T” > T’ > T > 701 (e.g., T” is 706)

- Q: What if A and B are concurrent?

Timestamps for write ordering: Limitations

- Never know whether some write from "the past" may yet reach your node…
  - So all entries in log must be tentative forever
  - And you must store entire log forever

Want to commit a tentative entry, so we can trim logs and have meetings

Fully decentralized commit

- Strawman: Update ⟨10, A⟩ committed when all nodes have seen all updates with TS ≤ 10

- Have sync always send in log order
- If you have seen updates with TS > 10 from every node then you'll never again see one < ⟨10, A⟩
  - So ⟨10, A⟩ is committed

- Why doesn't Bayou do this?
  - A node that remains disconnected prevents commiting
    - So many writes may be rolled back on re-connect

How Bayou commits writes

- Bayou uses a primary commit scheme
  - One designated node (the primary) commits updates

- Primary marks each write it receives with a permanent CSN (commit sequence number)
  - That write is committed
  - Complete timestamp = ⟨CSN, local TS, node-id⟩

Advantage: Can pick a primary node close to locus of update activity

How Bayou commits writes (2)

- Nodes exchange CSNs when they sync
- CSNs define a total order for committed writes
  - All nodes eventually agree on the total order
  - Tentative writes come after all committed writes

Committed vs. tentative writes

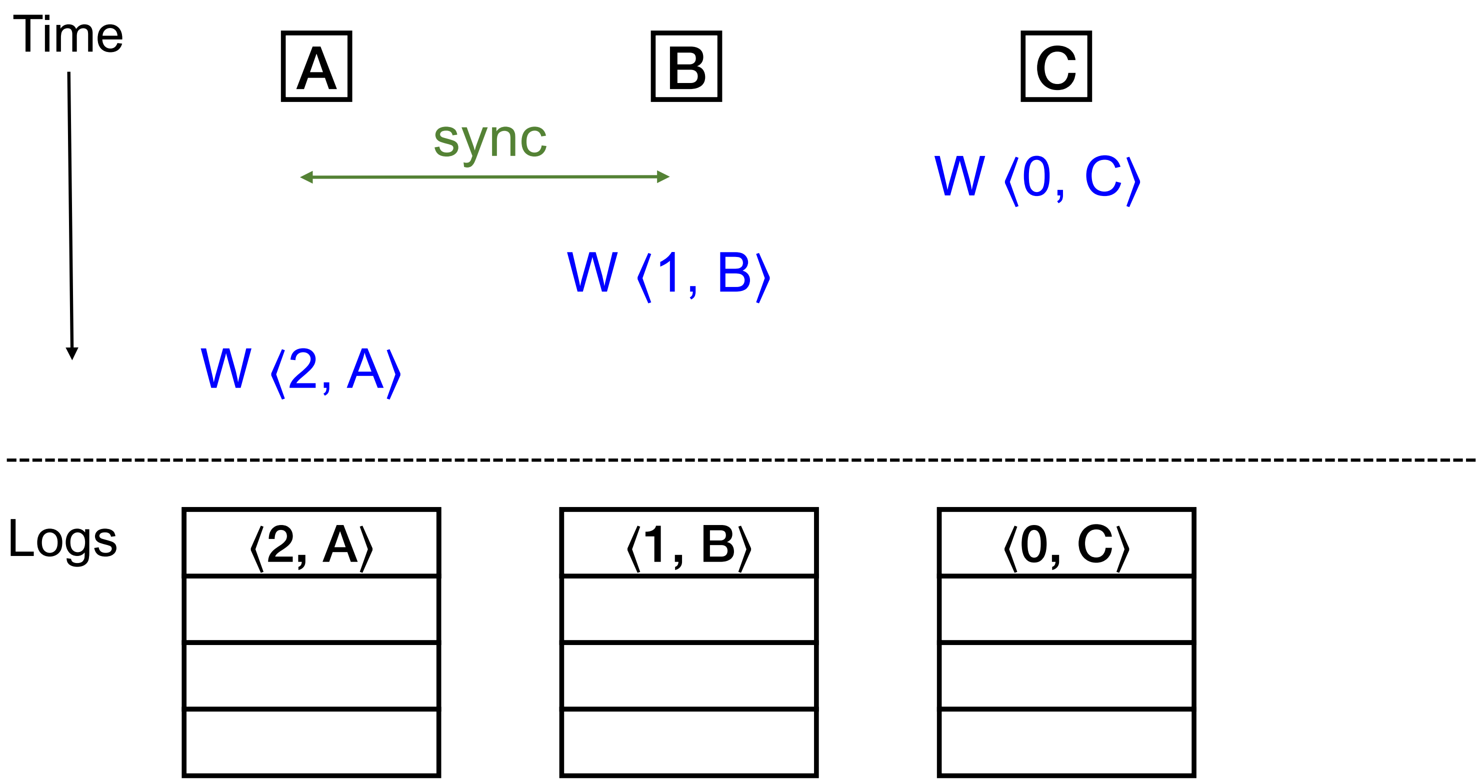
- Suppose a node has seen every CSN up to a write, as guaranteed by propagation protocol
  - Can then show user the write has committed
    - Mark calendar entry "Confirmed"
- Slow/disconnected node cannot prevent commits!
  - Primary replica allocates CSNs

