

Eventual Consistency: Bayou



CS 240: Computing Systems and Concurrency
Lecture 8

Marco Canini

Availability versus consistency

- Totally-Ordered Multicast **kept replicas consistent** but had **single points of failure**
 - **Not available** under failures
- (*Later*): **Distributed consensus algorithms**
 - **Strong consistency** (ops in same order everywhere)
 - But, **strong reachability 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
 - **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 when paper was written: Dawn of PDAs, laptops, tablets
 - 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
 - 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 mobile device
 - No master copy
- But phone has only **intermittent connectivity**
 - **Mobile data** expensive when roaming, **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** entire databases conflict
 - e.g., Mon 10am meeting in room 3 and Tuesday 11am meeting in room 4
 2. “Distinct record in each database changed” can **falsely conclude** no conflict
 - e.g., Mon 10–11am meeting in room 3 Doug attending, Mon 10-11am meeting in room 4 Doug attending, ...

Application-specific conflict resolution

- Want intelligence that **knows how to 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 update takes form:
 - “10 AM meeting, Room=305, CS-240 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=305, CS-240 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

- **X** syncs with **A**, then **B**
- **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 database contents
- If we obey the above, “sync” is a **simple merge** of two ordered lists


Agreeing on the update order

- **Timestamp:** \langle local timestamp T , originating node ID \rangle
- 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

- $\langle 701, A \rangle$: A asks for meeting **M1** at 10 AM, else 11 AM
- $\langle 770, B \rangle$: 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

- $\langle 701, A \rangle$: A asks for meeting **M1** at 10 AM, else 11 AM
- $\langle 770, B \rangle$: 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?

- $\langle 701, A \rangle$: **A** asks for meeting **M1** at 10 AM, else 11 AM
- $\langle 700, B \rangle$: **Delete update** $\langle 701, A \rangle$
 - 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 $E1 \rightarrow E2$ then $TS(E1) < TS(E2)$

Lamport clocks respect causality

- $\langle 701, A \rangle$: A asks for meeting M1 at 10 AM, else 11 AM
- ~~$\langle 700, B \rangle$: Delete update $\langle 701, A \rangle$~~
- $\langle 706, B \rangle$: Delete update $\langle 701, A \rangle$

- With Lamport clocks:
 - When A sends $\langle 701, A \rangle$, it includes its clock, T (> 701)
 - When B receives $\langle 701, A \rangle$, it updates its clock to $T' > T$
 - When B creates the delete, it timestamps it with 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 proposal:** Update $\langle 10, A \rangle$ is **committed** when **all nodes** have seen all updates with $TS \leq 10$
- 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 $< \langle 10, A \rangle$
 - So $\langle 10, A \rangle$ is committed
- Why doesn't Bayou do this?
 - A server that **remains disconnected** could prevent writes from committing
 - 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** = $\langle \text{CSN, local TS, node-id} \rangle$

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

How Bayou commits writes (2)

