A Brief History Of Time

In Riak

Time in Riak

- Logical Time
- Logical Clocks
- Implementation details

Mind the Gap

How a venerable, established, simple data structure/algorithm was botched multiple times.

Order of Events

- Dynamo And Riak
- Temporal and Logical Time
- Logical Clocks of Riak Past
- Now

Why Riak?

Scale Up

$$$Big Iron
(still fails)

Scale Out

Commodity Servers

CDNs, App servers

DATABASES!!

Fundamental Trade Off

- Lipton/Sandberg ’88
- Attiya/Welch ’94
- Gilbert/Lynch ’02

Low Latency/Availability:
- Increased Revenue
- User Engagement

Strong Consistency:
- Easier for Programmers
- Less user “surprise”

Consistency

There must exist **a total order on all operations** such that each operation looks **as if it were completed at a single instant.** This is equivalent to requiring requests of the distributed shared memory to **act as if they were executing on a single node**, responding to operations one at a time.

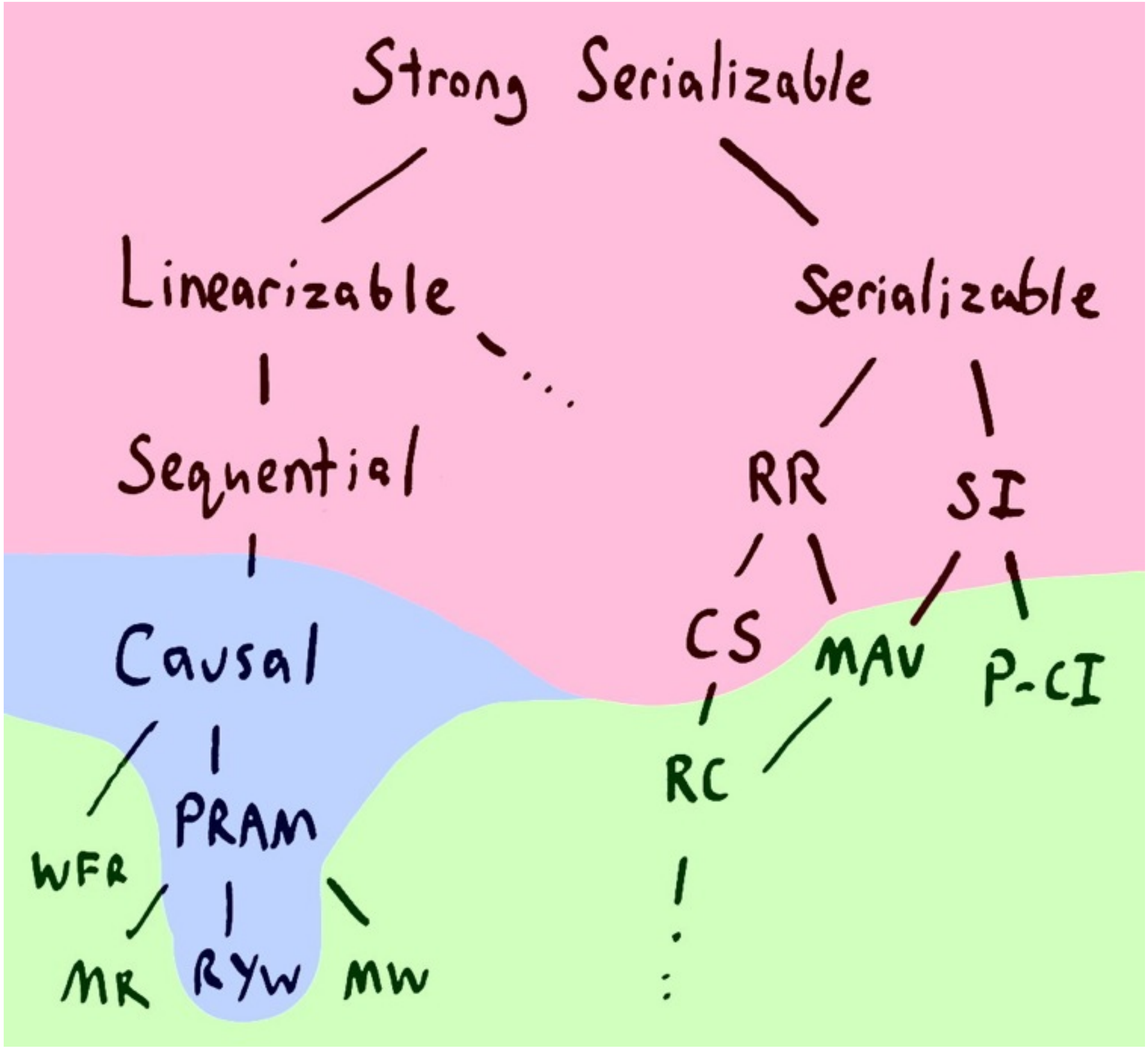
--Gilbert & Lynch

Consistency

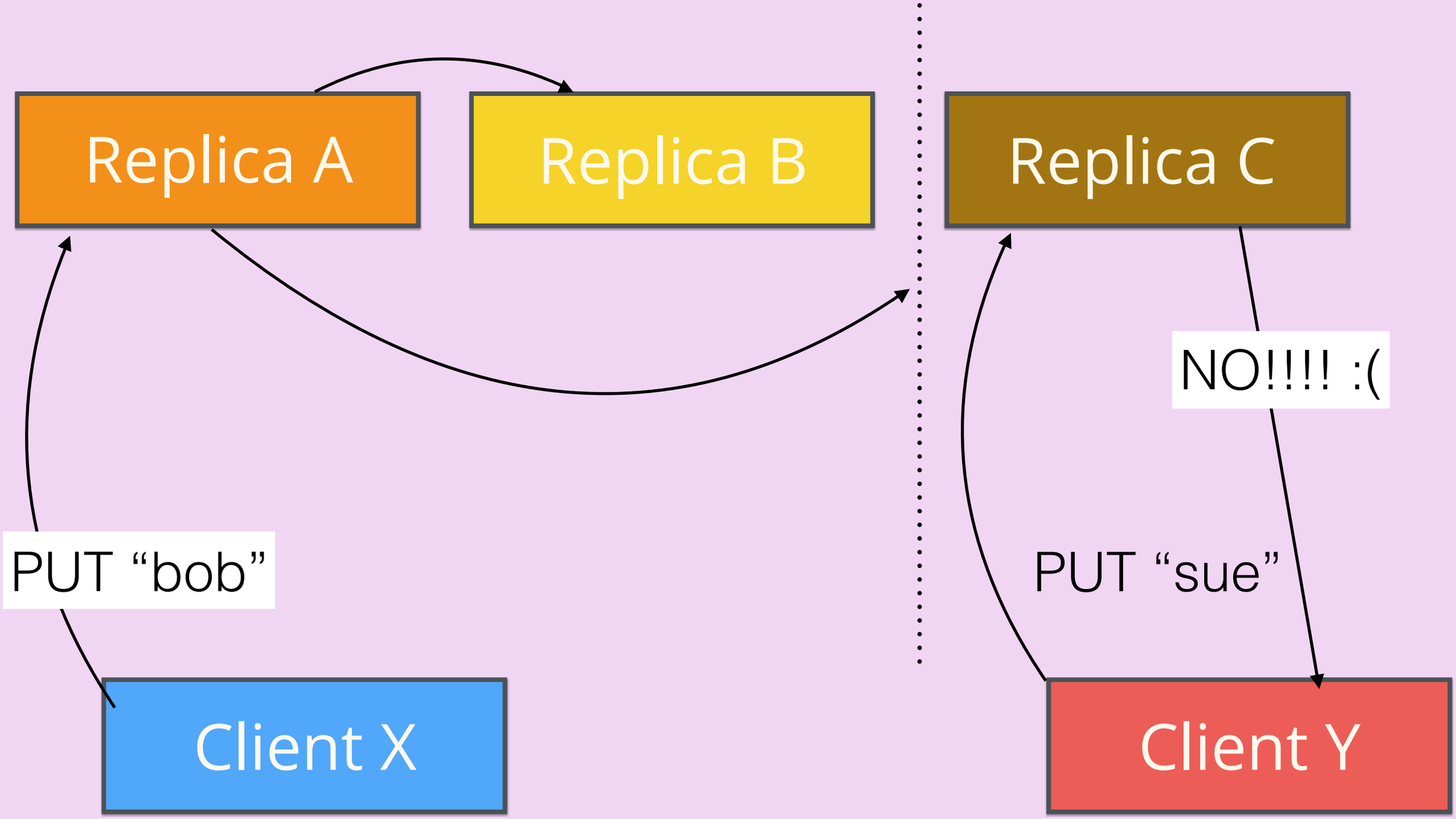
One important property of an atomic read/write shared memory is that

any read operation that begins after a write operation completes must return that value, or the result of a later write operation. This is the consistency guarantee that generally provides the easiest model for users to understand, and is most convenient for those attempting to design a client application that uses the distributed service

--Gilbert & Lynch



https://aphyr.com/posts/313-strong-consistency-models

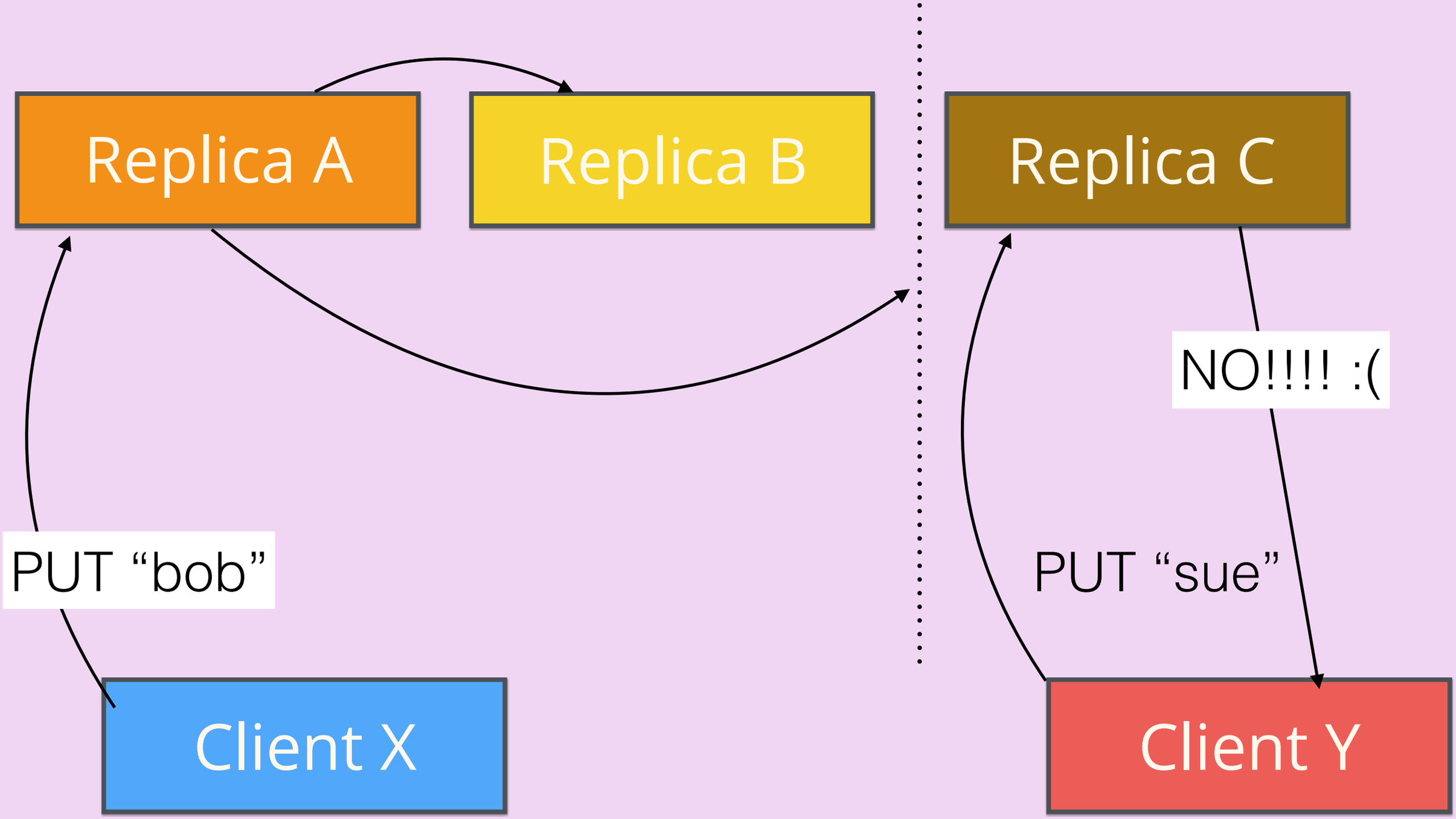

Replica A
Replica B
Replica C
NO!!!! :(
PUT “bob”
PUT “sue”
Client X
Client Y

Consistent

Availability

Any non-failing node can respond to any request

--Gilbert & Lynch

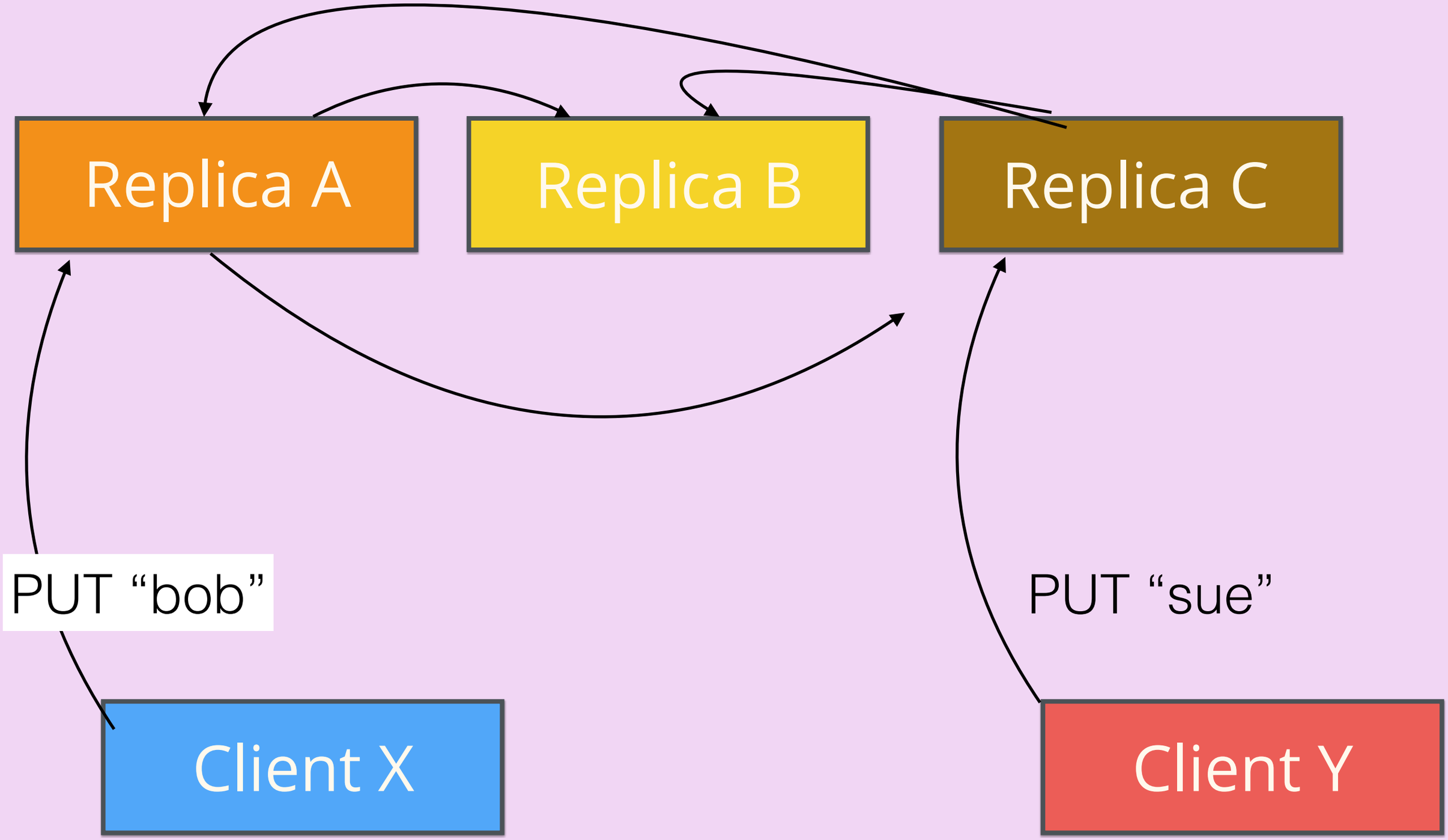

Replica A
Replica B
Replica C
NO!!!! :(
PUT “bob”
PUT “sue”
Client X
Client Y

Consistent

Consensus for a total order of events

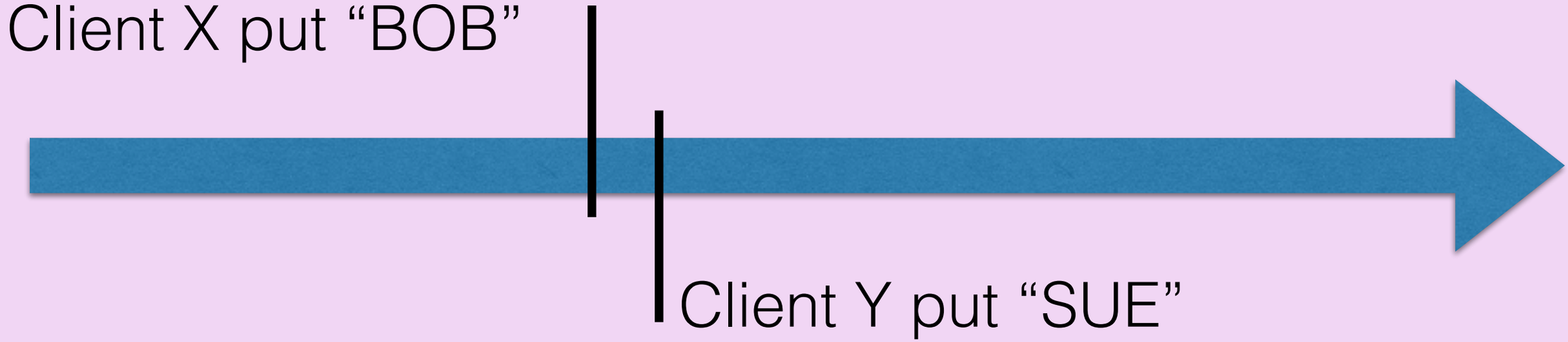
Requires a quorum

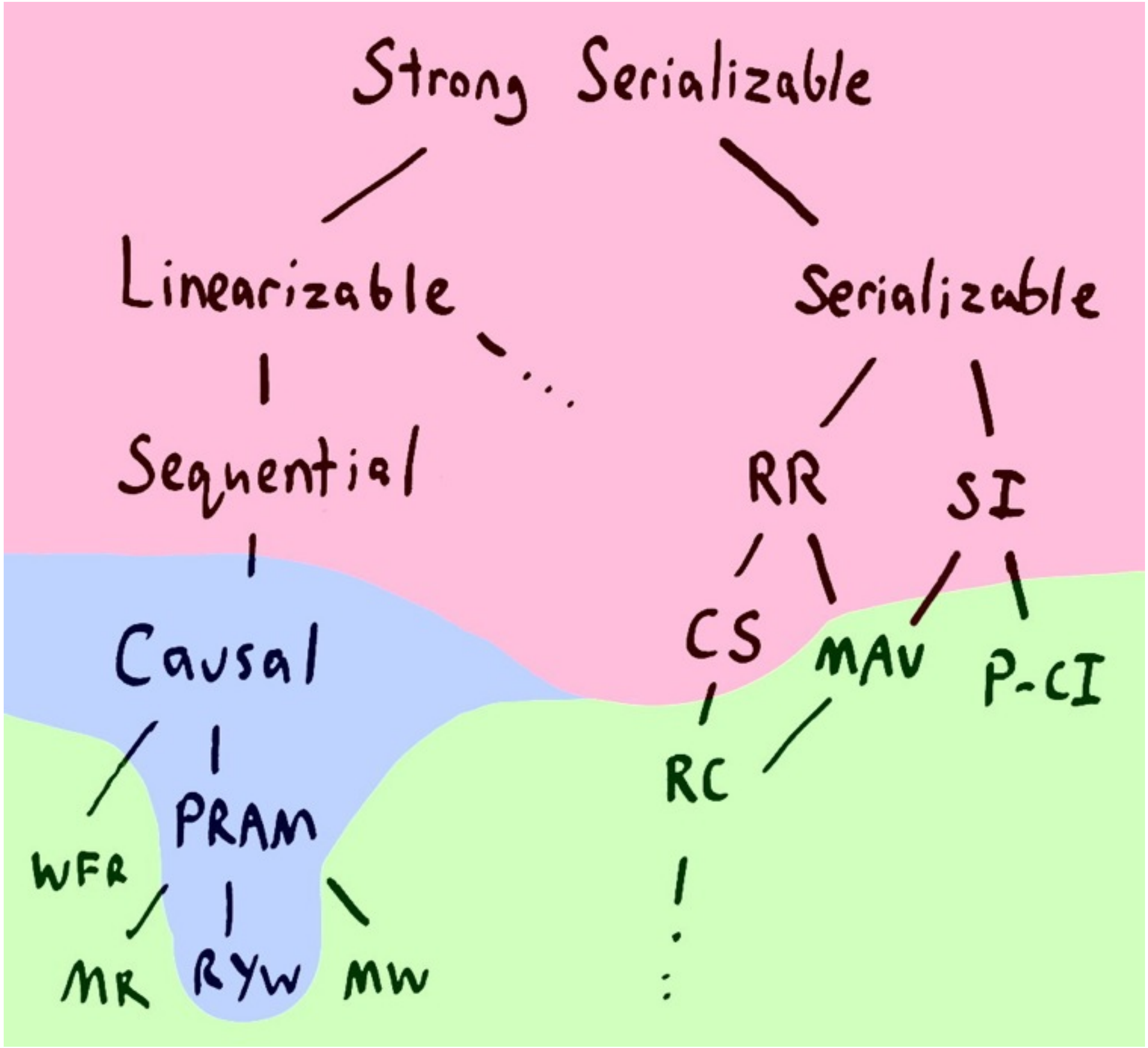
Coordination waits


Replica A
Replica B
Replica C
PUT “bob”
PUT “sue”
Client X
Client Y

Consistent

Events put in a TOTAL ORDER





https://aphyr.com/posts/313-strong-consistency-models

Eventual Consistency

Eventual consistency is a consistency model used in distributed computing that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

--Wikipedia

Replica A

Replica B

Replica C

C′

A′

B′

PUT “bob”

PUT “sue”

Client X

Client Y

Available

Availability

When serving reads and writes matters more than consistency of data. Deferred consistency.

Fault Tolerance

Low Latency

Low Latency

Amazon found every 100ms of latency cost them 1% in sales.

Low Latency

Google found an extra 0.5 seconds in search page generation time dropped traffic by 20%.

Replica A

Replica B

Replica C

C′

A′

B′

PUT “bob”

PUT “sue”

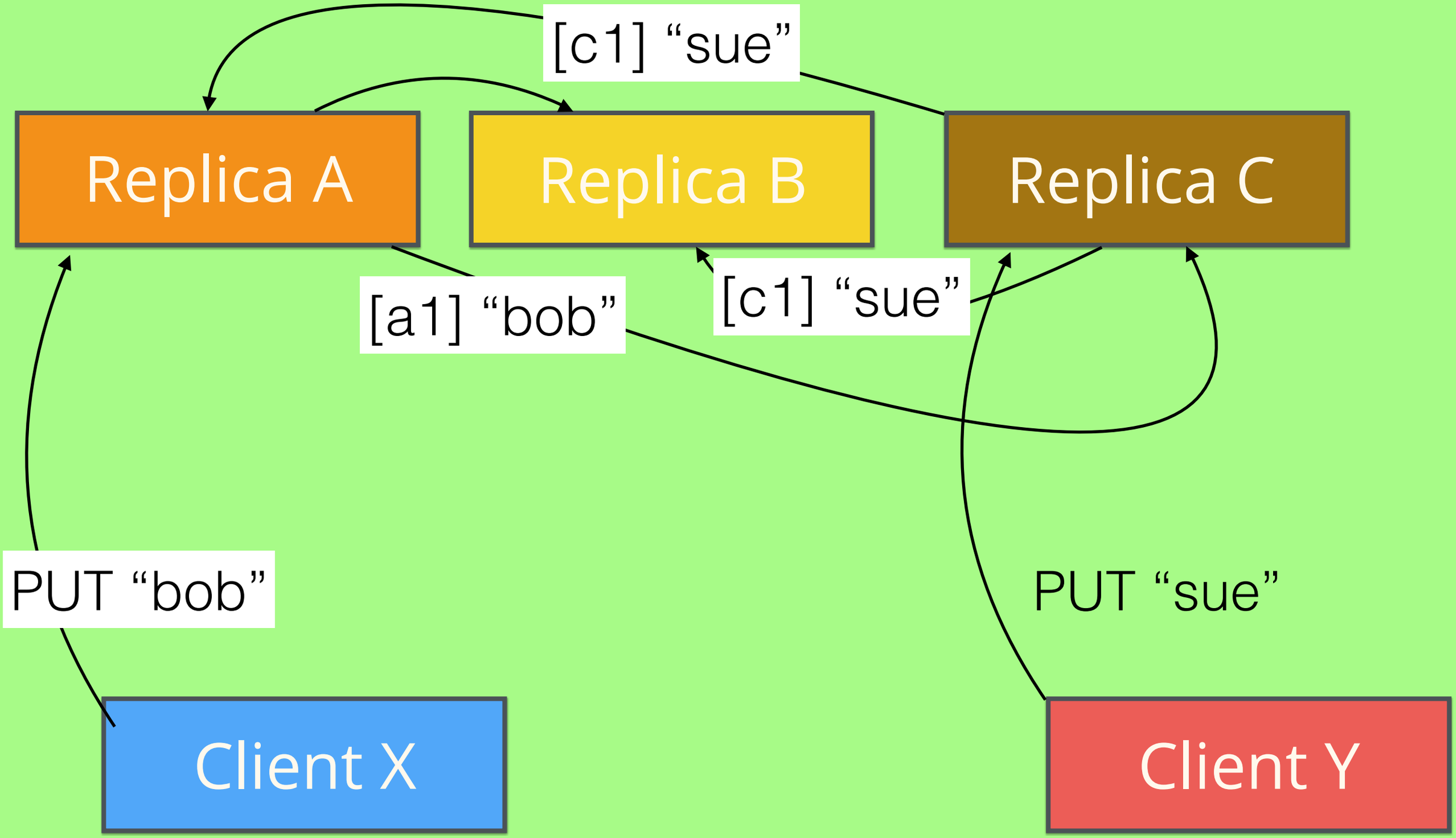
Client X

Client Y

Available

Optimistic replication

No coordination - lower latency

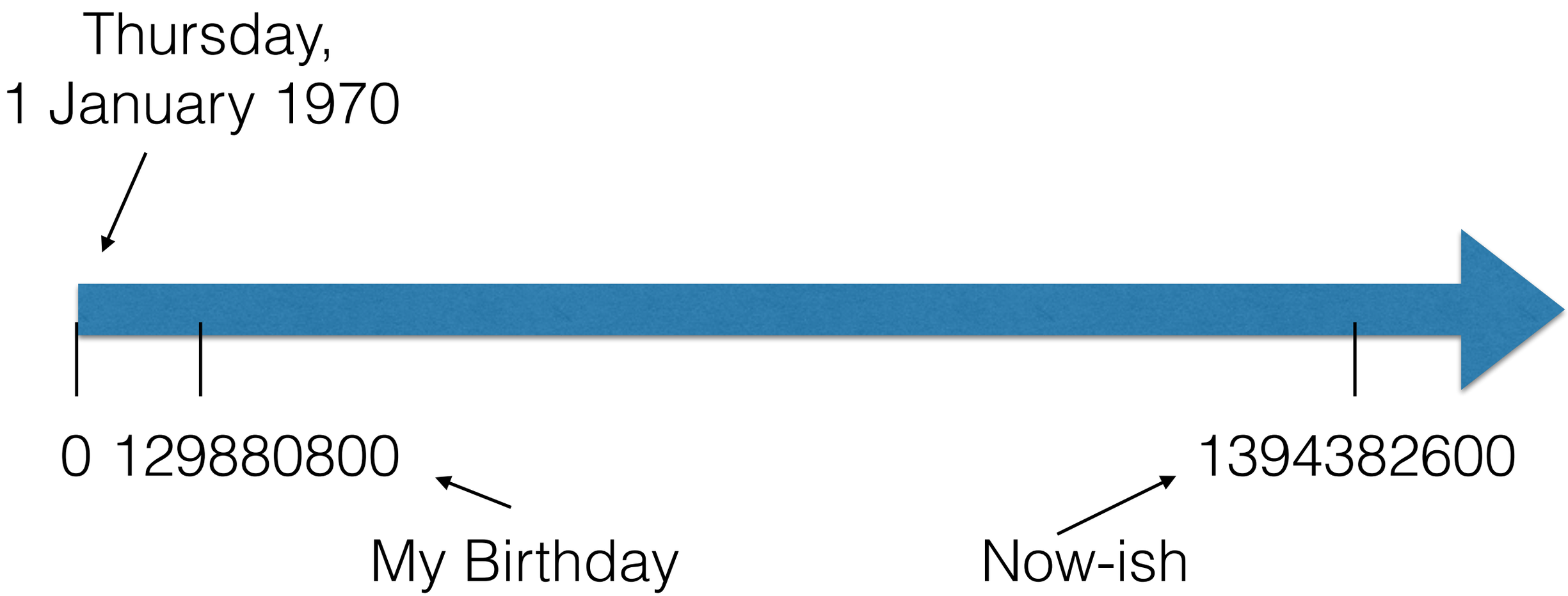

[c1] “sue”
Replica A
Replica B
Replica C
[a1] “bob”
[c1] “sue”
PUT “bob”
PUT “sue”
Client X
Client Y

Low Latency

How Do We Order Updates?

