# Eventual Consistency

Institute of Parallel and Distributed Systems
Rong Chen

*Some slides adapted from Jinyang Li and Wyatt Lloyd

# Recap: Consistency

- Consistency so far
  - Concurrency forces us to think about meaning of R/W
  - Sequential: everyone sees same R/W order (IVY)
  - Release: everyone sees W in unlock order (TreadMarks)

# Recap: Consistency

- Sequential consistency properties:
  - All read/write ops follow *some* total ordering
  - Read must see result of latest write
- Release consistency properties:
  - Delay changes visible to other processors until certain synchronization accesses occurs
  - Eager vs. Lazy

# Disadvantages of sequential/release consistency

- Very Slow
  - Must ask before each operation
  - IVY: ask manager
    - read faults and write faults
  - TreadMarks: ask lock manager
    - acquire and release

# Disadvantages of sequential/release consistency

- Requires highly available connections
  - Lots of chatter between clients/servers
- Not suitable for certain scenarios:
  - Disconnected clients (e.g. your laptop)
  - Apps might prefer potential inconsistency to loss of availability

# 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 and conflicting writes…

# Sequential vs. Eventual Consistency

- Sequential: pessimistic conflict handling
  - Updates cannot take effect unless they are serialized first
- Eventual: optimistic conflict handling
  - 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 (possibly) stale value than blocking for the latest value
- Anomalies:
  - Write-write conflict
  - Stale read
    - Loss of causality

# Eventual Consistency

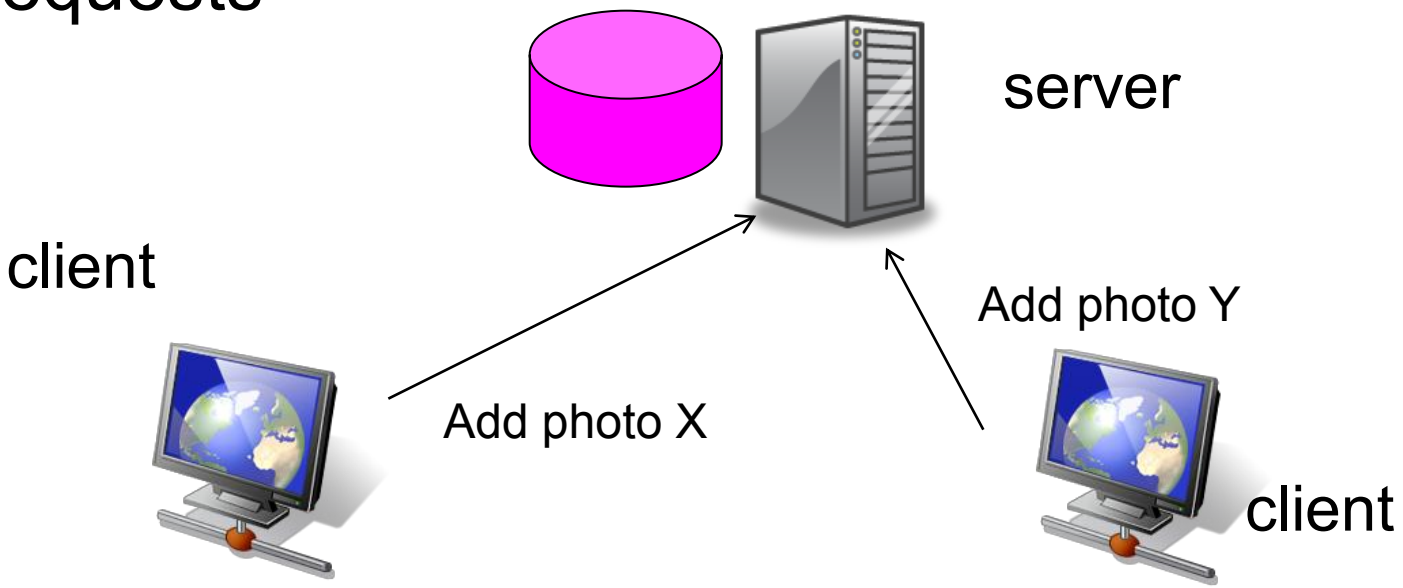
- Desired properties
  - Eventual convergence of state
  - Causality preserving

# Example: Photo Gallery

- Photo Gallery
  - Consists of many albums, each identified by an “aid”
  - Each album (aid) contains an ordered list of photos each identified by a “pid”
- Operations
  - Add/delete album, add photo to album …

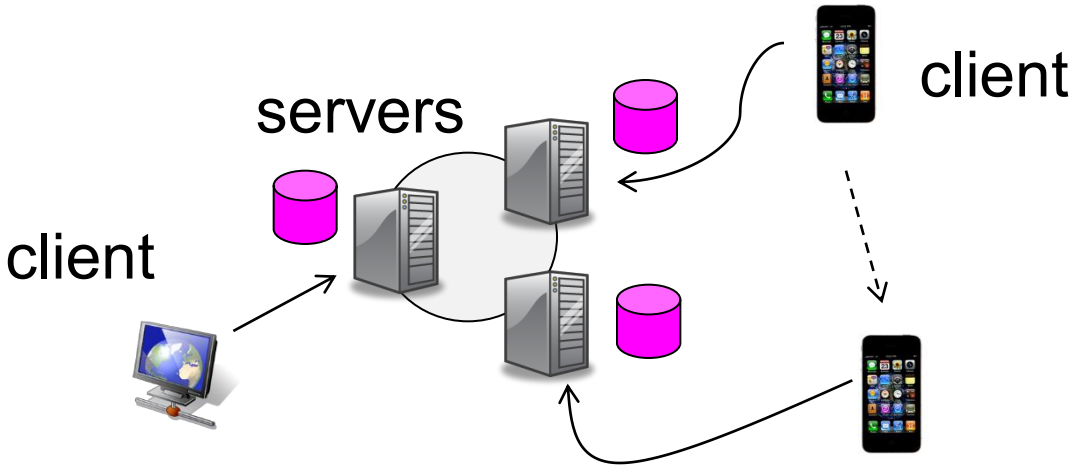
# Example: Photo Gallery

- Traditional Approach: one server
  - Server processes requests one at a time
  - Server implicitly chooses order for concurrent requests



# Example: Photo Gallery

- Why aren't we satisfied with central server?
  - I want to operate on my 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.



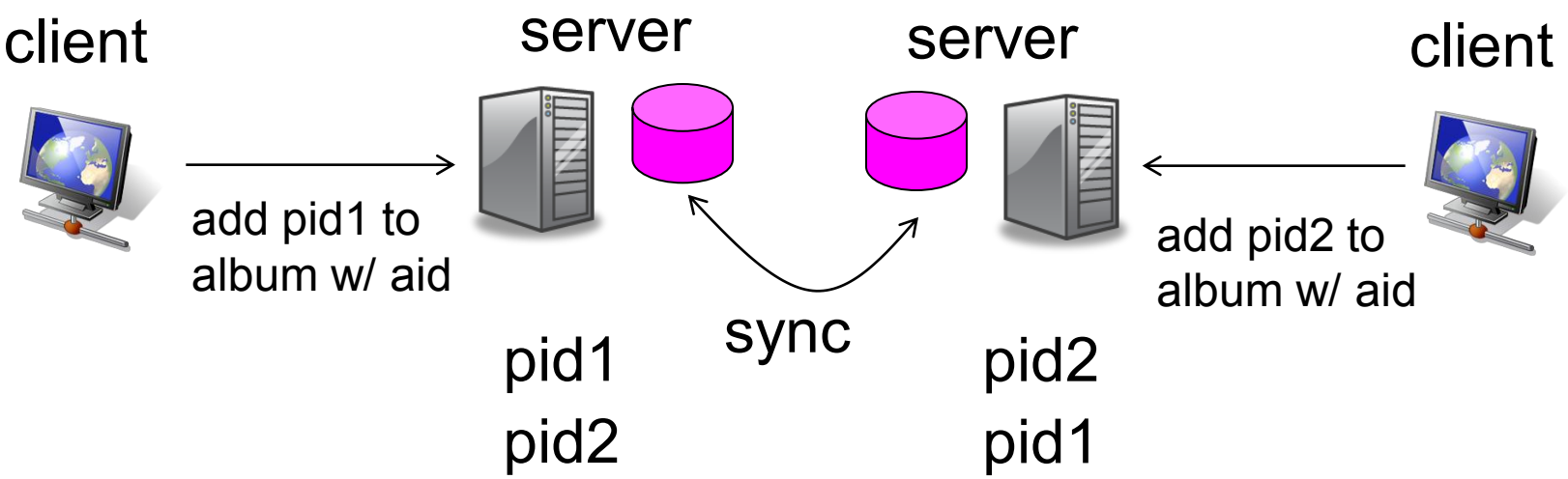
# Straw man: Merge Storage

- What if there's a write-write conflict?
  - Add two photos to the same album "at the same time"
  - But we want automatic conflict resolution

# Solution: Update Functions

- Have update be a function, not a new value
  - e.g. add “pid” to album w/ “aid”
- Read current state of storage, decide best change
- Function must be deterministic
  - Otherwise nodes will get different answers

# Challenge: Ordering




Will X show pid1 in front of pid2, and Y show pid2 in front of pid1?

