SE3331-1 (2023 Fall)

Consistency models and eventual consistency

Xingda Wei, Yubin Xia

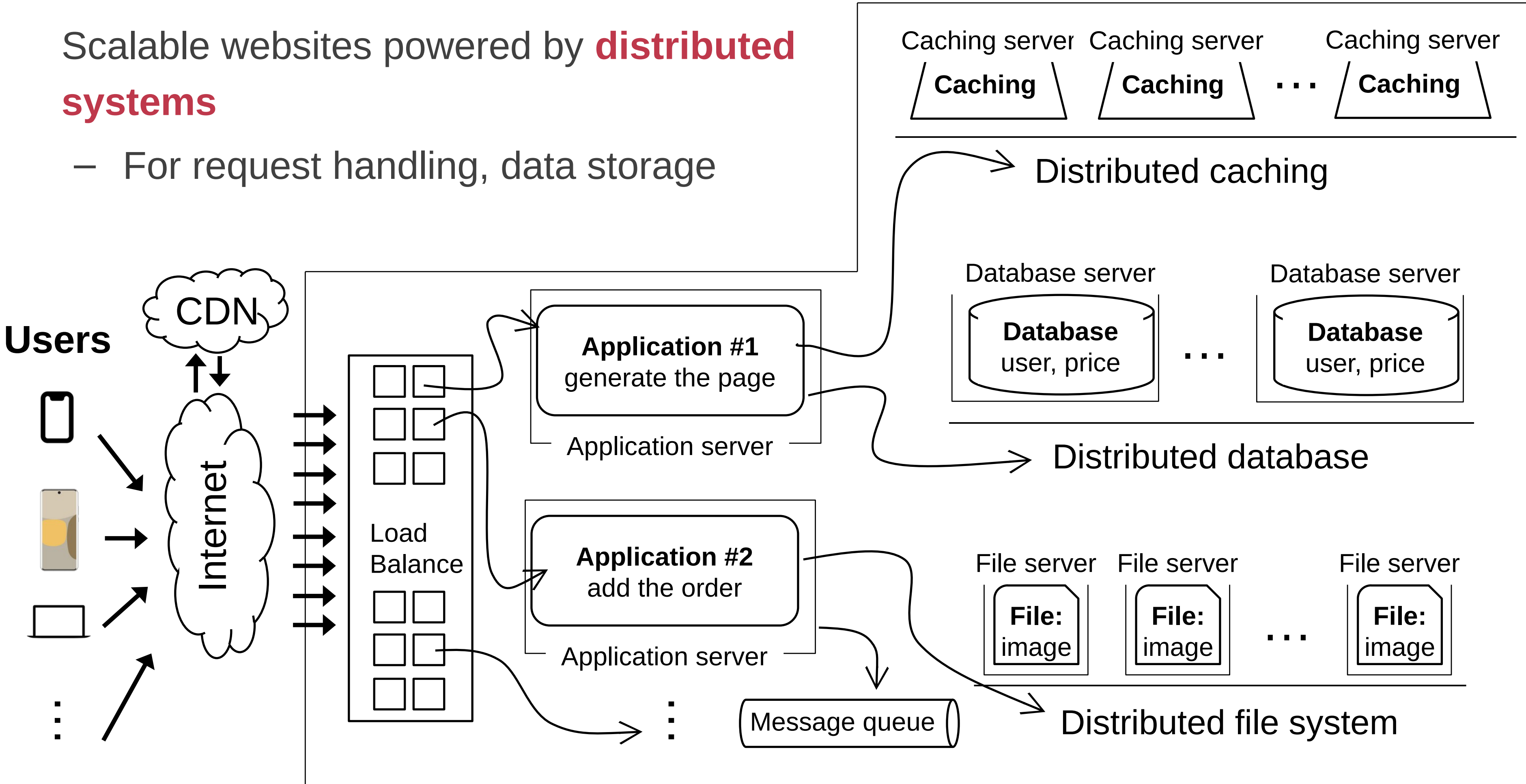
IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Credits: Rong Chen@IPADS, jinyang Li@NYU

Scalable websites overview

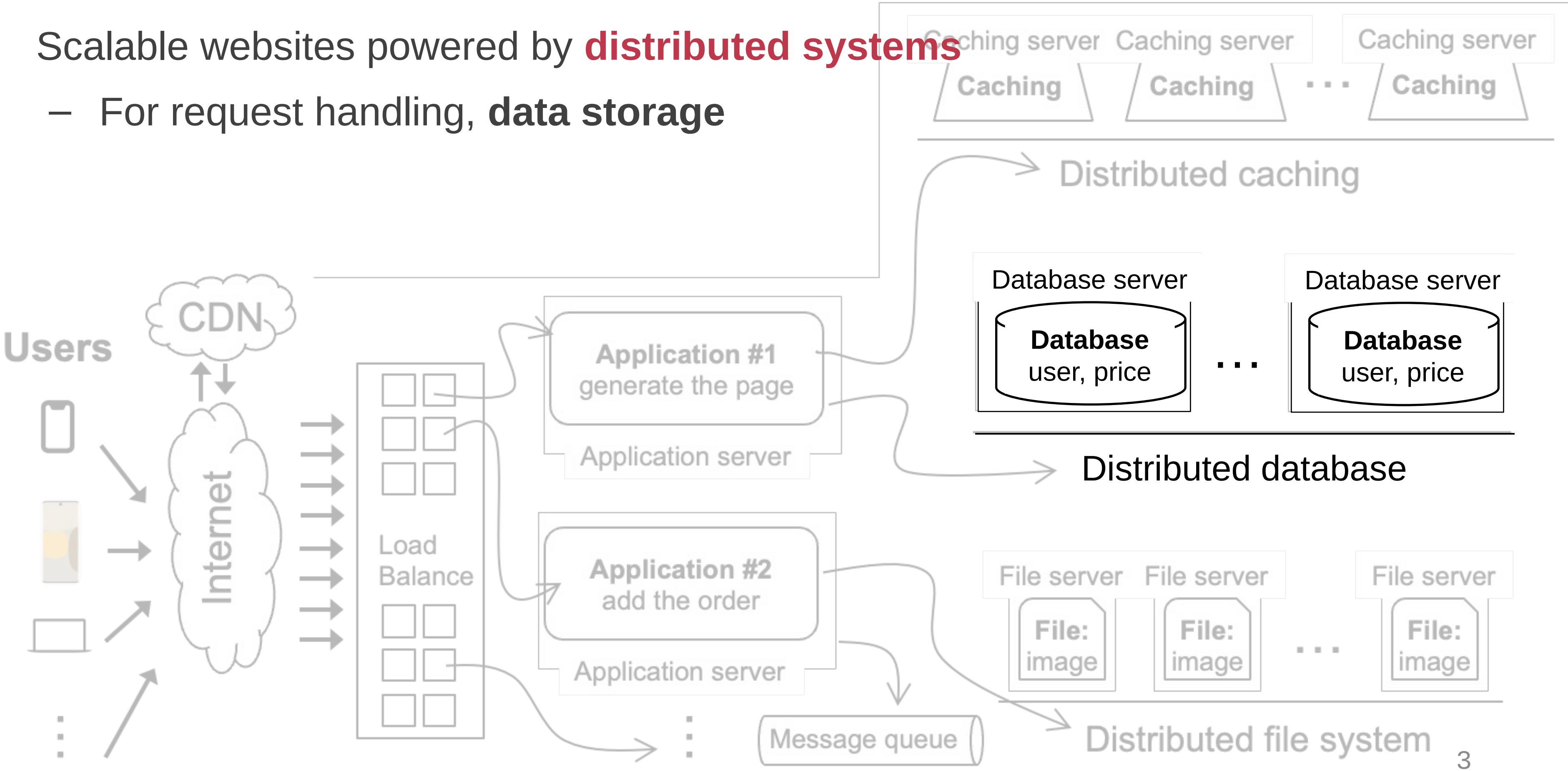
- Scalable websites powered by **distributed systems**
  - For request handling, data storage



Scalable websites overview

Scalable websites powered by **distributed systems**

- For request handling, **data storage**



Storage system is a very large topic

(Informally) A storage system is system to store & manage the data

Data can be

- File (Filesystem)
- Key-value pairs (Key-value store)
- More complex model, e.g., Graph. (Graph store)

Requirements considered by the storage system

Consistency

- Is the returned data correct?

Performance

- How fast is my data access?

Reliability

- Can my stored data tolerate failures?

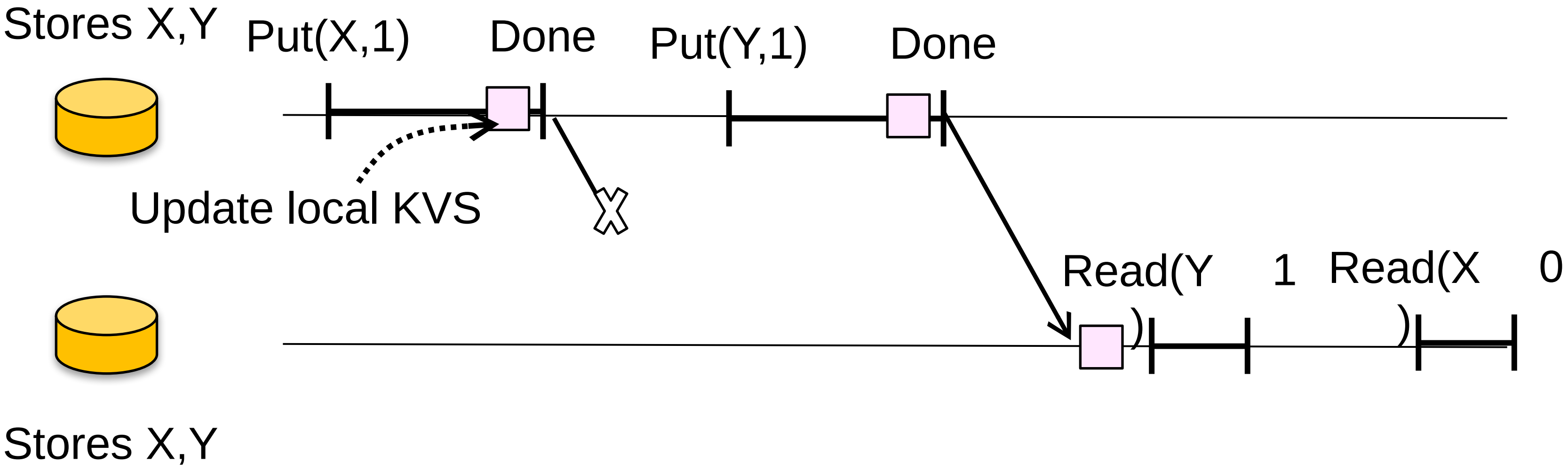
Review: replicated KVS, write sync no wait

Read: return the latest copy on the local KVS

Write: update the local KVS, sync with the others in background & return

Unexpected behavior can happen

- Suppose the data, X and Y are both 0 at all KVS



Summary: unexpected behavior of naïve replicated KVS

Though it is fast

- It has unexpected behavior, or usually called inconsistency happens

How can we write correct distributed programs w/ shared storage?

- The system imposes of **a consistency model** when operating the distributed data

Review: what is consistency model?

> **Consistency model** defines **rules** for the apparent order and visibility of updates, and it is a continuum with **tradeoffs**.
>
> \- Todd Lipcon

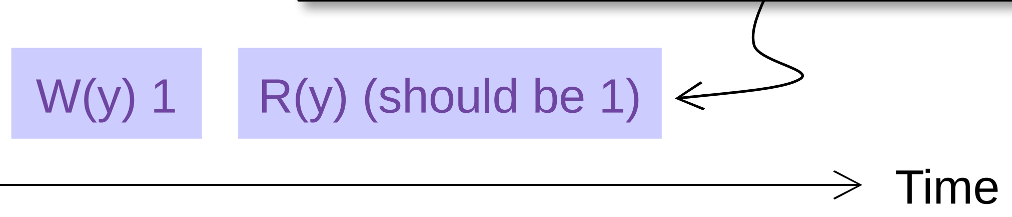
Review: what is consistency model?

> **Consistency model** defines **rules** for the apparent order and visibility of updates, and it is a continuum with **tradeoffs**.
>
> \- Todd Lipcon

Single object consistency is also called "coherence"

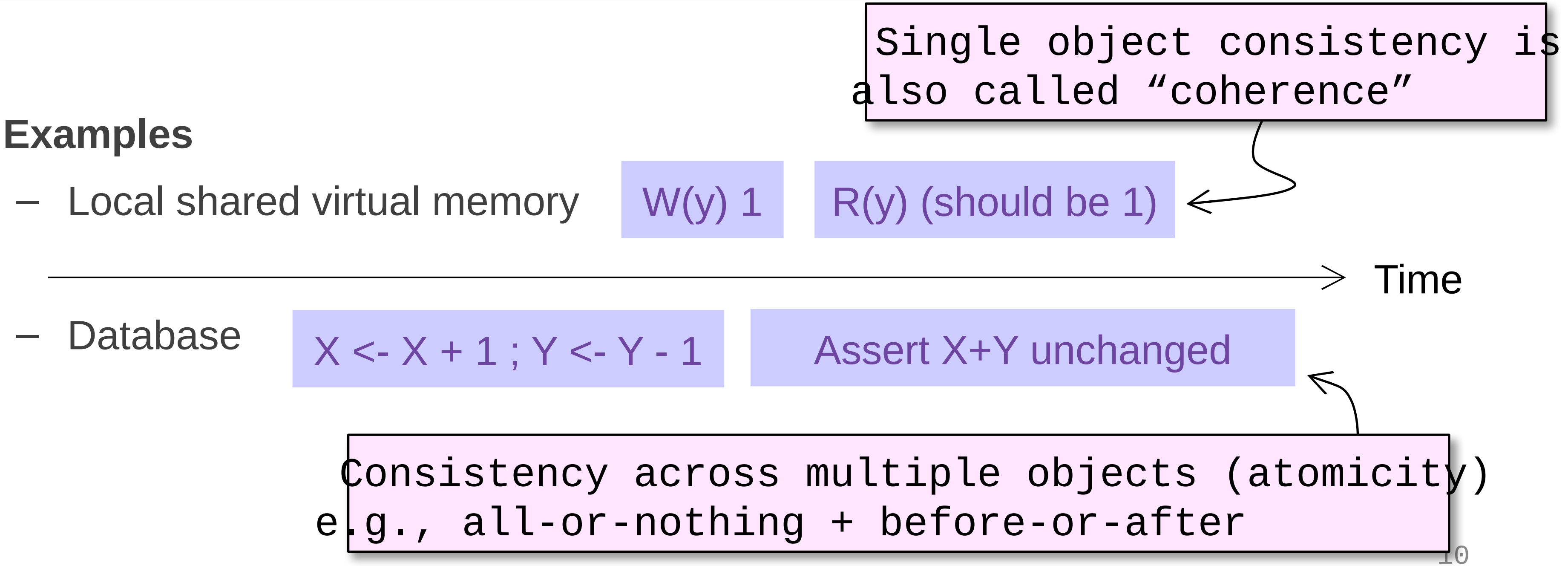
Examples

– Local shared virtual memory



Review: what is consistency model?

