Distributed Systems

Spring Semester 2020

Lecture 25: Dynamo

John Liagouris
liagos@bu.edu

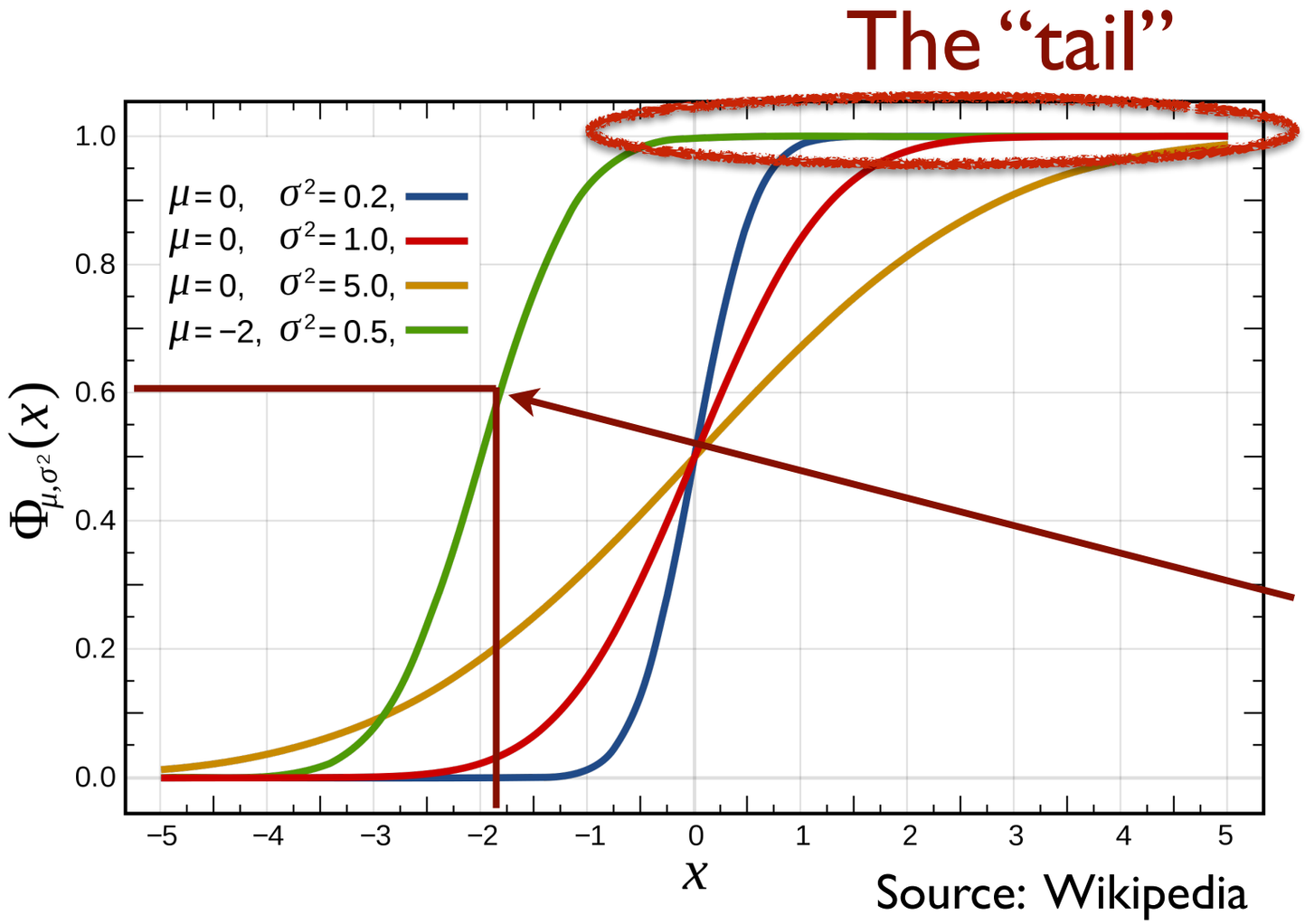
Why this paper

- Google, Yahoo, Facebook … Amazon ;-) Another real system
- Optimistic concurrency control — BAYOU but in a DB — write to any replica!
- Cool mixture of ideas and techniques
- Inspired FaceBook's Cassandra now Apache Cassandra
- Strives hard for no single point of failure — High Availability — can always write to it — No Quorum needed