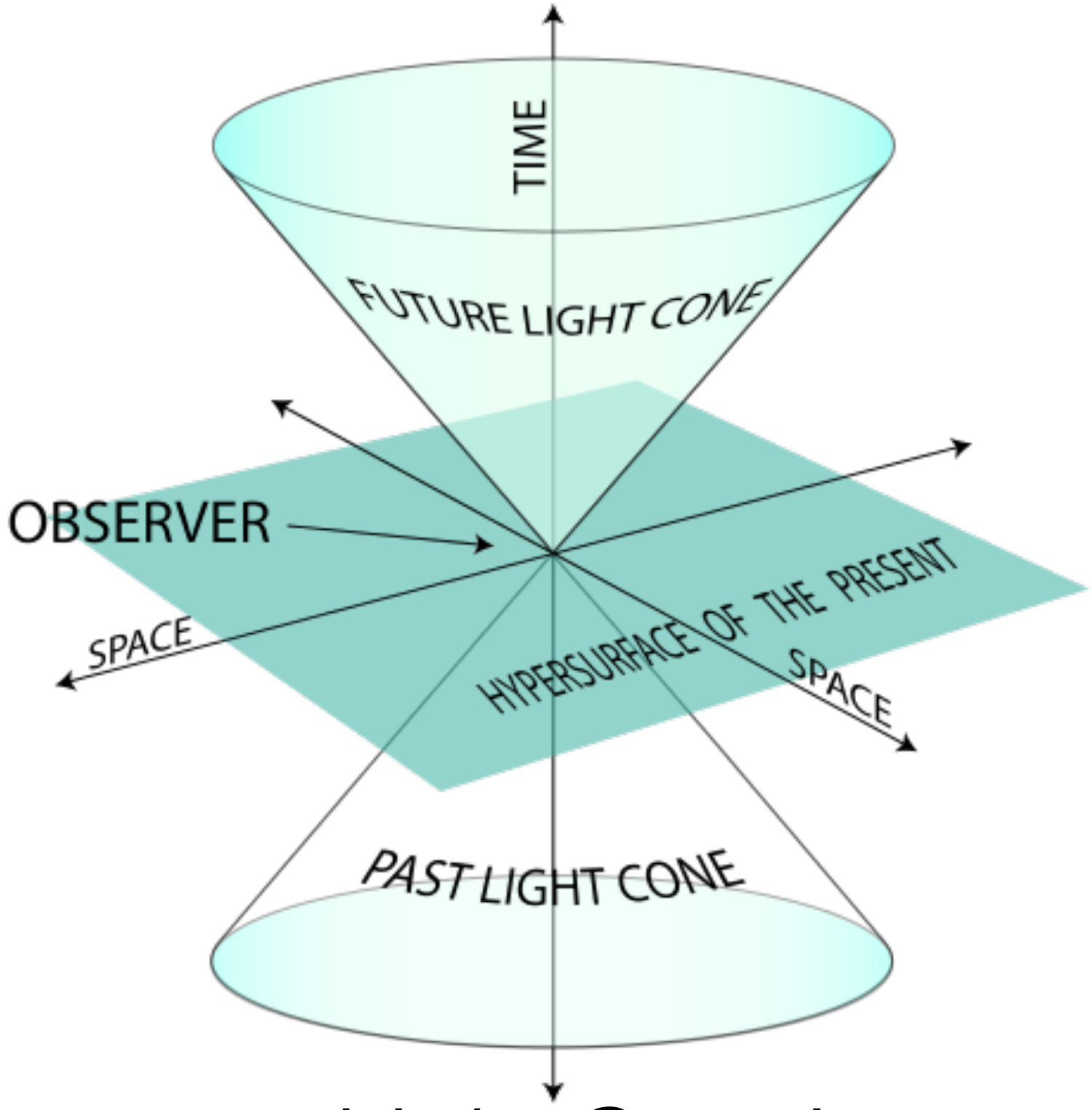
"'Time,' he said, 'is what keeps everything from happening at once.'"

–Google Book Search p.148 “The Giant Anthology of Science Fiction”, edited by Leo Margulies and Oscar Jerome Friend, 1954



Temporal Clocks

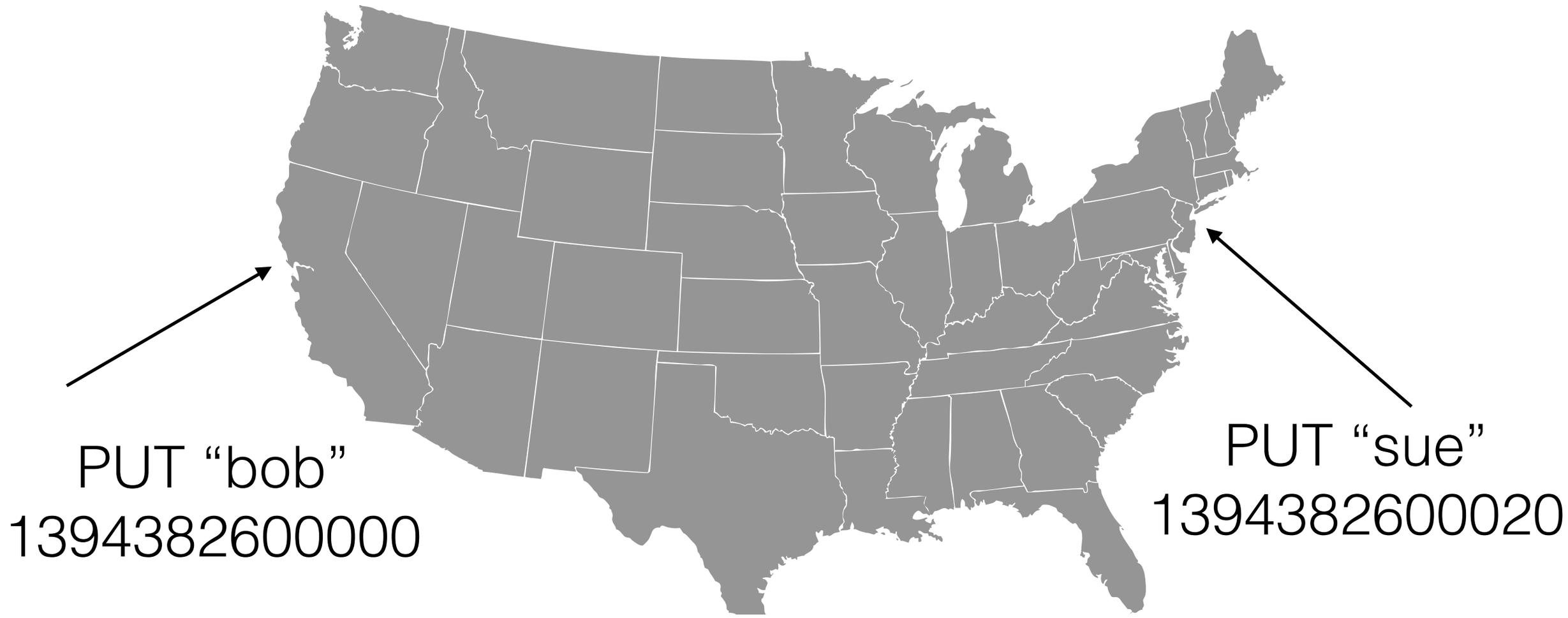
posix time number line



Light Cone!

By SVG version: K. Aainsqatsi at en.wikipediaOriginal PNG version: Stib at en.wikipedia - Transferred from en.wikipedia to Commons.(Original text: self-made), CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2210907

Physics Problem



temporal clocks

✳CAN

- A could NOT have caused B
- A could have caused B

✳CAN’T

- A caused B

Dynamo

The Shopping Cart

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available

Created by Creative Stall
from Noun Project

Created by Creative Stall
from Noun Project

Created by Creative Stall
from Noun Project

Created by Creative Stall
from Noun Project

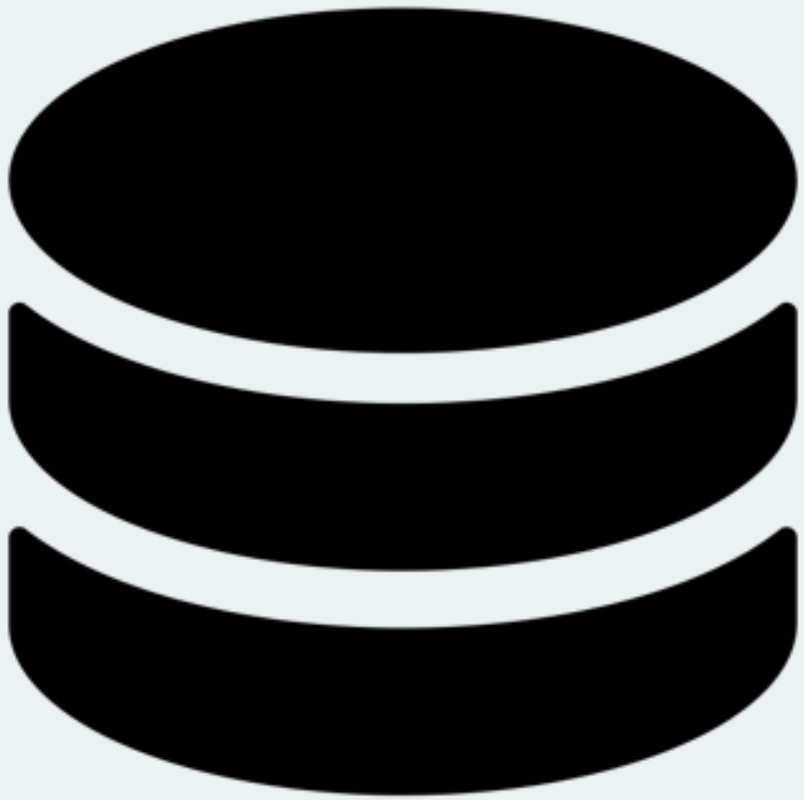
Created by Creative Stall
from Noun Project

Created by Creative Stall
from Noun Project

2 REPLICAS

Created by Creative Stall
from Noun Project

1 KEY

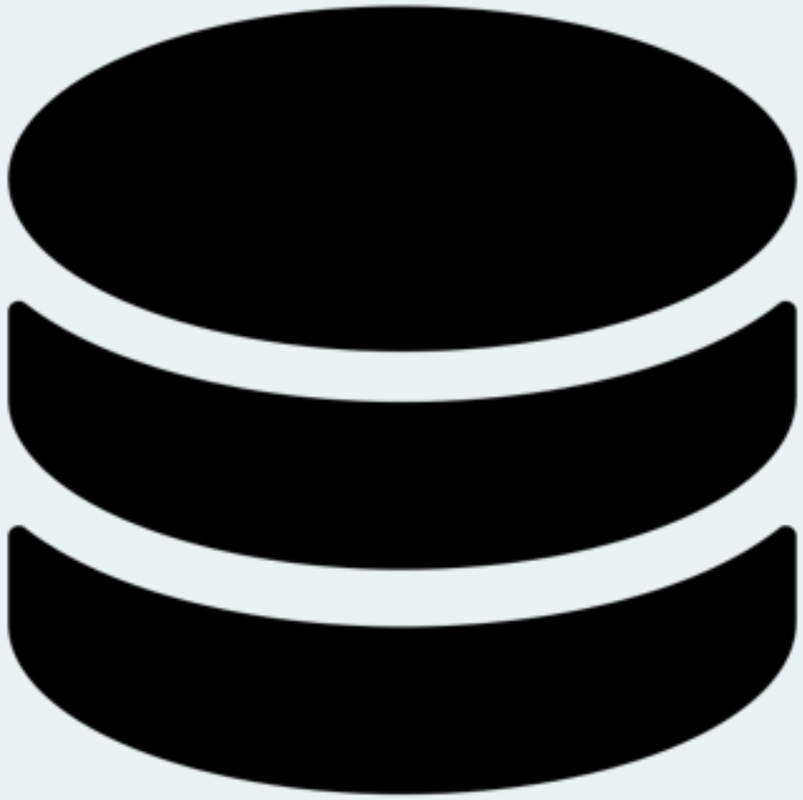


Created by Creative Stall
from Noun Project

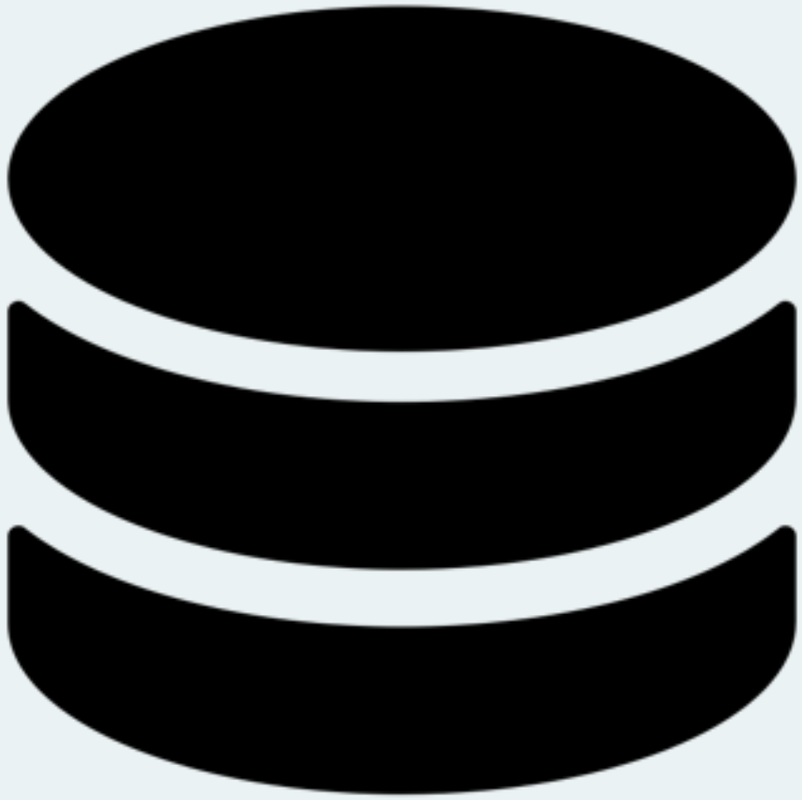
1 CLIENT

Optimistic replication

No coordination - lower latency



REPLICATE



Created by Creative Stall
from Noun Project

Created by Creative Stall
from Noun Project

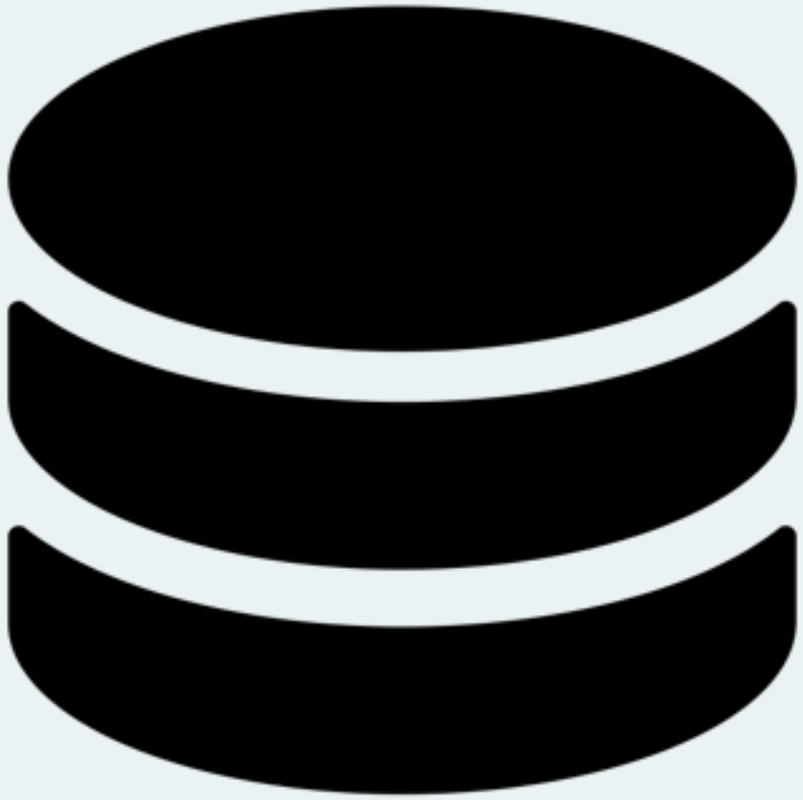
GET

PUT

UPDATE

Created by Amy Schwartz
from the Noun Project

Created by Amy Schwartz
from the Noun Project



Created by Creative Stall
from Noun Project

PUT

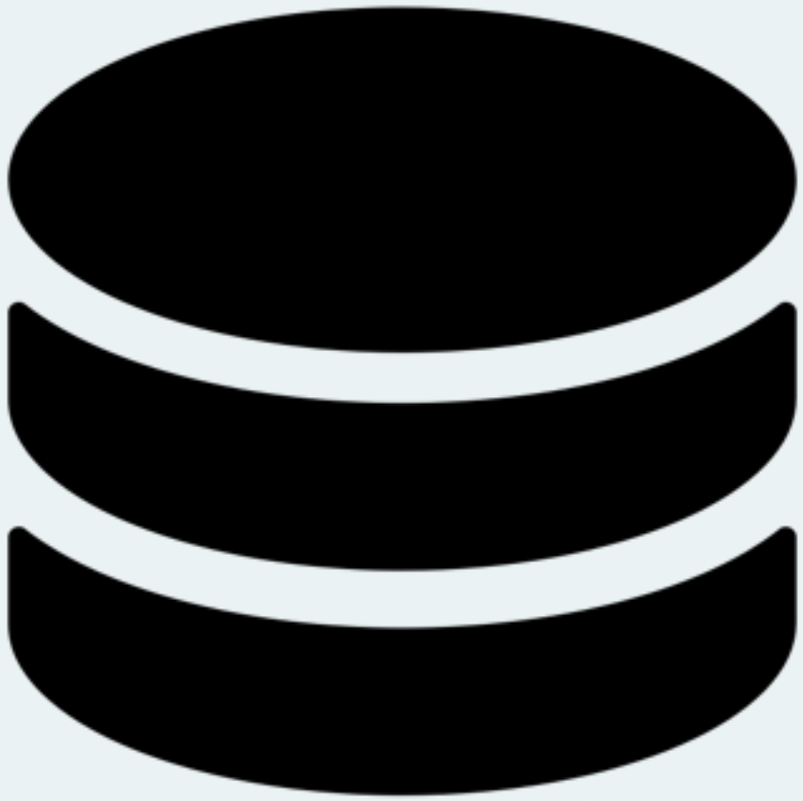
Created by Creative Stall
from Noun Project

PUT

Created by Amy Schwartz
from the Noun Project

Created by Amy Schwartz
from the Noun Project

Quorum

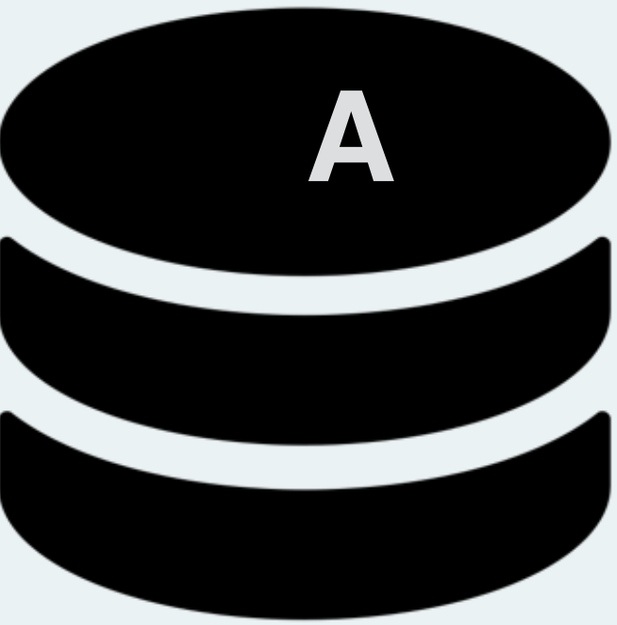


Created by Creative Stall
from Noun Project

Created by Creative Stall
from Noun Project

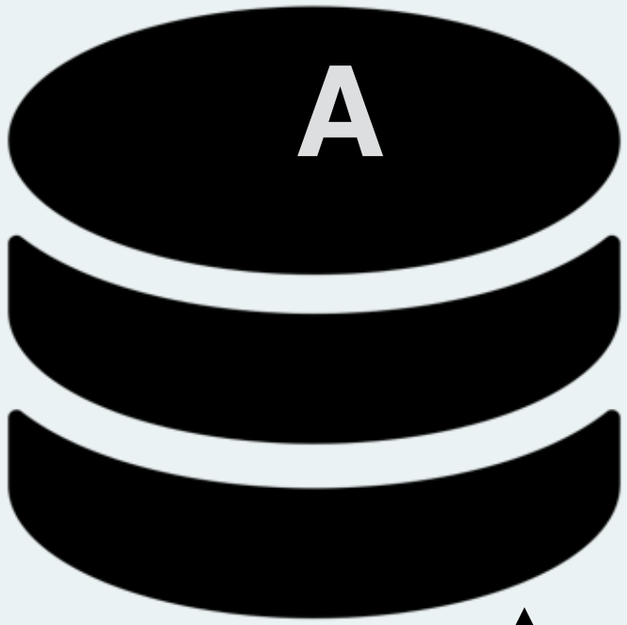
Created by Charlie Bob Gordon
from Noun Project

TEMPORAL TIME



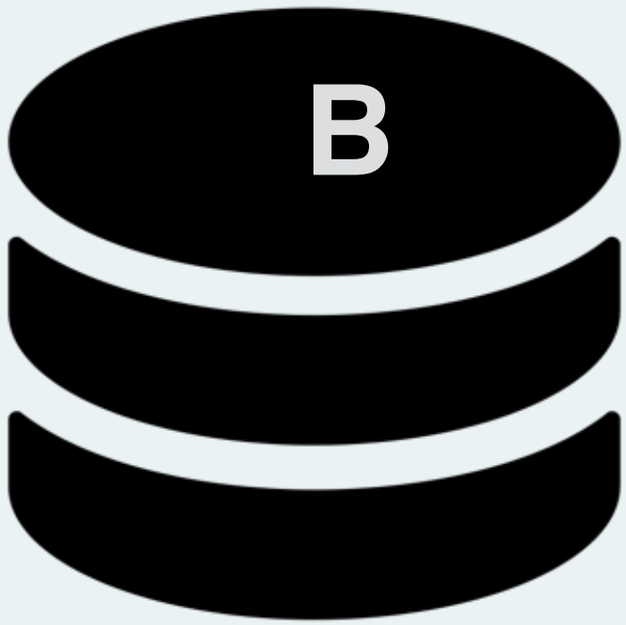
Created by Creative Stall
from Noun Project

GET



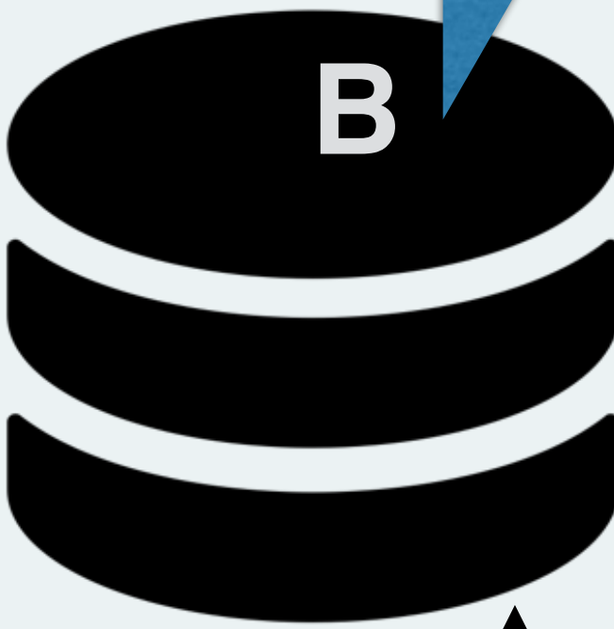
Created by Creative Stall
from Noun Project