Goal: eventual state convergence

# 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 same updates in log

# Solution: Ordered Update Log

- How can nodes agree on update *order*?
  - Update ID: <time T, node ID>
  - Assigned by node that creates the update

  - Ordering updates a and b:

    a < b if a.T < b.T or (a.T = b.T and a.ID < b.ID)

# Solution: Ordered Update Log

- Example

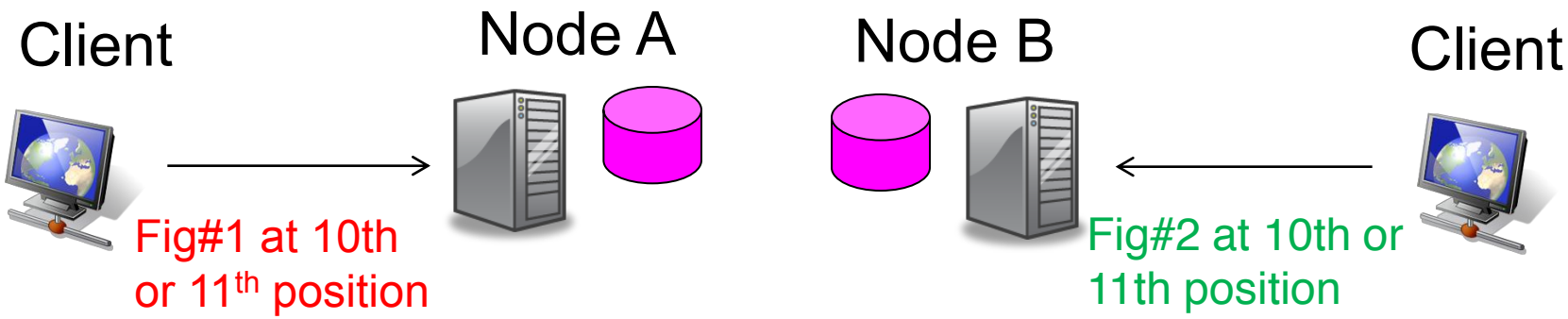
  <10,A>: Fig#1 at 10th or 11th position for album aid

  <20,B>: Fig#2 at 10th or 11th position for album aid

  Q: What's the correct final outcome?

  A: Fig#1 at 10th position, Fig#2 at 11th position

# Example

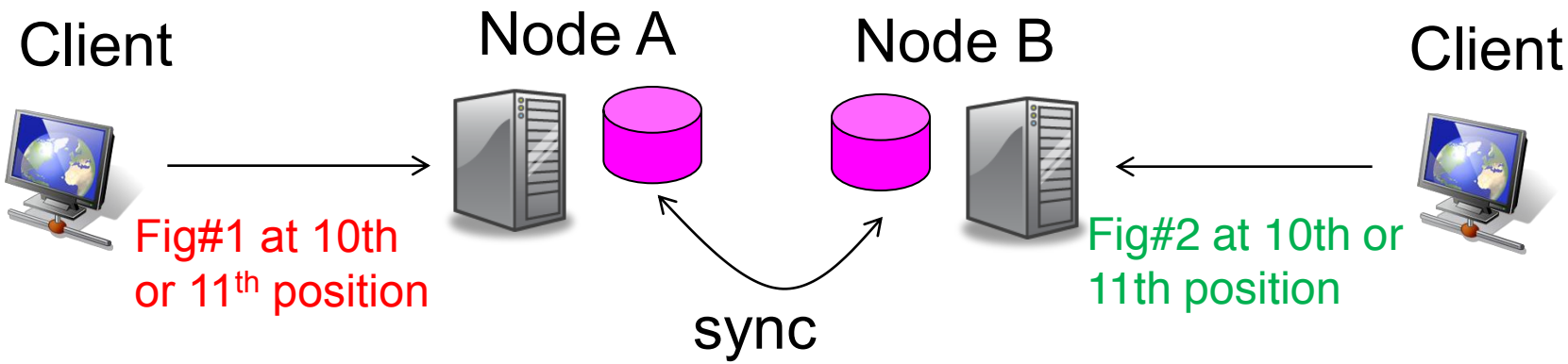



Fig#1 at 10th position for album aid

Fig#2 at 10th position for album aid

This is what client will see before syncing

# Example




Fig#1 at 10th position
for album aid

Fig#2 at 10th position
for album aid

Inconsistent

Fig#2 at 11th position
for album aid

Fig#1 at 11th position
for album aid

If just run the update function against storage state

# Solution: Rollback and Replay

- Roll back and replay
  - Re-run all update functions, starting from empty storage state
    - Node A and B have same set of updates
    - Arrive at same final state
  - We will optimize this in a bit

# Challenge: Clock Time

- Will update order be consistent with *wall-clock* time?
  - B's figure gets priority, even A posted first
    - A went first (in wall-clock time) with <10,A>
    - But B could then generates <9,B>

    *(Node clocks are not perfectly synchronized)*

# Example

Suppose

A adds a photo pid1

*then B sees it*

then B deletes pid1

Perhaps

<10,A> add fig#1

<9,B> delete fig#1

*(B's clock is slow)*

– Now delete will be ordered before add!

# Solution: *Lamport* Logical Clocks

- Want to Timestamp events
  - Node *observes* E1, then *generates* E2
    => TS(E1) < TS(E2)
  - E1 then E2 on same node
    => TS(E1) < TS(E2)
  - Other nodes will order E1 and E2 the same way

# Solution: *Lamport* Logical Clocks

- Logical clocks
  - Each node keeps a clock T
  - Increments T as real time passes, one second per second
  - T = max(T, T'+1) if sees T' from another node

A adds a photo pid1

*then B sees it*

then B deletes pid1

<10,A> add fig#1
B.T = 11 (or more)
<11,B> delete fig#1

# Challenge: Tentative

- Displayed photo positions are "*tentative*"
  - Never know if there's some other photo from nodes you haven't yet *synced*
  - Long-delayed update with *lower TS*
    - Cause the results of my update to change

# Challenge: Tentative

- Would be nice if updates were eventually "*stable*“
  - no changes in update order up to that point
  - results can never again change
    - e.g. you know for sure fig#1 is at position 10
  - no need to re-run update function

# Solution: Decentralized

- Commit scheme
  - *Sync* always send in log order
  - If you have seen updates w/ TS > 10 from every node
    - never again see one lower <10,A>
    - <10,A> is *stable*

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

# Solution: Primary

- Commit scheme (*Bayou'95*)
  - One node designated "primary"
    - Assign a total commit order: CSN (commit-seq-no)
    - Complete time stamp: <CSN, local-TS, node-id>
    - Any write with a known CSN is stable
  - All stable writes are ordered before tentative writes
  - CSN notifications are exchanged between nodes
    - The CSNs define a total order for committed updates

# Solution: Primary

- Will commit order match tentative order?
  - Often “Yes”
  - Syncs send in log order
    - Including updates learned from other nodes
  - Example

    So if A's update log says

    <-,10,X>

    <-,20,A>

    A will send both to primary, in that order

    Primary will assign CSNs in that order

    Commit order will match tentative order

# Solution: Primary

- Will commit order *always* match tentative order?
  - "No": primary may see newer updates before older ones
  - Example

    A has just: <-,10,A> M1

    B has just: <-,20,B> M2

    If C sees both, Tentative order: M1 M2

    B syncs w/ primary, gets CSN=5: <5, 20, B> M2

    Later A syncs w/ primary, gets CSN=6: <6, 10, A> M1

    When C syncs w/ primary, Commit order: M2 M1

# Summary

- Replicas, write any copy, and sync are good ideas
  - Now 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

# Summary

- Bayou introduced some very influential design ideas
  - Update functions
  - Ordered update log is the real truth, not the DB
  - Allowed general purpose conflict resolution
- Bayou made good use of existing ideas
  - Eventual consistency
  - Logical clock