> **Consistency model** defines **rules** for the apparent order and visibility of updates, and it is a continuum with **tradeoffs**.
>
> \- Todd Lipcon



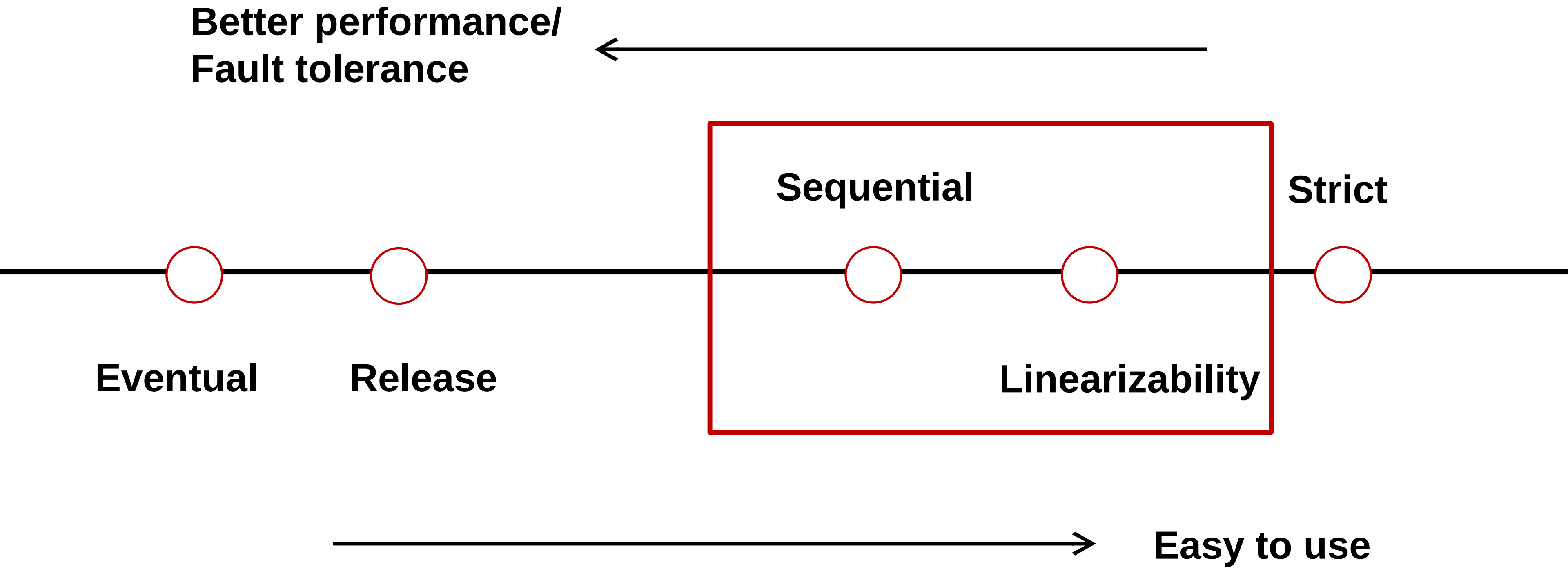
Single object consistency is also called “coherence”

Examples

- Local shared virtual memory
- Database

Consistency across multiple objects (atomicity) e.g., all-or-nothing + before-or-after

Consistency so far



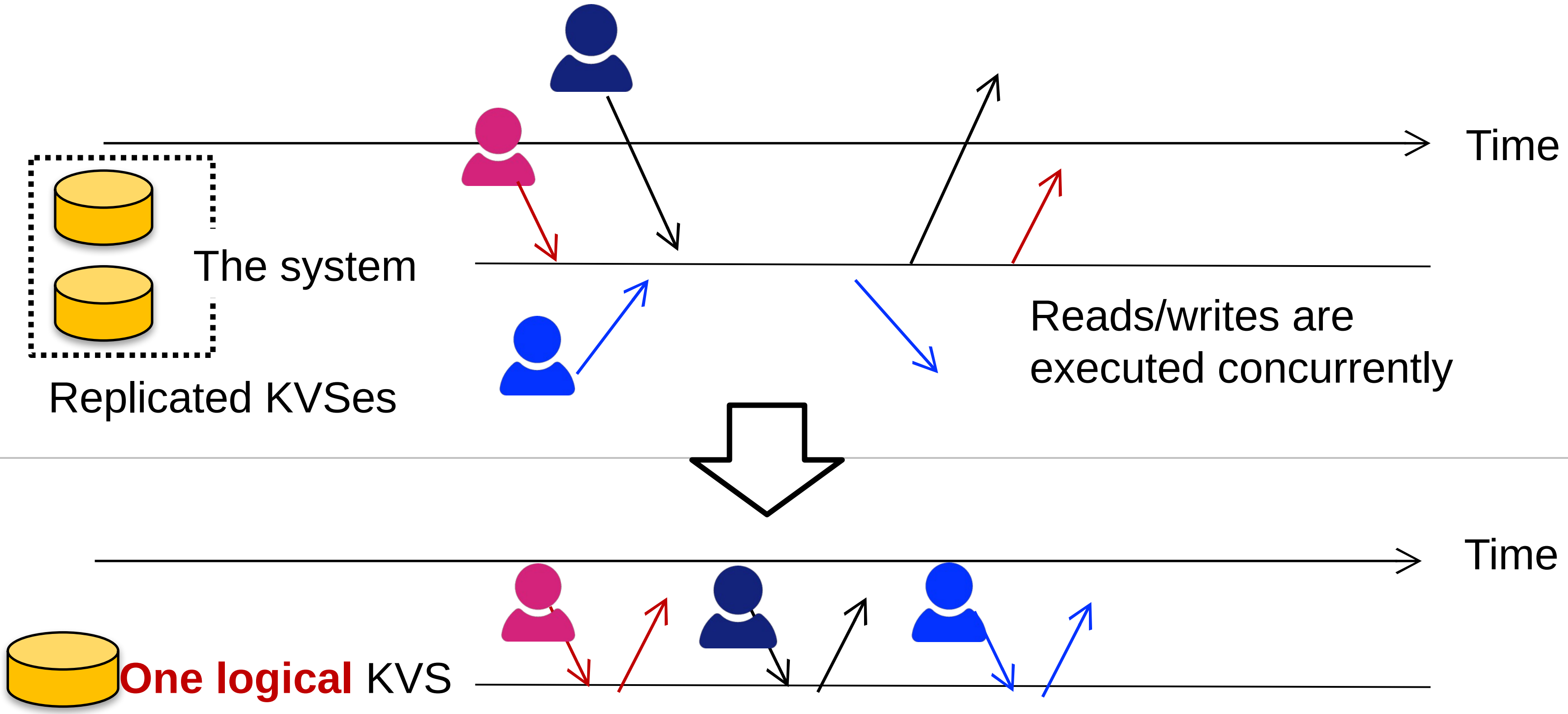
Note that many other models exists

Review: desired model for the developers/users

It's easy for users to reason about correctness assuming

- Everything has only one-copy
- The overall behavior is equivalent to **some serial behavior**

Review: desired model for the developers/users



Time

The system

Replicated KVSes

Reads/writes are executed concurrently

Time

One logical KVS

Reads/writes are executed **logically in a serial manner**.

Question: which serial order?

First， there is some inherent order we need to preserve

- e.g., if write(x) has been observed by a concurrent read(x), in the serial order write(x) is before read(x)

Different serial order will affect user experience

- E.g., If the system reports done of one op (read/write), a later started op should be ordered after it!

Three order defined

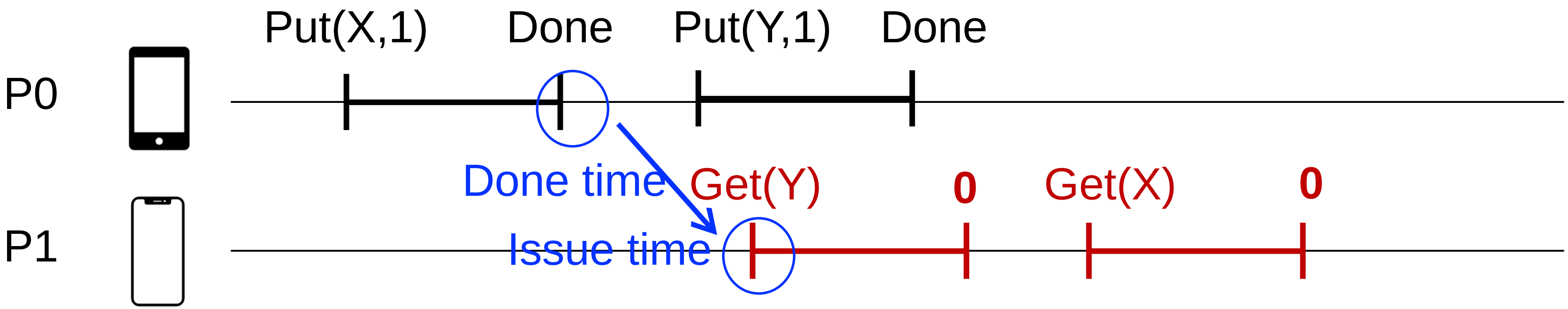
- #1. Global issuing order (strict consistency)
- #2. Per-process issuing/completion order (sequential consistency)
- #3. Global "completion-to-issuing" order (linearizability)

A desirable model

Use linearizability

Linearizability "completion-to-issuing" order

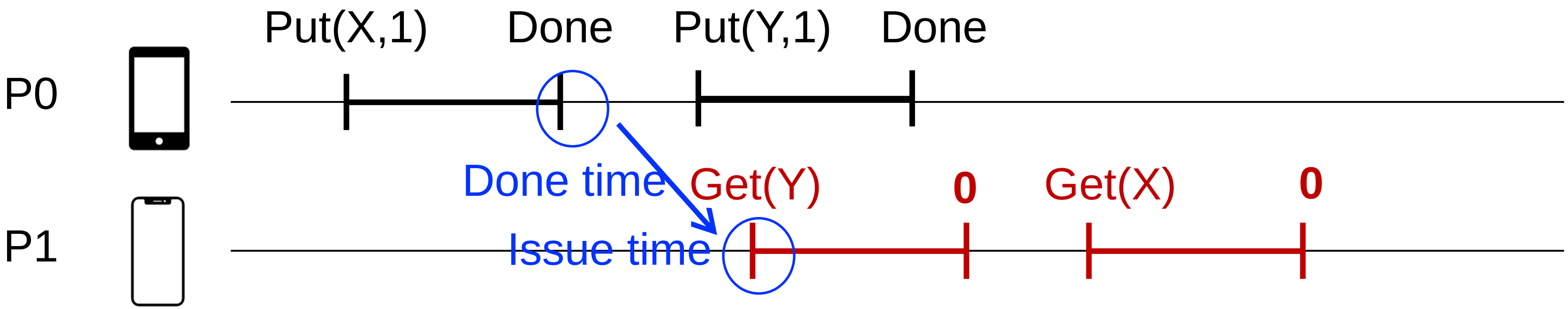
- All the concurrent execution is equivalent to a serial execution
- The order of each op matches "completion-to-issuing"



Use linearizability

Linearizability: "completion-to-issuing" order

- All the concurrent execution is equivalent to a serial execution
- The order of each op matches "completion-to-issuing"



Questions:

- Is the above execution serial?
- Is the current execution linearizable?

IMPLEMENTING LINEARIZABILITY OF KVS

Warmup property of linearizability: The local property [1]

If each object's op is linearizable, then overall system is linearizable

(Very) Simplified & (very) informal proof (By contradiction)

- Suppose we have two ops on x,y, e1 (x) & e2 (y)
- If non-linearizable, then we must have e1 < e2 & e2 < e1
- This is impossible:
  - real_time(e1_ok) < real_time(e2_start)   // e1 < e2
  - real_time(e2_start) < real_time(e2_ok)   // execution have time
  - real_time(e2_ok) < real_time(e1_start)   // e2 < e1
  - **real_time(e1_ok) < real_time(1_start)**   // combine the upper formulars
  - Since real_time(e1_start) < real_time(e1_ok), contradiction happens
- The concrete proof needs to reason on multiple ops & objects (w/ graph)

[1] Linearizability: A Correctness Condition for Concurrent Objects， TPLS'90

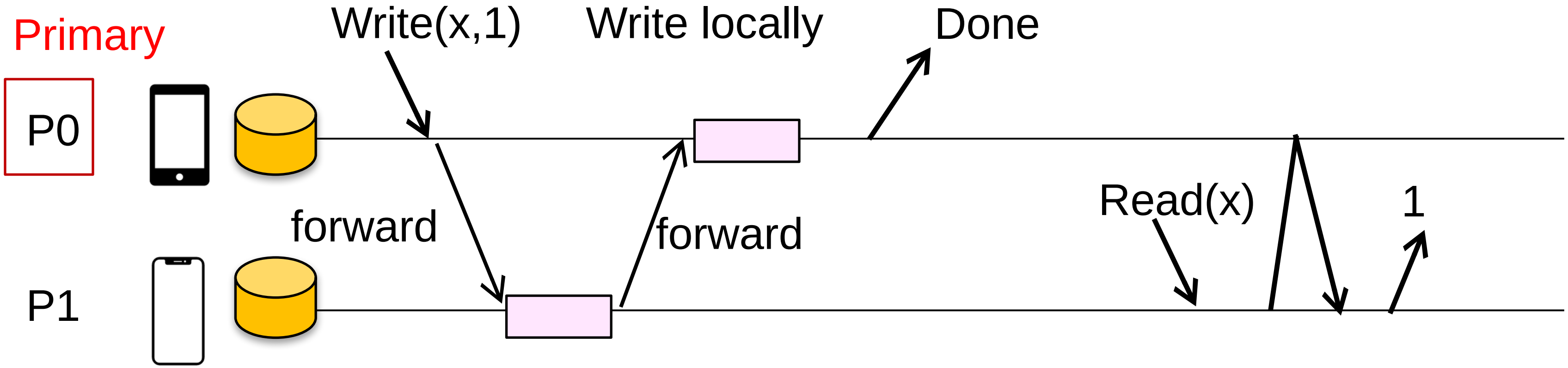
Implementing linearizability: Primary-backup approach

For each object, Clients send all reads/writes at a designated machine

For writes

1. Primary forwards writes to all the replicas
2. M0 executes writes locally (in order)
3. Respond OK

For reads
1. return the local copy of the data of the primary



Question: what are the drawbacks of primary-backup?

Performance issues

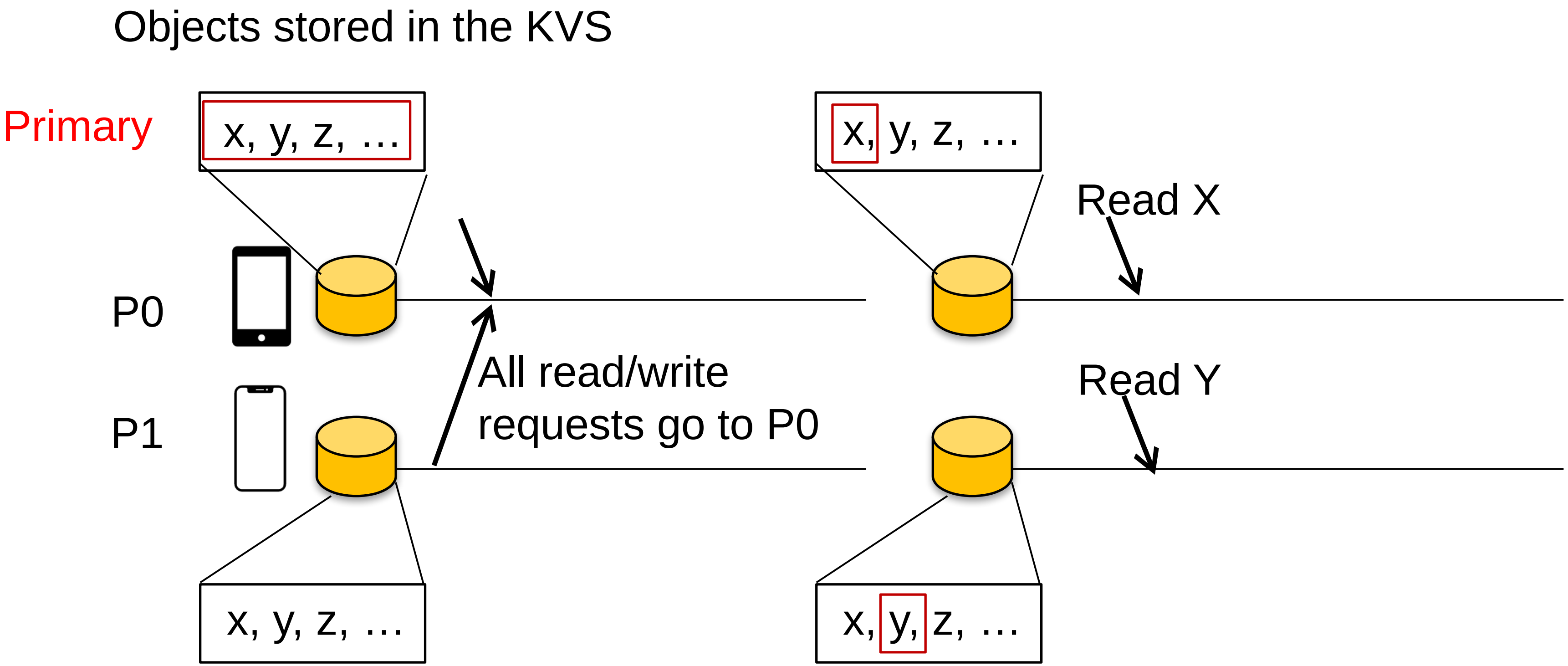
- Fallback to the centralized KVS case for non-primary devices
  - E.g., Extra roundtrip, Primary becomes the bottleneck, etc.

