

Eventual Consistency

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/rong_chen

What is Consistency?

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

Examples

Local memory:

Write x=5 Read x (should be 5)

$X \leftarrow X+1; Y \leftarrow Y+1$ Assert($X+Y == \text{Const}$)

Database:

Single object consistency is also called “coherence”

Consistency across multiple objects (all or nothing)

Consistency Challenges

No right or wrong consistency models

- Tradeoff between ease of programmability and efficiency
(e.g., what's the consistency model for web pages?)

Consistency is hard in (distributed) systems

- Data replication (Caching)
- Concurrency (no shared clock)
- Failures (machine or network)

Consistency so far

Concurrency forces us to think about meaning of read/write

Sequential consistency: everyone sees same read/write order (e.g., IVY)

Release consistency: everyone sees write in unlock order (e.g., TreadMarks)

Recap Consistency

Sequential consistency

- All read/write ops follow some total ordering
- Read must see result of latest write

Release consistency

- Delay changes visible to other processors until certain synchronization access occurs
- Update-base vs. Invalidate-based protocol
- Eager vs. Lazy release

Disadvantages of S/RC

Very slow

- Must ask before each operation
- IVY (sequential): ask master
 - Read and write (Copy_Set, Owner)
- TreadMarks (release): ask lock server
 - Acquire or release (Write Notices)

Disadvantages of S/RC

Require highly **available** connections

- Lots of chatter between master/worker

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., twitter)

Why (not) **Eventual** Consistency?

Support **disconnected** operations

- Better to read a **stale** value than **nothing**
- Better to save writes **somewhere** than **nothing**

Potentially **anomalous** application behavior

- Stale reads
- Conflicting writes
- ...

Eventual Consistency

Sequential vs. Eventual

Sequential: **pessimistic** conflict handling

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

Eventual: **optimistic** conflict handling

- Conflicting writes is **rare** case
- Let updates happen, worry about whether they can be serialized **later**

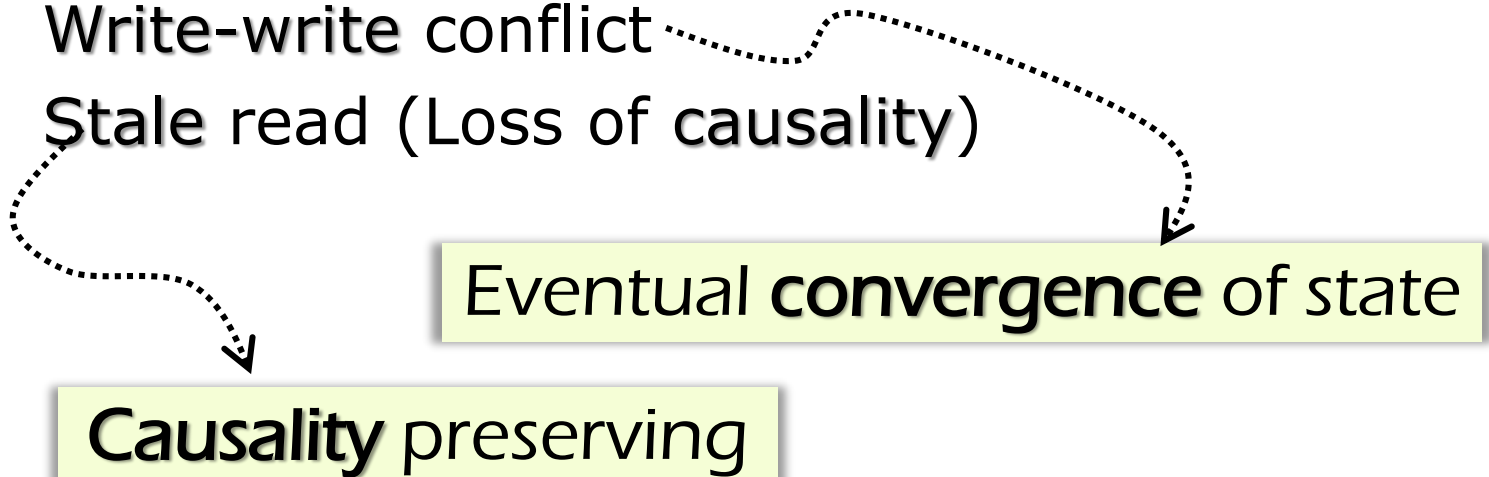
Eventual Consistency

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

Anomalies

- Write-write conflict
- Stale read (Loss of causality)



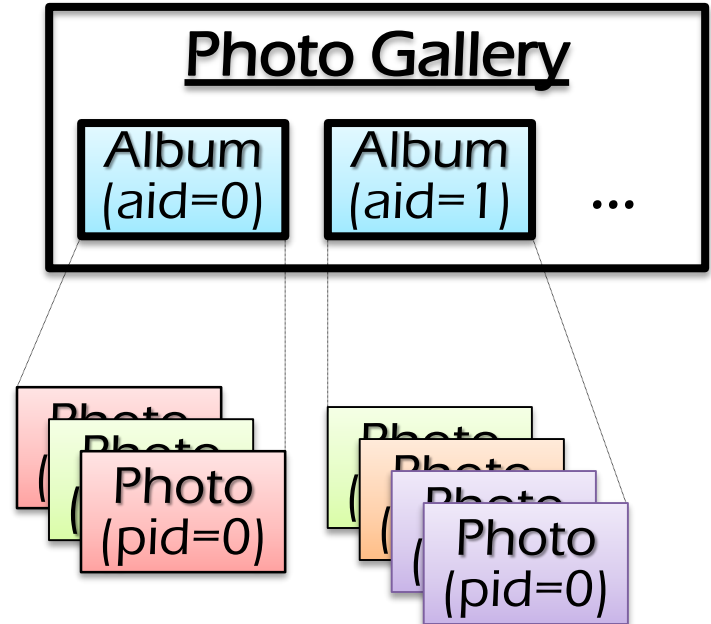
Example: Photo Gallery

App: photo gallery

- Consist of many **albums** (identified by an 'aid')
- Each album contains an ordered list of **photos** (identified by a 'pid')

Operations

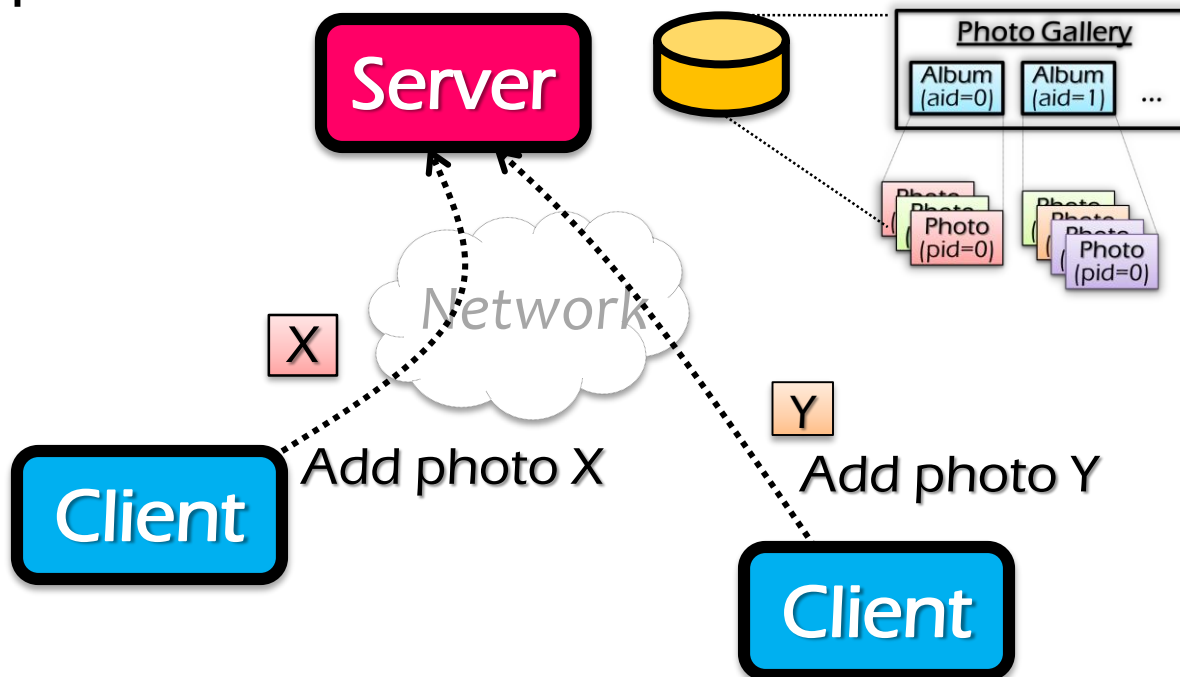
- Add/delete album
- Add photo to album
- ...



Traditional Solution

One server

- Server processes requests one at a time
- Server implicitly choose order for concurrent requests

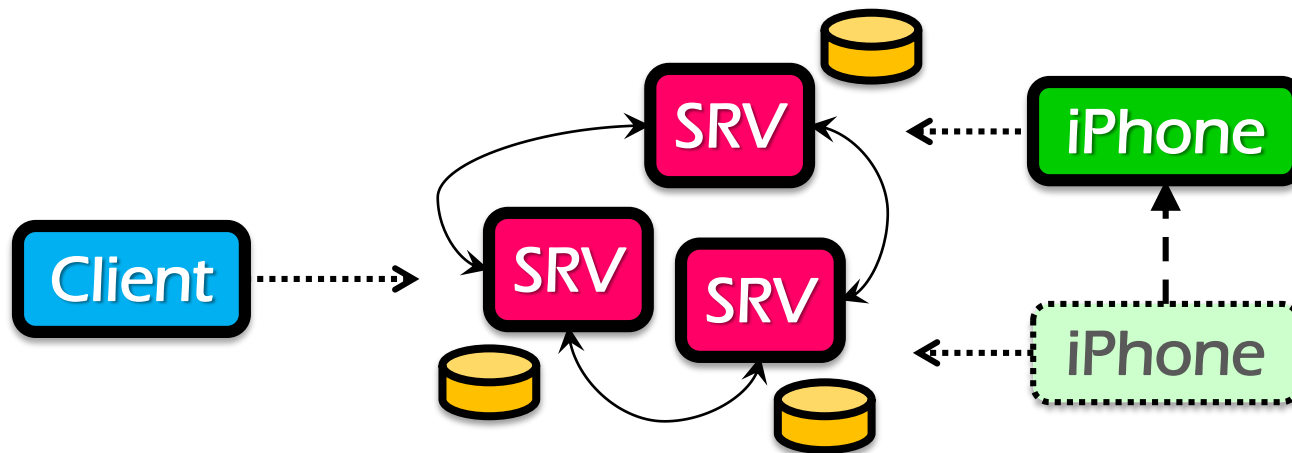


Why not Central Server?