# COPS: Scalable Causal Consistency for Wide-Area Storage

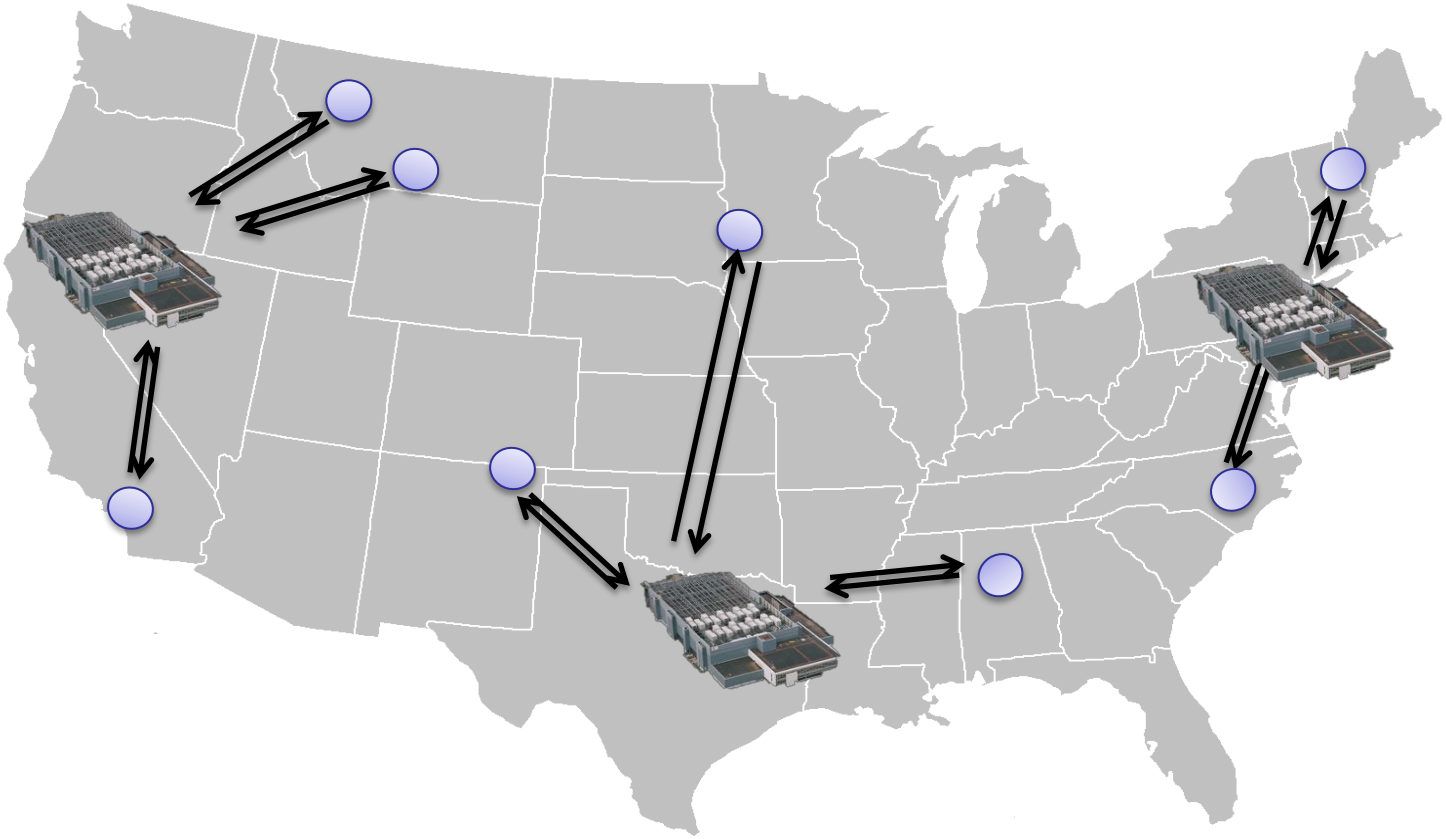

**Stores:**
Status Updates
Likes
Comments
Photos
Friends List

**Stores:**
Posts
+1s
Comments
Photos
Circles

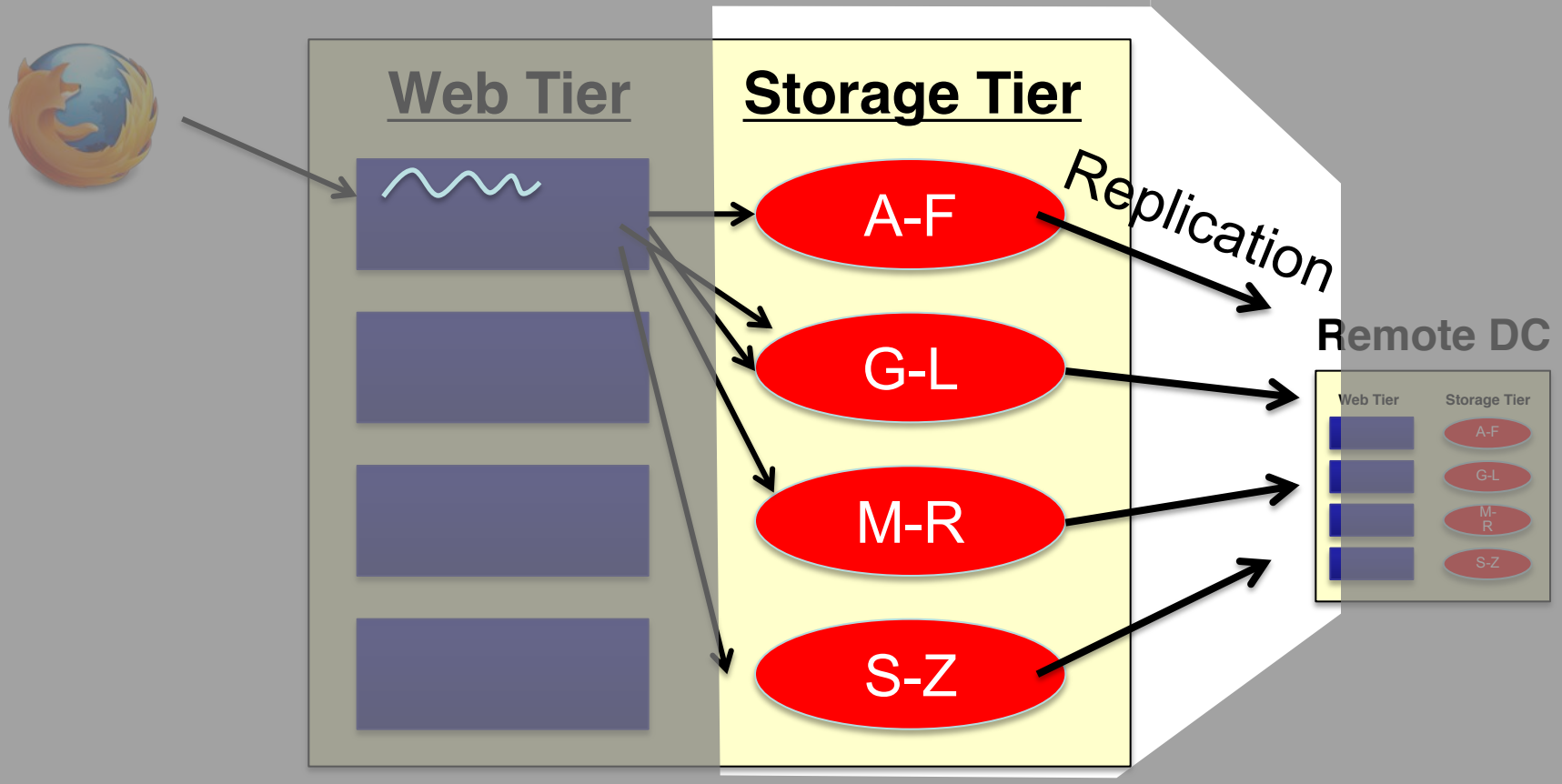
**Stores:**
Tweets, Favorites,
Following List

# Wide-Area Storage

- Multiple DCs, 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 | ✖ | ✔ |
| • Causal+ Consistency | ✔ | ✔ |

ALP = “always on”

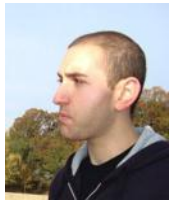
# Basic COPS Summary

- Serve operations locally, replicate in background (*async*)
  - "Always On"
- Control replication with dependencies
  - Causal+ Consistency
- Partition keyspace onto many nodes
  - Scalability

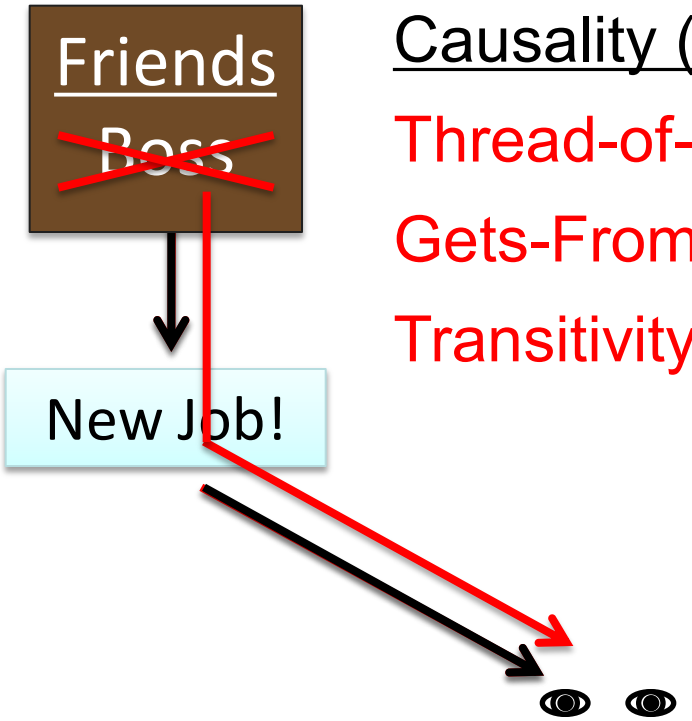
# Causality By Example

Remove boss from friends group

Post to friends: "Time for a new job!"