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

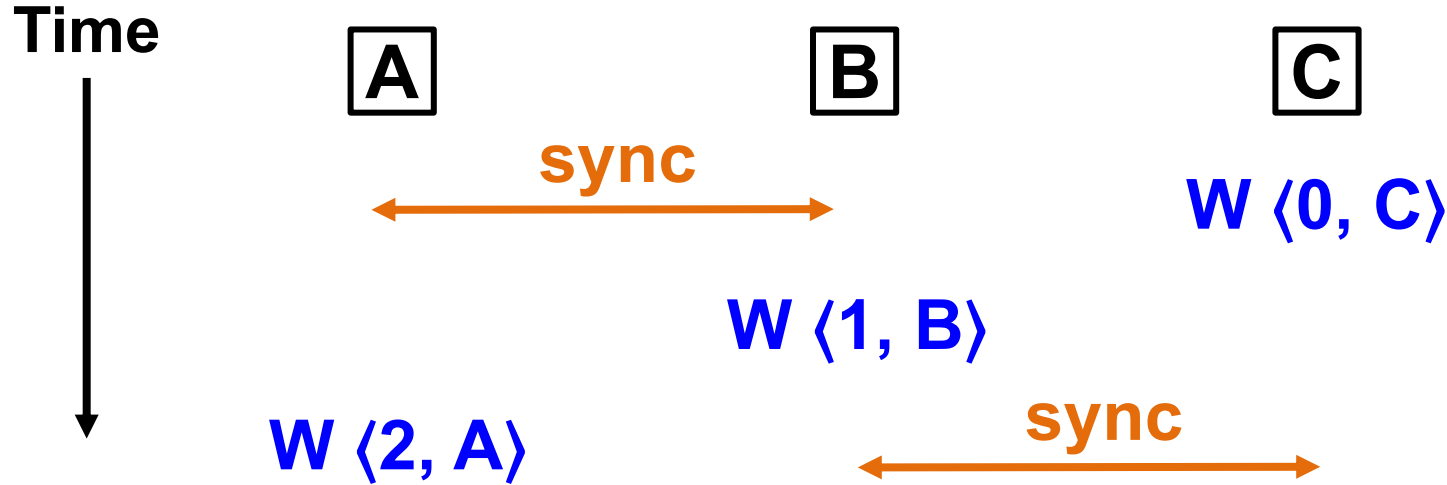
Committed vs. tentative writes

- 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** these disagreements permanently

Scenario 1: nodes that have synced disagree



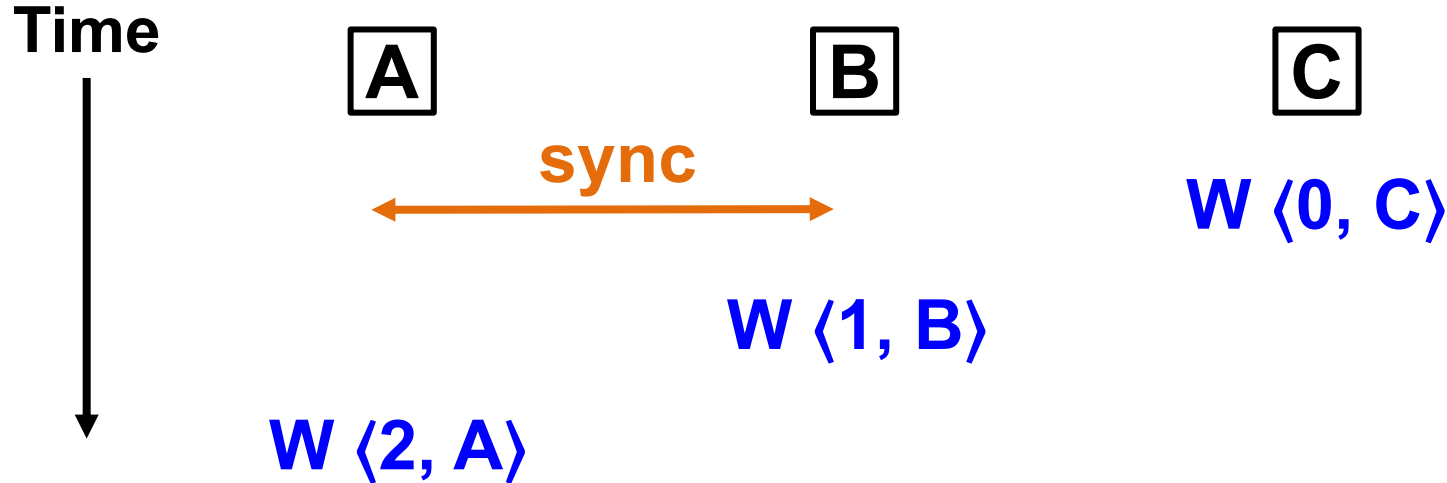
Logs

?
?
?
?