Tentative writes

- What about tentative writes, though? How do they behave, as seen by users?

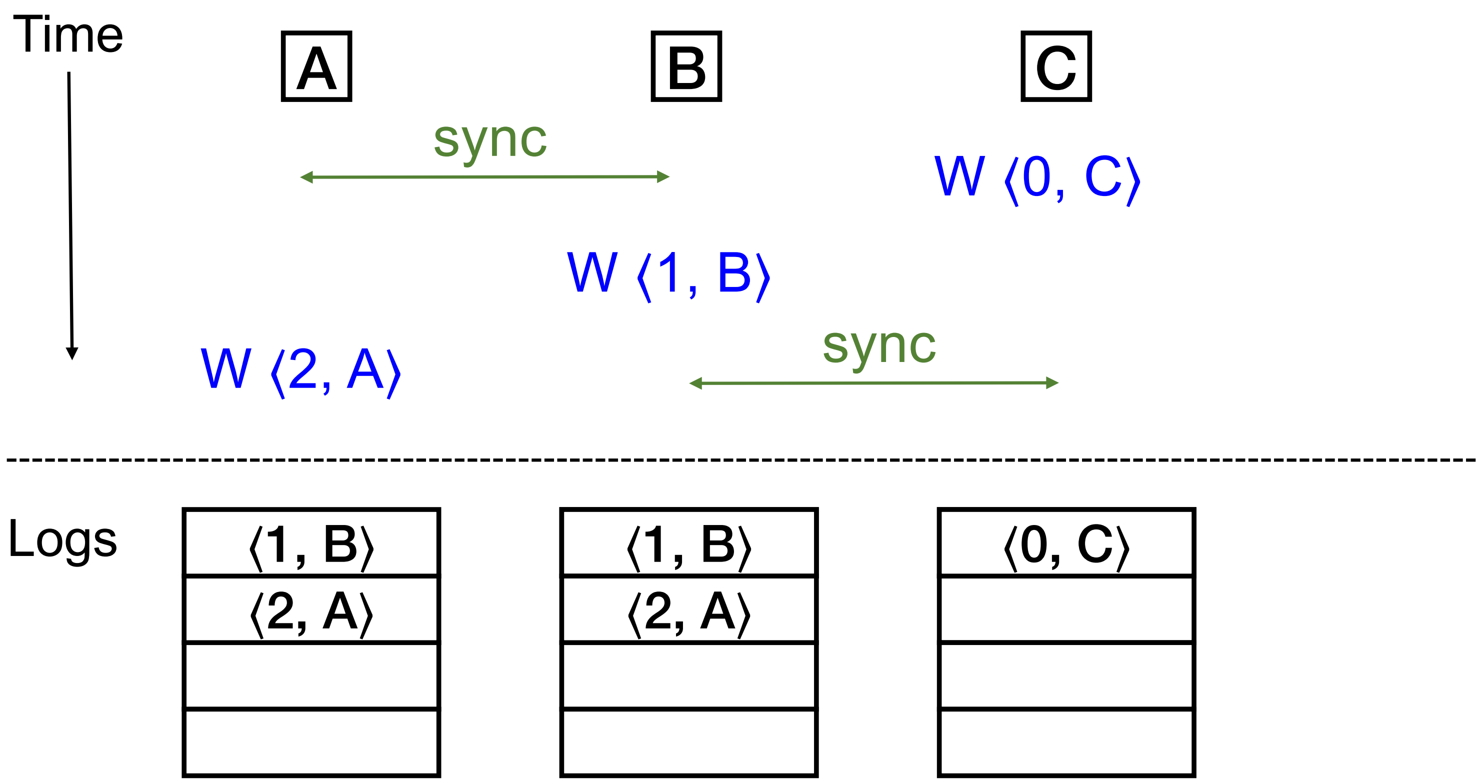
- Two nodes may disagree on meaning of tentative writes
  - Even if those two nodes have synced with each other!
  - Only CSNs from primary replica can resolve disagreements permanently

