

Eventual Consistency: Bayou



جامعة الملك عبد الله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency Lecture 13

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.
Selected content adapted from B. Karp, R. Morris.

Availability versus consistency

- NFS and 2PC all had single points of failure
 - **Not available** under failures
- Distributed consensus algorithms allow **view-change** to elect primary
 - Strong consistency model
 - 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 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 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


LAPTOPS...WE STARTED IT ALL



In 1983, we introduced

the world's first laptop computer. We went on to engineer built-in software,



easy-to-read screens  and hard drives in notebook-size PCs.

AND NOW... THE NEW TANDY® 2810 HD NOTEBOOK PC,

286 Power

The 80C286 micro-processor runs at 16 MHz for speed-intensive applications like Microsoft® Windows.

VGA Graphics

Brilliant clarity with 640 x 480 graphics and a sharp 16:12 gray scale.

Built-In Hard Drive

20 megabytes of internal storage for rapid access, plus a 3.5" 1.44MB floppy drive.

MS-DOS® 4.01

The latest version of MS-DOS (4.01) comes already installed on the built-in hard drive.

DeskMate® Interface

The DeskMate Graphical User Interface with ten applications is installed on the hard drive for instant-on ease of use.

Resume Mode

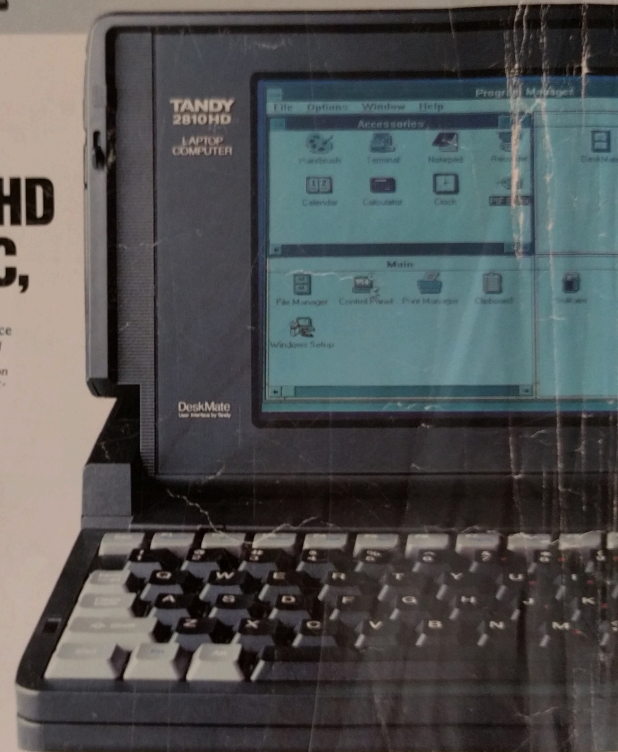
Lets you shut off and come back right where you left off—also shuts down automatically to save battery life.

External Support

Attach a 101-key keyboard, a VGA color monitor, a printer, an external floppy drive and more.

1MB Memory

Expandable to five megabytes.



DESKTOP PERFORMANCE IN A 6.7-lb. PORTABLE

Continuing our tradition of innovation, the Tandy 2810 HD is a lightweight laptop for heavy use—at the office, at home, or on the road. With extremely durable construction, it's built for travel—but it can also support a full-size keyboard and monitor for true desktop power. AT® compatibility, stunning VGA graphics and DeskMate® productivity software. Only at Radio Shack. Again.

Radio Shack
AMERICA'S
TECHNOLOGY
STORE™

Radio Shack is a division of Tandy Corporation. Microsoft and MS-DOS licensed from Microsoft Corp. AT/Reg. TM IBM Corp.

Circle 224 on Reader Service Card

What's wrong with a central server?

- Want my calendar on a disconnected mobile phone
 - *i.e.*, each user wants database replicated on her mobile device
 - No master copy
- 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 conflict, *i.e.*, two concurrent meetings?
 - iPhone sync keeps both meetings
 - Want to do better: **automatic conflict resolution**

Automatic conflict resolution

- 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
 2. “Distinct record in each database changed” can **falsely conclude** no conflict

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

What's in a write?

- 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”

Want all nodes to execute **same instructions in same order, eventually**

Problem

- 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 to DB replicas

Insight: Total ordering of 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**
- So, in the 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**

Is update order consistent with wall clock?