Friend reads post




Causality (⟶)

Thread-of-Execution

Gets-From

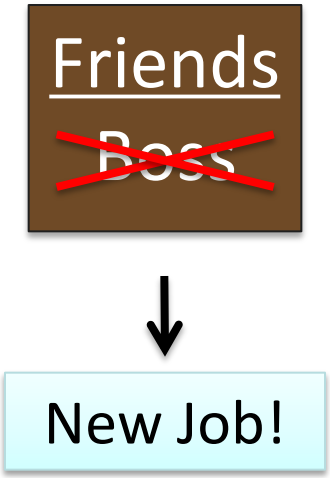
Transitivity

# Causal Consistency

- Partial orders
  1. if op1 and op2 are in the single thread of execution and op1 is issued before op2, then op1 => op2 (Thread-of-Execution)
  2. If op2 reads the result written by op1, then op1 => op2 (Gets-From)
  3. if op1 => op2, op2 => op3, then op1 => op3 (Transitivity)

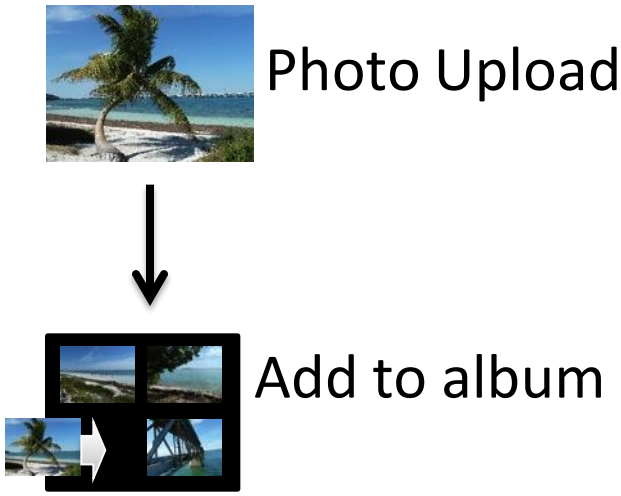
# Causality Is Useful

For Users:



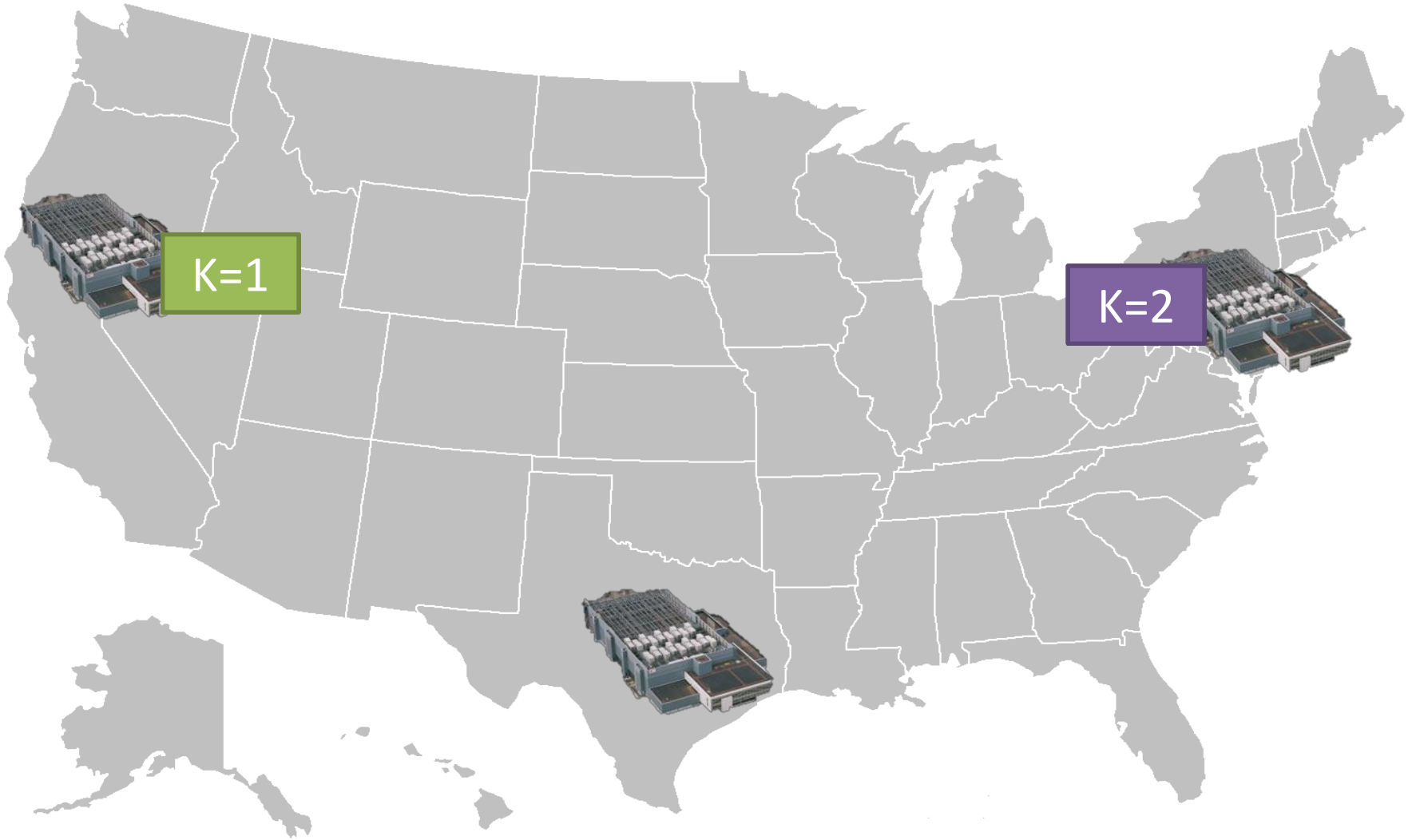

Employment Integrity

For Programmers:

