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

TL;DR

- The Gap between theory and practice is:
  - Real
  - Deep and steep sided
- Scattered Invariants are hard to enforce

Why Riak?

Scale **Up**

$$$Big Iron
(still fails)

Scale **Out**

Commodity Servers
CDNs, App servers
Expertise

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%.

Trade Off

CAP

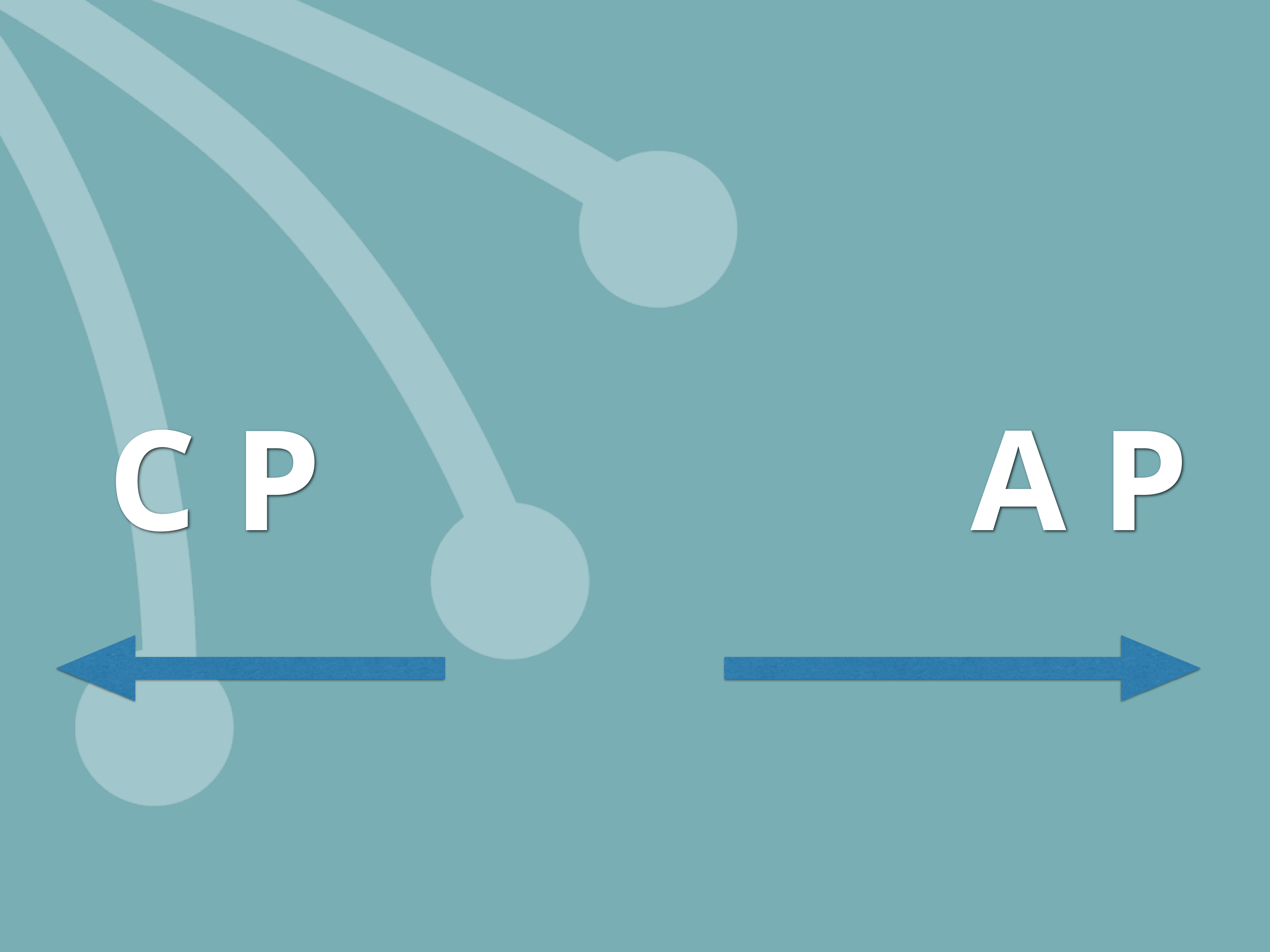
http://aphyr.com/posts/288-the-network-is-reliable

C A

C A

C

A


CP
AP

Availability

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

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

riak

Riak Overview

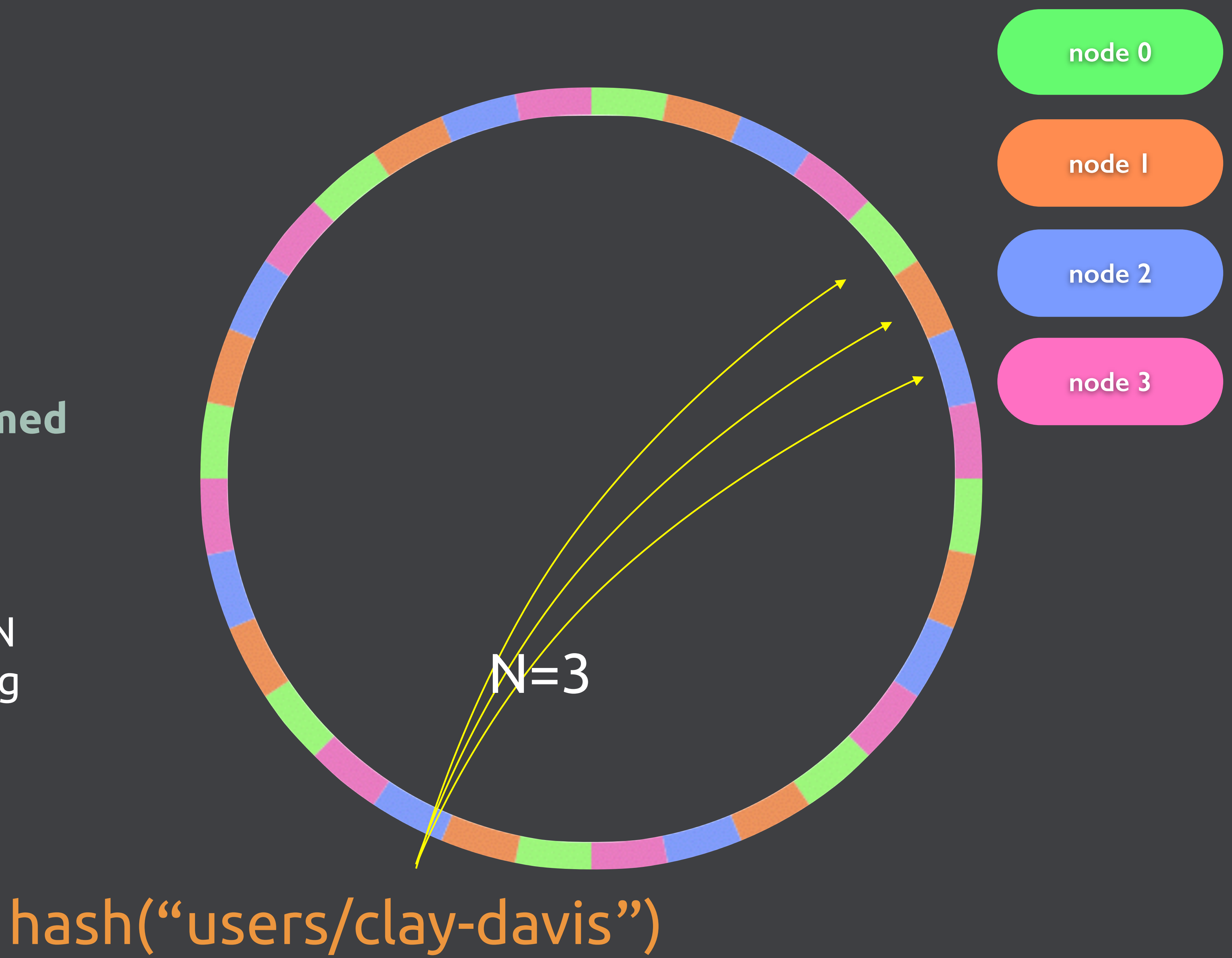
Riak Overview

Erlang implementation of Dynamo

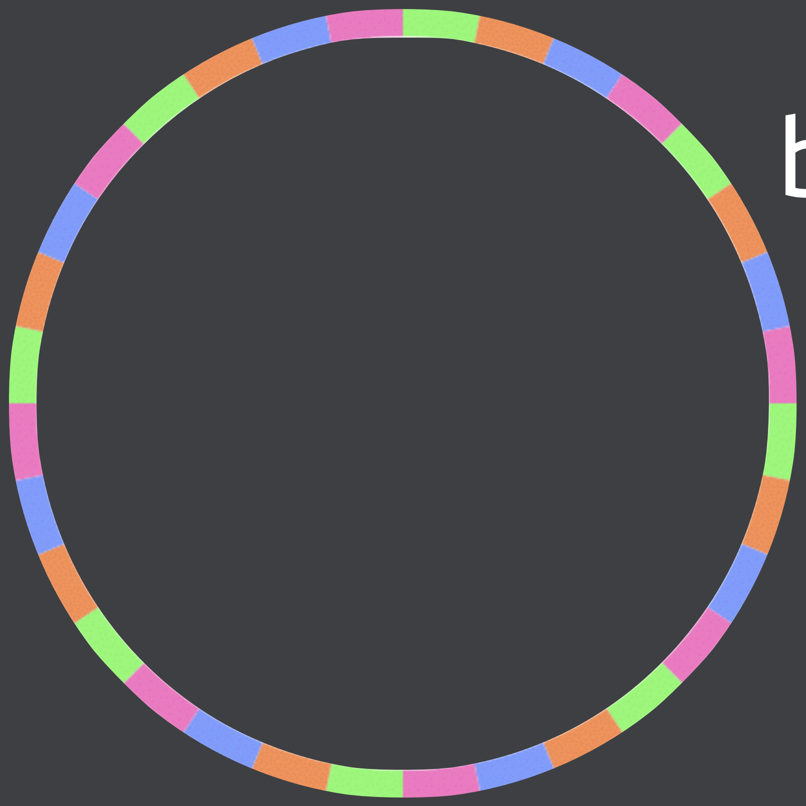
{“**key**”: “**value**”}

Riak Overview
Consistent Hashing

- 160-bit integer **keyspace**
- divided into **fixed** number of **evenly-sized partitions/ ranges**
- partitions are **claimed** by nodes in the cluster
- replicas go to the N partitions following the key



The Ring



supervisor process

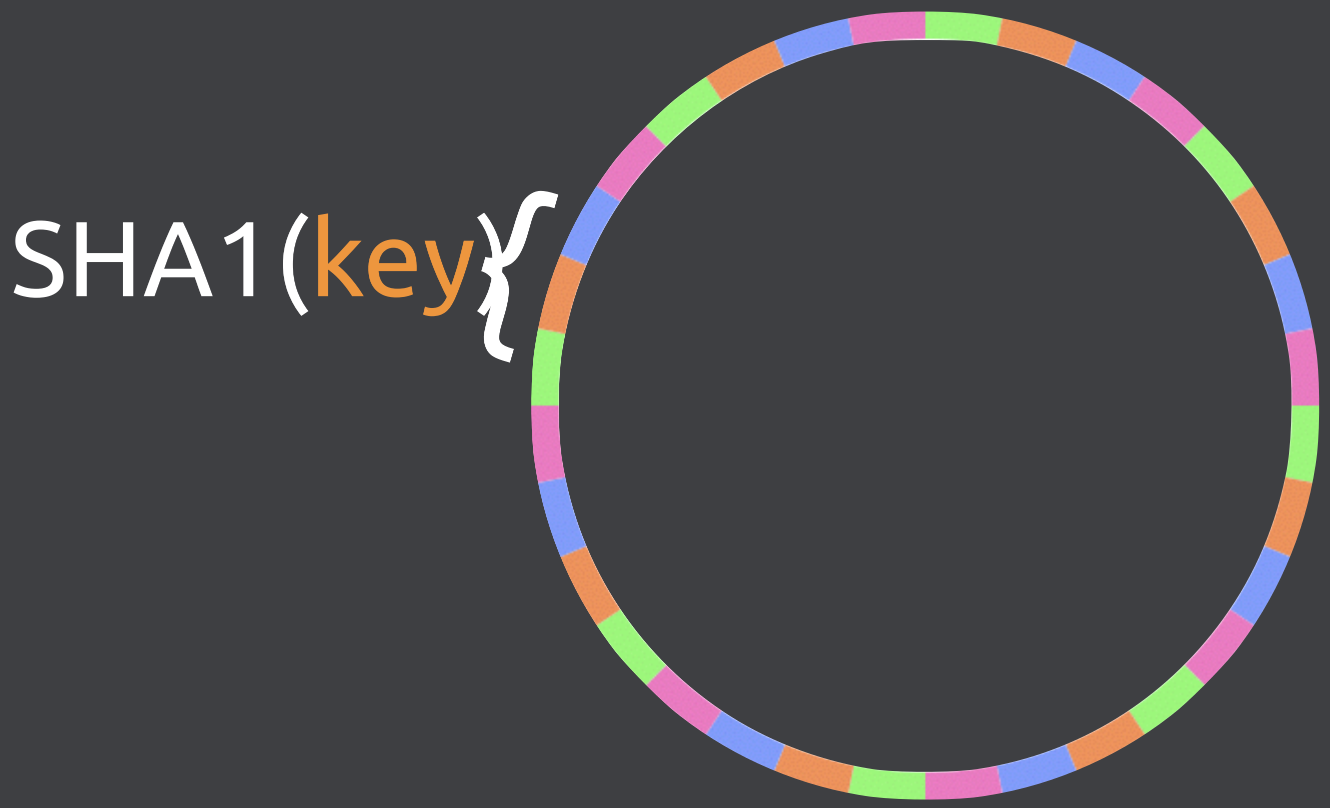
basic unit of concurrency

32, 64, 128, 256, 512....

vnodes = **ring_size**

10-50 vnodes / node

VNODES



preflist

Availability

Any non-failing node can respond to any request

--Gilbert & Lynch

node 0

node 1

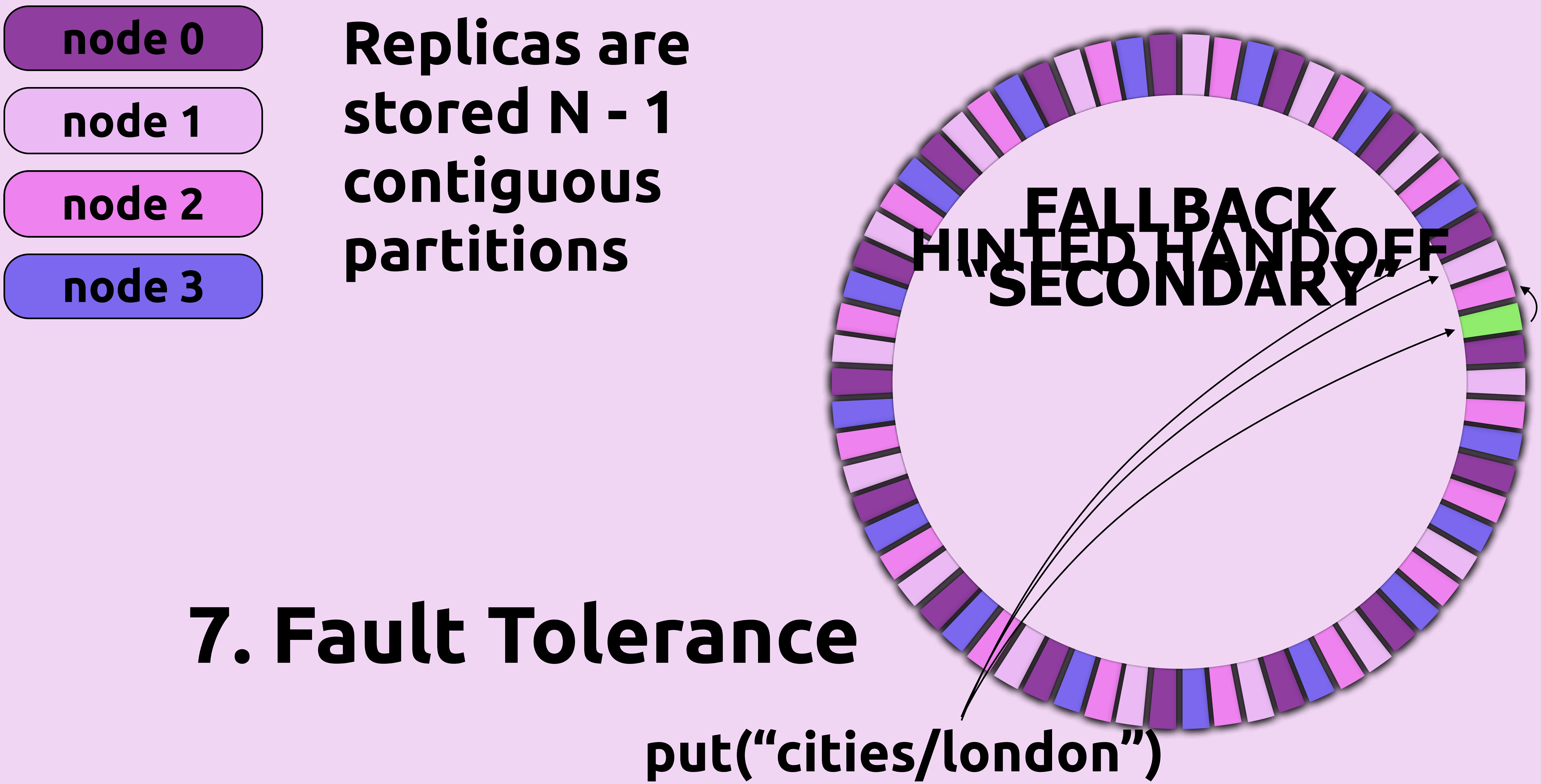
offline

node 3

Replicas are stored N - 1 contiguous partitions

7. Fault Tolerance

Replicas are stored N - 1 contiguous partitions



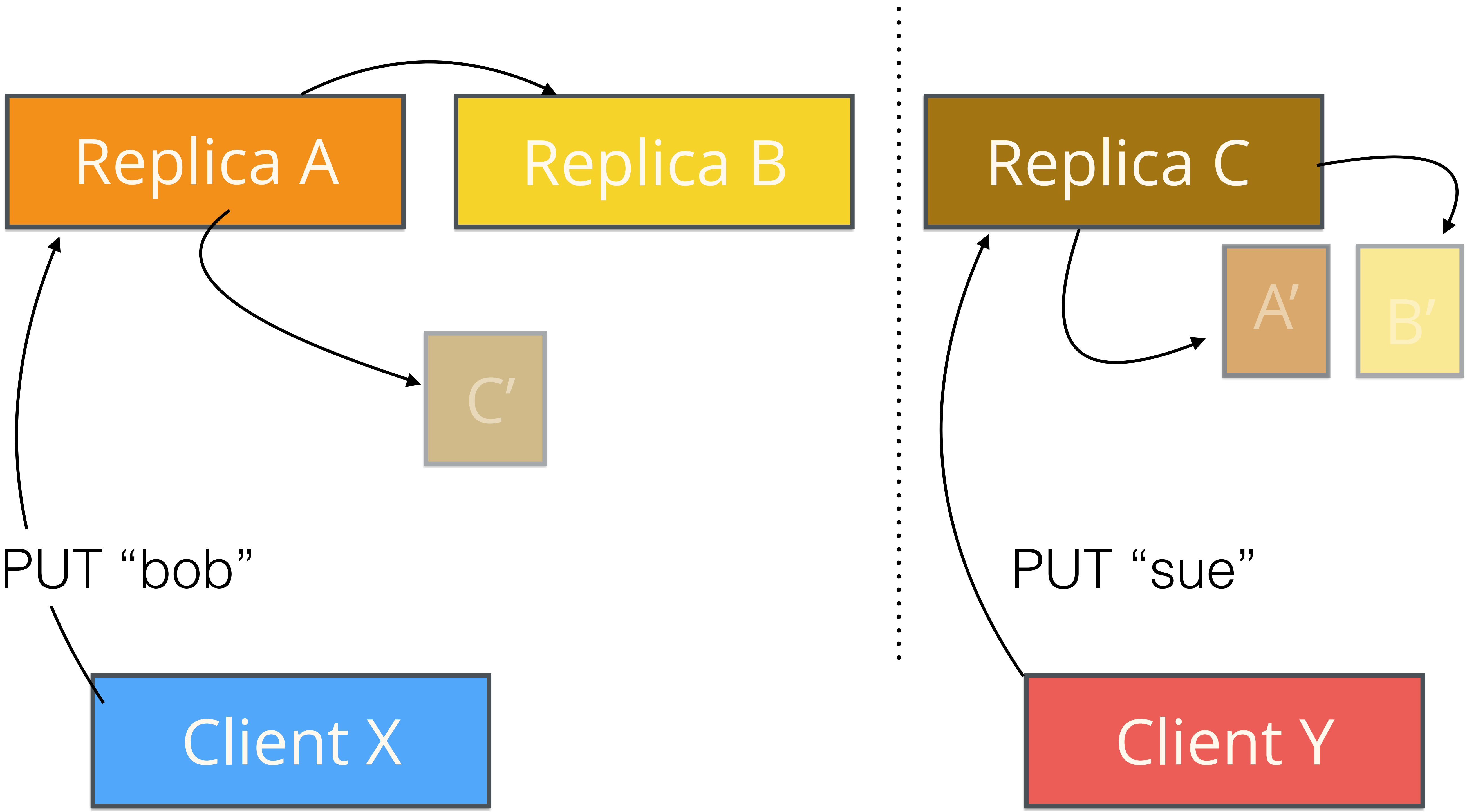
7. Fault Tolerance

Riak Overview

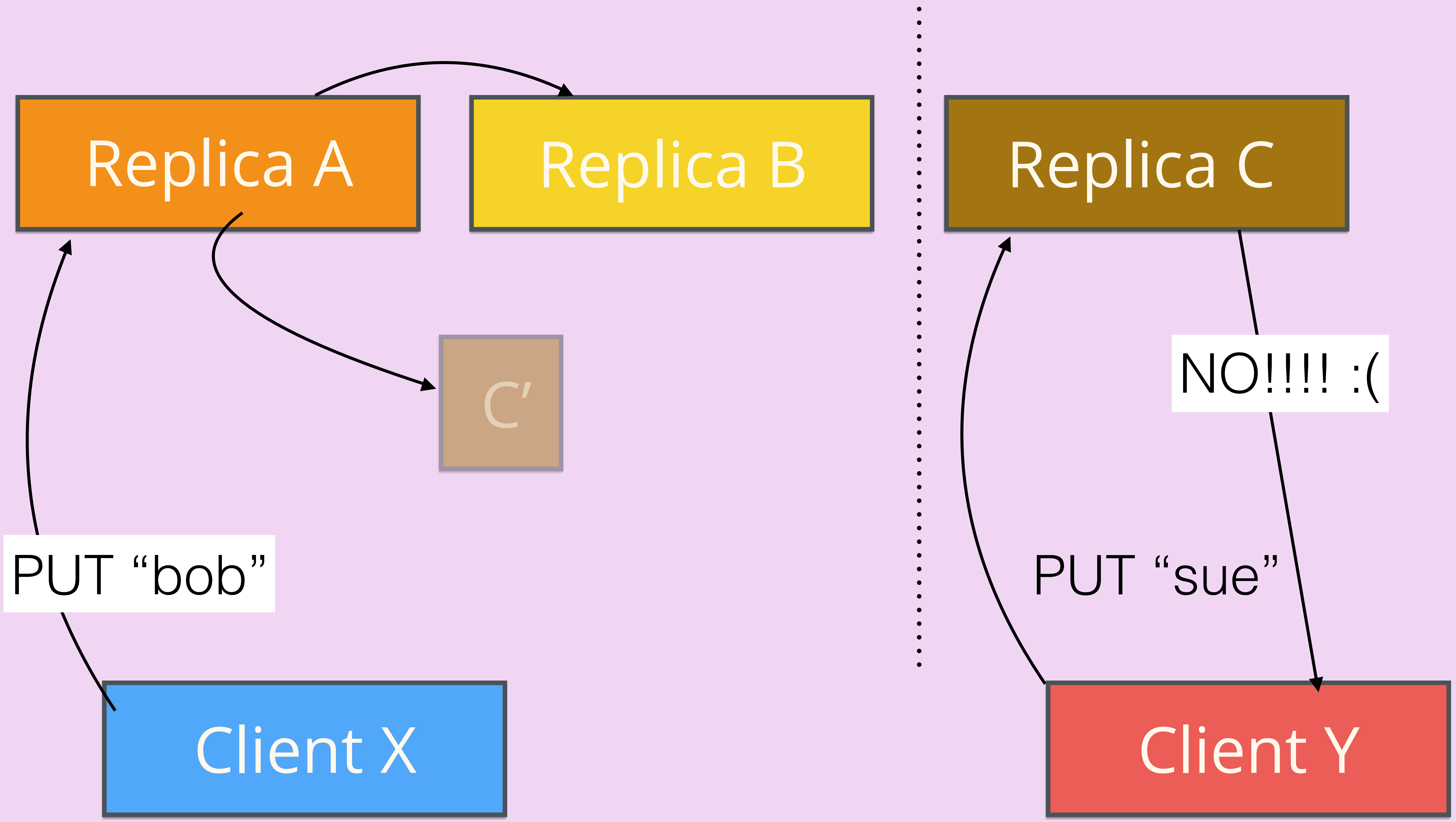
Quorum

Quora: For Consistency

- How many Replicas must respond: 1, quorum, all?
- Strict Quorum: Only Primaries are contacted
- Sloppy Quorum: Fallbacks are contacted
- Fast Writes? W=1
- Fast Reads? R=1
- Read Your Own Writes? PW+PR>N