PUT



Created by Creative Stall
from Noun Project

GET



Created by Creative Stall
from Noun Project

PUT

Created by Amy Schwartz
from the Noun Project

Created by Amy Schwartz
from the Noun Project

Created by Amy Schwartz
from the Noun Project

Created by Amy Schwartz
from the Noun Project

Timestamp - total order

155196119890 > **155196118001**

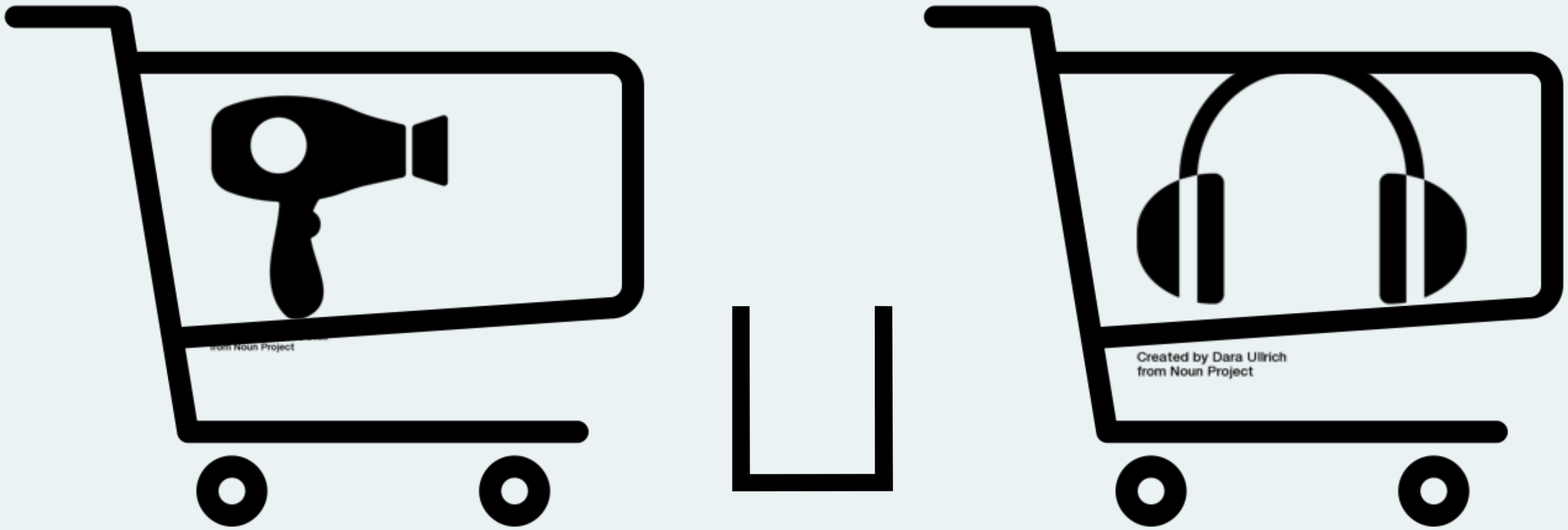
Created by Amy Schwartz
from the Noun Project

Created by Dara Ullrich
from Noun Project

Created by Amy Schwartz
from the Noun Project

Created by Amy Schwartz
from the Noun Project

Logical clock - partial order



Created by Amy Schwartz
from the Noun Project

Created by Amy Schwartz
from the Noun Project

Created by Amy Schwartz
from the Noun Project

Clocks, Time, And the Ordering of Events

- Logical Time
- Causality

- A influenced B
- A and B happened at the same time
- Per-process clocks, only tick when something happens

Leslie Lamport http://dl.acm.org/citation.cfm?id=359563

Detection of Mutual Inconsistency in Distributed Systems

http://zoo.cs.yale.edu/classes/cs426/2013/bib/parker83detection.pdf

Version Vectors - updates to a data item

Version Vectors or Vector Clocks?

version vectors - updates to a data item

http://haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/

Summary

- Distributed systems exist (scale out)
- There is a trade off (Consistency/Availability)
- To decide on a value we need to “order” updates
- Temporal time is inadequate
- Logical time can help

Version Vectors

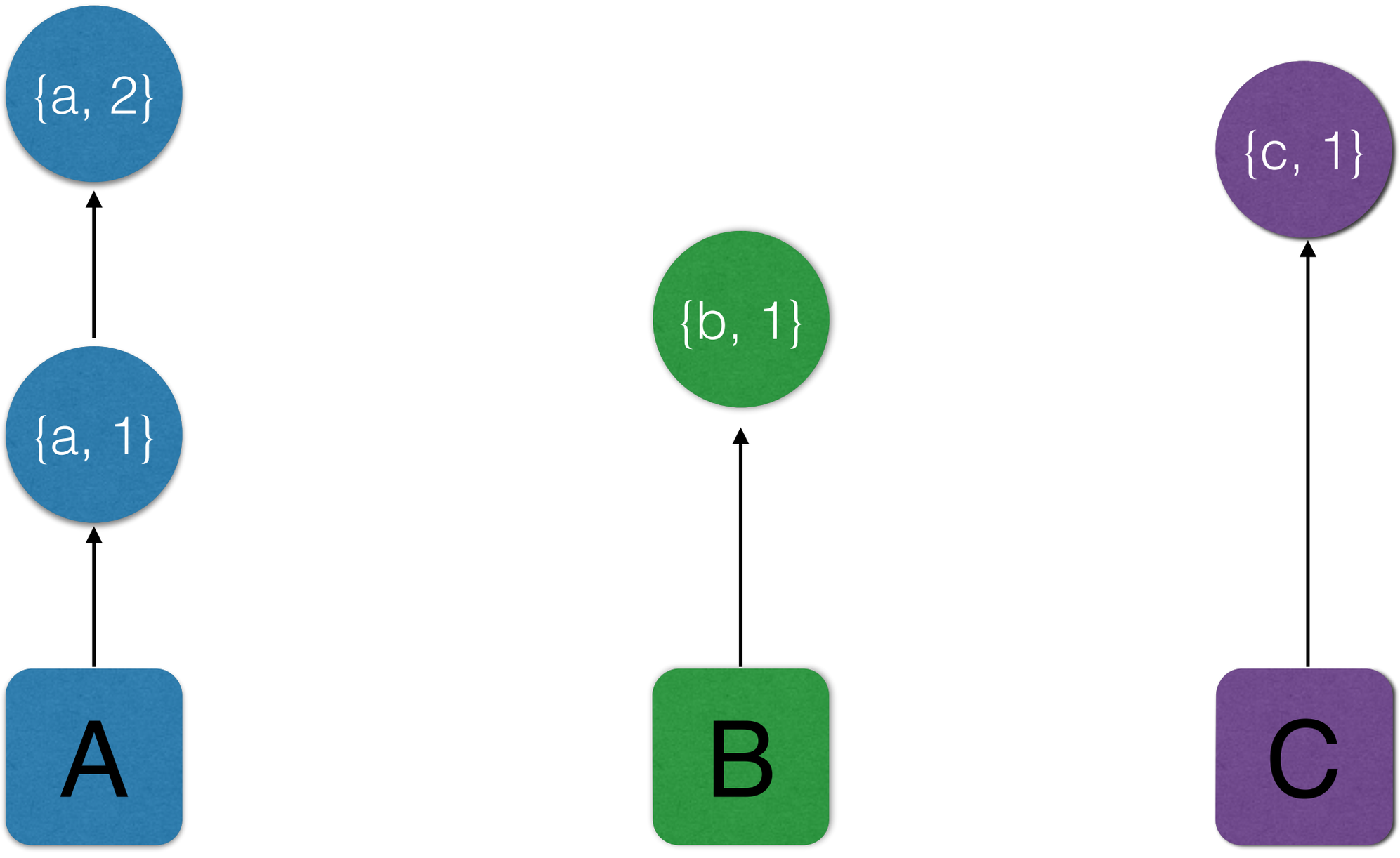
A

B

C

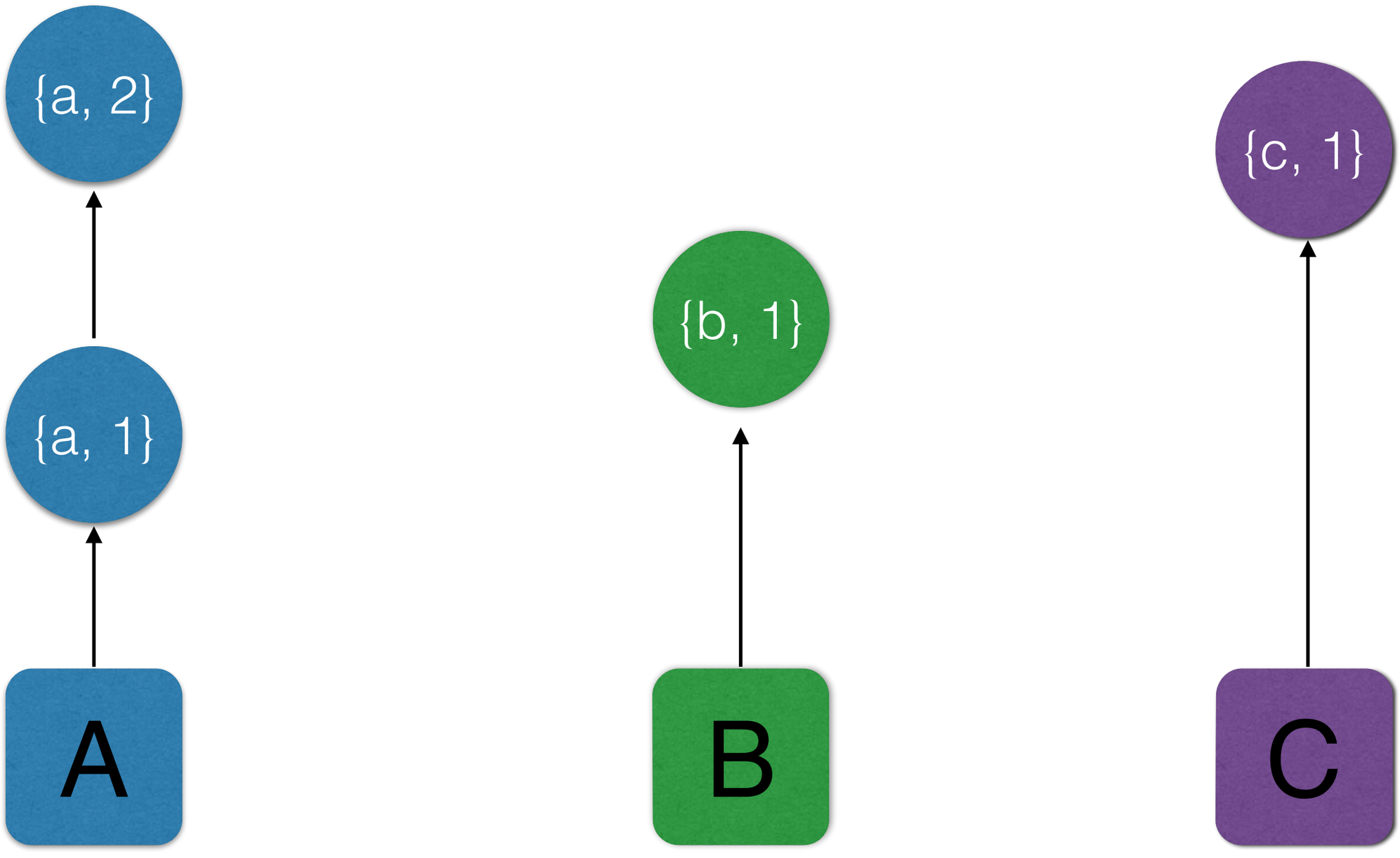
Version Vectors

[{a, 2}, {b, 1}, {c, 1}]



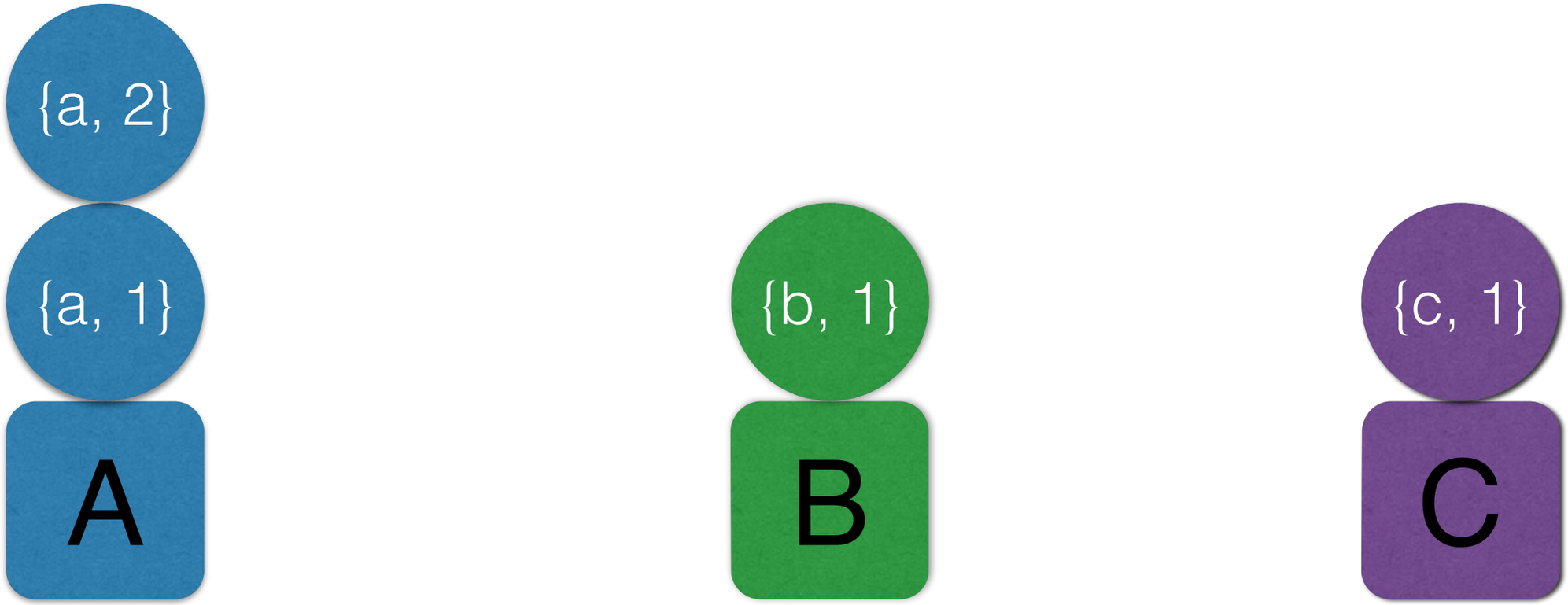
Version Vectors

[{a, 2}, {b, 1}, {c, 1}]



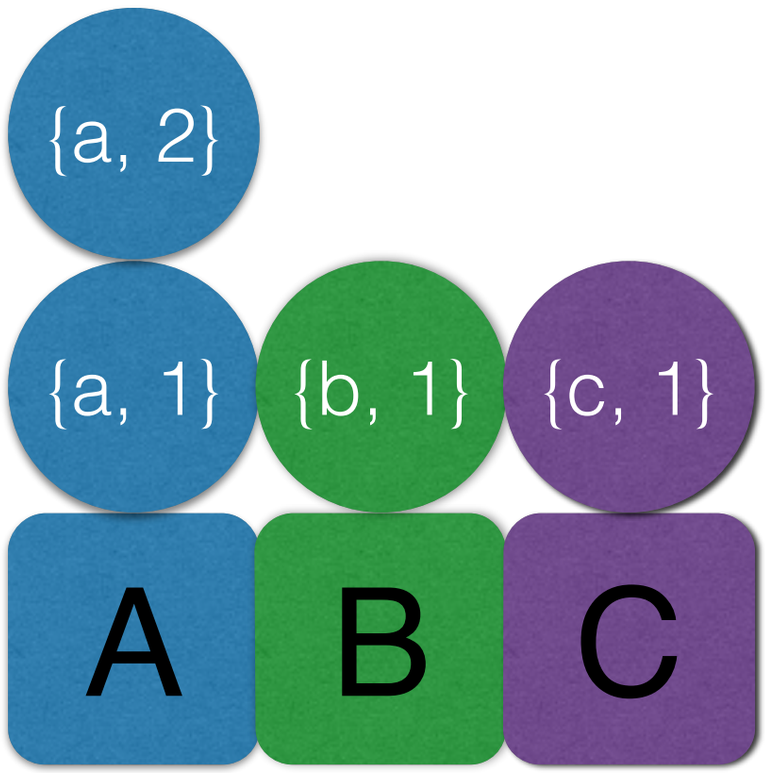
Version Vectors

[{a, 2}, {b, 1}, {c, 1}]



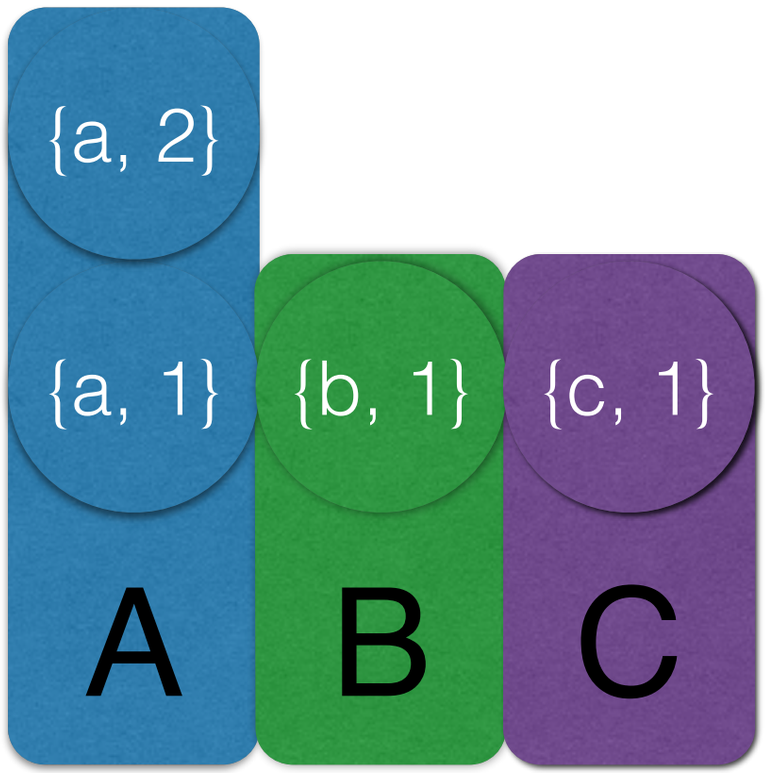
Version Vectors

[{a, 2}, {b, 1}, {c, 1}]



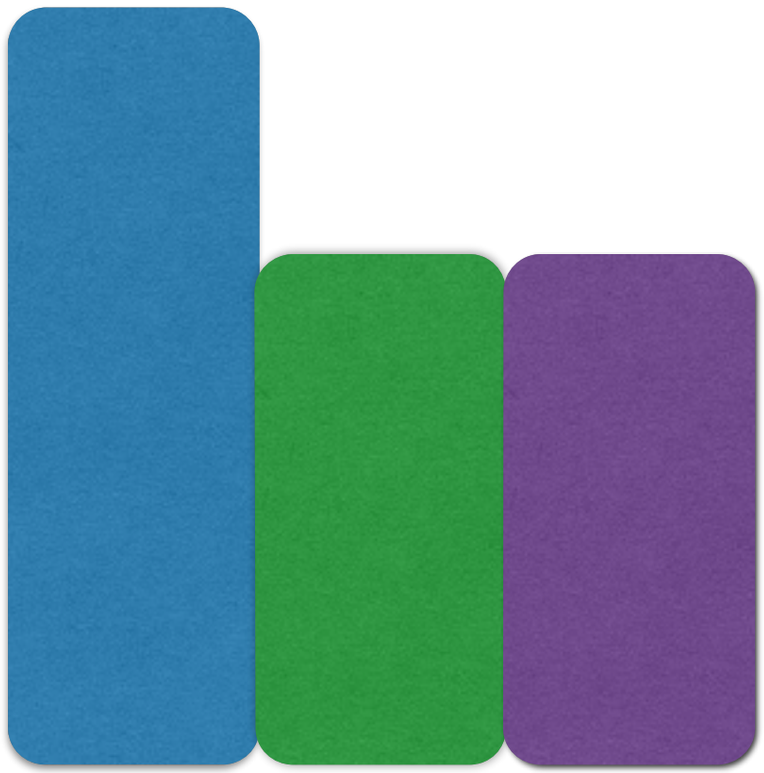
Version Vectors

[{a, 2}, {b, 1}, {c, 1}]



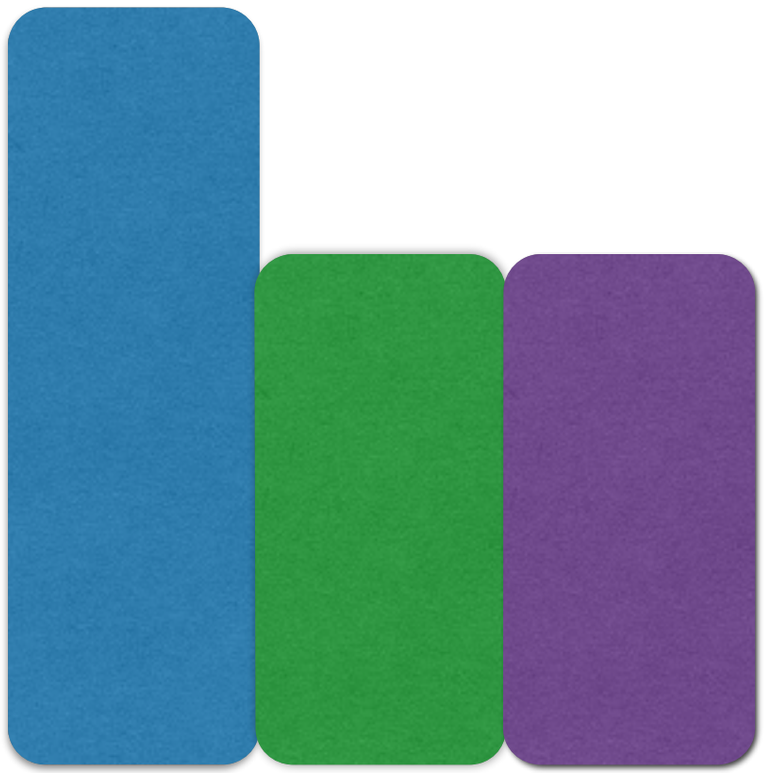
Version Vectors

[{a,2}, {b,1}, {c,1}]



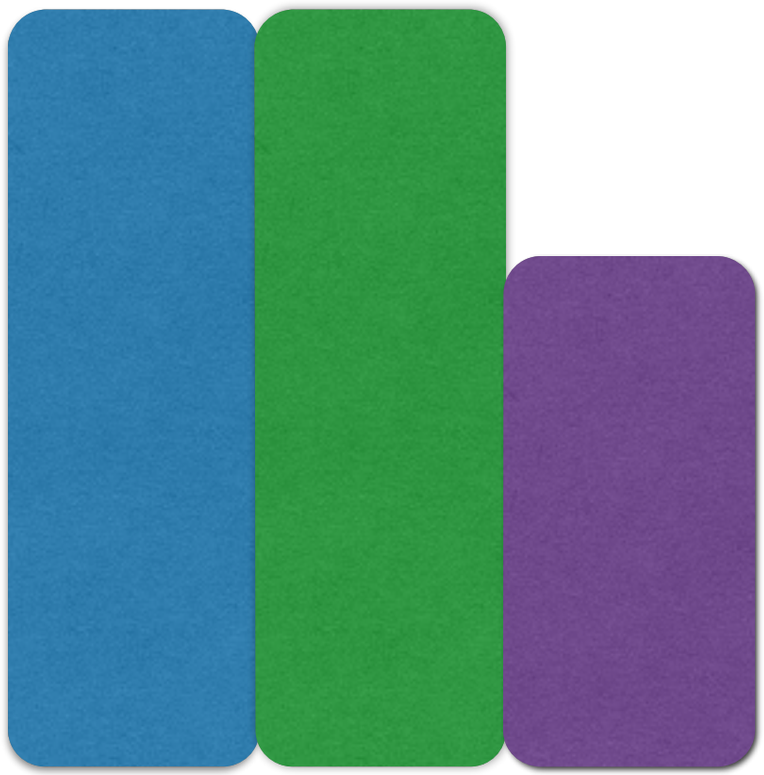
Version Vectors Update

[{a,2}, {b,1}, {c,1}]



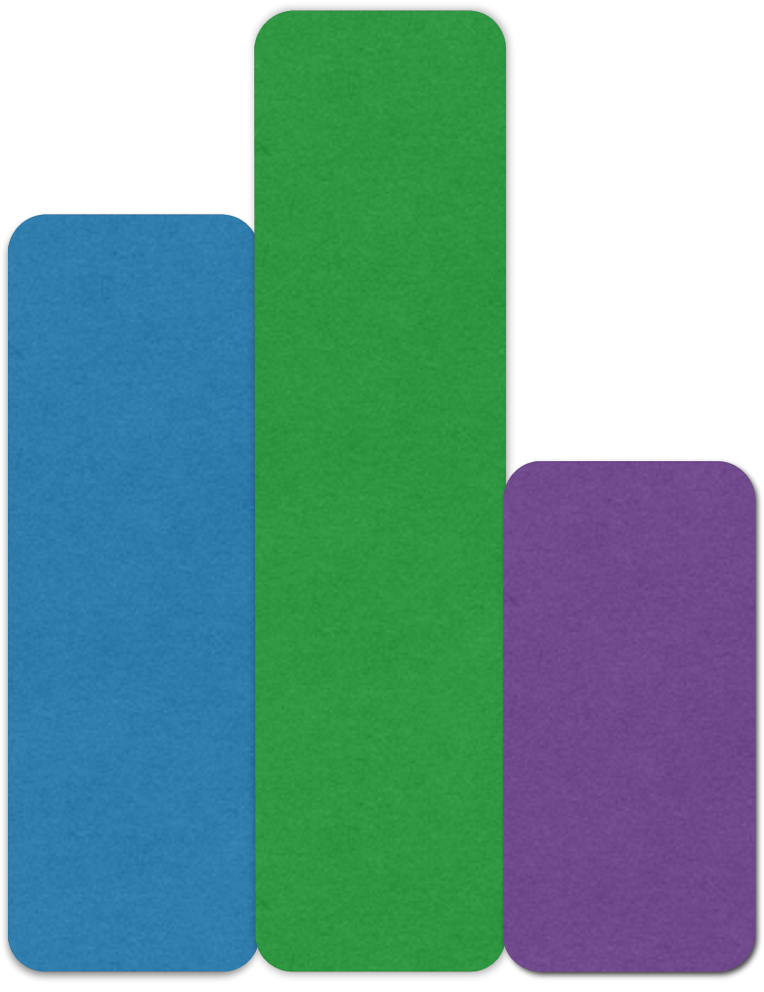
Version Vectors Update

[{a,2}, {b,2}, {c,1}]



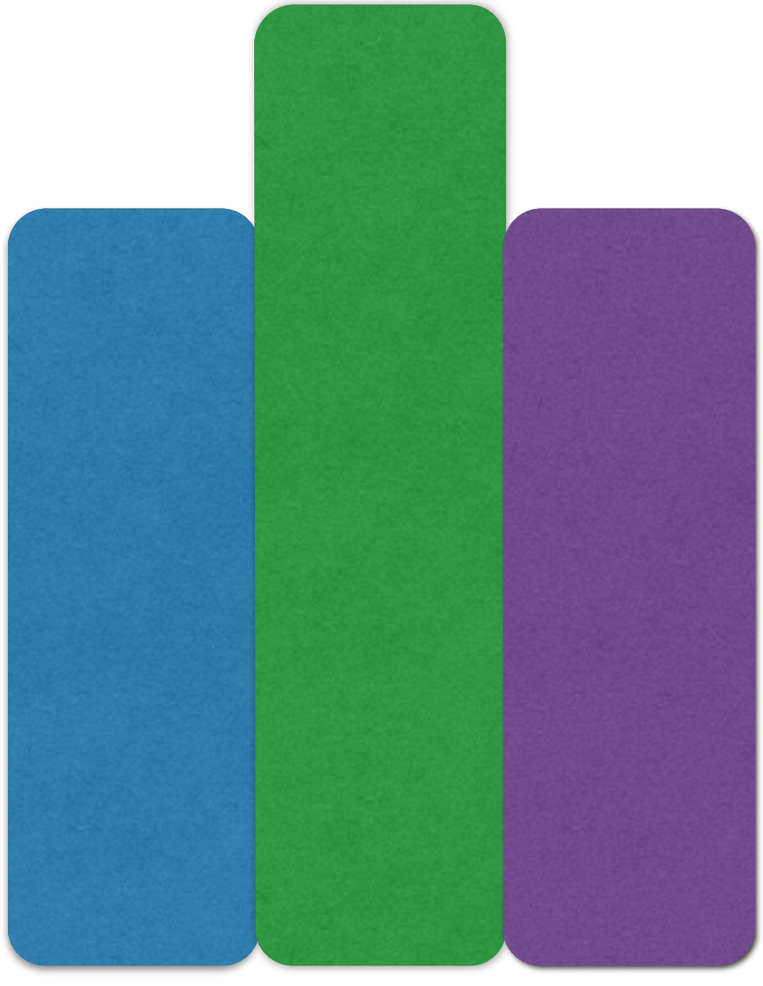
Version Vectors Update

[{a,2}, {b,3}, {c,1}]