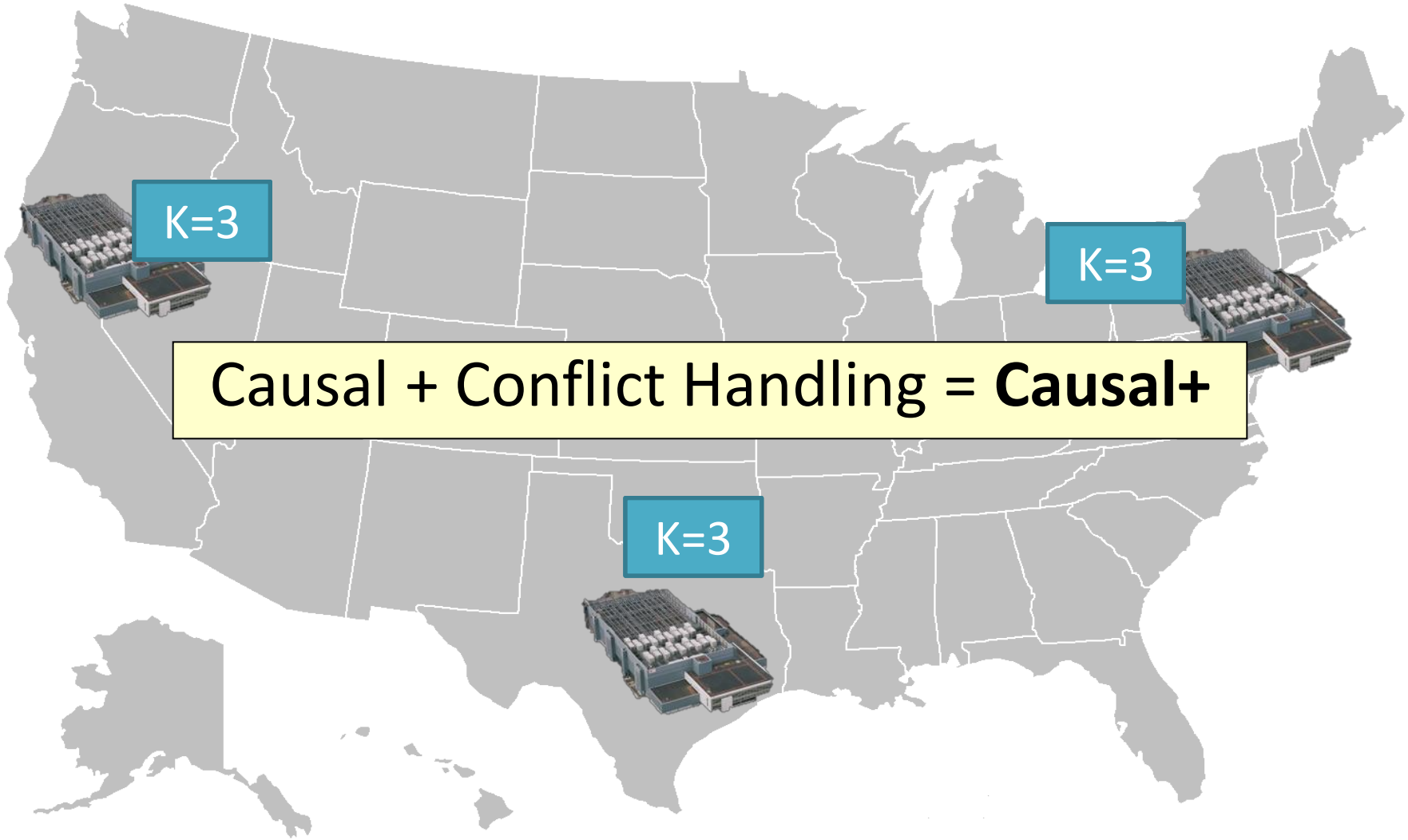



Referential Integrity

# Conflict in Causal



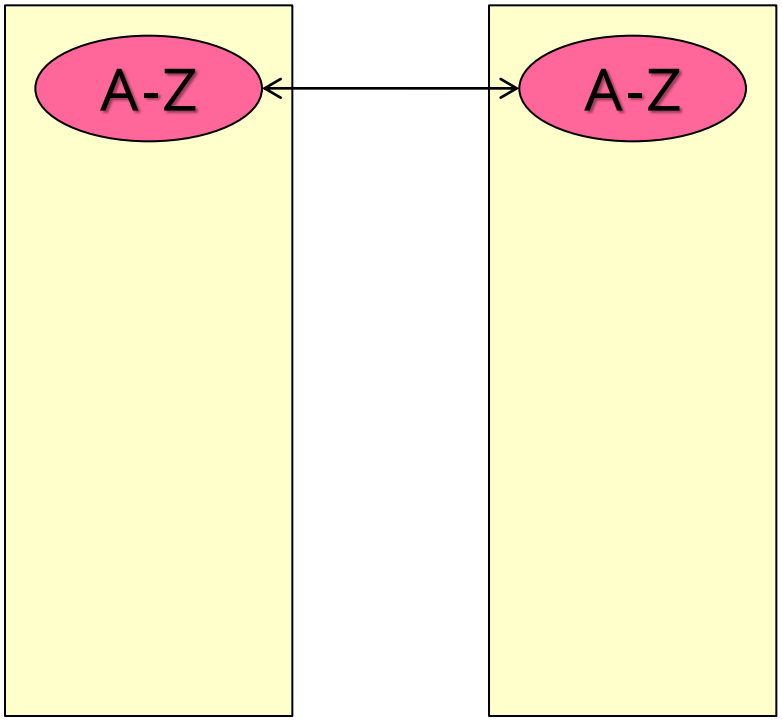
# Conflict in Causal




Causal + Conflict Handling = **Causal+**

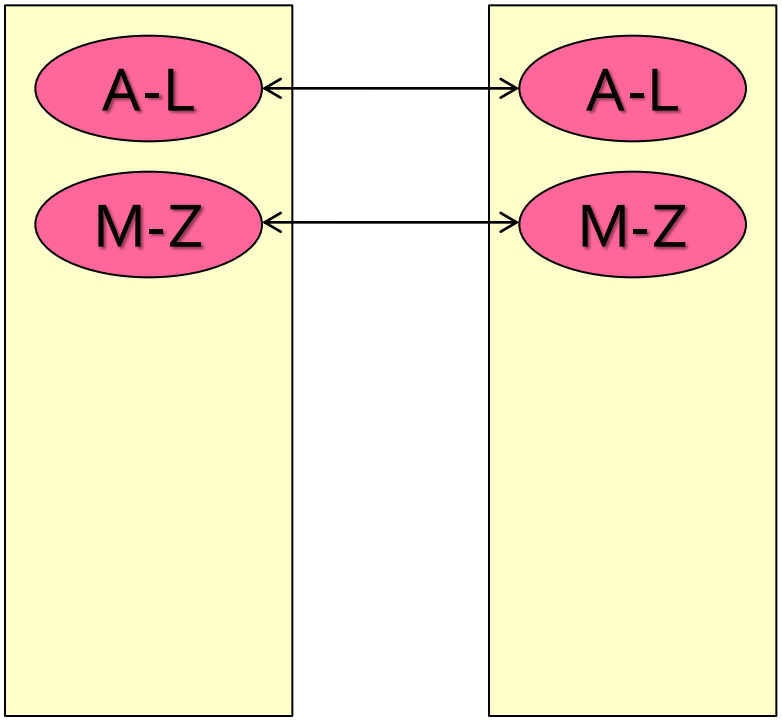
# Scalability

- Increase capacity and throughput in each datacenter



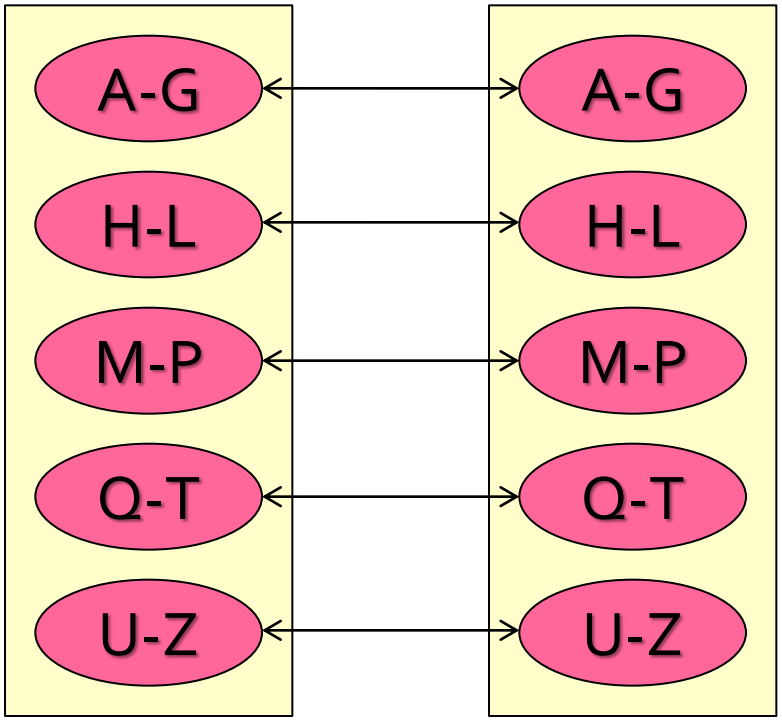
# Scalability

- Increase capacity and throughput in each datacenter



# Scalability

- Increase capacity and throughput in each datacenter



# Previous Causal+ Systems

- Bayou 94', TACT 00', PRACTI 06'