- $\langle 701, A \rangle$: 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

- Maybe **B** asked first by the wall clock
 - But because of clock skew, **A's** meeting has **lower timestamp**, so gets priority

- No, **not “externally consistent”**

Does update order respect causality?

- Suppose **another example**:
- $\langle 701, A \rangle$: **A** asks for meeting **M1** at 10 AM, else 11 AM
- $\langle 700, B \rangle$: **Delete update** $\langle 701, A \rangle$
 - **B's** clock was **slow**
- Now **delete will be ordered before add**

Lamport logical clocks respect causality

- Want event timestamps so that **if** a node observes **E1** then generates **E2**, then **$TS(E1) < TS(E2)$**
- **T_{max}** = highest TS seen from any node (including self)
- $T = \max(T_{\max} + 1, \text{wall-clock time})$, to generate TS
- Recall properties:
 - **E1** then **E2** on same node $\rightarrow TS(E1) < TS(E2)$
 - But $TS(E1) < TS(E2)$ **does not imply** that **E1** necessarily came before **E2**

Lamport clocks solve causality problem

- $\langle 701, A \rangle$: A asks for meeting M1 at 10 AM, else 11 AM
- ~~$\langle 700, B \rangle$: Delete update $\langle 701, A \rangle$~~
- $\langle 702, B \rangle$: Delete update $\langle 701, A \rangle$



- Now when **B** sees $\langle 701, A \rangle$ it sets $T_{\max} \leftarrow 701$
 - So it will then generate a **delete update** with a **later timestamp**

Timestamps for write ordering: Limitations

- Ordering by timestamp **arbitrarily constrains order**
 - **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**

Problem: How can we allow committing a tentative entry, so we can **trim logs** and **have meetings**

Fully decentralized commit

- **Strawman proposal:** Update $\langle 10, A \rangle$ is **stable** if **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 stable
- Why doesn't Bayou do this?
 - A server that **remains disconnected** could prevent writes from stabilizing
 - So **many writes** may be **rolled back** on re-connect

Criteria for committing writes

- For log entry **X** to be committed, all servers must agree:
 1. On the **total order** of all **previous** committed writes
 2. That **X** is **next** in the total order
 3. That all **uncommitted** entries are “**after**” **X**

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
 - **Uncommitted** writes come **after** all **committed writes**

Showing users that writes are committed

- **Still not safe** to show users that an appointment request has committed!
- Entire **log up to newly committed write** must be **committed**
 - Else there might be **earlier committed write** a node doesn't know about!
 - And upon learning about it, would have to **re-run conflict resolution**
- Bayou propagates writes between nodes to enforce this invariant, *i.e.* Bayou **propagates writes in CSN order**