Ex: Disagreement on tentative writes



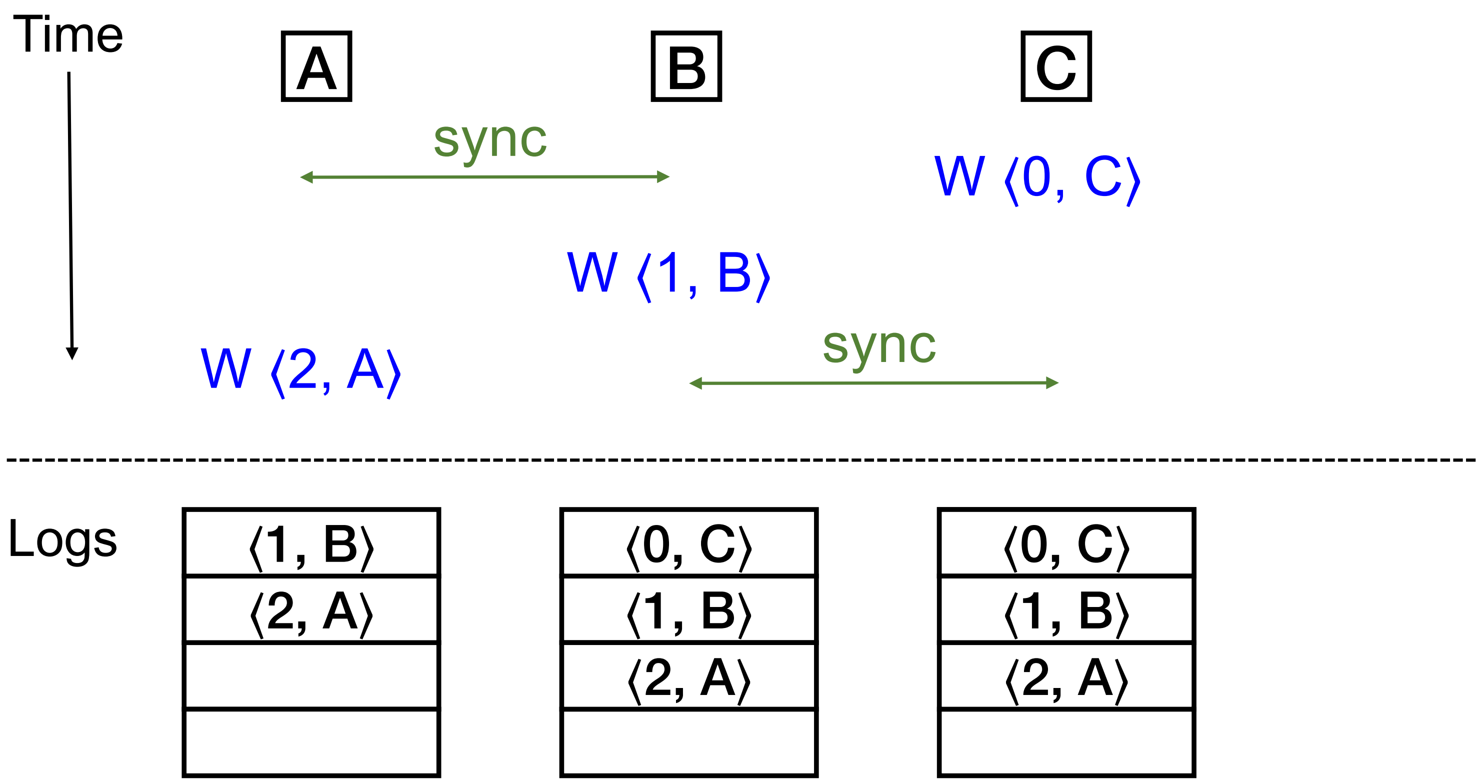
⟨ local TS, node-id⟩

Ex: Disagreement on tentative writes



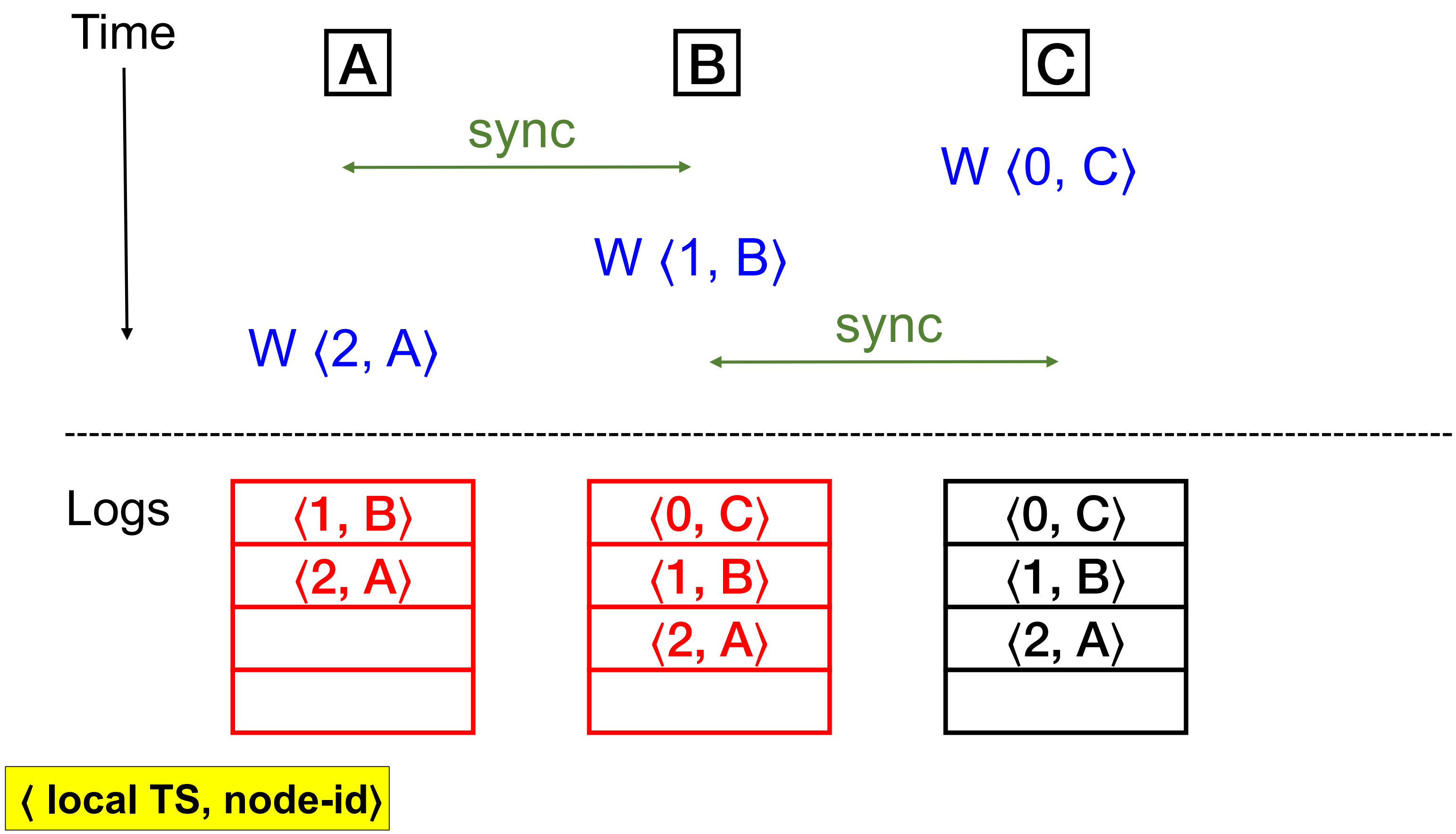
⟨ local TS, node-id⟩

Ex: Disagreement on tentative writes

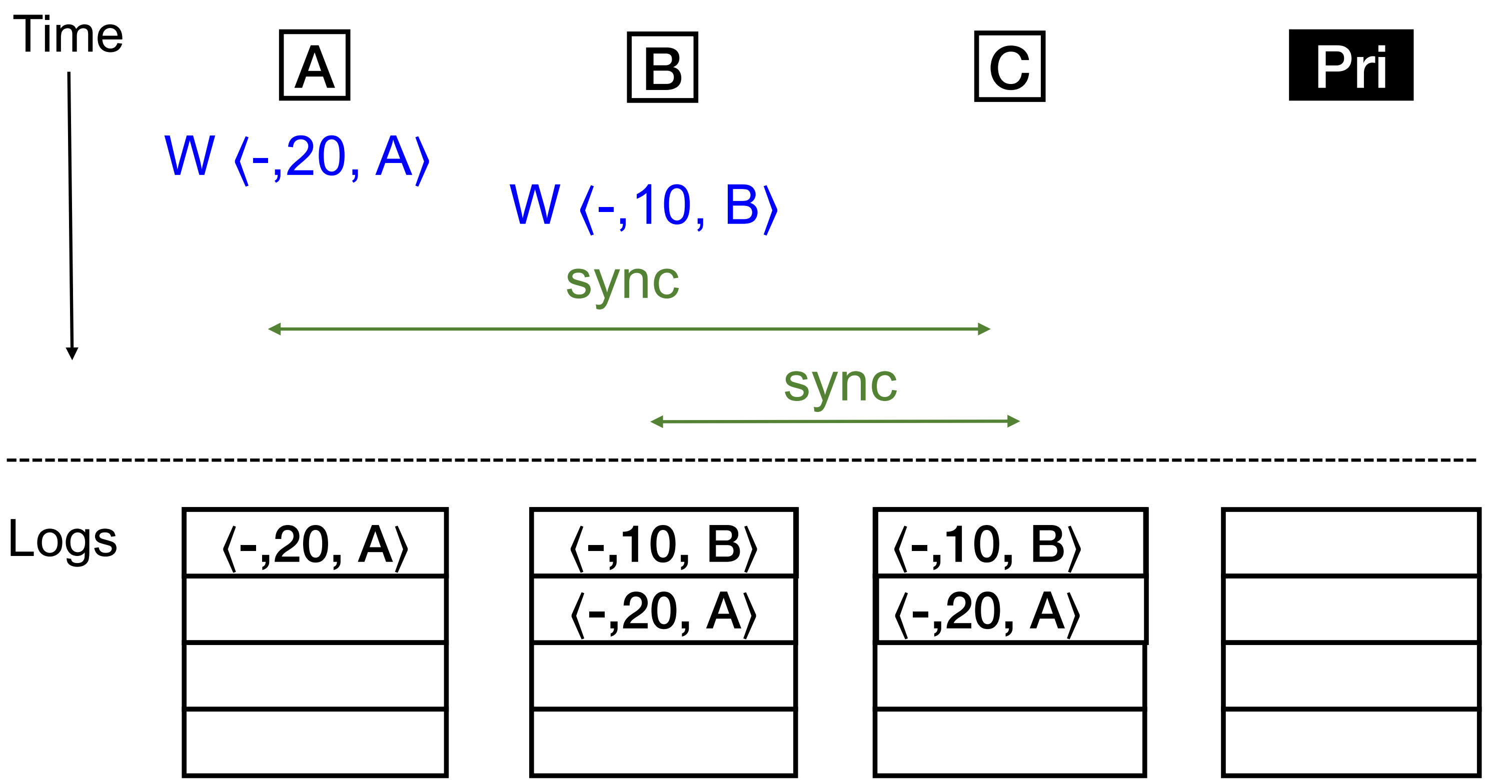


⟨ local TS, node-id⟩

Ex: Disagreement on tentative writes

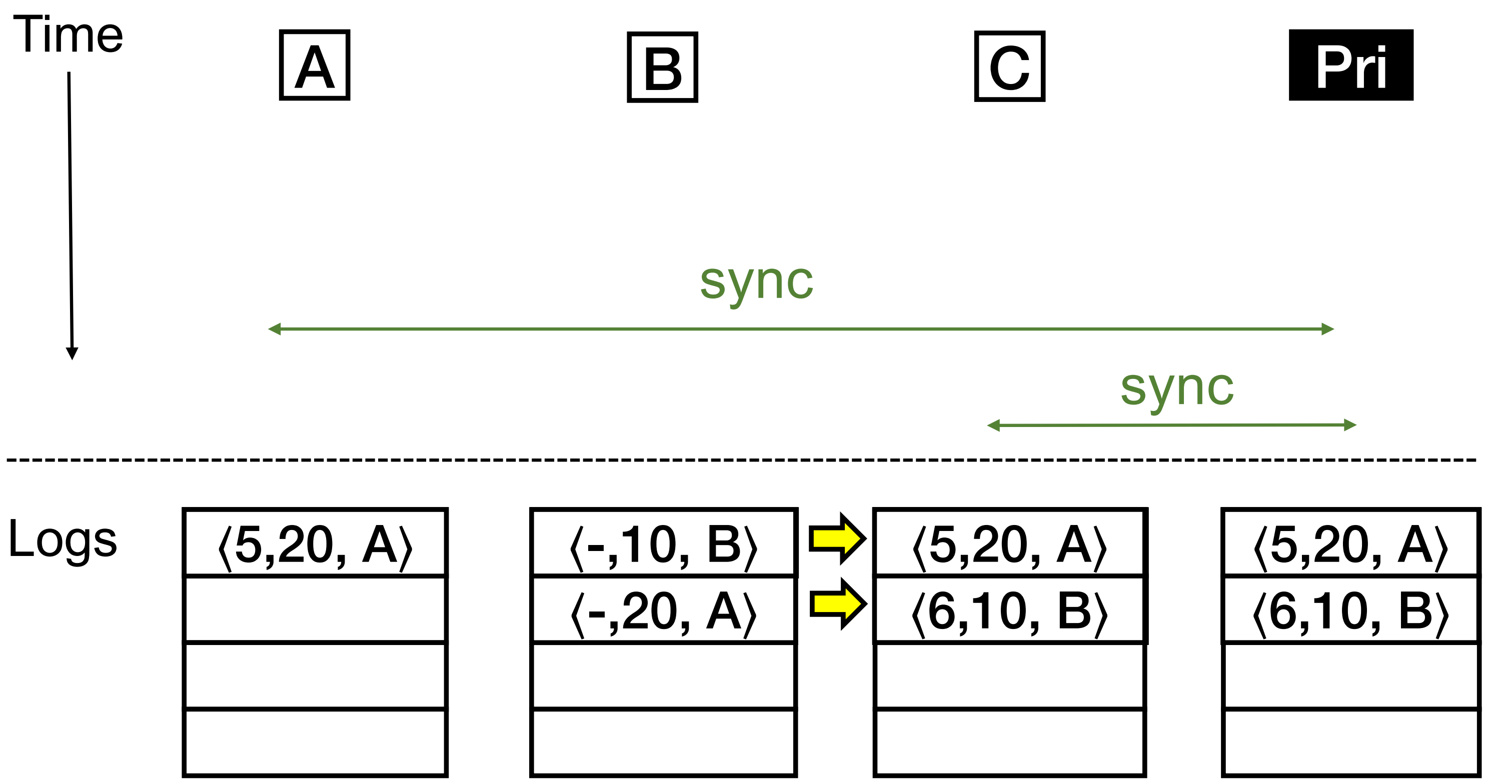


Tentative order ≠ commit order



⟨ CSN, local TS, node-id⟩

Tentative order ≠ commit order



⟨ CSN, local TS, node-id⟩