Techniques used in Dynamo

- Consistent hashing — also in Parameter Server
- Vector clocks — also in TreadMarks, Bayou, and PS
- Quorum-based distributed consensus — weaker than Raft and Zookeeper
- Eventual Consistency — also in Bayou, PNUTS, and TAO
- OCC with conflict resolution — also in Bayou
- Gossip-based replica synchronization — also in Bayou
- Single-hop request routing — wait for DHTs next week

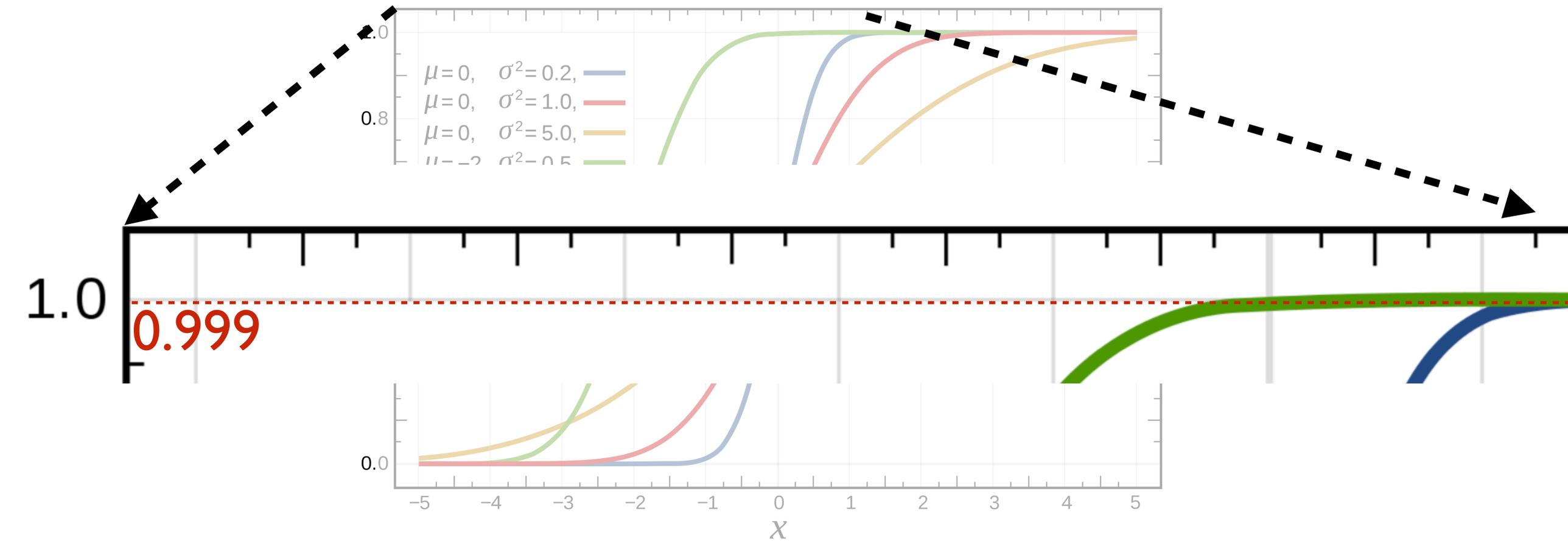
99.9th percentile



- The tail at scale: https://research.google/pubs/pub40801/

99.9th percentile

"Available for all clients"