Reliability issues

- If primary crashes, others cannot work

Handling primary bottleneck: partitioning

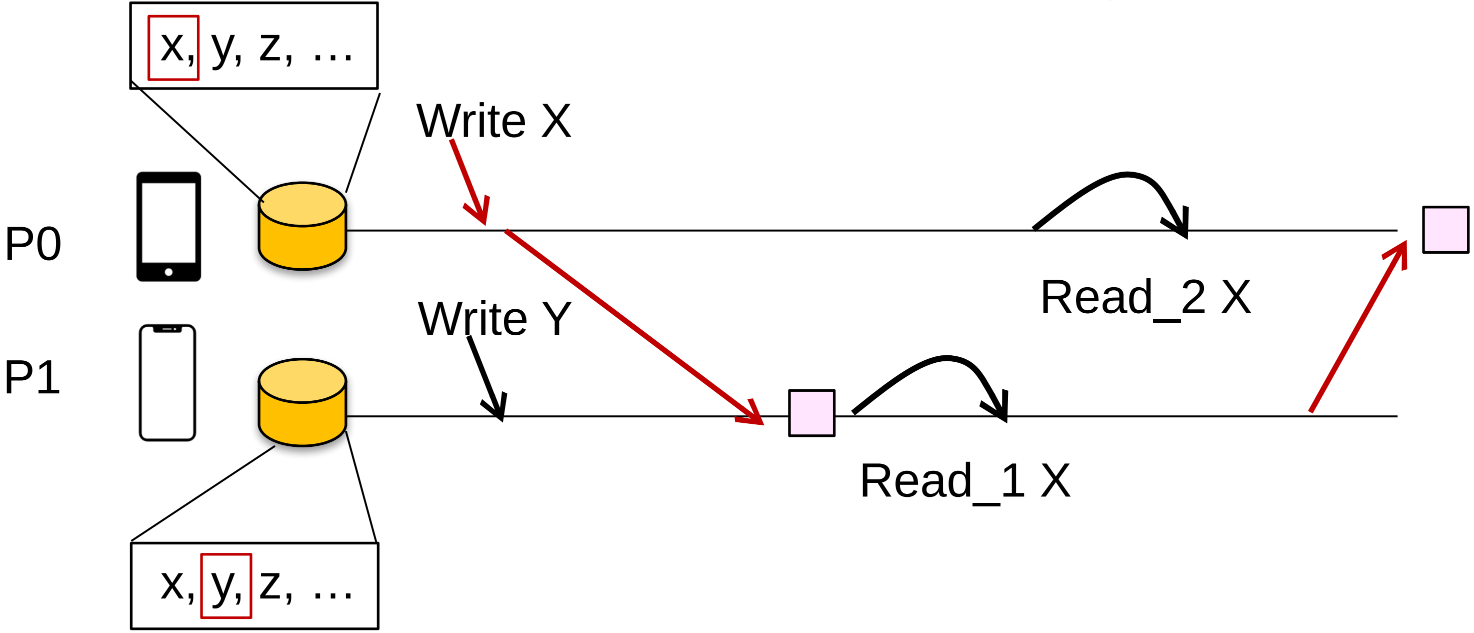
Different objects have different primaries



Handling extra RTT: read cache

e.g., Let each process directly return the local copy (as a cache)

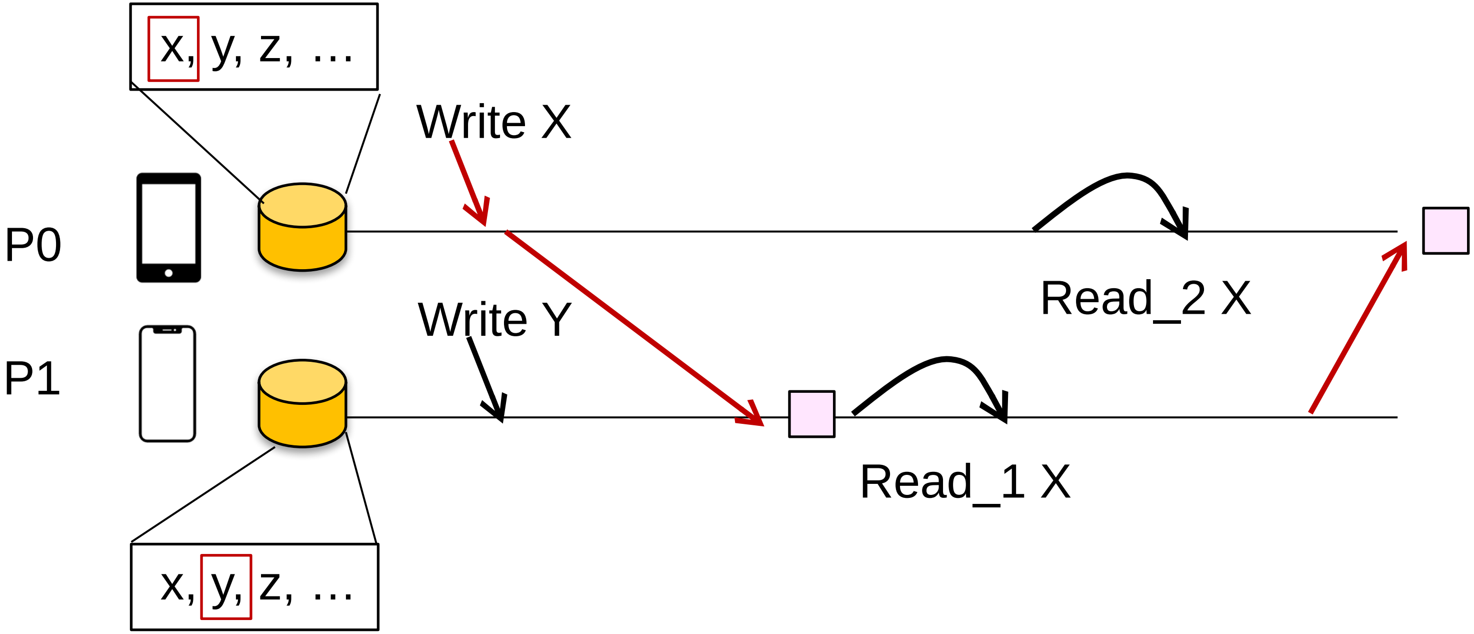
- May violate linearizability!
  - Read-1 < Read2: the start time of read_2 is after the completion time of read_1
  - Read-2 < Read1: read-2 does not see write x, but read 1 does



Correctly handling caches (Similar to cache coherence)

Each object has two modes: read-only mode and read-write mode

- If in read-only mode, the primary cannot do the update, others can read cache
- If in read-write mode, others cannot read the local cache



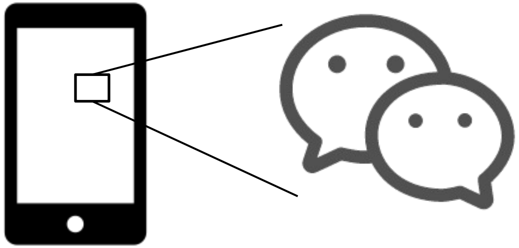
Drawbacks of linearizability in our example?

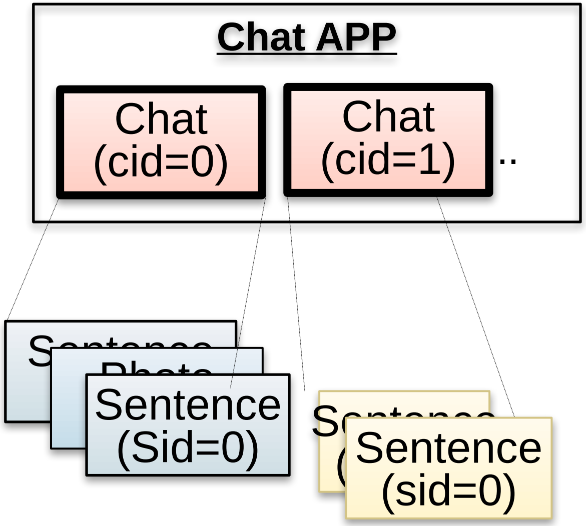
Impl. Primary-backup model

- Each read and write must be handled by a designated device
- Write must wait for all the replicas to acknowledge

Not practical for our chat applications

1. Performance issue: adding a sentence requires syncing all my devices
2. Availability issue: what if some device (e.g., iPad) is offline?





Drawback of linearizability

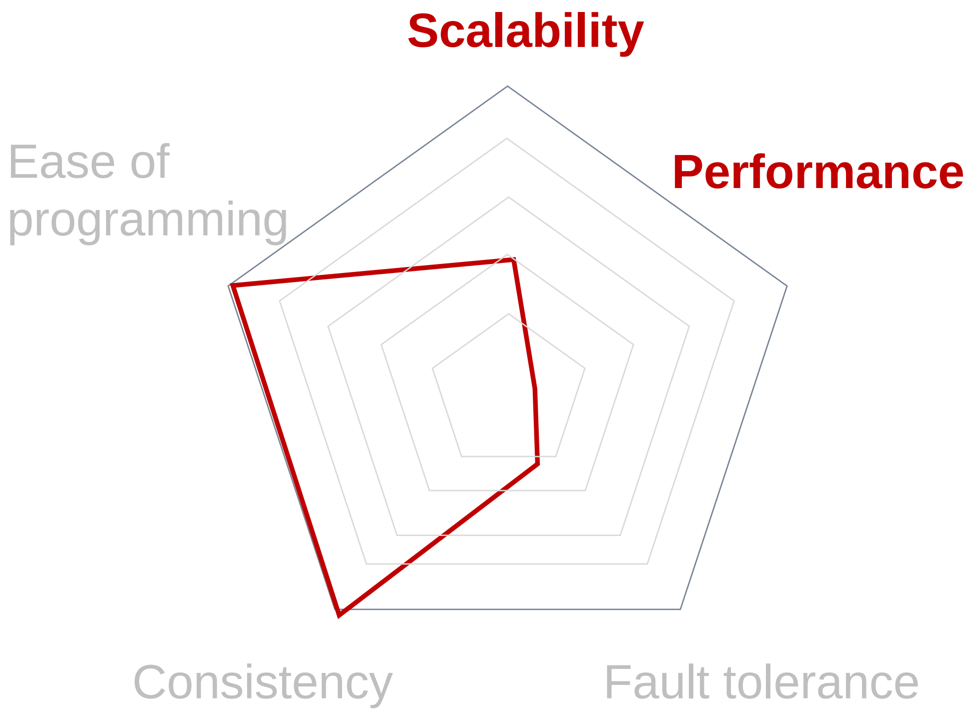
Very slow

- Must ask before each operation
- Single primary may become the performance bottleneck

Hard to provide fault tolerance

- Actually, not so hard, at the cost of performance. We will see solutions in later lectures

`KVS with linearizability`



Drawback of linearizability

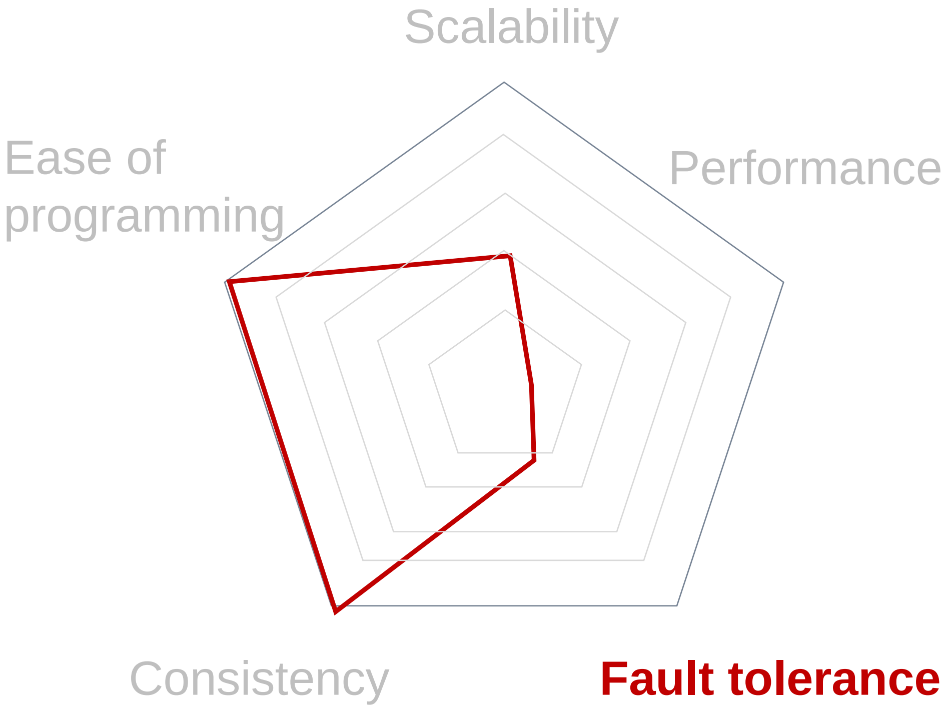
Require highly available network connections

- Lots of chatter between primary/backups

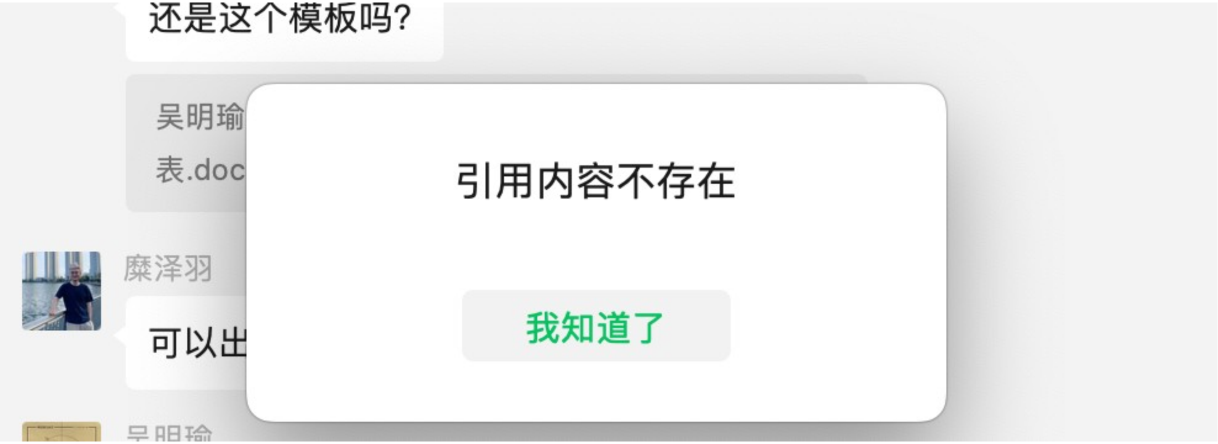
Not suitable for certain scenarios

- Disconnected clients (e.g., laptop, iPhone)
- Apps might prefer potential inconsistency to loss of availability (e.g., shopping cart)
- Geo-graphically distant apps (e.g., facebook)

KVS with linearizability



The common ChatAPP does not implement linearizability



还是这个模板吗?