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

Sloppy

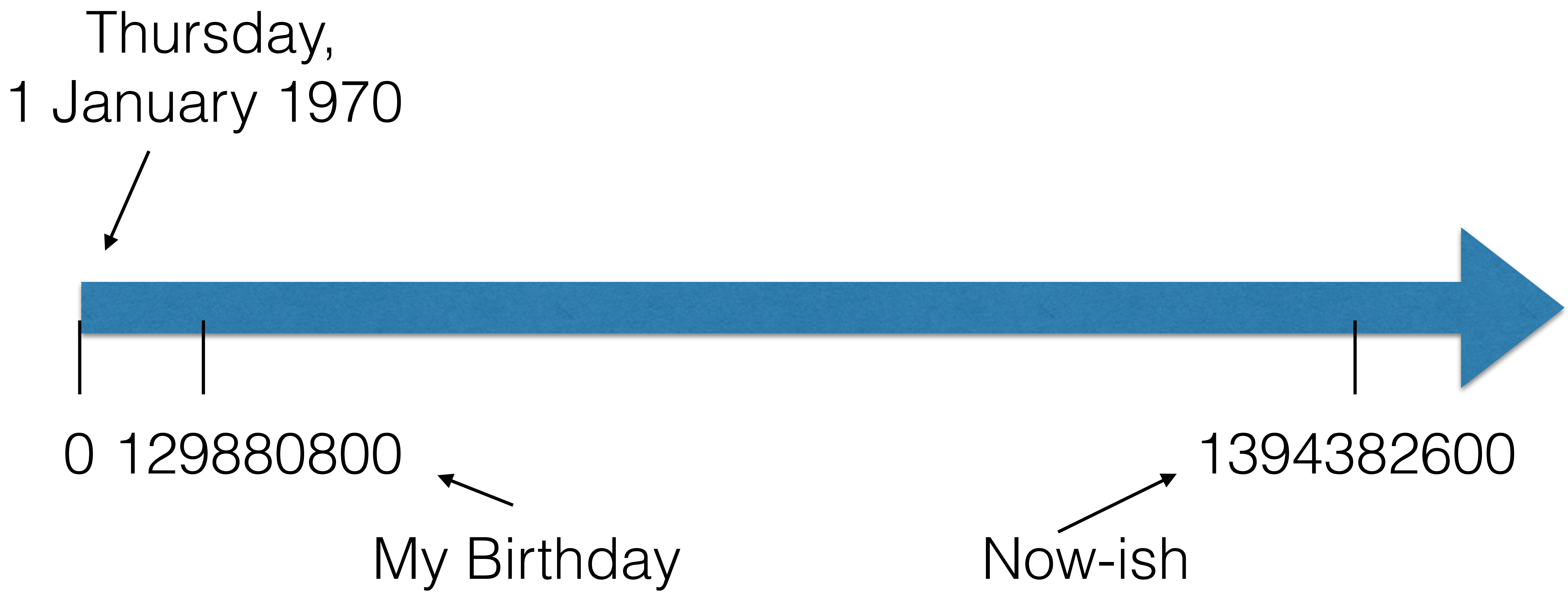


Replica A
Replica B
Replica C
C'
NO!!!! :(
PUT "bob"
PUT "sue"
Client X
Client Y

Strict

"'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



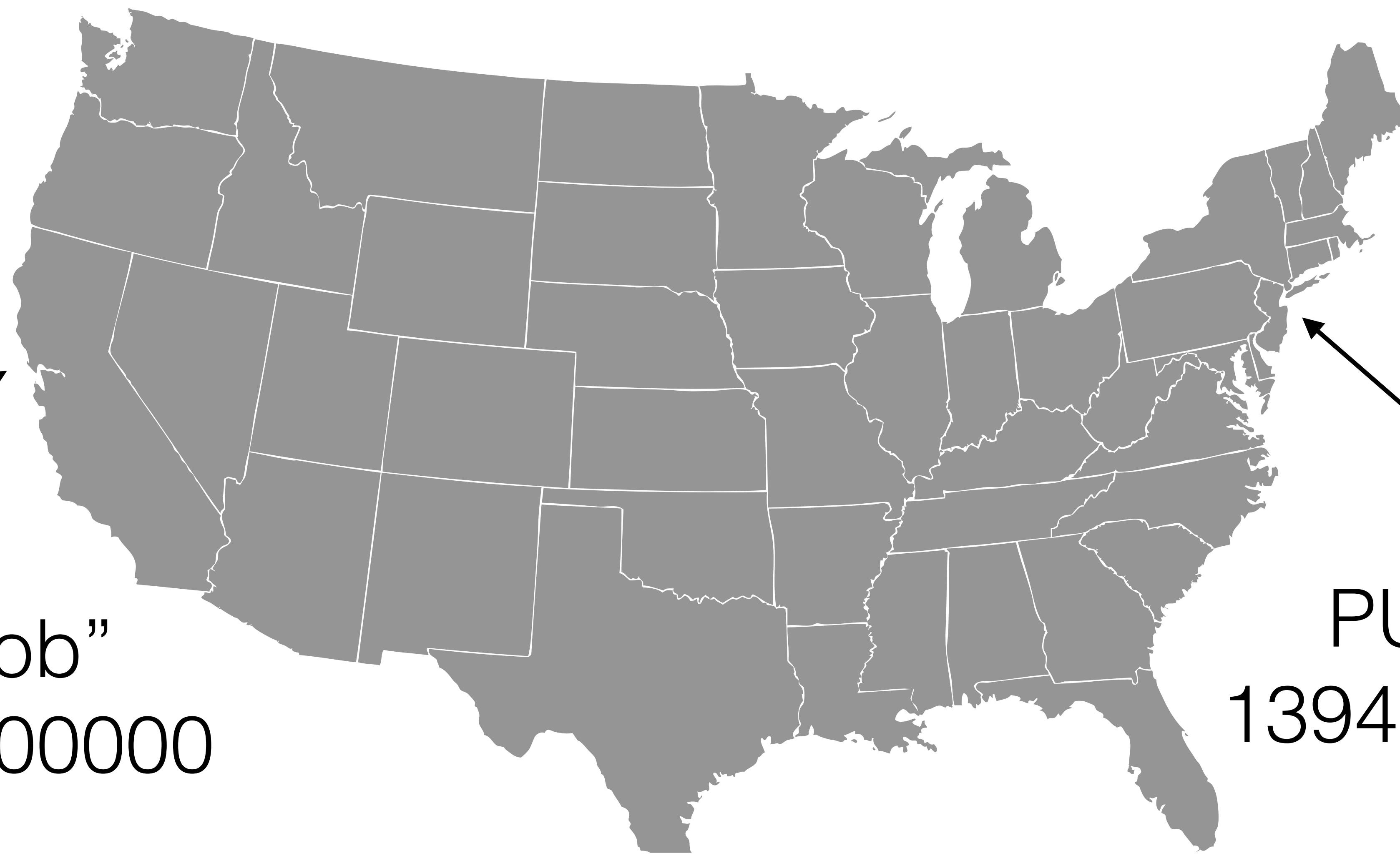
Temporal Clocks

posix time number line

temporal clocks

- CAN
  - A could NOT have caused B
  - A could have caused B
- CAN’T
  - A caused B

Physics Problem



PUT “bob”
1394382600000

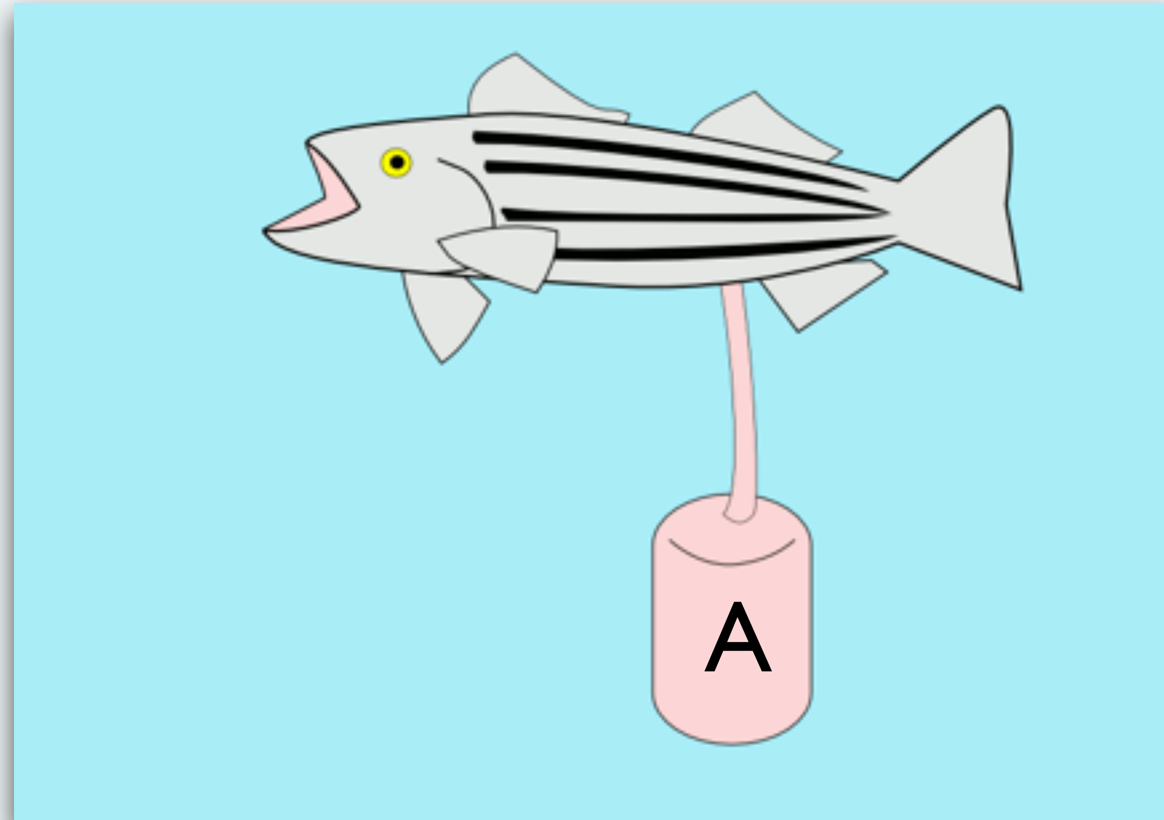
PUT “sue”
1394382600020

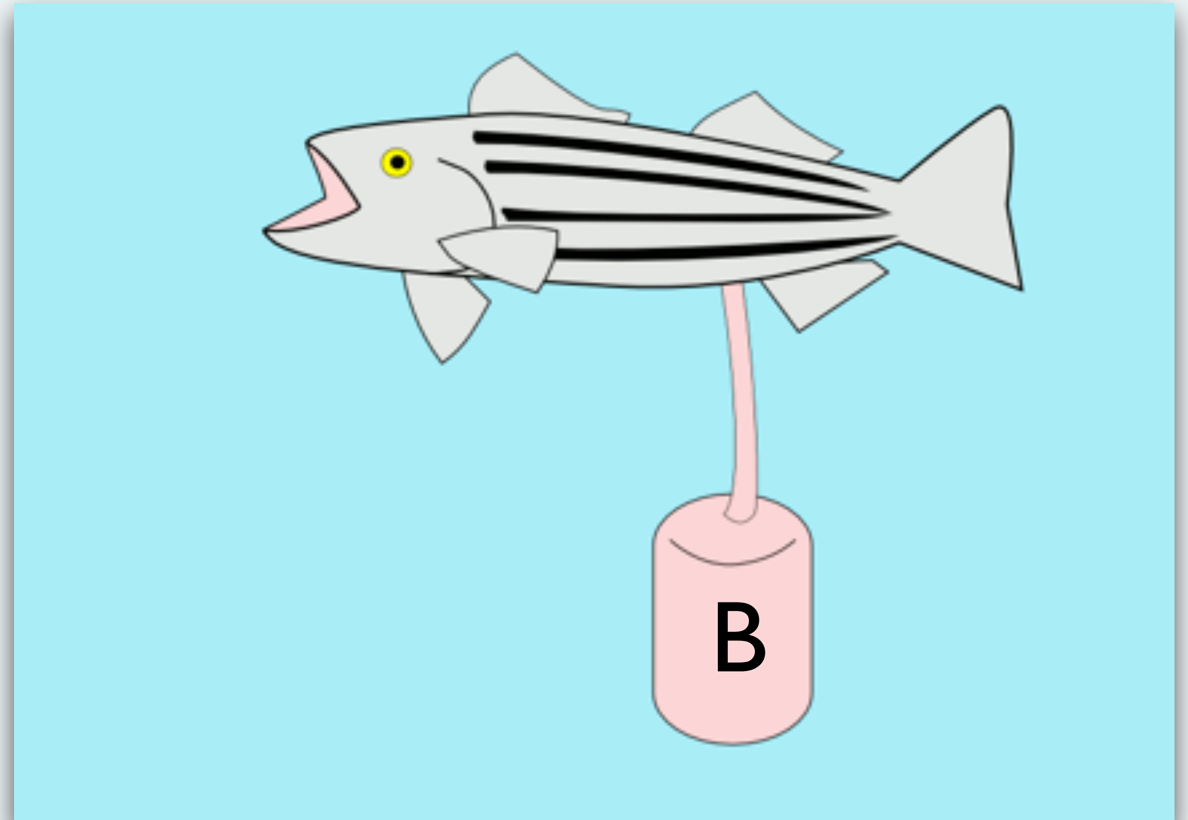
SF → NY

4,148 km
14 ms Light
21 ms fibre

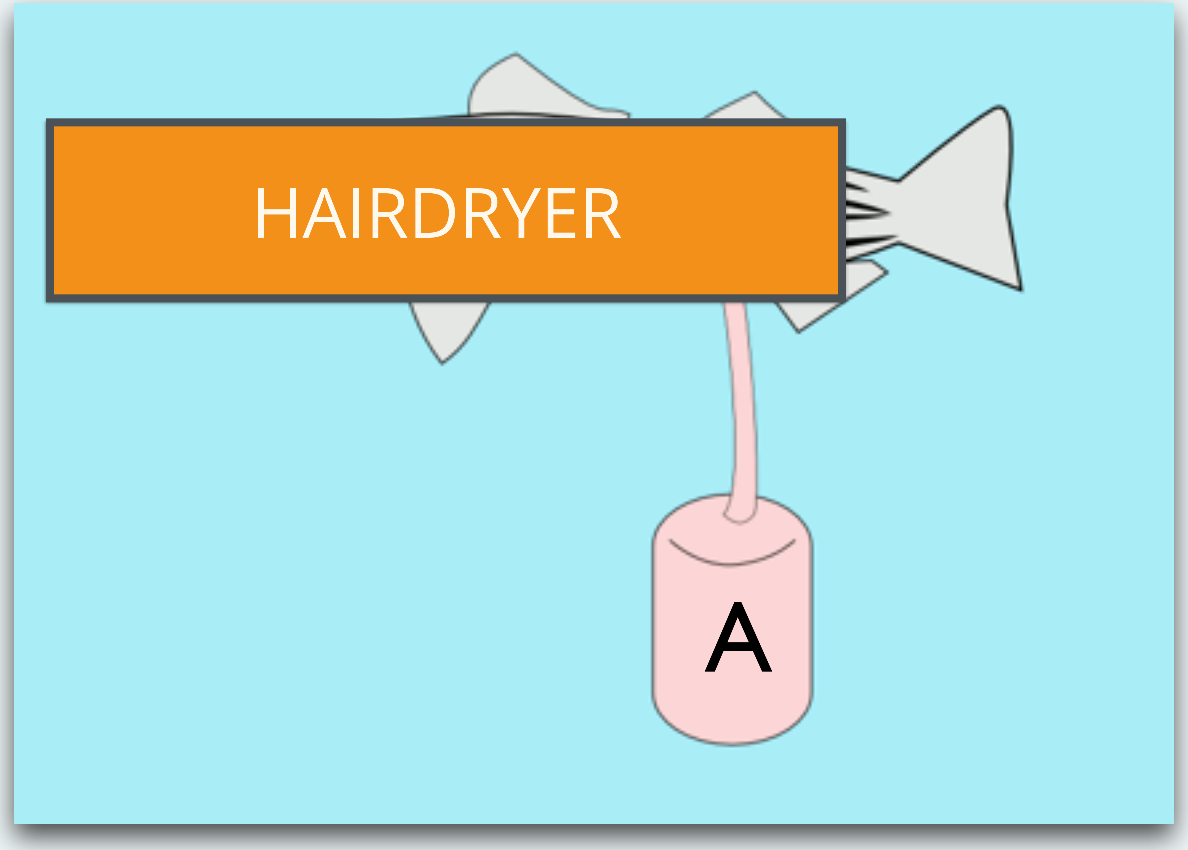
Dynamo

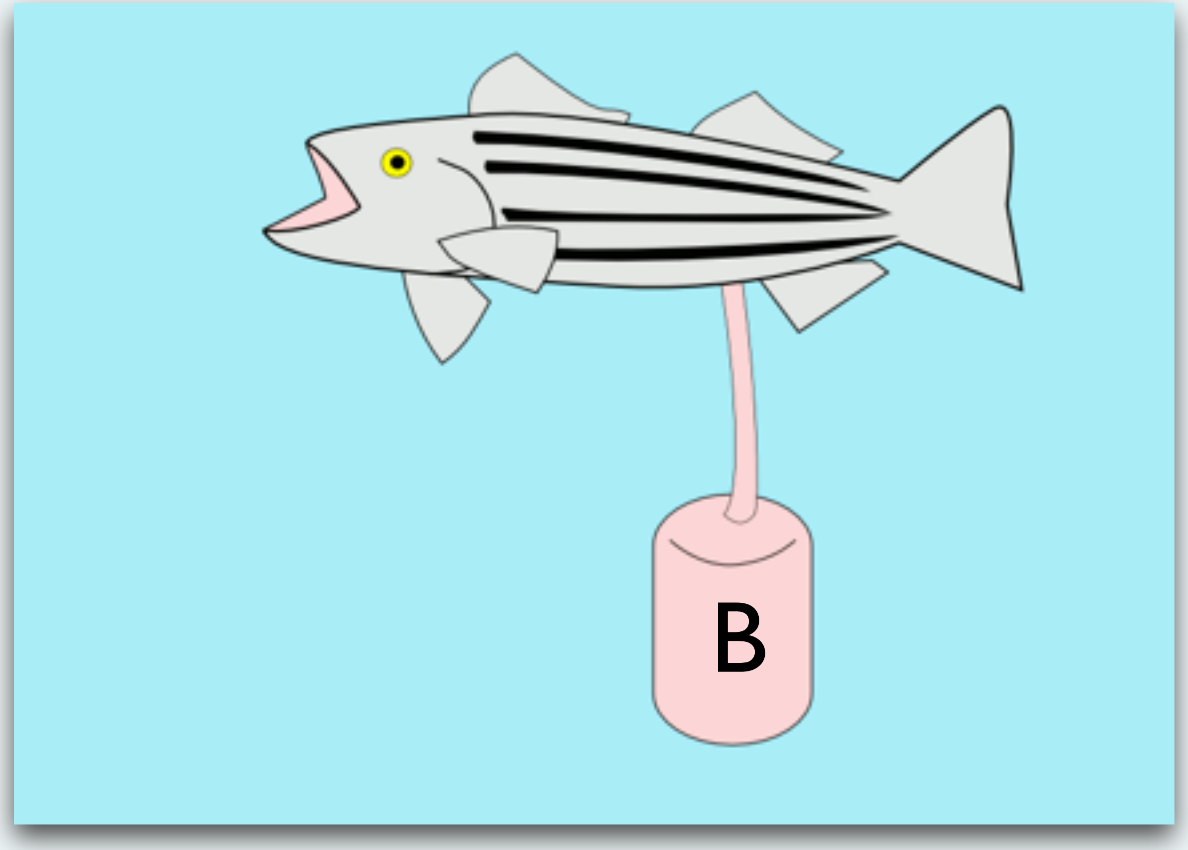
The Shopping Cart



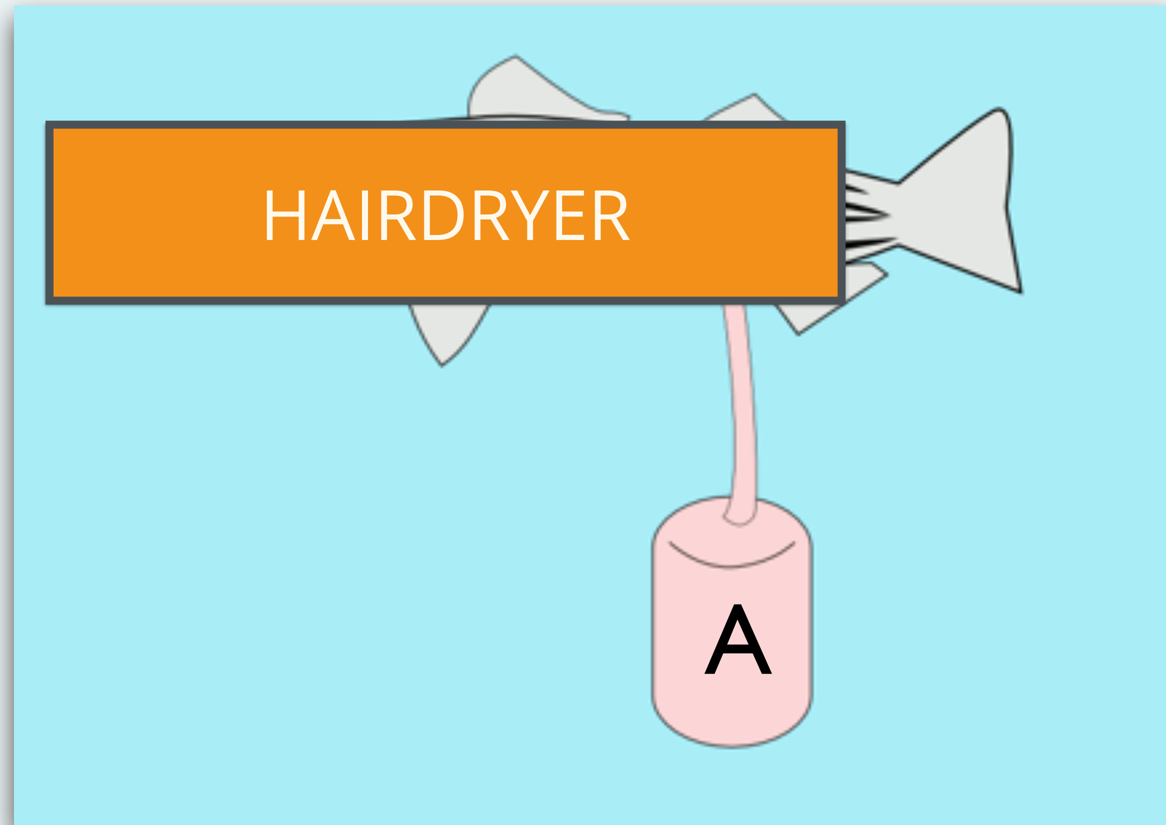


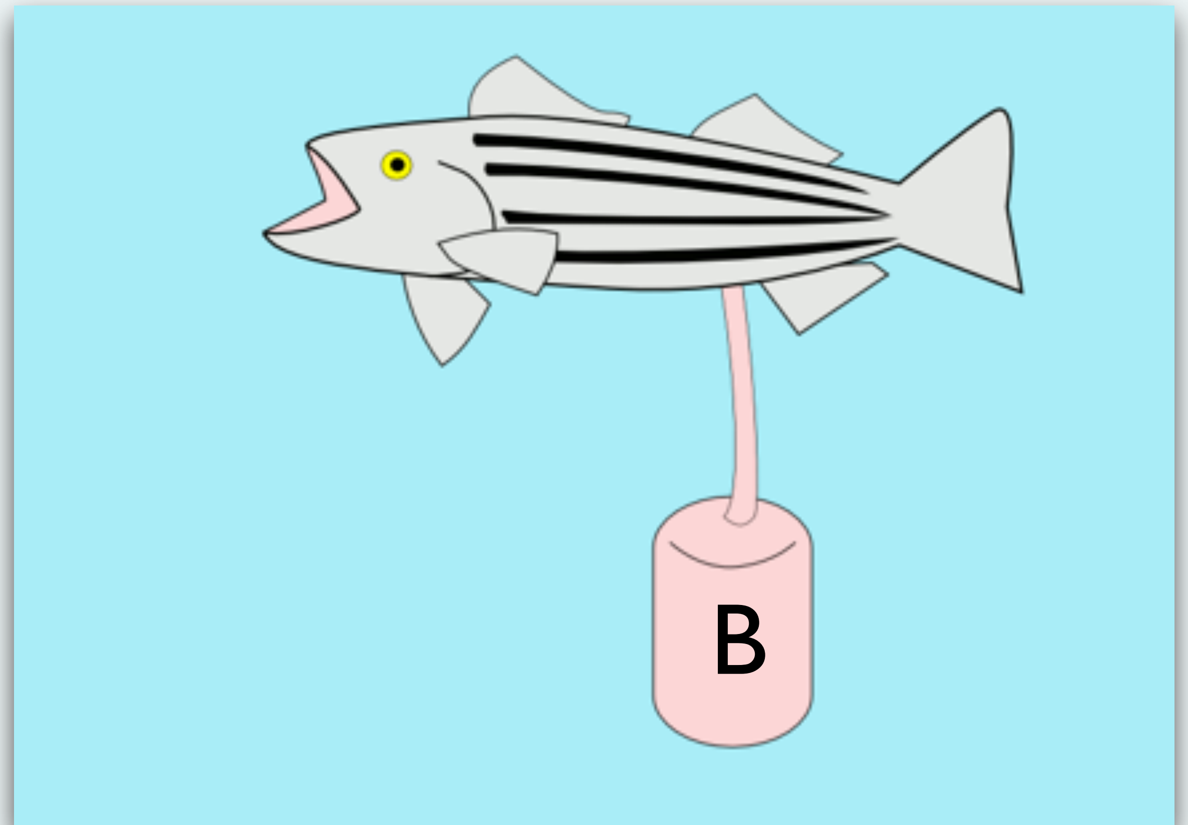
HAIRDRYER



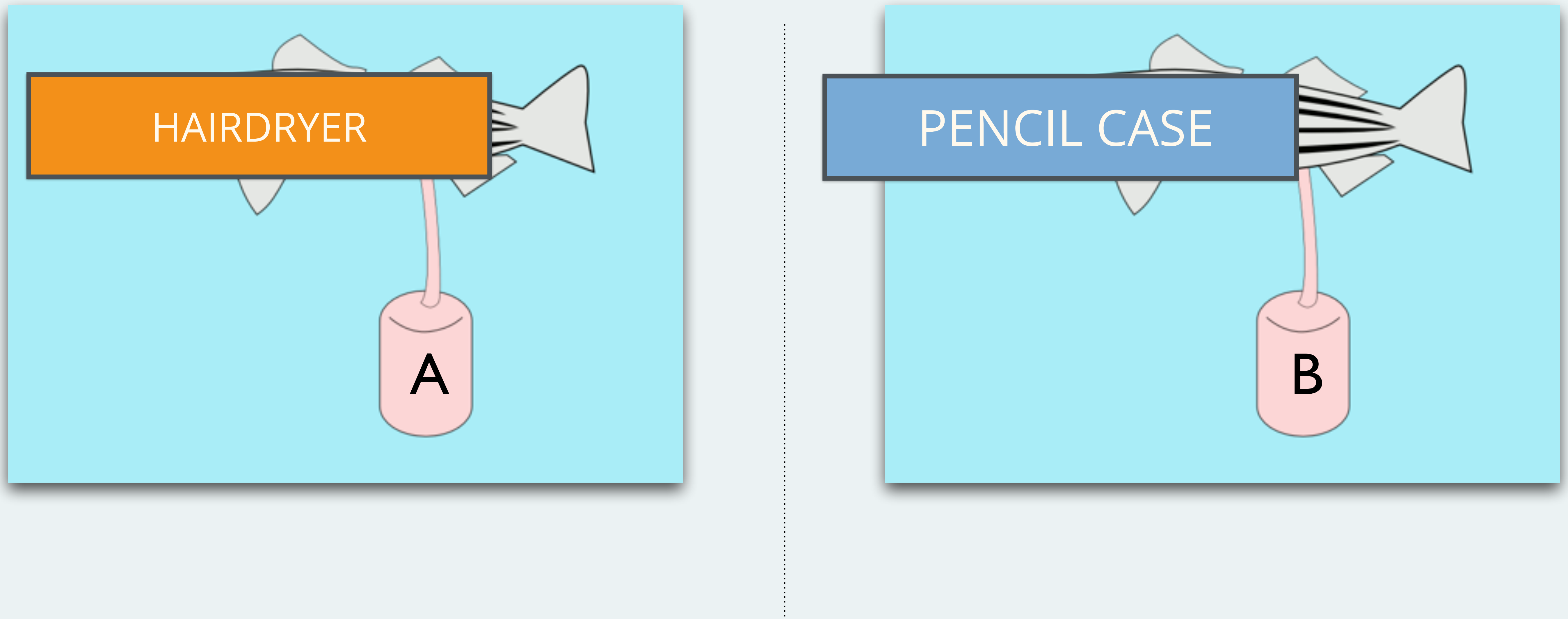


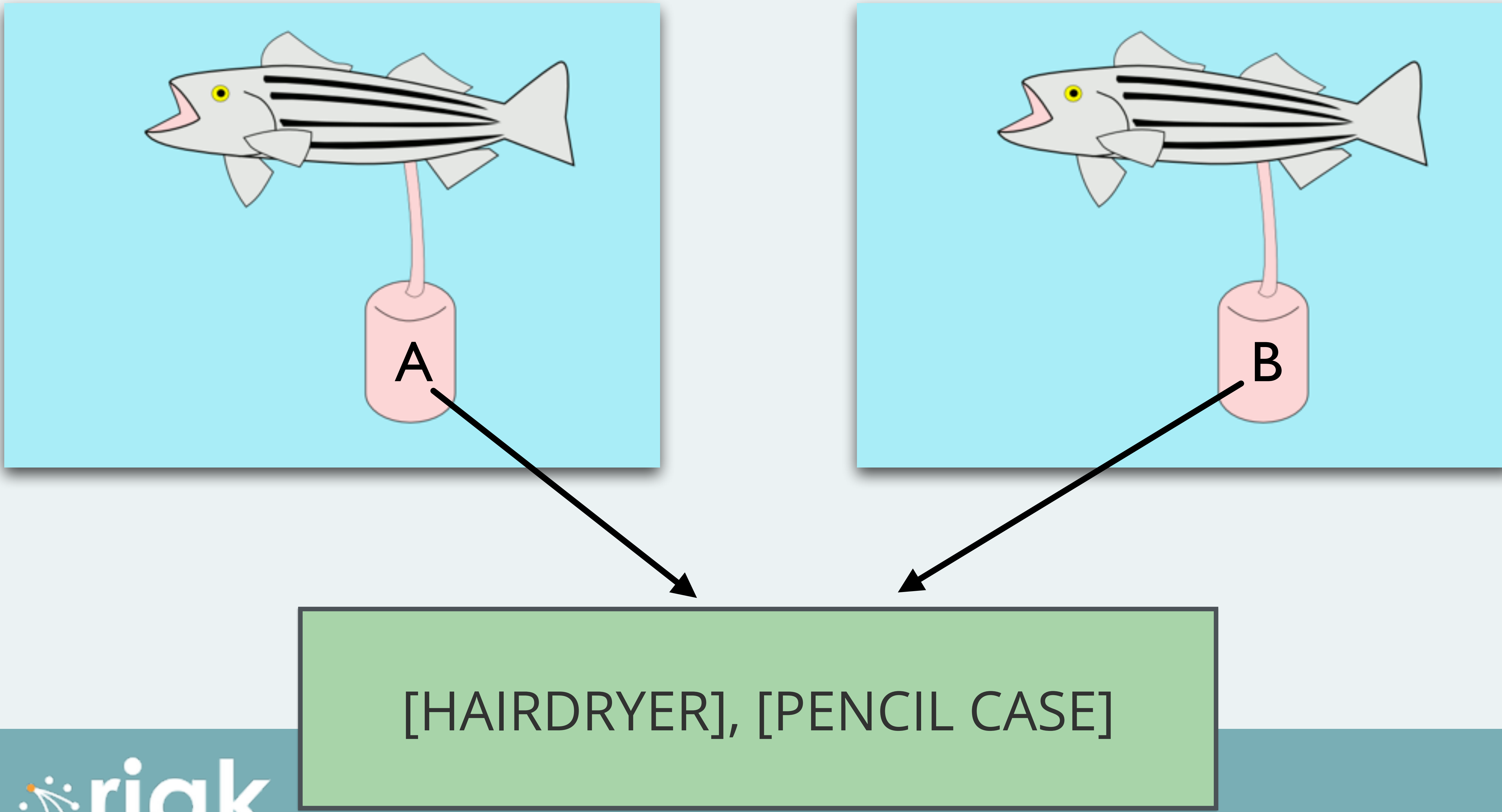
riak





PENCIL CASE


HAIRDRYER
A
PENCIL CASE
B



riak

Clocks, Time, And the Ordering of Events

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

- Logical Time
- Causality
- A influenced B
- A and B happened at the same time

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?