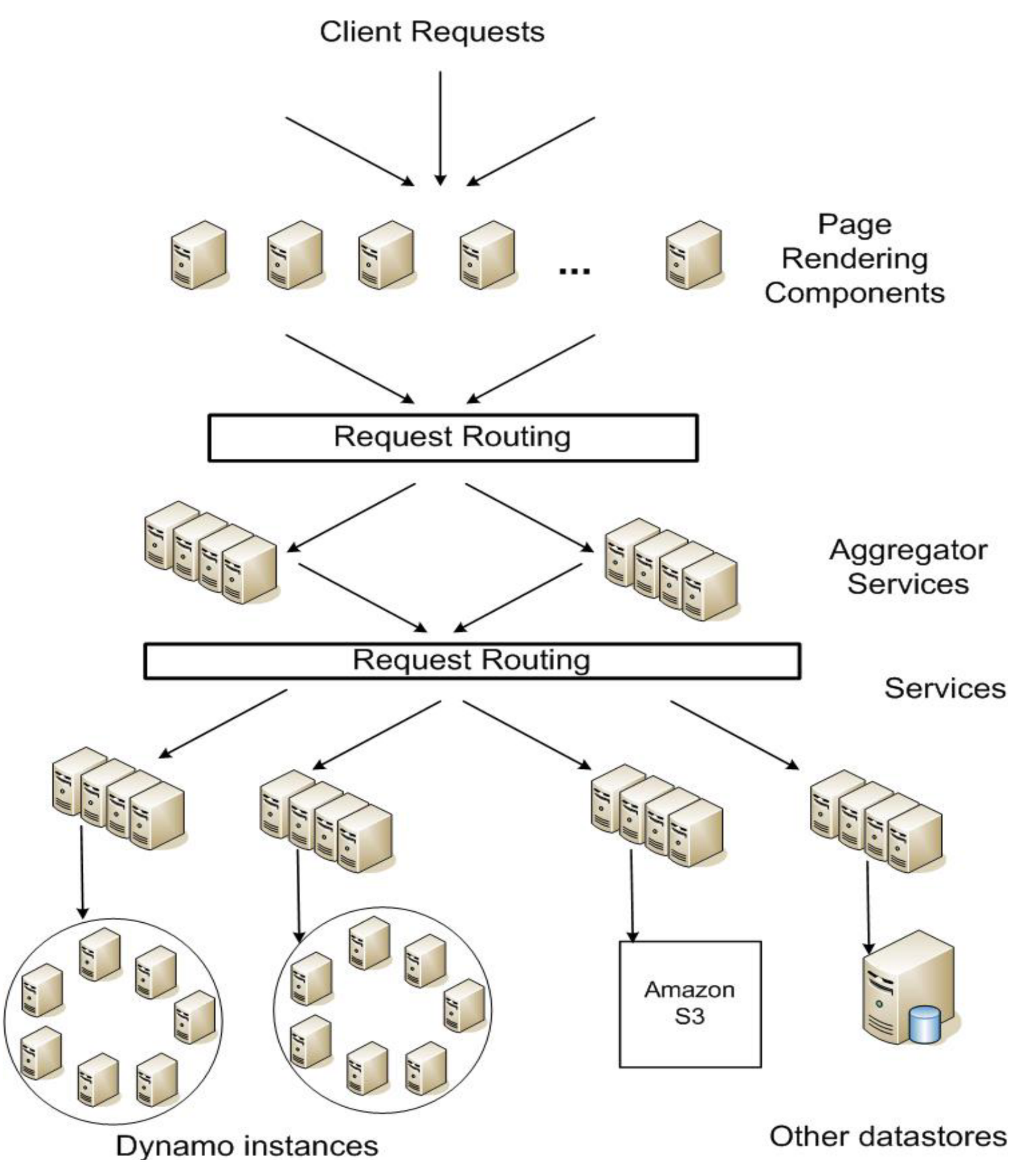


- The tail at scale: https://research.google/pubs/pub40801/

The Amazon Context

- Culture — Customer Satisfaction and Reliability is Money
- Apps developers are experts who push hard to meet Amazon's goals
  - Apps must work in the face of failures at all levels
- System must be able to be incrementally scaled
- Dynamo an always available storage service that is ALWAYS available

The Big Picture



Each item replicated at a few random nodes, by key hash

Why replicas at just a few sites? Why not replica at every site?