吴明瑜

表.doc

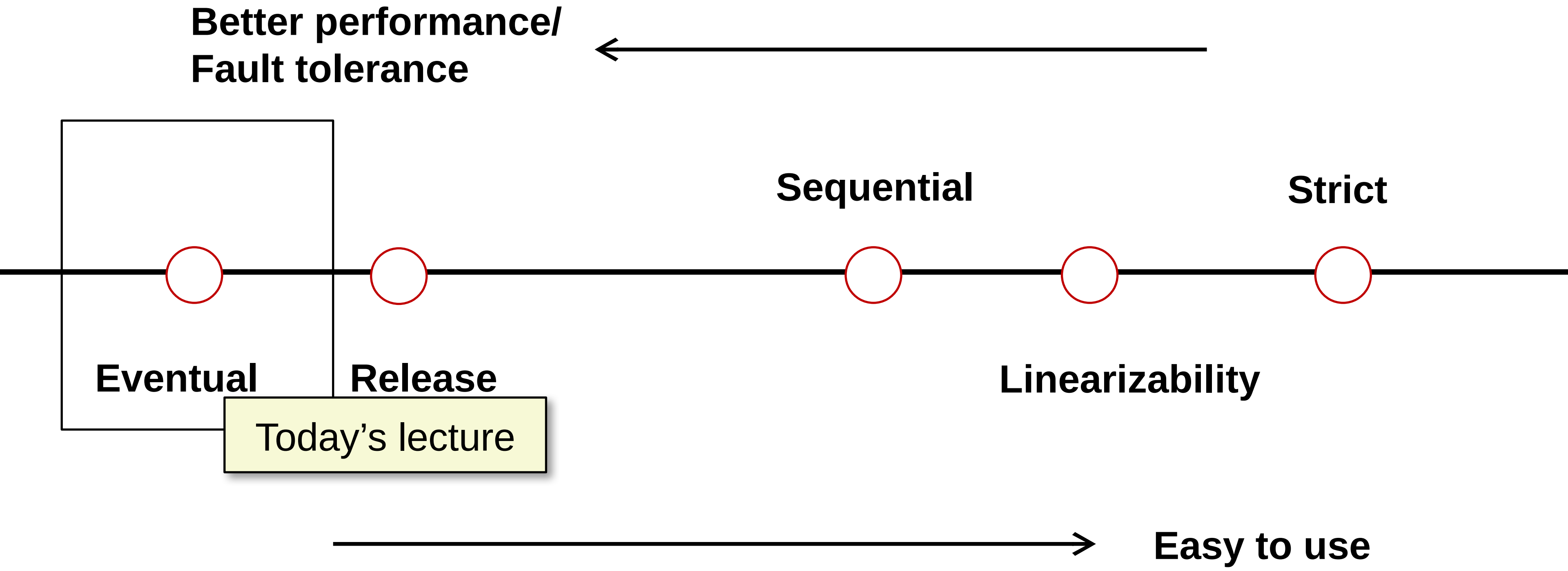
糜泽羽

可以出

吴明瑜

引用内容不存在

Spectrum of Consistency Models



Better performance/ Fault tolerance

Sequential

Strict

Eventual

Release

Linearizability

Today’s lecture

Easy to use

Note that many other models exists

Eventual consistency

Trade consistency for performance does not necessarily mean no system problem to address

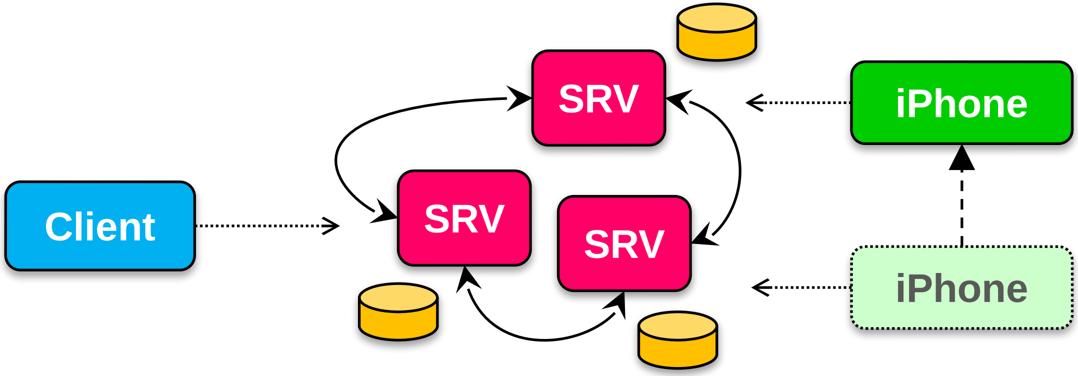
Ideal write execution flow for performance

Directly write to the server closest to the clients

- After server acks the writes, propagate the updates to the other servers later

Question

- What can go wrong in this setup? (assuming only writes now)
- Except for the read/write consistency issue we discussed before

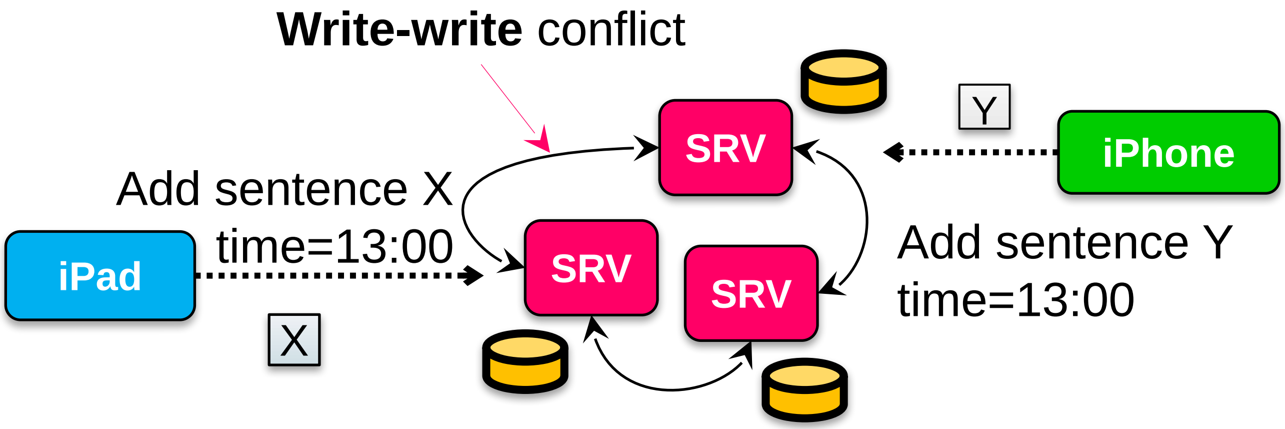


Problem: write-write conflict

A common anomaly that can happen is called write-write conflict

- Update the same thing concurrently without knowing each other

Result: the data diverges, i.e., one server add X then Y, the other in a reverse order, so the final data is not one copy!



Linearizability: forbids write-write conflict

The implementation will do pessimistic conflict handling

- Assumption: conflicting writes is common case
- Updates cannot take effect unless they are serialized first
    - Therefore, no write-write conflict, the first write that received by the primary wins!

Linearizability vs. Eventual consistency

Linearizability : pessimistic conflict handling

- Conflicting writes is common case
- Updates cannot take effect unless they are serialized first

Eventual consistency : optimistic conflict handling

- Conflicting writes is rare case
- Let updates happen, serialized the data later

The final goal is the same: we want the data to be the same across multiple device, eventually

Eventual consistency

A specific of weak consistency model, informally:

- All servers eventually receives all writes, and servers holding the same set of writes will have the same data contents
- Thus, if no new updates are made to the data, eventually all accesses will return the last update value

Possible read/write rule implementations

- Read: return the **latest local** copies of the data
- Write: write locally (and directly returns), propagate the writes to all the servers in background (e.g., upon sync)

Eventual consistency

Trade consistency for better performance

- Accept a write before being able to serialize it
- Reads can return a (possible) stale value instead of blocking for the latest value; there is no global ordering between reads & writes

But still need to cope with anomalies

- Write-write conflict
- Loss of causality

Need **converging** state

Need **causality** preserving

Eventual consistency

For better performance

- Accept a write before being able to serialize it
- Reads can return a (possible) stale value than blocking for the latest value

But still need to cope with anomalies

- **Write-write conflict**
- Loss of causality

Need **converging** state

Handling write-write conflict

Solution

- After a local write, send writes to all the servers in the background
- Each server apply the writes, resolve the conflicts if necessary

Problem #1: how to apply updates ?

- Can we directly overwrite the original content?
- Assumption: chats are in a key-value store, where the value is a vector
- Overwrite origin content: chat[cid] = […, new_sid]

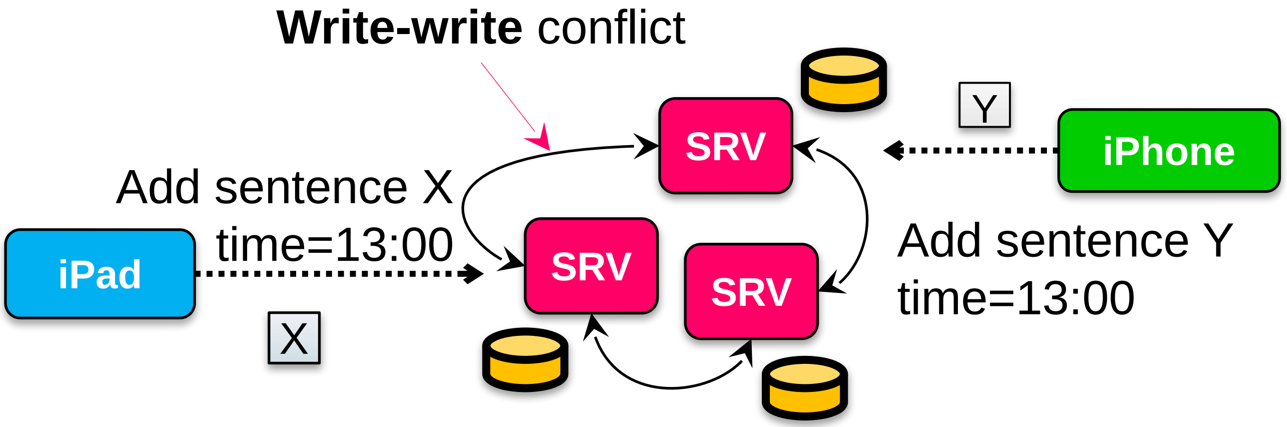
Apply the updates

Strawman

- Can we directly replace the old value with the new writes?
- No. losing application semantic will also be "inconsistent"

Conflicts handling is application specific

- Different applications may have different requirements



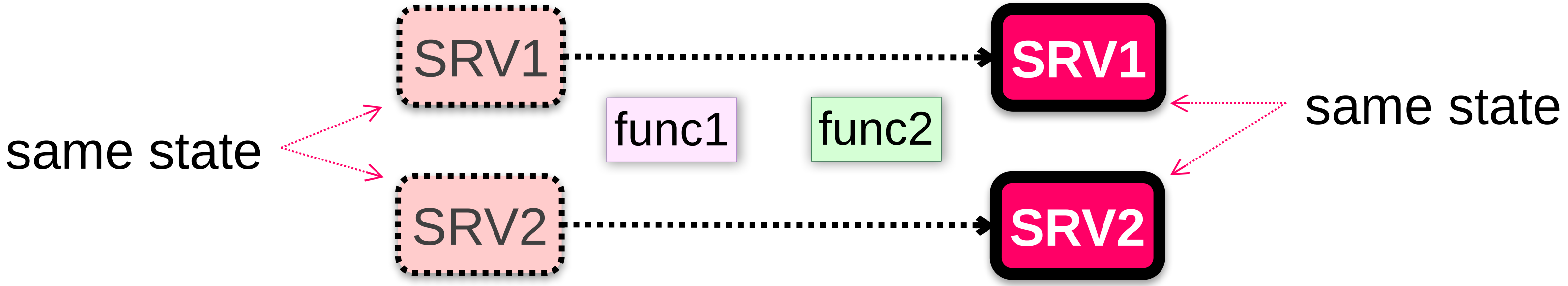
Solution: update function

Have update be a function (provided by the user), not a new value

- e.g., function: 1. allocate a sid; 2. *chat[cid].append(sid)*

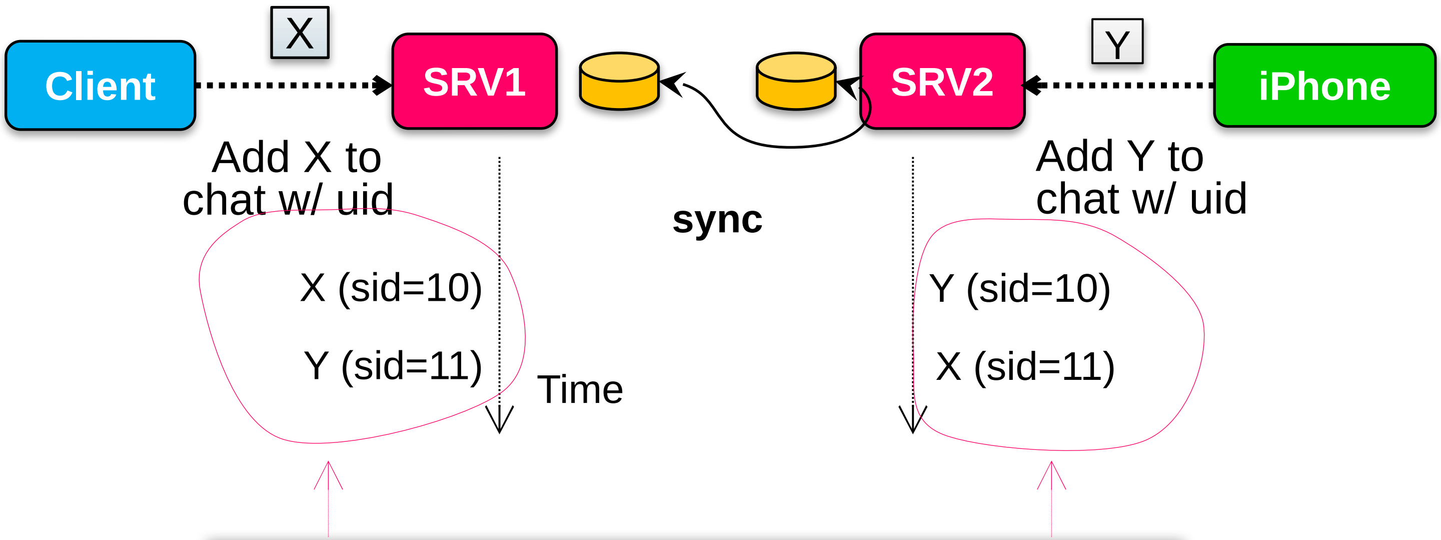
Function must be deterministic

- Otherwise servers will get different answers



Question: does update function alone sufficient?

Challenge: ordering



Will Server1 show X in front of Y, and Server2 show Y in front of X?