  - Log-exchange based

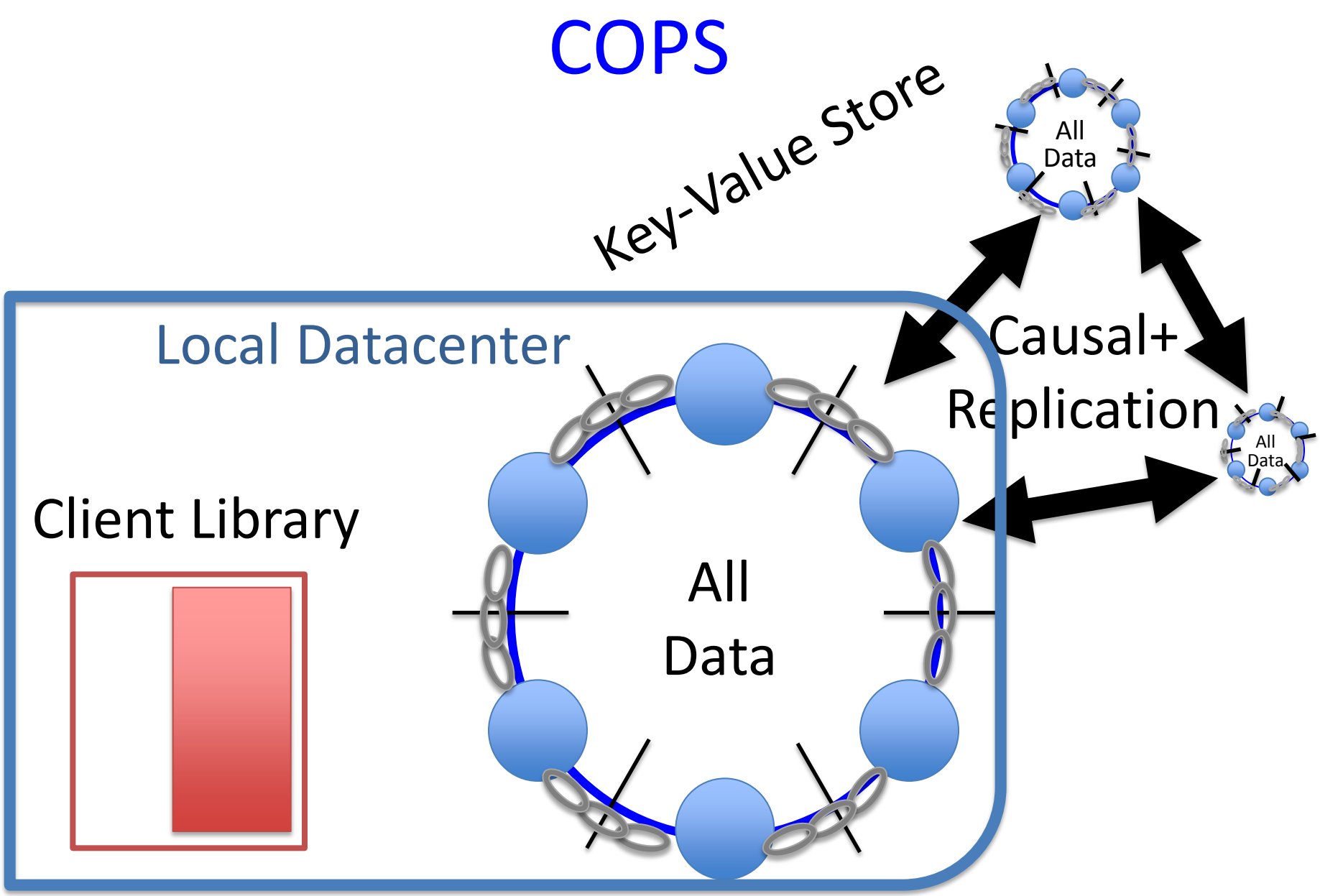
- Log is single serialization point
  - **Implicitly** captures and enforces causal order
  - Limits scalability
    - One node store all keys
    - Provide log for all ops across a cluster
  - No cross-server causality

# Scalability Key Idea

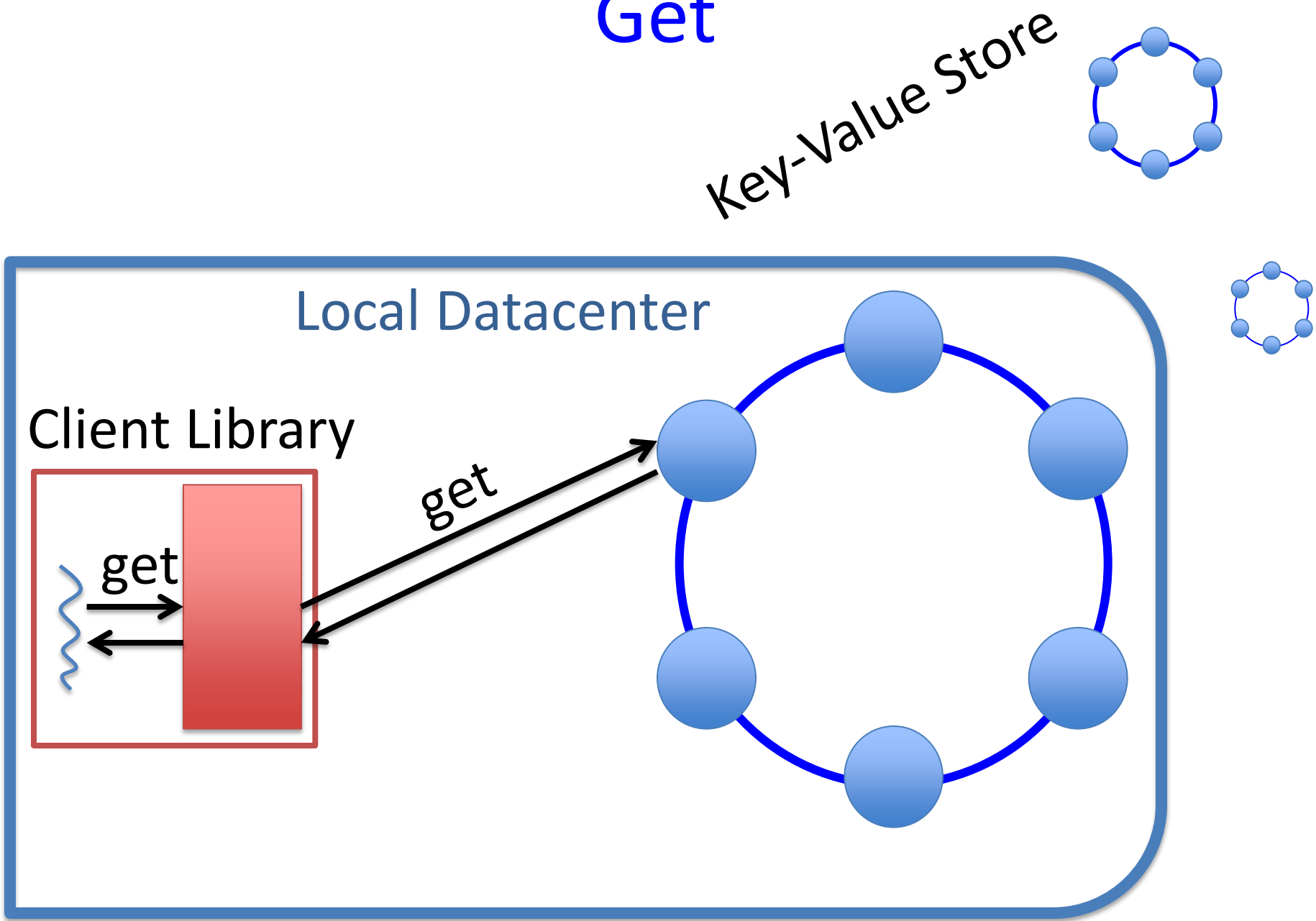
- Dependency metadata explicitly captures *causality*
- Distributed verifications replace single serialization
  - *Delay* exposing replicated puts until all dependencies are satisfied in the datacenter