Version Vectors Update

[{a,2}, {b,3}, {c,2}]



Version Vectors Descends

- A descends B : A >= B
- A has seen all that B has
- A summarises at least the same history as B

Version Vectors Descends

[{a,2}, {b,3}, {c,2}]

[{a,2}, {b,3}, {c,2}]

[{a,4}, {b,3}, {c,2}]

>=

[{a,2}, {b,3}, {c,2}]

[]

[{a,2}, {b,3}, {c,2}]

Version Vectors Dominates

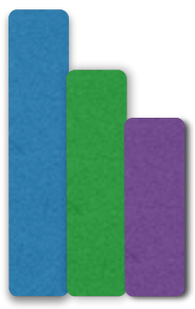
- A dominates B : A > B
- A has seen all that B has, and at least one more event
- A summarises a greater history than B

Version Vectors Dominates

[{a,1}]

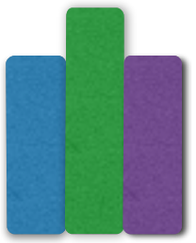
[]

[{a,4}, {b,3}, {c,2}]

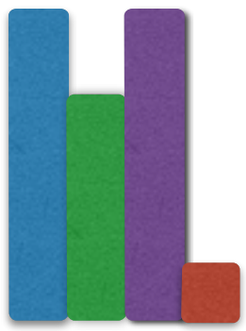


>

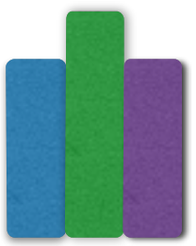
[{a,2}, {b,3}, {c,2}]



[{a,5}, {b,3}, {c,5}, {d, 1}]



[{a,2}, {b,3}, {c,2}]



Version Vectors
Concurrent

- A concurrent with B : A | B
- A does not descend B AND B does not descend A
- A and B summarise disjoint events
- A contains events unseen by B AND B contains events unseen by A

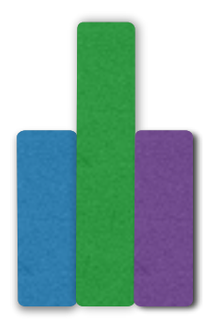
Version Vectors
Concurrent

[{a,1}]

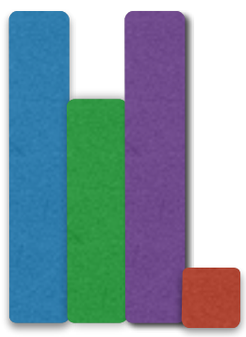
[{b,1}]

[{a,4}, {b,3}, {c,2}]

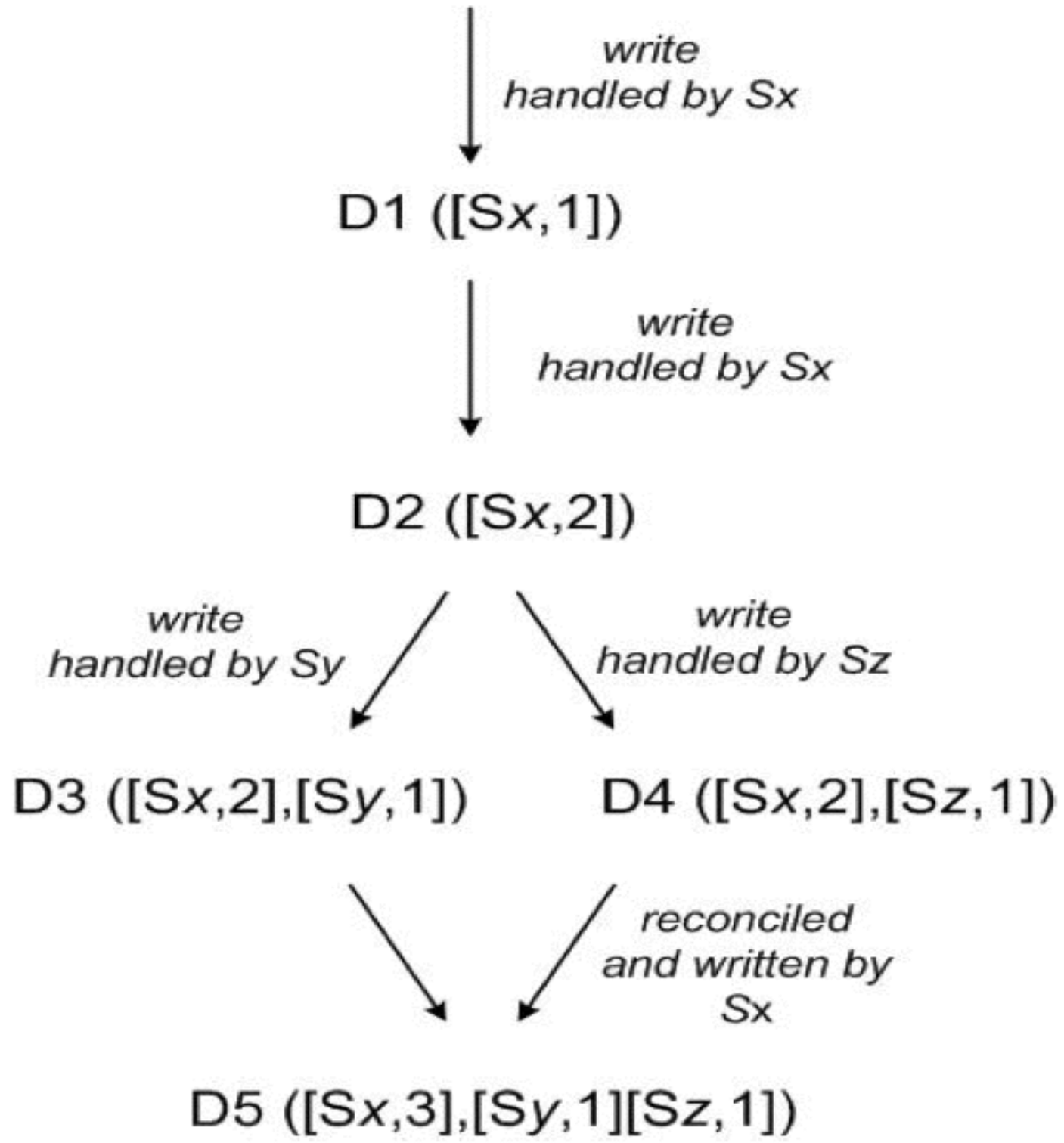
[{a,2}, {b,4}, {c,2}]



[{a,5}, {b,3}, {c,5}, {d, 1}]



[{a,2}, {b,4}, {c,2}, {e,1}]

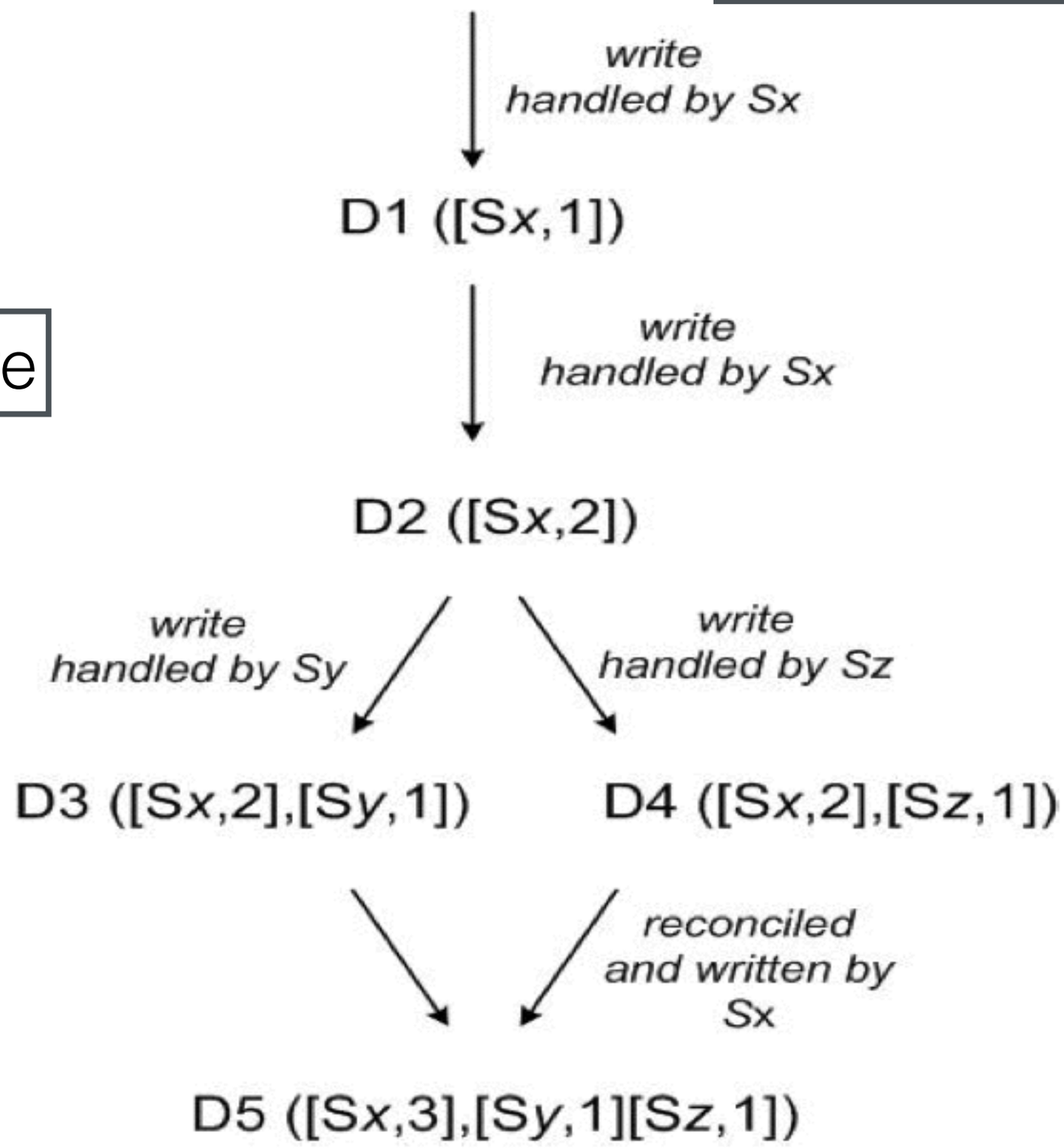

write
handled by Sx
D1 ([Sx,1])
write
handled by Sx
D2 ([Sx,2])
write
handled by Sy
write
handled by Sz
D3 ([Sx,2],[Sy,1])
D4 ([Sx,2],[Sz,1])
reconciled
and written by
Sx
D5 ([Sx,3],[Sy,1][Sz,1])

Logical Clocks

happens before

concurrent
divergent

convergent



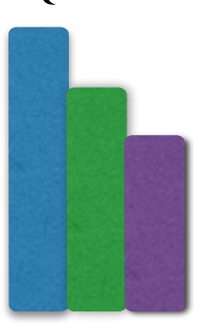
Version Vectors Merge

- A merge with B : A ⊔ B
- A ⊔ B = C
- C >= A and C >= B
- If A | B C > A and C > B
- C summarises all events in A and B
- Pairwise max of counters in A and B

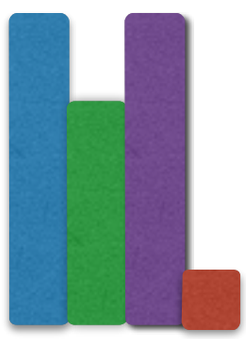
Version Vectors Merge

[{a,1}]

[{a,4}, {b,3}, {c,2}]



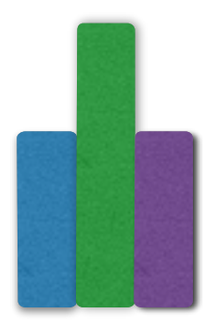
[{a,5}, {b,3}, {c,5}, {d, 1}]



⊔

[{b,1}]

[{a,2}, {b,4}, {c,2}]



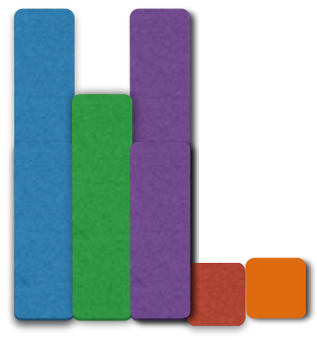
[{a,2}, {b,4}, {c,2}, {e,1}]

Version Vectors Merge

[{a,1}{b,2}]

[{a,4}, {b,4}, {c,2}]

[{a,5}, {b,3}, {c,5}, {d, 1},{e,1}]



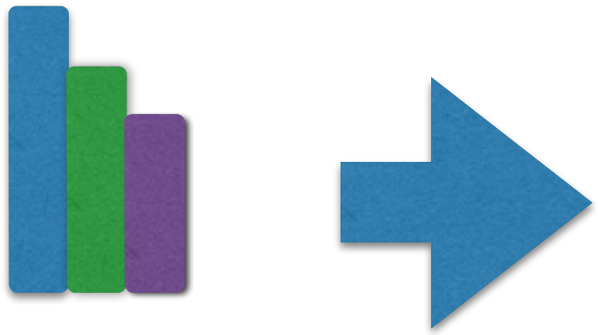
Syntactic Merging

- Discarding “seen” information
- Retaining concurrent values
- Merging divergent clocks

Temporal vs Logical

A

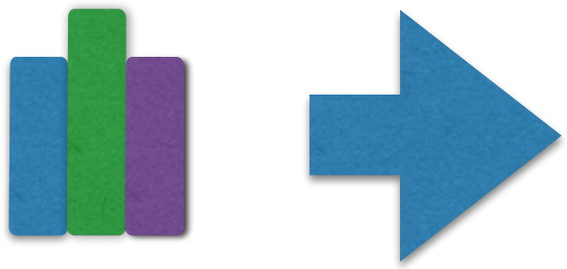
[{a,4}, {b,3}, {c,2}]



“Bob”

B

[{a,2}, {b,3}, {c,2}]



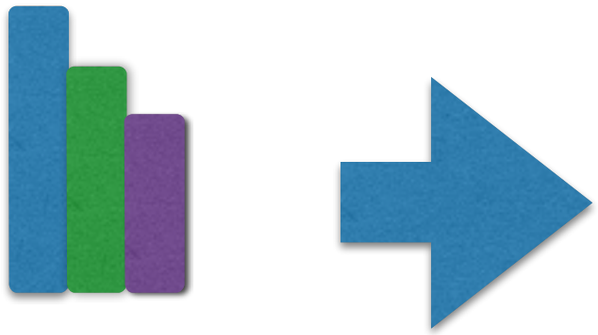
“Sue”

?

Temporal vs Logical

A

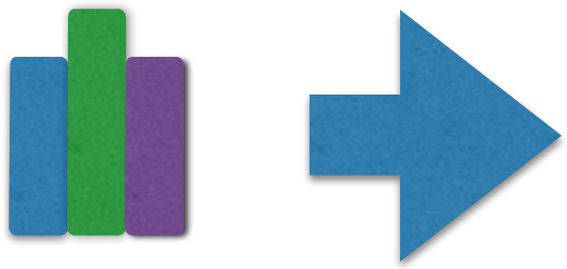
[{a,4}, {b,3}, {c,2}]



“Bob”

B

[{a,2}, {b,3}, {c,2}]



“Sue”

Bob

Temporal vs Logical

A

1429533664000 → “Bob”

B

1429533662000 → “Sue”

?

Temporal vs Logical

A

1429533664000 → “Bob”

B

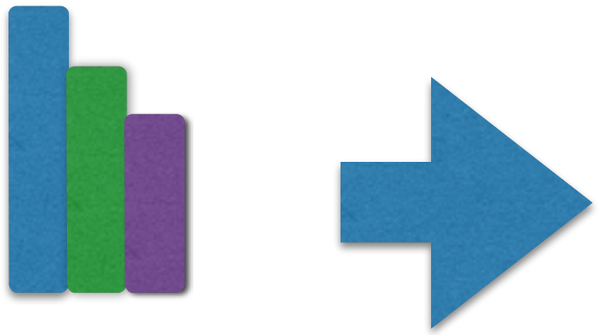
1429533662000 → “Sue”

Bob?

Temporal vs Logical

A

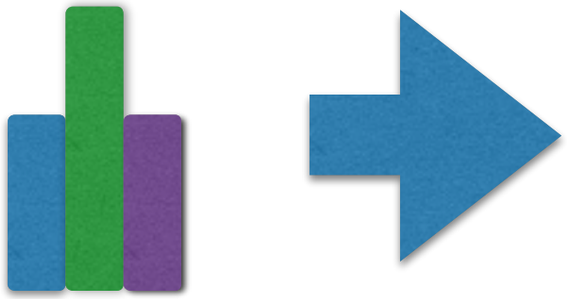
[{a,4}, {b,3}, {c,2}]



"Bob"

B

[{a,2}, {b,4}, {c,2}]



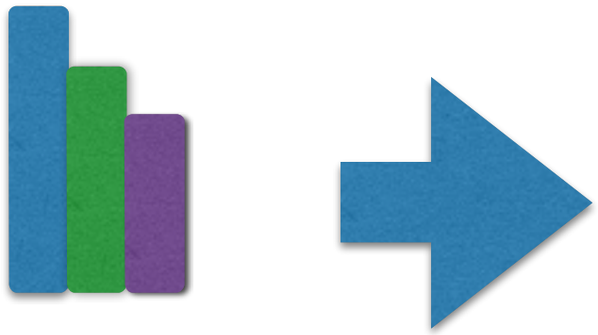
"Sue"

?

Temporal vs Logical

A

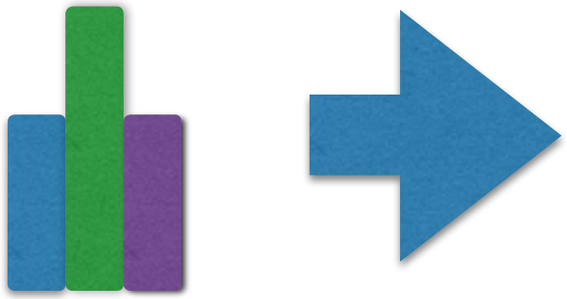
[{a,4}, {b,3}, {c,2}]



"Bob"

B

[{a,2}, {b,4}, {c,2}]



"Sue"

[Bob, Sue]

Temporal vs Logical

A

1429533664000 → “Bob”

B

1429533664001 → “Sue”

?

Temporal vs Logical

A

1429533664000 → “Bob”

B

1429533664001 → “Sue”

Sue?

Summary

- Eventually Consistent Systems allow concurrent updates
- Temporal timestamps can’t capture concurrency
- Logical clocks (Version vectors) can
- Version Vectors are easy

History Repeating

“Those who cannot remember the past are condemned to repeat it"

Terms

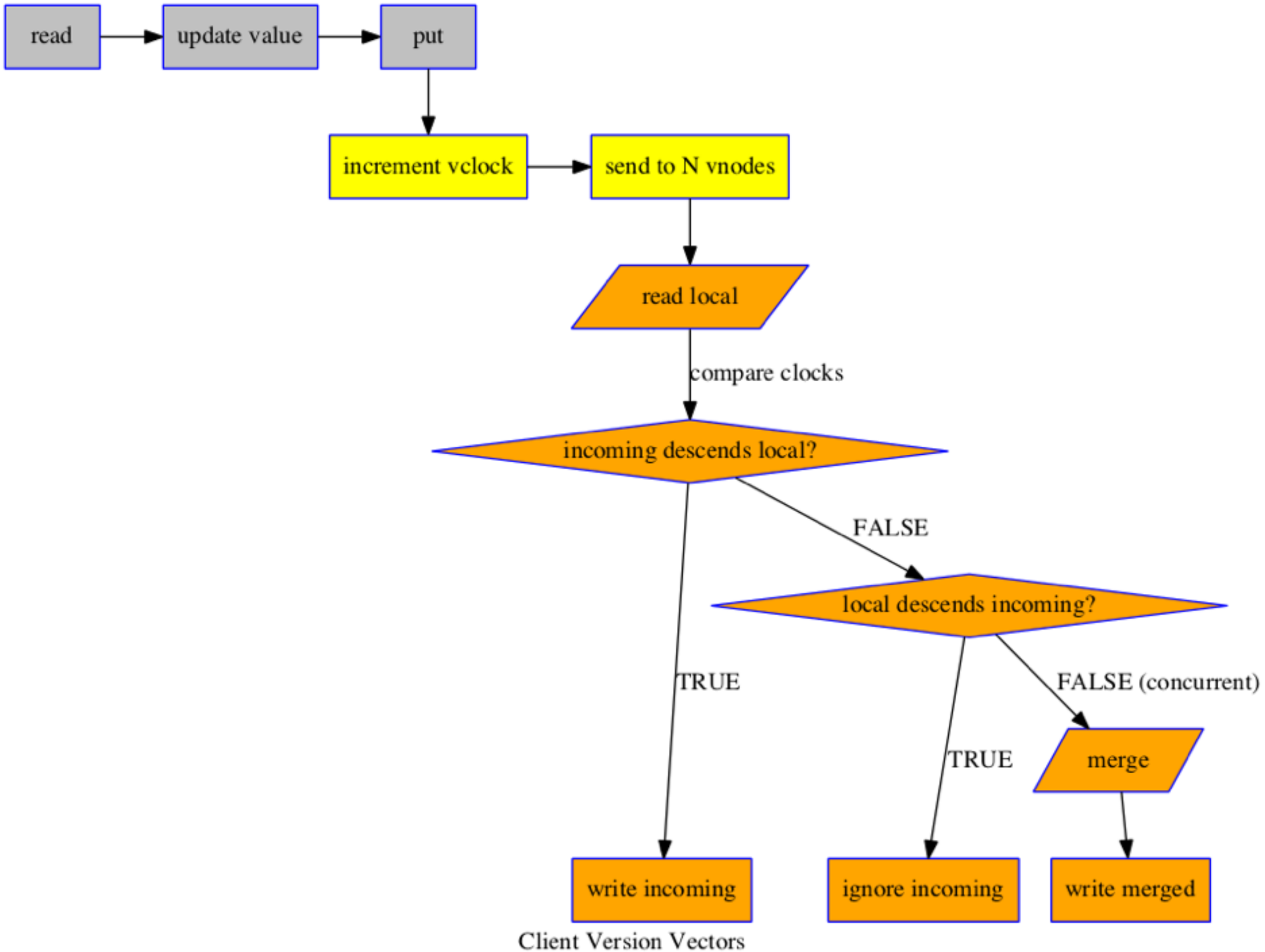
- Local value - stored on disk at some replica
- Incoming value - sent as part of a PUT or replication
- Local clock - The Version Vector of the Local Value
- Incoming clock - The Version Vector of the Incoming Value

Riak Version Vectors

Who’s the actor?

Riak 0.n
Client Side IDs

- Client Code Provides ID
- Riak increments Clock at API boundary
- Riak syntactic merge and stores object
- Read, Resolve, Rinse, Repeat.


read
update value
put
increment vclock
send to N vnodes
read local
compare clocks
incoming descends local?
TRUE
FALSE
local descends incoming?
TRUE
FALSE (concurrent)
TRUE
merge
write incoming
ignore incoming
write merged
Client Version Vectors

Client

read → update value → put

put → increment vclock → send to N vnodes

Riak API

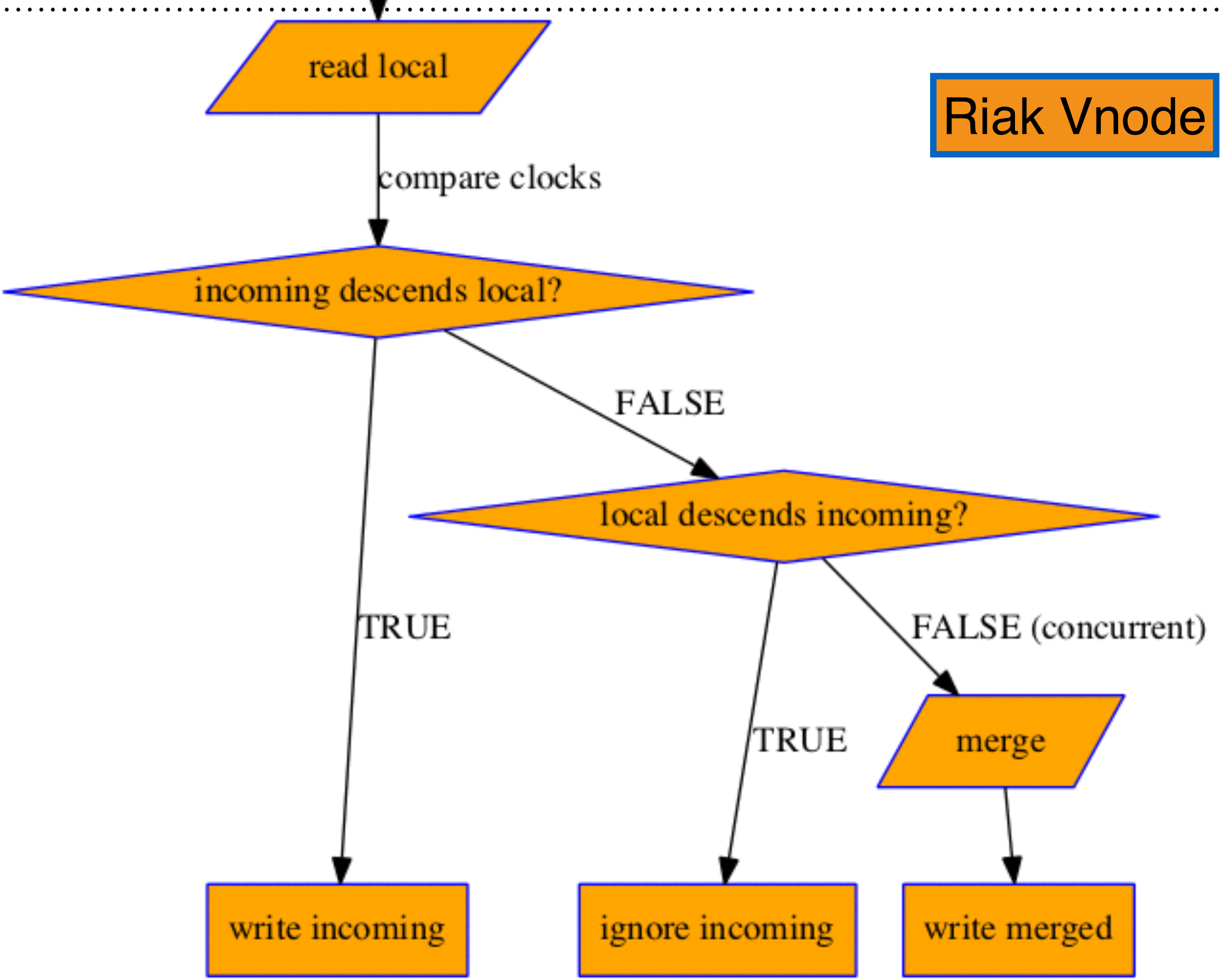
Riak Vnode

read local

compare clo

incoming descends local?