Goal: eventual state convergence

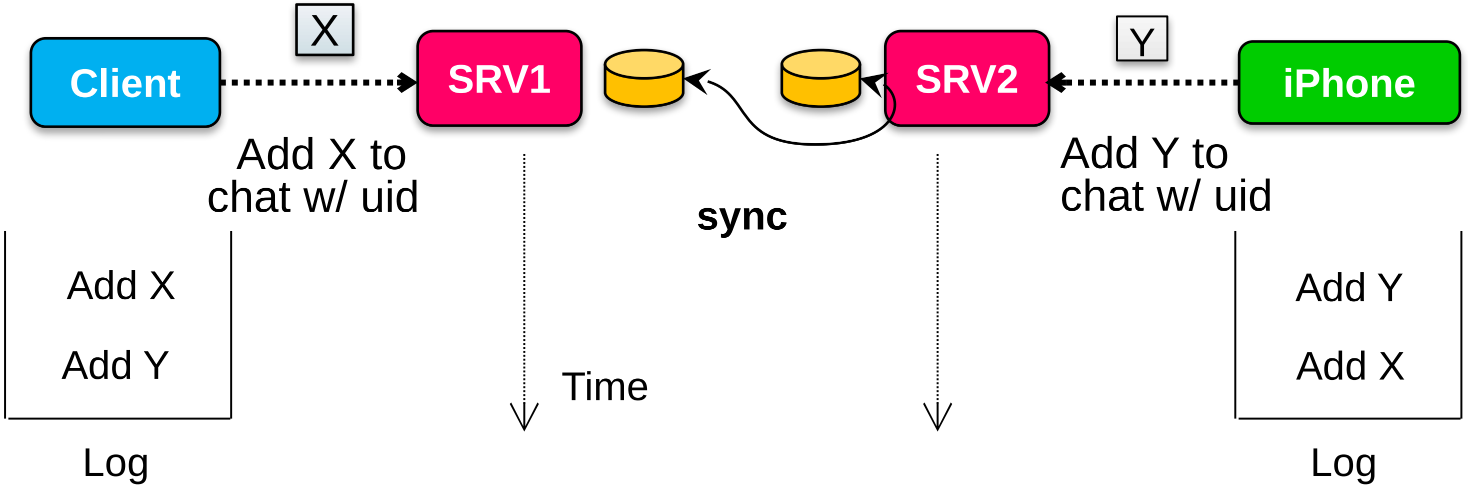
Solution: Ordered Update Log

Ordered list of updates at each node

- Record the updates in a log, and sort it according to some order

Delay the updates, until we are sure that it can be ordered

Syncing: ensure both nodes have the same updates in log



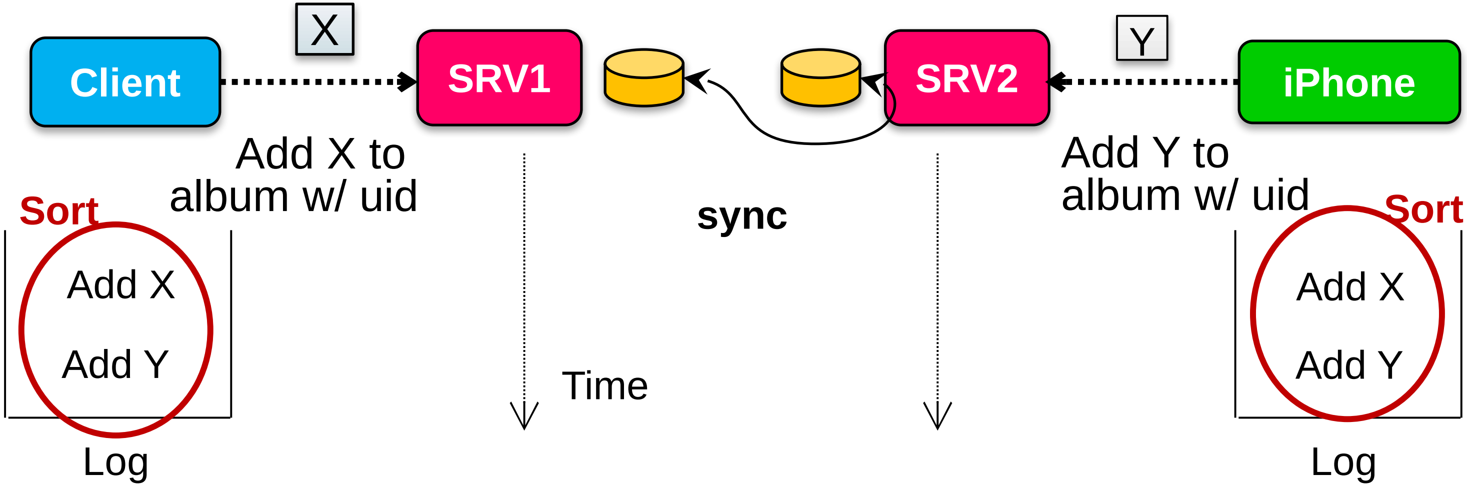
Solution: Ordered Update Log

Ordered list of updates at each node

- Record the updates in a log, and sort it according to some order

Delay the updates, until we are sure that it can be ordered

Syncing: ensure both nodes have the same updates in log



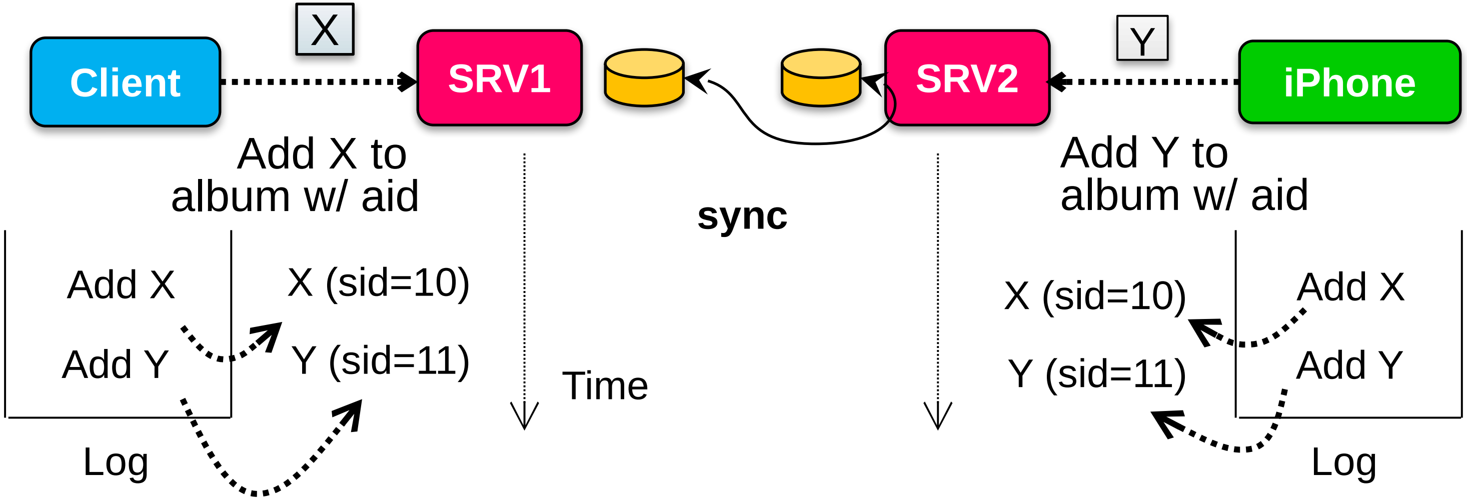
Solution: Ordered Update Log

Ordered list of updates at each node

- Record the updates in a log, and sort it according to some order

Delay the updates, until we are sure that it can be ordered

Syncing: ensure both nodes have the same updates in log



Few other things to solve

1. **Sort by which order?**

Determine the order of updates: using time

Tag update with the time the server receiving the events

- Since each update has a timestamp, we can sort it in the log

Challenge: there are ties

- Two events happen to be at the same time
  (even with well synchronized clocks)

Solution: assign unique IDs to the updates

Update ID: <time T, node ID>

- Assigned by node that creates the update
- Node ID is used to break the tie

Ordering update X and Y:

X < Y iff **(X.T < Y.T)**

or

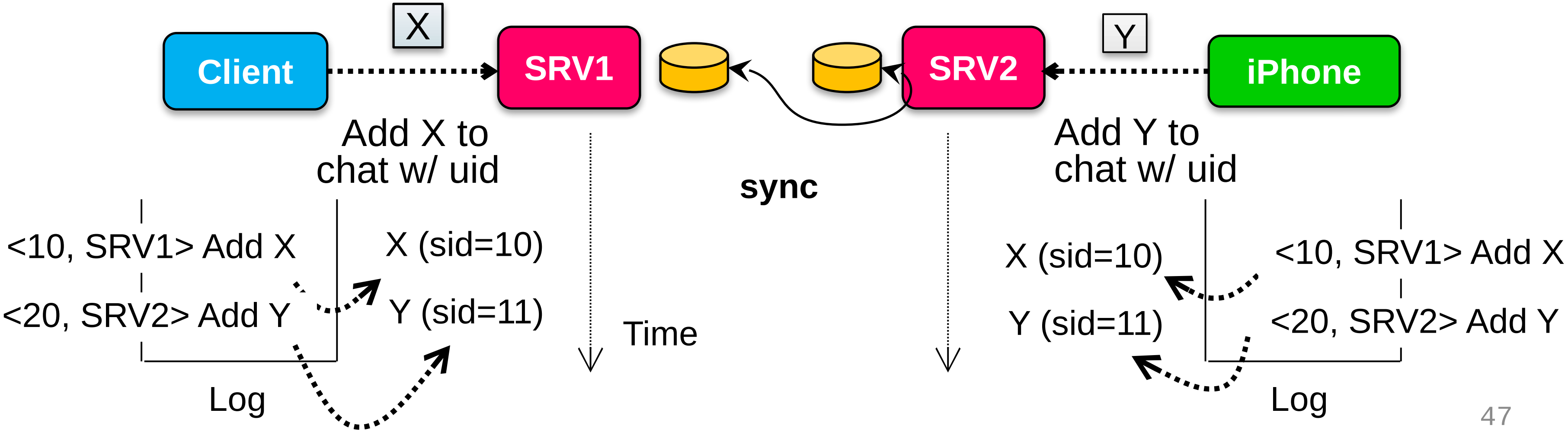
(X.T=Y.T and X.ID < Y.ID)

Using unique update IDs to sort entries in the log

Update ID: <time T, node ID>

<10, Srv1>: sentence **X** at **10**th or **11**th position for chat#1

<20, Srv2>: sentence **Y** at **10**th or **11**th position for chat#1



Few other things to solve

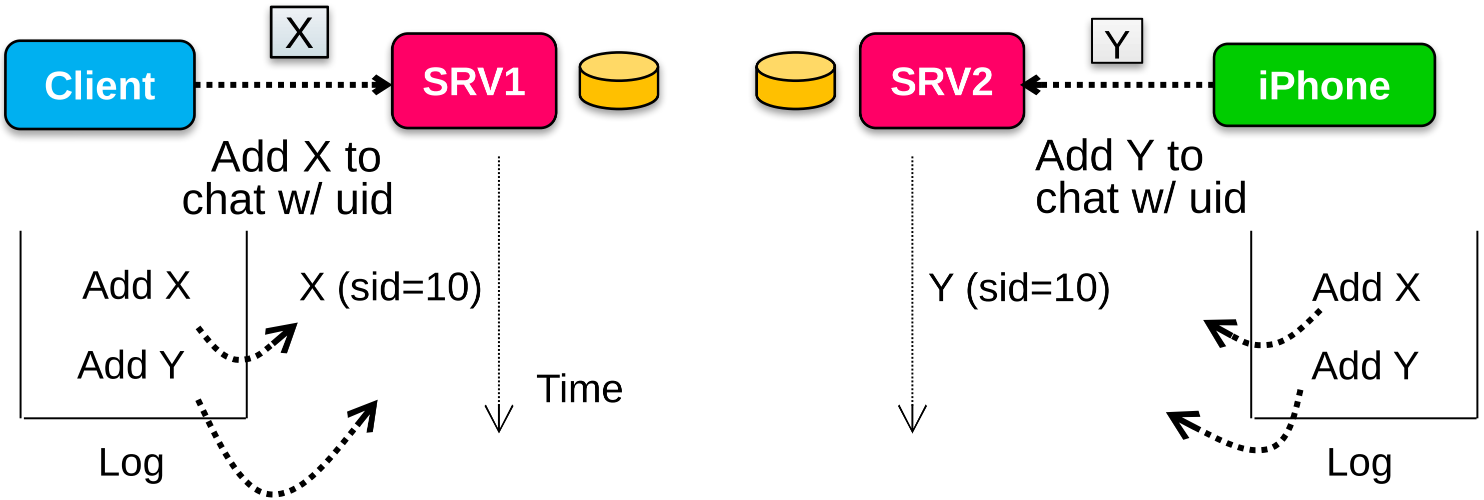
1. **Sort by which order?**

2. **When to apply the updates?**
   - To achieve state convergence, we must delay the updates
   - However, this is unsatisfactory : I added a sentence, but didn’t show the on my phone!
   - Question
     - Can we directly apply the updates locally?

Problem: local updates break the order of the log

If we naively update the local storage, we cannot directly run the update functions after the sync

– Because the initial state of the servers are different!



Solution: Rollback and Replay

Rollback all the updates before the sync

- Essentially clean the storage to an empty state

Re-run all update functions, starting from empty storage state

- After syncing, Srv1 and Srv2 have same set of updates (ordered logs)
- Srv1 and Srv2 arrive at same final state

Problems (We will come back to these issues later)

- Slow sync process
- Large log size

How to choose the unique IDs is important

Recall: we choose <time T, node ID> as the unique ID of updates

Question#1: can we use <node ID, time T> as the ID?

- Sort the updates: no problem
- User-experience (or causality preserviung): no so good. For example, I will always see my iphone's data before my Ipad's data, even I posted the sentence earlier

Question#2: can <time T, node ID> prevent the above user-experience issue?

- Depends on the clock time of the nodes
  - Yet we know that clocks are unsynchronized

Challenge: unsynchronized clock time

Goal: clock time reflect the (real wall clock time) update order

Will update order be consistent with wall-clock time?

- E.g., if I first post a chat on iPhone, and later post another chat on iPad, the iphone's time will be smaller

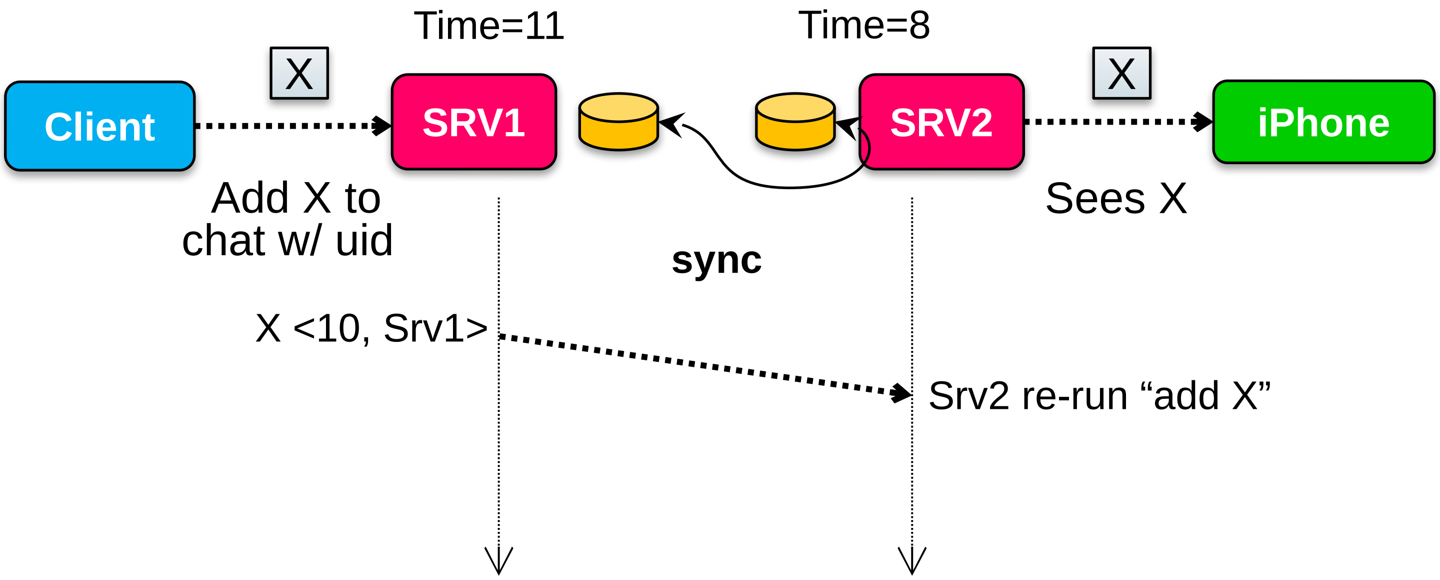
However, the clock time between nodes is not synchronized

- Example

Srv1 went first in **wall-clock** time with **<10, Srv1>**

Srv2 could still generate update ID **<9, Srv2>**

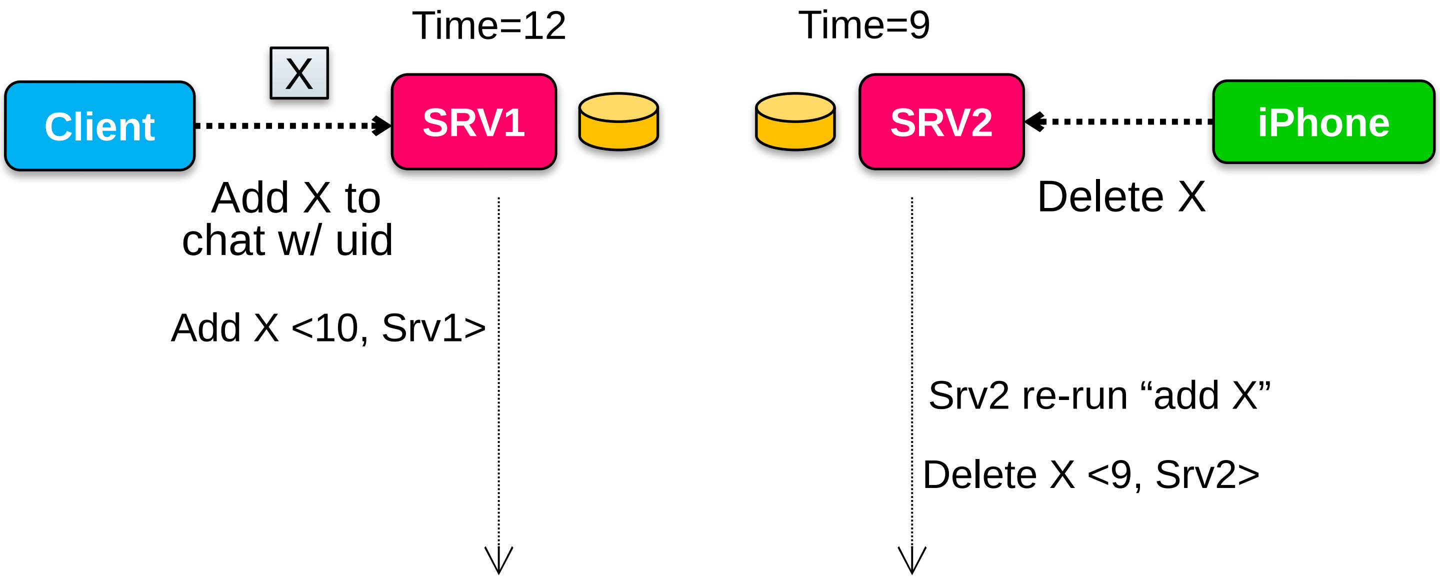
Unsynchronized clocks can cause subtle problem