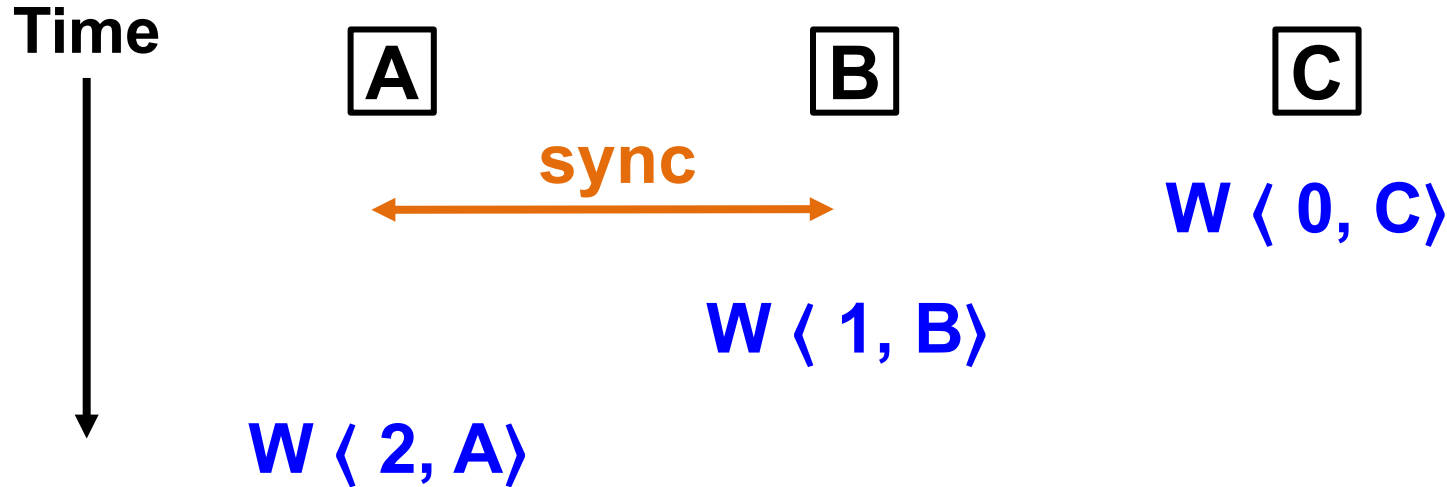
Committed vs. tentative writes

- Suppose a node has seen every CSN up to a write, as guaranteed by propagation protocol
 - Can then **show user** the write has **committed**
- **Slow/disconnected** node **cannot prevent commits!**
 - Primary replica allocates CSNs; global order of writes may not reflect real-time write times

Tentative writes

- What about **tentative writes**, though—how do they behave, as seen by users?
- Two nodes may **disagree** on meaning of **tentative (uncommitted) writes**
 - Even if those two nodes have **synced** with each other!
 - Only **CSNs** from primary replica can **resolve** these disagreements permanently