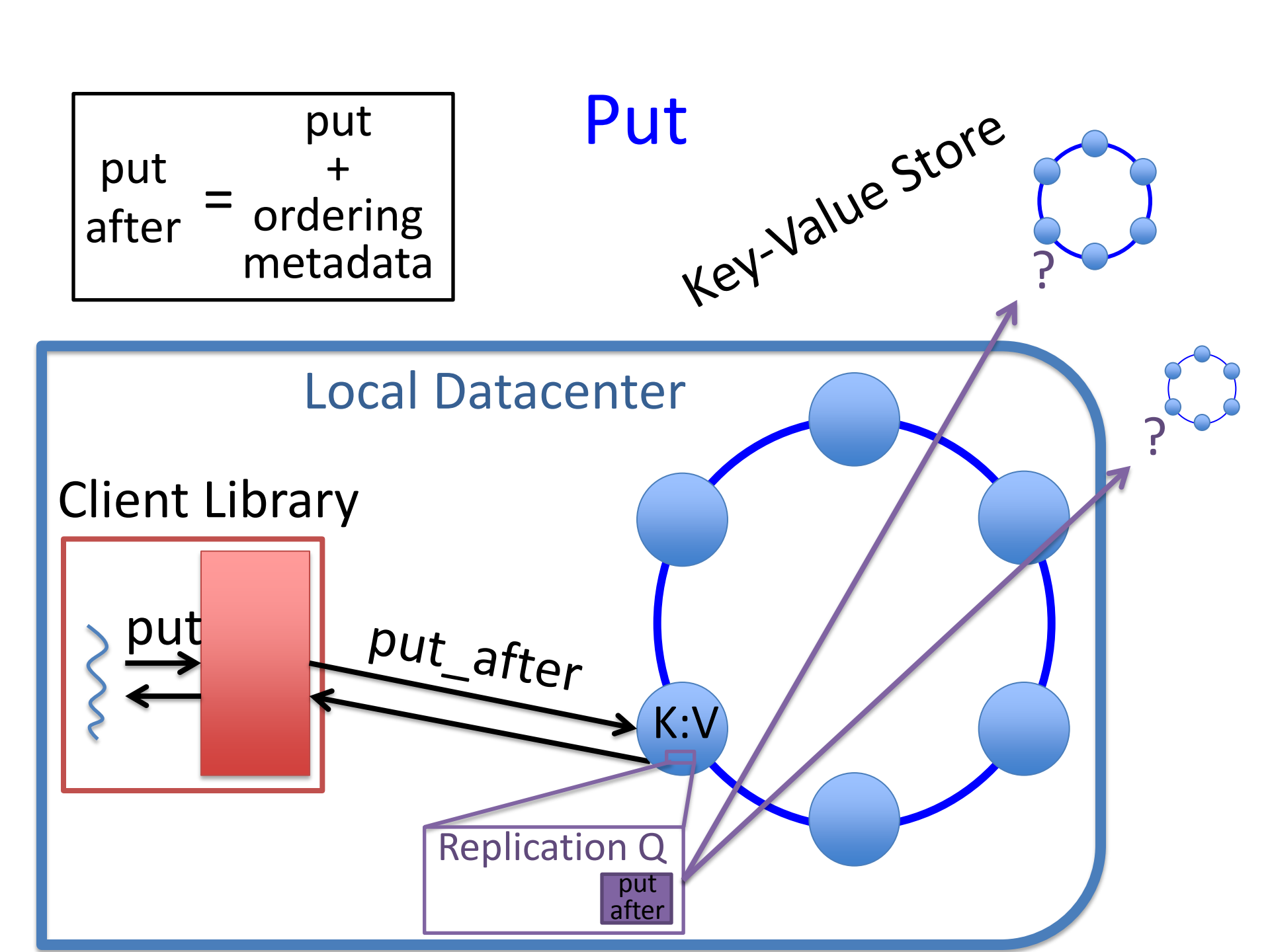
# Explicit Dep. Propagate

- Explicitly keep track explicit dependencies (*partial orders*) 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.


COPS
Key-Value Store
All Data
Causal+ Replication
All Data
Local Datacenter
Client Library
All Data

# Get



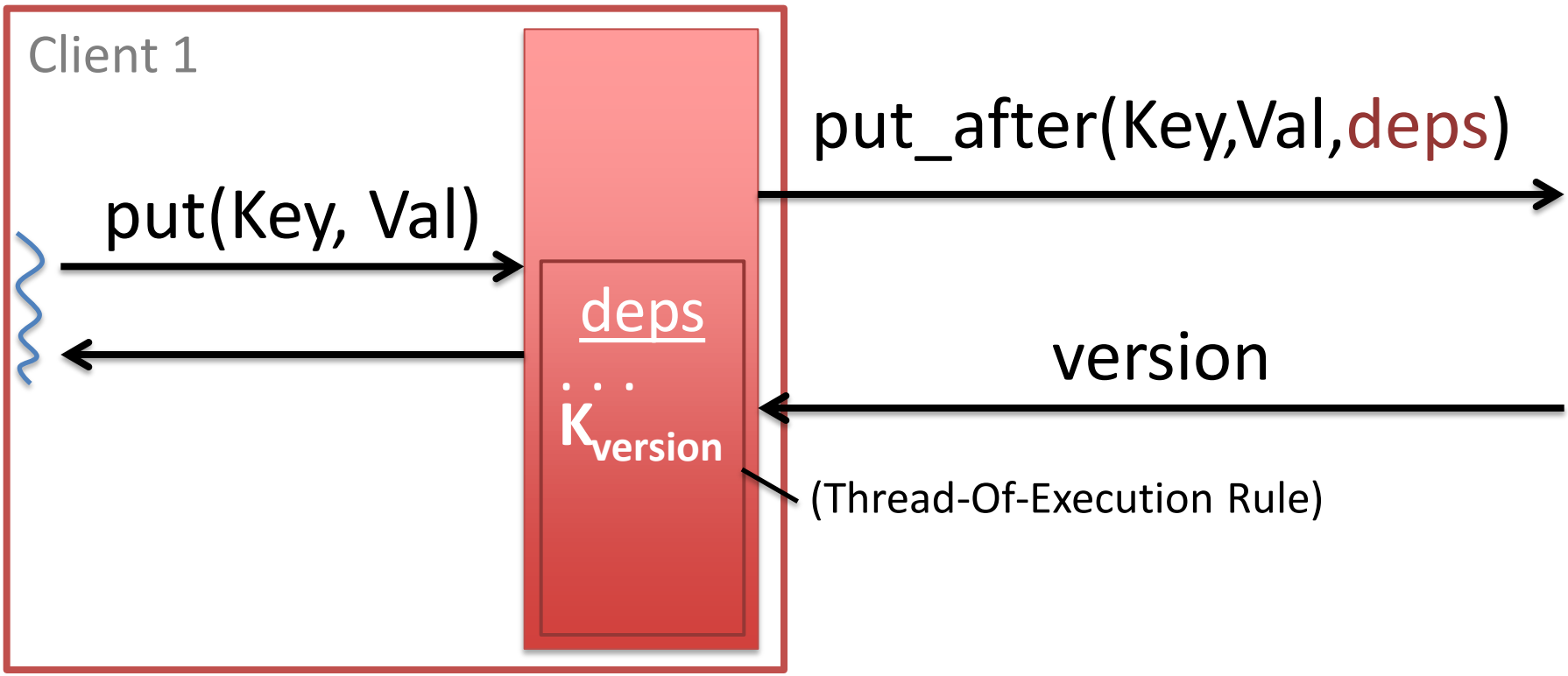

put after = put + ordering metadata
Put
Key-Value Store
?
?
Local Datacenter
Client Library
put
put_after
K:V
Replication Q
put after

# Dependencies

- Dependencies are explicit metadata on values
- Library tracks and attaches them to put_afters

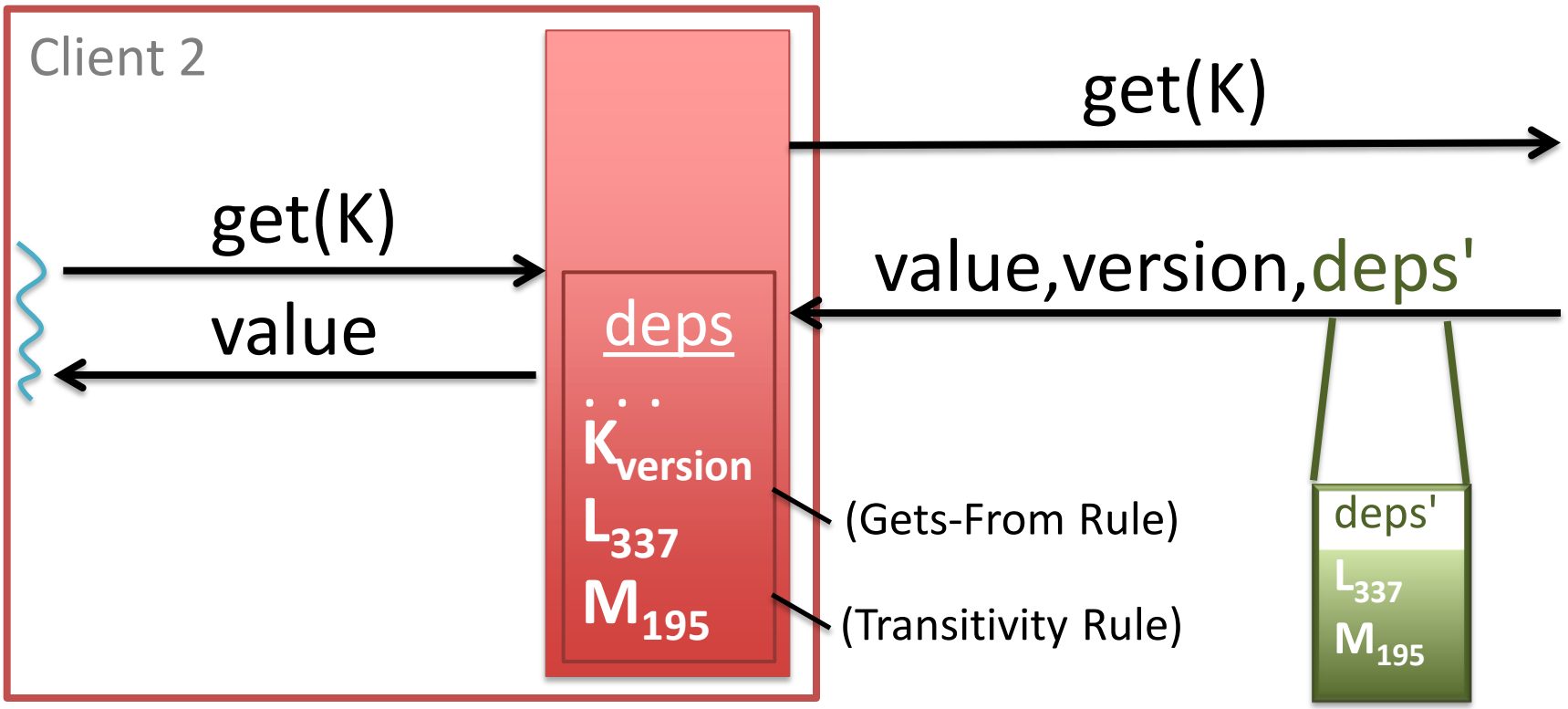
# Dependencies

- Dependencies are explicit metadata on values
- Library tracks and attaches them to put_afters