How it works : Interface

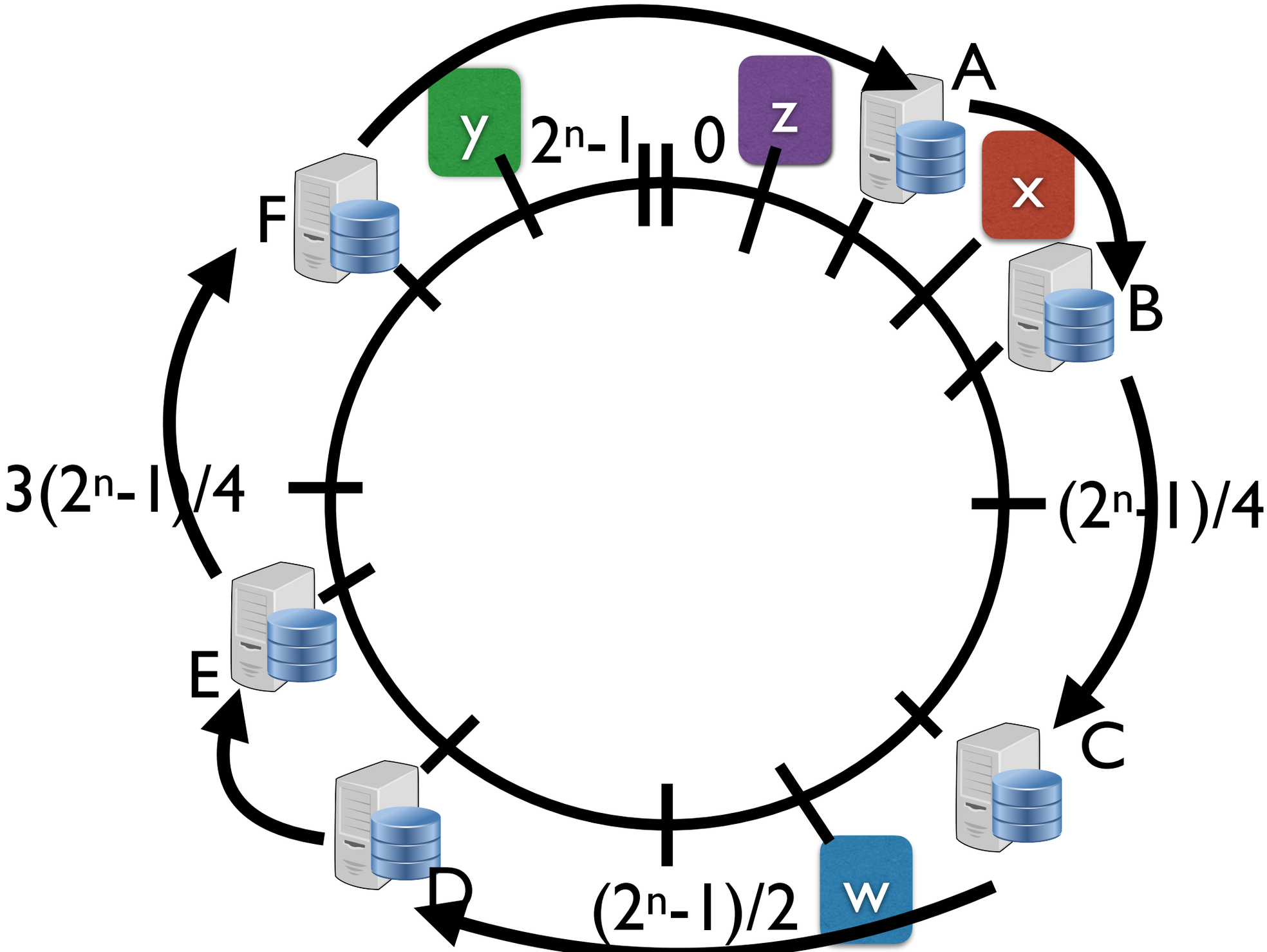
object,context = get(k)

put(k,object,context)

MD5 Hash of k used to generate **128 bit identifier**

- primary key data store: k to blob
- get locates object's replicas and returns one or more versions if there are conflicts — context encodes things like vector clock of replicas
- put locates where replicas should go and gets them to disk