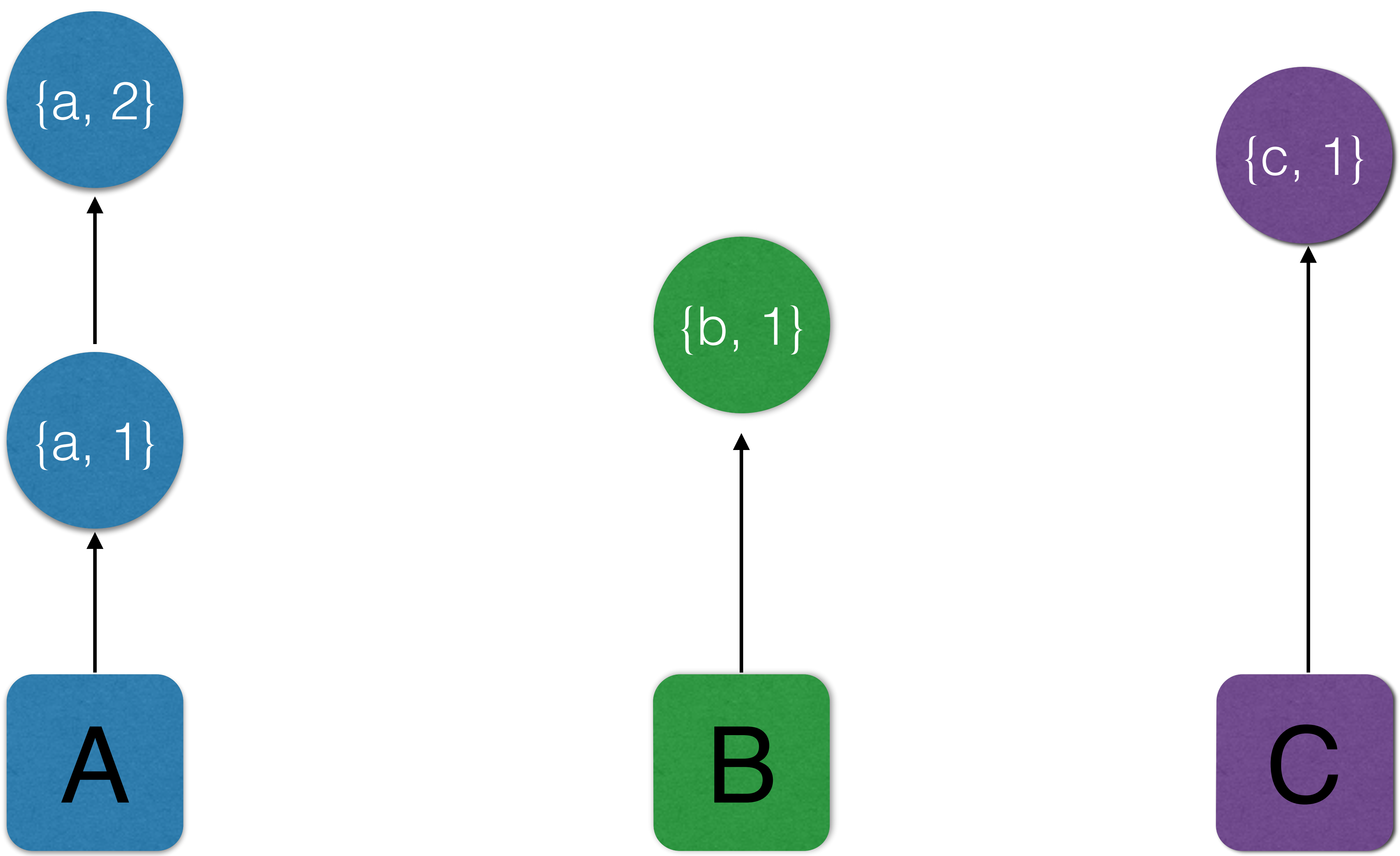
version vectors - updates to a data item

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

Version Vectors

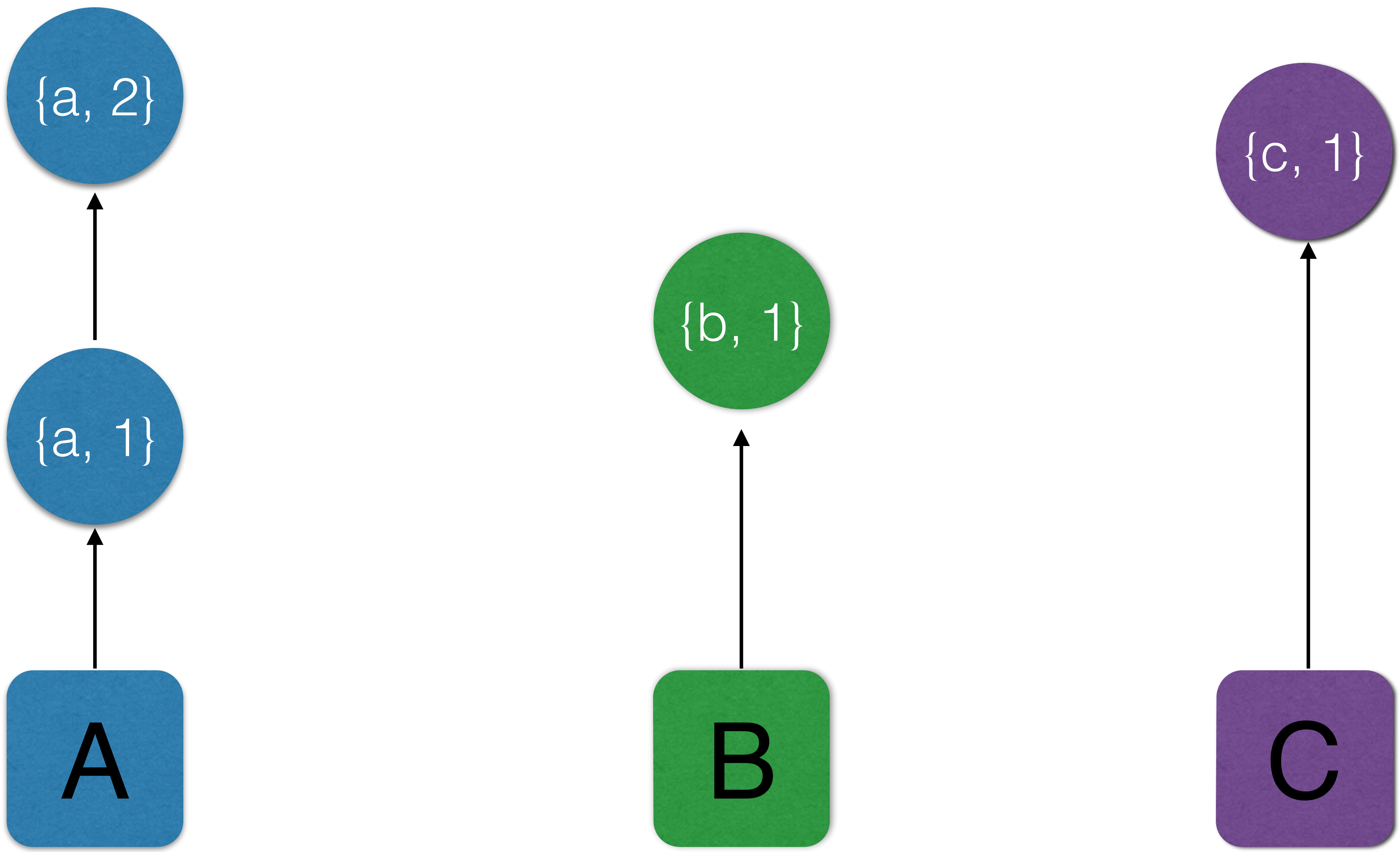
Version Vectors

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



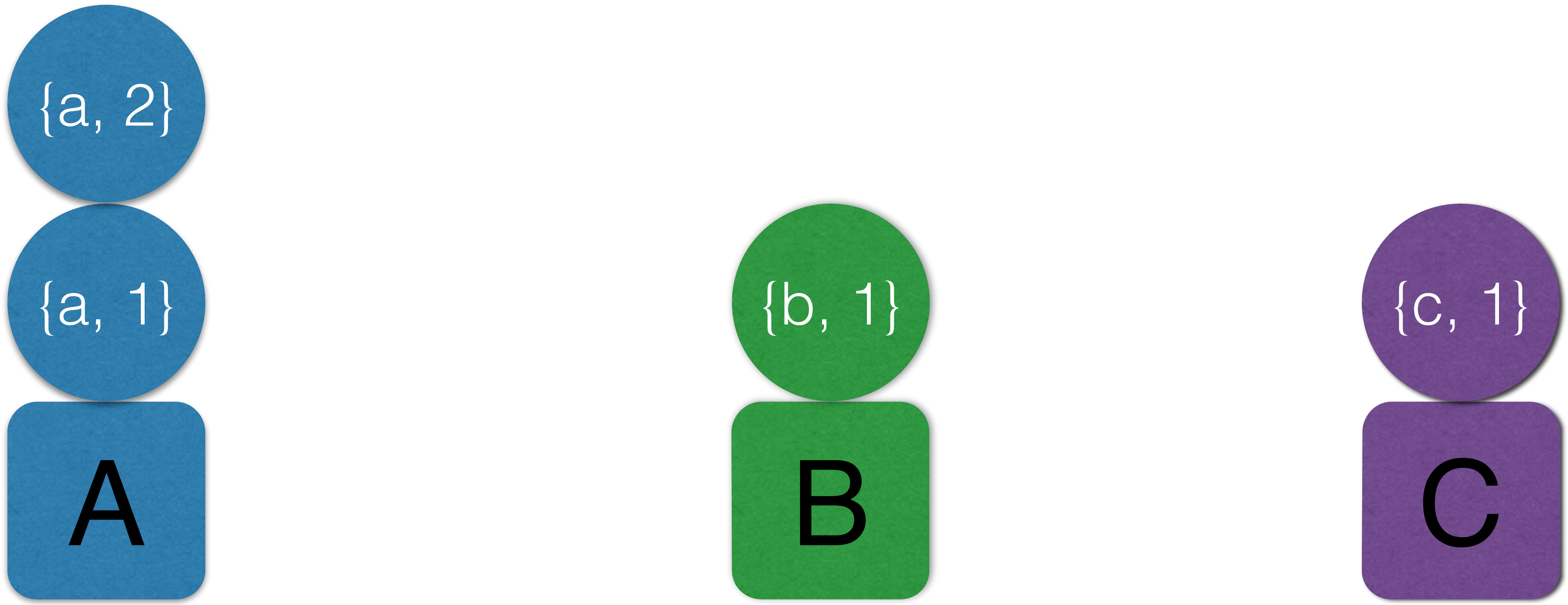
Version Vectors

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



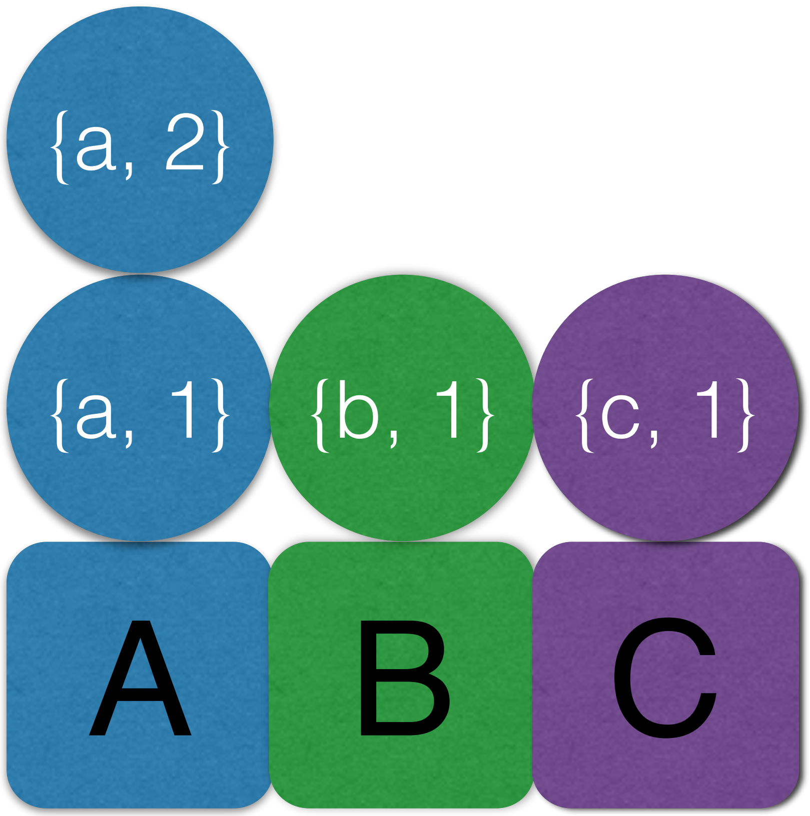
Version Vectors

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



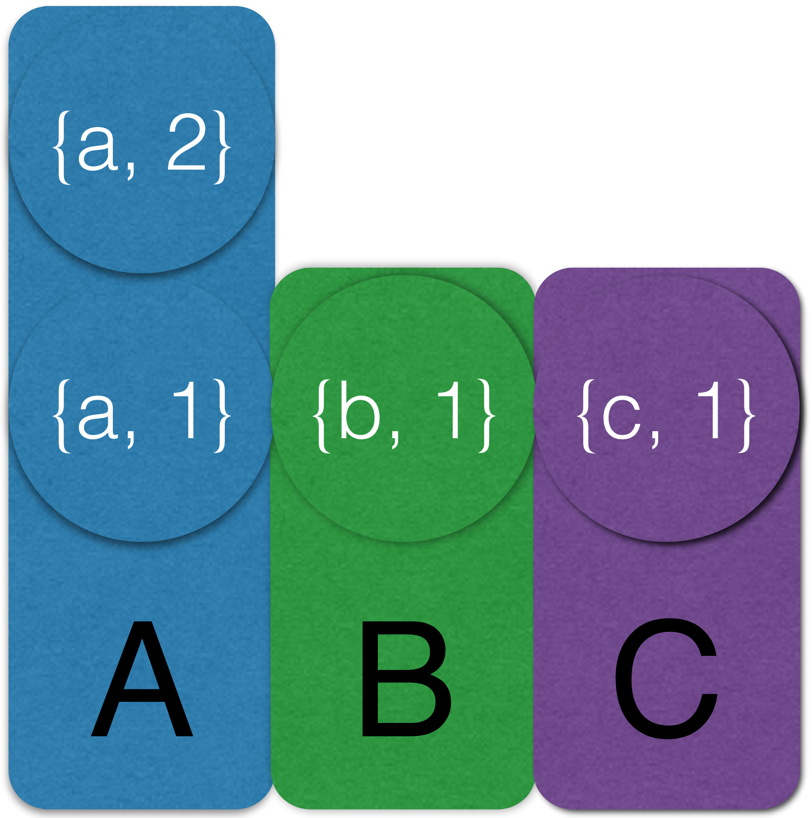
Version Vectors

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



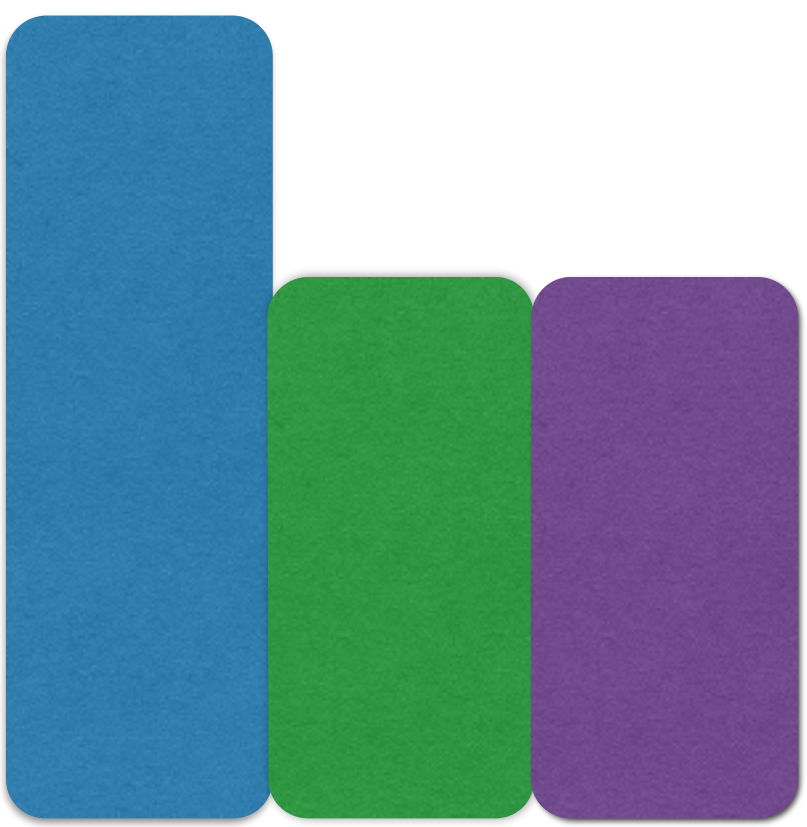
Version Vectors

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



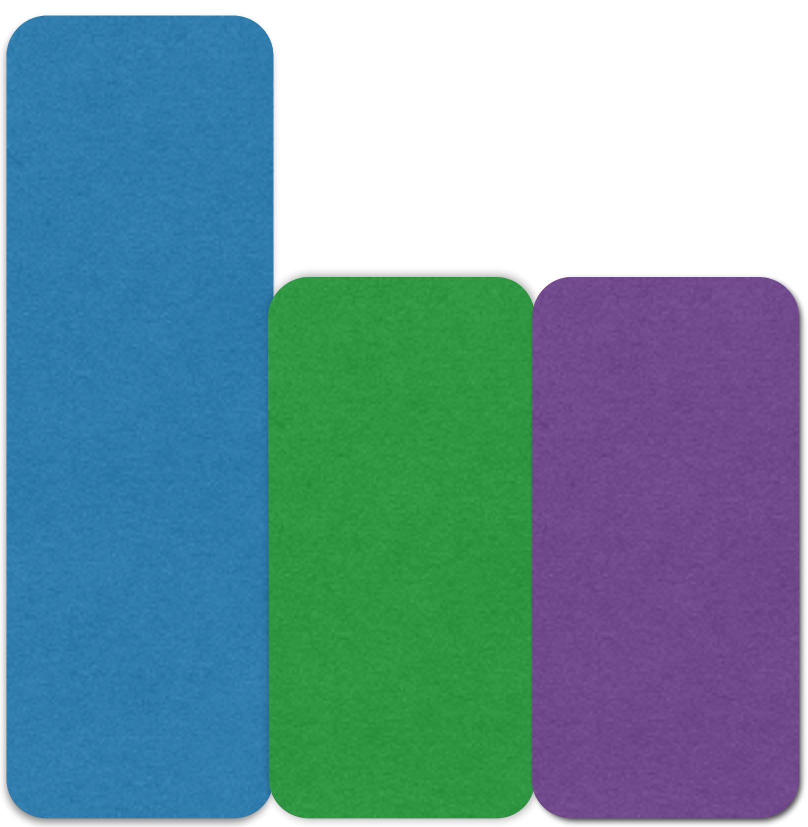
Version Vectors

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



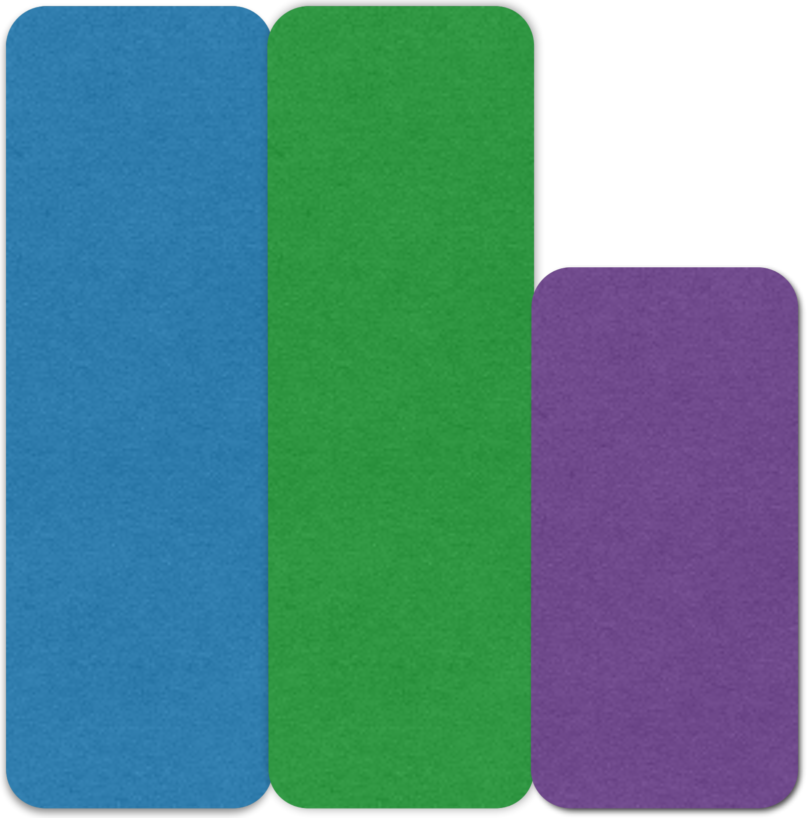
Version Vectors
Update

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



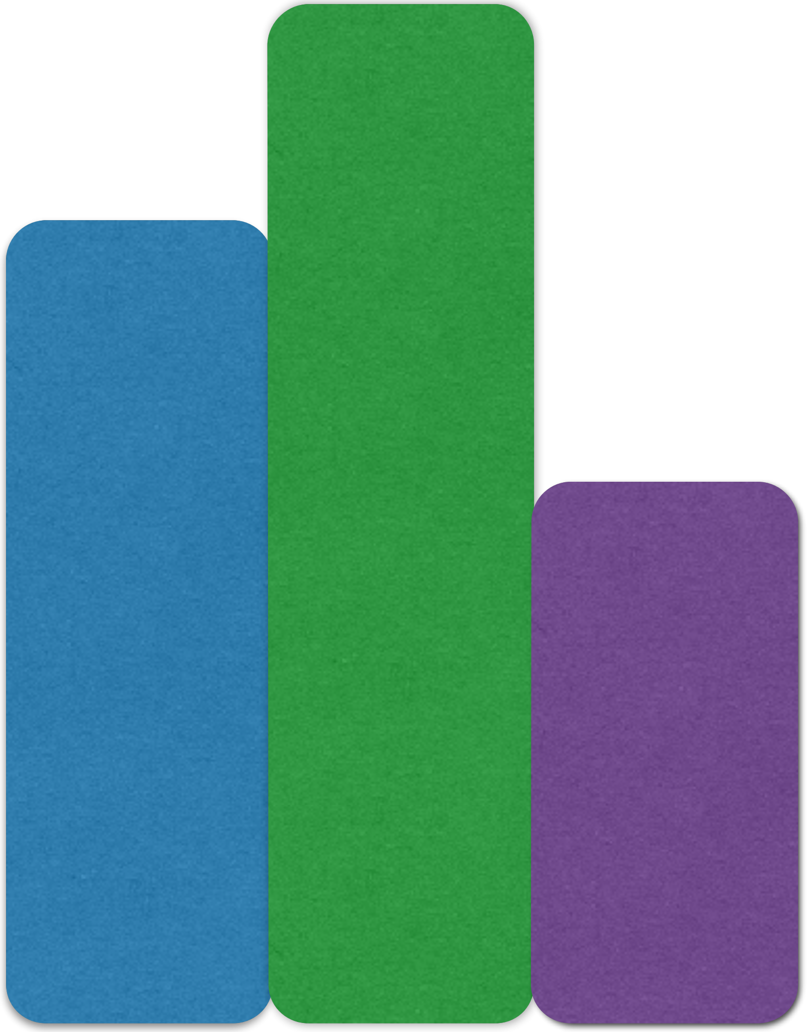
Version Vectors
Update

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



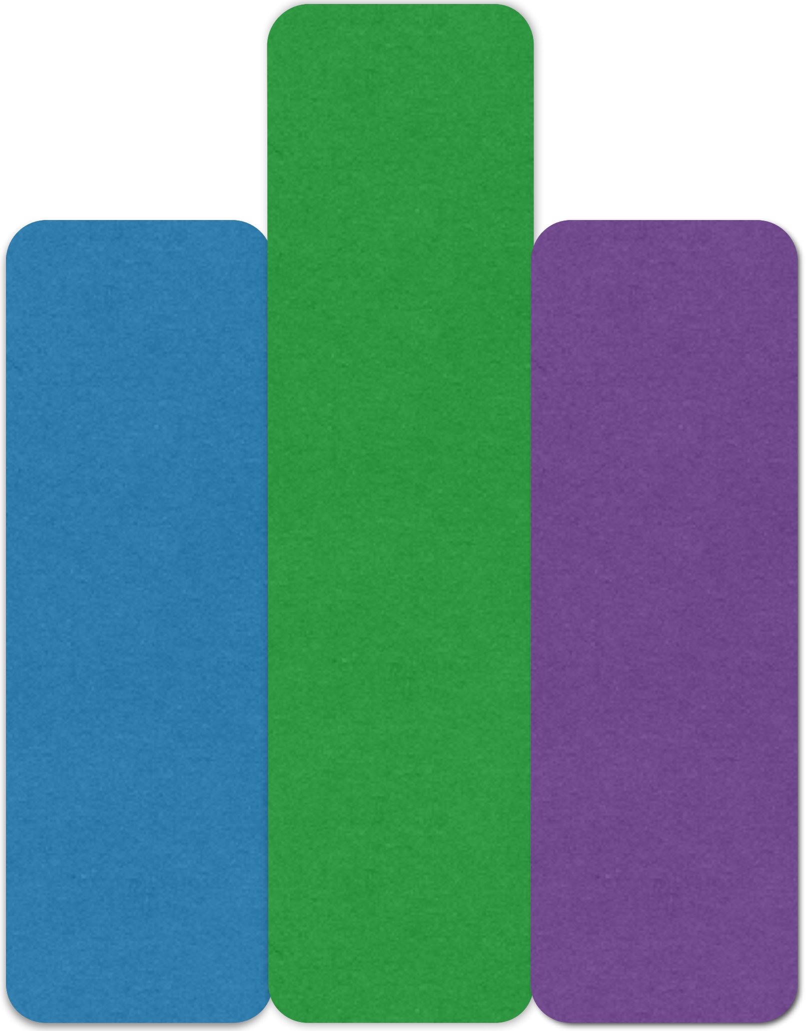
Version Vectors
Update

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



Version Vectors
Update

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

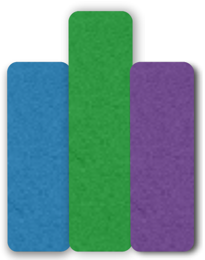


Version Vectors Descends

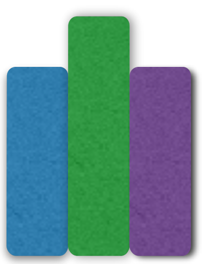
- A descends B : A >= B
- A has seen all that B has
- A summarises at least the same history as B

Version Vectors
Descends

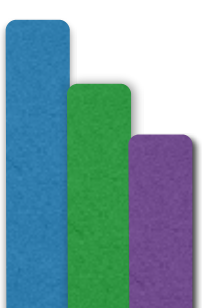
[{a,2}, {b,3}, {c,2}]



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

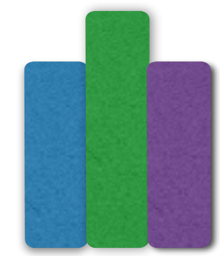


[{a,4}, {b,3}, {c,2}]



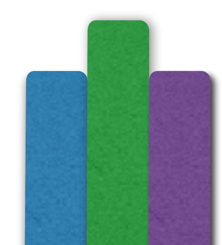
>=

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



[]

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



Version Vectors
Dominates

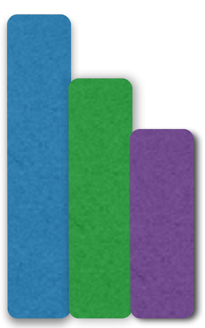
- A dominates B : A > B
- A has seen all that B has, and at least one more event
- A summarises a greater history than B

Version Vectors
Dominates

[{a,1}]

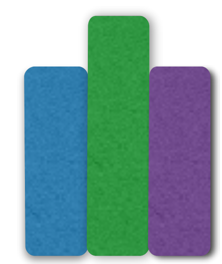
[]

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

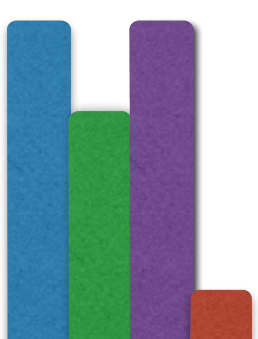


>

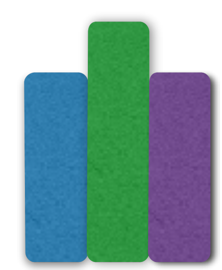
[{a,2}, {b,3}, {c,2}]