Example: Disagreement on tentative writes



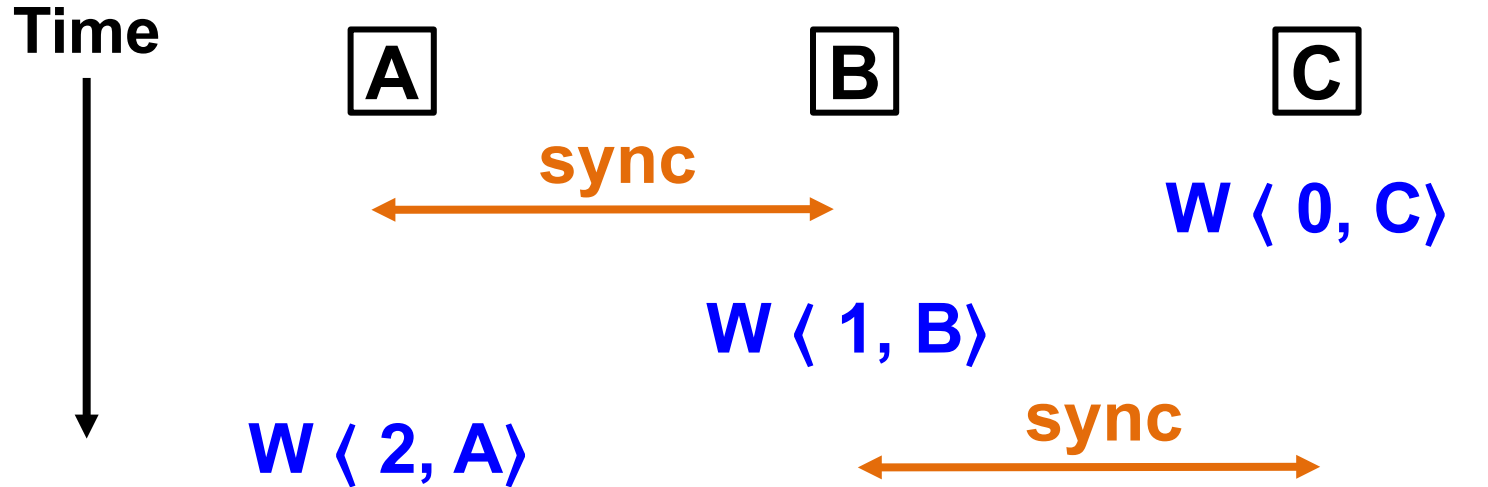
Logs

$\langle 2, A \rangle$

$\langle 1, B \rangle$

$\langle 0, C \rangle$

Example: Disagreement on tentative writes



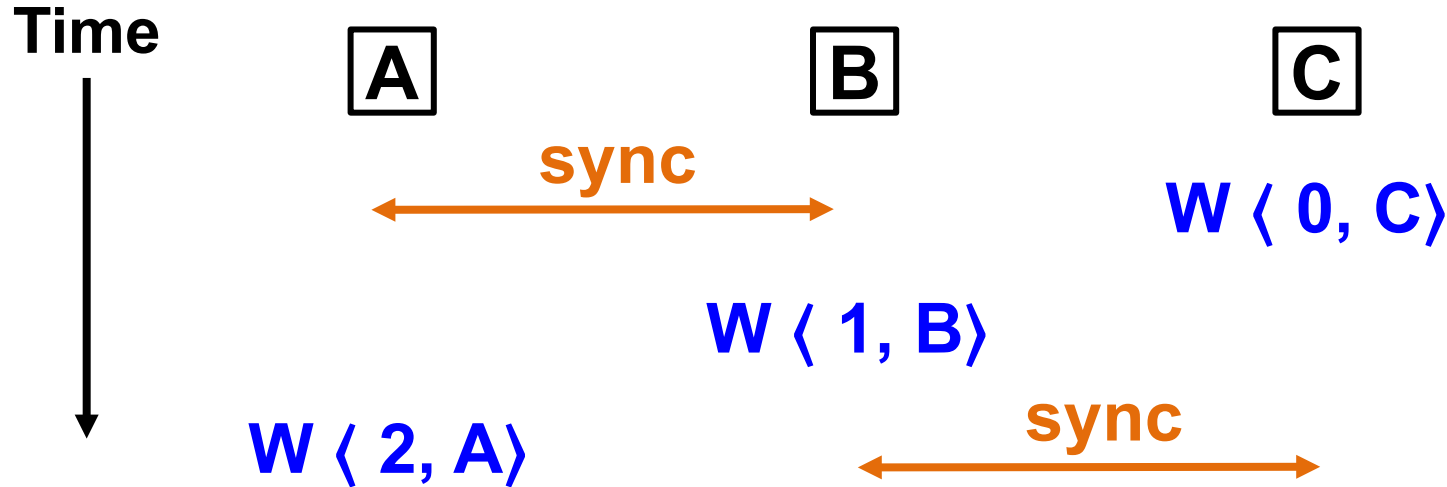
Logs

< 1, B >
< 2, A >

< 1, B >
< 2, A >

< 0, C >