?
?
?
?

?
?
?
?

Example: Disagreement on tentative writes



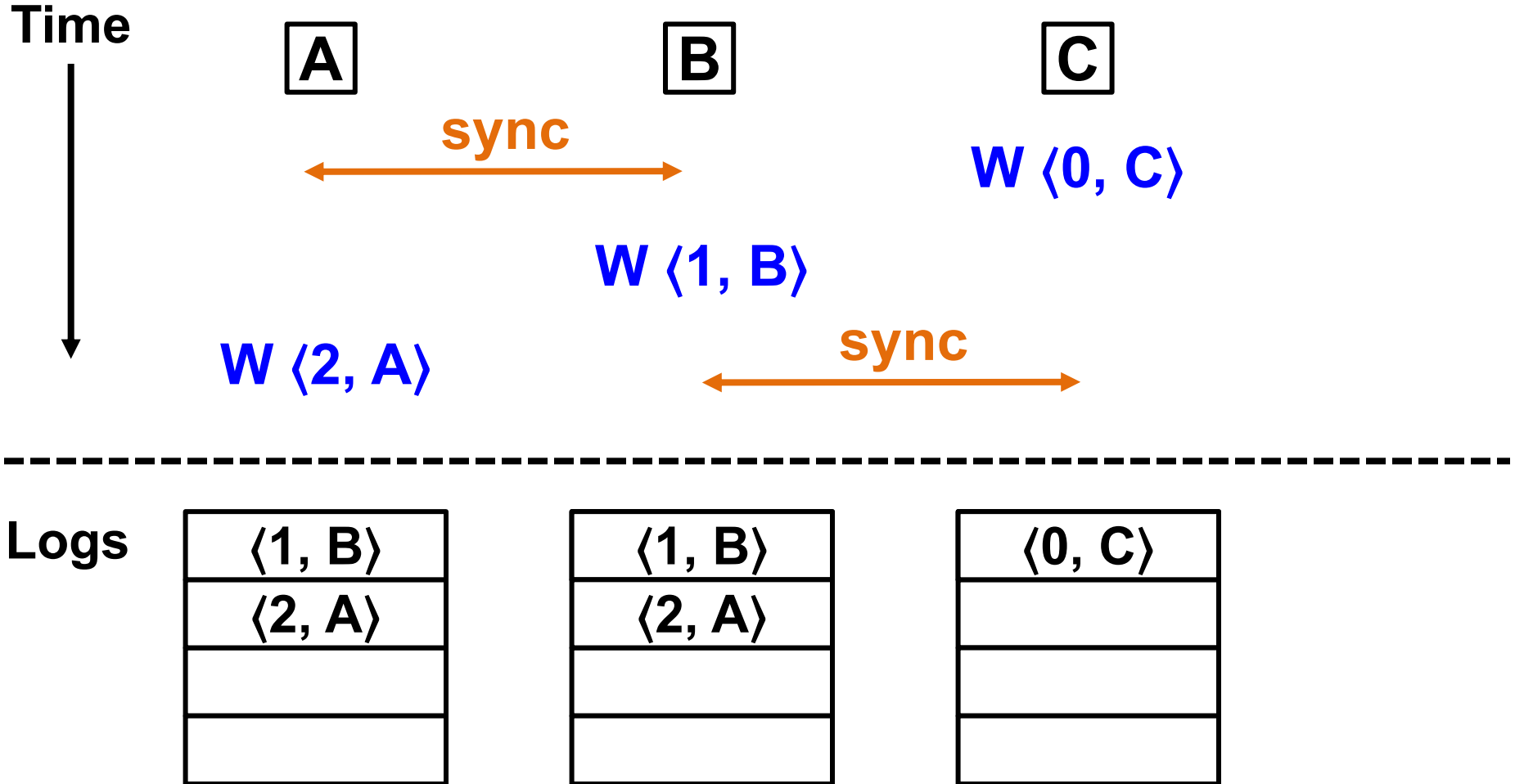
Logs

$\langle 2, A \rangle$