Riak Vnode

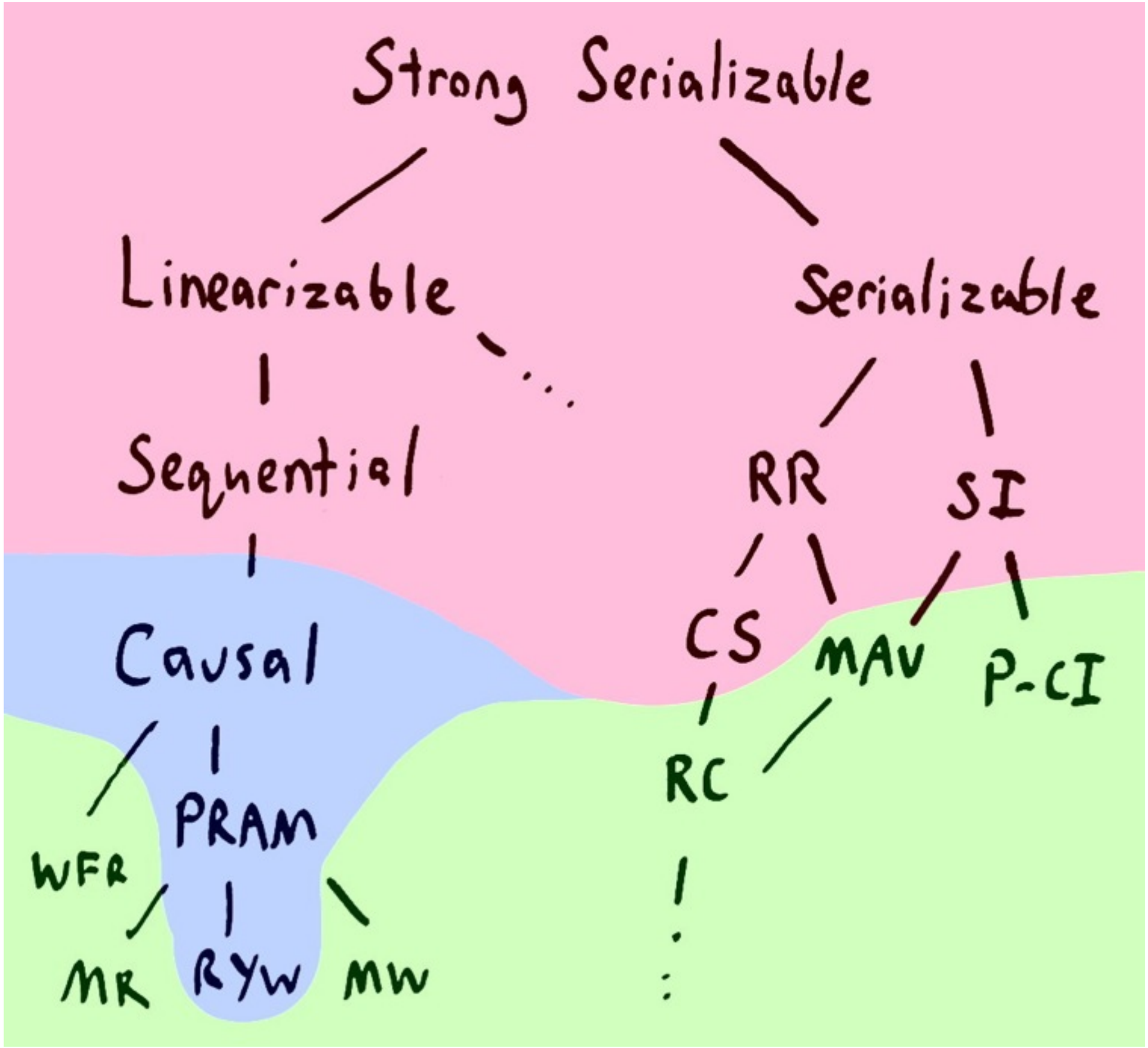

read local
compare clocks
incoming descends local?
TRUE
FALSE
local descends incoming?
TRUE
FALSE (concurrent)
merge
write incoming
ignore incoming
write merged

Conflict Resolution

- Client reads merged clock + sibling values
  - sends new value + clock
  - new clock descends old (eventually!)
  - Store single value

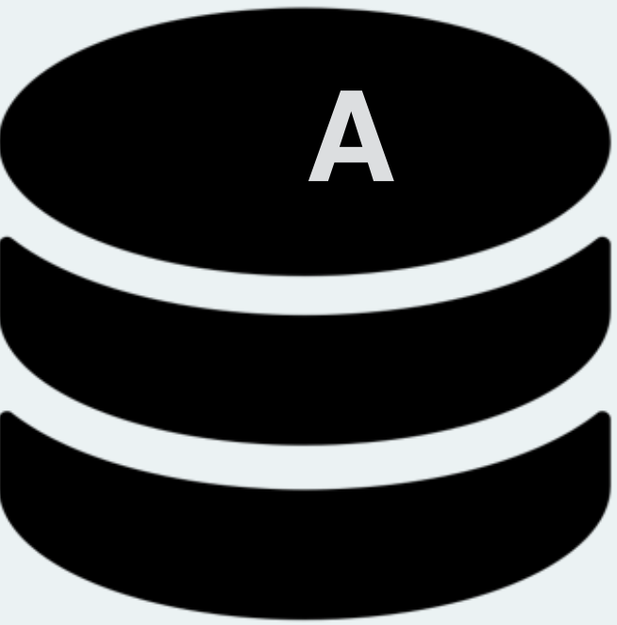
Client Version Vector

What Level of Consistency Do We Require?



https://aphyr.com/posts/313-strong-consistency-models

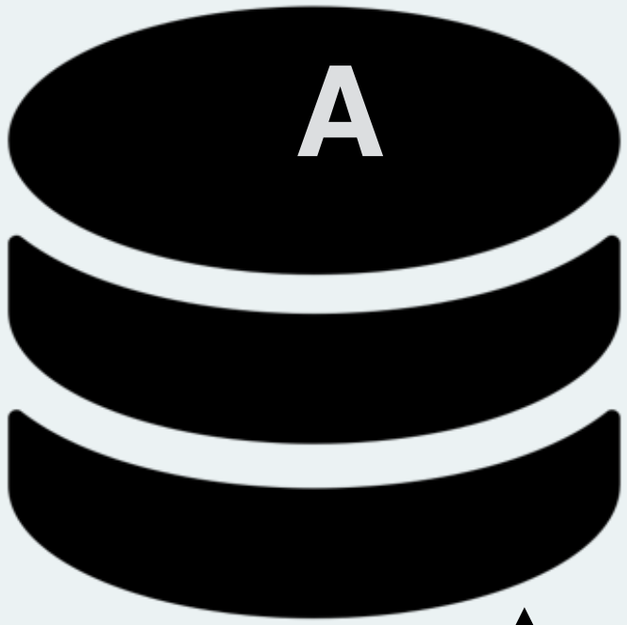
TEMPORAL TIME



Created by Creative Stall
from Noun Project

GET

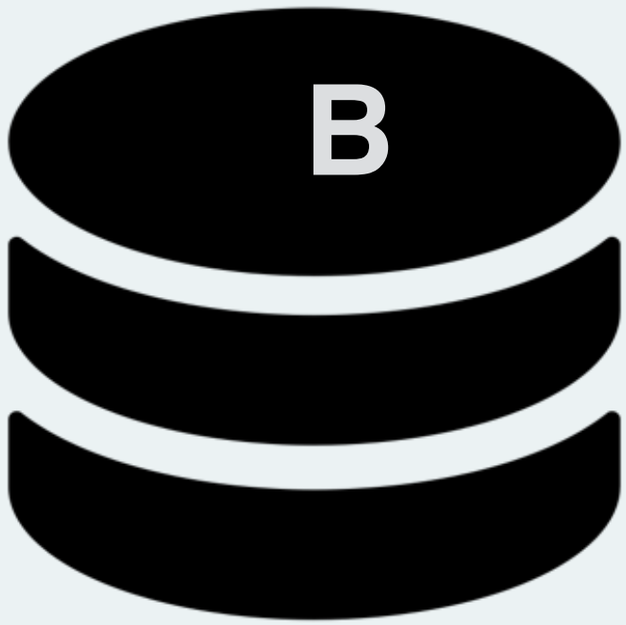
Created by Amy Schwartz
from the Noun Project



Created by Creative Stall
from Noun Project

PUT

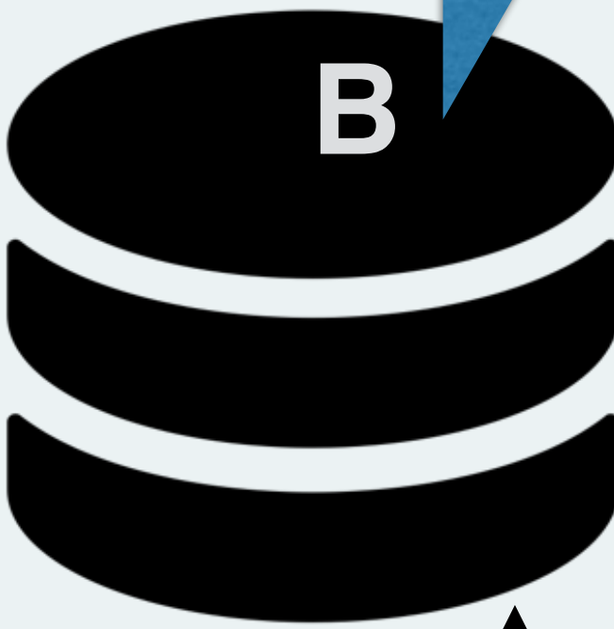
Created by Amy Schwartz
from the Noun Project



Created by Creative Stall
from Noun Project

GET

Created by Amy Schwartz
from the Noun Project



Created by Creative Stall
from Noun Project

PUT

Created by Amy Schwartz
from the Noun Project

RYOW

- Invariant: strictly increasing events per actor.
- PW+PR > N
    - Availability cost
    - Bug made it impossible!

Client VClock

- Read not_found []
- store “bob” [{c, 1}]
- read “bob” [{c, 1}]
- store [“bob”, “sue”] [{c, 2}]

Client VClock

- Read not_found []
- store “bob” [{c, 1}]
- read not_found []
- store “sue” [{c, 1}]

Riak Vnode

FALSE

local descends incoming?

TRUE

FALSE (concurrent)

TRUE

merge

ncoming

ignore incoming

write merged

n Vectors

Client VClock

- If local clock: ([{c, 1}])
  descends
  incoming clock: ([{c,1}])
- discard incoming value

Client Side ID
RYOW

- Read a Stale clock
- Re-issue the same OR lower event again
- No total order for a single actor
- Each event is not unique
- System discards as “seen” data that is new

Client Side IDs
Bad

- Unique actor ID:: database invariant enforced by client!
- Actor Explosion (Charron-Bost)
    - No. Entries == No. Actors
- Client Burden
- RYOW required - Availability Cost

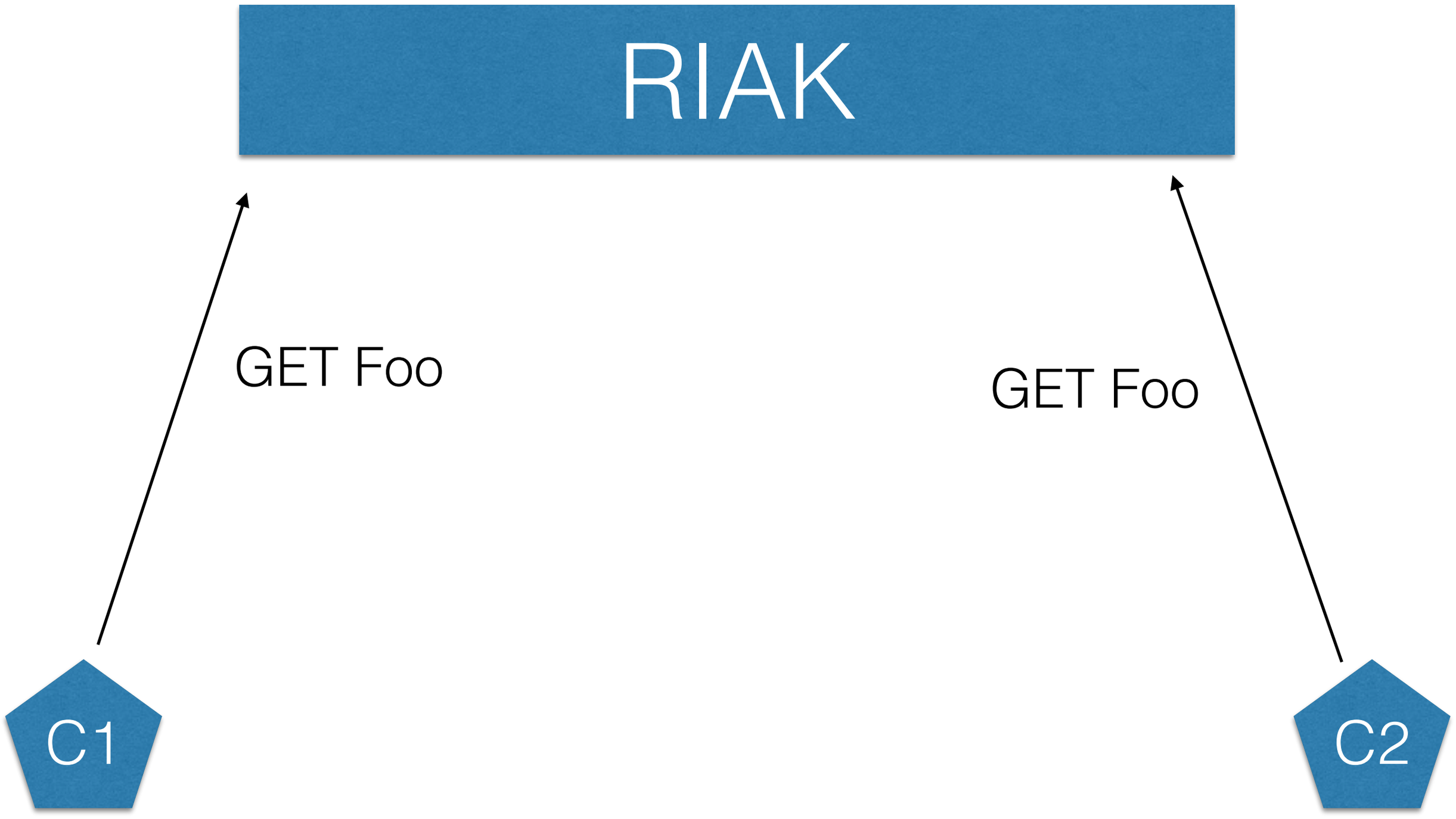
Riak Version Vectors

Who’s the actor?

Vnode Version Vectors
Riak 1.n

- No more Version Vector, just say Context
- The Vnode is the Actor
  - Vnodes act serially
  - Store the clock with the Key
- Coordinating Vnode, increments clock
- Deliberate false concurrency

Vnode VClocks
False Concurrency



Vnode VClocks
False Concurrency

RIAK

[{a,1},{b4}]->”bob”

[{a,1},{b4}]->”bob”

C1

C2

Vnode VClocks
False Concurrency

RIAK

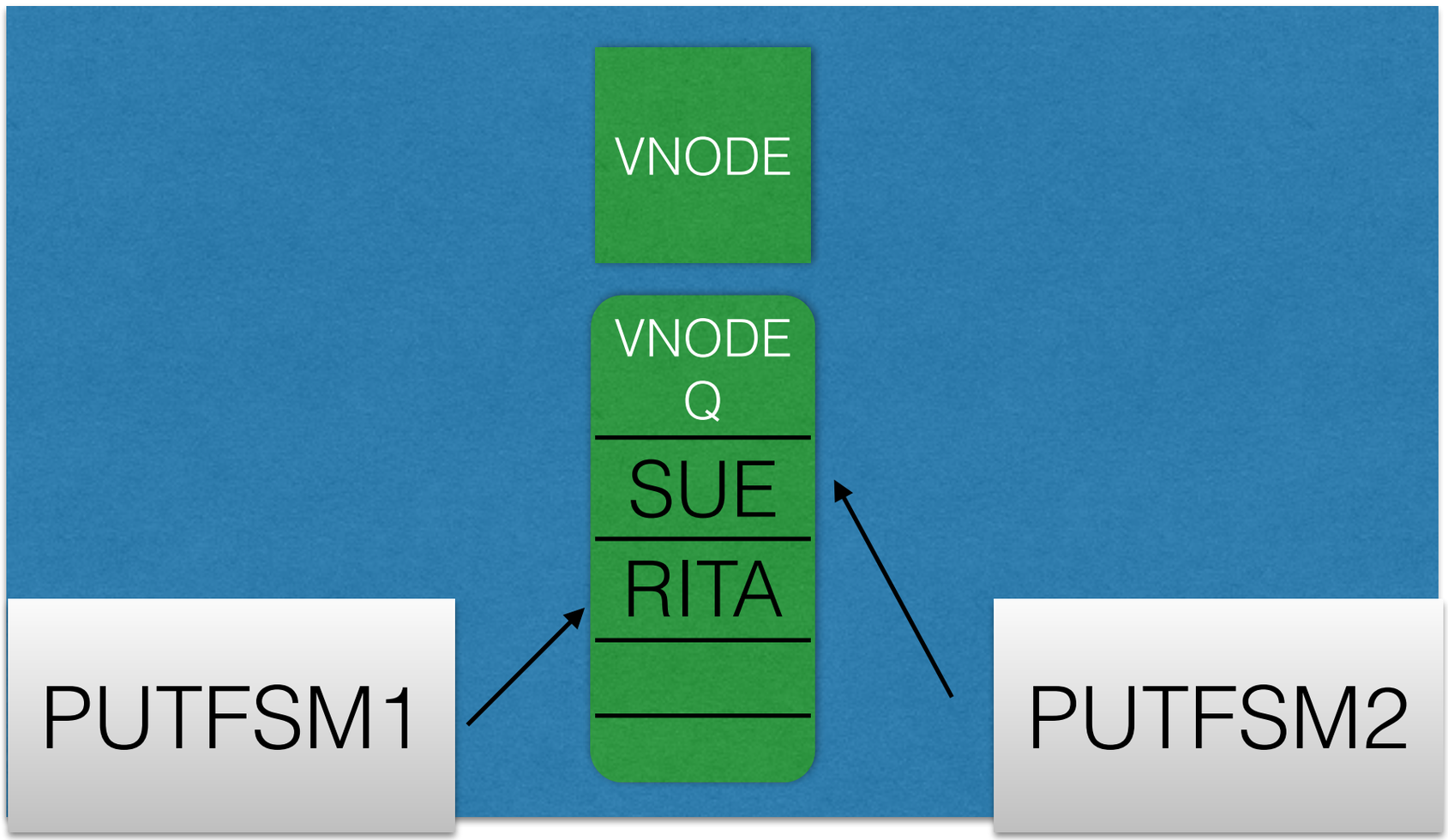
PUT [{a,1},{b,4}]="Sue"

PUT [{a,1},{b,4}]="Rita"