We want to operate on my photo gallery at the iPhone

- *i.e.*, storage replicated in every node
- Modify on any node, as well as read

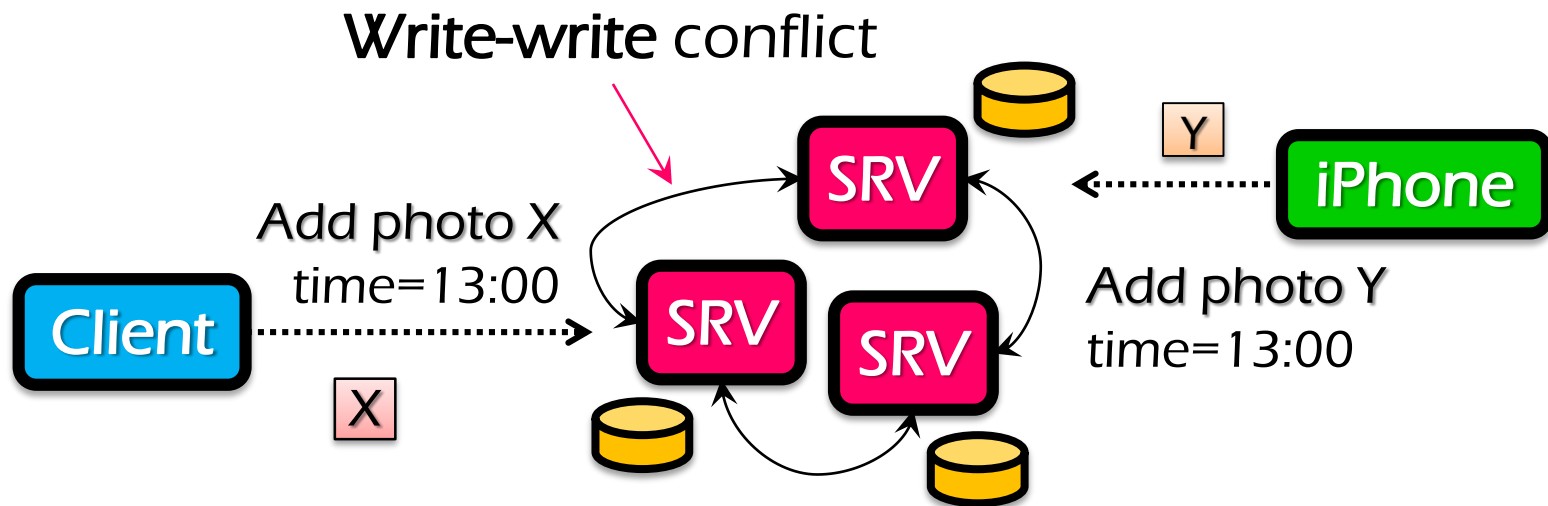
Periodic connectivity to net and other nodes



Straw Man: Merge Storage

What if there's a **write-write** conflict

- Add **two** photos to the same album at the same time



We want **automatic** conflict resolution

Solution: Update Functions

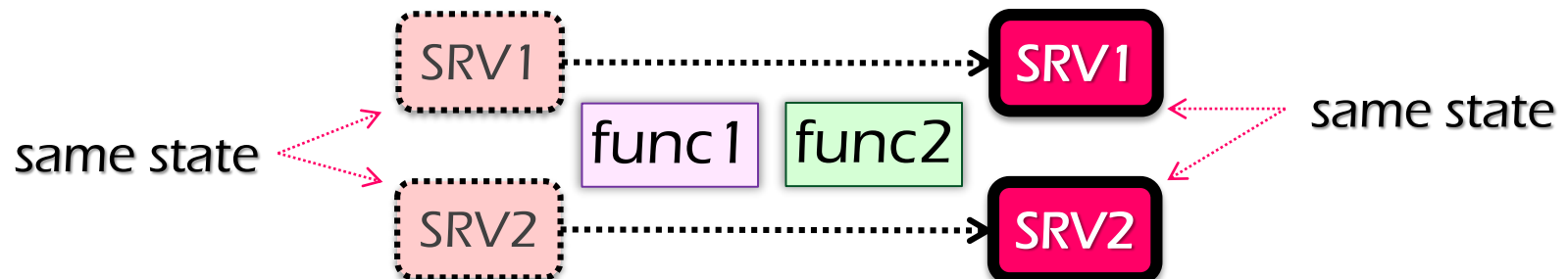
Have **update** be a function, not a new value

- e.g., function: add 'pid' to album with 'aid'

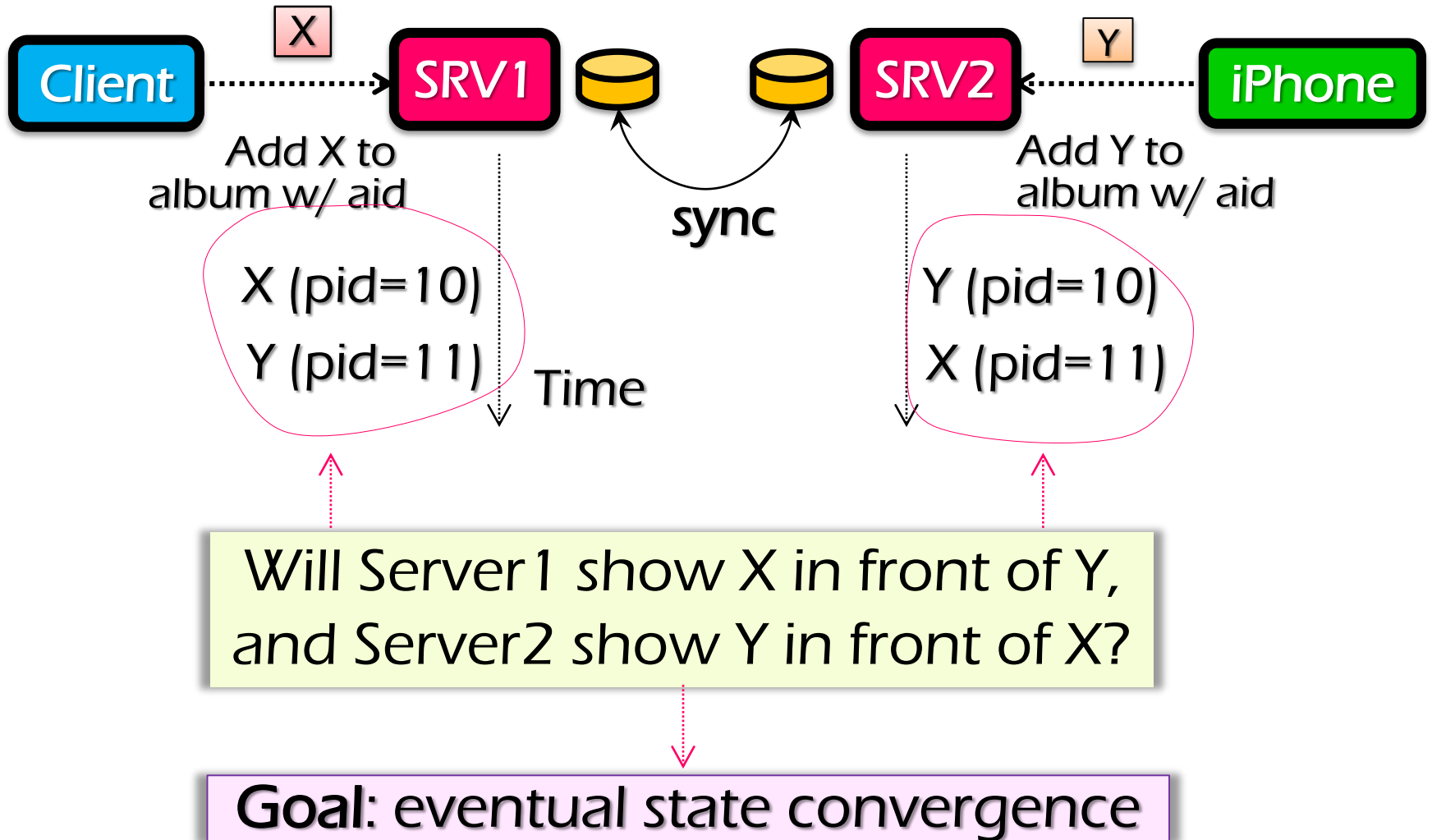
Read **current** state of storage, decide best change

Function must be **deterministic**

- Otherwise nodes will get different answers



Challenge: Ordering



Solution: Ordered Update Log

Ordered list of updates at each node

Storage state is result of applying updates in order

Syncing: ensure both nodes have the same updates in log

How can nodes **agree** on **update order**?

Solution: Ordered Update Log

Update ID: **<time T, node ID>**

Assigned by node that creates the update

Ordering update **X** and **Y**:

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

or

$(X.T = Y.T \text{ and } X.ID < Y.ID)$

Solution: Ordered Update Log

Example:

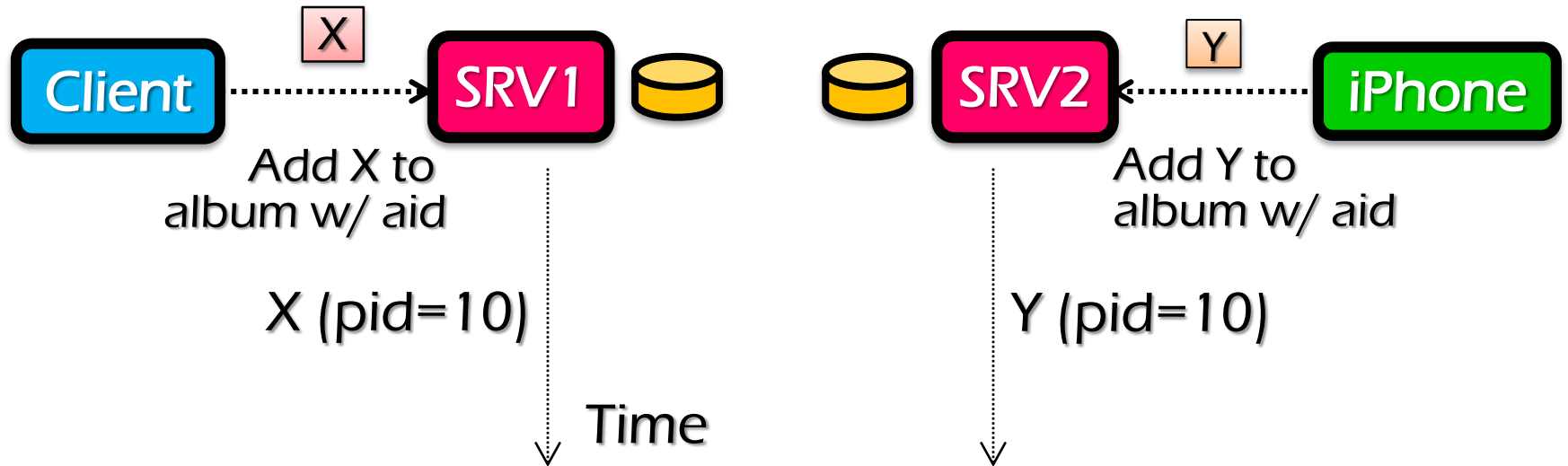
<10, Srv1>: Photo **X** at 10th or 11th position for album#1

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

Question: What's the correct final outcome?

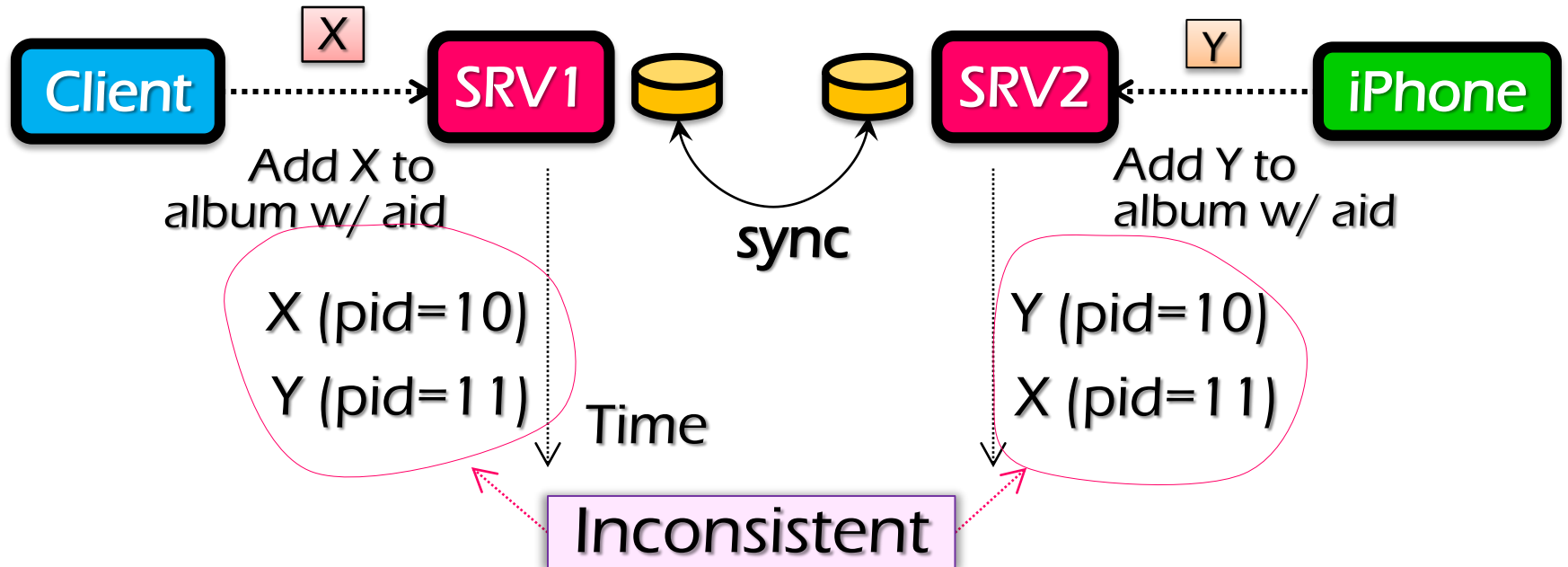
Photo **X** at 10th and Photo **Y** at 11th position

Example: Photo Gallery



The client will see before syncing

Example: Photo Gallery



If just run the update function against storage state

Solution: Rollback and Replay

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

We will optimize this in a bit

Challenge: Clock Time

Will update order be consistent with **wall-clock** time?