Primary commit order constraint

- Suppose user creates meeting, then deletes or changes it
  - What CSN order must these ops have?
    - Create first, then delete or modify
    - Must be true in every node's view of tentative log entries, too

- Rule: Primary's total write order must preserve causal order of writes. (But how?)

Primary preserves causal order

- Rule: Primary's total write order must preserve causal order of writes

- How?
  - Nodes sync full logs
    - If **A** → **B** then **A** is in all logs before **B**
  - Primary orders newly synced writes in tentative order
    - Primary will commit **A** and then commit **B**

Trimming the log

- When nodes receive new CSNs, can discard all committed log entries seen up to that point
  - Sync protocol → CSNs received in order

- Keep copy of whole database as of highest CSN

- Result: No need to keep years of log data

Let's step back

- *Is eventual consistency a useful idea?*
- Yes: we want fast writes to local copies iPhone sync, Dropbox, Dynamo, *...*

- *Are update conflicts a real problem?*
- Yes—all systems have some more or less awkward solution

Is Bayou's complexity warranted?

- Update functions, tentative ops, …
- Only critical if you want peer-to-peer sync
  - i.e. disconnected operation AND ad-hoc connectivity

What are Bayou's take-away ideas?

1. Eventual consistency: if updates stop, all replicas eventually the same
2. Update functions for automatic app-driven conflict resolution
3. Ordered update log is the real truth, not the DB
4. Use Lamport clocks: eventual consistency that respects causality