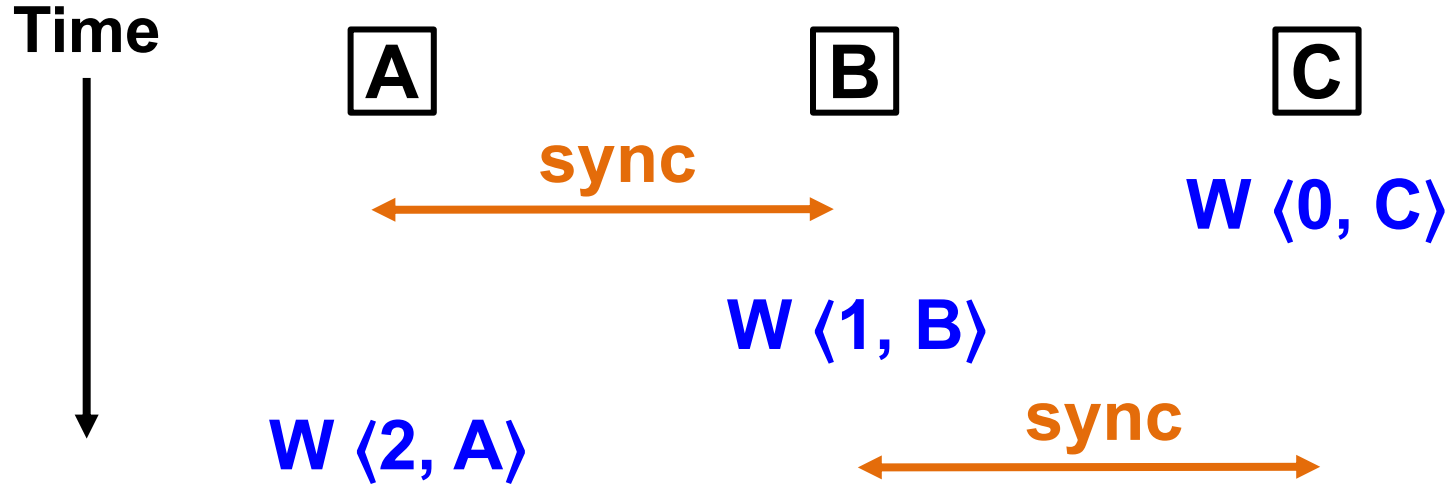
$\langle 1, B \rangle$

$\langle 0, C \rangle$

Example: Disagreement on tentative writes