C1

C2

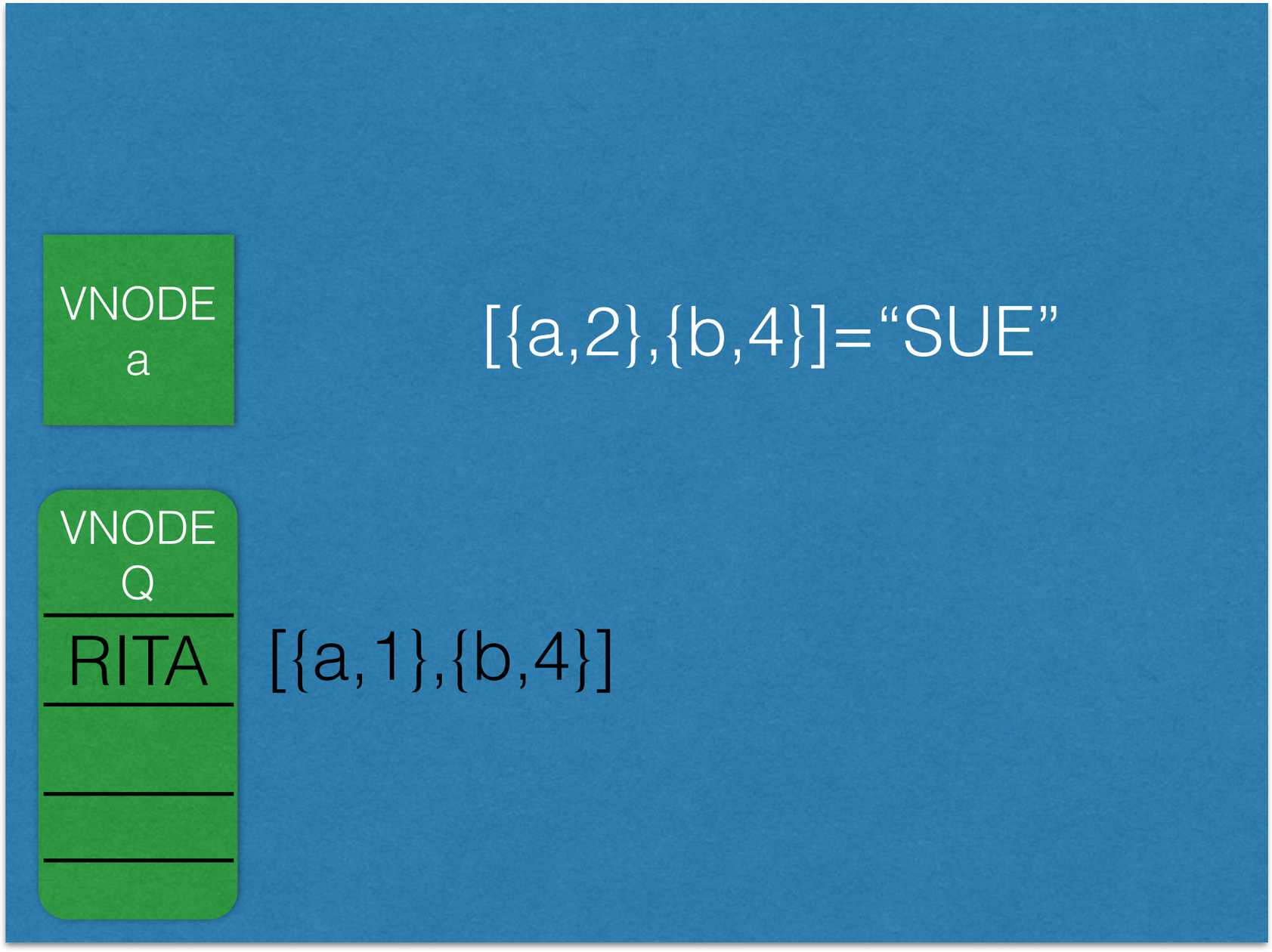
Vnode VClocks
False Concurrency



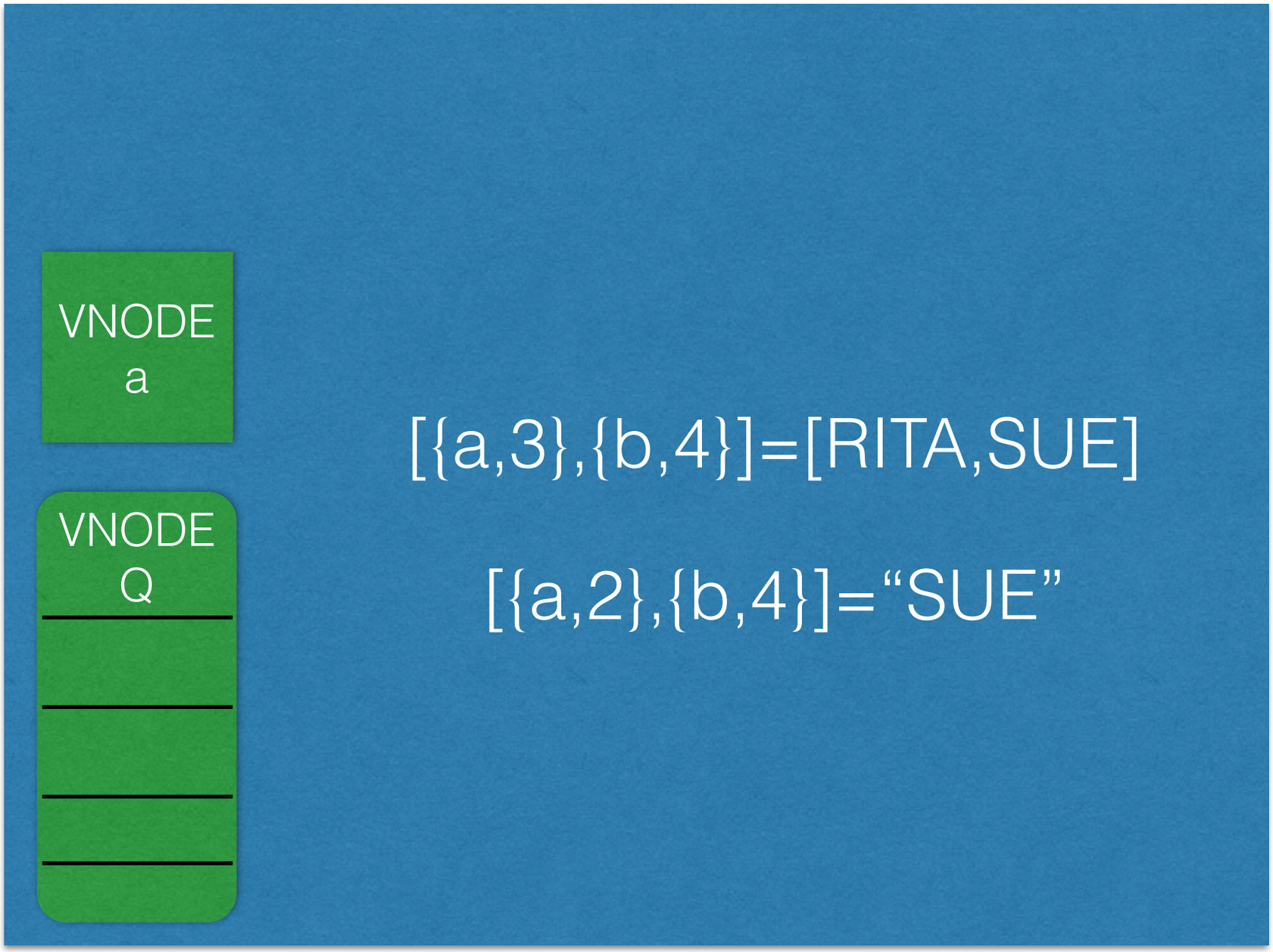
C1

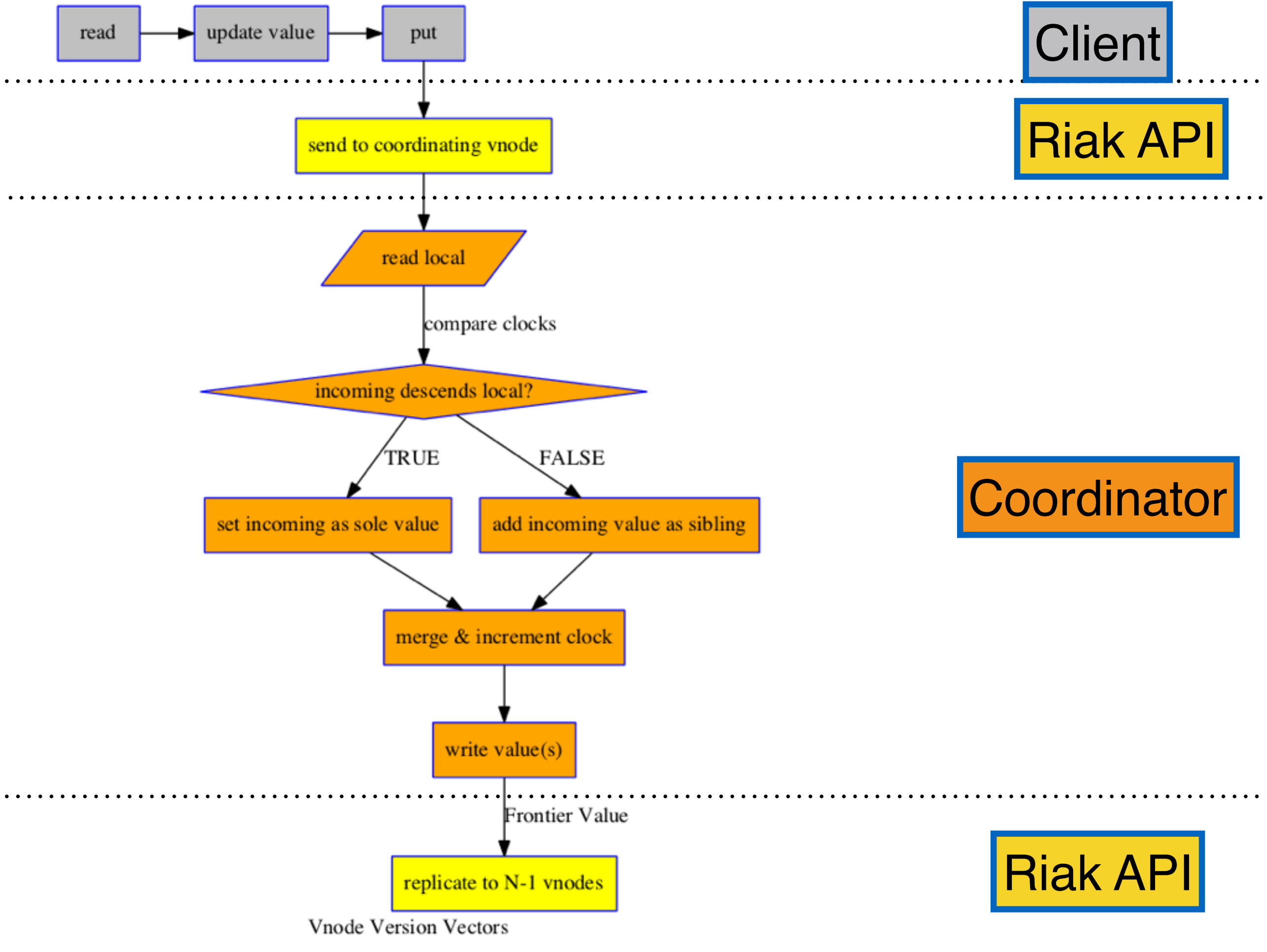
C2

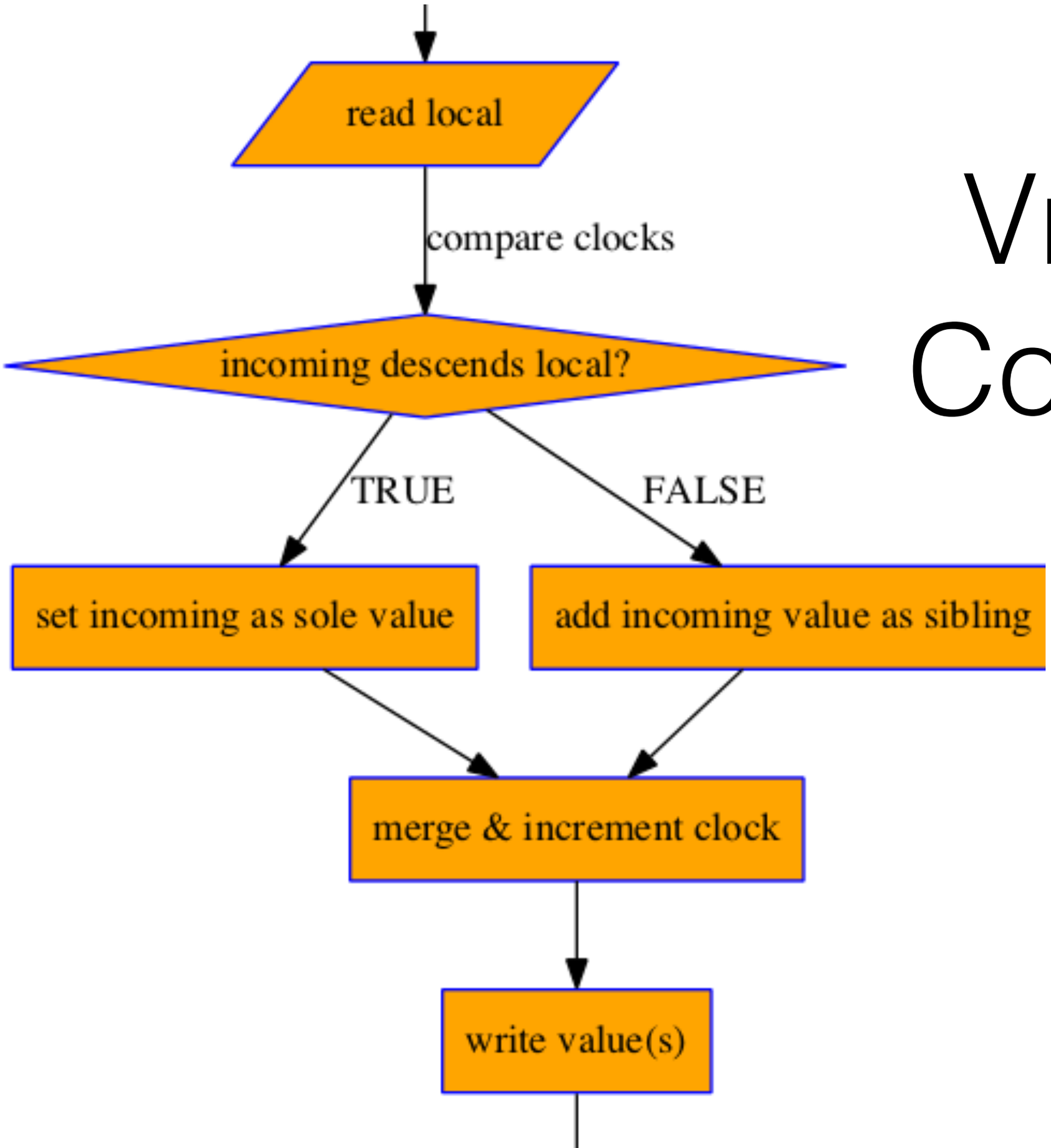
Vnode VClocks
False Concurrency



Vnode VClocks
False Concurrency




read
update value
put
send to coordinating vnode
read local
compare clocks
incoming descends local?
TRUE
FALSE
set incoming as sole value
add incoming value as sibling
merge & increment clock
write value(s)
Frontier Value
replicate to N-1 vnodes
Vnode Version Vectors
Client
Riak API
Coordinator
Riak API



Vnode VV Coordinator

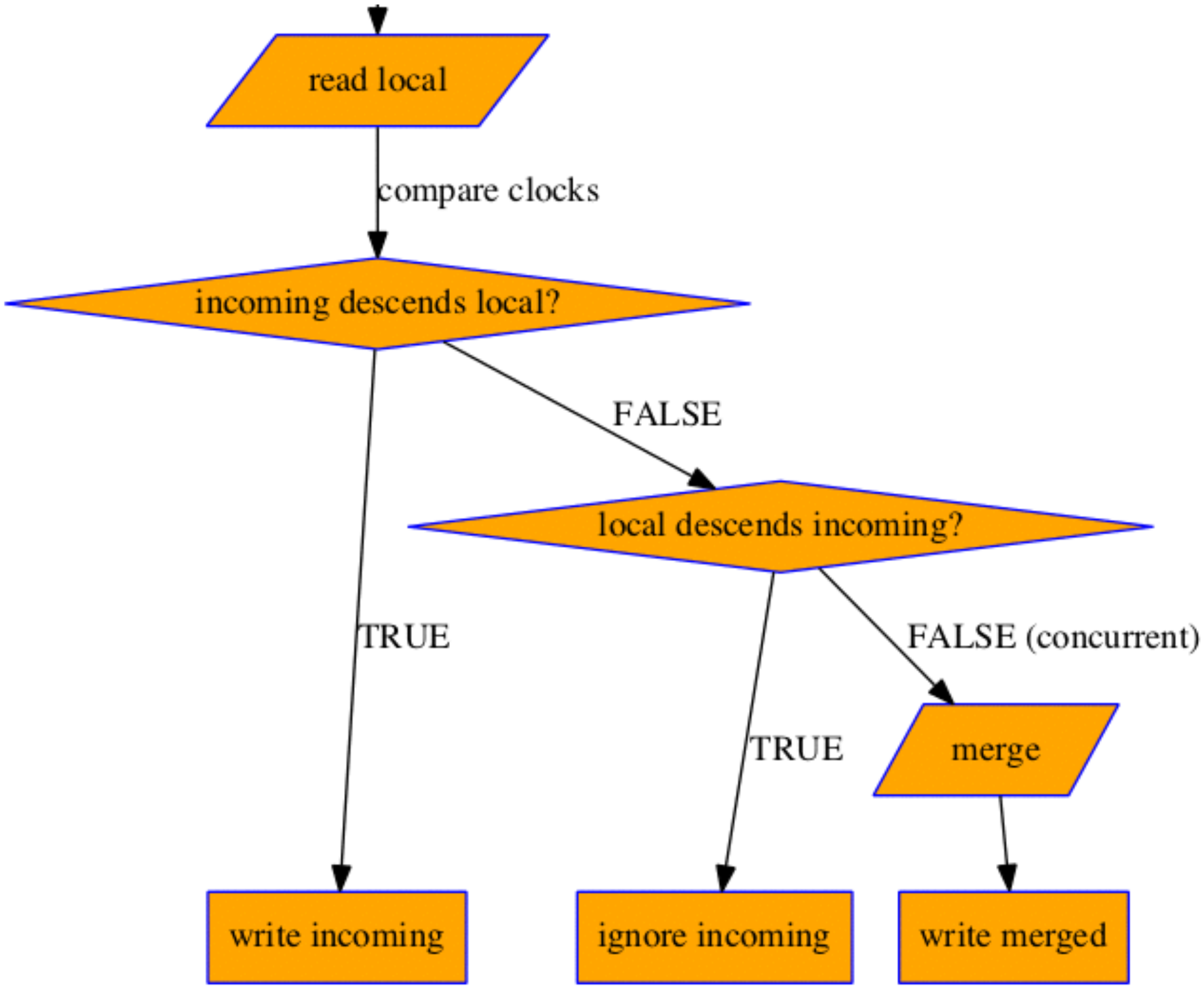
Vnode VV - Coordinator

- If incoming clock descends local
  - Increment clock
  - Write incoming as sole value
  - Replicate

Vnode VV - Coordinator

- If incoming clock does not descend local
    - Merge clocks
    - Increment Clock
    - Add incoming value as sibling
    - Replicate

Vnode VV - Replica



Vnode VClock
GOOD

- Far fewer actors
- Way simpler
- Empty context PUTs are siblings

Vnode VClock
BAD

- Possible latency cost of forward
- No more idempotent PUTs
    - Store a SET of siblings, not LIST
- Sibling Explosion
    - As a result of too much false concurrency

Sibling Explosion

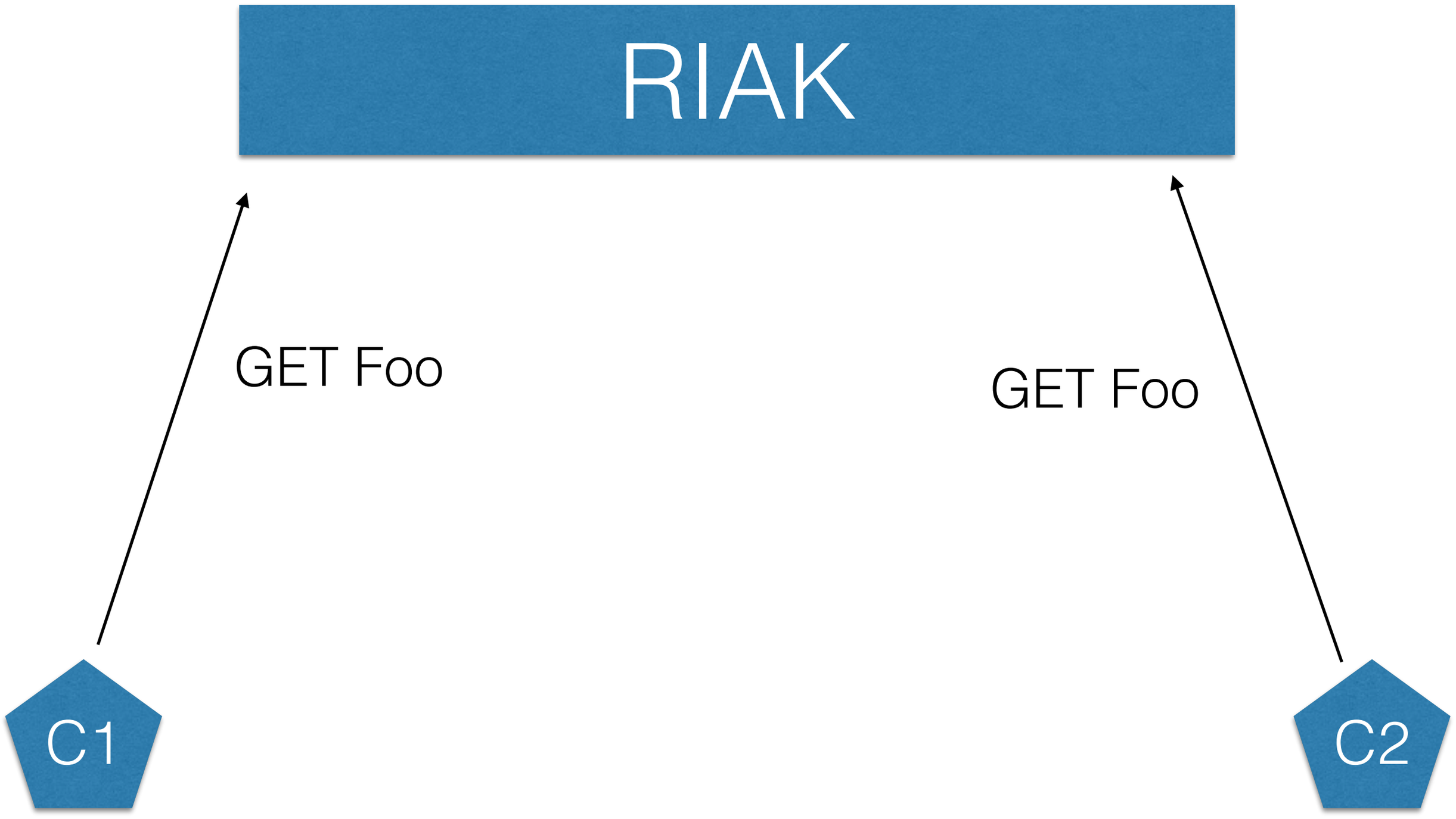
- False concurrency cost
- Many many siblings
- Large object
- Death

Sibling Explosion

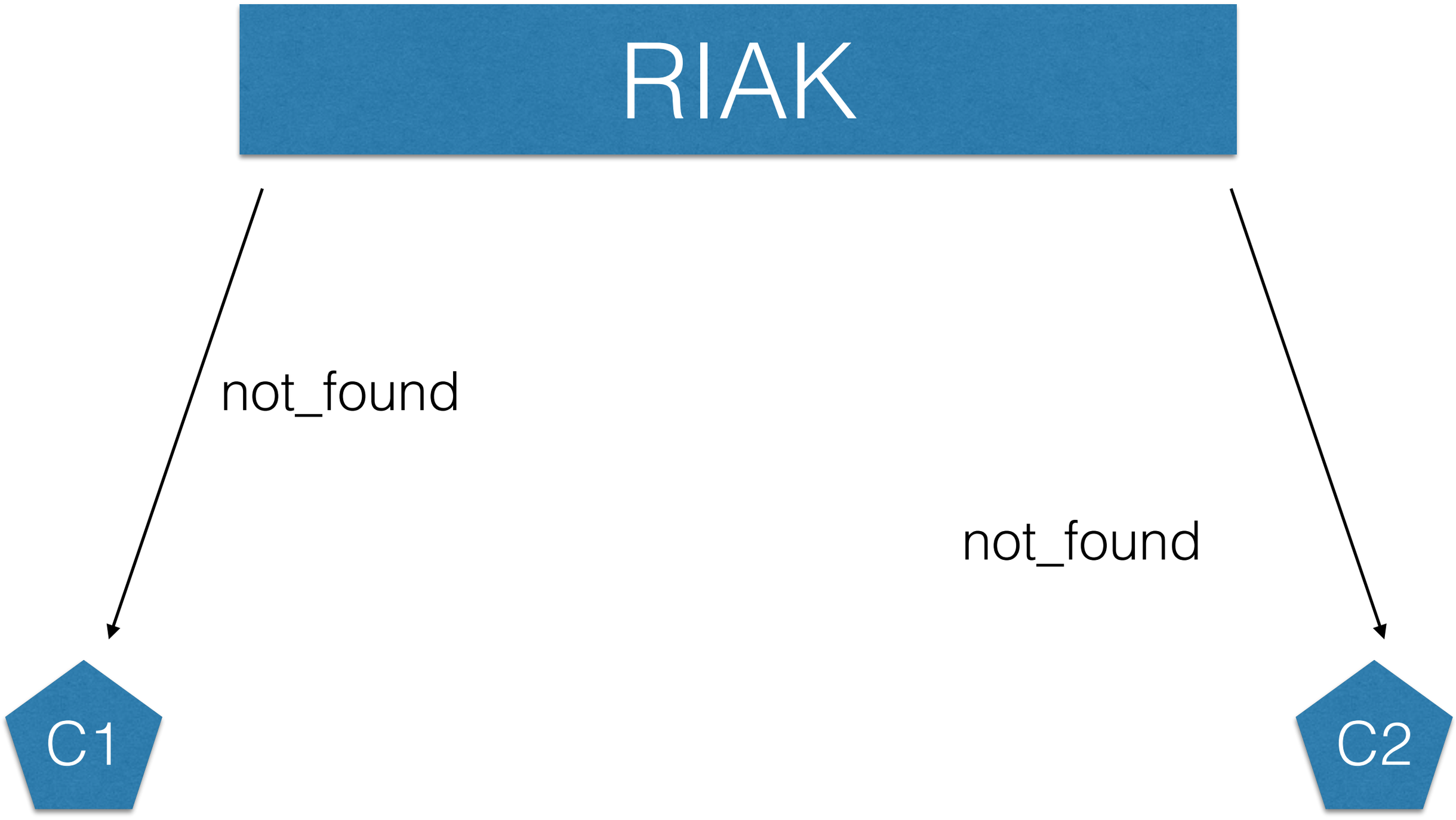
- Data structure
  - Clock + Set of Values
- False Concurrency

Sibling Explosion

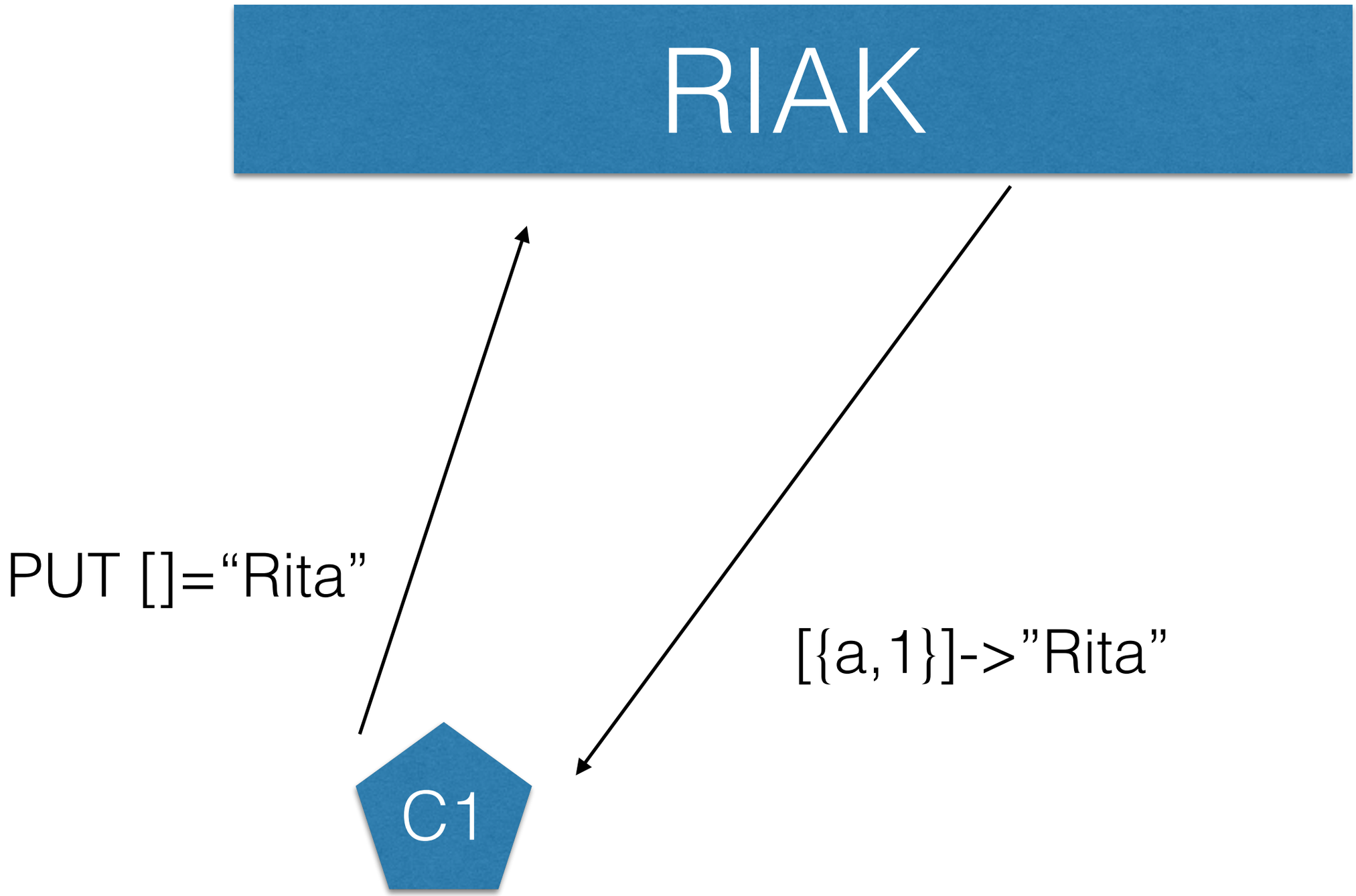
Sibling Explosion



Sibling Explosion



Sibling Explosion



Sibling Explosion

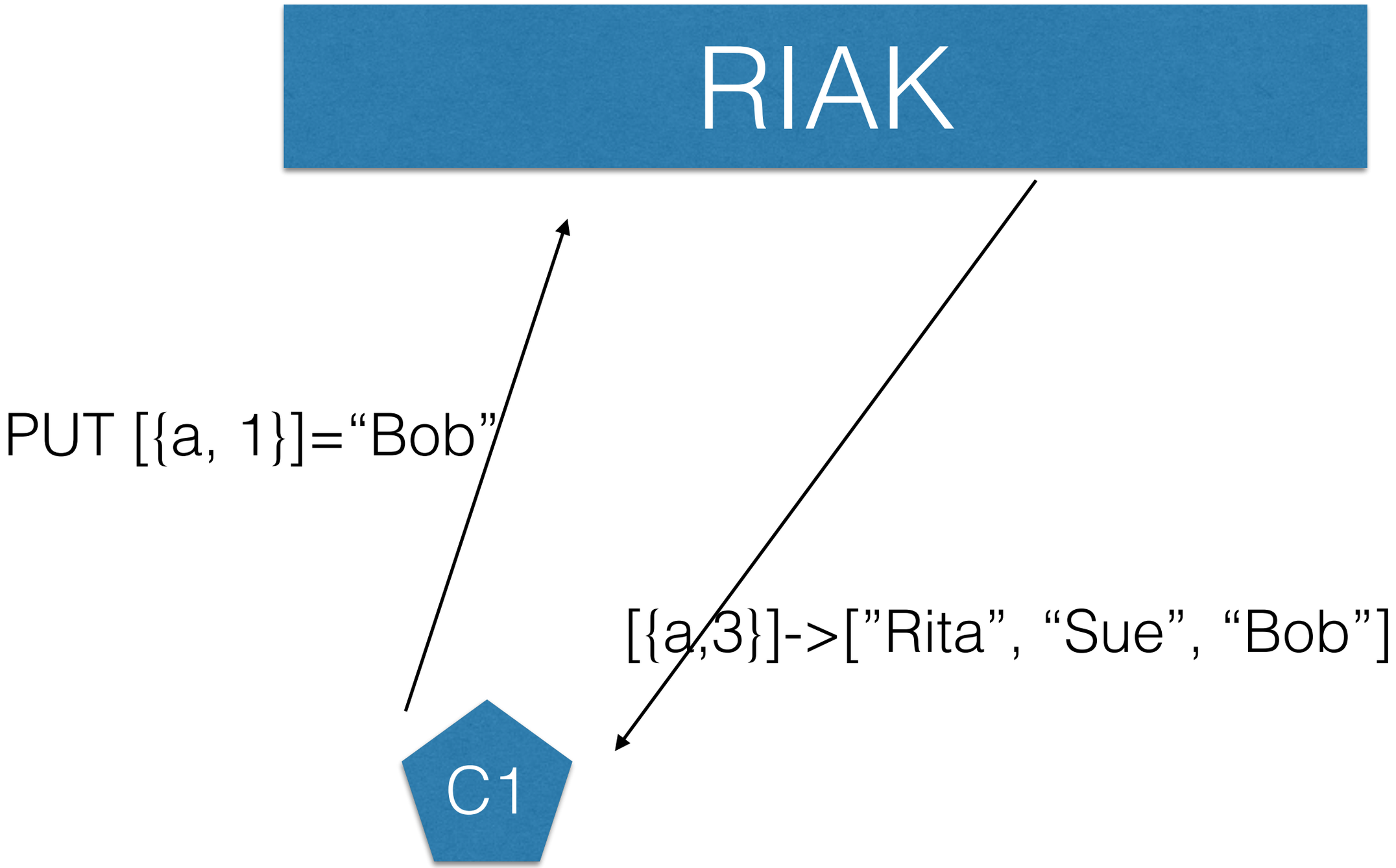
RIAK

PUT []="Sue"