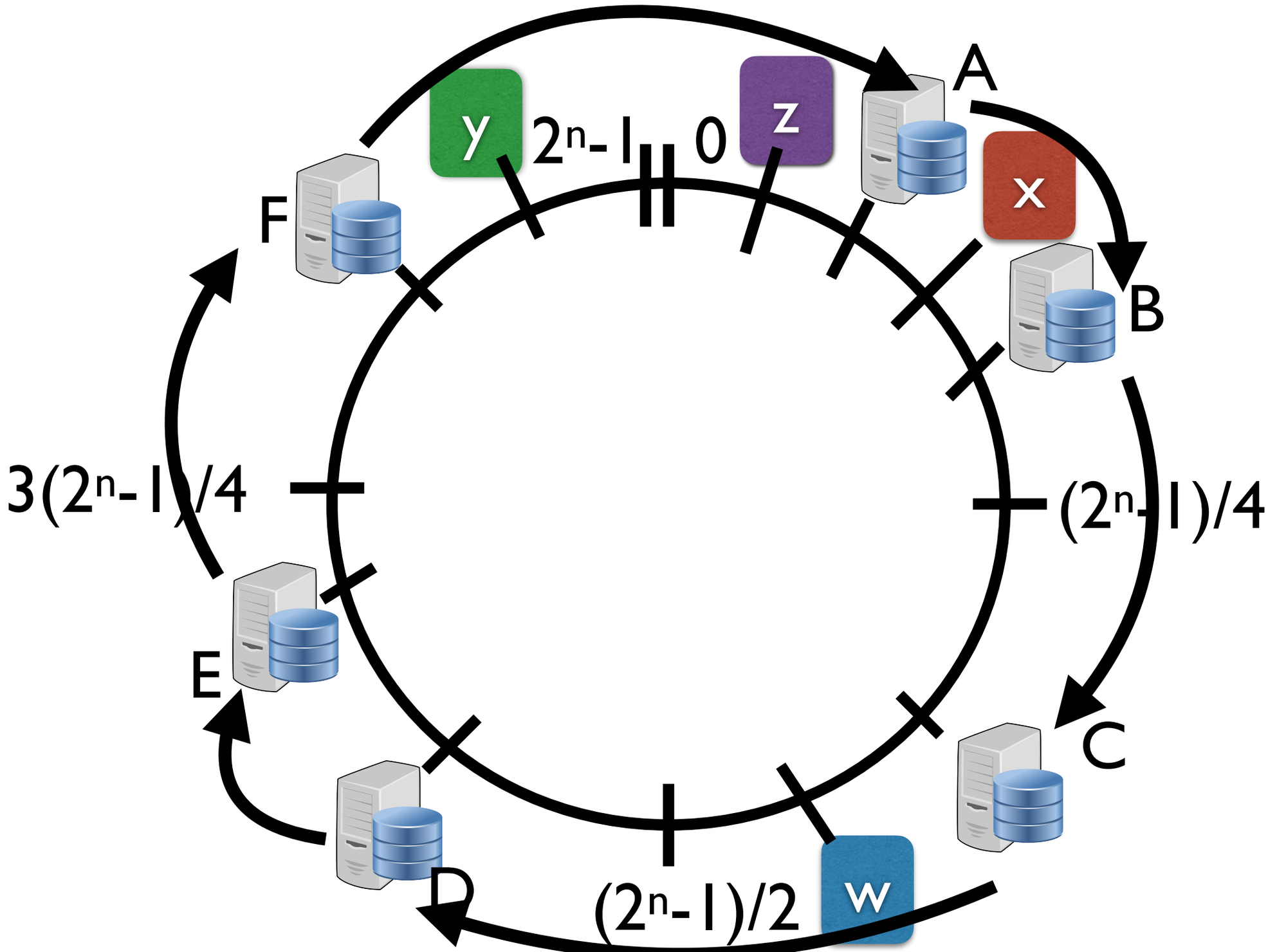
Big idea is that nodes can be inserted and removed by only talking to neighbors to exchange data rest of ring is untouched!