Example: Disagreement on tentative writes



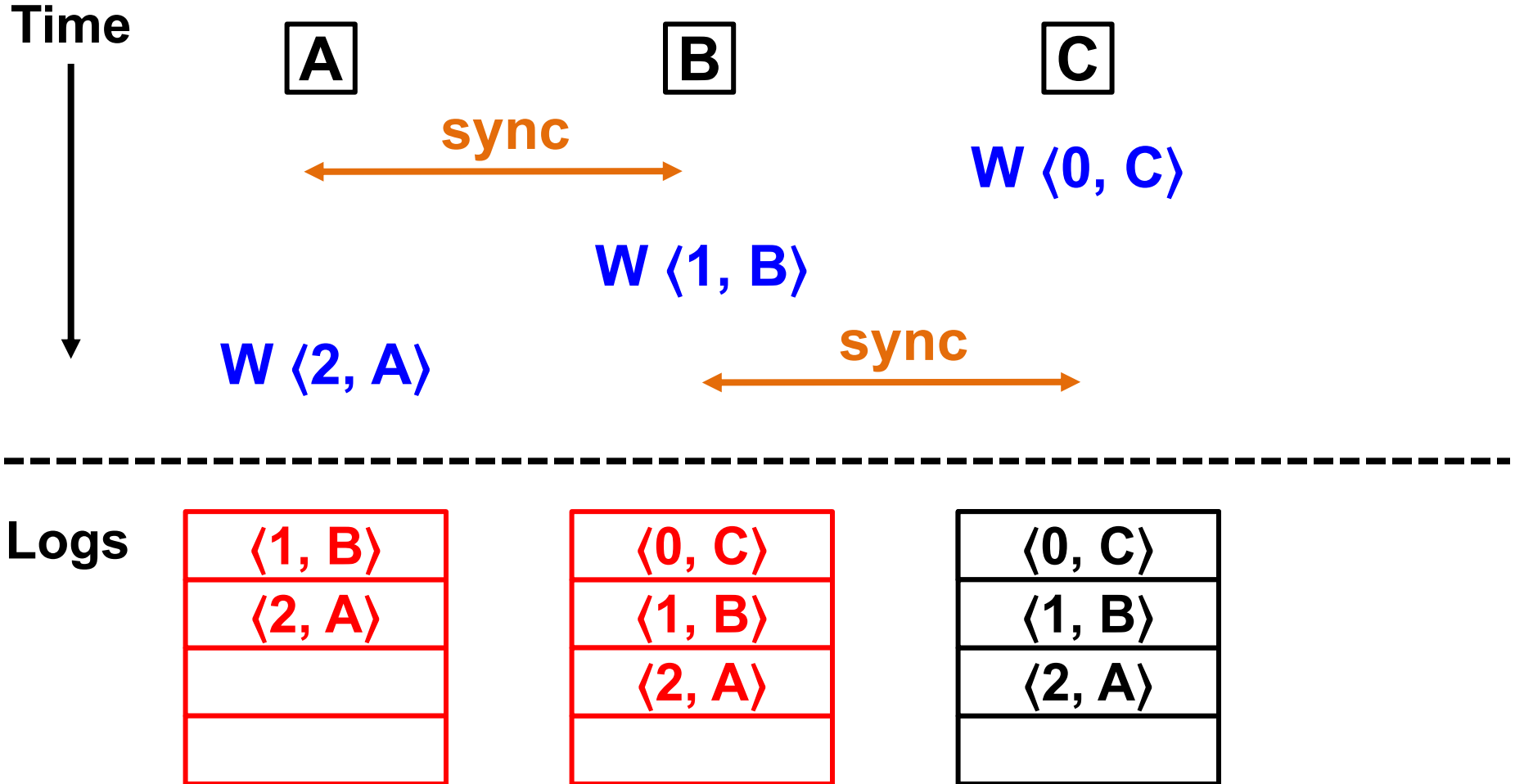
Logs

$\langle 1, B \rangle$