Example: Disagreement on tentative writes



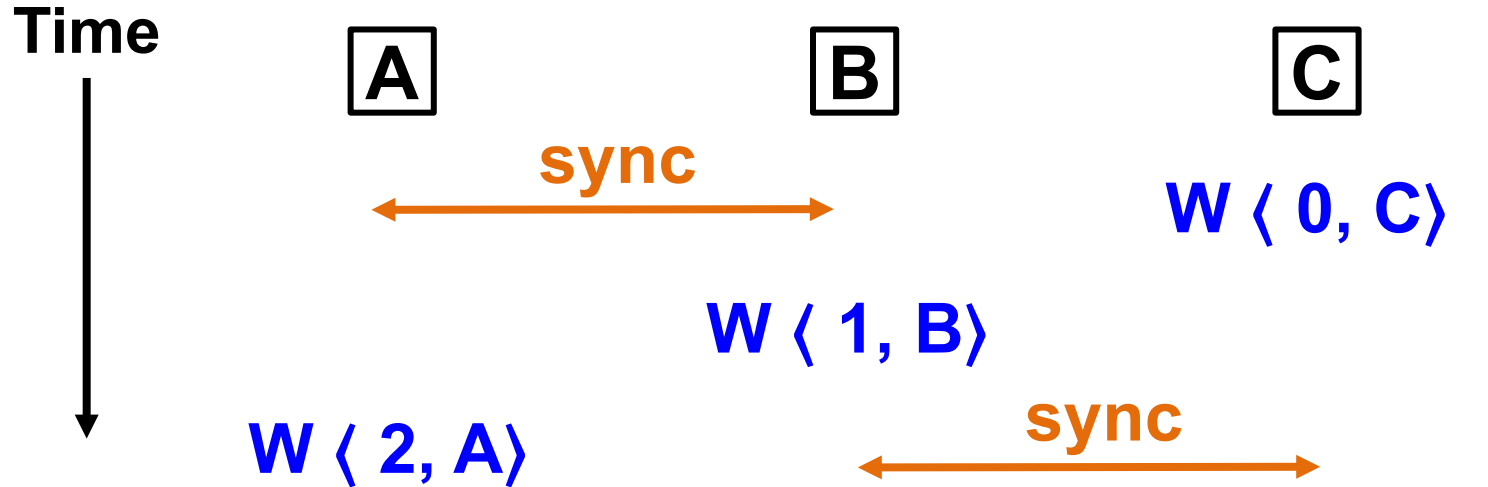
Logs

< 1, B >
< 2, A >

< 0, C >
< 1, B >
< 2, A >

< 0, C >
< 1, B >
< 2, A >

Example: Disagreement on tentative writes



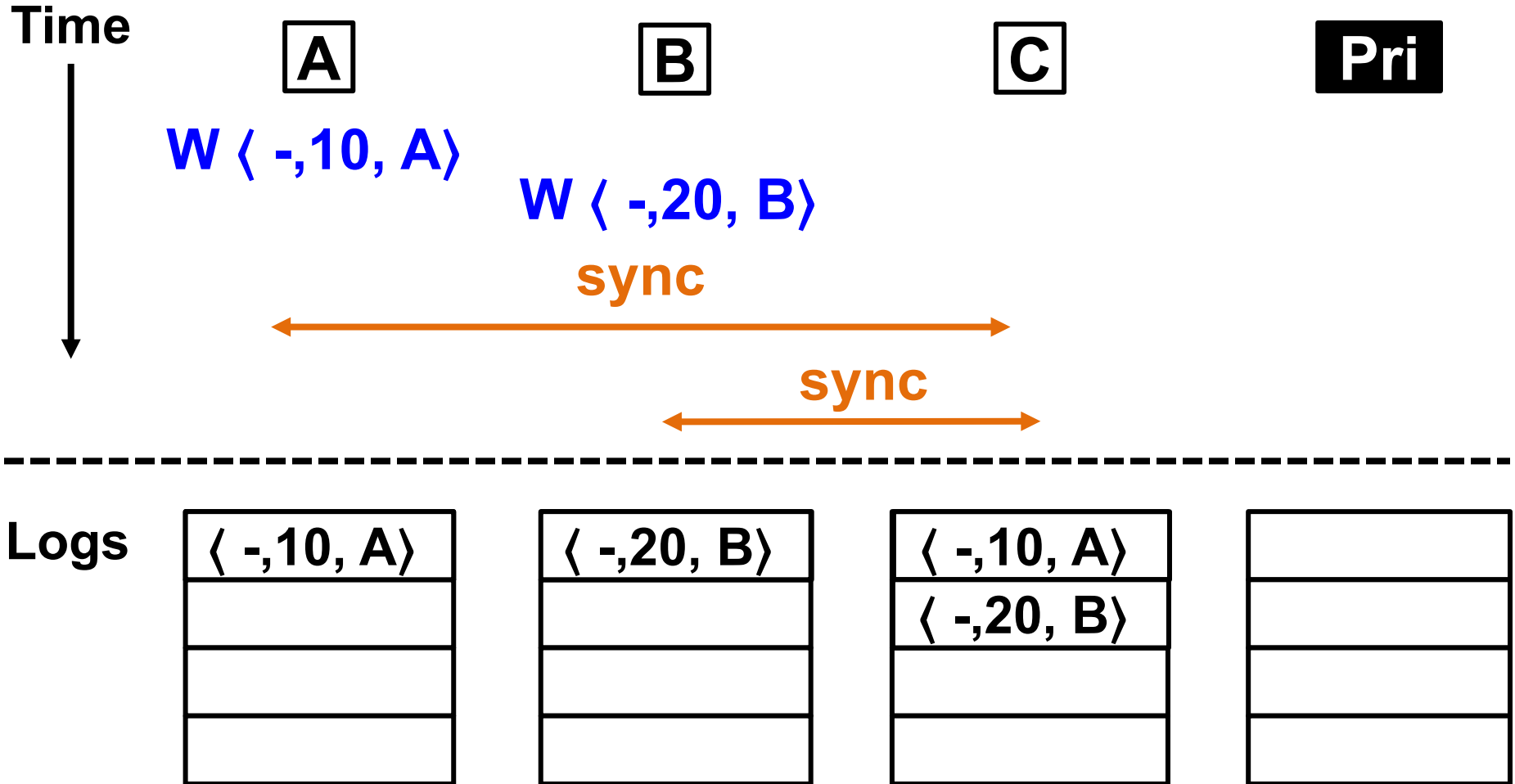
Logs

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