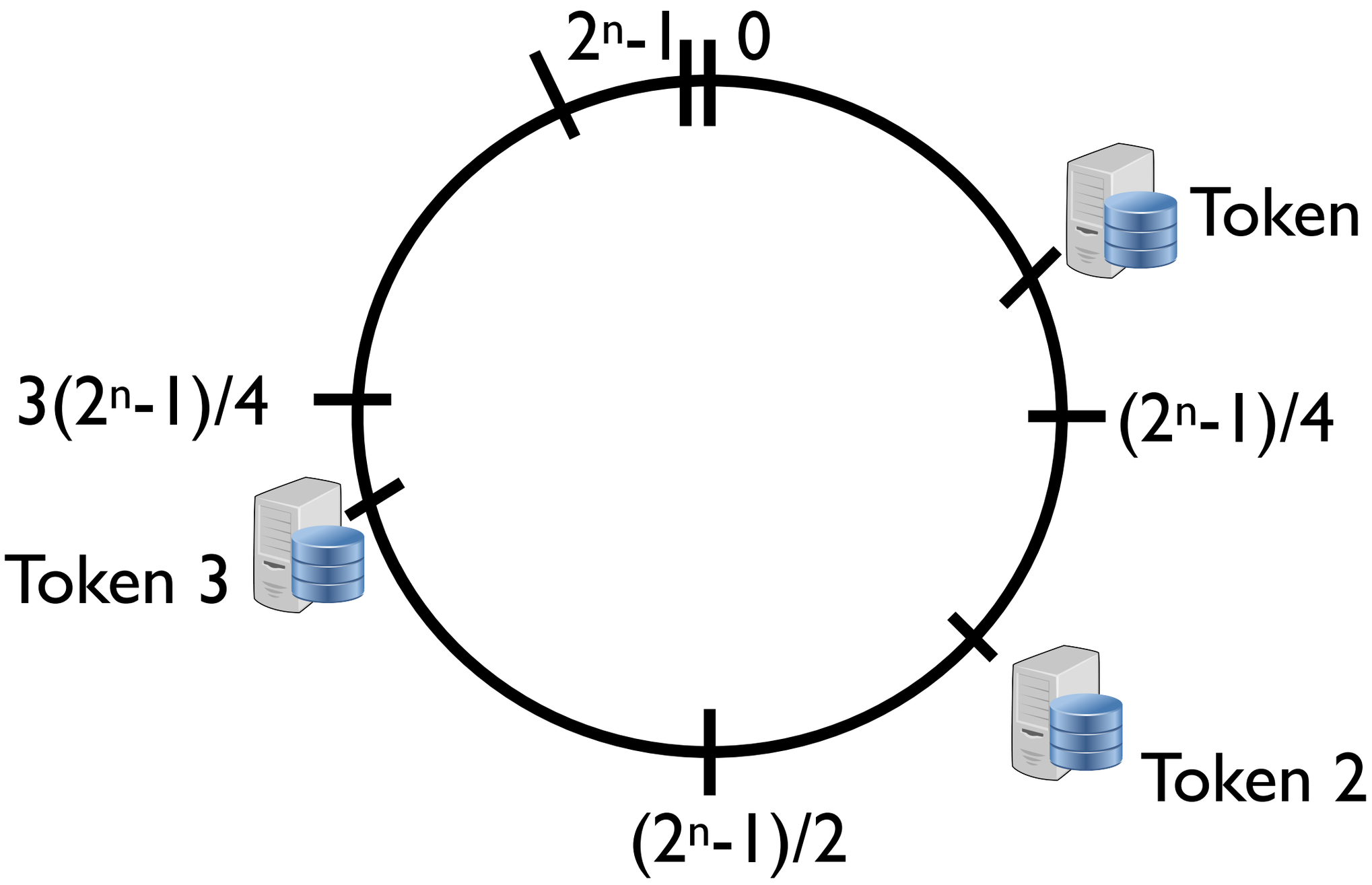
Why consistent hashing?