$\langle 2, A \rangle$

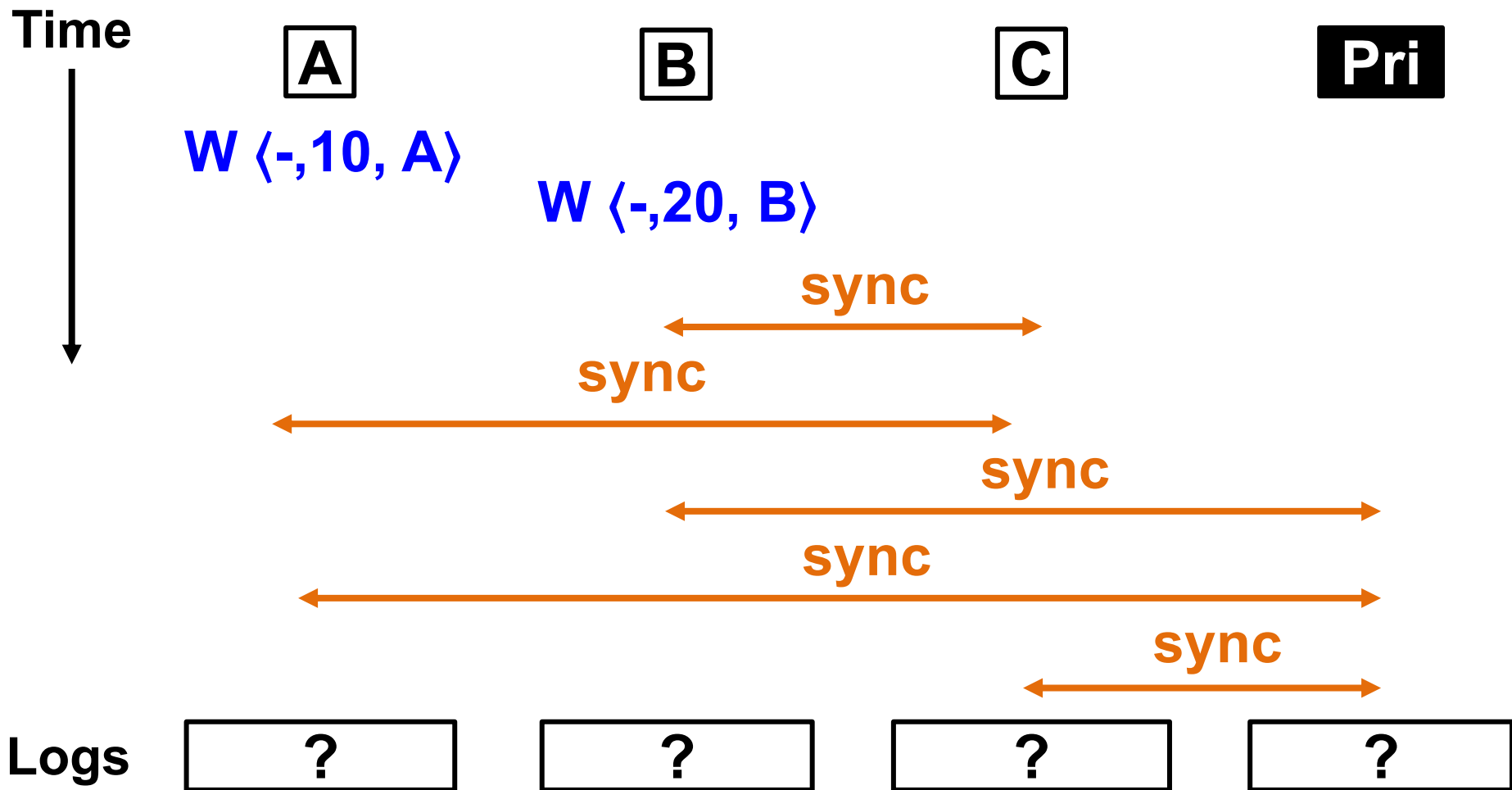
$\langle 0, C \rangle$
$\langle 1, B \rangle$
$\langle 2, A \rangle$