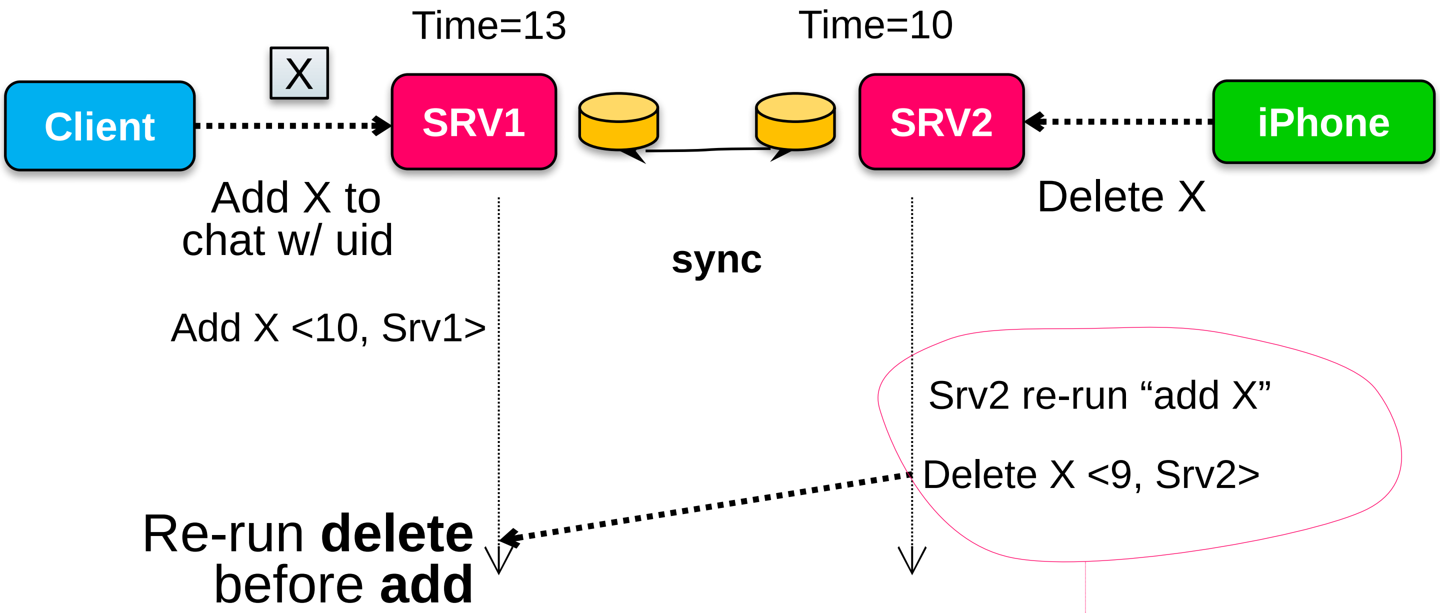
In our setup, updates apply to local storage immediately

- So that we can see the latest results
- Inconsistency will be rollback and replay later

Unsynchronized clocks can cause subtle problem



Unsynchronized clocks can cause subtle problem



Causality preserving

Causal ordering

Definition of causal relationship: X -> Y iff

- X,Y is created on the same node and X happens before Y (single-thread)
- X, Y created by node$_i$ and node$_j$, and X “causes” Y
- Example
    - X is add chat, Y is delete the chat
- There exists Z, such that X -> Z -> Y

Causal ordering

<Time, ID> does not preserve the casual ordering

- Why? The lock is unsynchronized

Goal

- Clock should reflect the causal order

Idea

- Sync a logical clock using the events dependencies

Solution: Lamport clock

We need causality timestamp

Lamport clock: a logical clock used to assign timestamps to events at each node

- If event #1 and event #2 are created on the same node, and event #1 is before event #2
  - ✉ TS(event #1) < TS(event #2)
- If event #1 “cause” event #2,
  - ✉ TS(event #1) < TS(event #2)

Lamport clock algorithms

Lamport logical clock

① Each server keeps a clock **T**

② Increments T as real time passes, e.g., one second per second

③ Modify **T = Max(T, T'+1)**
if sees **T'** from another server

Srv1: Add X with **<10, Srv1>,** Suppose **T_1 = 10**

Srv2: Sees "add X", then update **T_2 = Max(T_2, 11)**

Srv2: Delete X with **<11, Srv2>**

Now, **causality** preserving!

Questions on Lamport clock

Can lamport clock give a total order to all the events?

- i.e., given e1 & e2, T(e1) < T(e2) or T(e1) > T(e2)
- No. There are ties

Yet, we can trivially break the tie by adding an additional order:

- E.g., <Logical time, node ID>
- The total ordering preserves the causality

Why total ordering is ideal?

- If replicas apply writes according to total ordering, then replicas converge

Questions on Lamport clock

If TS(event #1) < TS(event #2), what does it say about event #1 (create on node #1) and event #2 (created on node #2)?

① Event #1 occurred at a physical time earlier than event #2

② Node #1 must have communicated with Node #2

③ If event #1 has been synced to node #2, then event #1 must have occurred at a physical time earlier than event #2

Lamport clock gives a total order to events

For every pair of events, we can compare their time

- i.e., T(e1) < T(e2) or T(e1) > T(e2)

Problem: too strong

- The order may disagree with the external observer, especially for un-related events
- E.g., suppose I add a sentence to Alice, then adds a sentence to Bob independently
    - The system may order Bob’s event earlier than me with Lamport‘s clock, even they are unrelated

Requirement: partial order

Consider two events e1 & e2, they may have

– With lamport clock, they are always comparable

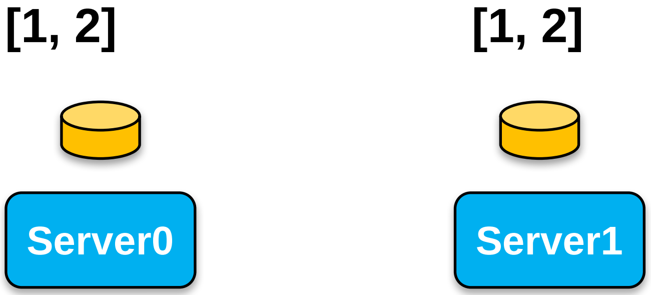
Partial order: we need incomparable timestamps

– So we can identify them as concurrent events

Realizing partial order: vector clock

Clocks are represented as a vector (one entry per server)

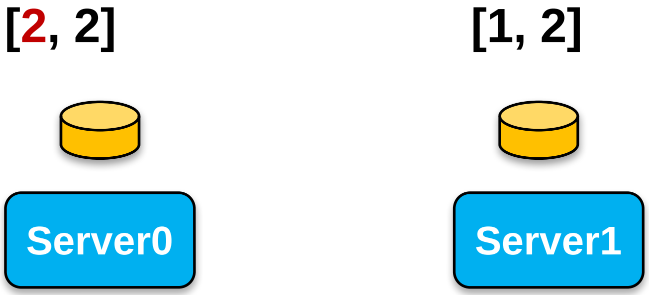
- Each entry corresponds to a local lamport clock on the server



Realizing partial order: vector clock

Clocks are represented as a vector (one entry per server)

- Each entry corresponds to a local lamport clock on the server
- Increments local T as real time passes, e.g., one second per second

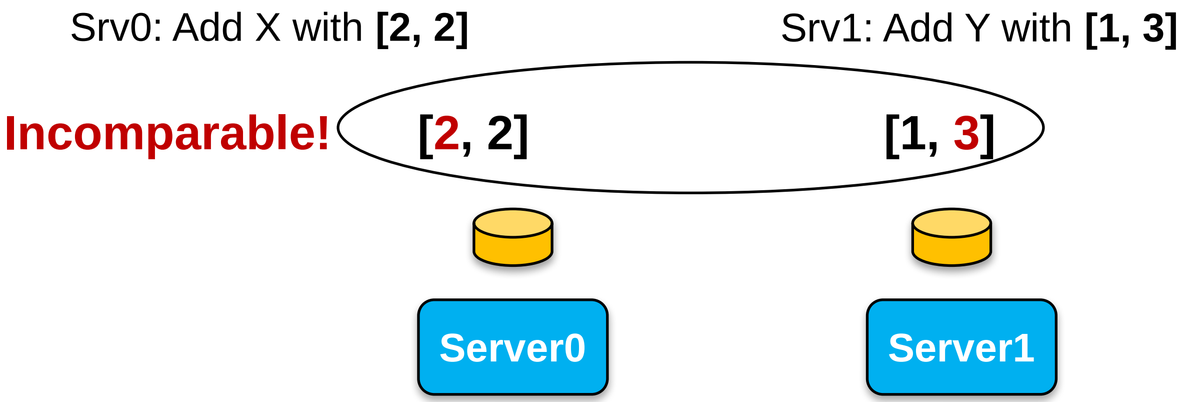


Realizing partial order: vector clock

Clocks are represented as a vector (one entry per server)

- Each entry corresponds to a local lamport clock on the server
- Increments local T as real time passes, e.g., one second per second

Suppose we have two independent events here



Srv0: Add X with **[2, 2]**

Srv1: Add Y with **[1, 3]**

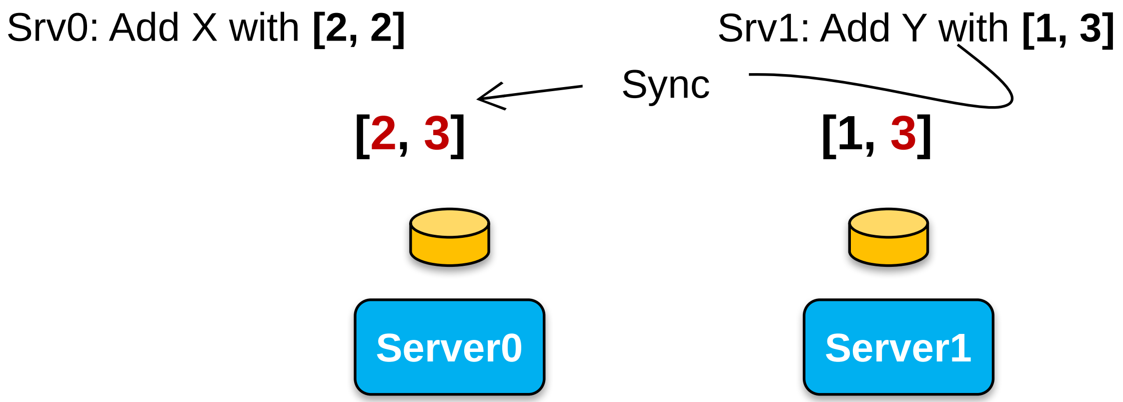
Incomparable!

Realizing partial order: vector clock

Clocks are represented as a vector (one entry per server)

- Each entry corresponds to a local lamport clock on the server
- Increments local T as real time passes, e.g., one second per second
- Modify $\mathbf{T_i = Max(T_i, T_i'+1)}$

  if sees **T** from another server

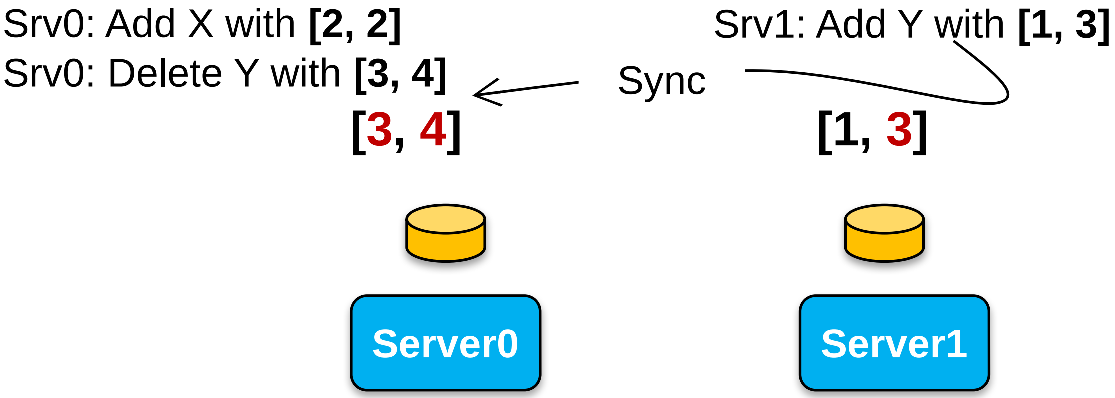


Realizing partial order: vector clock

Clocks are represented as a vector (one entry per server)

- Each entry corresponds to a local lamport clock on the server
- Increments local T as real time passes, e.g., one second per second
- Modify $\mathbf{T_i = Max(T_i, T_i'+1)}$

  if sees **T** from another server



Srv0: Add X with **[2, 2]**
Srv0: Delete Y with **[3, 4]**

Srv1: Add Y with **[1, 3]**

Lamport clock vs. Vector clock

Both captures the causal relationships between events

In comparison:

- Lamport clock: total order
- Vector clock: partial order

For many scenarios, Lamport clock is sufficient

- Sufficient to order events
- Save space for storing the clock

Final problem: we need to truncate log

Recall

- For performance, we place a write immediately at the local node
- However, the write may be **unstable** (we call **tentative writes**)
- Our previous solution will re-run all update functions, starting from empty storage state
  - Inefficient

Goal

- Avoid re-run all the update functions
- Reduce the log size

Idea: distinguish tentative writes from stable ones

Each server's log consists of 2 portions:

- Stable writes, followed by
- Tentative writes

Stable writes are not rolled backup upon sync

Question

- How to determine which writes are stable? (hint: using the lamport clock!)

Answer

- An update (W) is stable iff no entries will have a lamport timestamp < W

De-centralized approach

Commit (write) scheme

- Update <10, A> is **stable** if all nodes have seen all updates with time <= 10
  - Because it will never see a timestamp < 10 (according to the rule of lamport time)

Problem: If **any** node is **offline**, the **stable** portion of **all** logs stops growing

  - Still so many writes may be rolled back on re-connect

Centralized approach

CSN: Commit-Seq-No

Commit scheme

① One server designated “**primary**”

- Assign a total commit order: CSN to each write
- Complete timestamp: <**CSN**, local-TS, SrvID>
- Any write with a known CSN is stable

② All stable writes are ordered before tentative writes

③ CSNs are exchanged between servers

- CSNs define a total order for committed update

Advantage: as long as the primary is up, writes can be committed and stabilized

Does CSN order preserve causality?

Complete timestamp: <CSN, local-TS, SrvID>

Should do some works

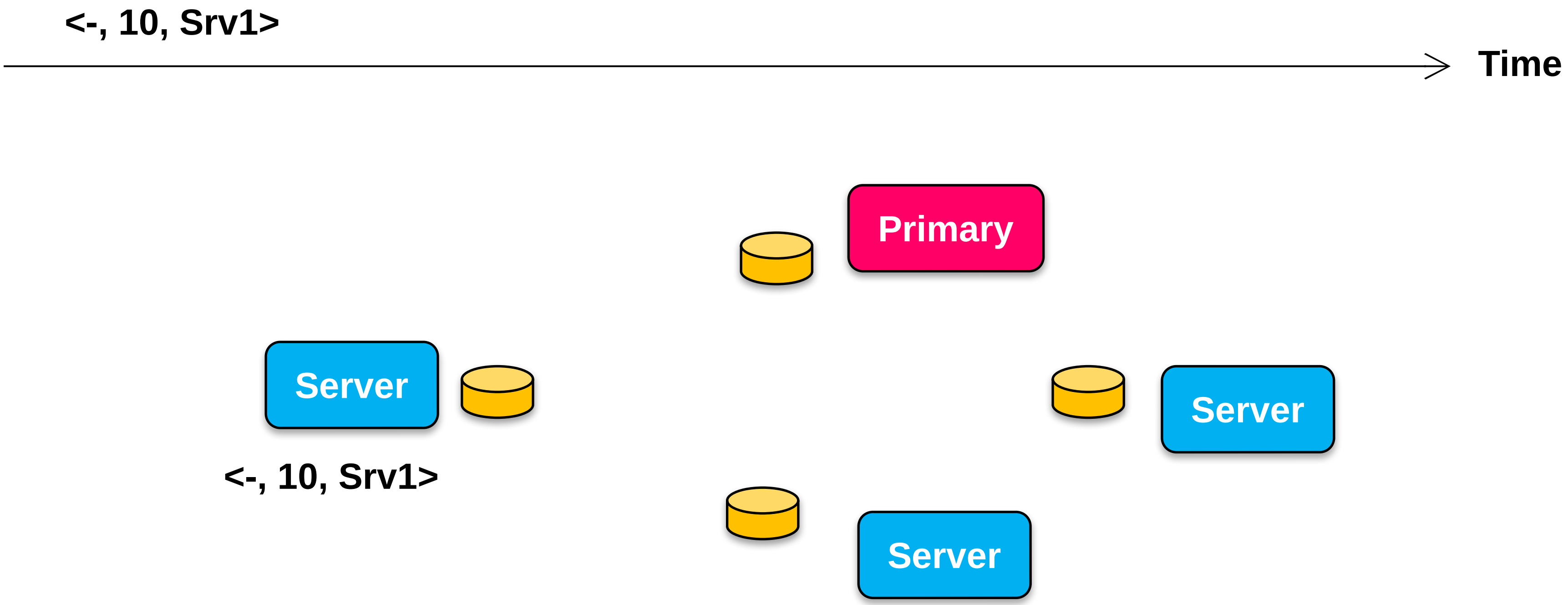
- A server asks the primary to assign CSN for all tentative writes (include those received from others)
  - i.e., the primary can assign a larger CSN for a dependent write

Question

- Does complete timestamp always match the tentative order?