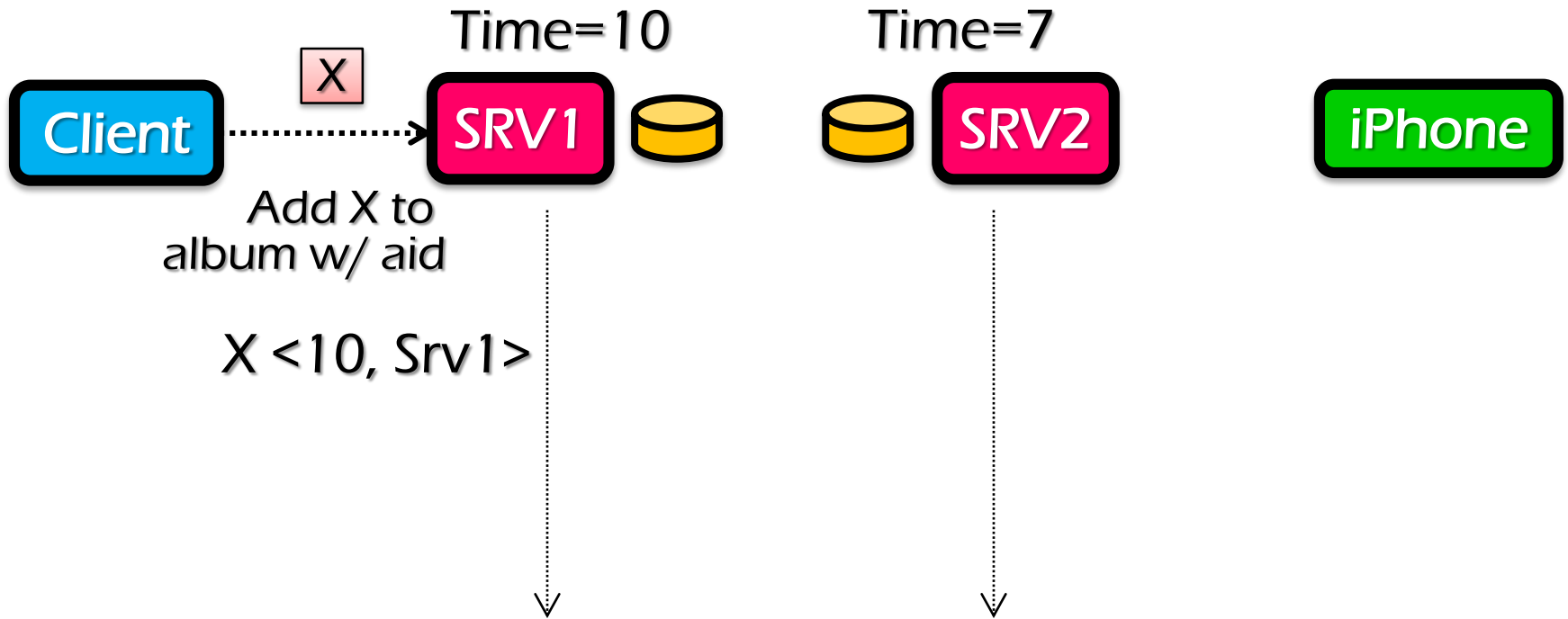
- Srv2's photo gets priority, even Srv1 post first
→ Wall-clocks are not perfectly synchronized

Example:

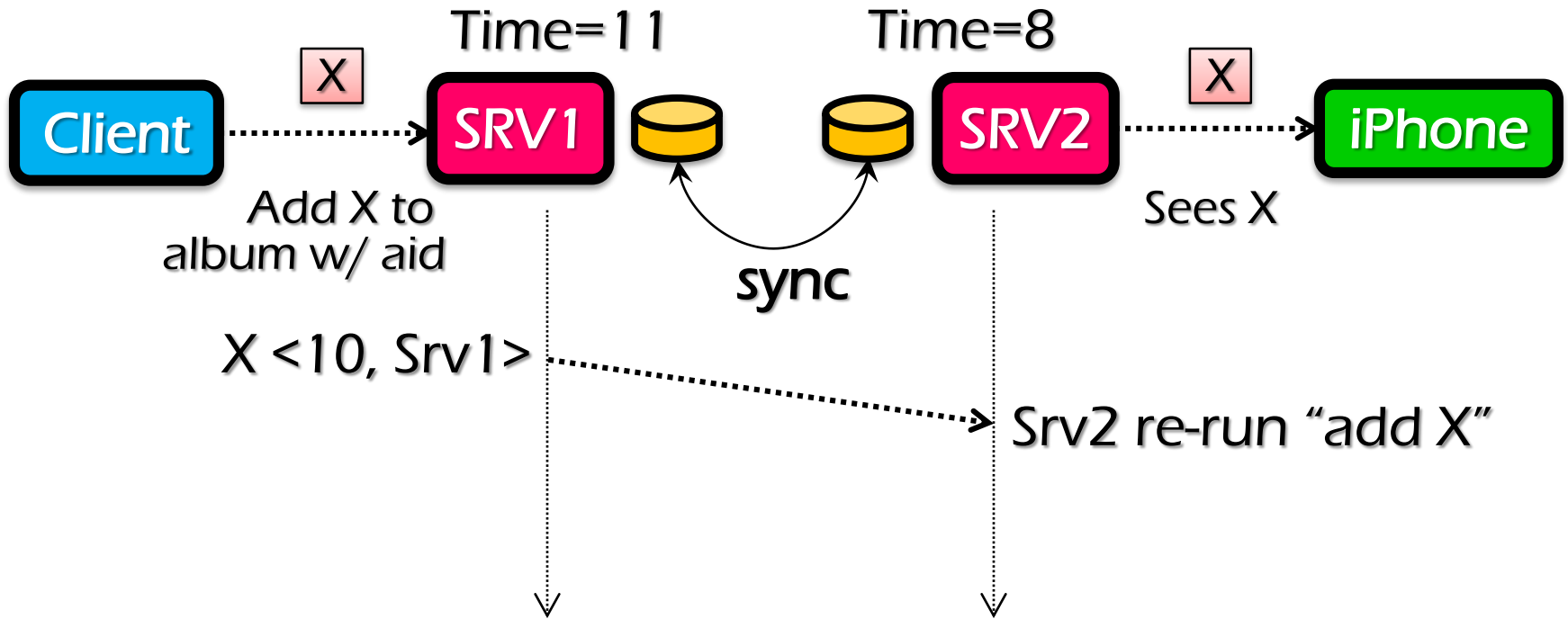
Srv1 went first in **wall-clock** time with **<10, Srv1>**