- Naturally decentralised
- Load balancing — well, not always (cf. Section 6.2)
- Dynamic addition/removal of nodes

Dynamo does a bunch of optimizations — eg. virtual nodes, replication, single hop routing

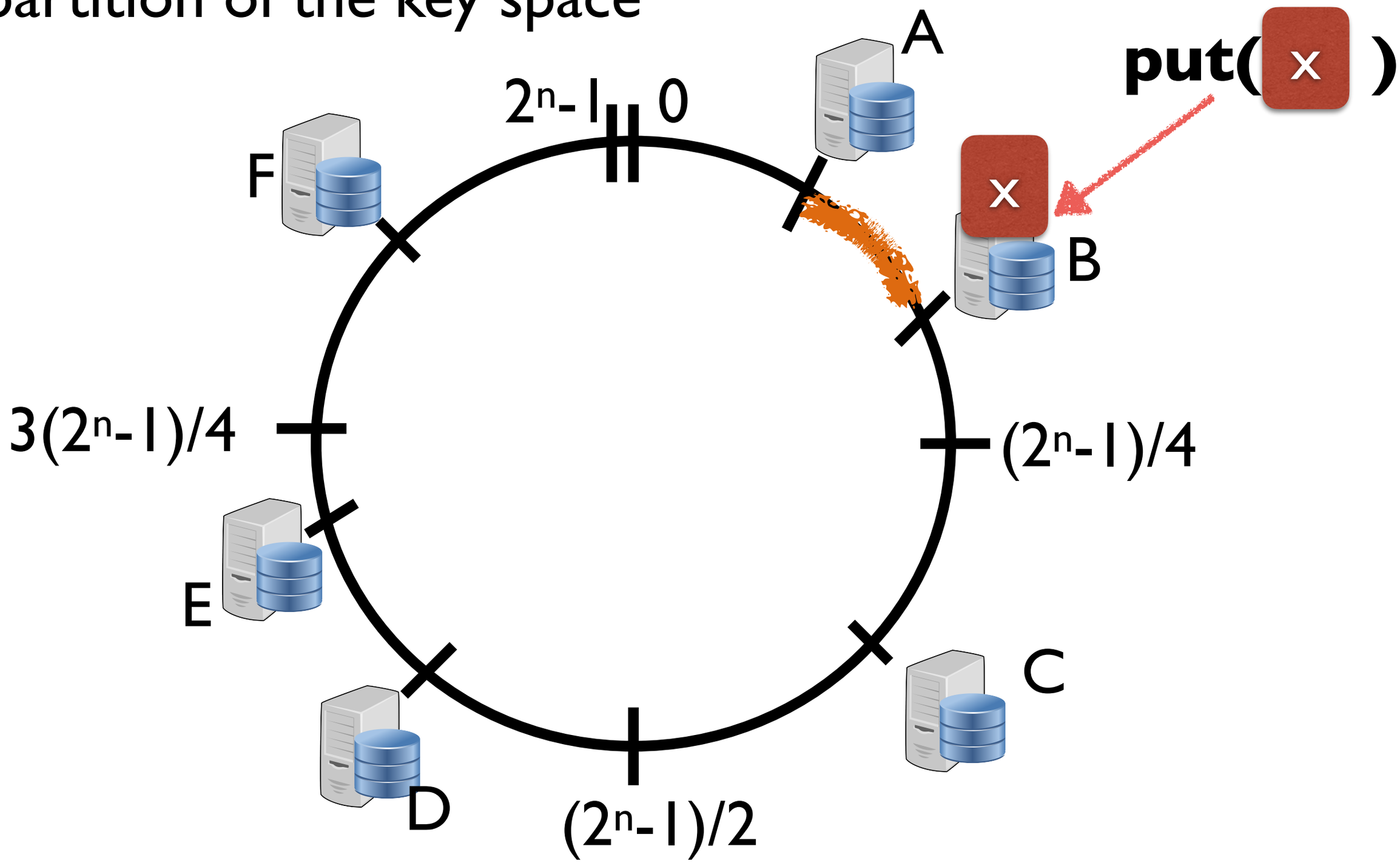


When adding a physical node (HostID) to the ring, the node is assigned many positions (virtual nodes) or “tokens”