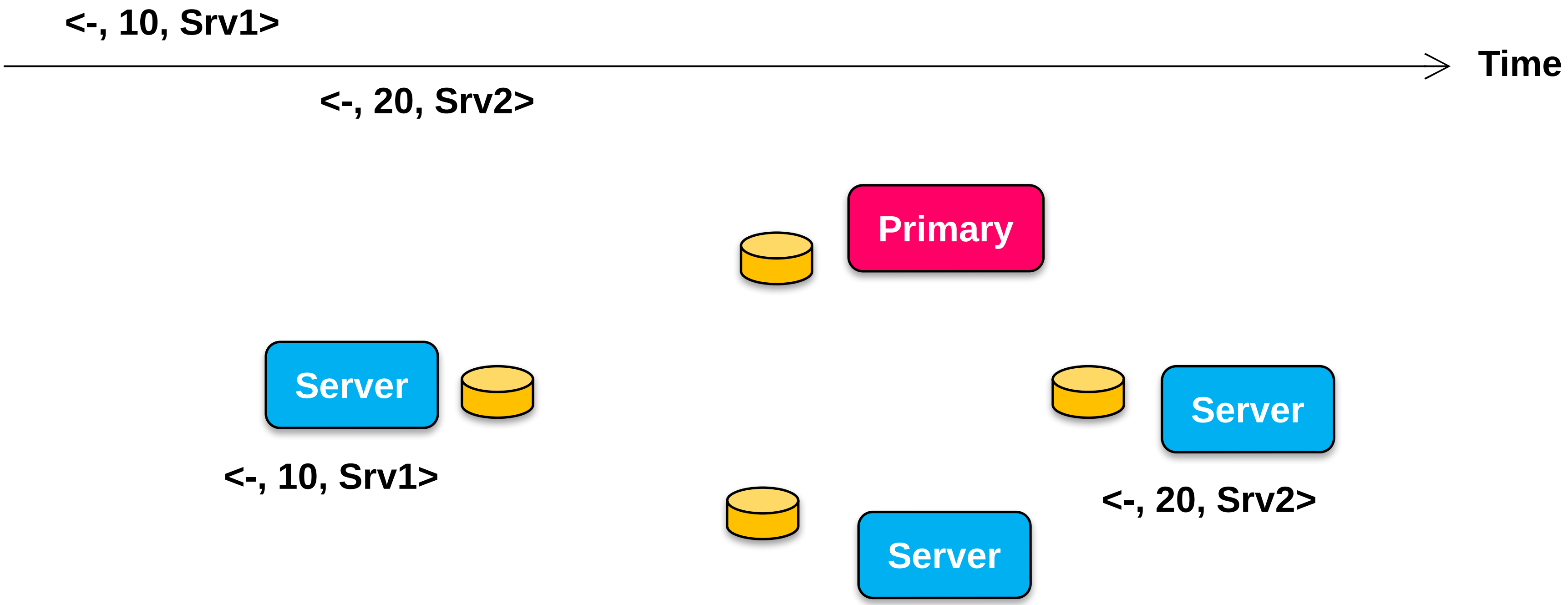
Will CSN matches update order?

Not always. The primary can see newer updates before the old ones, as long as they don't have casualty relationship



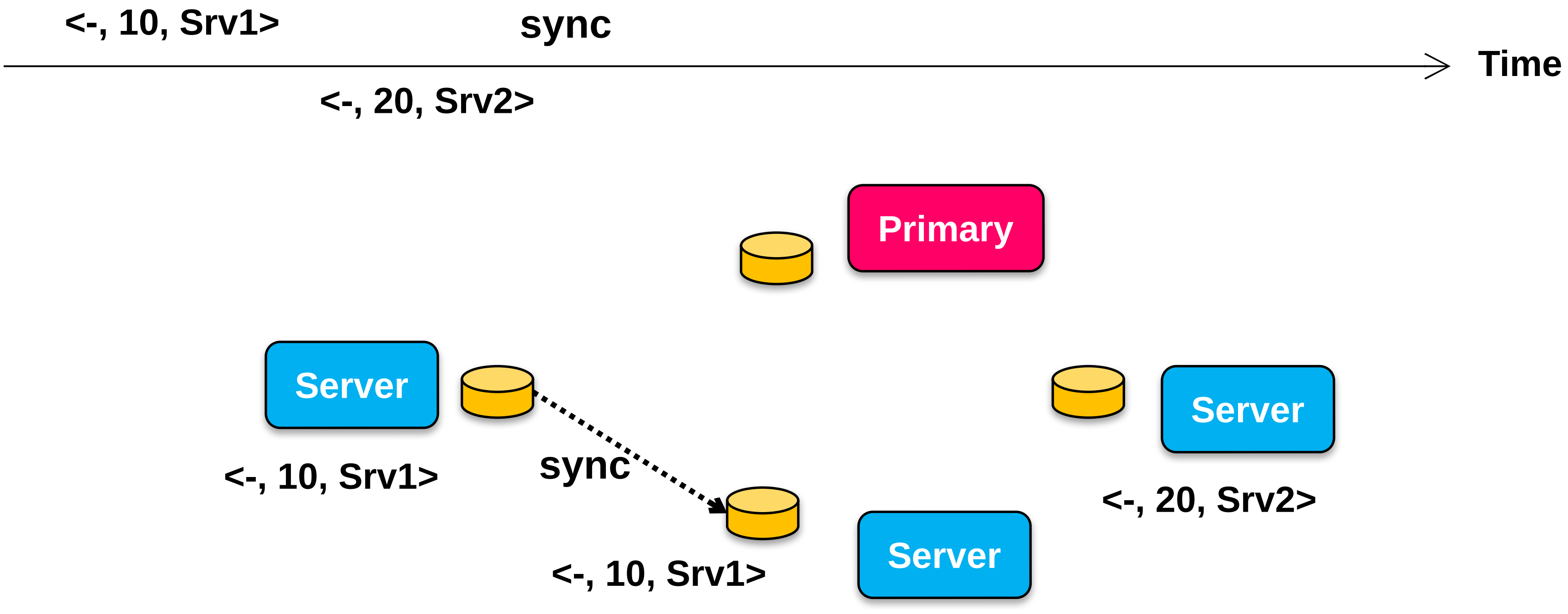
Will CSN matches update order?

Not always. The primary can see newer updates before the old ones, as long as they don't have casualty relationship



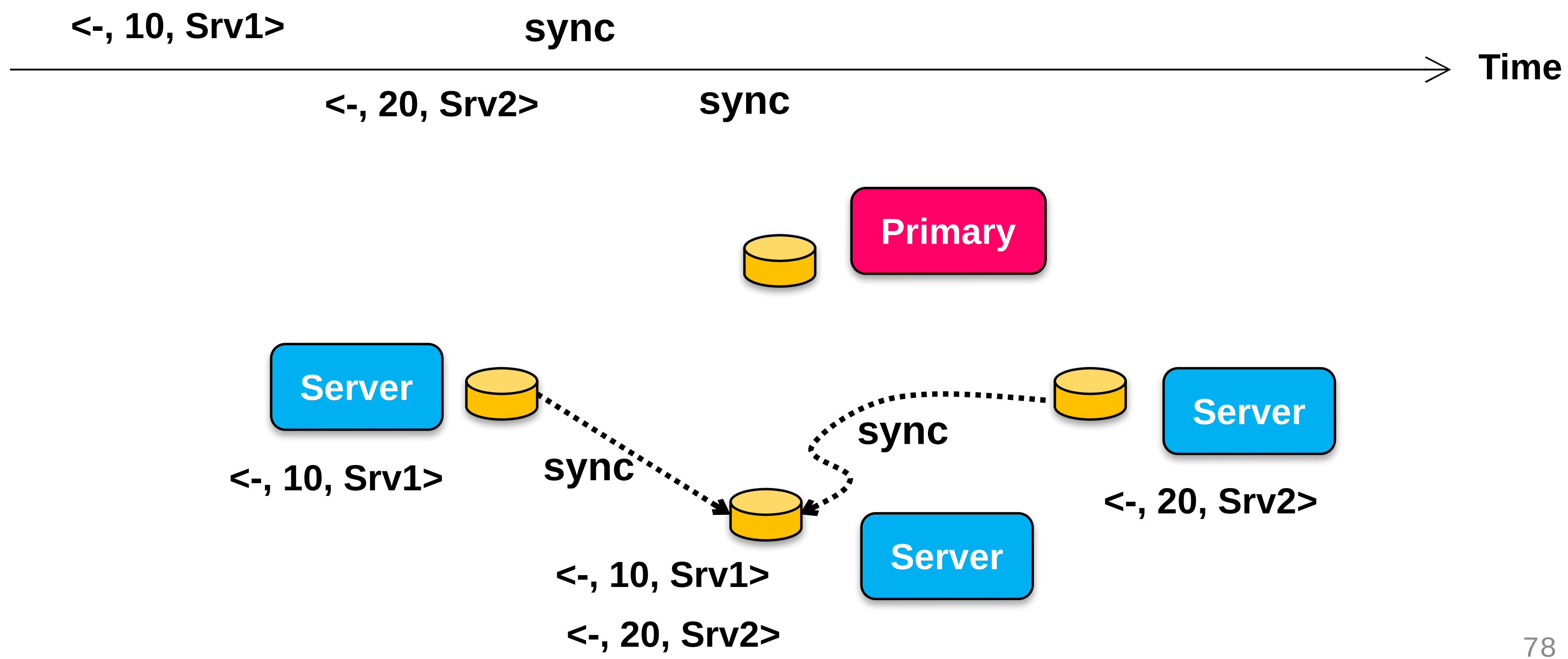
Will CSN matches update order?

Not always. The primary can see newer updates before the old ones, as long as they don't have casualty relationship



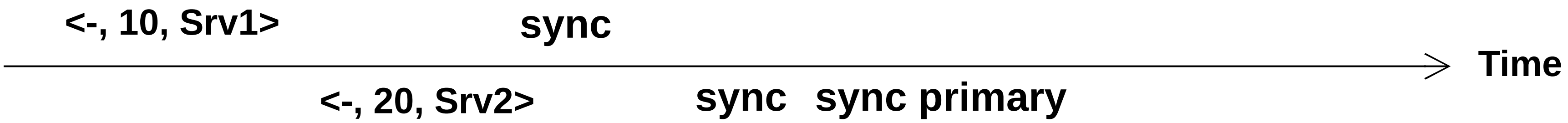
Will CSN matches update order?

Not always. The primary can see newer updates before the old ones, as long as they don't have casualty relationship



Will CSN matches update order?

Not always. The primary can see newer updates before the old ones, as long as they don’t have casualty relationship