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

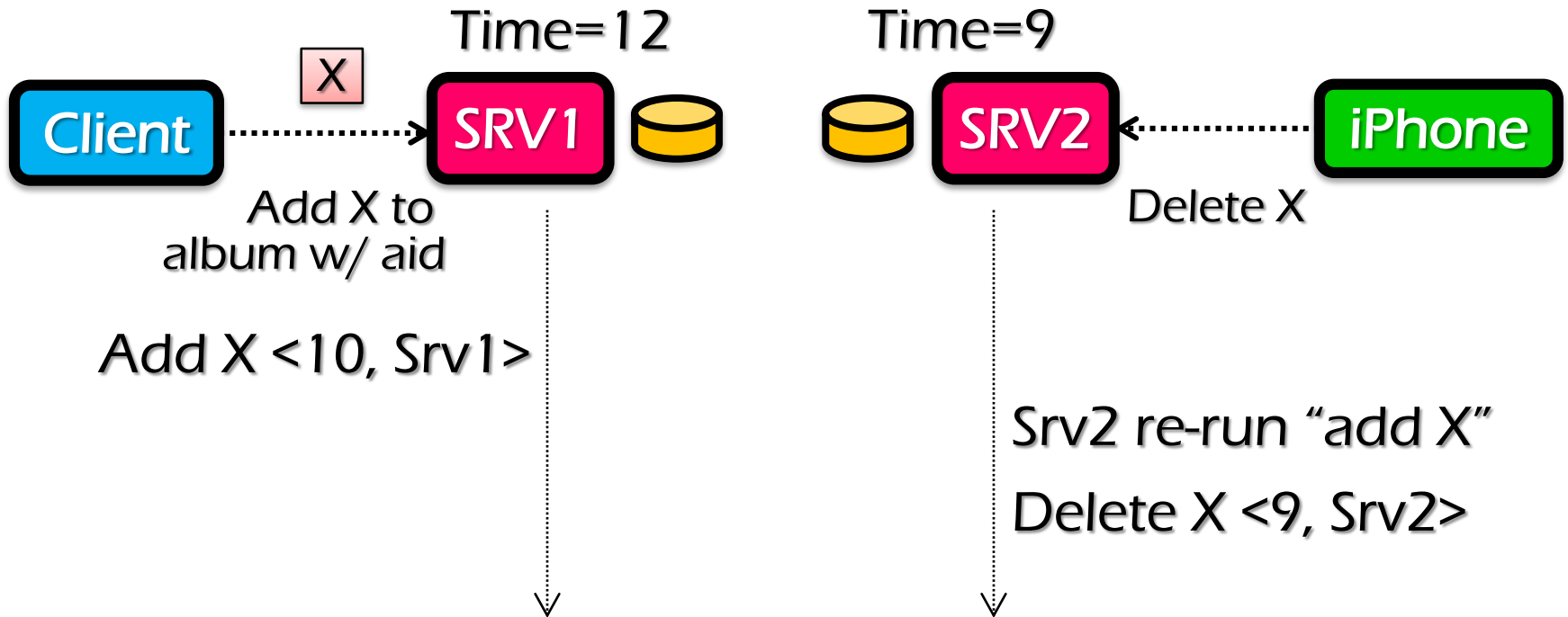
Example: Photo Gallery



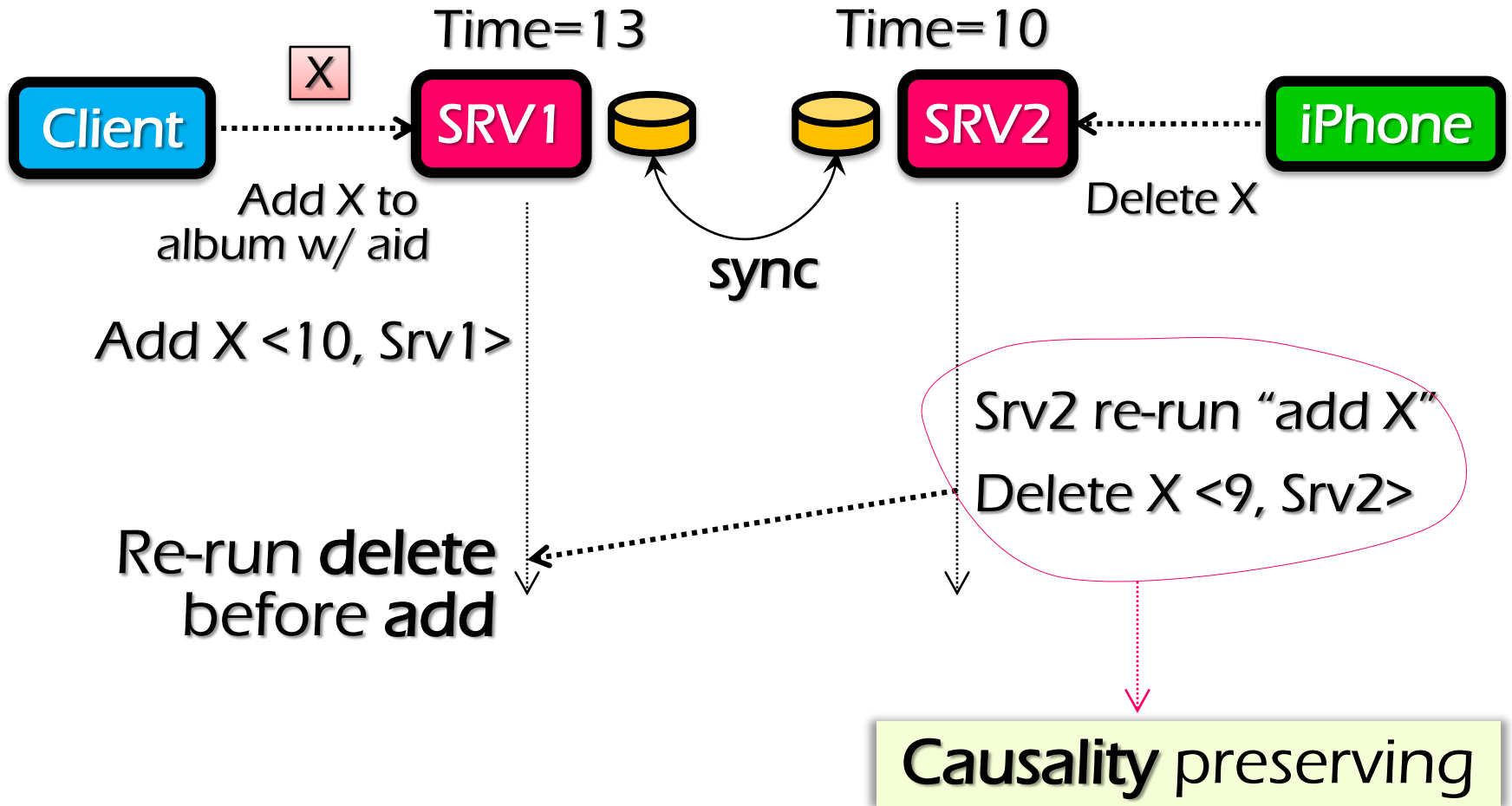
Example: Photo Gallery



Example: Photo Gallery



Example: Photo Gallery



Solution: Logical Clock

We need causality timestamp

- if observe event#1, then generate event#2
→ **$TS(e1) < TS(e2)$**
- if event#1 and event#2 on the same server
→ **$TS(e1) < TS(e2)$**
- All servers will order events according TS

Solution: Logical Clock

Lamport logical clock

1. Each server keeps a clock **T**
2. Increments **T** as real time passes, one second per second
3. Modify **$T = \text{Max}(T, T' + 1)$**
if sees **T'** from another server

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

Srv2: Sees "add X", then update $T_2 = \text{Max}(T_2, 11)$

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

Now, **causality** preserving!

Challenge: Tentative

Displayed photo positions are “tentative”

- **Never know** if there’s some other photo from servers you haven’t yet synced
- Long-delayed updates with lower TS
→ the results of server to change

Would be **nice** if updates were eventually “**stable**”

- Results can never change again
e.g., photo X is at position 10
- No need to re-run update function

Solution: De-centralized

Commit scheme

1. Sync always send in log order
2. If you have seen updates with **TS > N** from every node
 - Never again see log lower **<N, Srv>**
 - **<N, Srv>** is stable

Question?

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

Solution: Centralized

Commit scheme

Commit-Seq-No

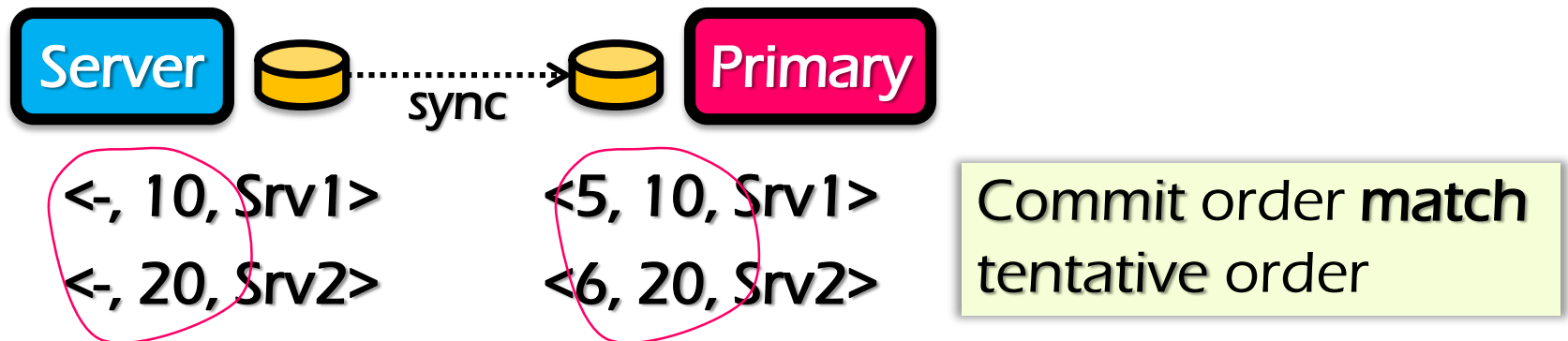
1. One server designated “primary”
 - Assign a total commit order: **CSN**
 - Complete timestamp: **<CSN, local-TS, SrvID>**
 - Any write with a known **CSN** is stable
2. All **stable** writes are ordered before **tentative** writes
3. CSNs are exchanged between servers
 - CSNs define a **total order** for committed update

Solution: Centralized

Question: will commit order match tentative order?

- Often “YES”
- Syncs send in log order (including updates learned from other servers)

Example:

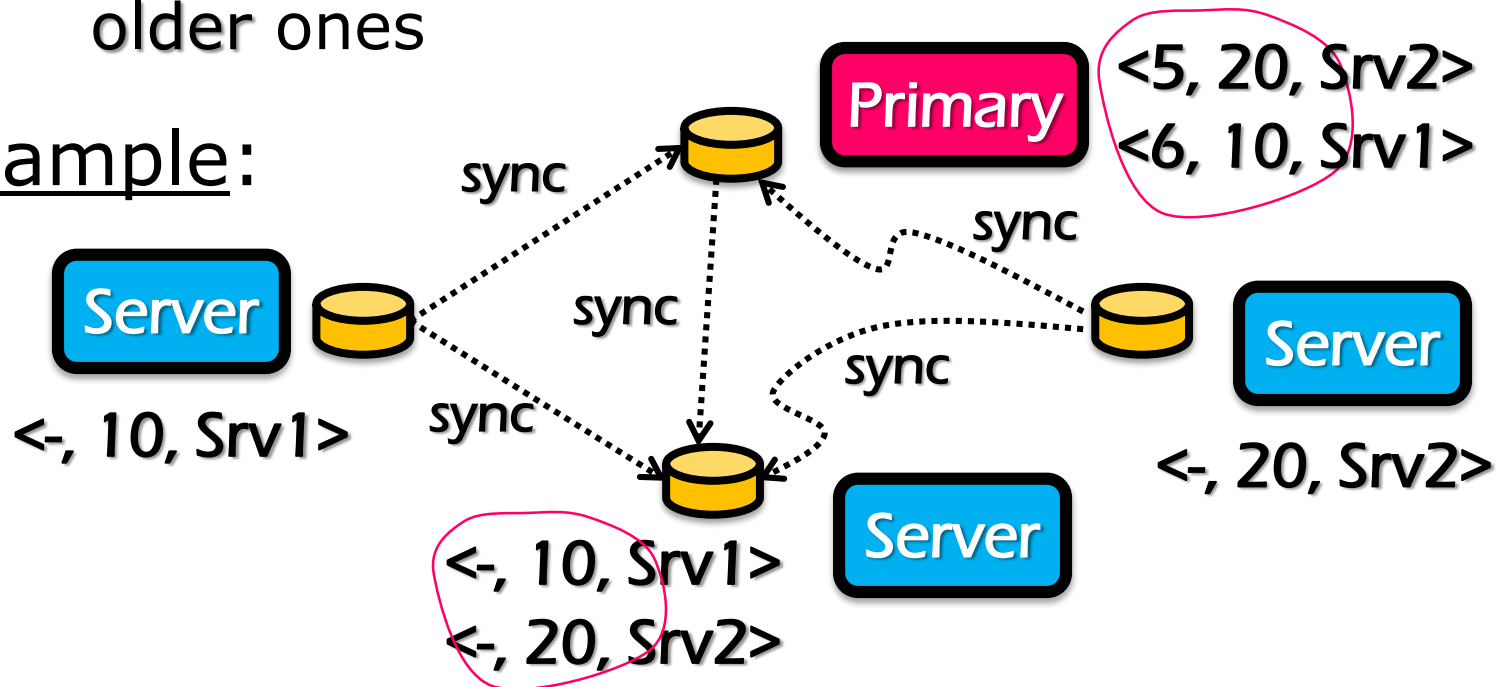


Solution: Centralized

Question: will commit order always match tentative order?

- "NO"
- Primary may see newer updates before older ones

Example:

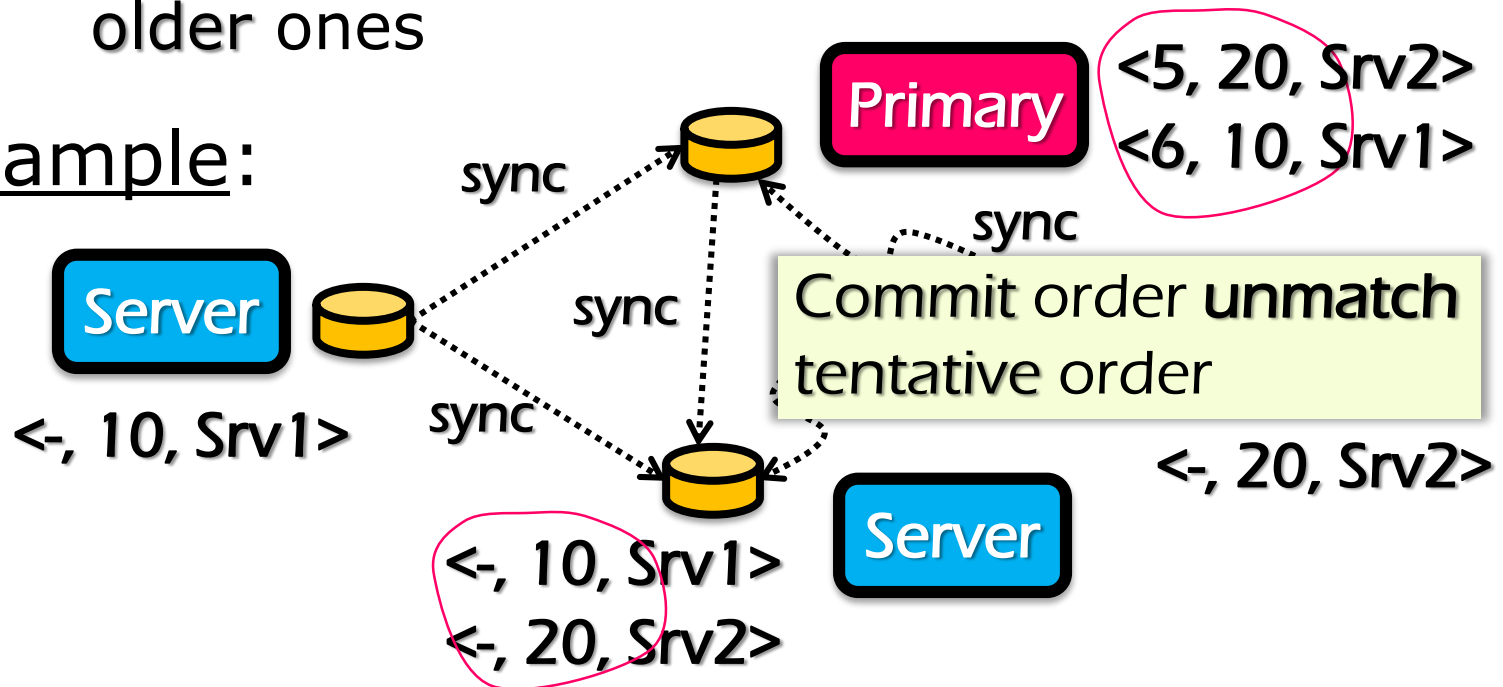


Solution: Centralized

Question: will commit order always match tentative order?