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



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



Version Vectors
Concurrent

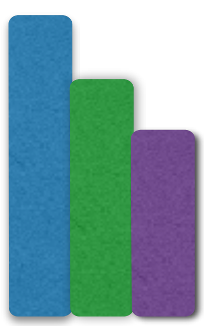
- A concurrent with B : A | B
- A does not descend B AND B does not descend A
- A and B summarise disjoint events
- A contains events unseen by B AND B contains events unseen by A

Version Vectors
Concurrent

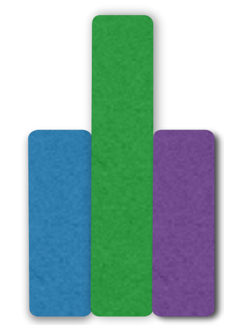
[{a,1}]

[{b,1}]

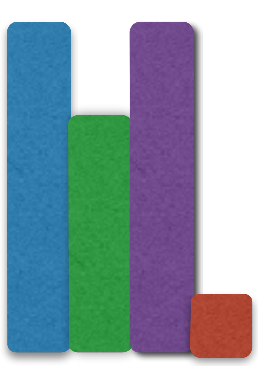
[{a,4}, {b,3}, {c,2}]



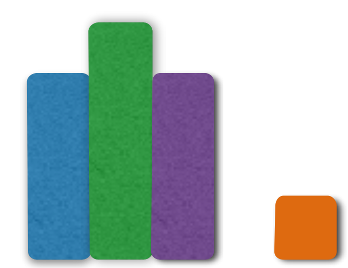
[{a,2}, {b,4}, {c,2}]

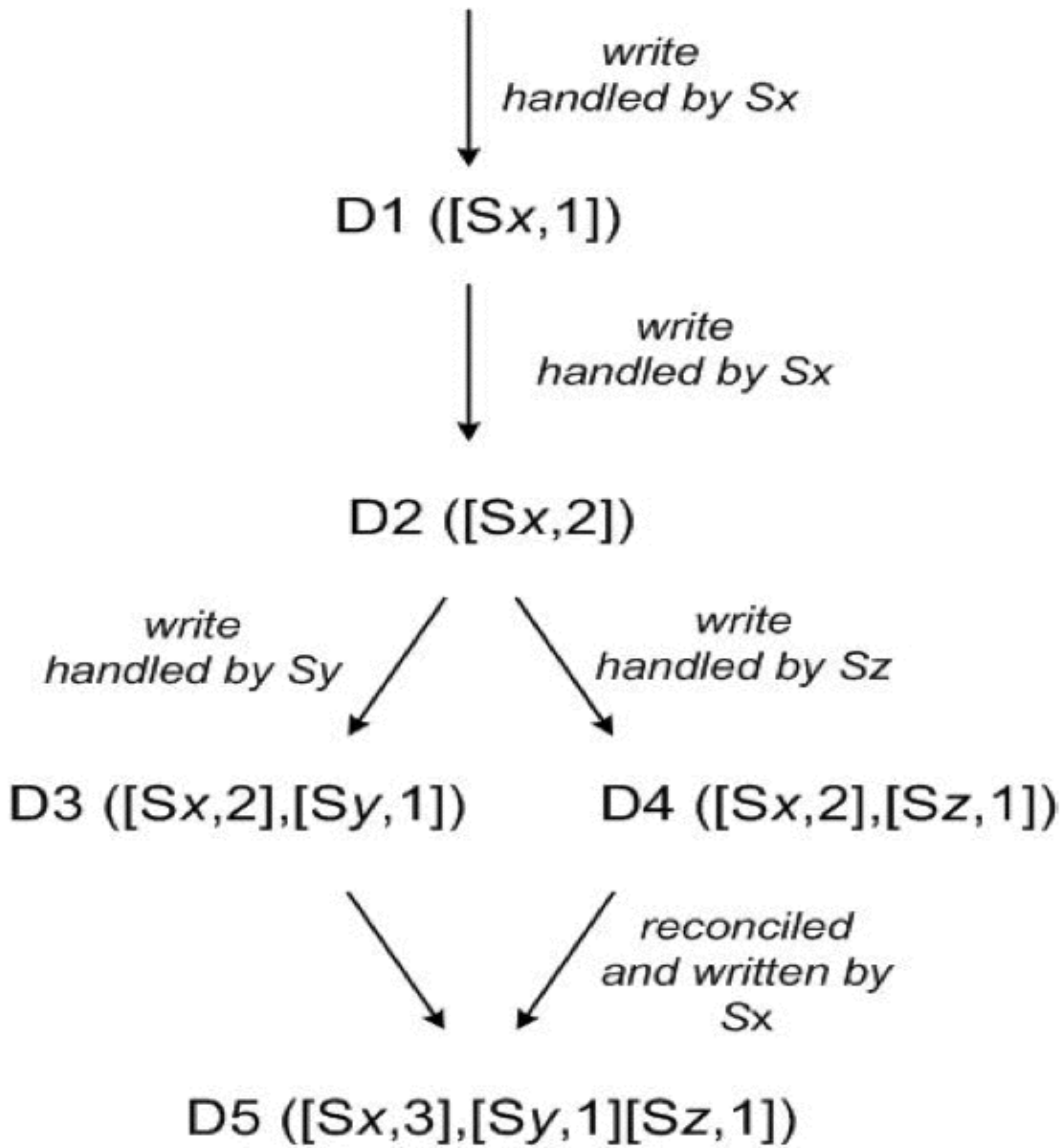


[{a,5}, {b,3}, {c,5}, {d, 1}]



[{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])

Version Vectors
Merge

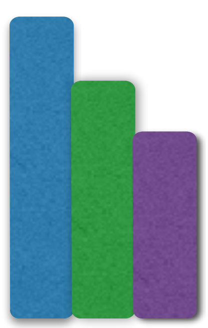
- A merge with B : $A \sqcup B$
- $A \sqcup B = C$
- C >= A and C >= B
- If A | B C > A and C > B
- C summarises all events in A and B
- Pairwise max of counters in A and B

Version Vectors
Merge

[{a,1}]

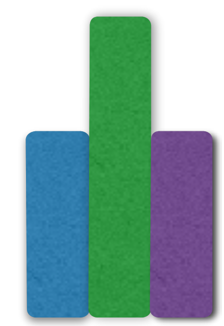
[{b,1}]

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

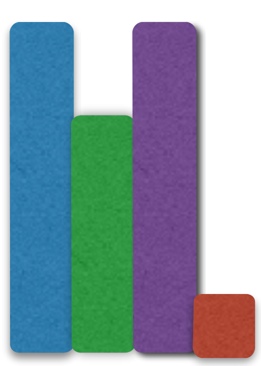


⊔

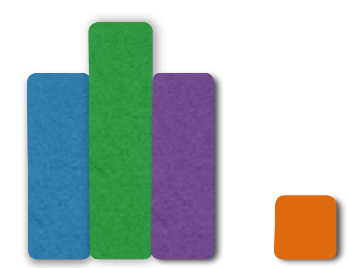
[{a,2}, {b,4}, {c,2}]



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



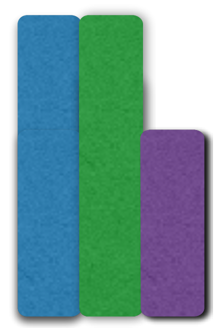
[{a,2}, {b,4}, {c,2}, {e,1}]



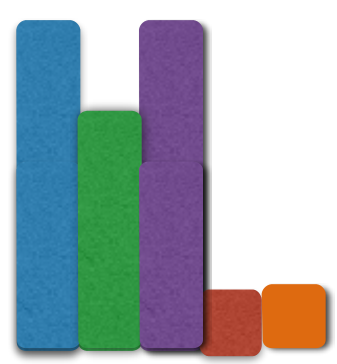
Version Vectors
Merge

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

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



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



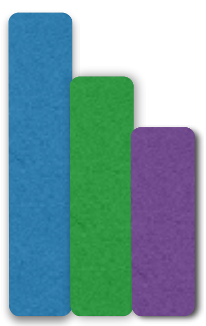
Syntactic Merging

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

Temporal vs Logical

A

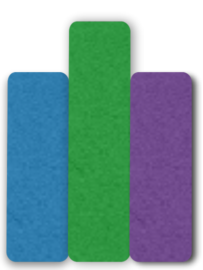
[{a,4}, {b,3}, {c,2}]



“Bob”

B

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



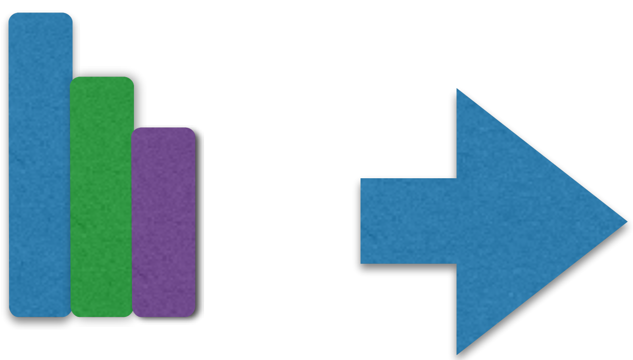
“Sue”

?

Temporal vs Logical

A

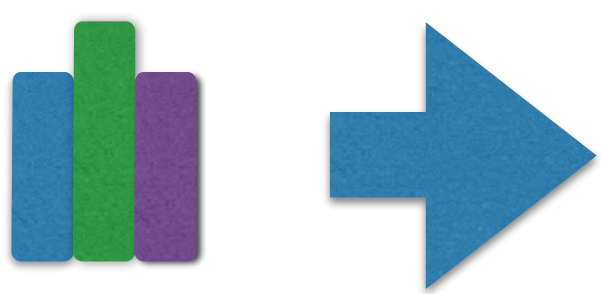
[{a,4}, {b,3}, {c,2}]



“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"

Bob?

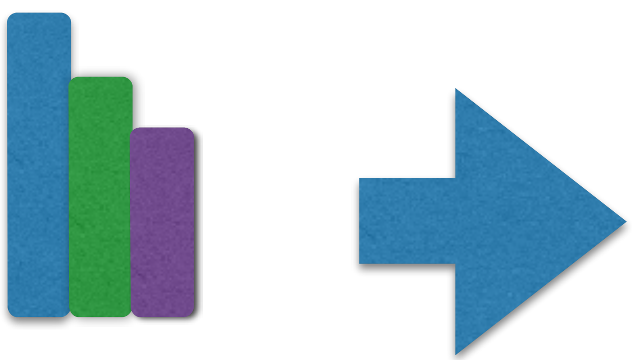
B

1429533662000 → "Sue"

Temporal vs Logical

A

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

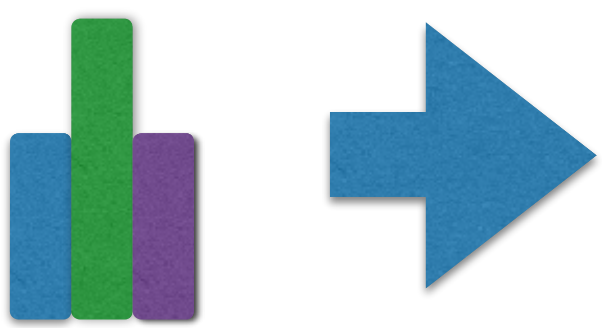


"Bob"

?

B

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

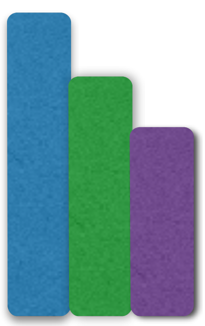


"Sue"

Temporal vs Logical

A

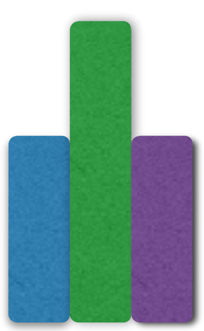
[{a,4}, {b,3}, {c,2}]



“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"

Sue?

B

1429533664001 → "Sue"

Summary

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

History Repeating

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

Terms

- Local value
- Incoming value
- Local clock
- Incoming clock

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.

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.

Client VClock

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

Client VClock

- If local clock descends incoming clock
  - discard incoming value

Client VClock

- If local and incoming clocks are concurrent
  - merge clocks
  - store incoming value as a sibling

Client VClock

- Client reads merged clock + sibling values
  - sends new value + clock
  - new clock dominates old
  - Store single value

Client VClock

- What Level of Consistency Do We Require?

Client Side IDs
Bad

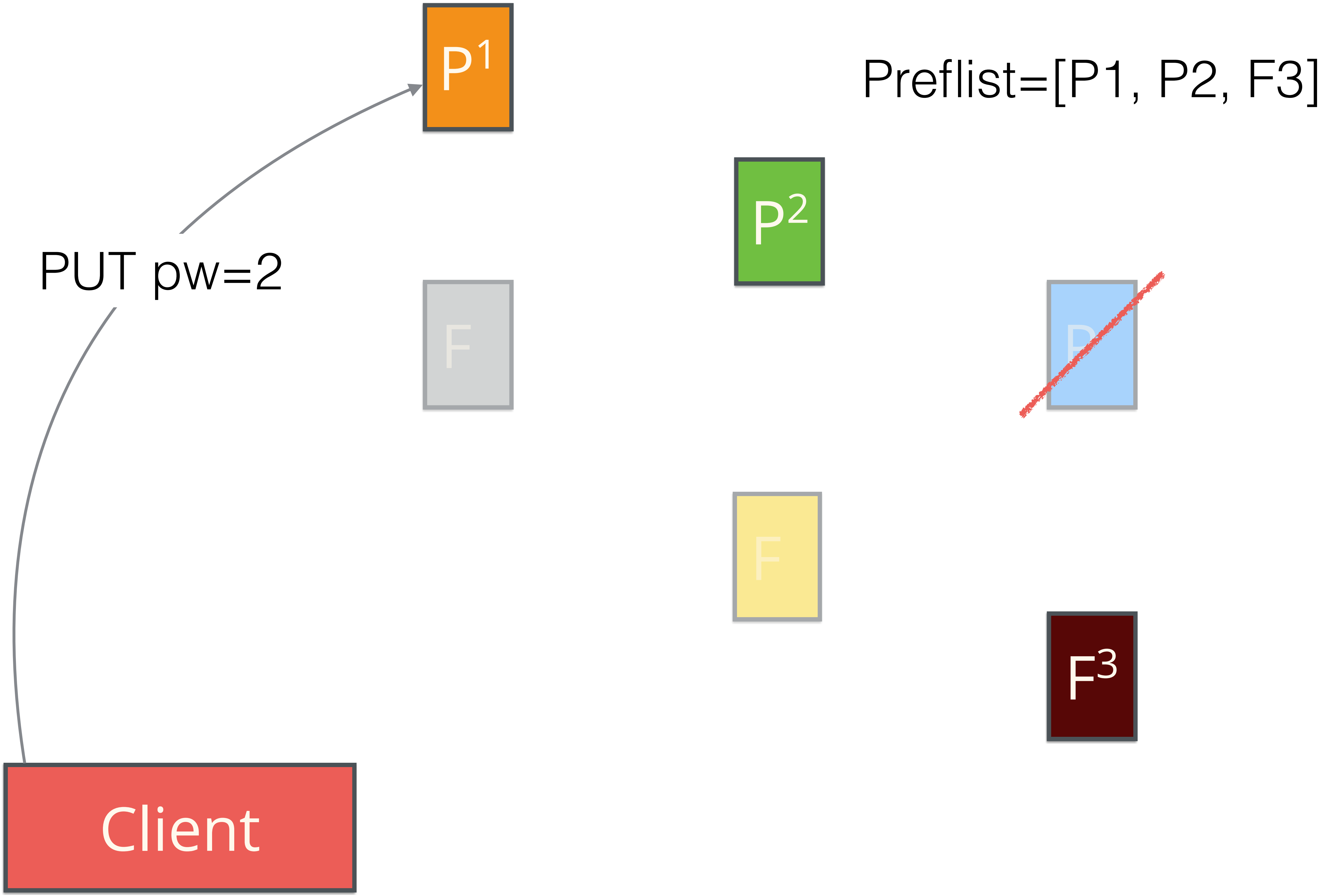
- Unique actor ID:: database invariant enforced by client!
- Actor Explosion (Charron-Bost)
    - No. Entries == No. Actors
- Client Burden
- RYOW required

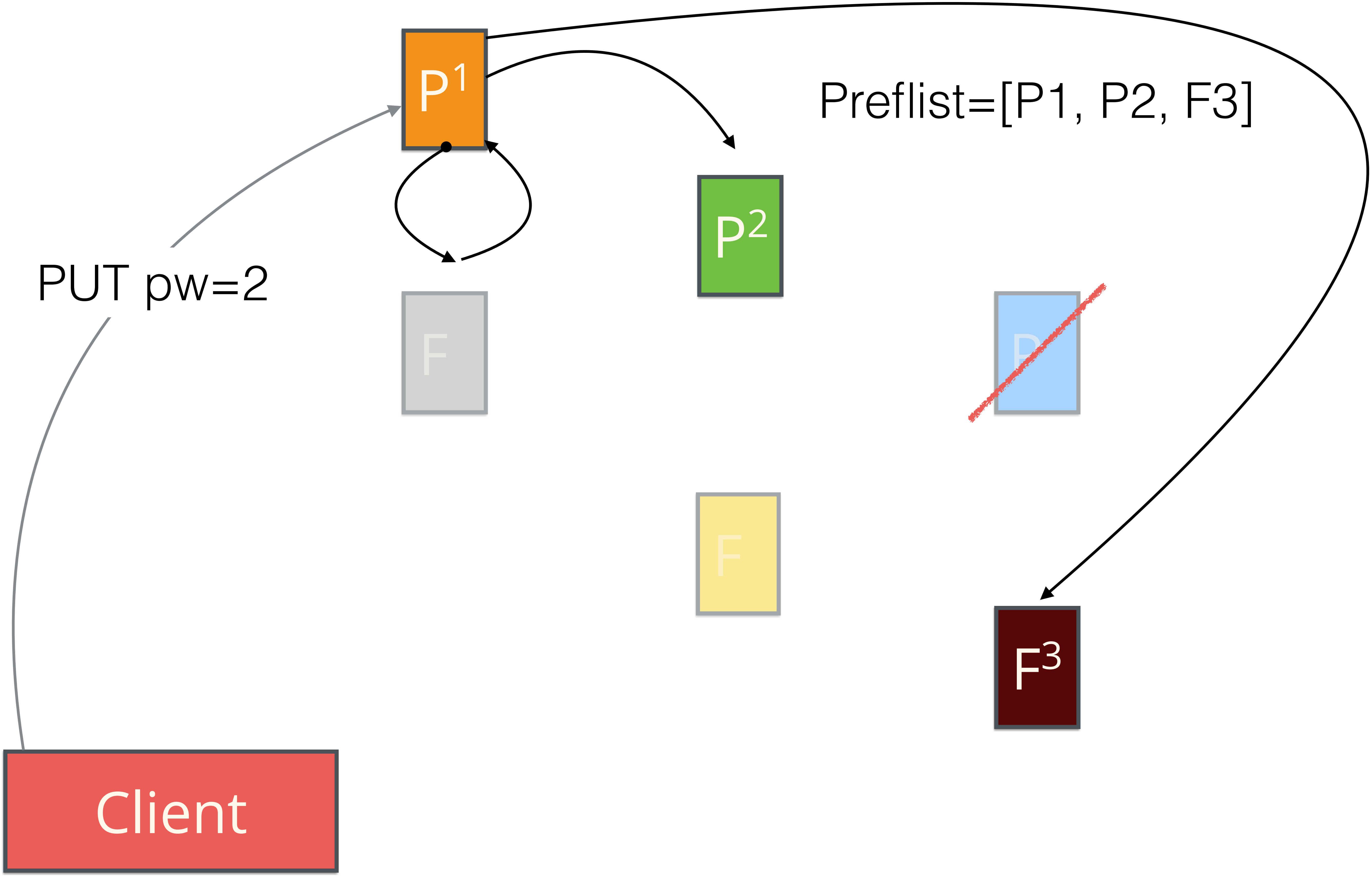
RYOW

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

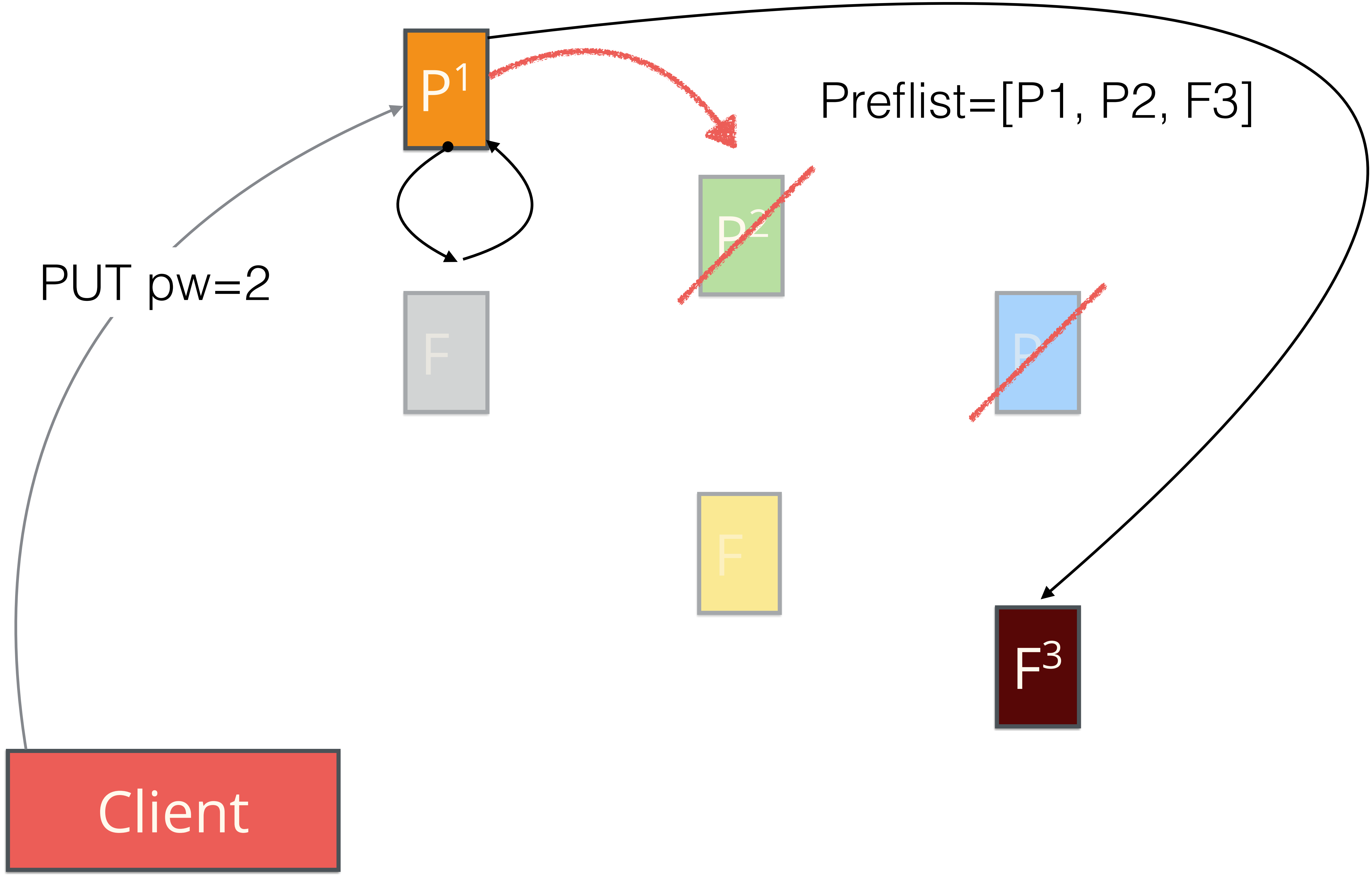
When P is F

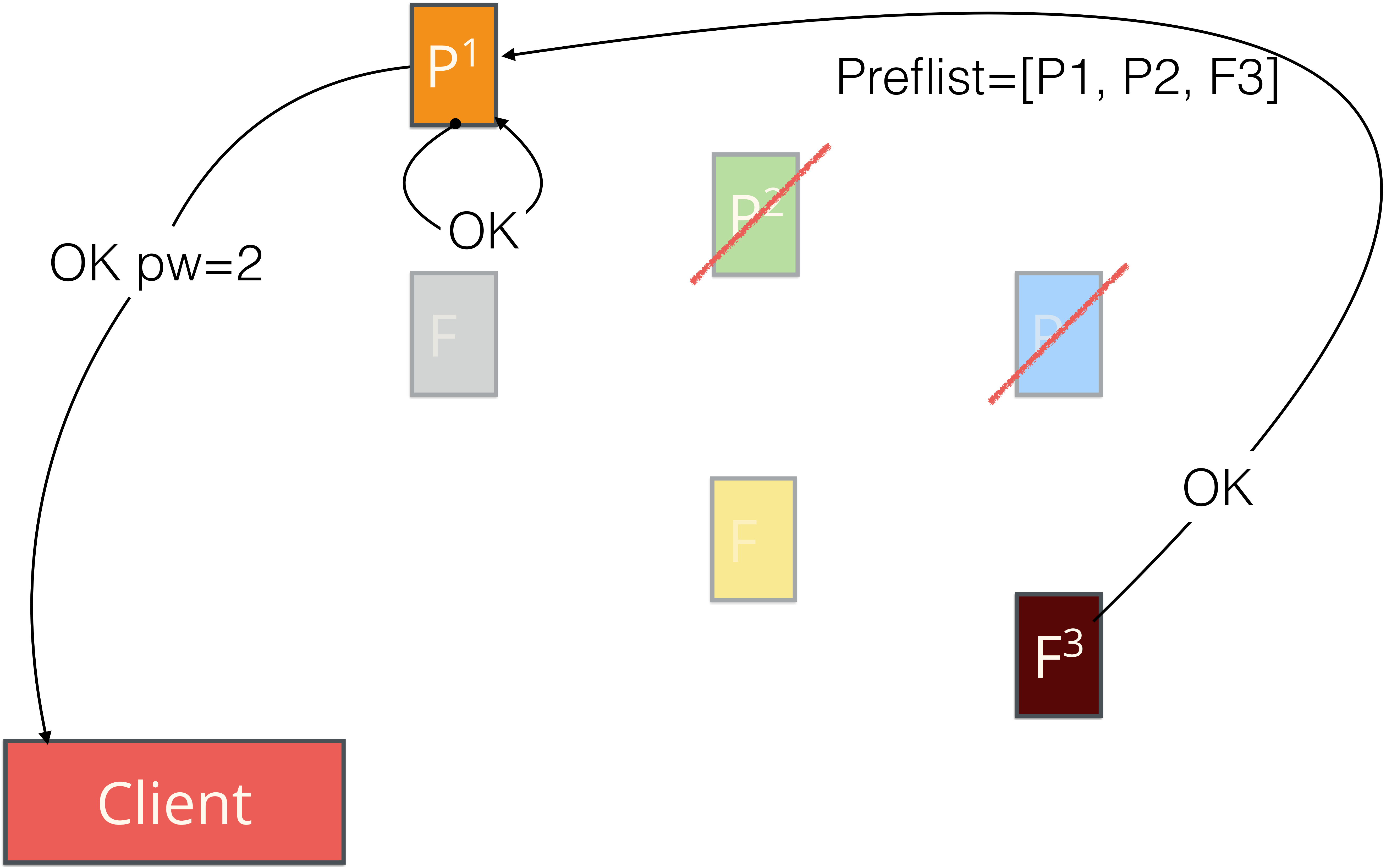
- Get preflist
- count primaries
- Send request to N
- Don’t check responder status!