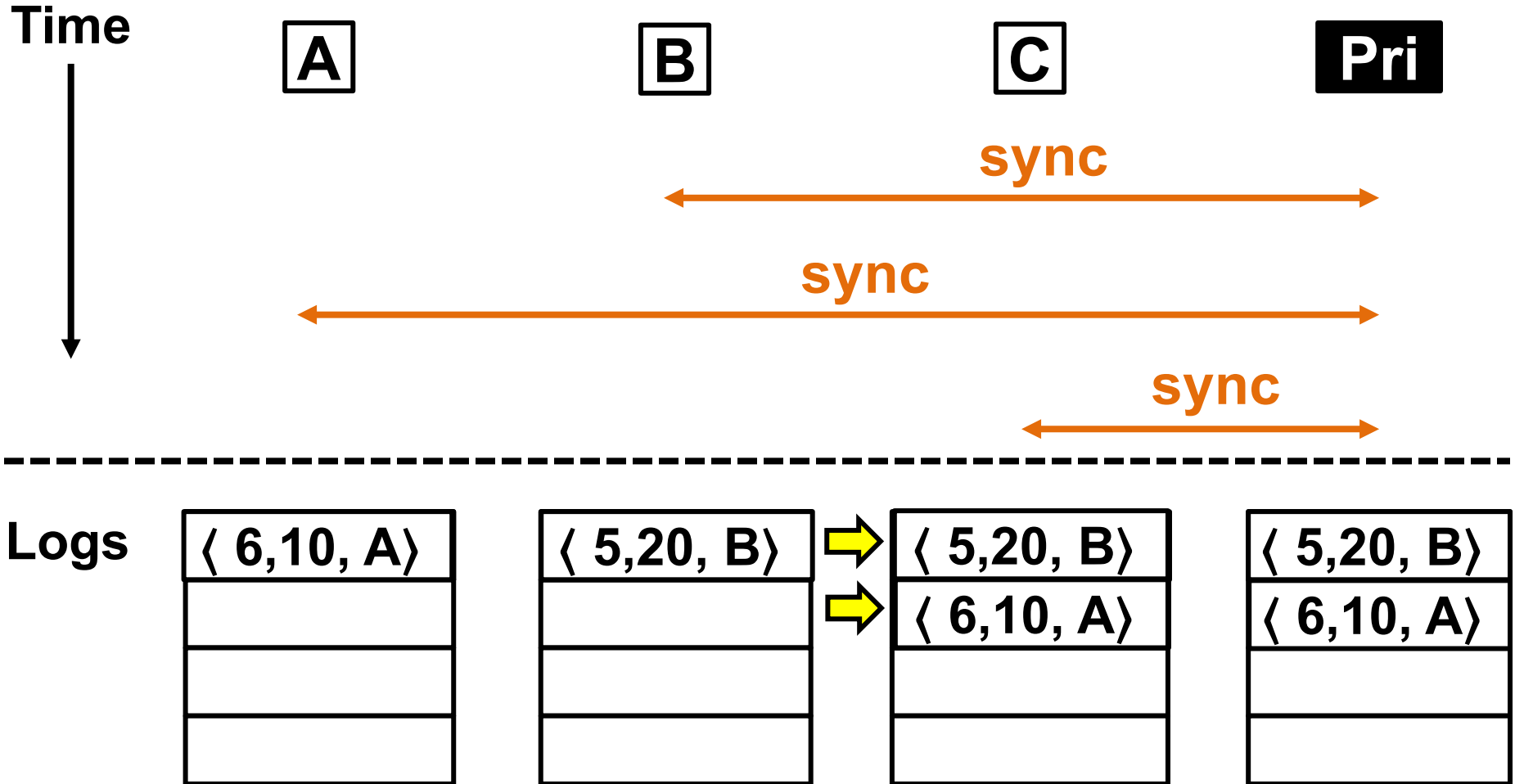
$\langle 0, C \rangle$
$\langle 1, B \rangle$
$\langle 2, A \rangle$

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

Tentative order \neq commit order



Tentative order \neq commit order



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**

Can primary commit writes in any order?