- "NO"
- Primary may see newer updates before older ones

Example:



Summary

Bayou⁺ introduced some very influential design ideas

- Update functions
- Ordered update log is the real truth, not the storage/database
- Allowed general purpose conflict resolution

Bayou made good use of existing idea

- Eventual consistency
- Logical clock

Summary

Replicas, write any copy, update function , and sync are good idea

- Now widely used by both user apps and multi-site storage systems

Central commit server seems reasonable

- *i.e.*, you don't need pure peer-to-peer commit
- Protocol much simpler since central server does all resolution

COPS: Scalable Causal Consistency for Wide-Area Storage

Scalable Causal Consistency



Stores:

Status Updates
Likes
Comments
Photos
Friends List



Stores:

Posts
+1s
Comments
Photos
Circles

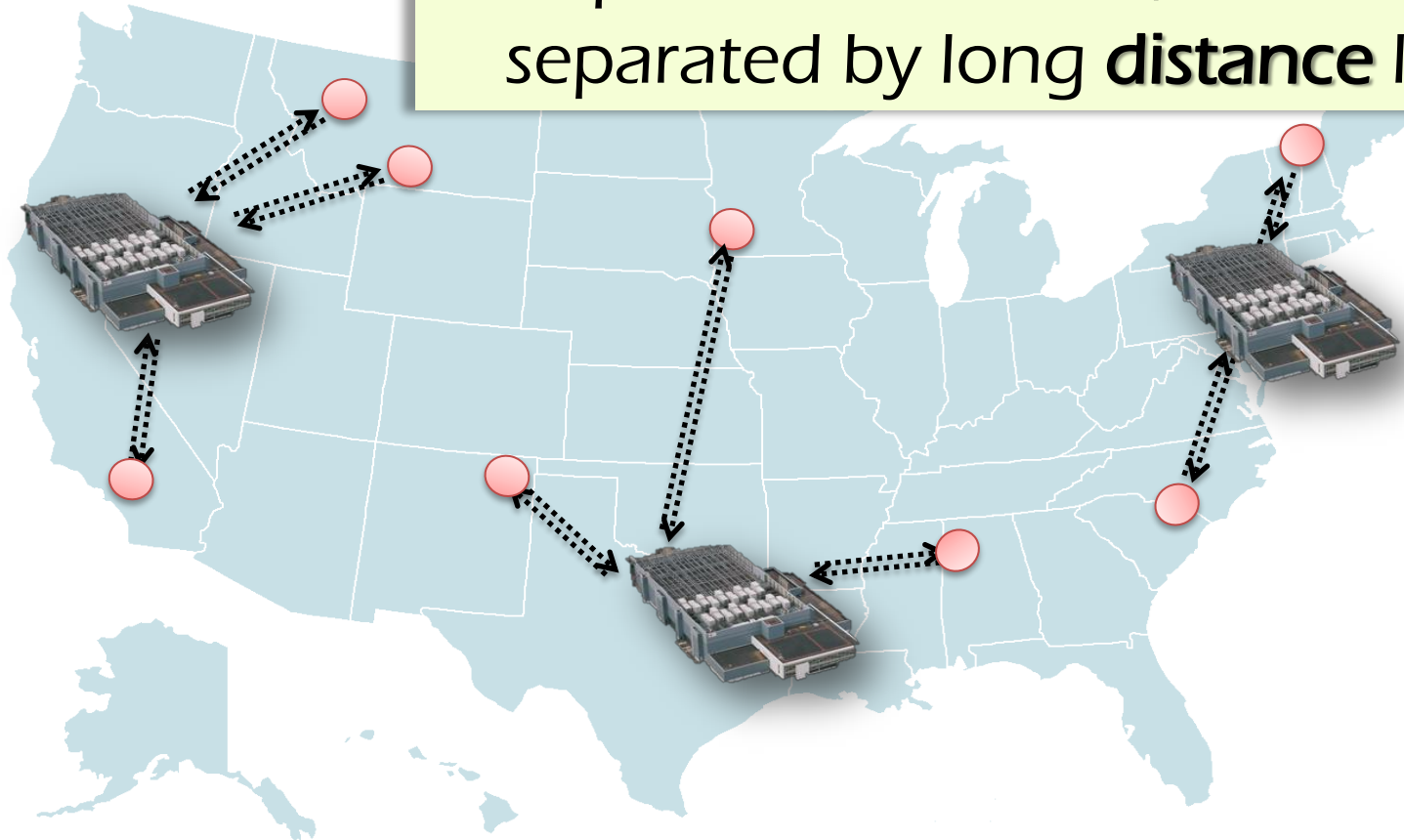


Stores:

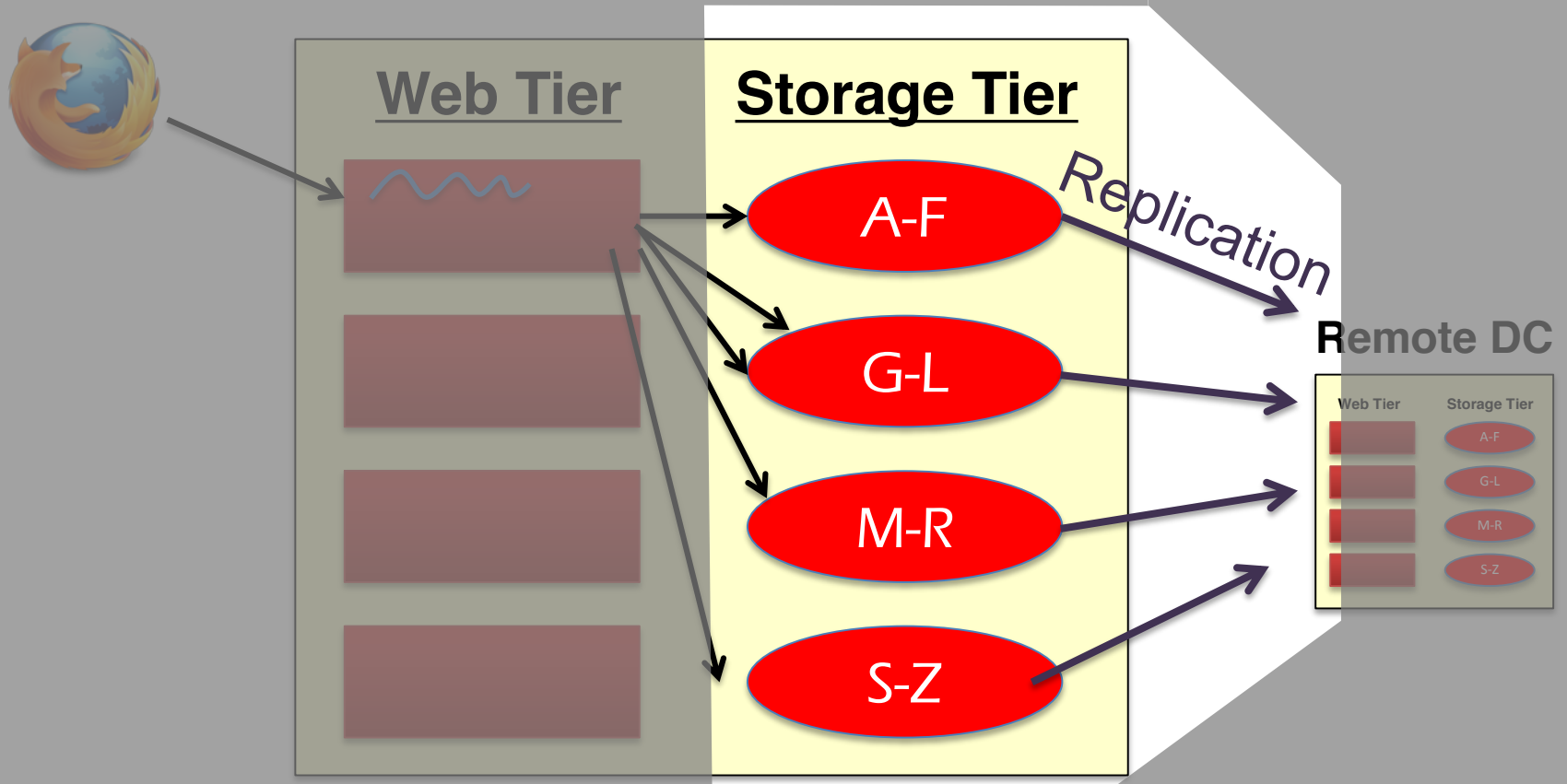
Tweets, Favorites,
Following List

Wide-Area Storage

Multiple **Datacenters**,
separated by long **distance** links



Inside the Datacenter



Each DC has many nodes

Storage state is **fully replicated** at each DC

Desired Properties

	Bayou	COPS
A vailability	✓	✓
L ow Latency	✓	✓
P artition Tolerance	✓	✓
S calability	✗	✓
<i>Causal</i> + Consistency	✓	✓

ALP = “always on”

Basic COPS Summary

Async: serve operations locally, replicate in background

- Always On

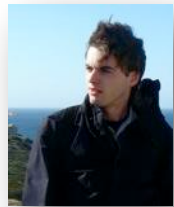
Control replication with dependencies

- Causal+ consistency

Partition key-space onto many nodes

- Scalability

Causality by Example

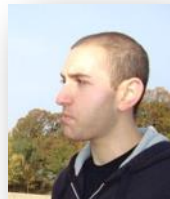


Remove boss from friends group

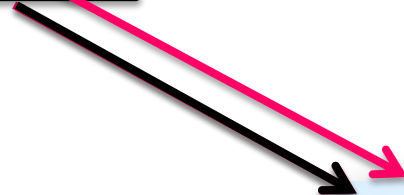


Post to friends:

"Time for a new job!"