Preflist=[P1, P2, F3]
P1
P2
F
F
F3
PUT pw=2
Client

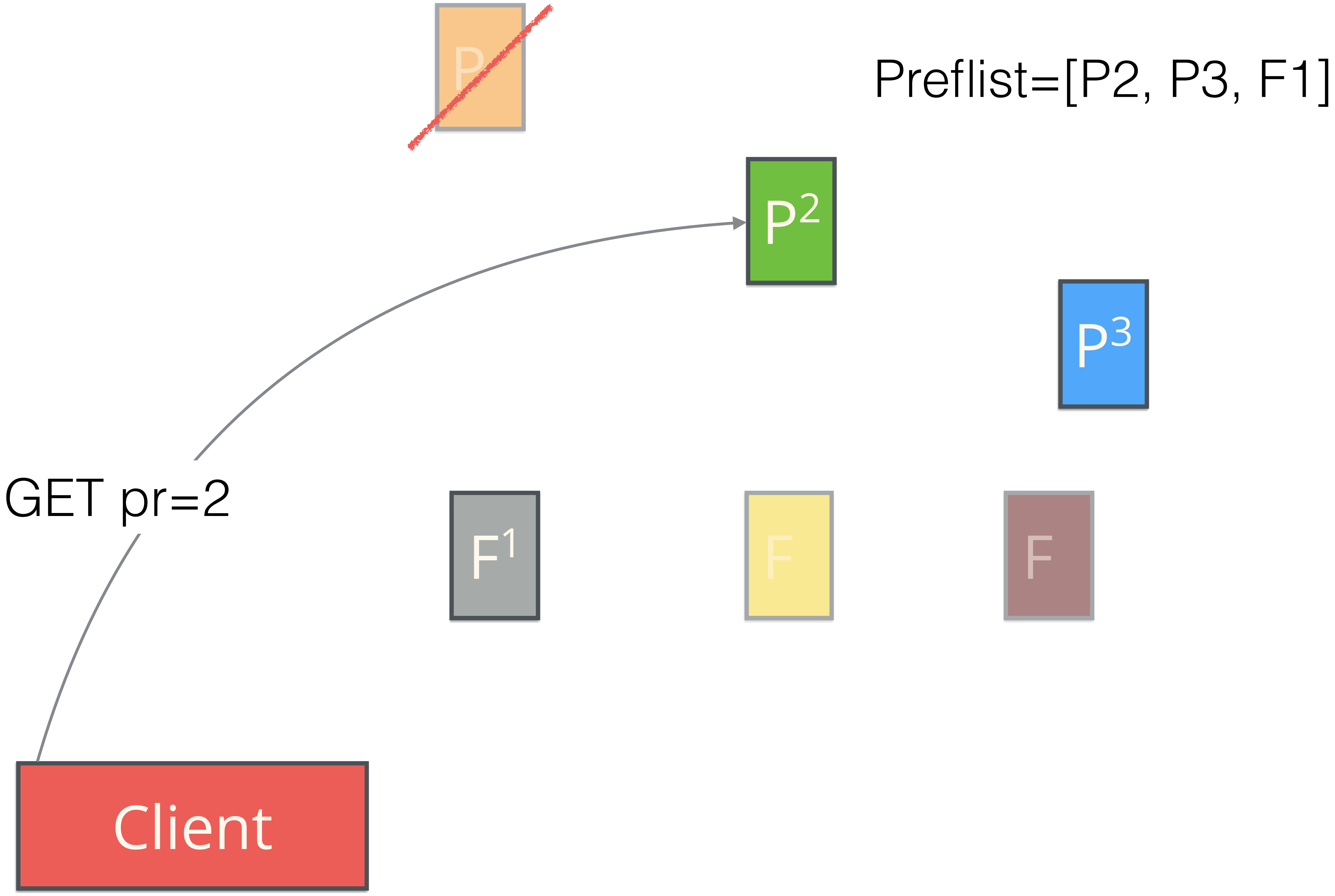


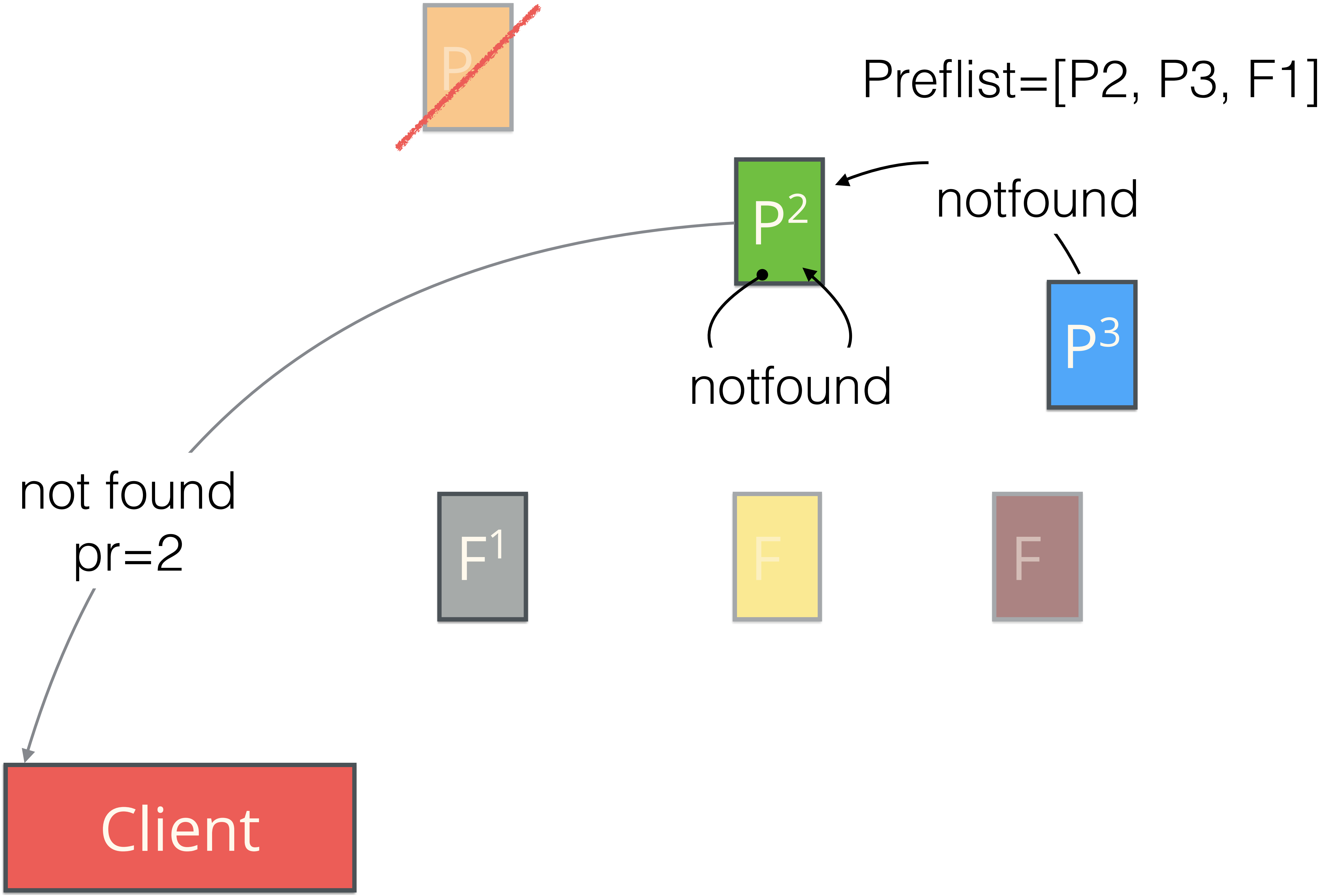
P1
Preflist=[P1, P2, F3]
P2
PUT pw=2
F
P
F
F3
Client


P1
Preflist=[P1, P2, F3]
P2
PUT pw=2
F
P
F
F3
Client



P1
Preflist=[P1, P2, F3]
OK
OK pw=2
P2
F
P
OK
F
F3
Client


P
Preflist=[P2, P3, F1]
P2
P3
GET pr=2
F1
F
F
Client



P
Preflist=[P2, P3, F1]
P2
notfound
P3
notfound
not found
pr=2
F1
F
F
Client

Client VClock

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

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}]