$\langle 0, C \rangle$
$\langle 1, B \rangle$
$\langle 2, A \rangle$

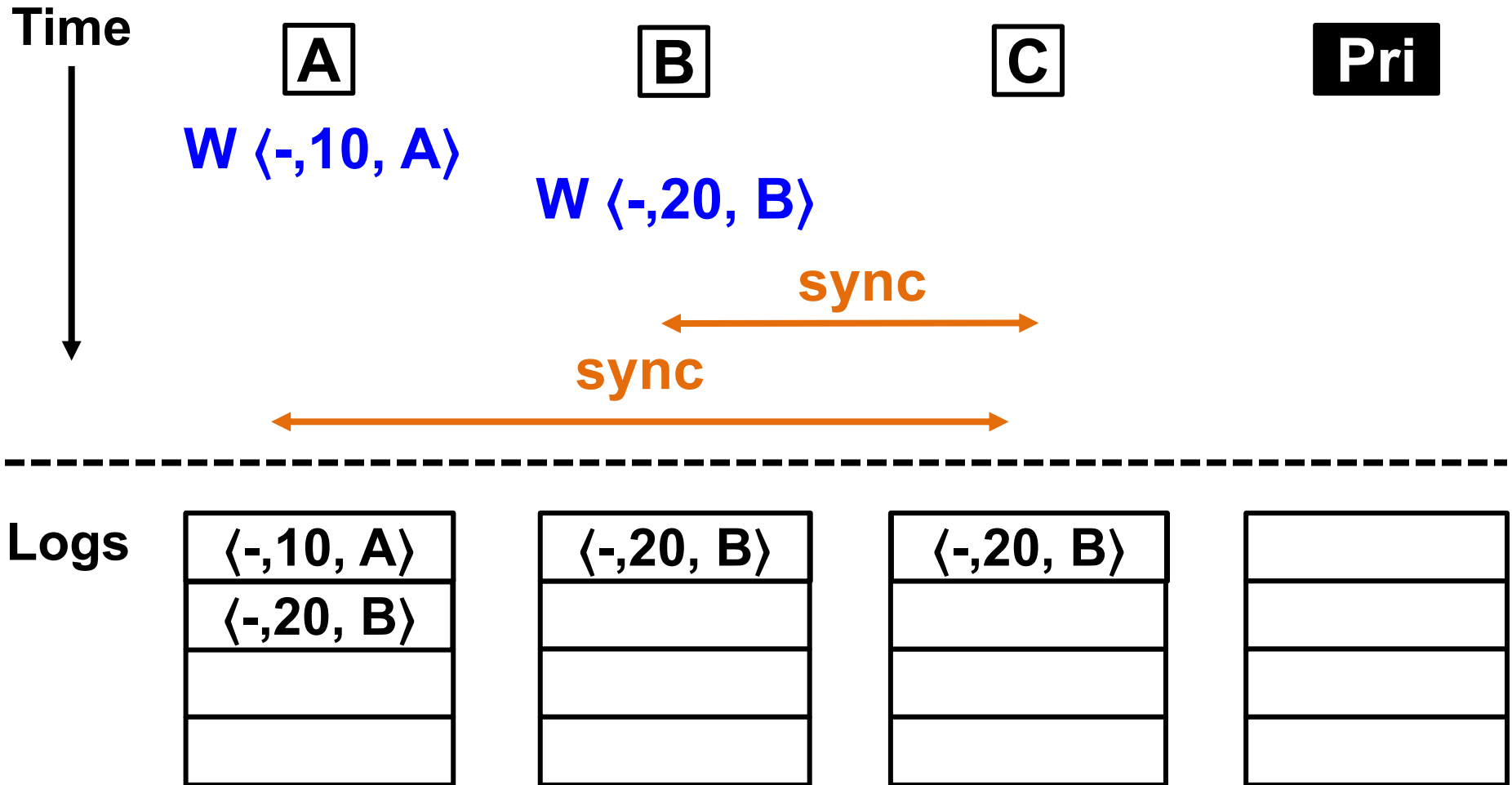
Example: Disagreement on tentative writes