- 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 made **at each node**
 - Not necessarily order among different nodes' writes

Syncing with trimmed logs

- Suppose nodes discard all writes in log with CSNs
 - Just keep a copy of the **“stable” DB**, reflecting discarded entries

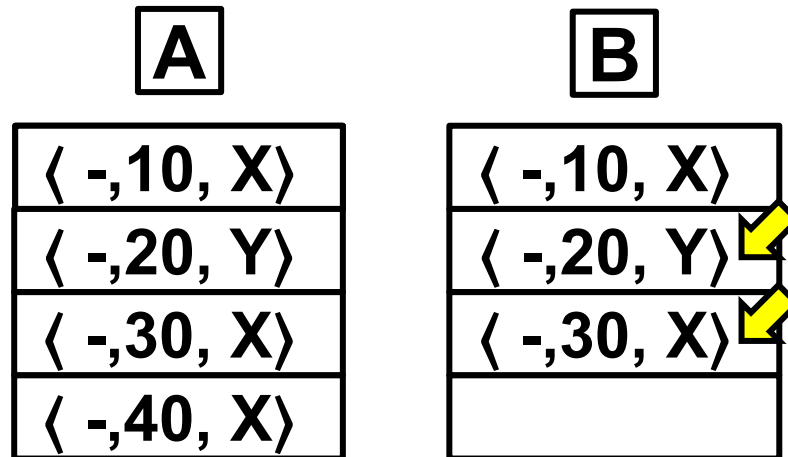
- **Cannot** receive writes that **conflict** with stable DB
 - Only could be if write had CSN less than a discarded CSN
 - **Already saw** all writes with lower CSNs in right order: if see them again, **can discard!**

Syncing with trimmed logs (2)

- To propagate to node **X**:
- If **X's** highest CSN **less than mine**,
 - Send **X** full stable DB; X uses that as starting point
 - **X can discard** all his **CSN** log entries
 - **X** plays his **tentative writes** into that DB
- If **X's** highest CSN **greater than mine**,
 - **X can ignore** my DB!

How to sync, quickly?

- What about **tentative updates**?



- B tells A: **highest local TS for each other node**

This is a **version vector** (“F” vector in Figure 4)

A’s F: [X:40, Y:20]

B’s F: [X:30, Y:20]

New server

- New server **Z** joins. Could it just start generating writes, e.g. $\langle -, 1, Z \rangle$?
 - And other nodes just start including **Z** in their version vectors?
- If **A** syncs to **B**, **A** has $\langle -, 10, Z \rangle$
 - But, **B has no Z** in its version vector
 - **A should pretend** B's version vector was $[Z:0, \dots]$

Server retirement

- We want to stop including **Z** in version vectors!
- **Z** sends update: $\langle -, ?, \mathbf{Z} \rangle$ “retiring”
 - If you see a retirement update, omit **Z** from VV
- **Problem:** How to deal with a VV that's missing **Z**?
 - A has log entries from **Z**, but B's VV has no **Z** entry
 - e.g. A has $\langle -, 25, \mathbf{Z} \rangle$, B's VV is just [A:20, B:21]
 - Maybe **Z** has **retired**, B knows, A does not
 - Maybe **Z** is **new**, A knows, B does not

Need a way to **disambiguate**

Bayou's retirement plan

- **Idea:** Z joins by contacting some server X
 - **New server identifier:** id now is $\langle T_z, X \rangle$
 - T_z is **X's logical clock** as of when Z joined
- X issues update $\langle -, T_z, X \rangle$ “new server Z”

Bayou's retirement plan

- Suppose Z's ID is $\langle 20, X \rangle$
 - A syncs to B
 - **A** has log entry from **Z**: $\langle -, 25, \langle 20, X \rangle \rangle$
 - **B's VV** has **no Z entry**
- One case: B's VV: $[X:10, \dots]$
 - $10 < 20$, so B hasn't yet seen X's "new server Z" update
- The other case: B's VV: $[X:30, \dots]$
 - $20 < 30$, so B once knew about Z, but then saw a retirement update

Let's step back

- *Is eventual consistency a useful idea?*
- **Yes:** people want **fast writes to local copies**
iPhone sync, Dropbox, **Dynamo**, & c.
- *Are update conflicts a real problem?*
- **Yes**—all systems have some more or less awkward solution

Is Bayou's complexity warranted?

- *i.e.* update function log, version vectors, 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. **Update functions** for automatic application-driven conflict resolution
2. **Ordered update log** is the real truth, not the DB
3. Application of **Lamport logical clocks** for causal consistency

Next topic:
Scaling Services: Key-Value Storage