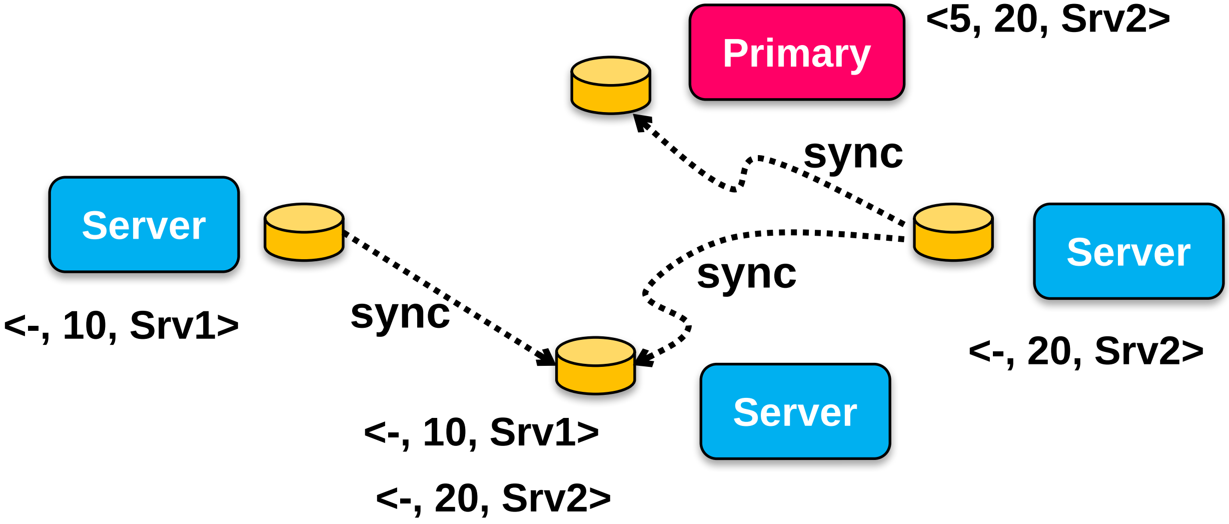
<-, 10, Srv1>

sync

Time

<-, 20, Srv2>

sync sync primary



Primary

<5, 20, Srv2>

sync

Server

Server

<-, 10, Srv1>

sync

sync

<-, 20, Srv2>

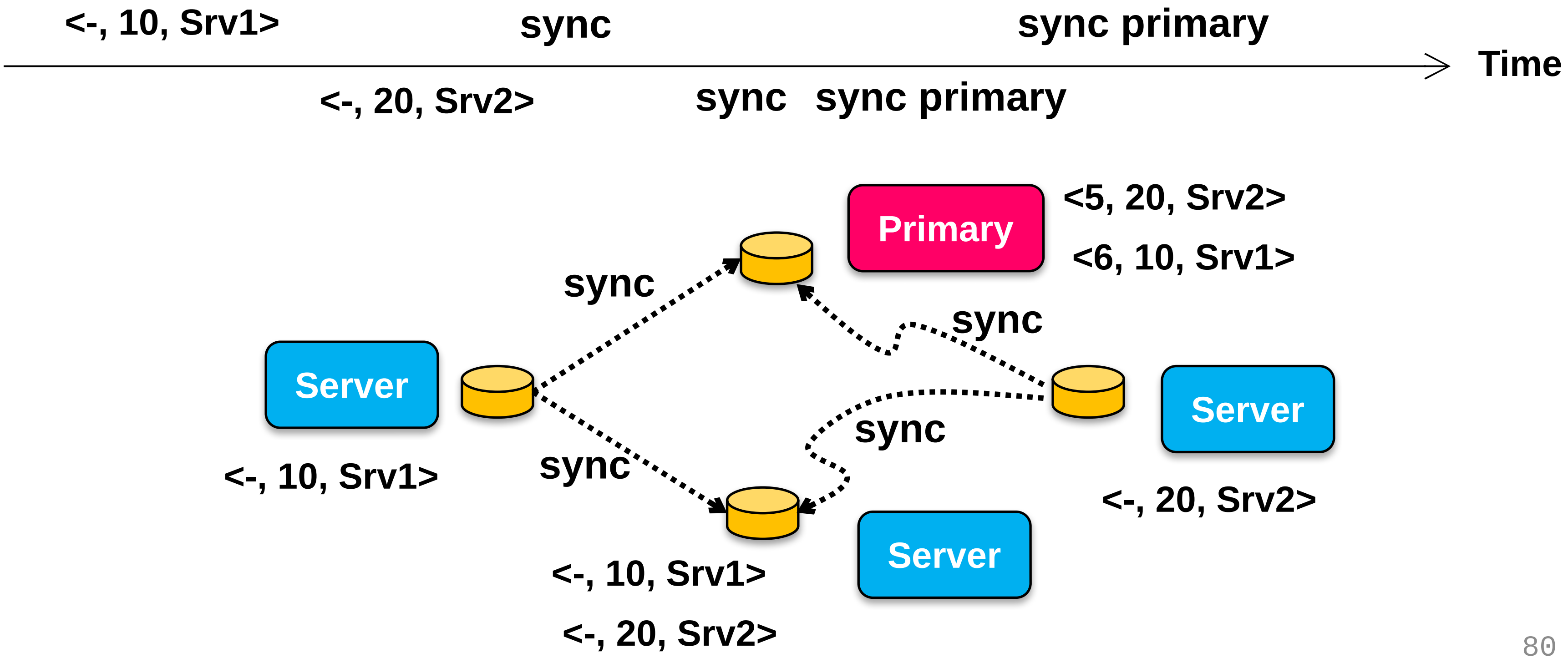
Server

<-, 10, Srv1>

<-, 20, Srv2>

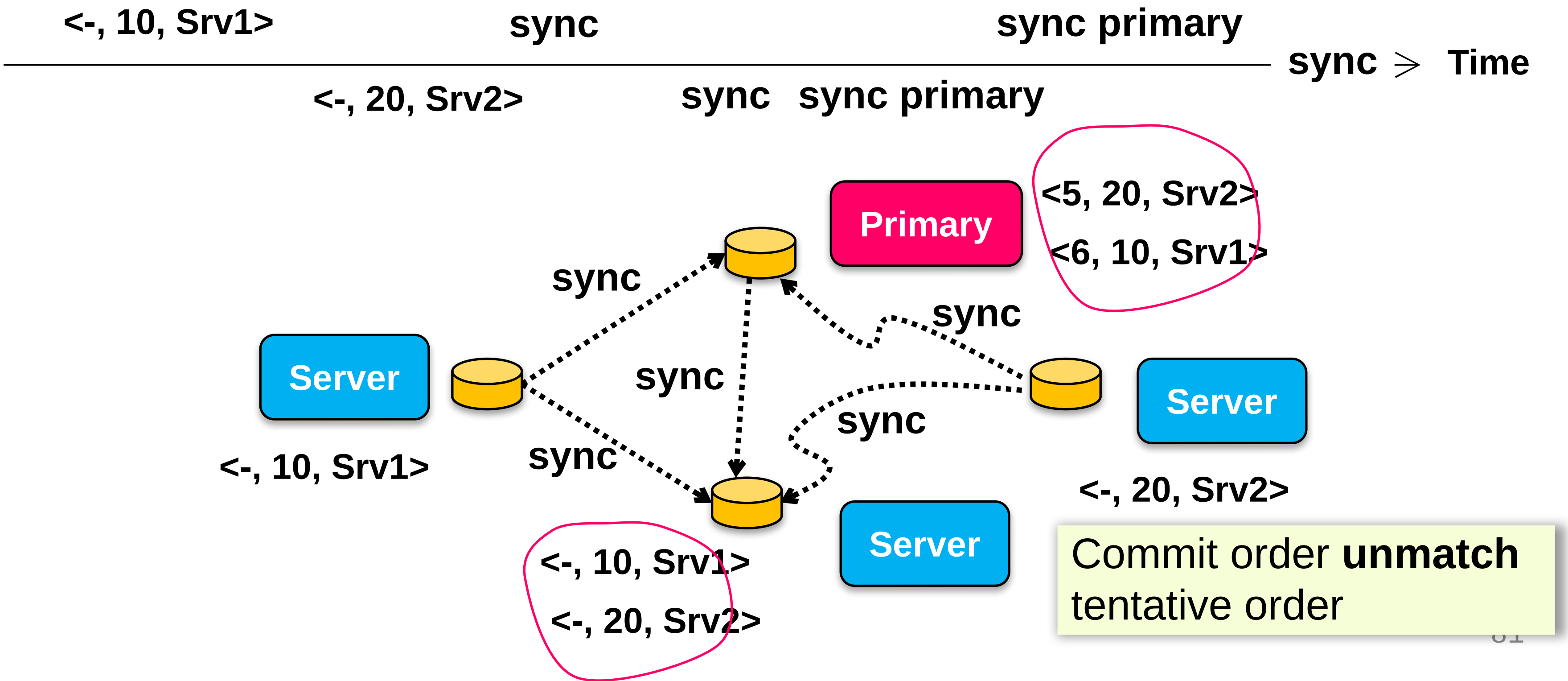
Will CSN matches update order?

Not always. The primary can see newer updates before the old ones, as long as they don't have casualty relationship



Will CSN matches update order?

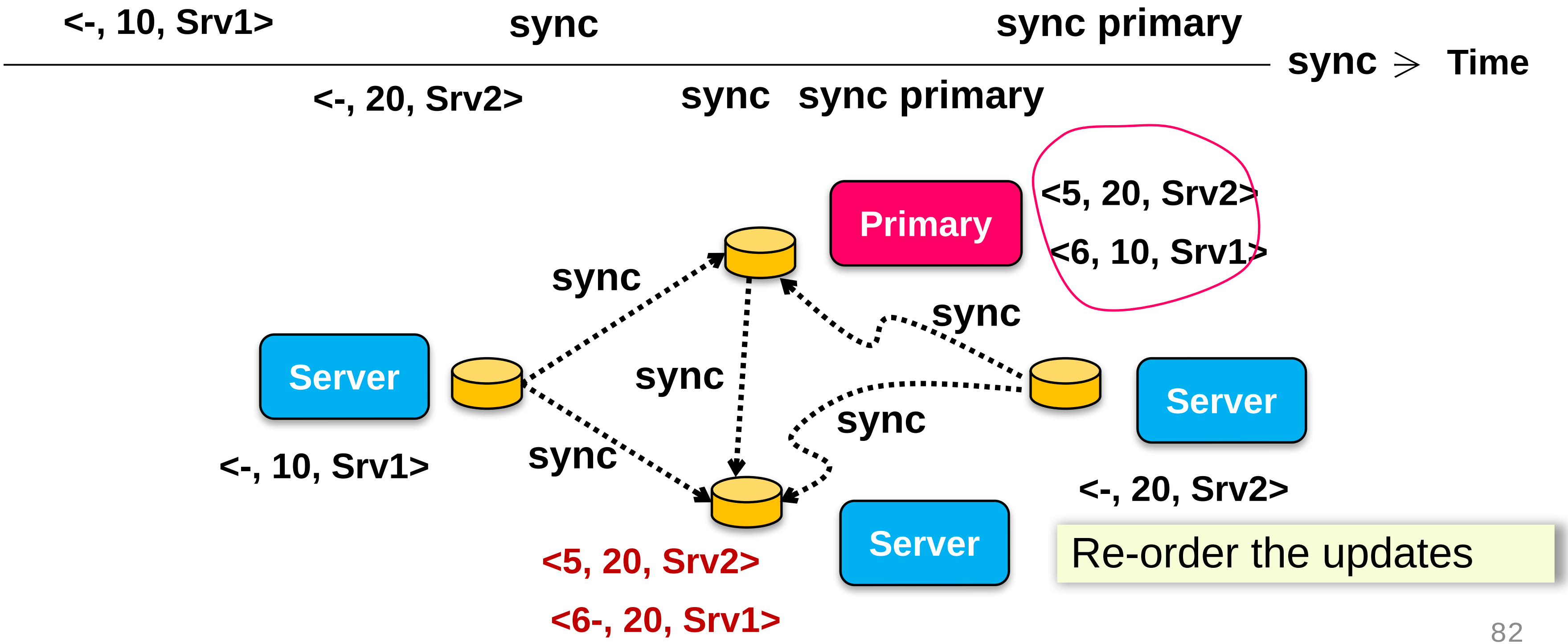
Not always. The primary can see newer updates before the old ones, as long as they don’t have casualty relationship



Commit order **unmatch** tentative order

Will CSN matches update order?

Not always. The primary can see newer updates before the old ones, as long as they don’t have casualty relationship



<-, 10, Srv1>

sync

sync primary

sync

Time

<-, 20, Srv2>

sync

sync primary

Primary

<5, 20, Srv2>

<6, 10, Srv1>

sync

sync

sync

sync

sync

Server

Server

Server

<-, 10, Srv1>

<-, 20, Srv2>

<5, 20, Srv2>

<6-, 20, Srv1>

Re-order the updates

Trimming the log

Logs can be kept long without trimming

When nodes receives new CSNs, can discard all committed log entries seen up to this point

- Result: No need to keep years of log data

Summary so far

Eventual consistency is a weak consistency model

- Result in anomalies (But can be fixed later, e.g., via sync)

Techniques to deal with these anomalies in this lecture comes from Bayou[+]

- Introduced some very influential design ideas
  - Update functions
  - Ordered update log is the real truth, not the storage/database
  - Use lamport clock for casually ordered writes

Techniques everyone should know when consider distributed systems

- Lamport clock, vector clocks, causal relationships

[+] Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System (SOSP'95)

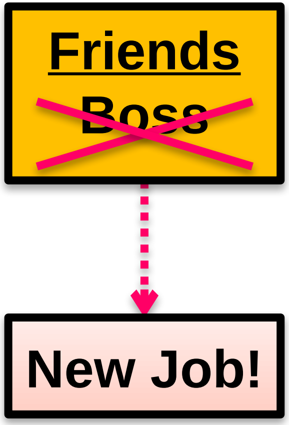
What about reads?

Possible read/write rule implementations

- Read: return the latest local copies of the data
- ~~Write: write locally (and directly returns), propagate the writes to all the servers in background (e.g., upon sync)~~
- Read may return
  - Tentative data
  - Outdated data
- **Trade** read/write consistency for high performance

Reading tentative data is sometimes dangerous

For Users:



Friends

~~**Boss**~~

New Job!

Employment Integrity

For Programmers:

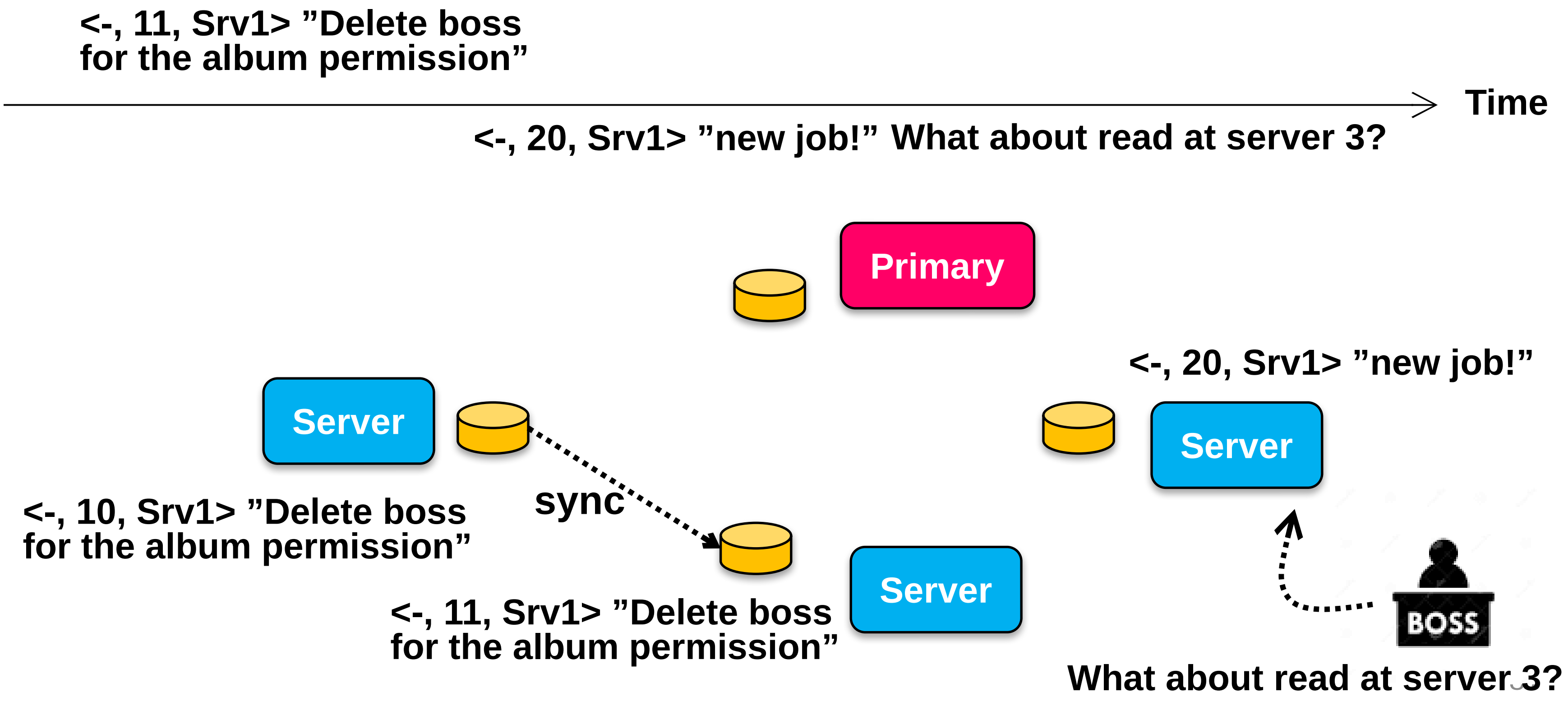
Photo Upload

Add to album

Referential Integrity

Can these happen in our current system?

Reading tentative data is sometimes dangerous



<-, 11, Srv1> ”Delete boss
for the album permission”

Time

<-, 20, Srv1> ”new job!” What about read at server 3?

<-, 20, Srv1> ”new job!”

<-, 10, Srv1> ”Delete boss
for the album permission”

<-, 11, Srv1> ”Delete boss
for the album permission”

What about read at server 3?

Eventual vs. Linearizability

Scalability

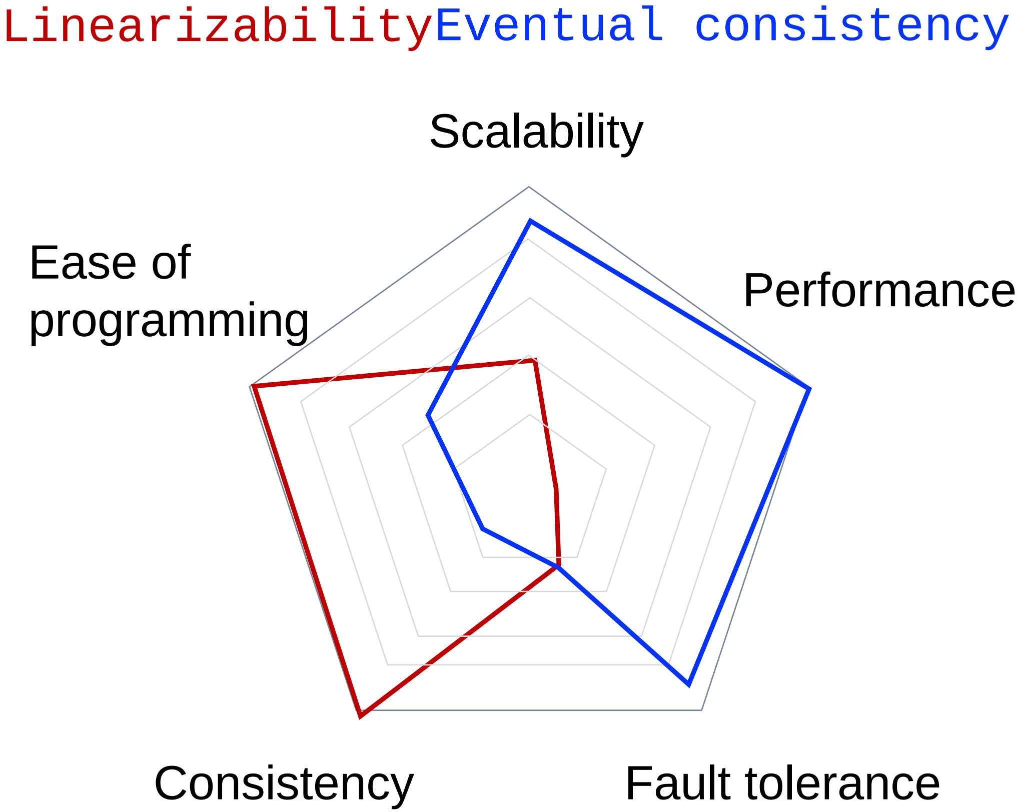
– Still have a single primary (but operations are much lightweight)

Ease of programming

– Not so easy: update function & anomalies to handle

Fault tolerance

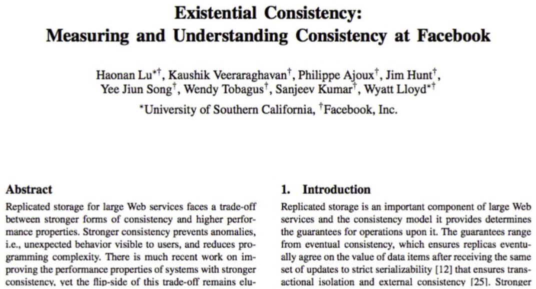
– As long as a single replica is up, client can work



Does eventual consistency anomalies matter?

It depends on the application scenarios

- Frequencies of the anomalies
- Importance of the anomalies



Existential Consistency:
Measuring and Understanding Consistency at Facebook

Haonan Lu[*†], Kaushik Veeraraghavan[†], Philippe Ajoux[†], Jim Hunt[†],
Yee Jiun Song[†], Wendy Tobagus[†], Sanjeev Kumar[†], Wyatt Lloyd[*†]
[*]University of Southern California, [†]Facebook, Inc.

Abstract

Replicated storage for large Web services faces a trade-off between stronger forms of consistency and higher performance properties. Stronger consistency prevents anomalies, i.e., unexpected behavior visible to users, and reduces programming complexity. There is much recent work on improving the performance properties of systems with stronger consistency, yet the flip-side of this trade-off remains elu-

1. Introduction

Replicated storage is an important component of large Web services and the consistency model it provides determines the guarantees for operations upon it. The guarantees range from eventual consistency, which ensures replicas eventually agree on the value of data items after receiving the same set of updates to strict serializability [12] that ensures transactional isolation and external consistency [25]. Stronger

SOSP 2015

For example, do conflicting operations happen a lot

- Facebook has conducted a research to measure the frequencies of anomalies under eventual consistency. Some highlights are:
  - Per-Object sequential Results: 1 anomaly per million reads (user should see their writes)
- A social networking website can tolerate many anomalies

However, many scenarios (e.g., Bank) require stronger models (even stronger than linearizability), see the later lectures.