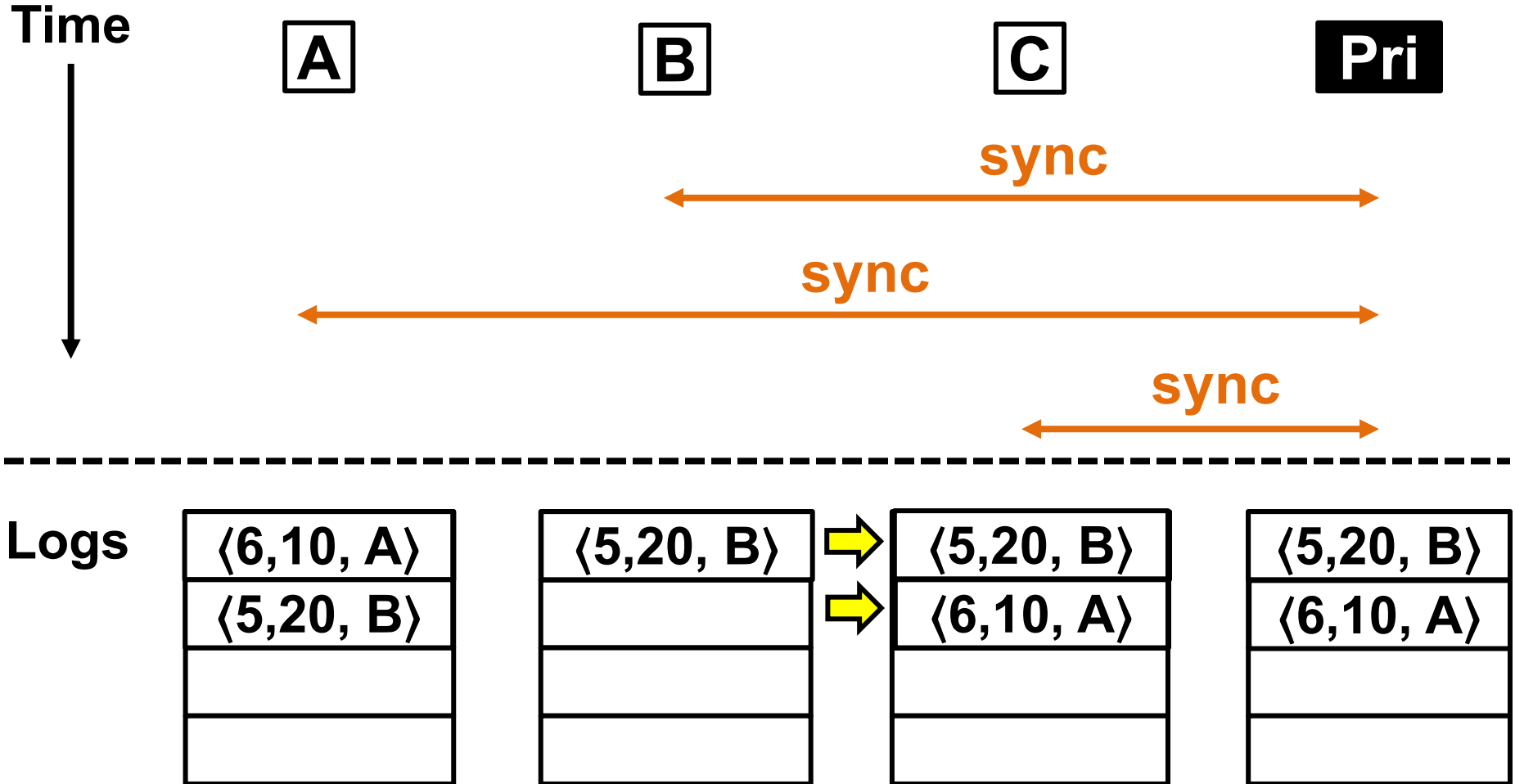
Scenario 2: tentative order changes after commit



Tentative order \neq commit order



Tentative order \neq commit order



Primary commit order constraint

- Suppose a user **creates meeting**, then decides to **delete or change 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
 - Q: 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
 - Update 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:** people 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.* both **disconnected operation and ad-hoc connectivity**
- Only tolerable if humans are main consumers of data
 - Otherwise you can sync through a central server
 - Or read locally but send updates through a master

What are Bayou's take-away ideas?

1. **Eventual consistency**, eventually if updates stop, all replicas are the same
2. **Update functions** for automatic application-driven conflict resolution
3. **Ordered update log** is the real truth, not the DB
4. Application of **Lamport clocks** for causal consistency