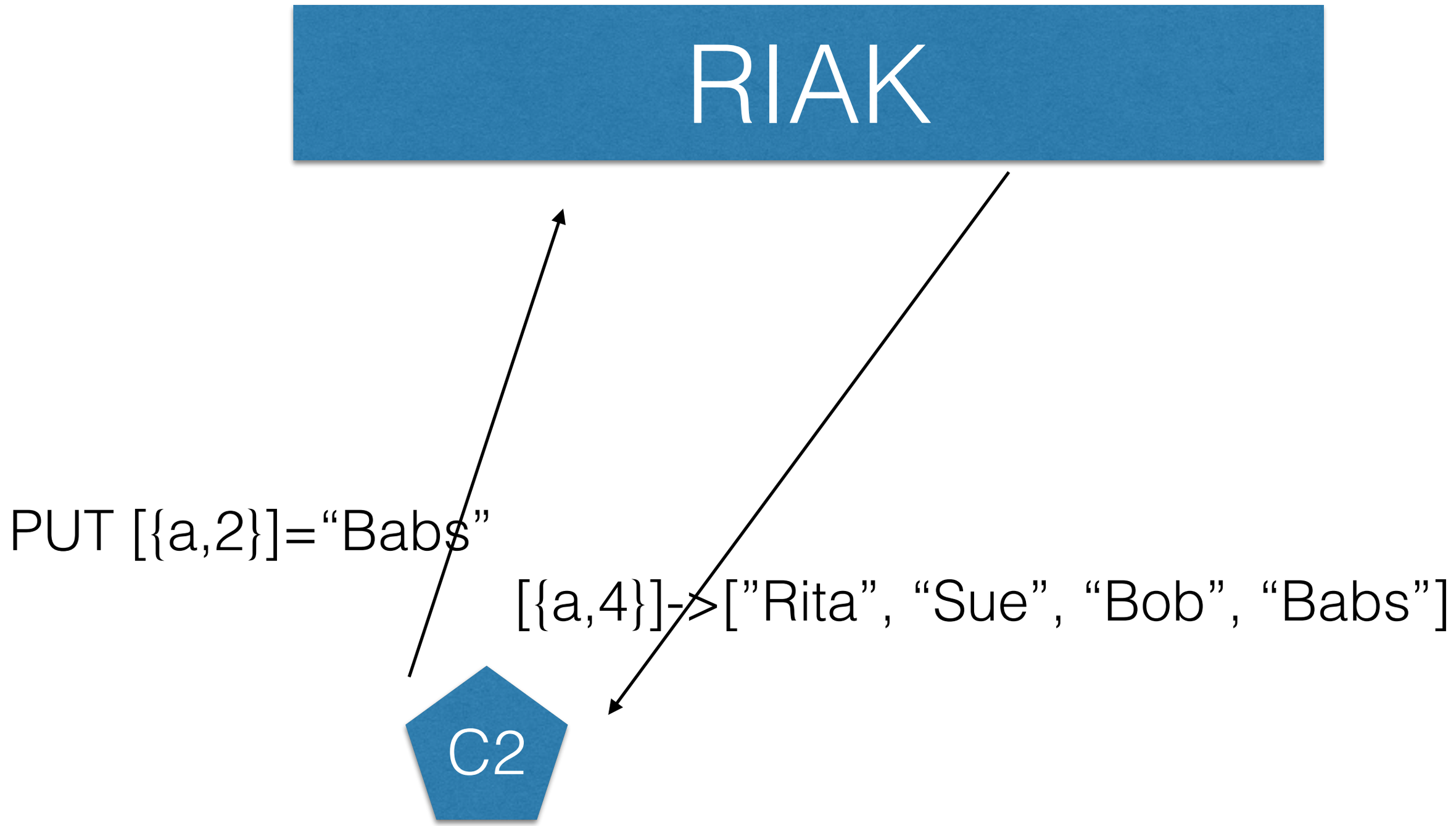
[{a,2}]->["Rita", "Sue"]

C2

Sibling Explosion



Sibling Explosion



Vnode VClock

- Trick to “dodge’ the Charron-Bost result
- Engineering, not academic
- Tested (quickchecked in fact!)
- “Action at a distance”

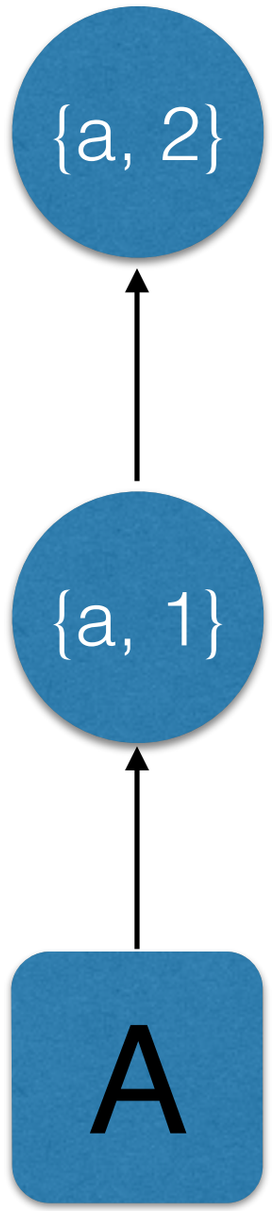
Dotted Version Vectors

Dotted Version Vectors: Logical Clocks for Optimistic Replication

http://arxiv.org/abs/1011.5808

Vnode VClocks + Dots
Riak 2.n

- What even is a dot?
  - That “event” we saw back a the start



Oh Dot all the Clocks

✳ Data structure

- Clock + List of Dotted Values

[{{a, 1}, “bob”}, {{a, 2}, “Sue”}]

Vnode VClock

✳ If incoming clock descends local

- Increment clock
- Get Last Event as dot (eg {a, 3})
- Write incoming as sole value + Dot
- Replicate

Vnode VClock

✳ If incoming clock does not descend local

- Merge clocks
- Increment Clock
- Get Last Event as dot (eg {a, 3})
- Prune siblings!
- Add incoming value as sibling
- Replicate

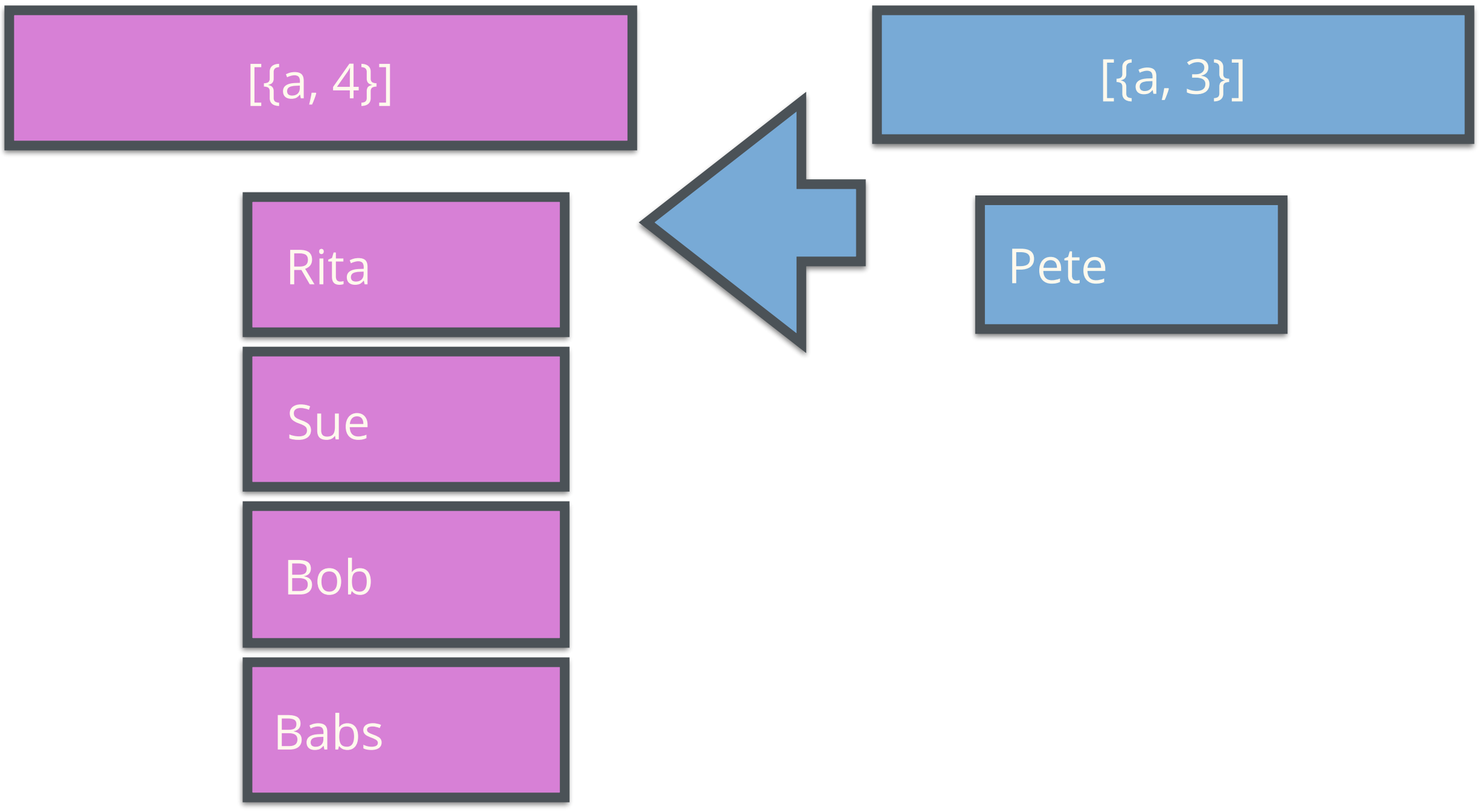
Oh drop all the dots

✳ Prune Siblings

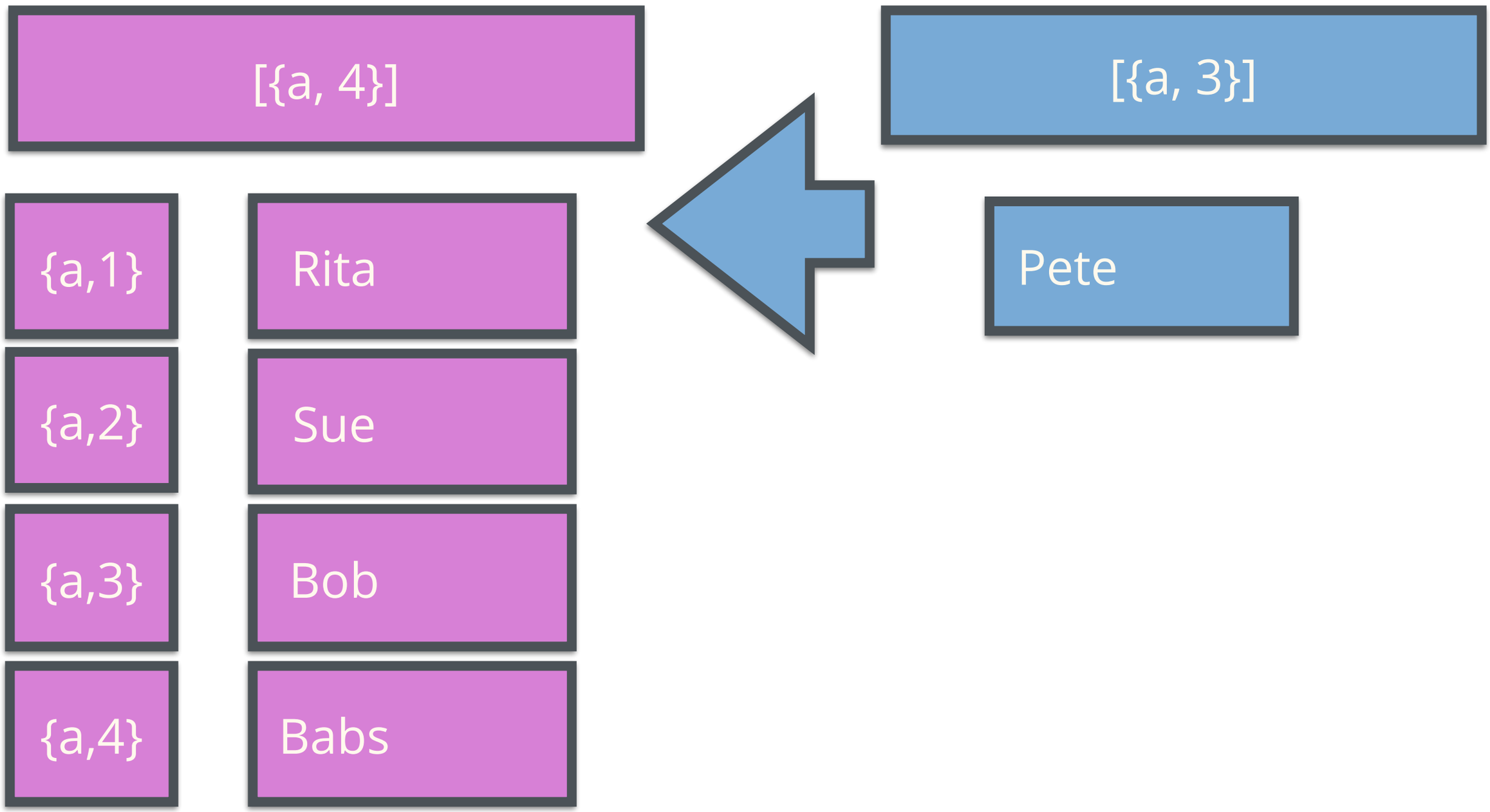
- Remove any siblings who's dot is seen by the incoming clock

- if Clock >= [Dot] drop Dotted value

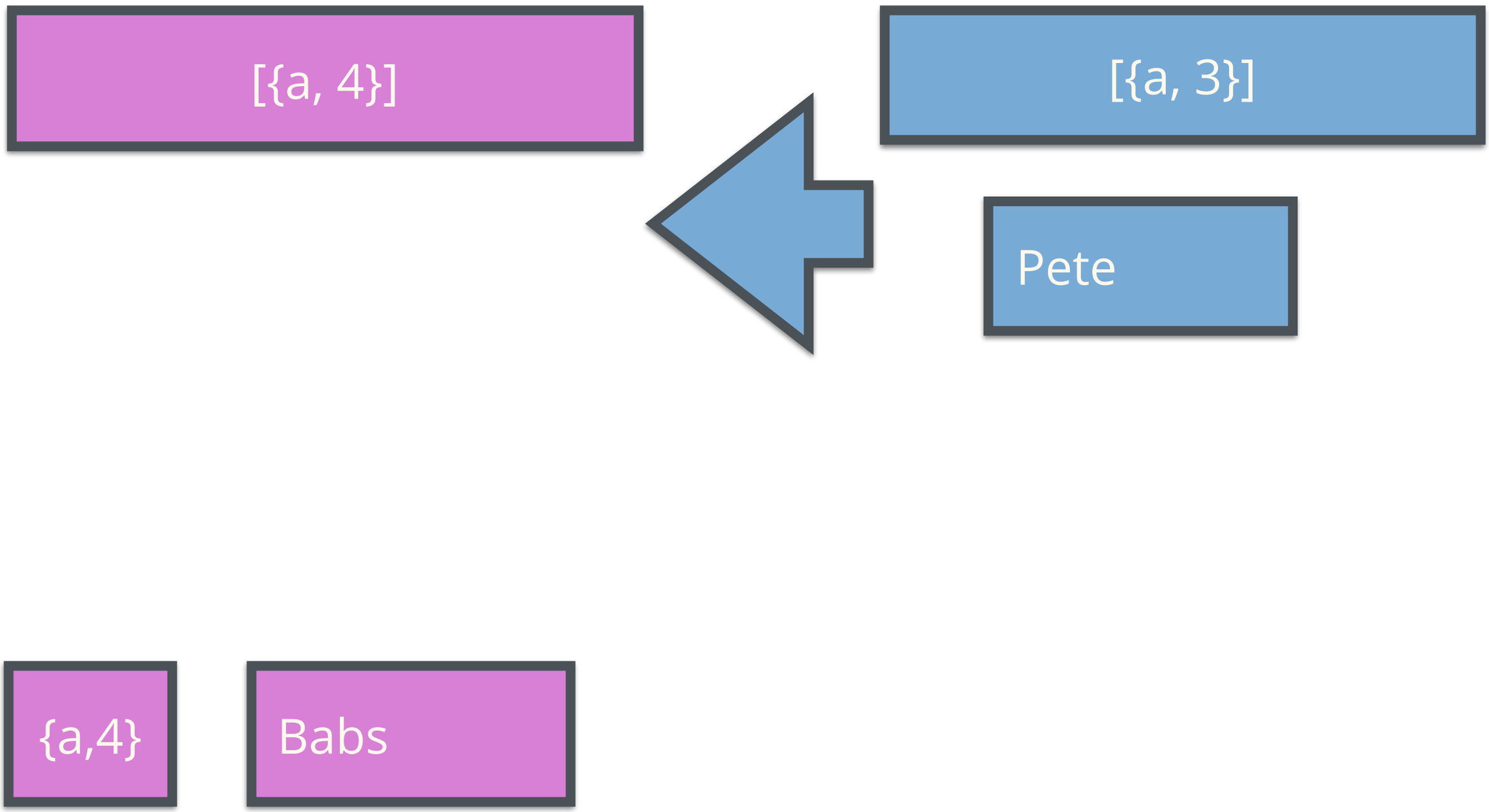
Vnode VClocks



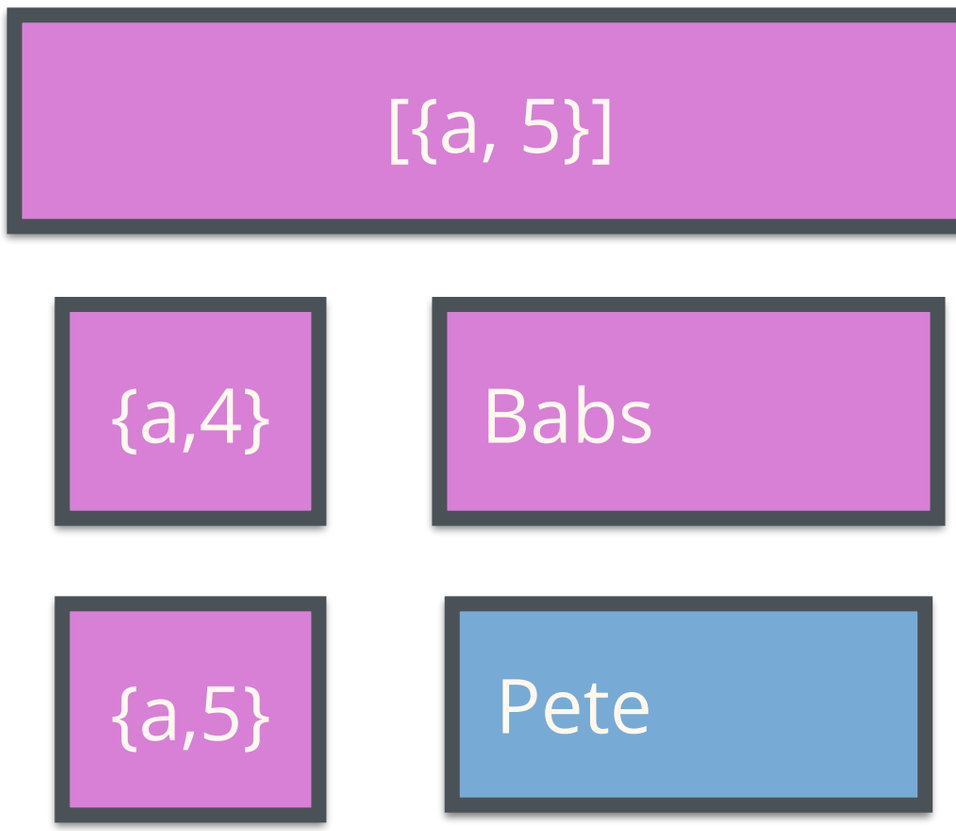
Vnode VClocks
+ Dots



Vnode VClocks
+ Dots



Vnode VClocks
+ Dots



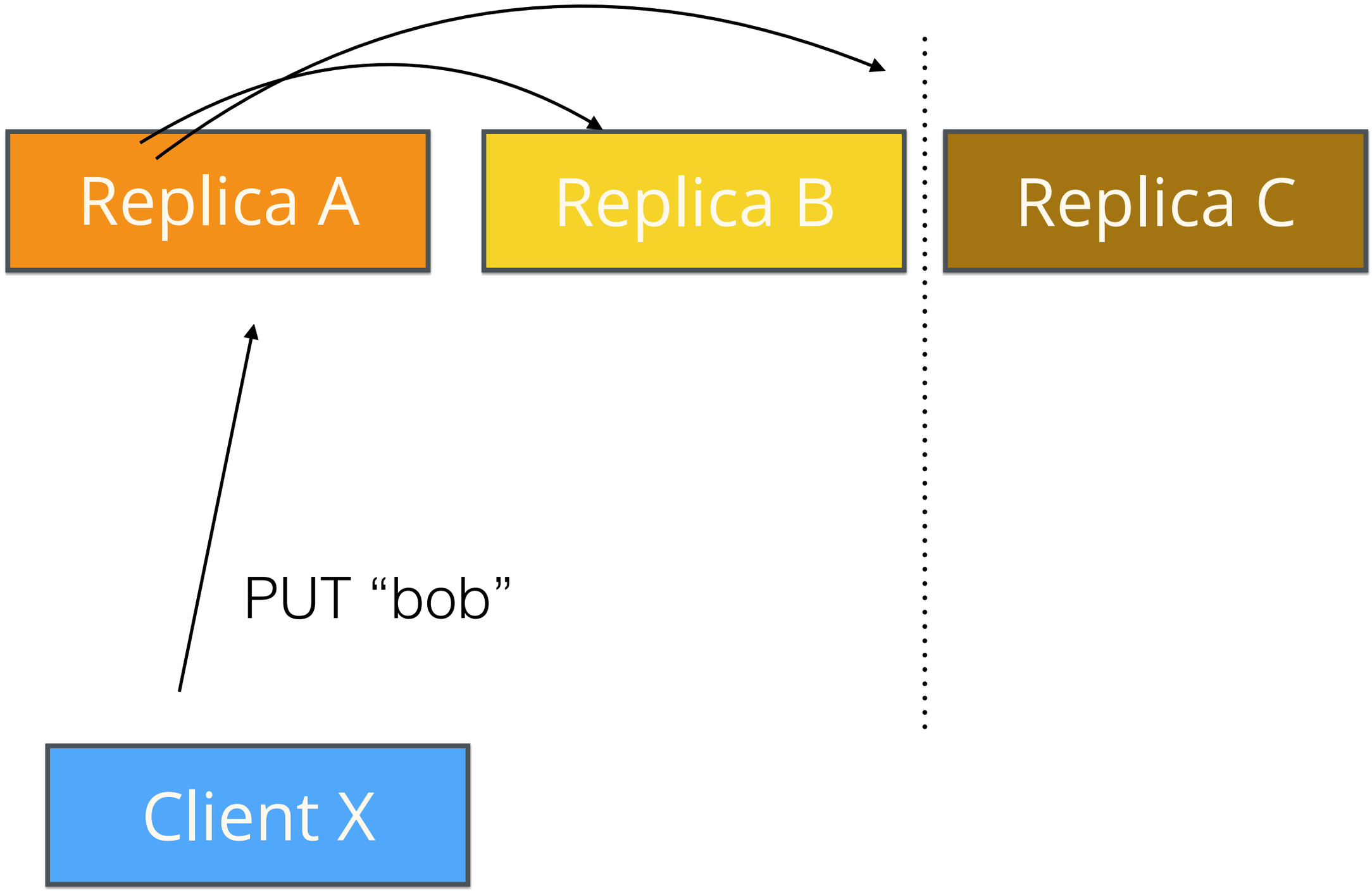
Dotted Version Vectors

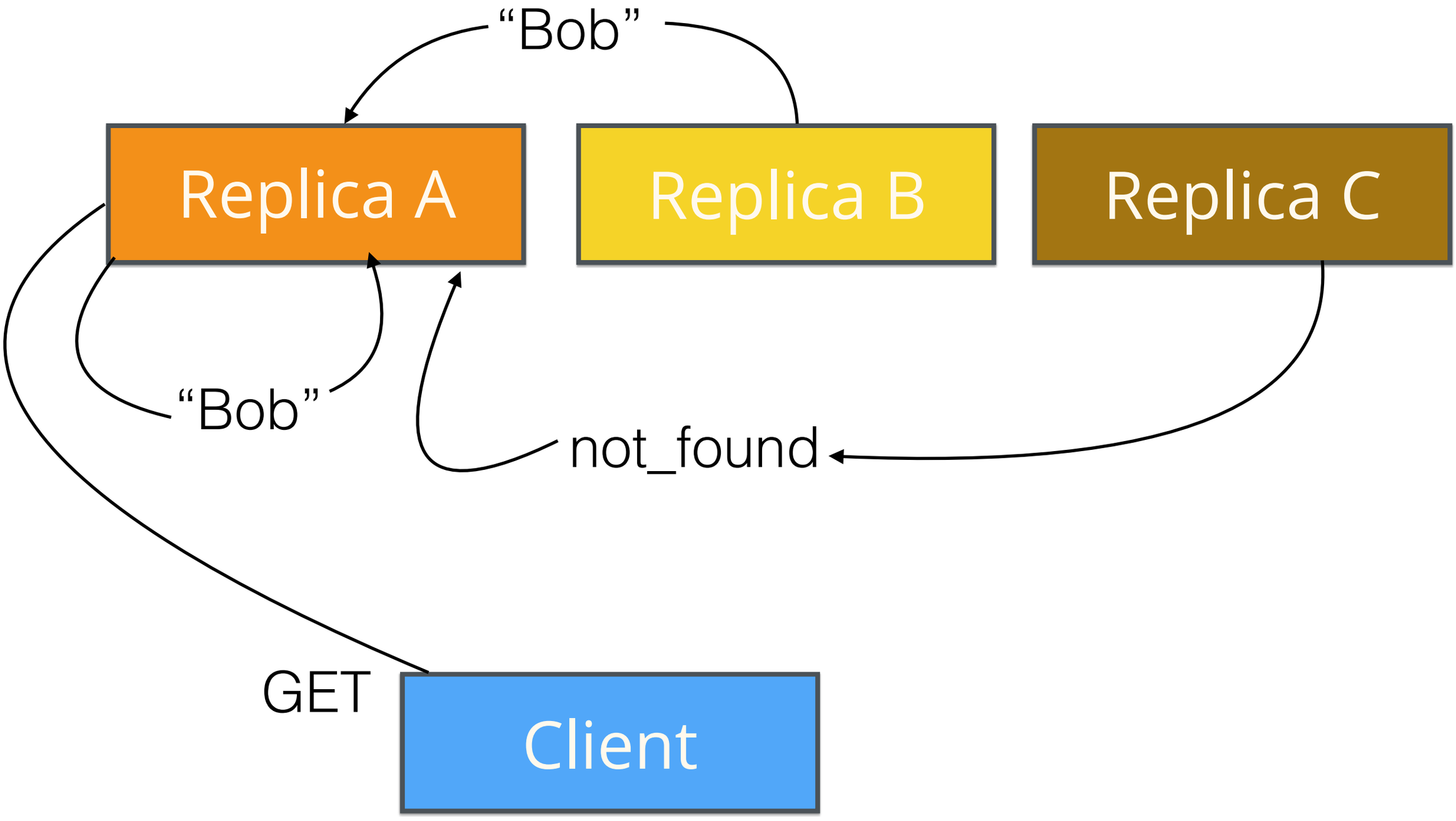
- Action at a distance
- Correctly capture concurrency
- No sibling explosion
- No Actor explosion

KV679

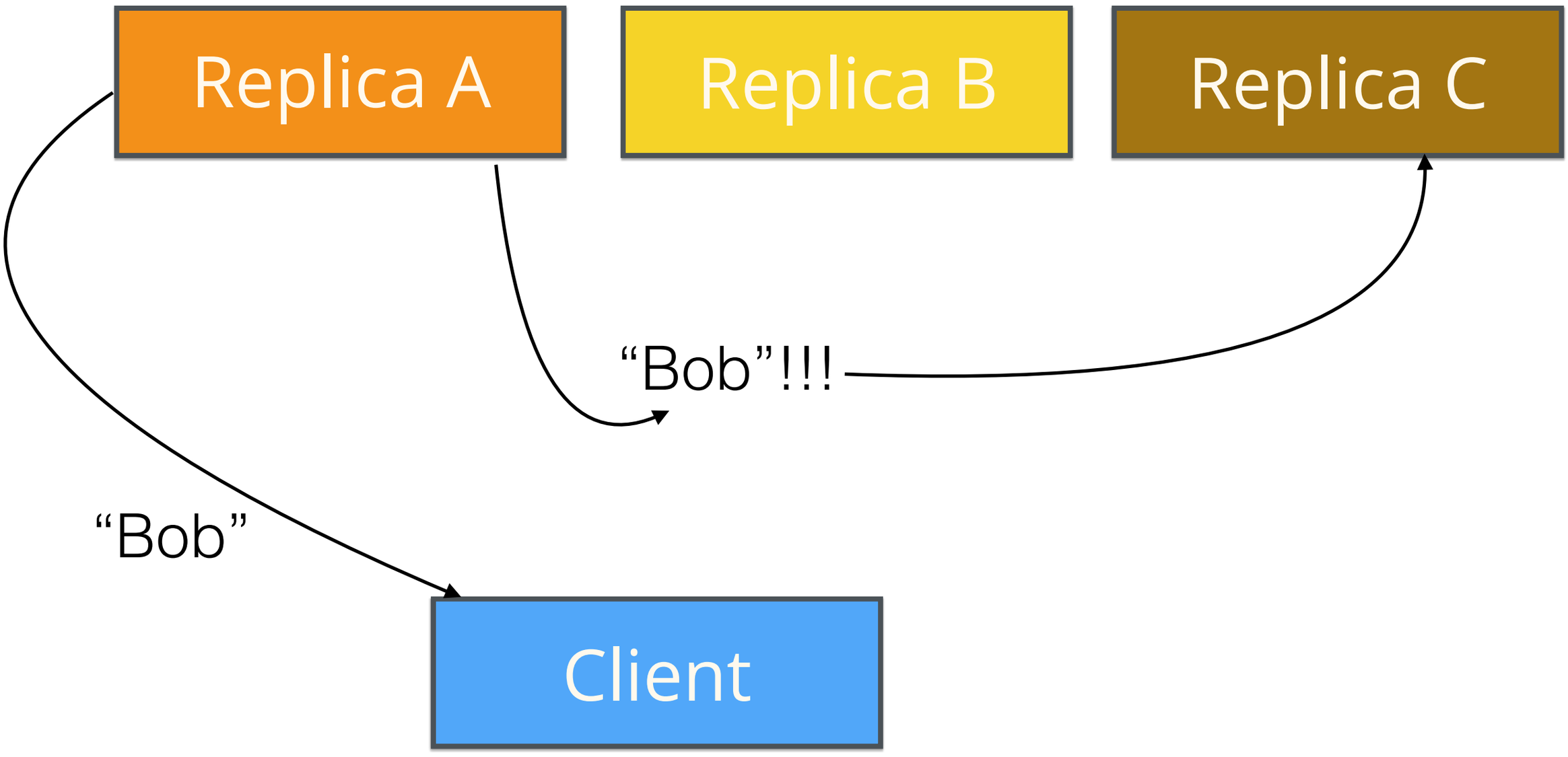
Riak Overview

Read Repair. Deletes.