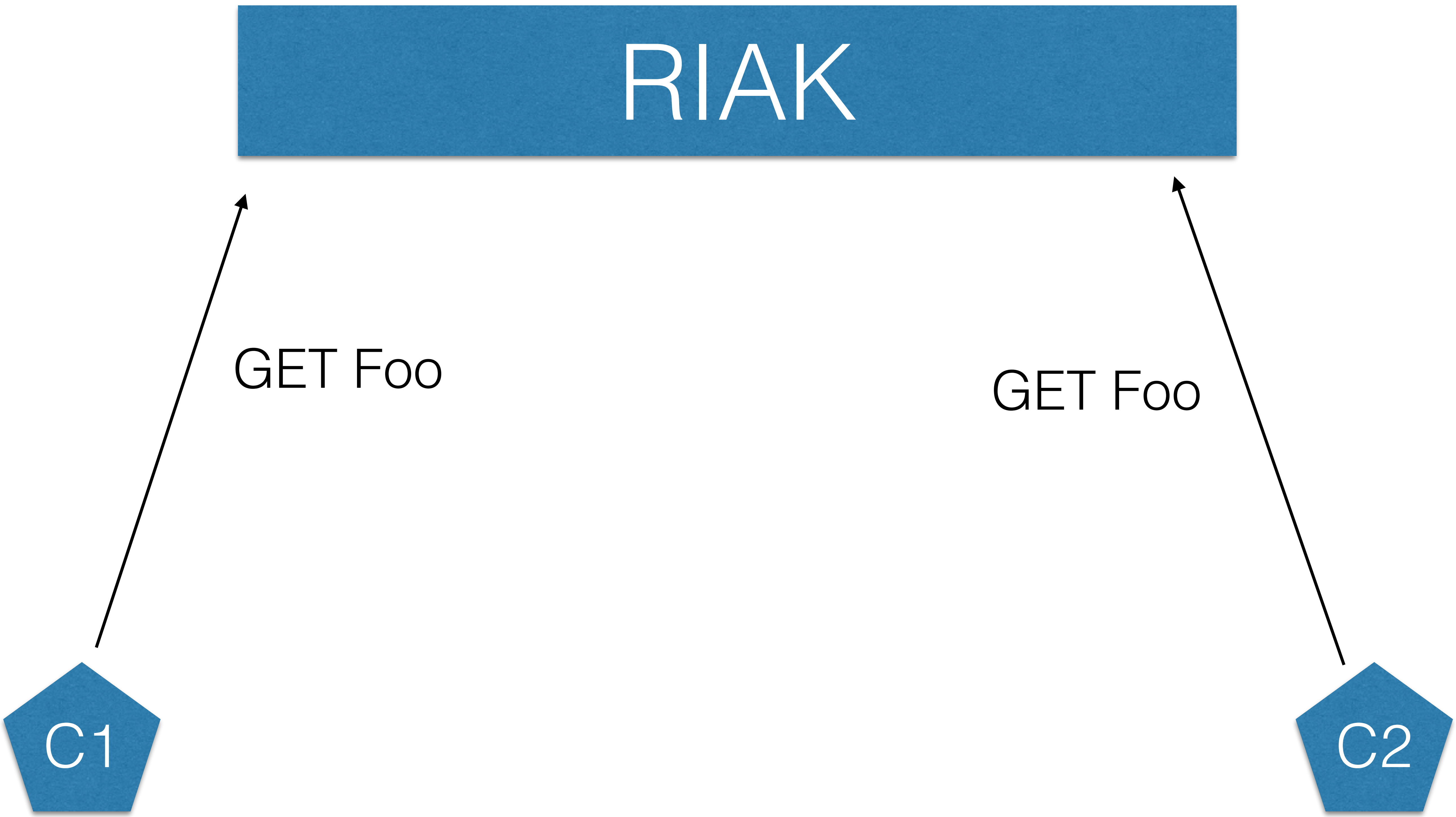
Client Side ID
RYOW

- 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

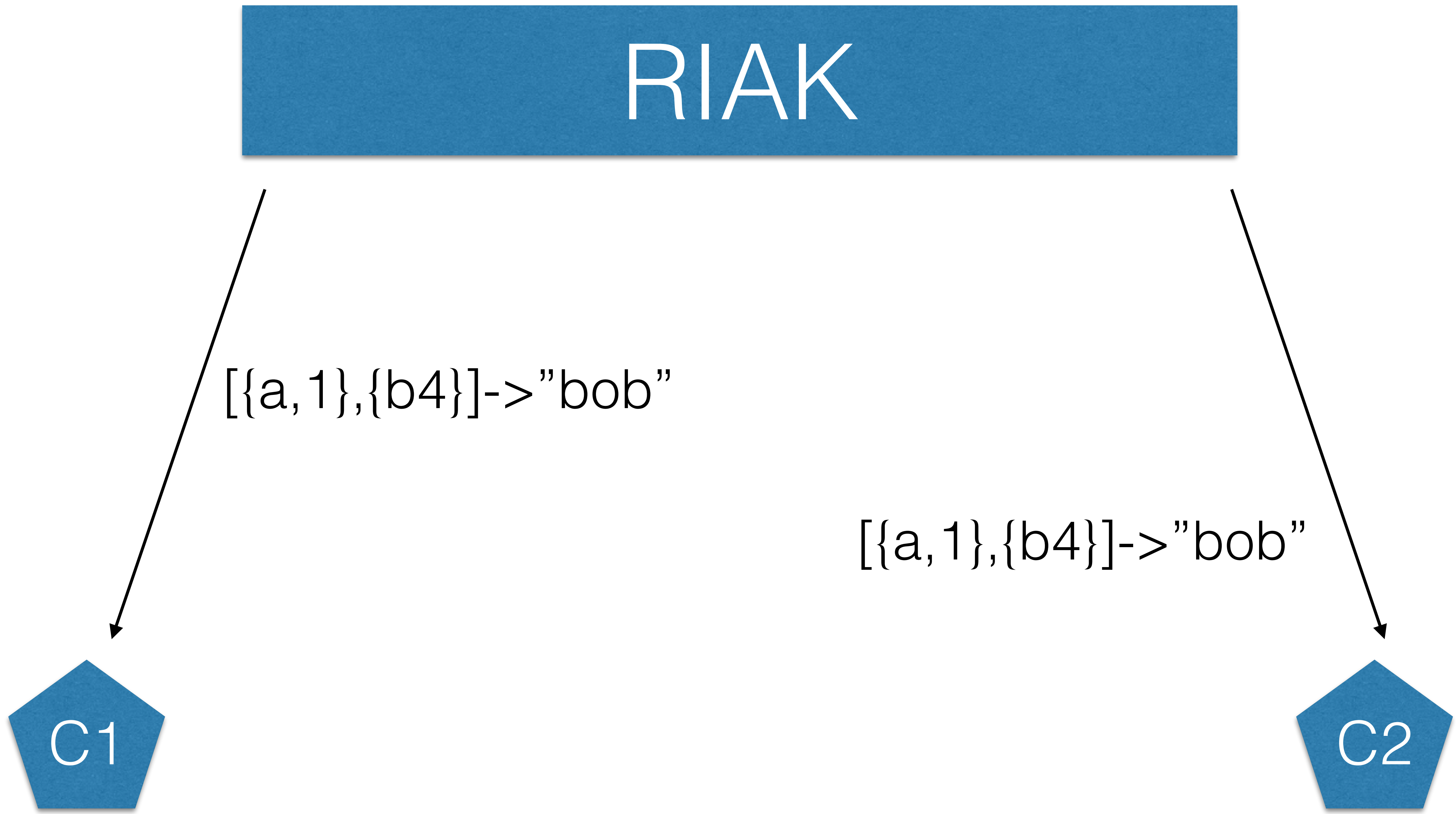
Vnode VClocks
Riak 1.n

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

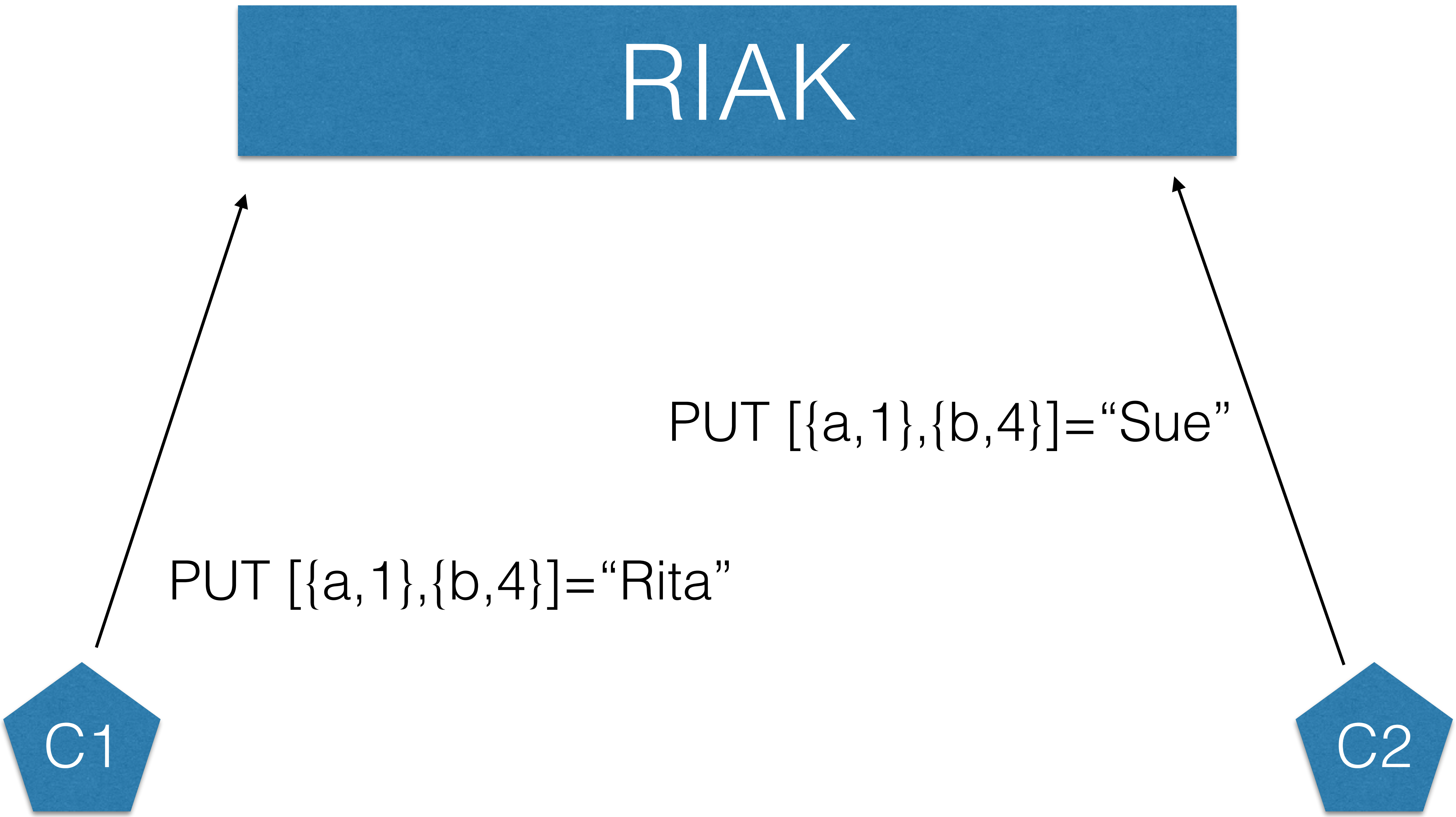
Vnode VClocks
False Concurrency



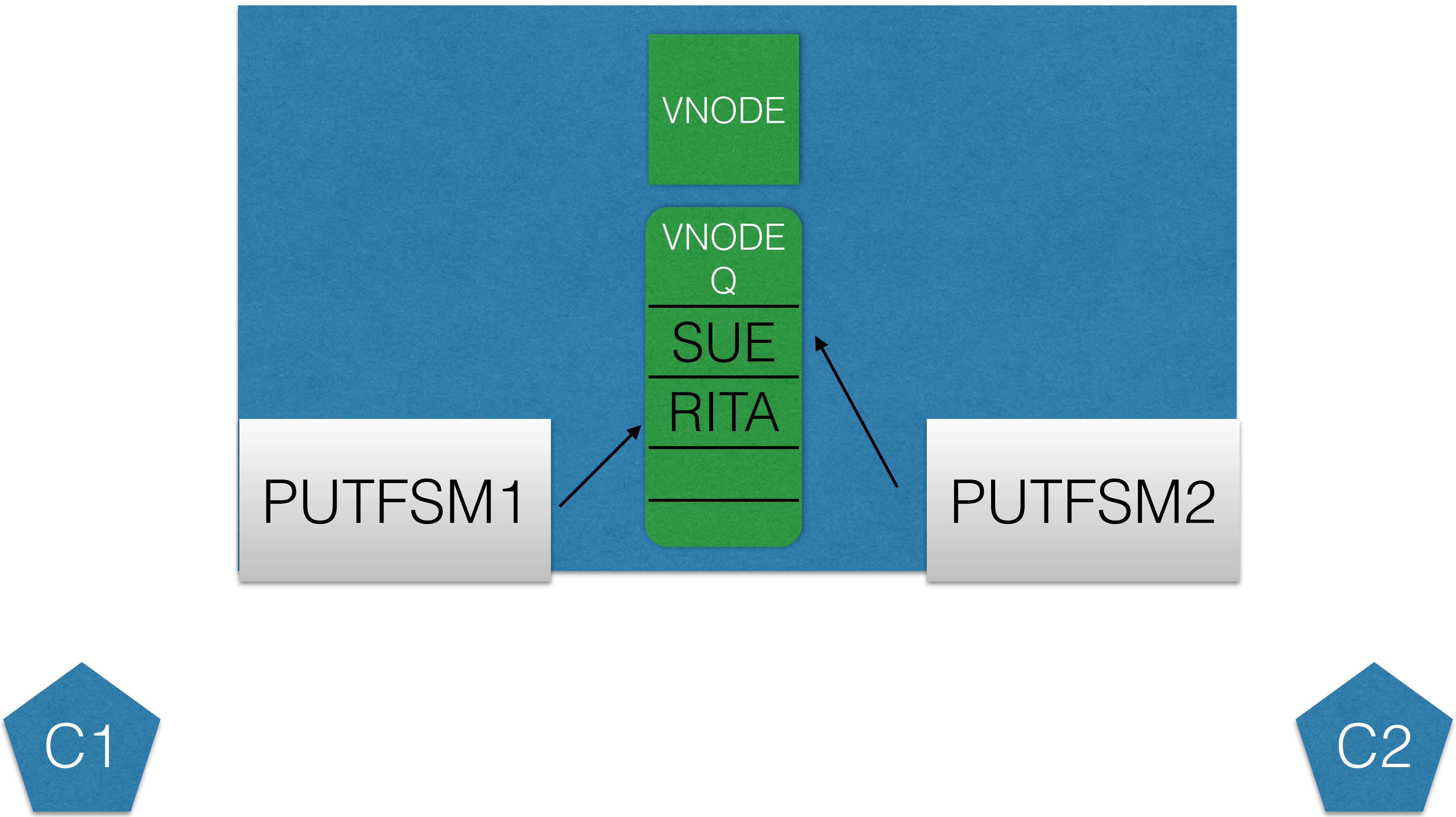
Vnode VClocks False Concurrency



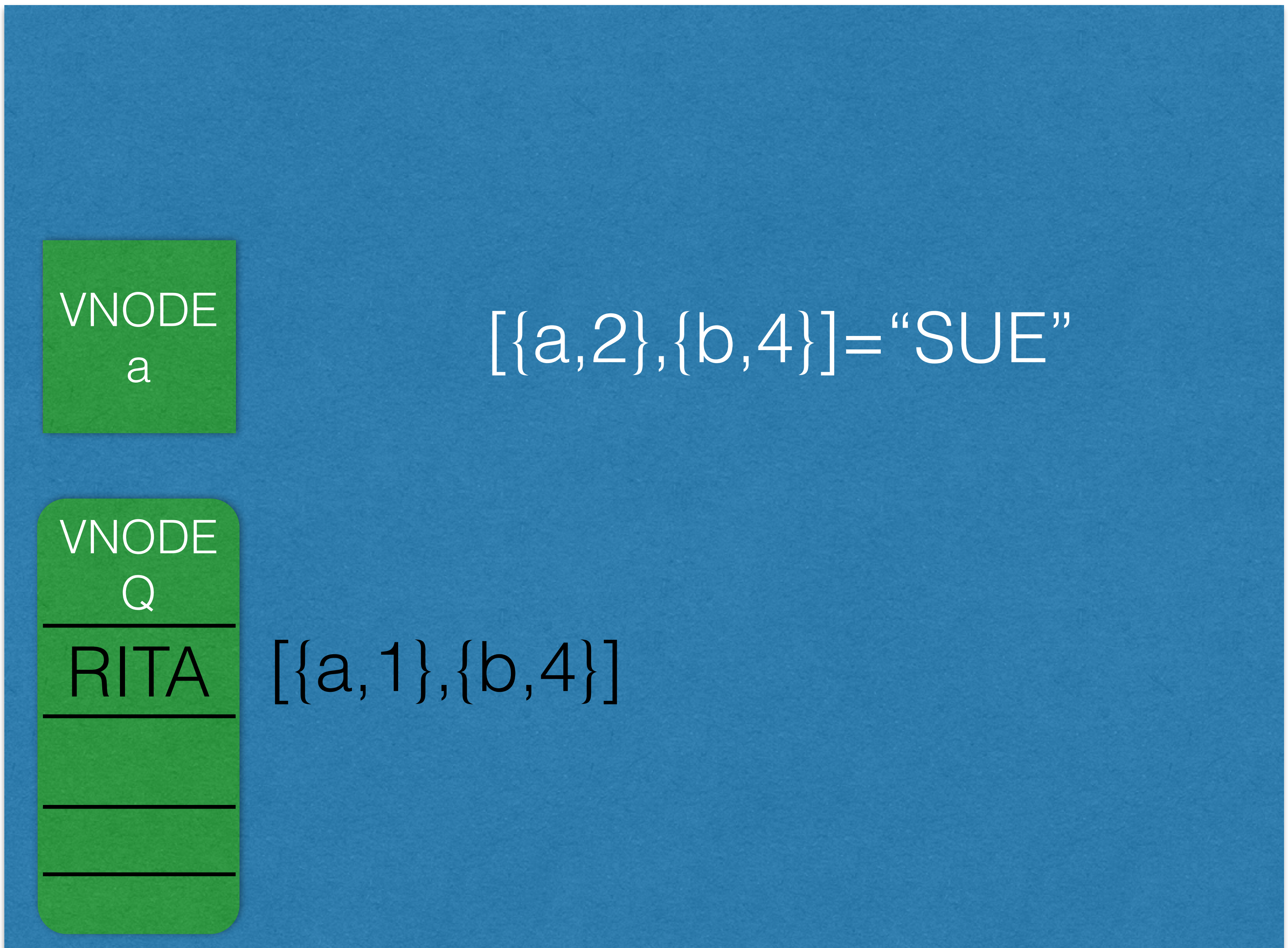
Vnode VClocks False Concurrency



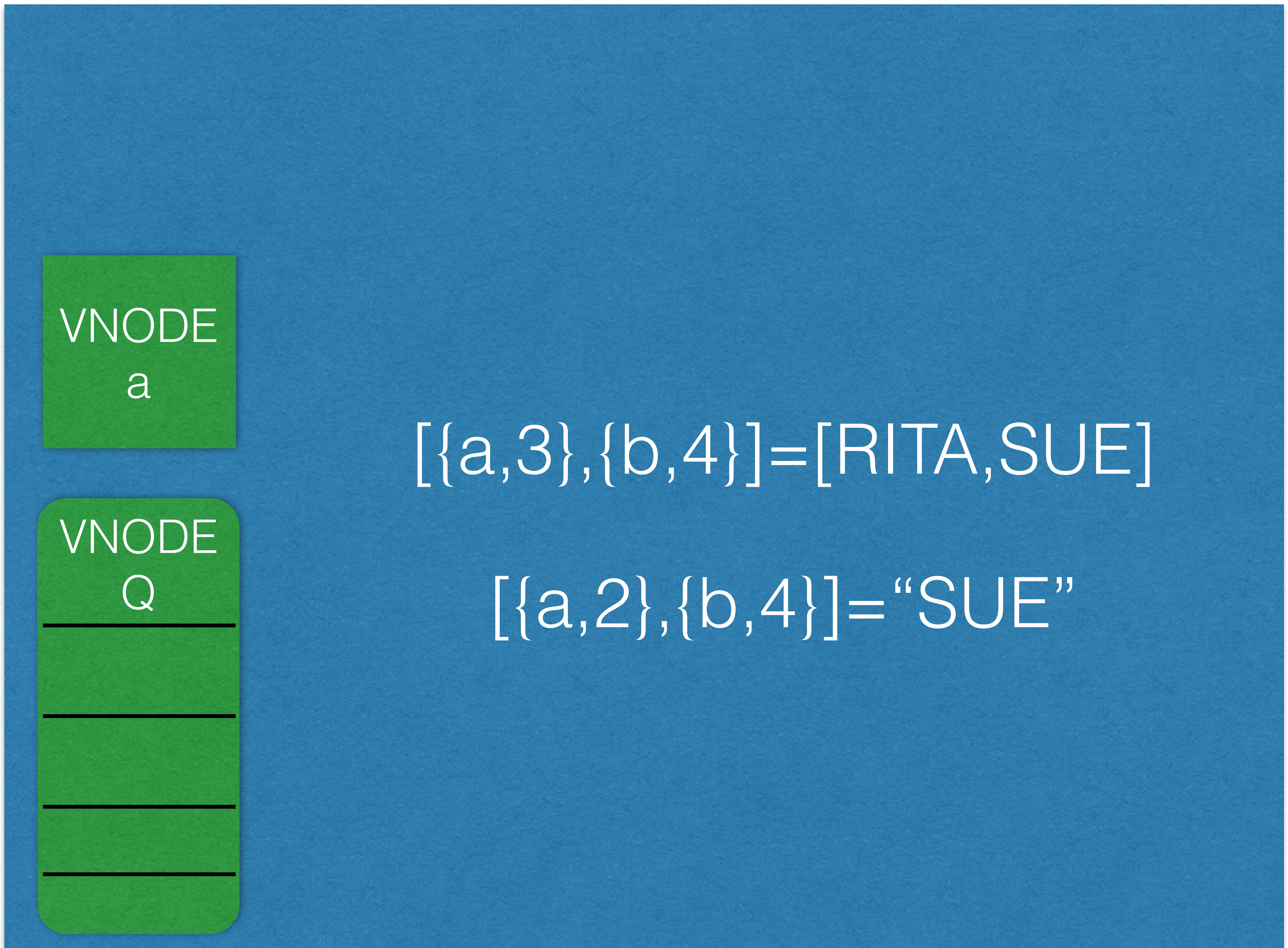
Vnode VClocks
False Concurrency



Vnode VClocks
False Concurrency



Vnode VClocks
False Concurrency



Vnode VClock

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

Vnode VClock

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

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

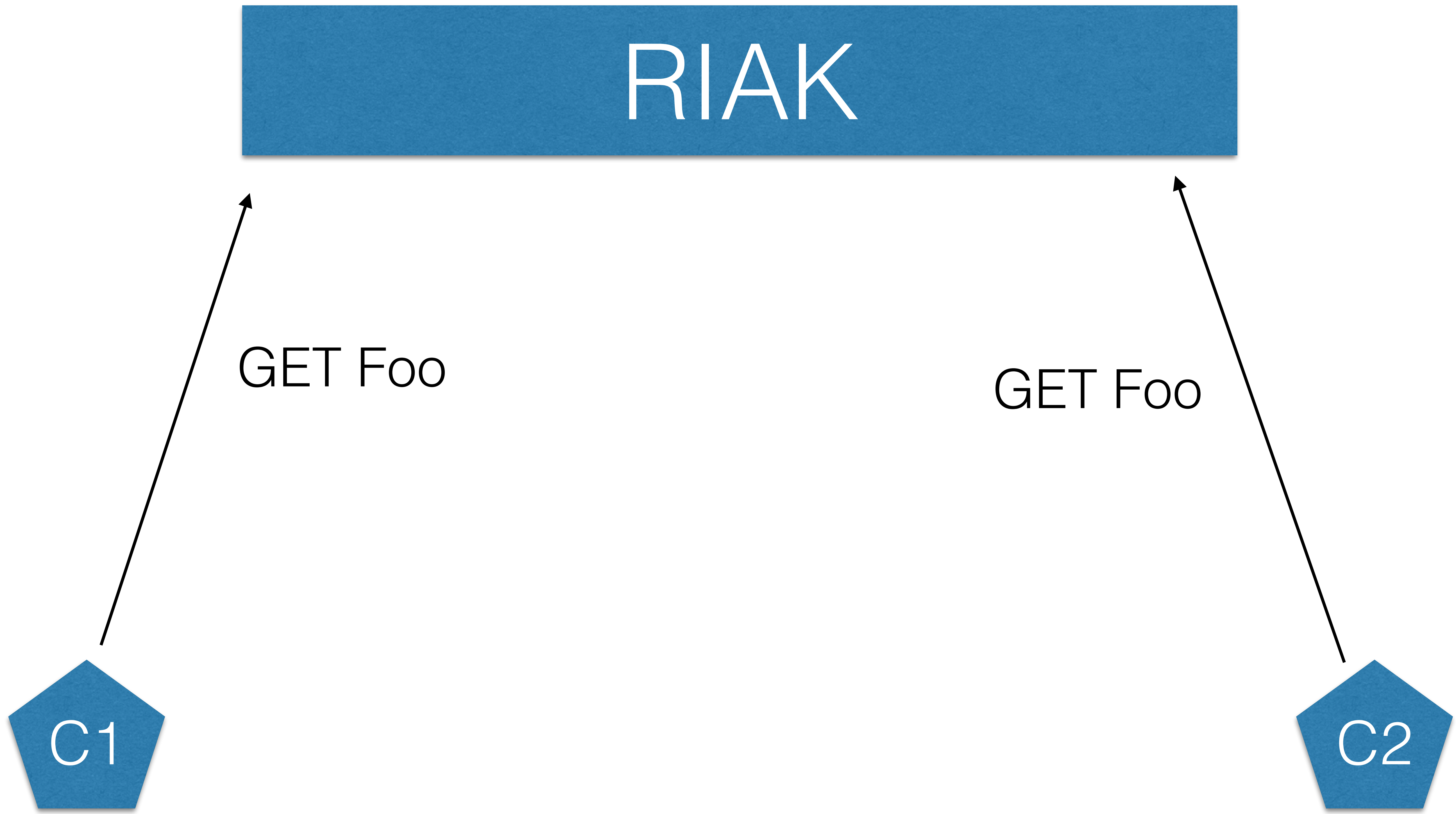
- False concurrency cost
- Many many siblings
- Large object
- Death

Sibling Explosion

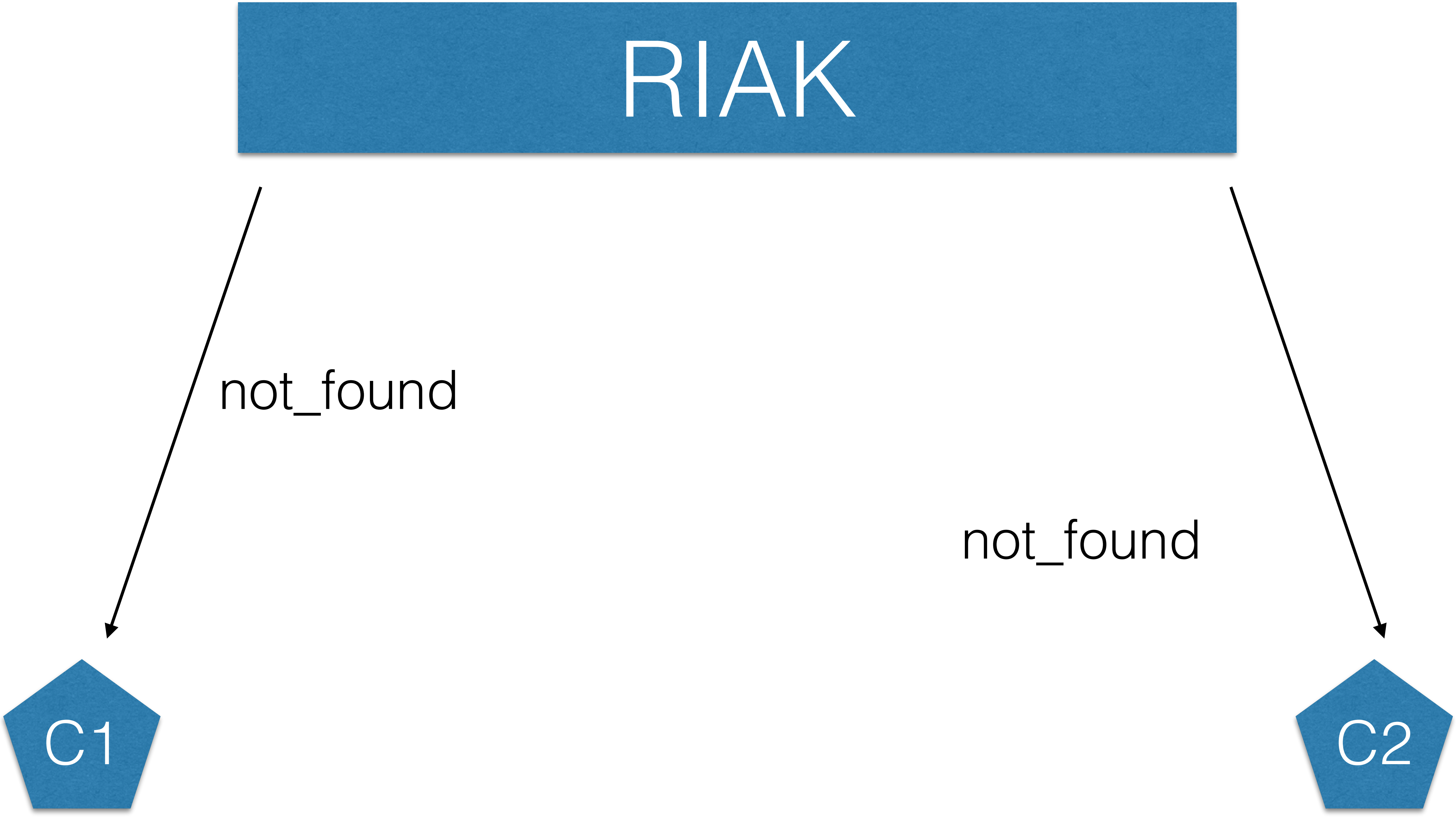
- Data structure
  - Clock + Set of Values
- False Concurrency

Sibling Explosion

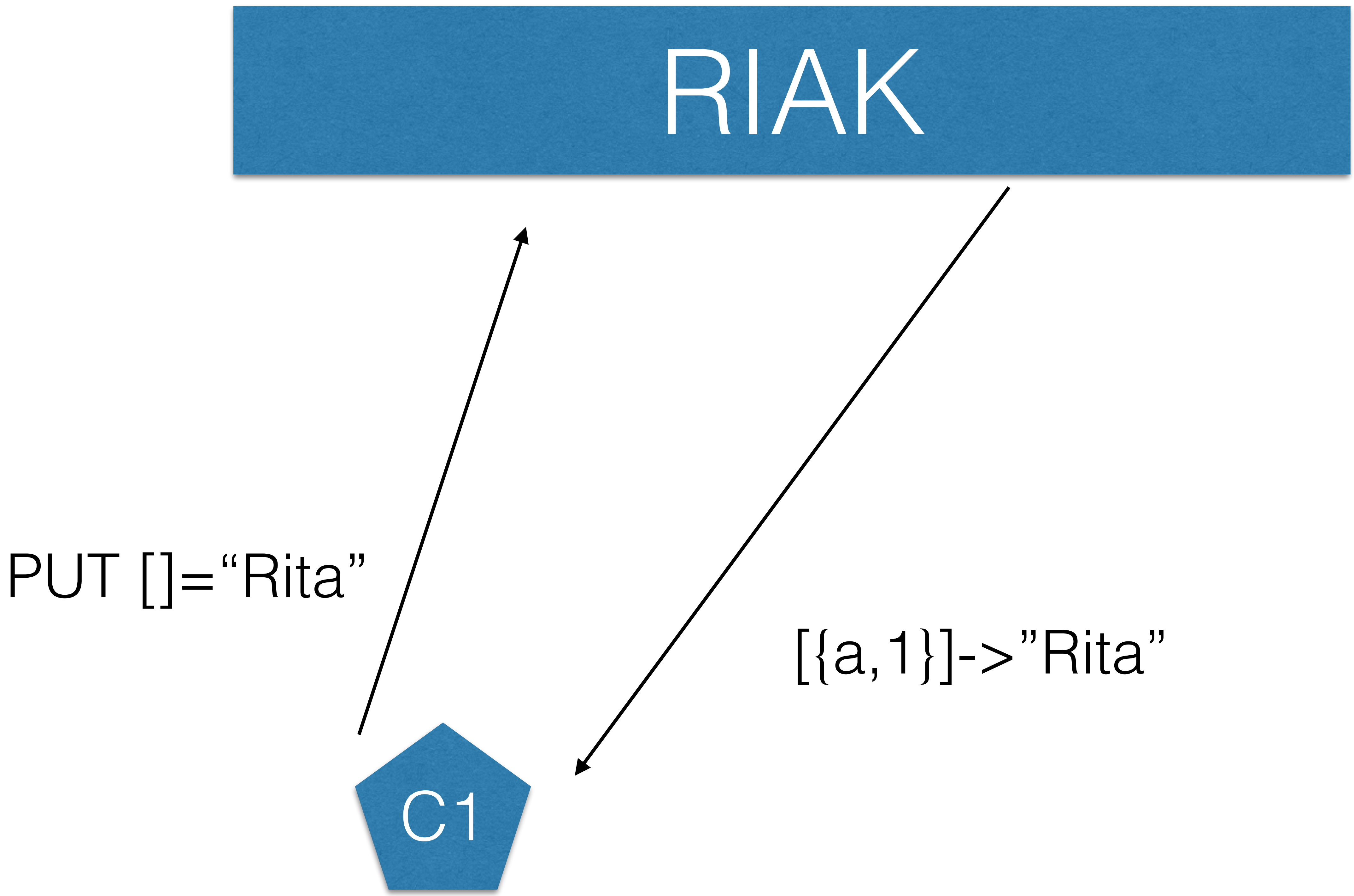
Sibling Explosion



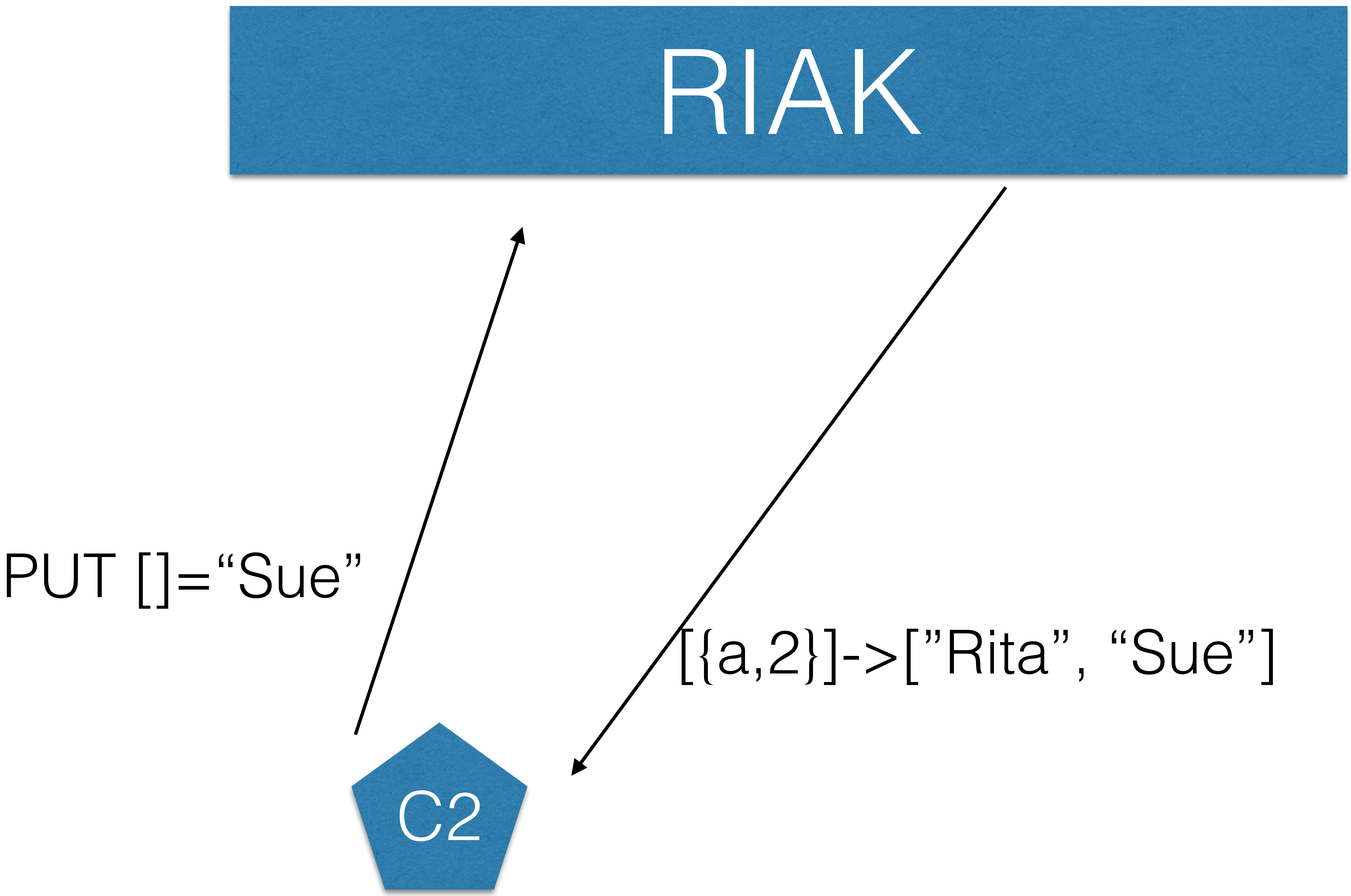
Sibling Explosion



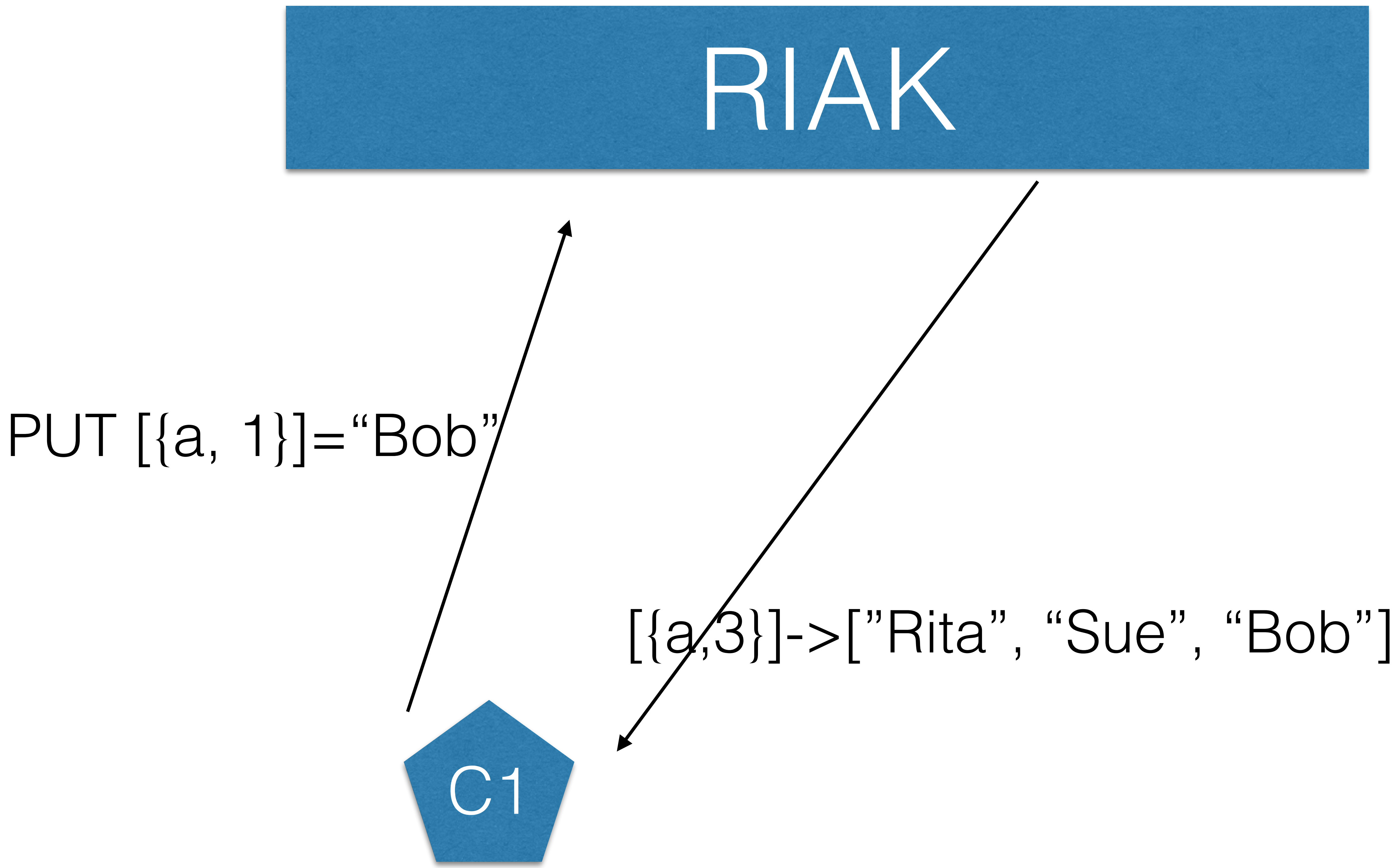
Sibling Explosion



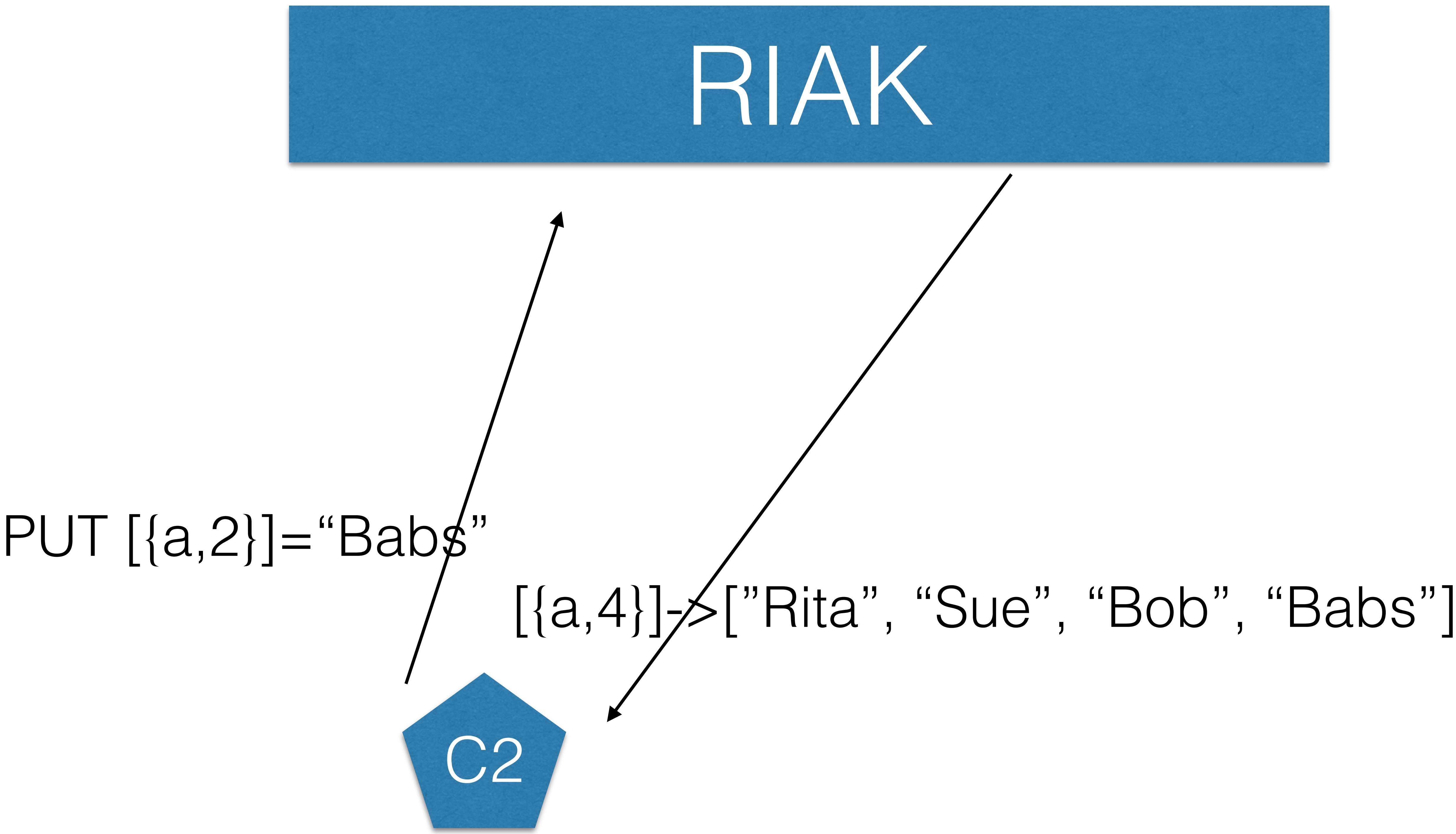
Sibling Explosion



Sibling Explosion



Sibling Explosion



Vnode VClock

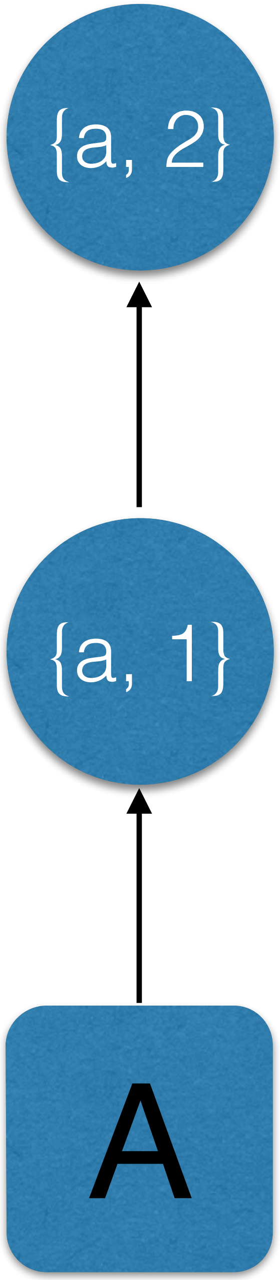
- Trick to “dodge’ the Charron-Bost result
- Engineering, not academic
- Tested (quickchecked in fact!)
- Action at a distance

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

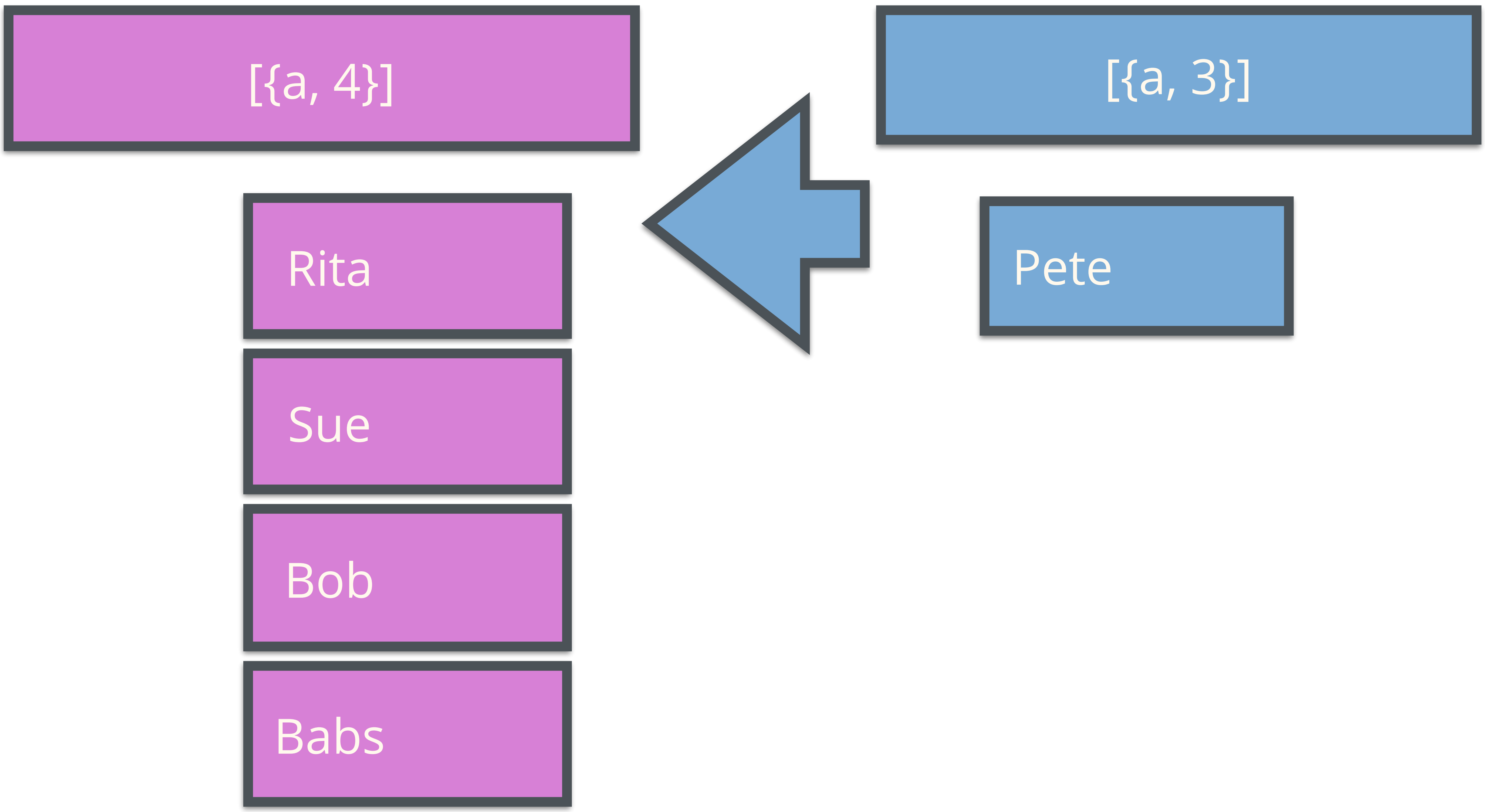
Vnode VClock

- 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

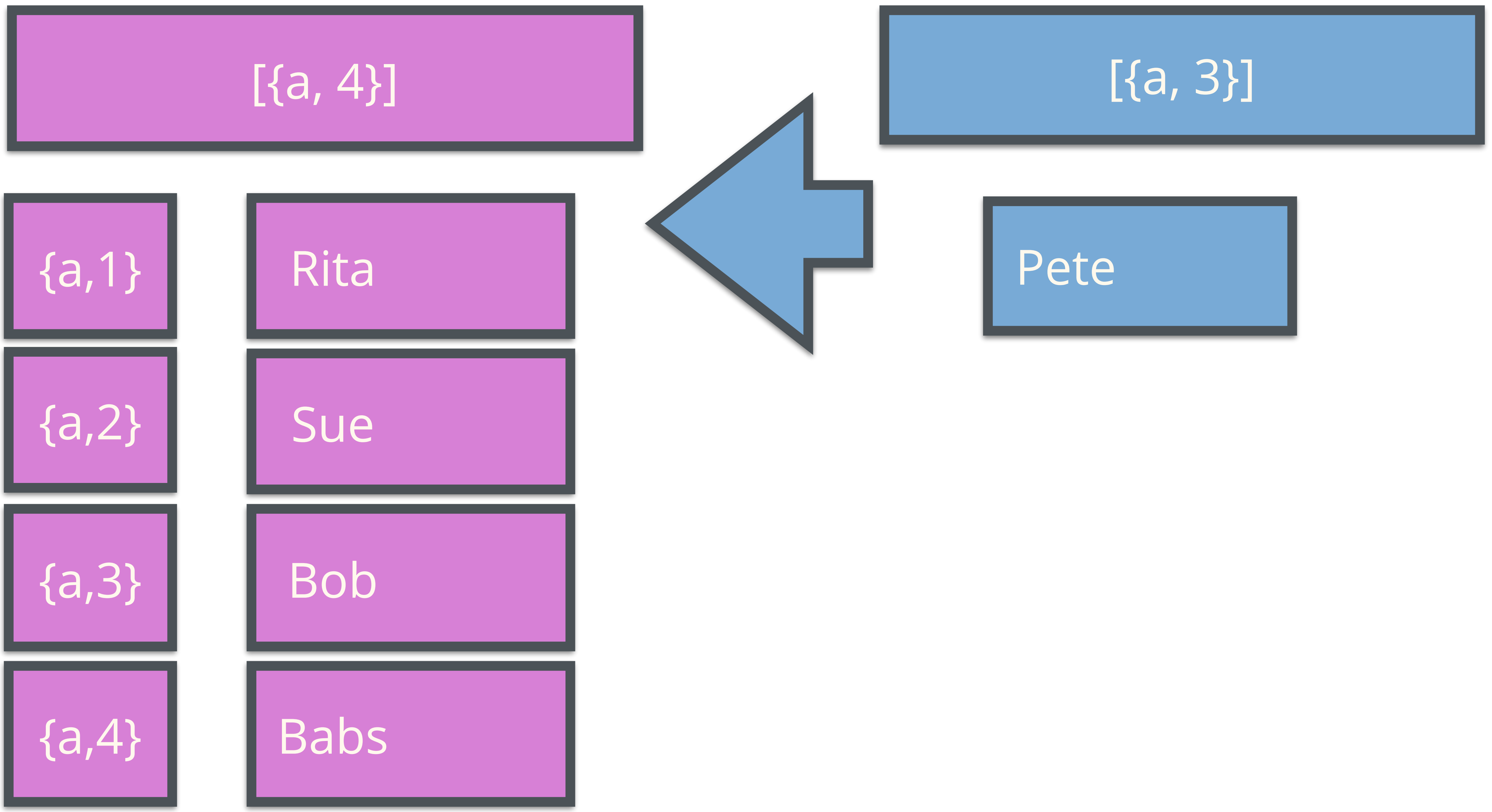
Oh drop all the dots

- Prune Siblings
  - Remove any siblings who’s dot is seen by the incoming clock
  - if Clock >= [Dot] drop Dotted value