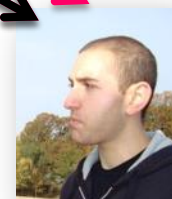


Friend reads post



Causality : →

Thread-of-Execution
Gets-From
Transitivity



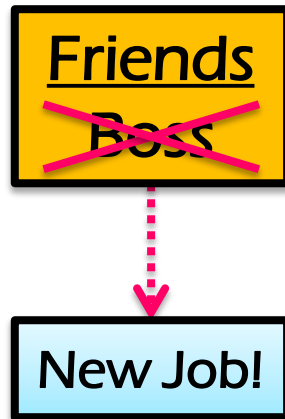
Causal Consistency

Partial orders

- if op1 and op2 are in the single thread of execution and op1 is issued before op2
→ op1 → op2 (**Thread-of-Execution**)
- if op2 read the results written by op1
→ op1 → op2 (**Get-from**)
- if op1 → op2, and op2 → op3
→ op1 → op3 (**Transitivity**)

Causal is Useful

For Users:



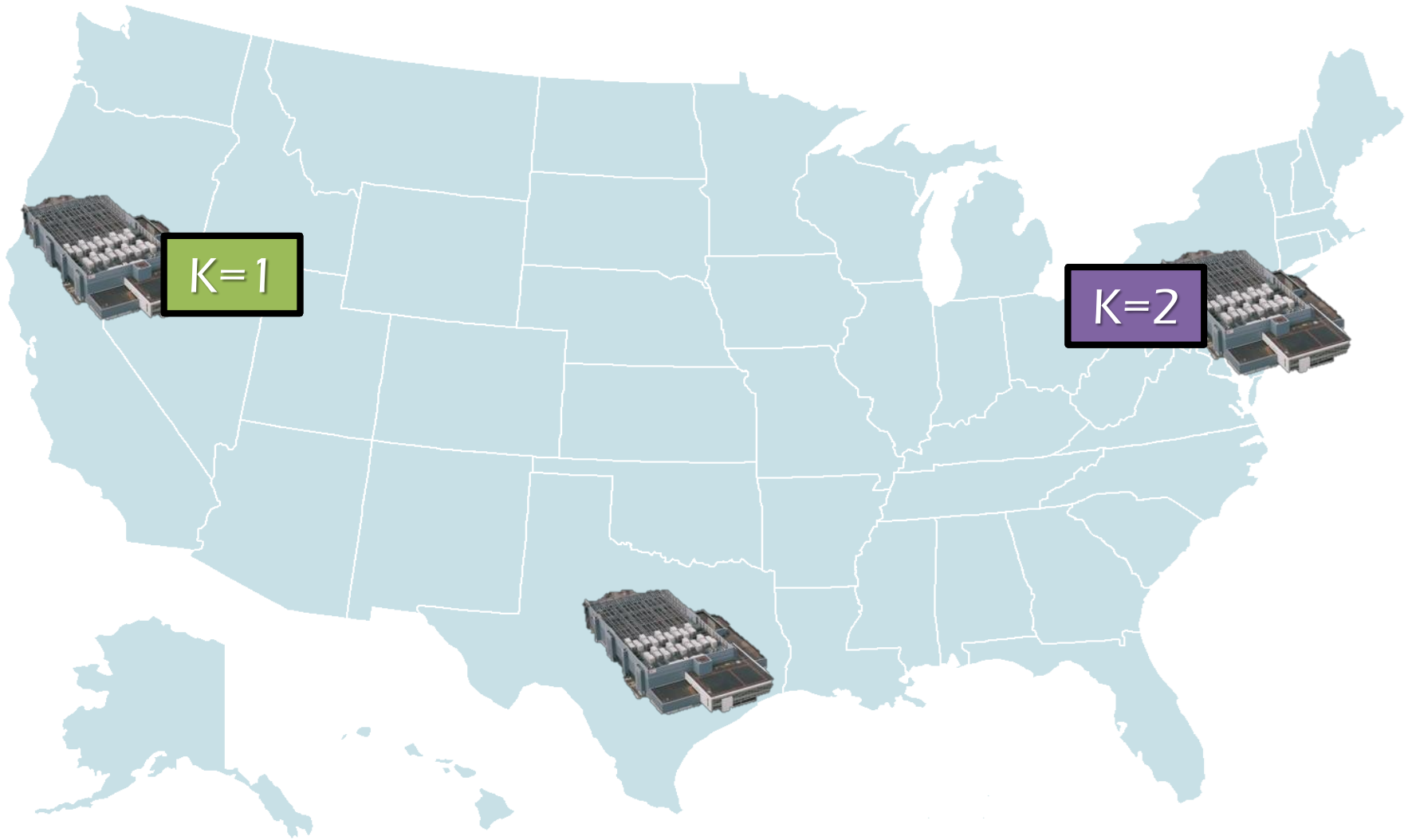
Employment Integrity

For Programmers:

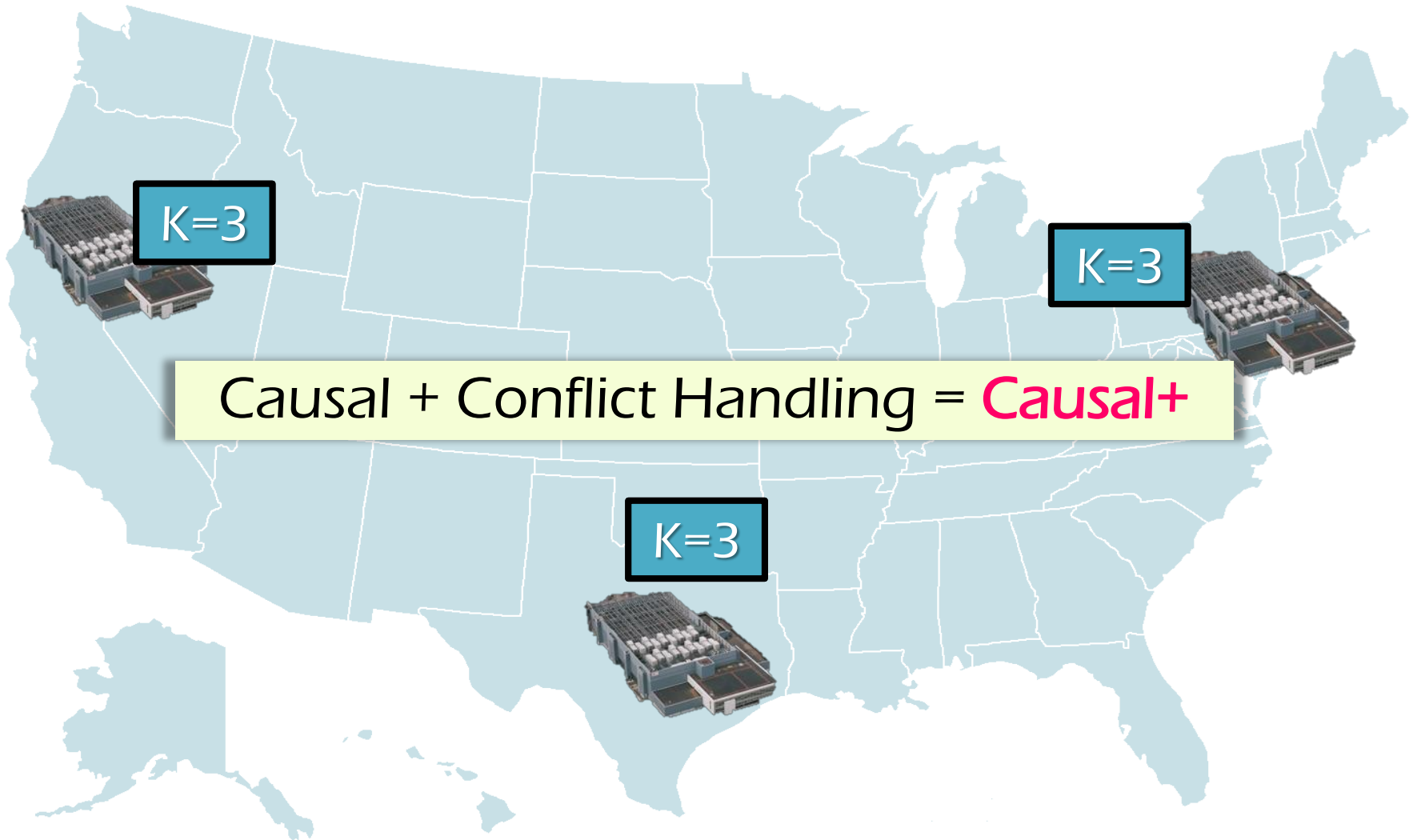


Referential Integrity

Conflict in Causal

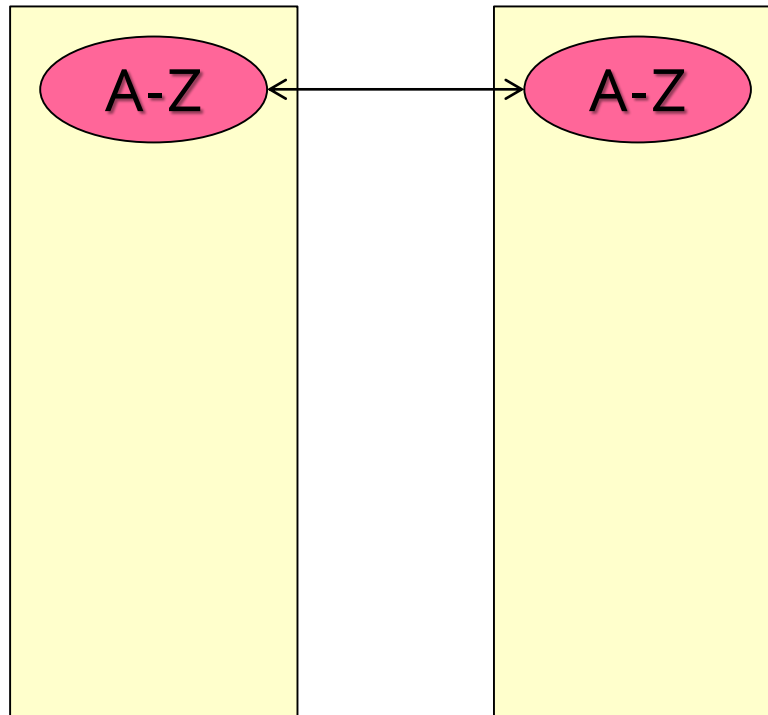


Conflict in Causal



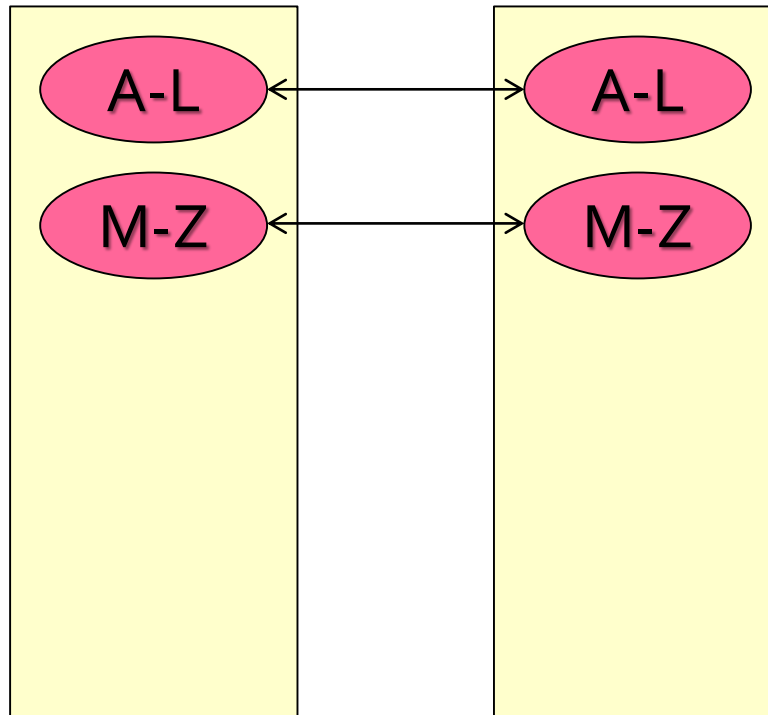
Scalability

Increase **capacity** and **throughput** in each DC



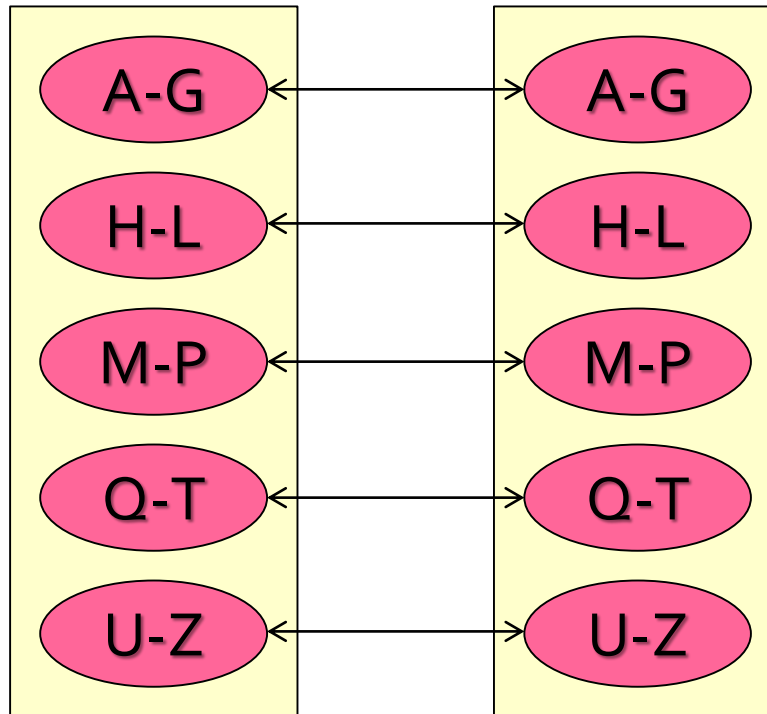
Scalability

Increase **capacity** and **throughput** in each DC



Scalability

Increase **capacity** and **throughput** in each DC



Previous Causal+ Systems

Bayou '95, TACT '00, PRACTI '06

- Log-exchange based

Log is single serialization point

- Implicitly captures and enforces causal order
- Limits scalability
 - One node store all keys
 - Provide log for all ops across a cluster
- No cross-site causality