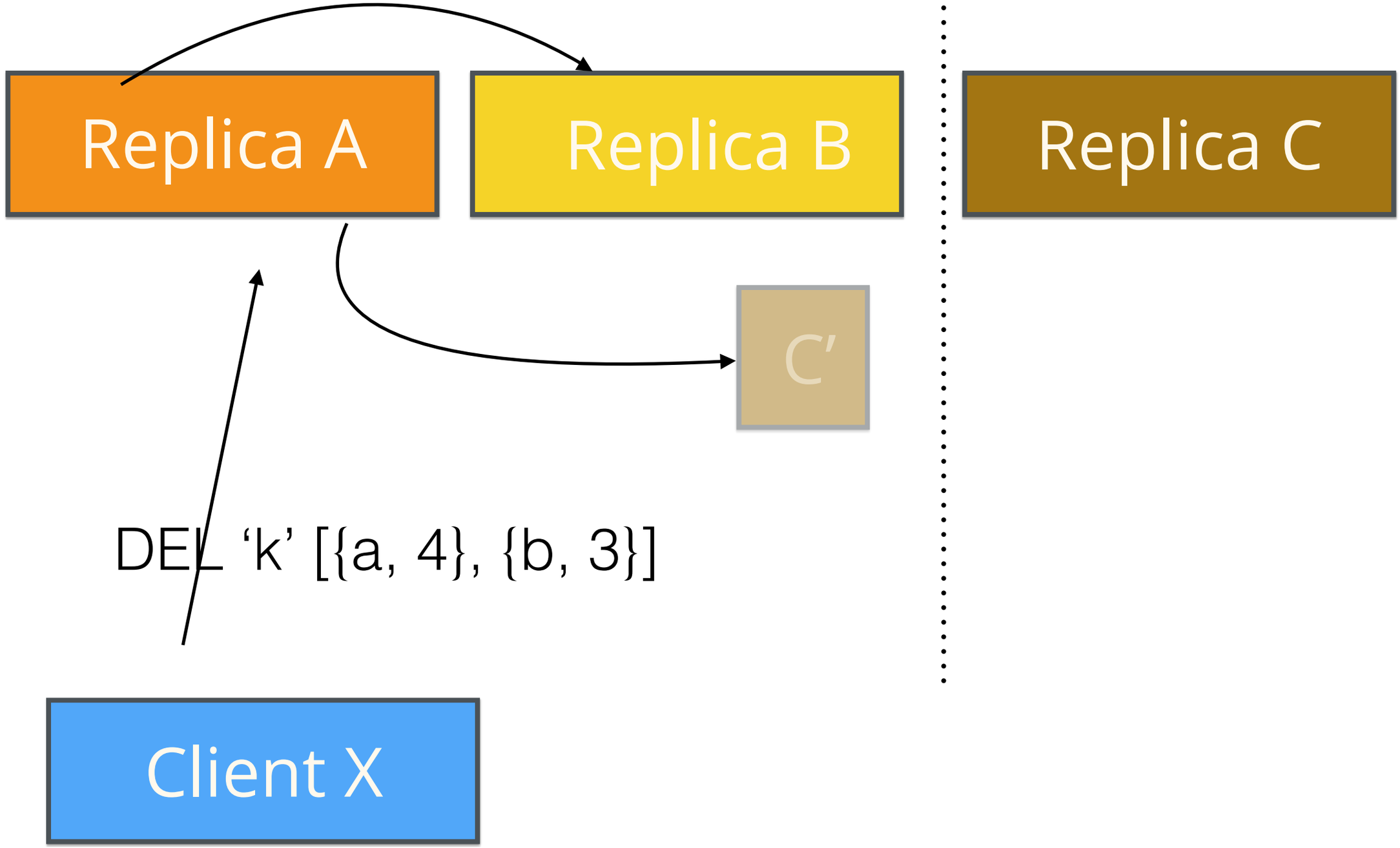

Replica A
Replica B
Replica C
PUT “bob”
Client X



Read Repair



Read Repair


Replica A
Replica B
Replica C
C′
DEL ‘k’ [{a, 4}, {b, 3}]
Client X

Replica A

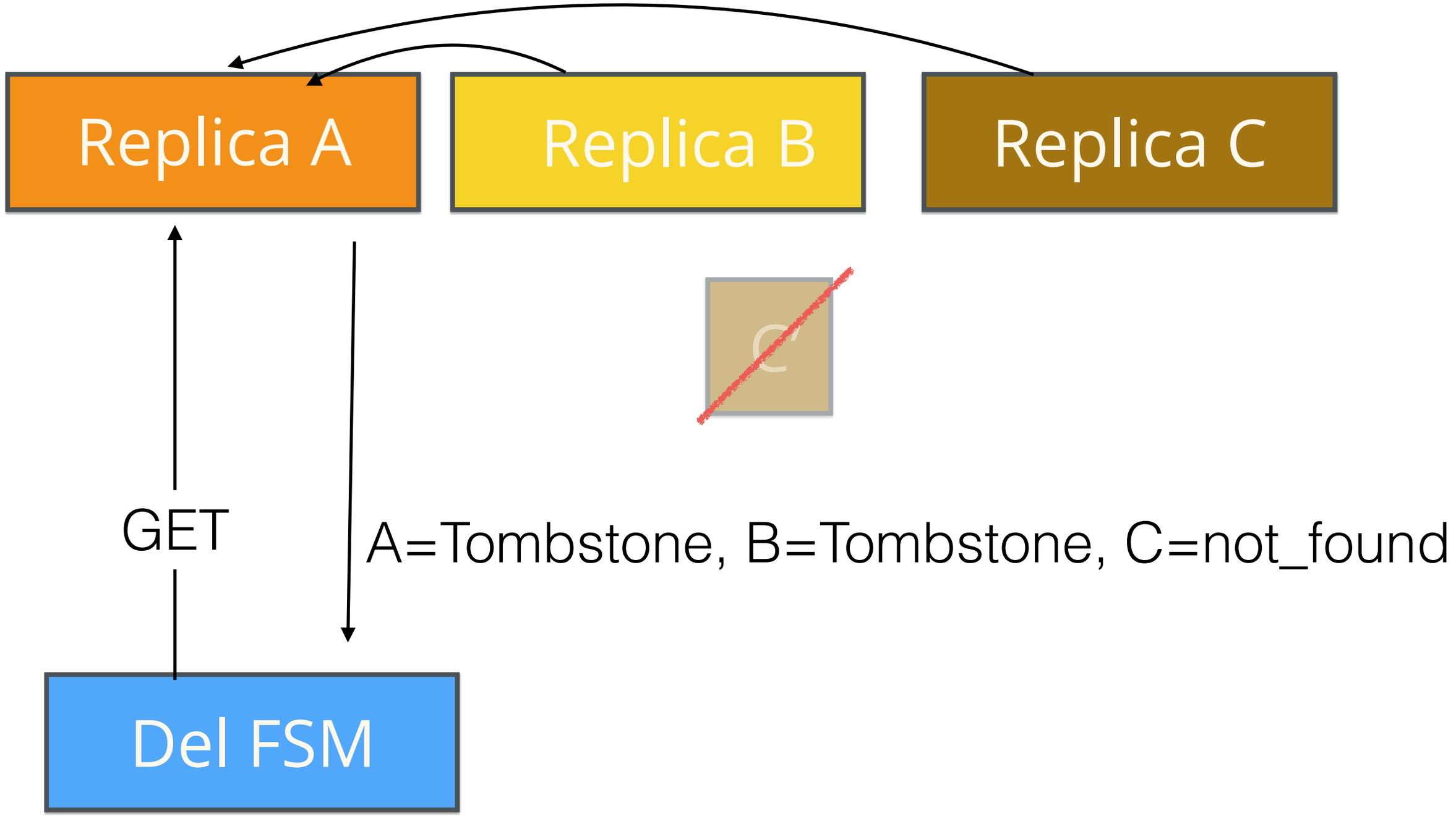
Replica B

Replica C

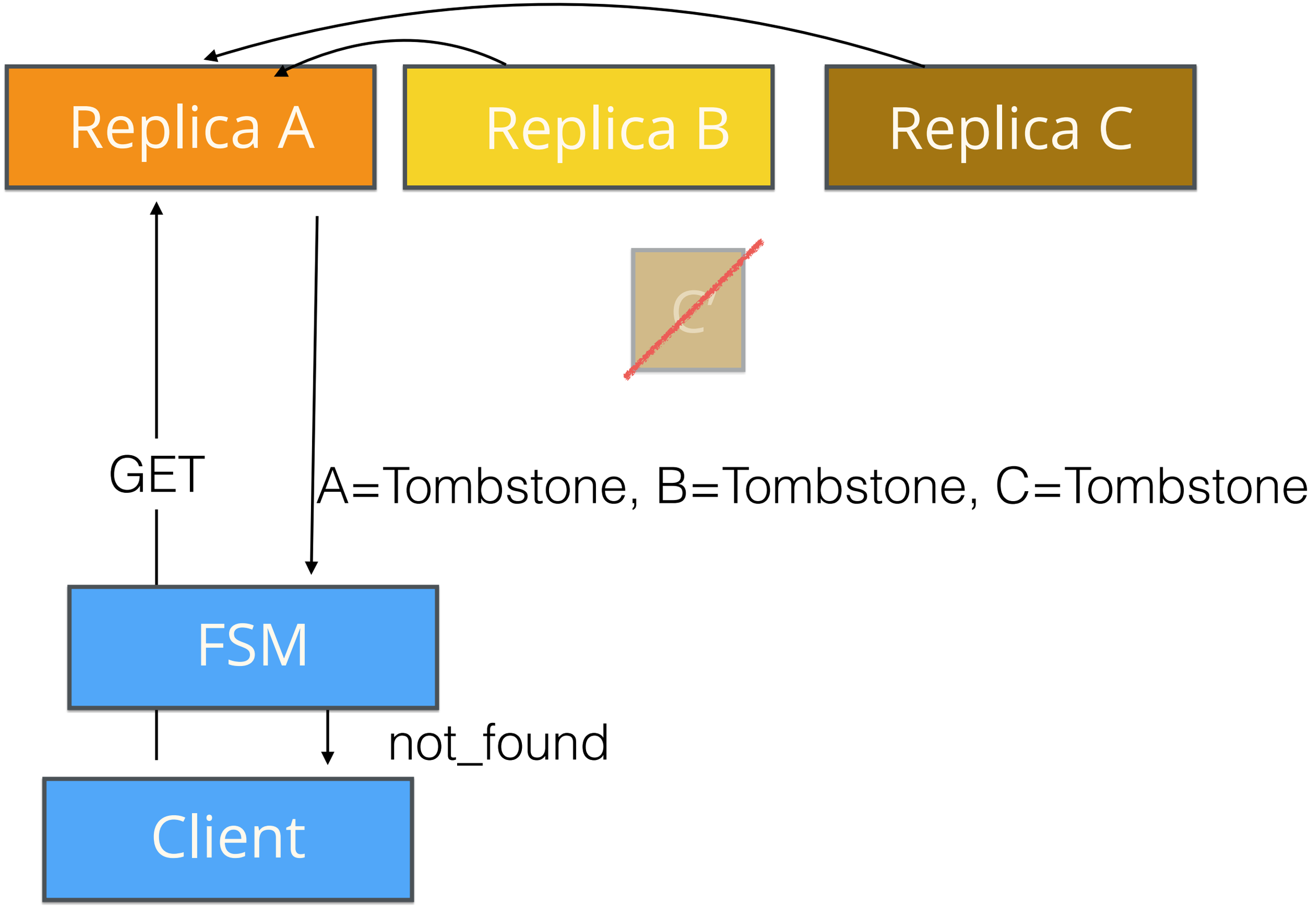
C'

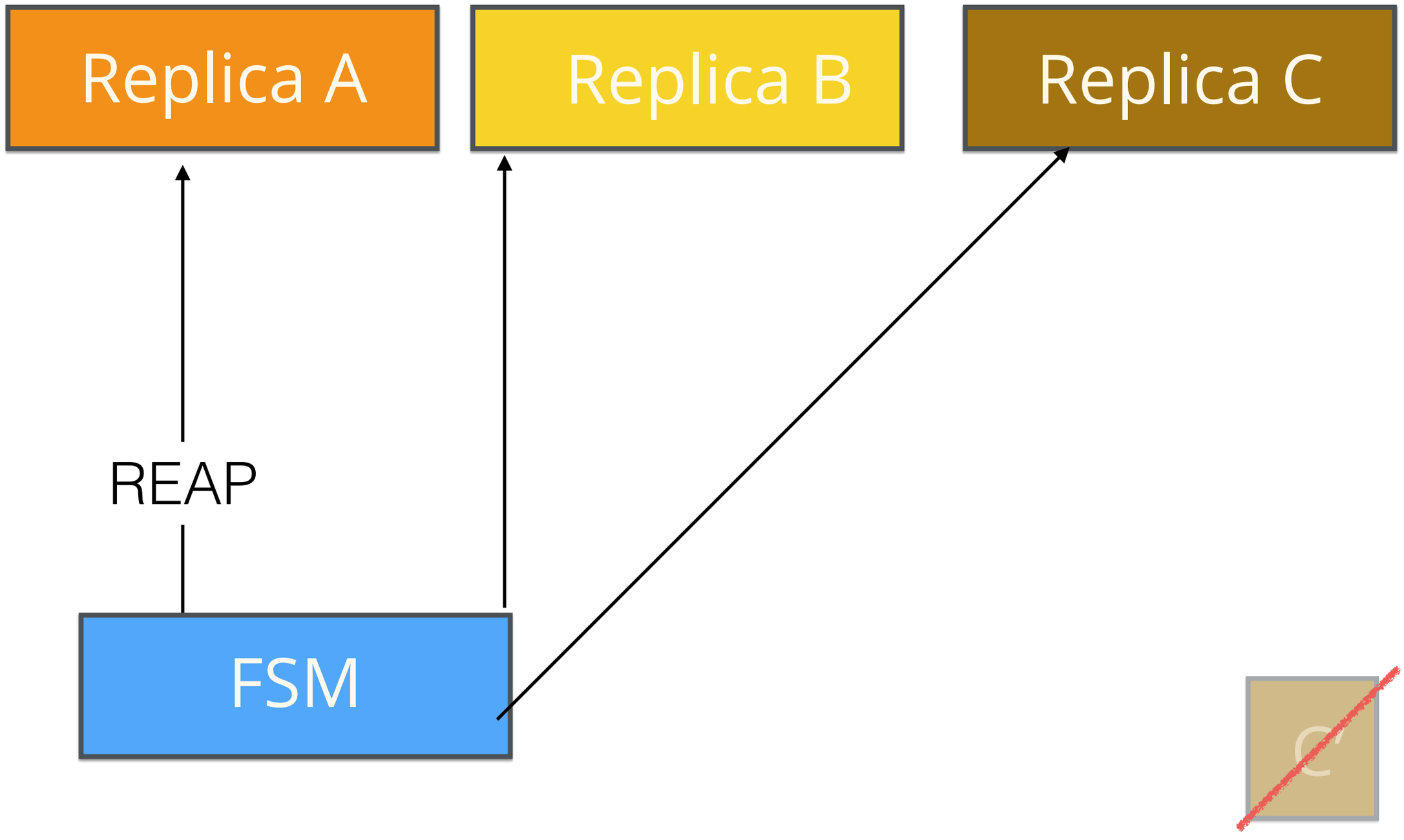
GET

Del FSM


Replica A
Replica B
Replica C
C'
GET
A=Tombstone, B=Tombstone, C=not_found
Del FSM

Read Repair


Replica A
Replica B
Replica C
C'
GET
A=Tombstone, B=Tombstone, C=Tombstone
FSM
not_found
Client


Replica A
Replica B
Replica C
REAP
FSM
C'

Sue [{a, 1}]

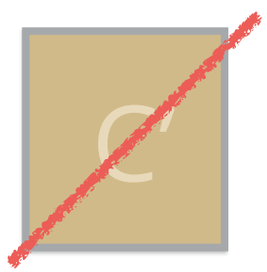
Replica A

Replica B

Replica C

PUT “sue” []

Client X



Replica A

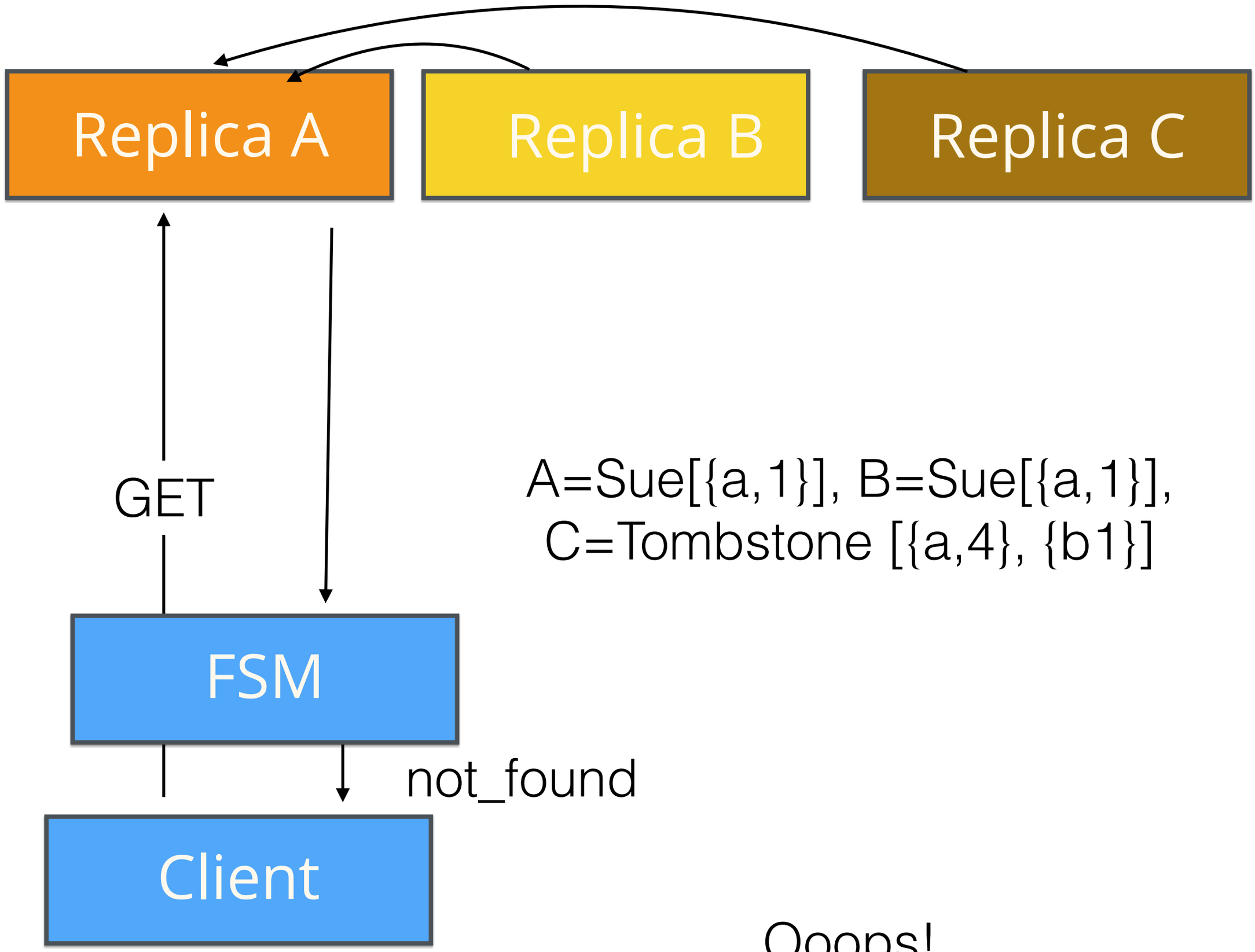
Replica B

Replica C

Hinted Hand off

tombstone

C′


Replica A
Replica B
Replica C
GET
FSM
not_found
Client
A=Sue[{a,1}], B=Sue[{a,1}],
C=Tombstone [{a,4}, {b1}]
Ooops!

KV679
Lingering Tombstone

- Write Tombstone
    - One goes to fallback
    - Read and reap primaries
    - Add Key again
    - Tombstone is handed off
    - Tombstone clock dominates, data lost

KV679
Other flavours

- Back up restore

- Failed local read (disk error, operator “error” etc)

KV679
RYOW?

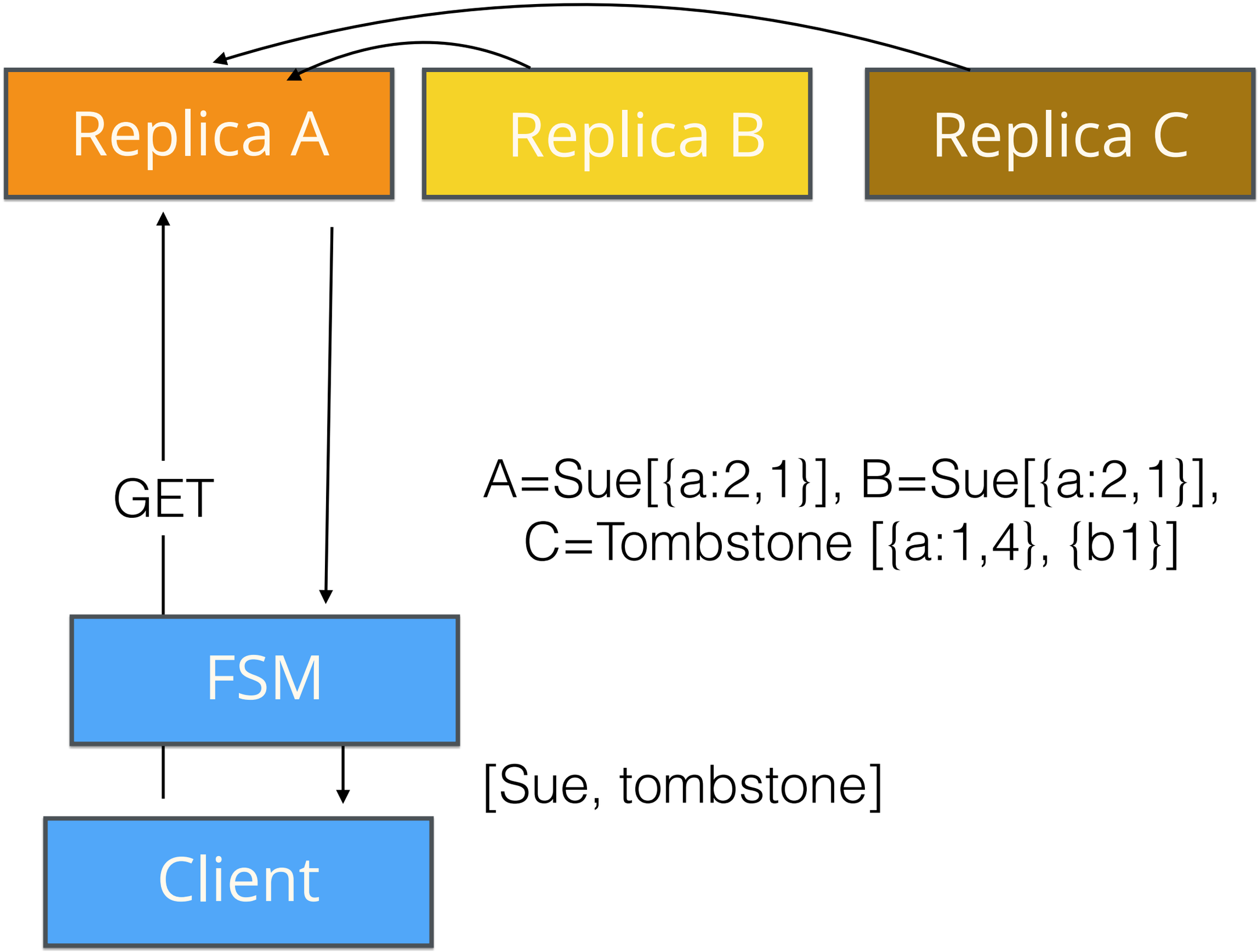
- Familiar
- History repeating

KV679
Per Key Actor Epochs

- Every time a Vnode reads a local “not_found”
  - Increment a vnode durable counter
  - Make a new actor ID
    - <<VnodeId, Epoch_Counter>>

KV679
Per Key Actor Epochs

- Actor ID for the vnode remains long lived
  - No actor explosion
- Each key gets a new actor per “epoch”
  - Vnode increments highest “Epoch” for it’s Id
    - <<VnodeId, Epoch>>


Replica A
Replica B
Replica C
GET
FSM
Client
A=Sue[{a:2,1}], B=Sue[{a:2,1}],
C=Tombstone [{a:1,4}, {b1}]
[Sue, tombstone]

Per Key Actor Epochs
BAD

- More Actors (every time you delete and recreate a key _it_ gets a new actor)
- More computation (find highest epoch for actor in Version Vector)

Per Key Actor Epochs GOOD

- No silent dataloss
- No actor explosion
- Fully backwards/forward compatible

Are we there yet?

?

Summary

- Client side Version Vectors
  - Invariants, availability, Charron-Bost
- Vnode Version Vectors
  - Sibling Explosion

Summary

- Dotted Version Vectors
  - “beat” Charron-Bost
- Per-Key-Actor-Epochs
  - Vnodes can “forget” safely

Summary

- Temporal Clocks can’t track causality

- Logical Clocks can

Summary

- Version Vectors are EASY!
- (systems using) Version Vectors are HARD!
- Mind the Gap!