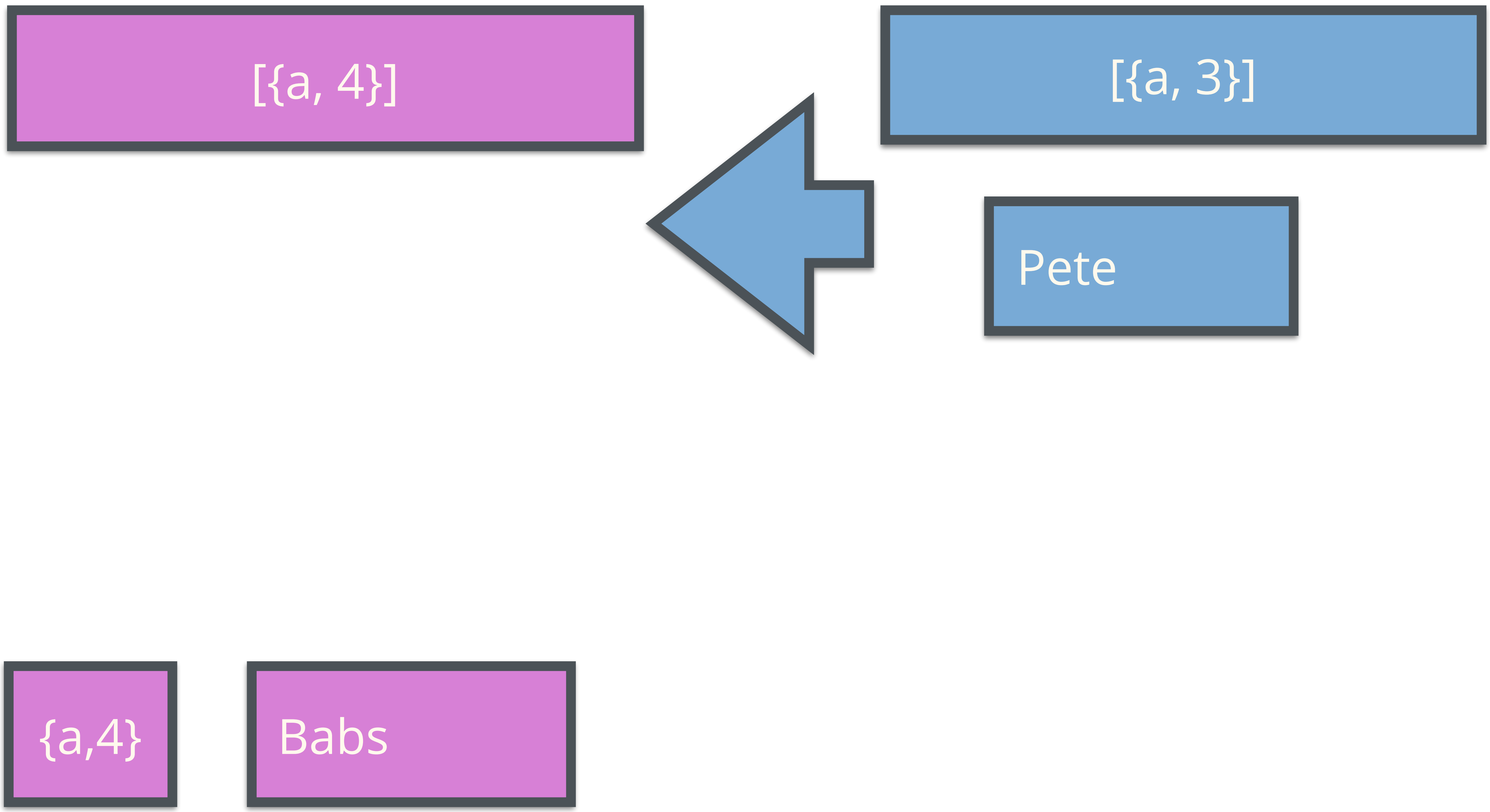
Vnode VClocks



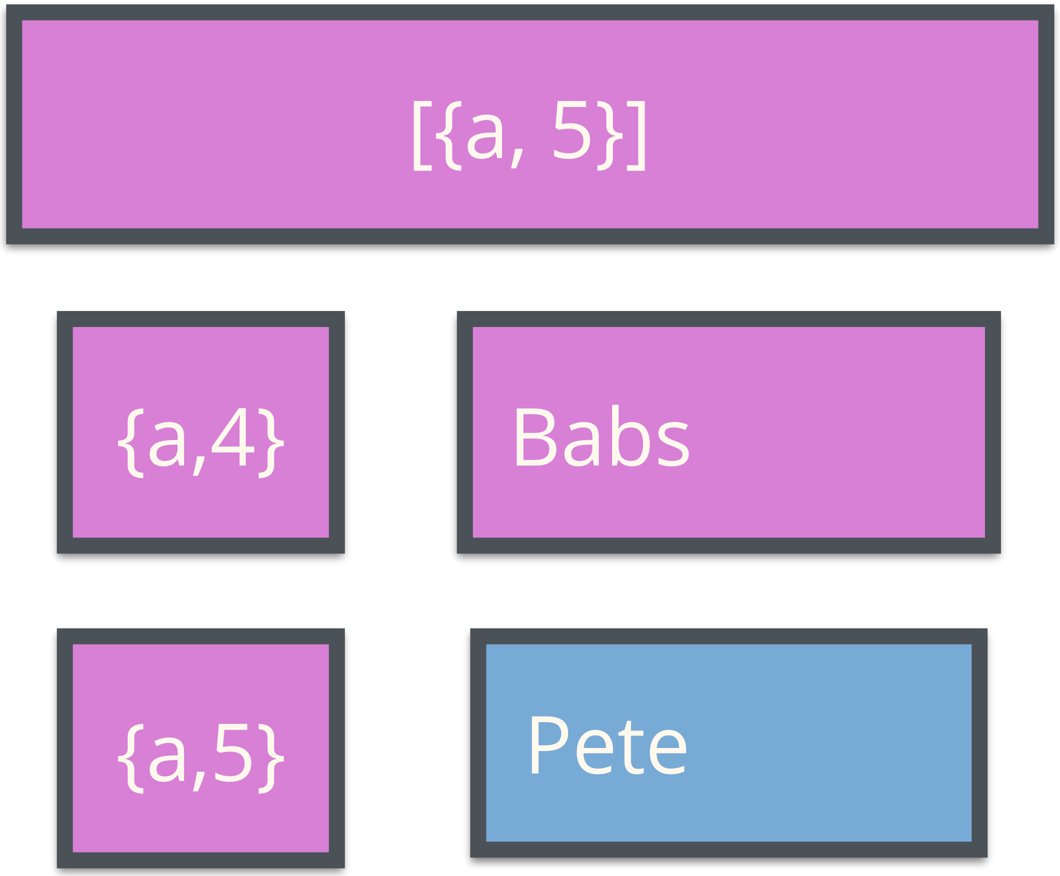
Vnode VClocks
+ Dots



Vnode VClocks
+ Dots



Vnode VClocks
+ Dots



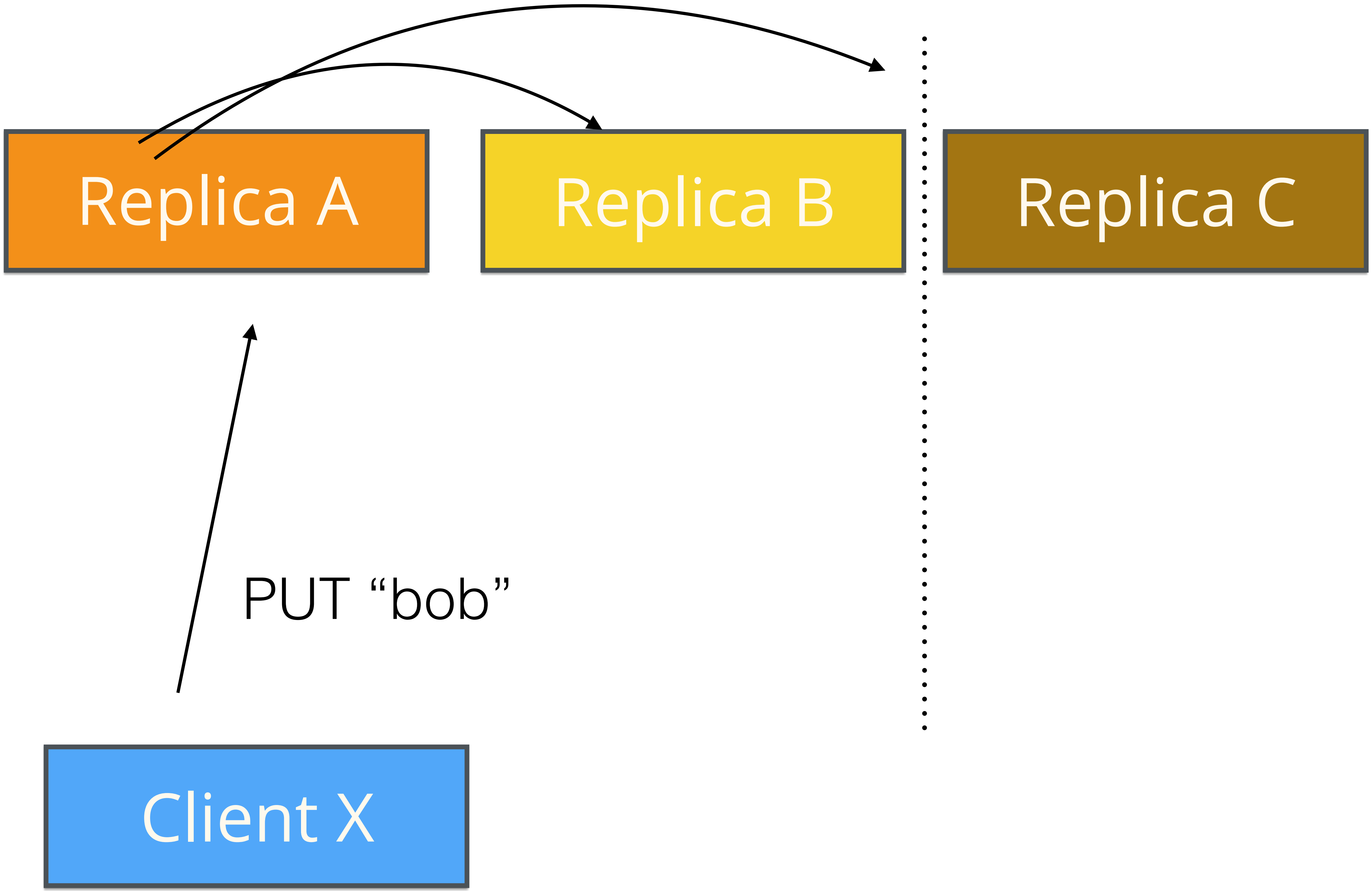
Dotted Version Vectors

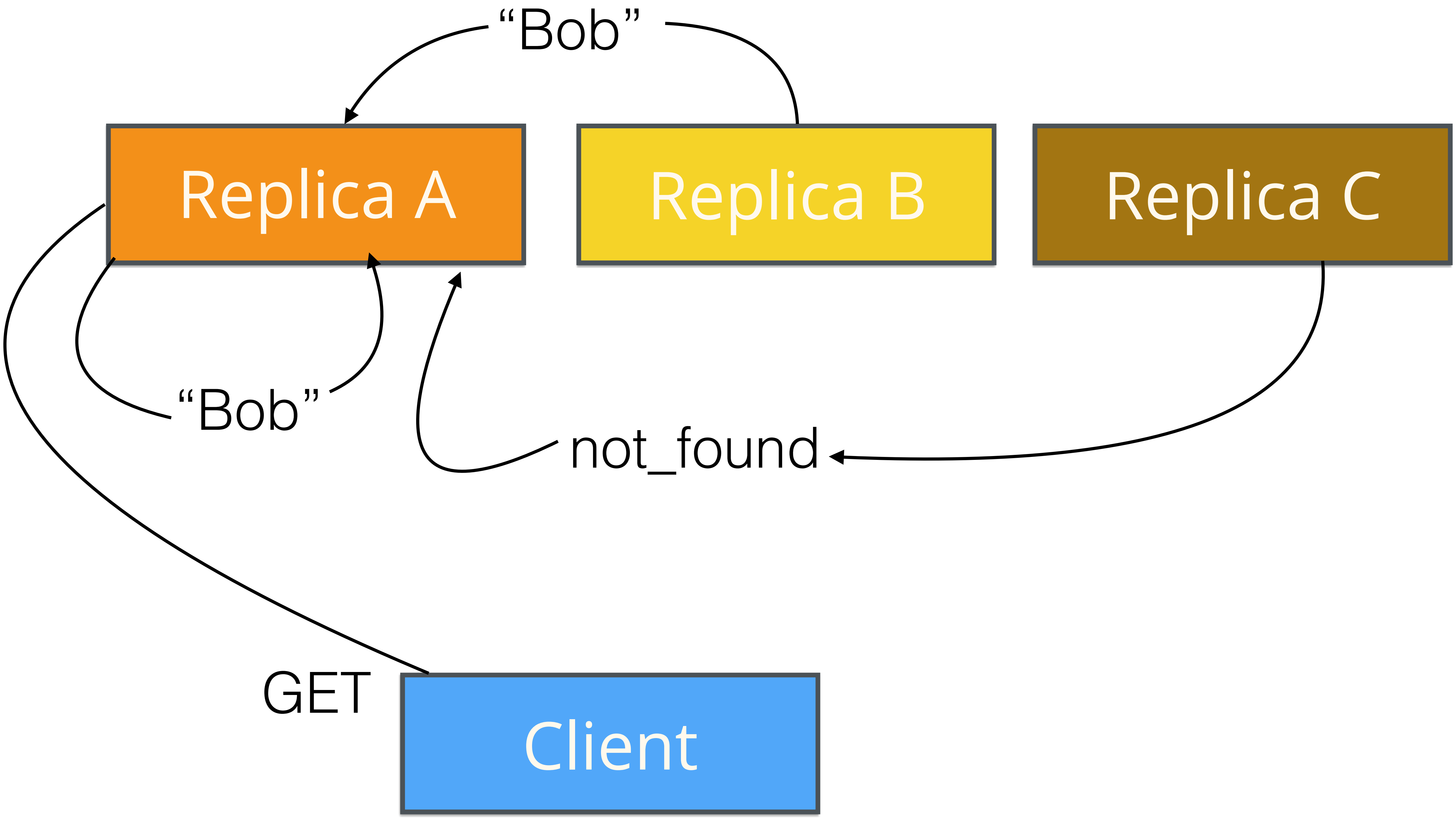
- Action at a distance
- Correctly capture concurrency
- No sibling explosion
- No Actor explosion

KV679

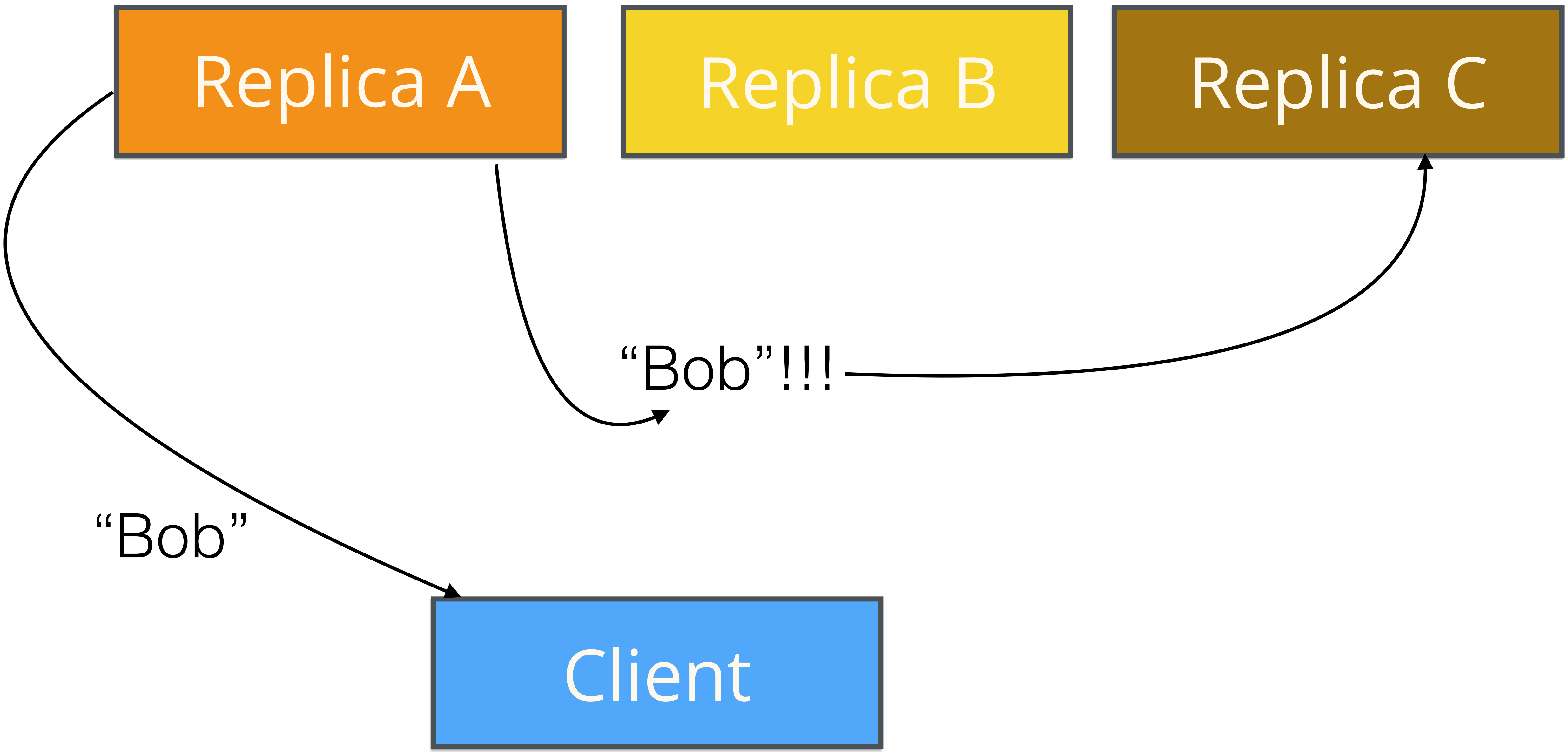
Riak Overview

Read Repair. Deletes.