Scalability Key Ideas

Dependency metadata explicitly capture causality

Distributed verifications replace single serialization

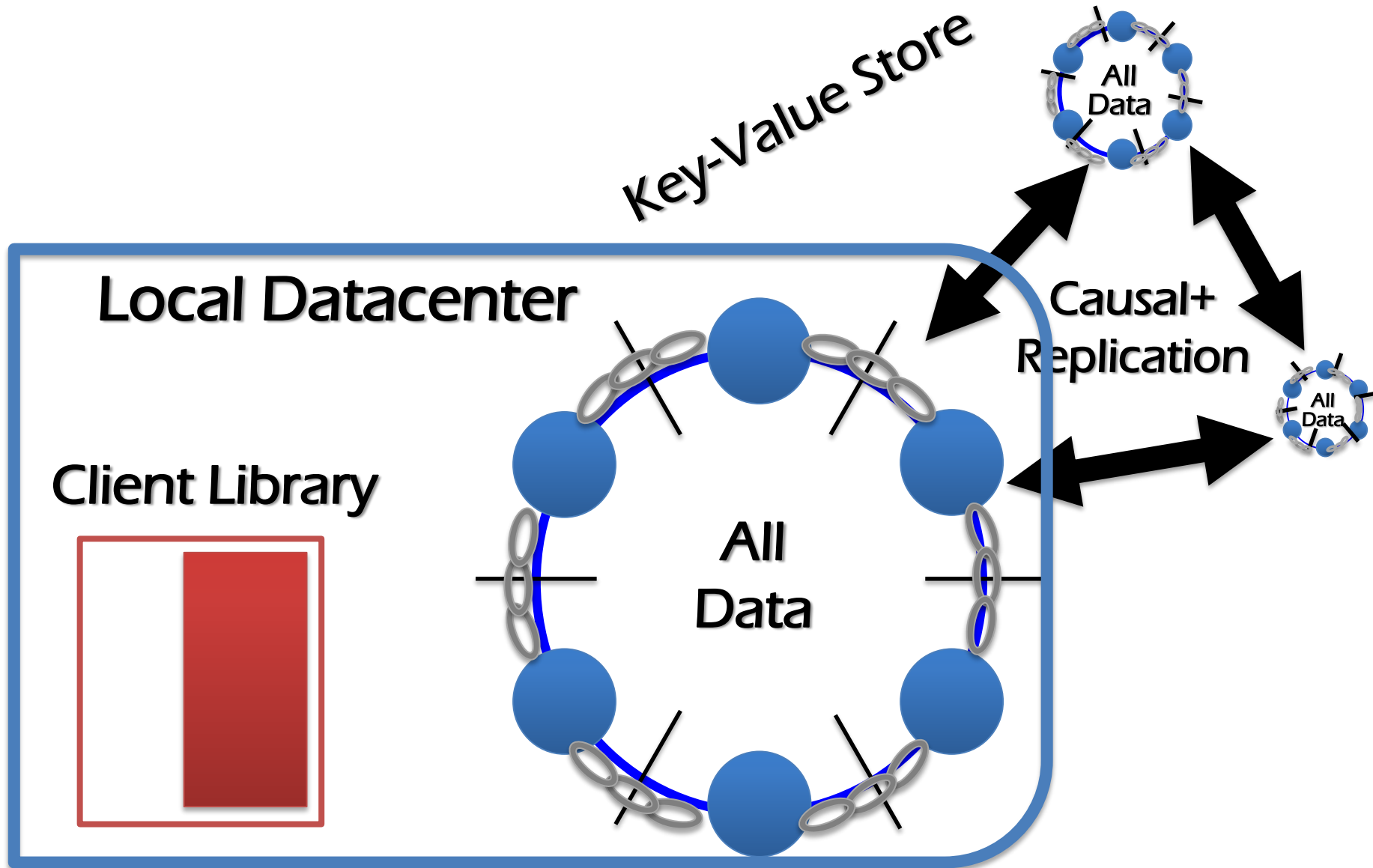
- Delay exposing replicated puts until all dependencies are satisfied in the datacenter

Explicit Dependency Propagate

Explicitly keep track explicit dependencies (partial order) for each write

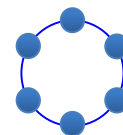
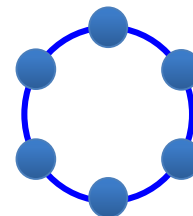
- Site A performs a write, replicates it together with the dependencies to another site B
- Site B waits until the write's dependencies are satisfied in B before committing the write

Architecture



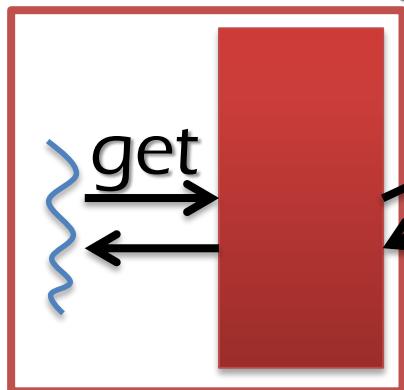
Get

Key-Value Store

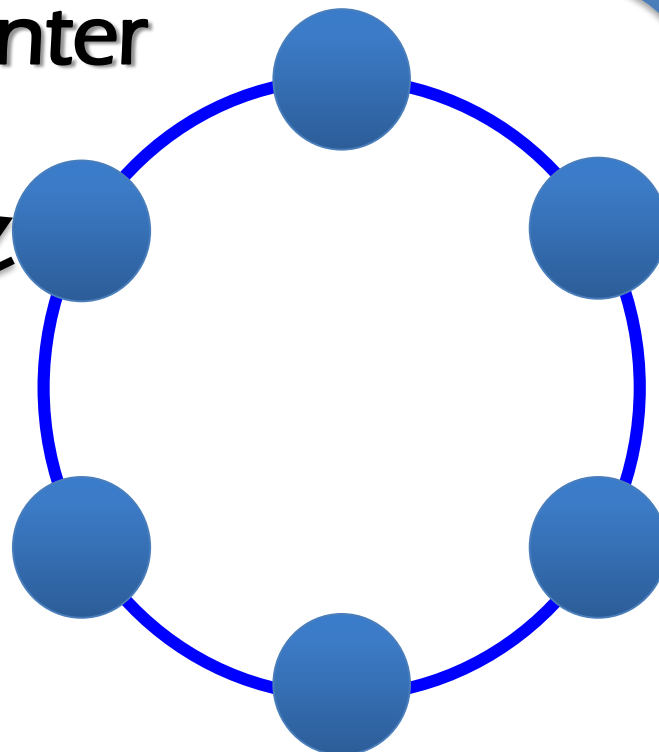


Local Datacenter

Client Library



get

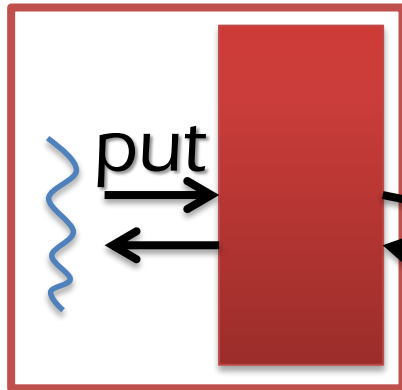


Put

$\text{put}_{\text{after}} = \text{put} + \text{ordering metadata}$

Local Datacenter

Client Library



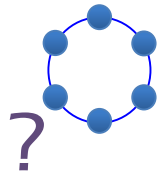
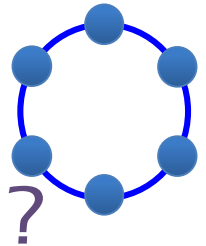
put_after