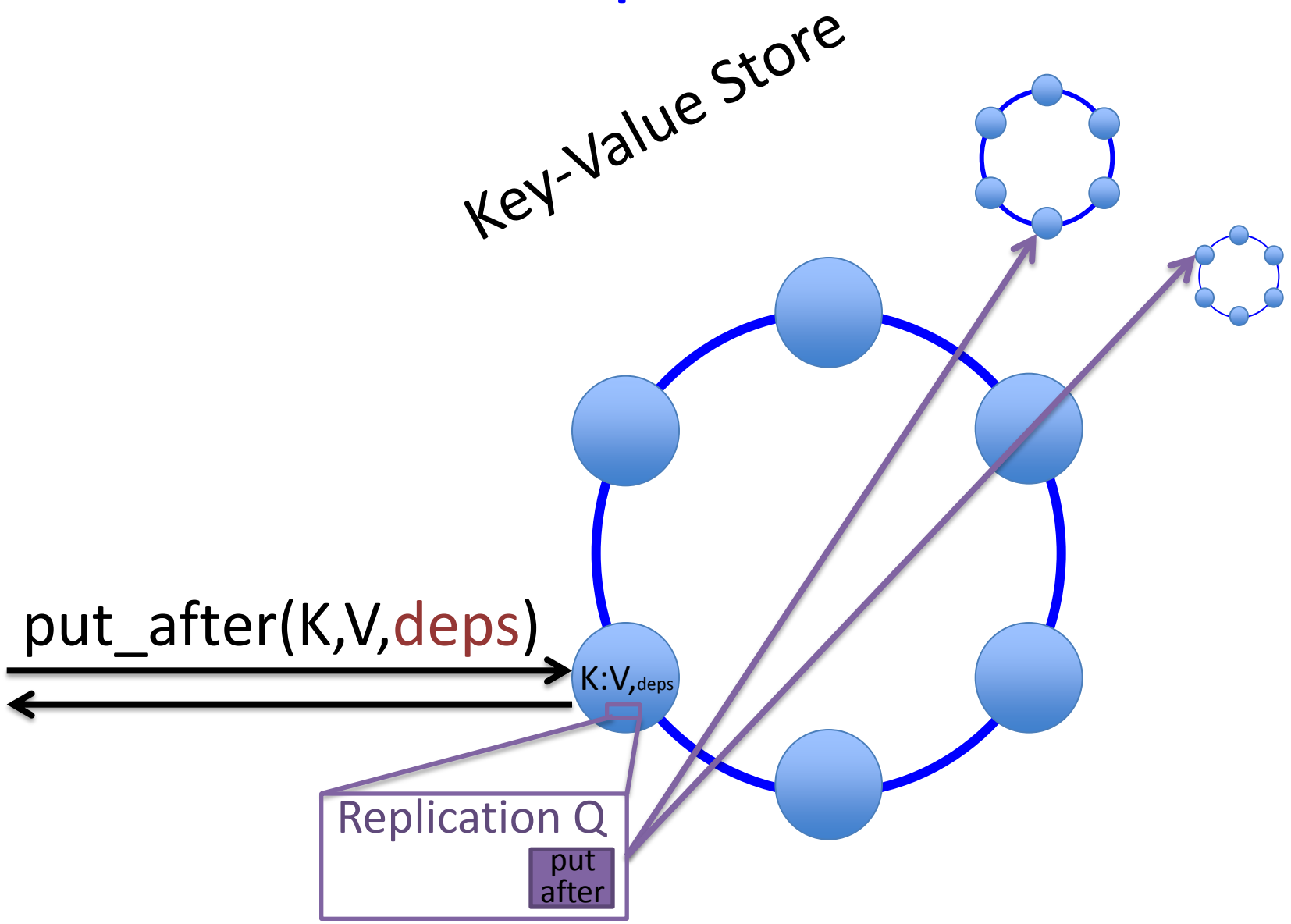


# Dependencies

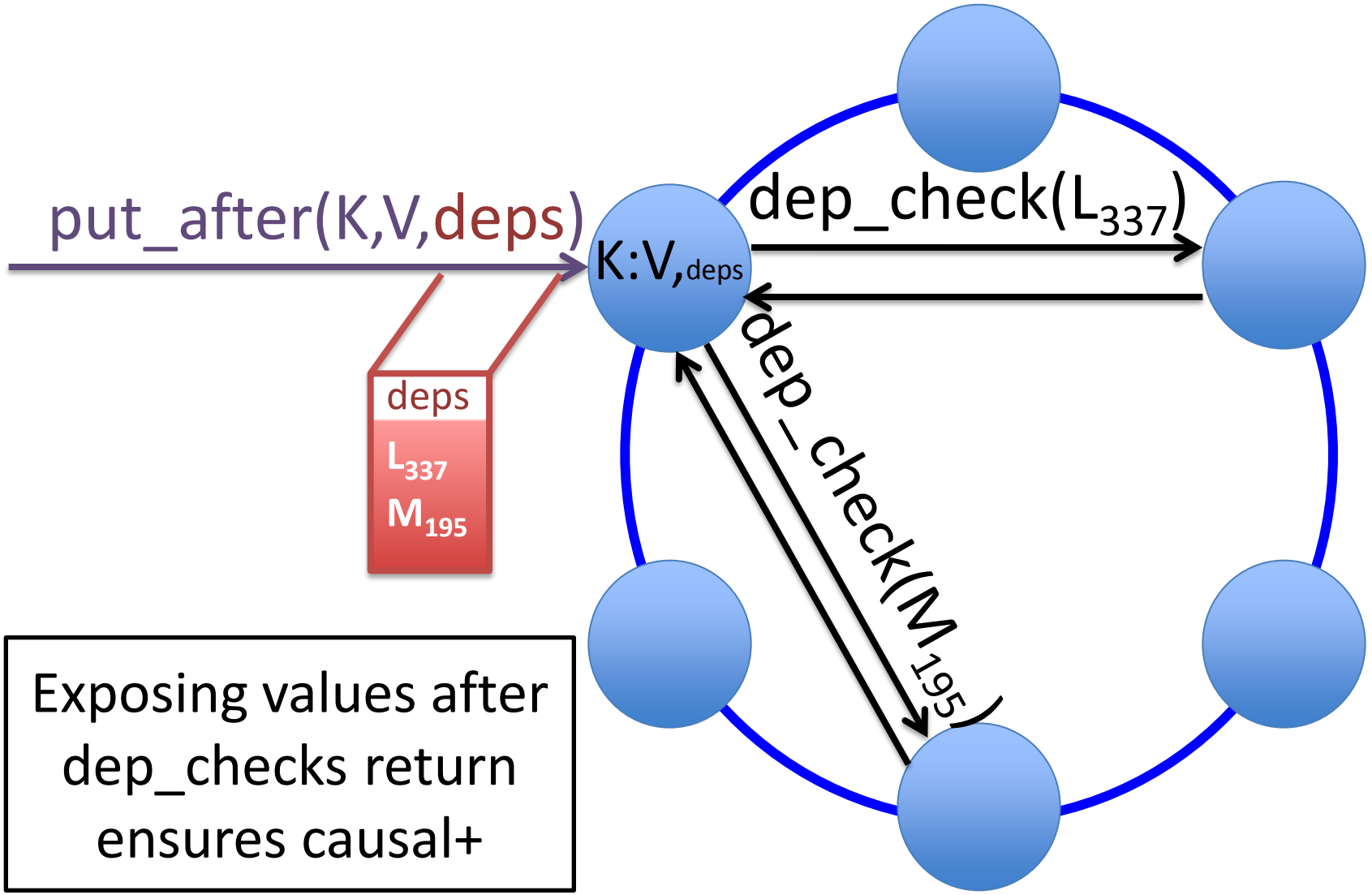
- Dependencies are explicit metadata on values
- Library tracks and attaches them to put_afters



# Causal+ Replication



# Causal+ Replication




Exposing values after dep_checks return ensures causal+

# Next Classes

- Fault Tolerance
  - Topic: Crash recovery & logging
  - Paper: Cedar

# Thank you !

Any questions?