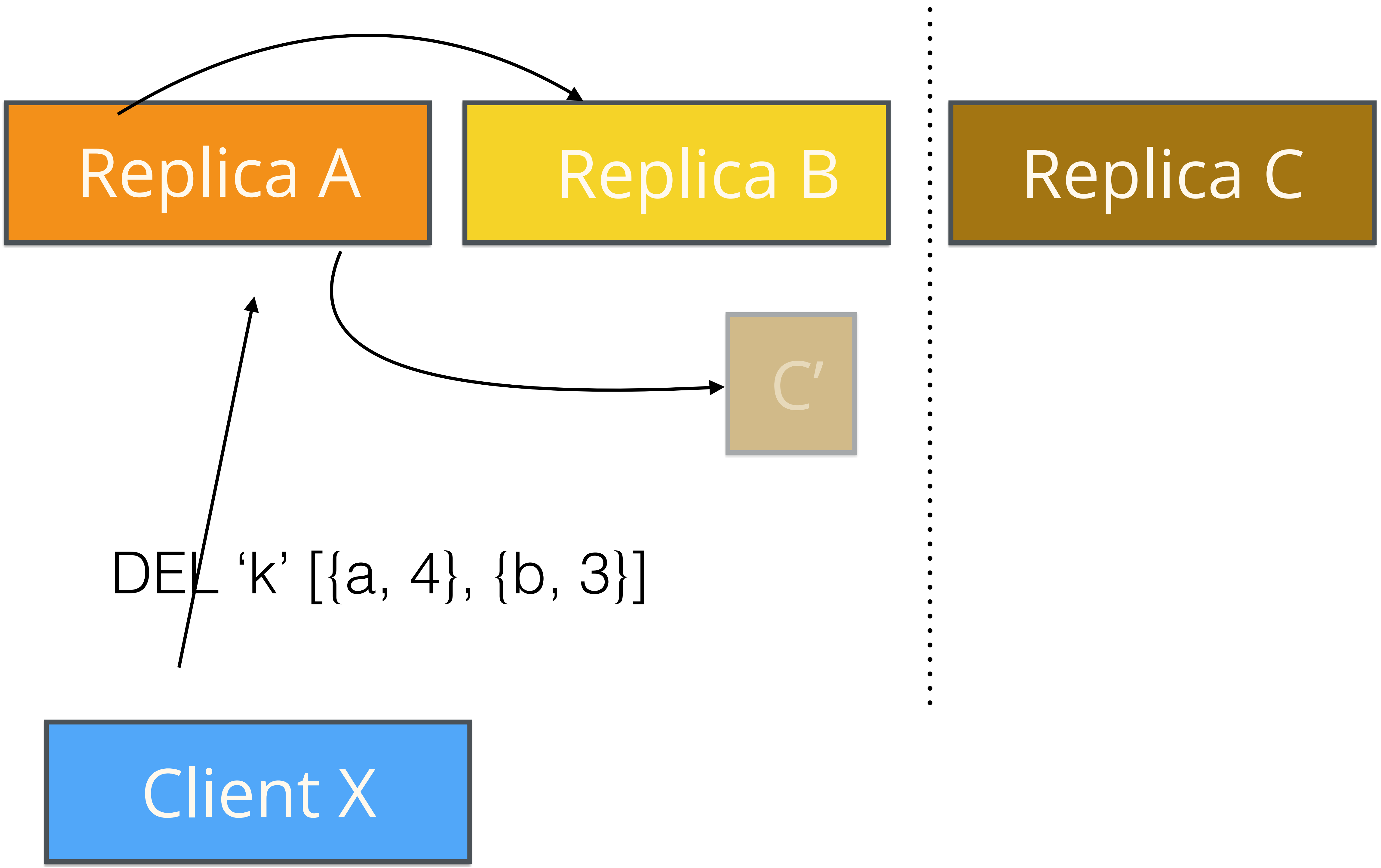

Replica A
Replica B
Replica C
PUT “bob”
Client X



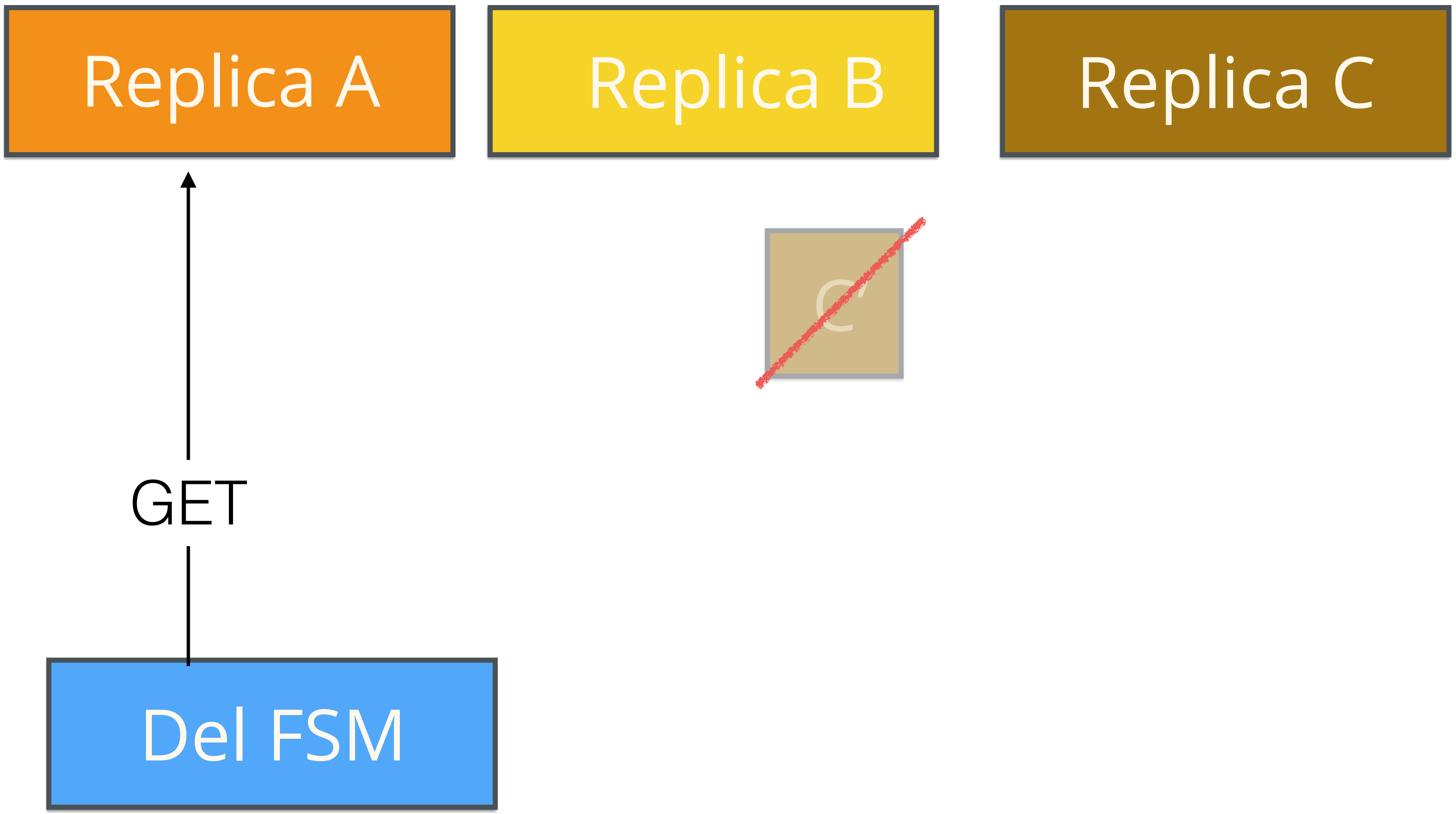
Read Repair



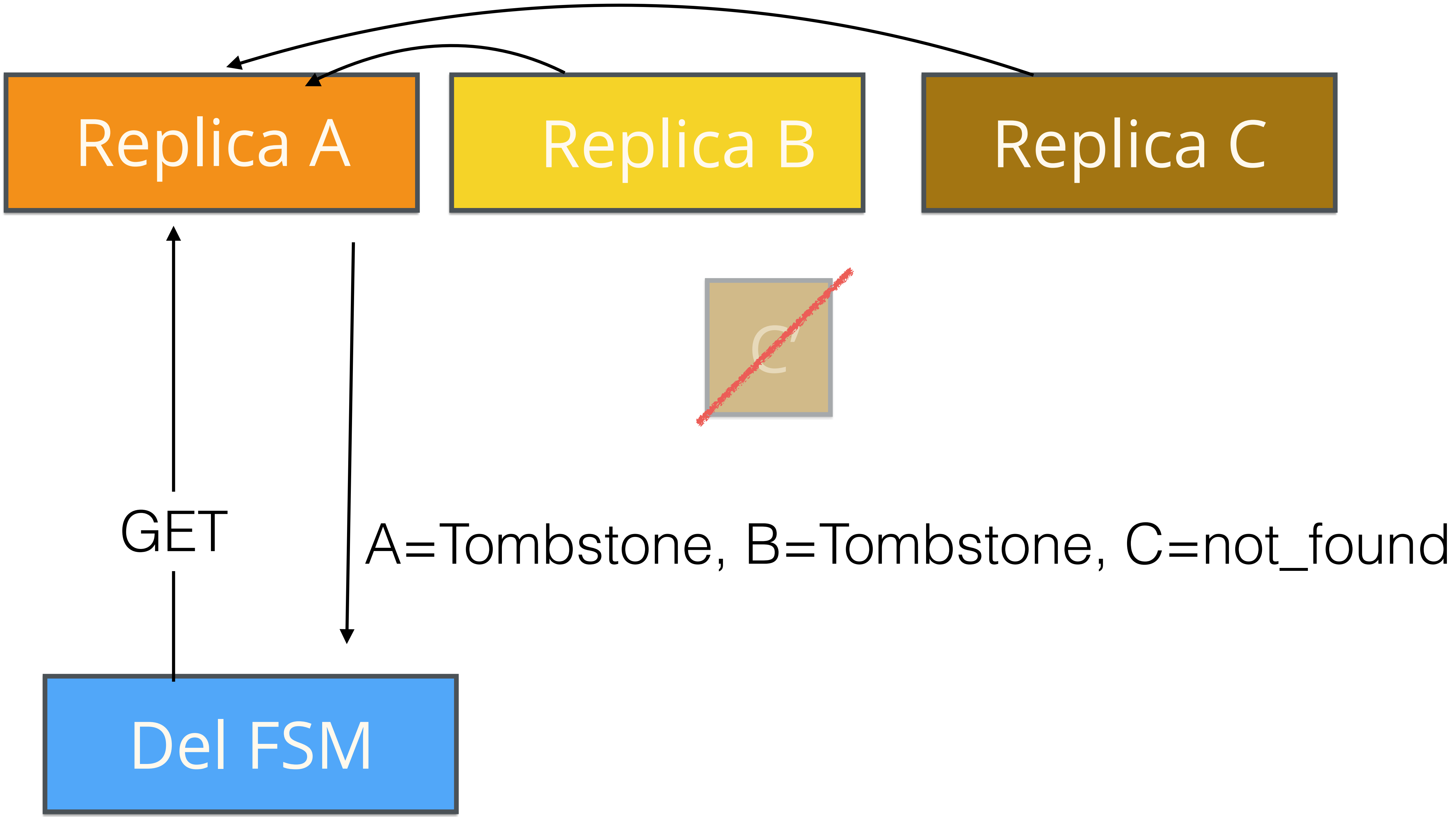
Read Repair



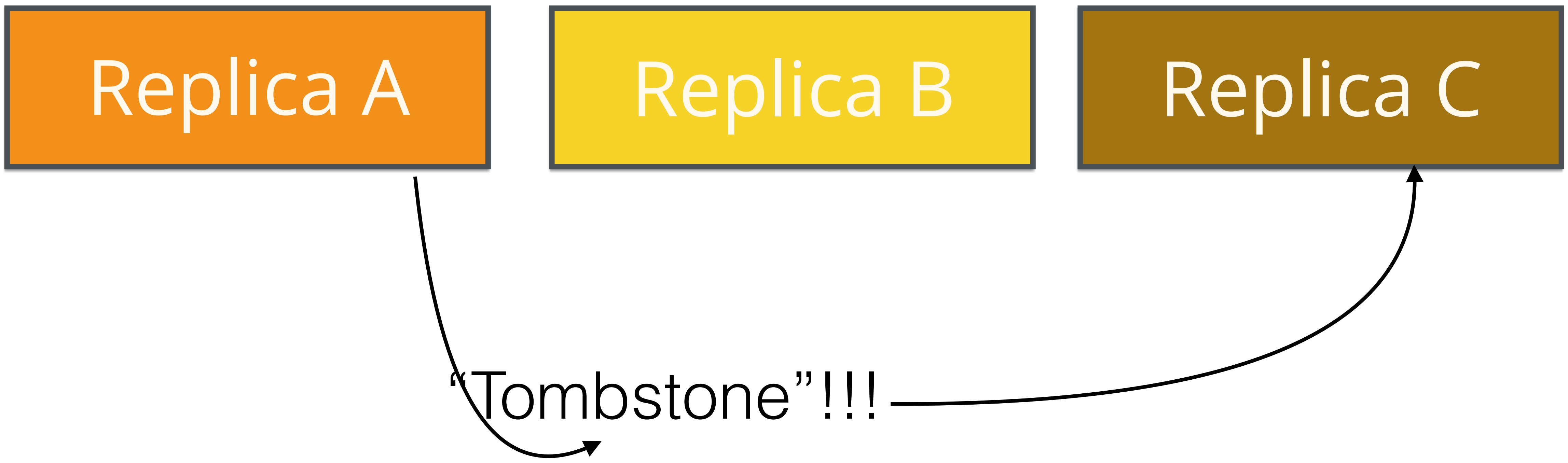
Replica A
Replica B
Replica C
C'
DEL 'k' [{a, 4}, {b, 3}]
Client X



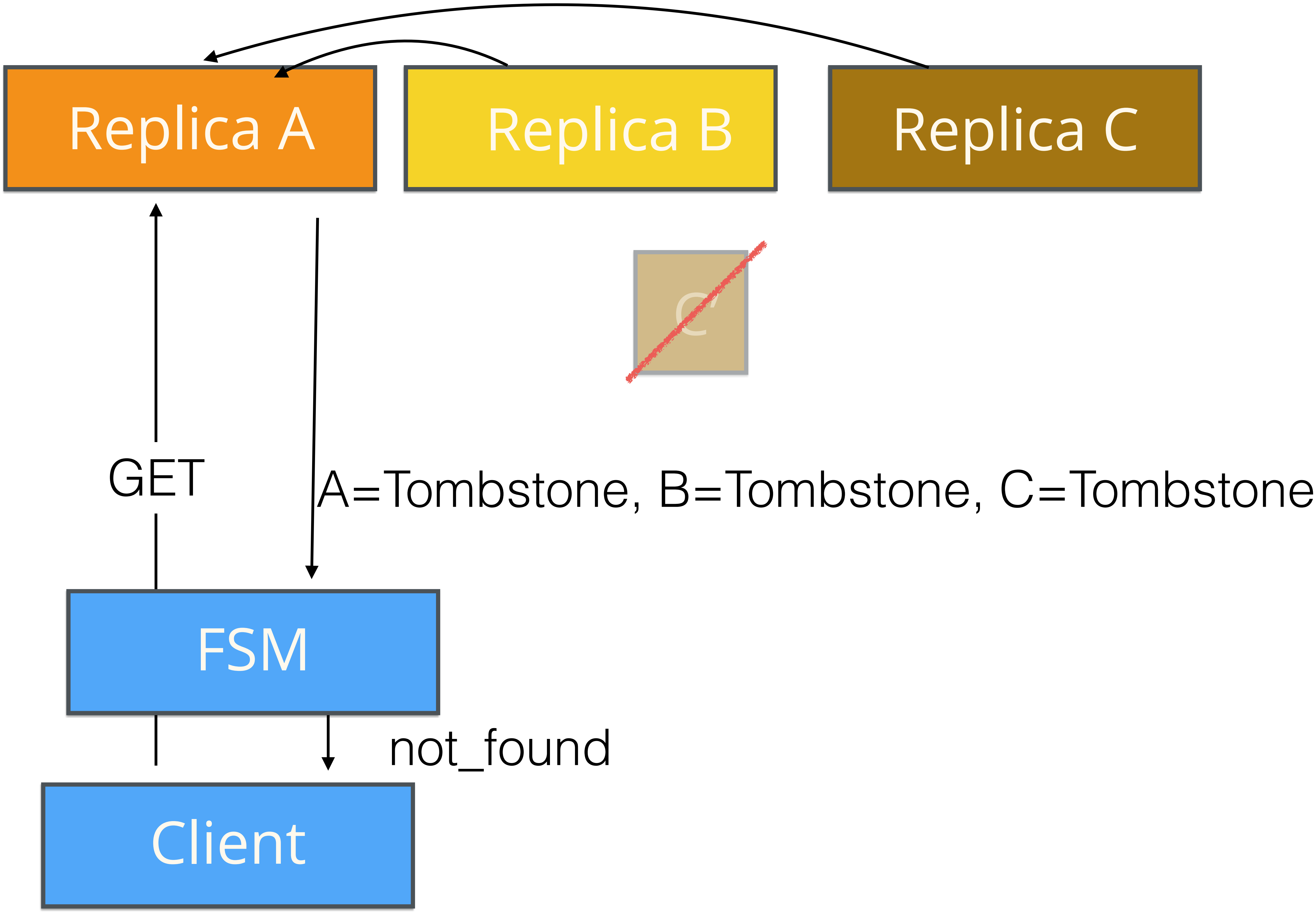
Replica A
Replica B
Replica C
C'
GET
Del FSM

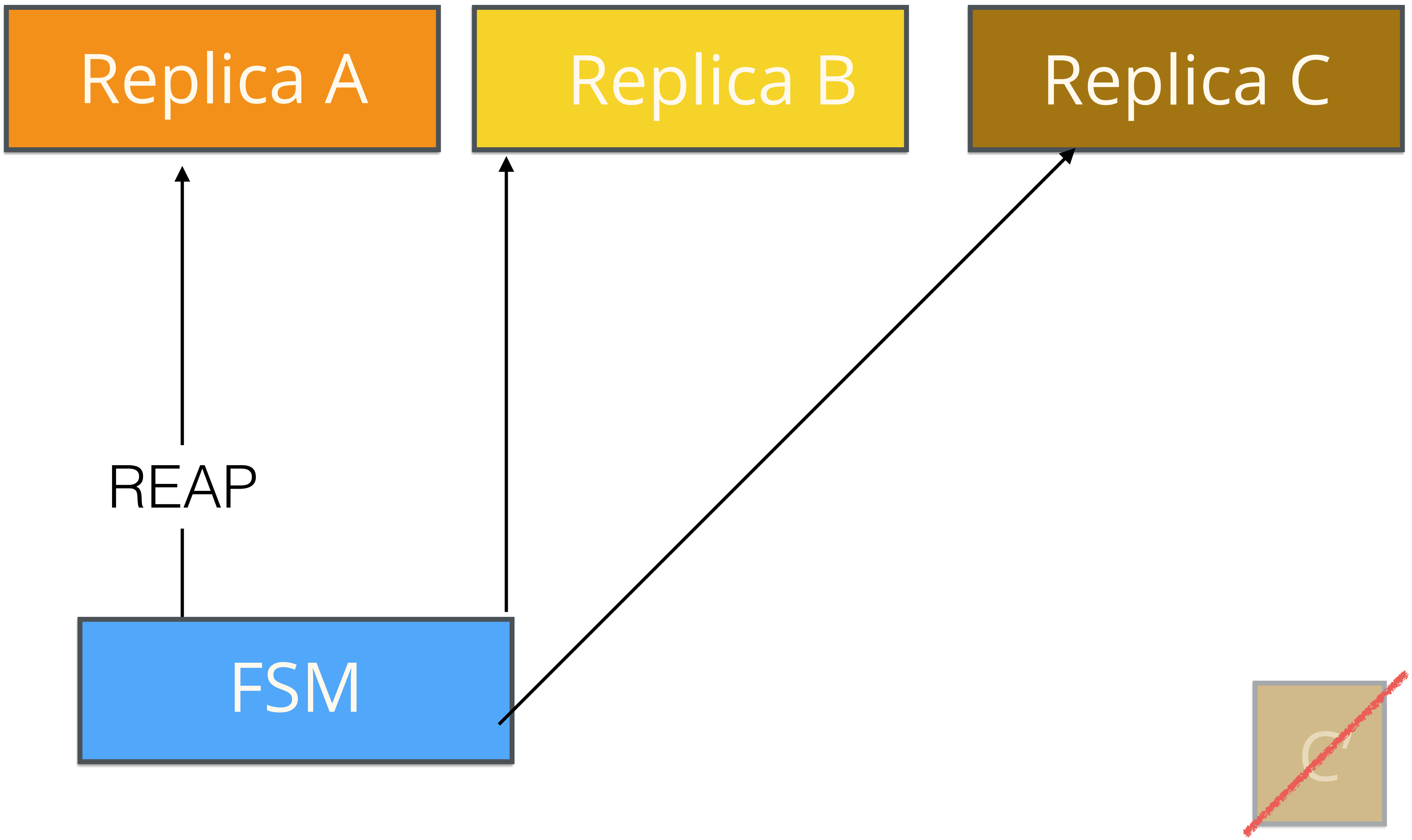


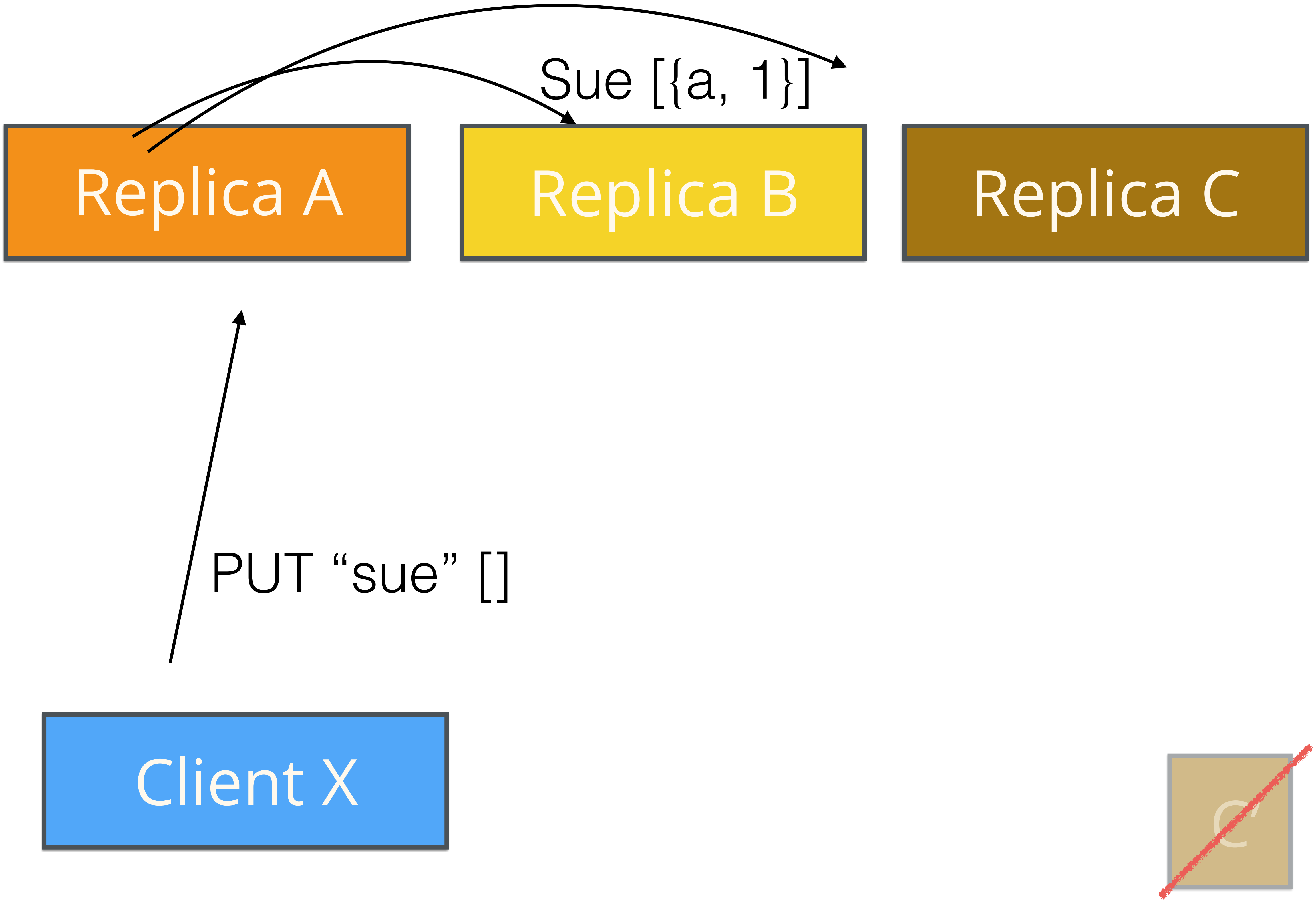
Replica A
Replica B
Replica C
C'
GET
A=Tombstone, B=Tombstone, C=not_found
Del FSM



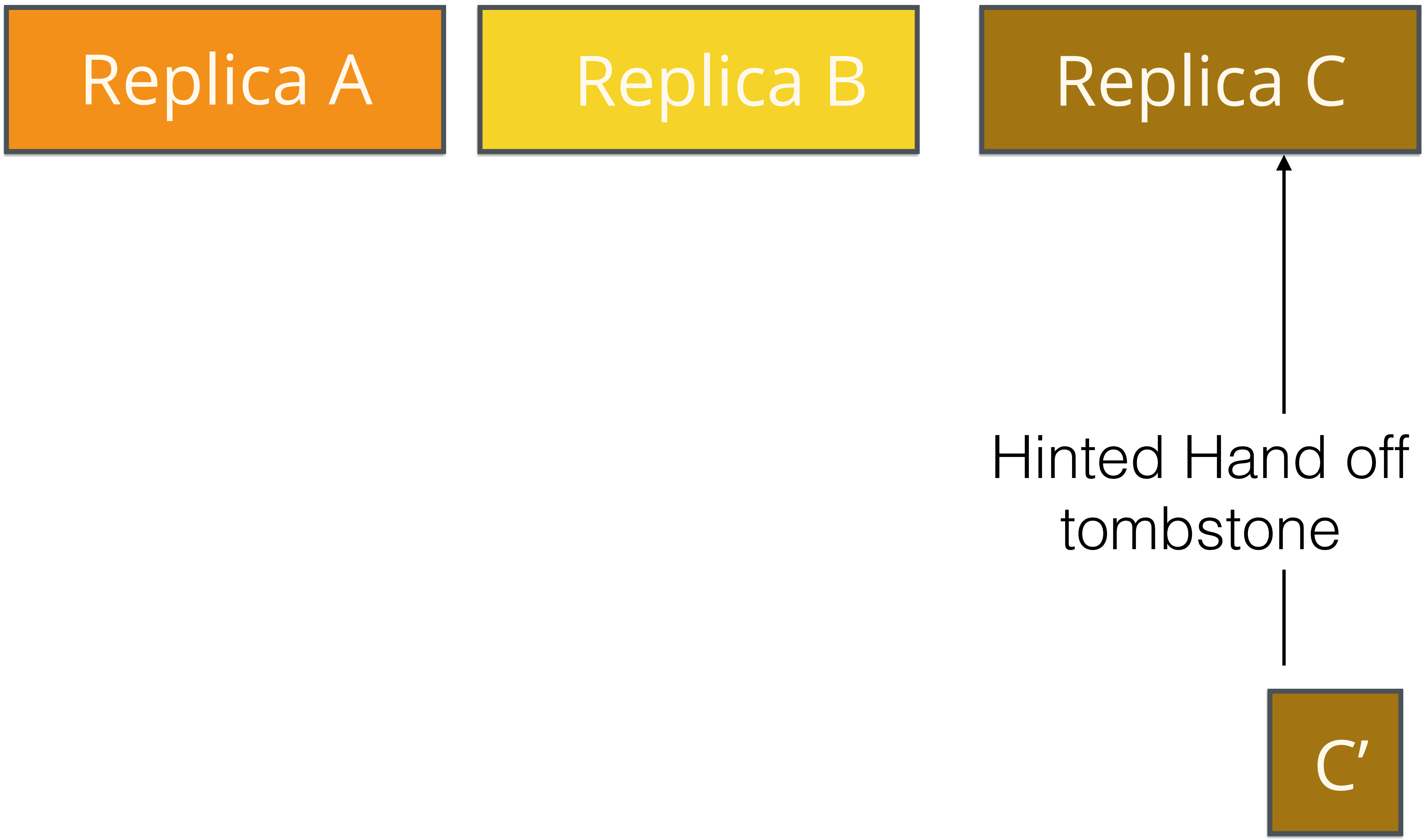
Read Repair

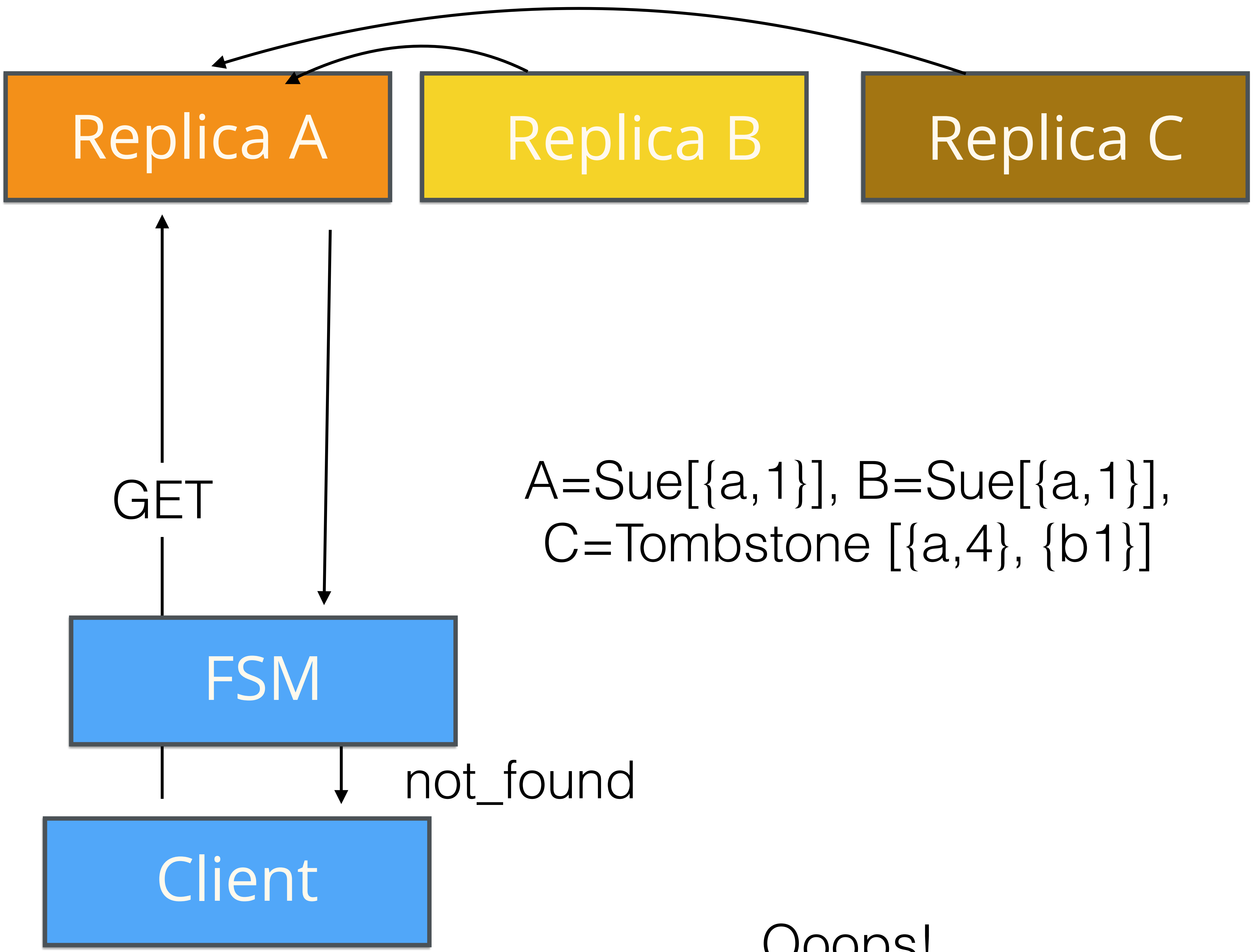

Replica A
Replica B
Replica C
C'
GET
A=Tombstone, B=Tombstone, C=Tombstone
FSM
not_found
Client


Replica A
Replica B
Replica C
REAP
FSM
C'



Sue [{a, 1}]
Replica A
Replica B
Replica C
PUT “sue” []
Client X
C’


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
- Read error

KV679
RYOW?

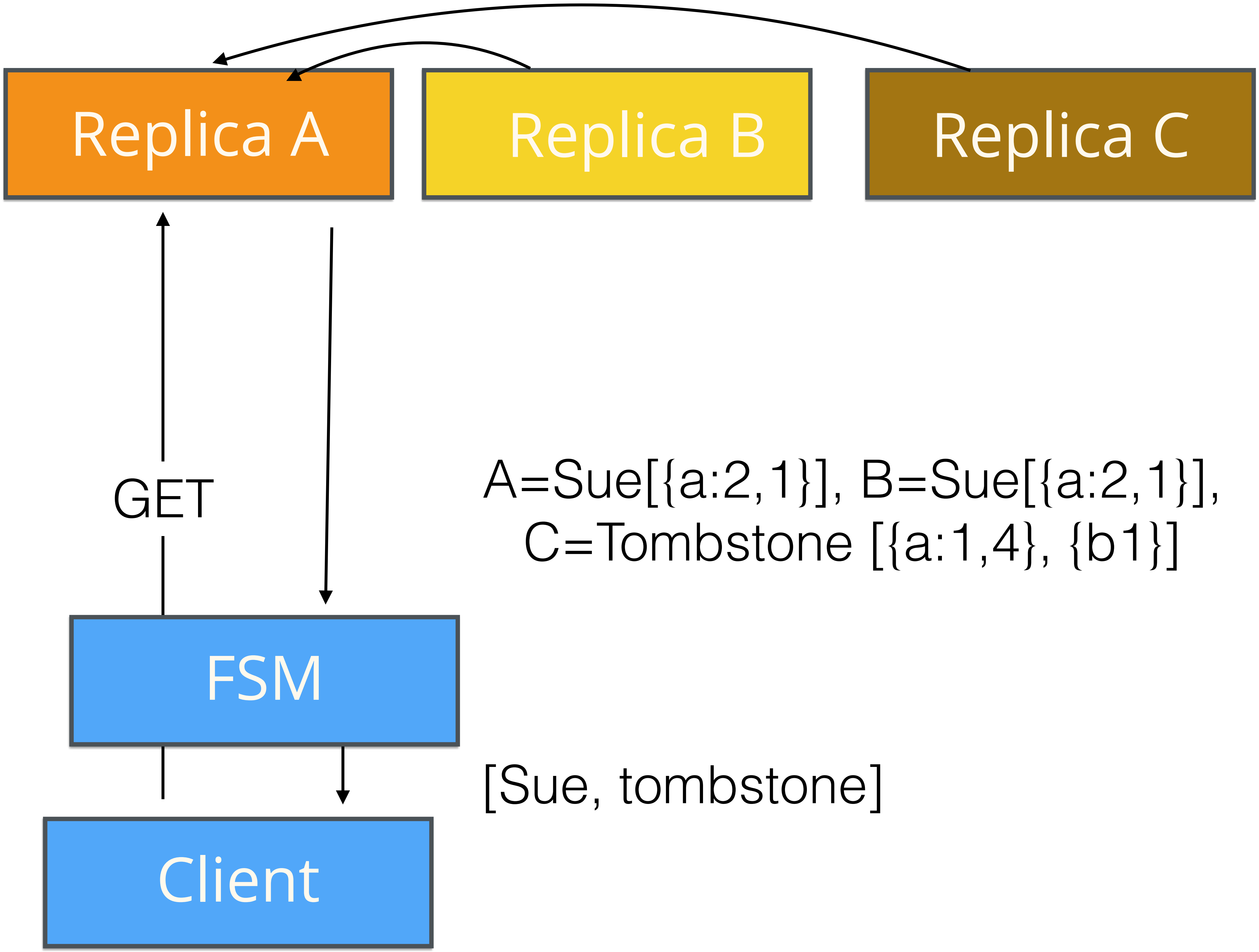
- Familiar
- History repeating

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

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!
- Version Vectors are HARD!
- Mind the Gap!