K:V

Replication Q

put
after

Key-Value Store



Dependency

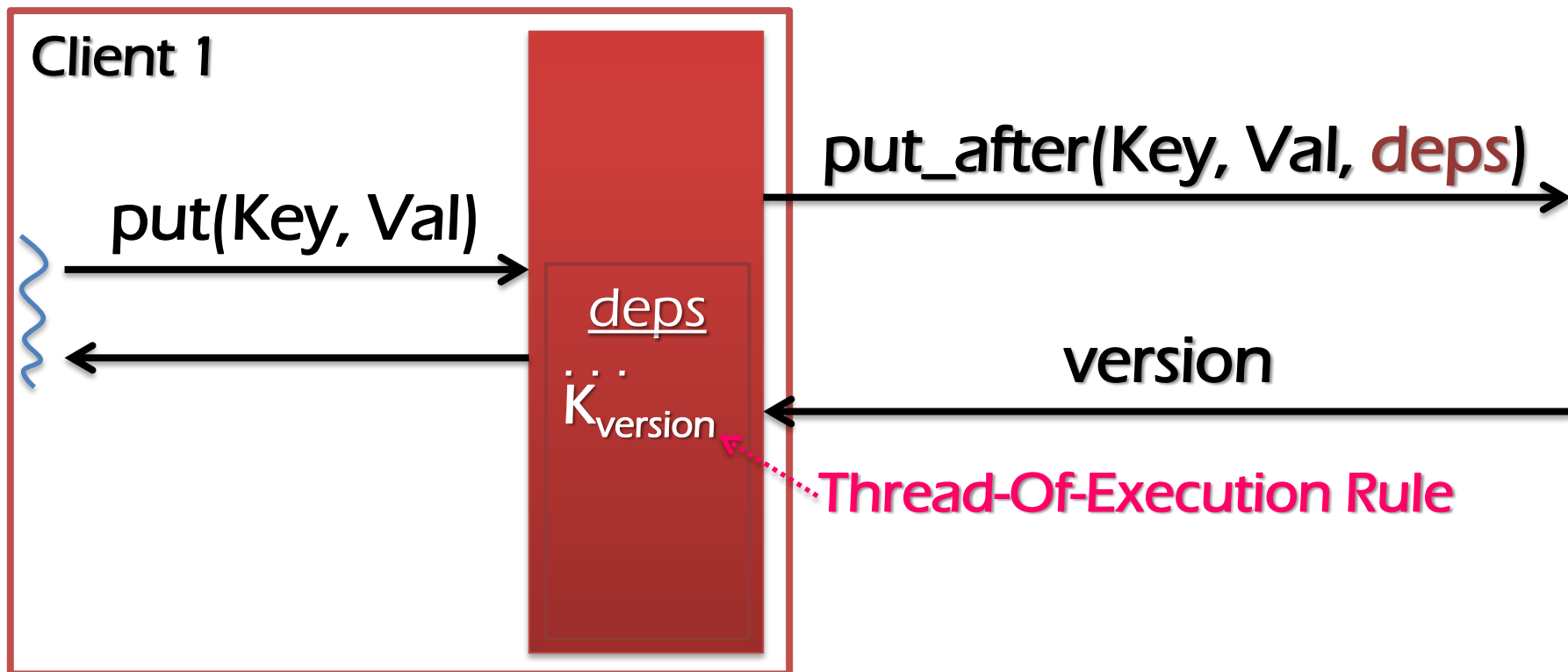
Deps are explicit metadata on values

Library track and attaches deps to **put_afters**

Dependency

Deps are explicit metadata on values

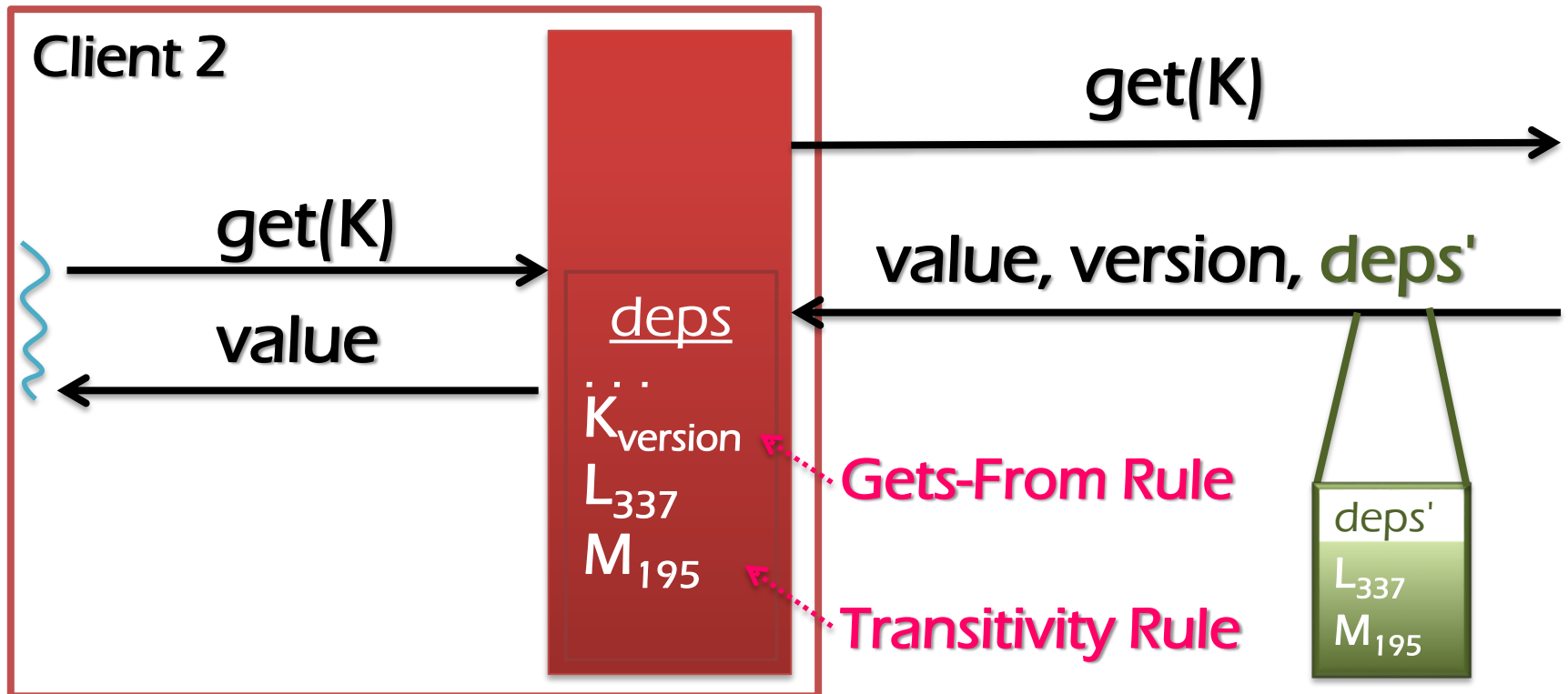
Library track and attaches deps to **put_afters**



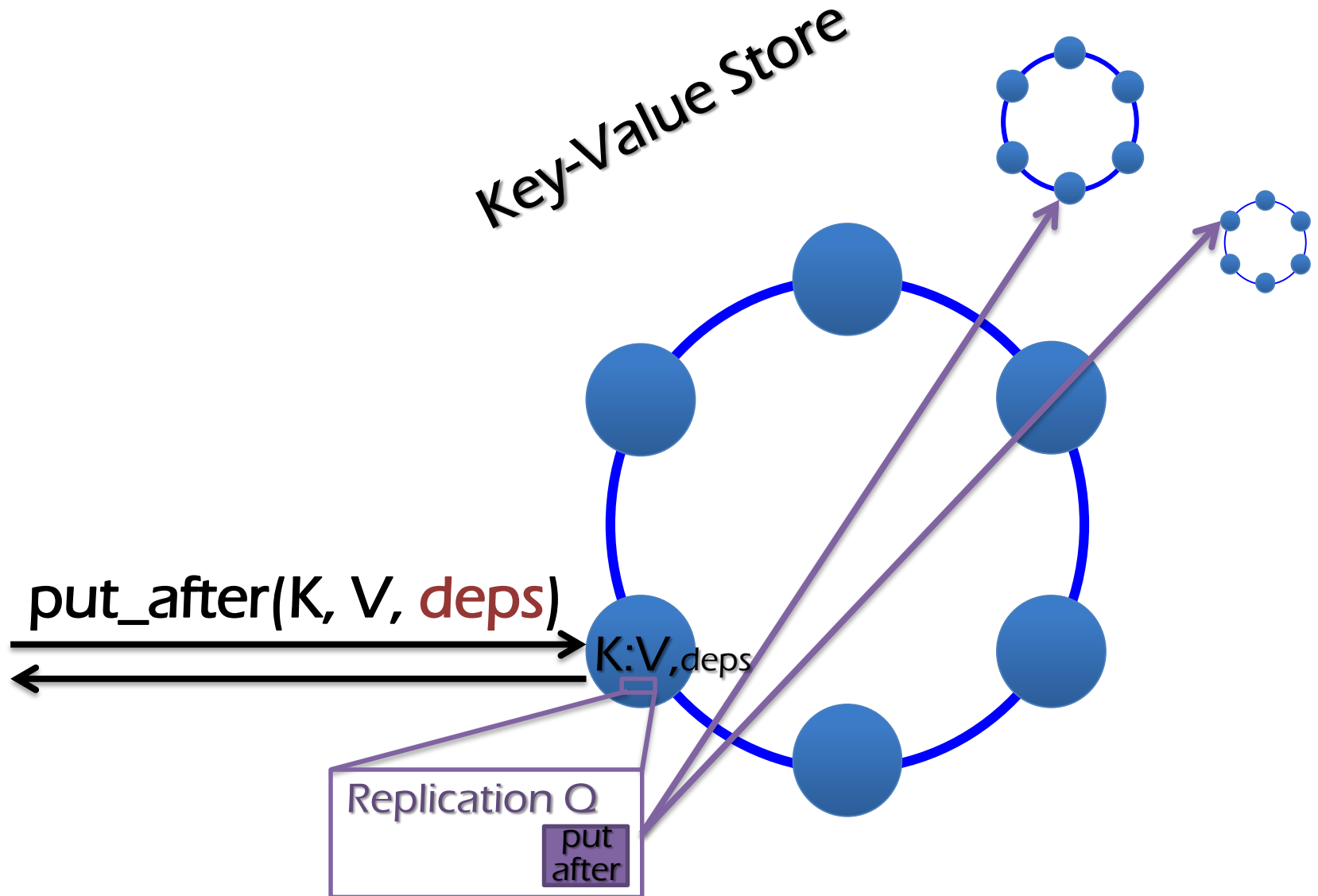
Dependency

Deps are explicit metadata on values

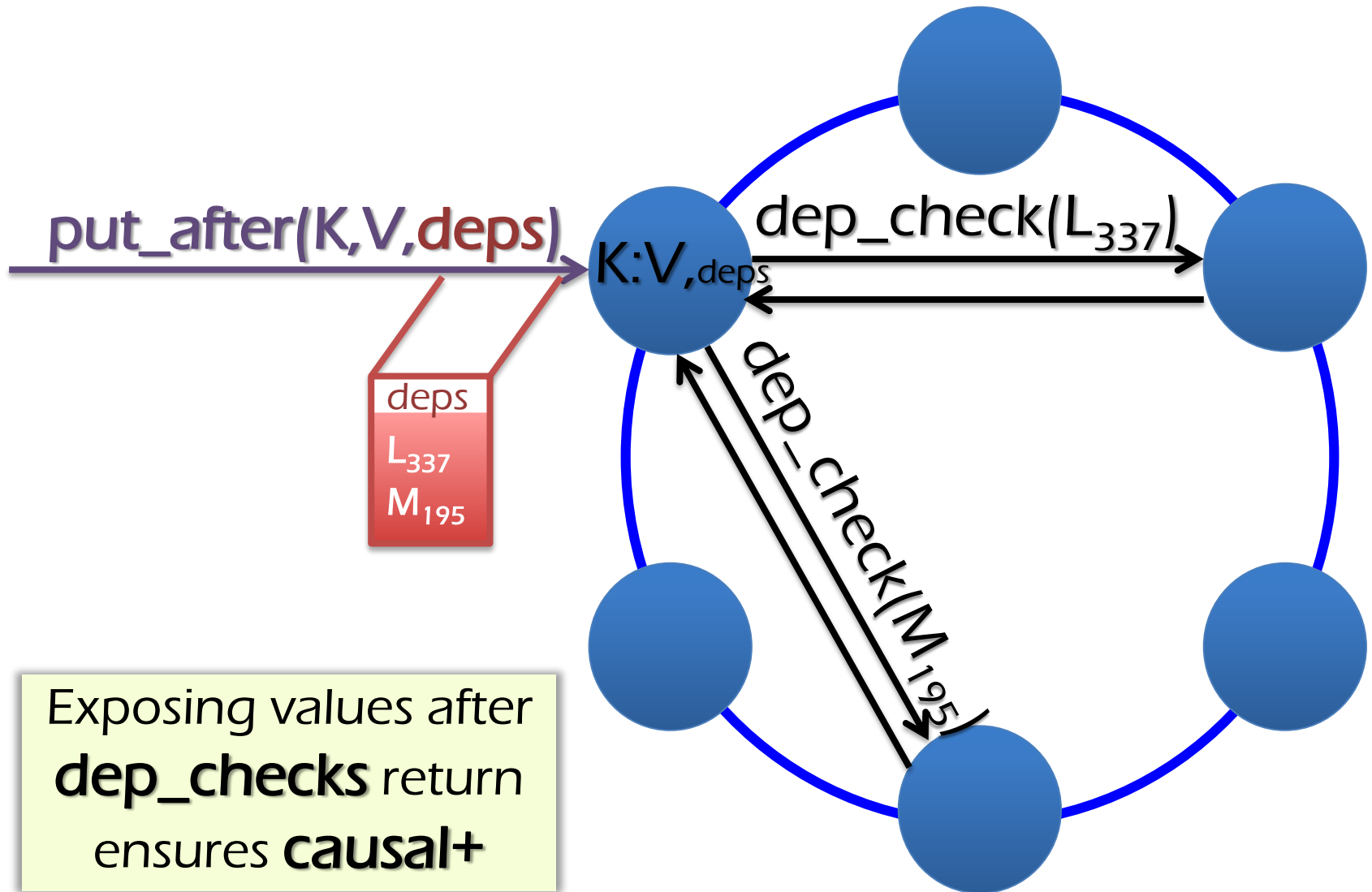
Library track and attaches deps to **put_afters**



Causal+ Replication



Causal+ Replication



Summary

COPS is a **scalable** distributed storage system that provides **causal+** consistency without sacrificing **ALPS** properties

COPS achieves causal+ consistency by tracking and **explicitly** checking that **causal dependencies** are satisfied before exposing writes in each cluster

Next Lecture

Topic: Crash Recovering & Logging

Paper

Robert Hagmann,

***“Reimplementing the Cedar File System
Using Logging and Group Commit”***. ACM

Symposium on Operating Systems Principles
(SOSP), **1987**

THANKS