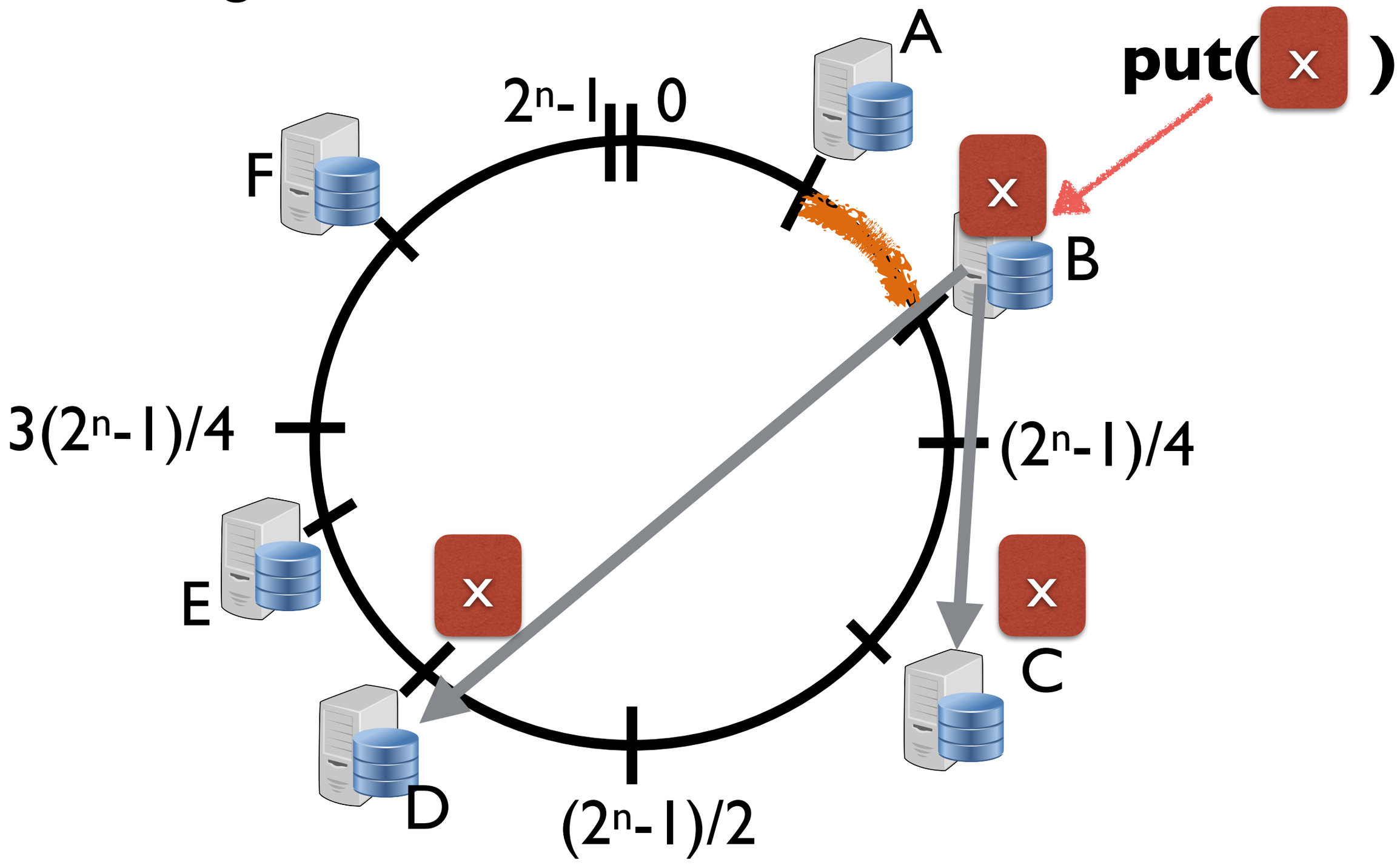
Replication

lets assume x is in B's partition of the key space

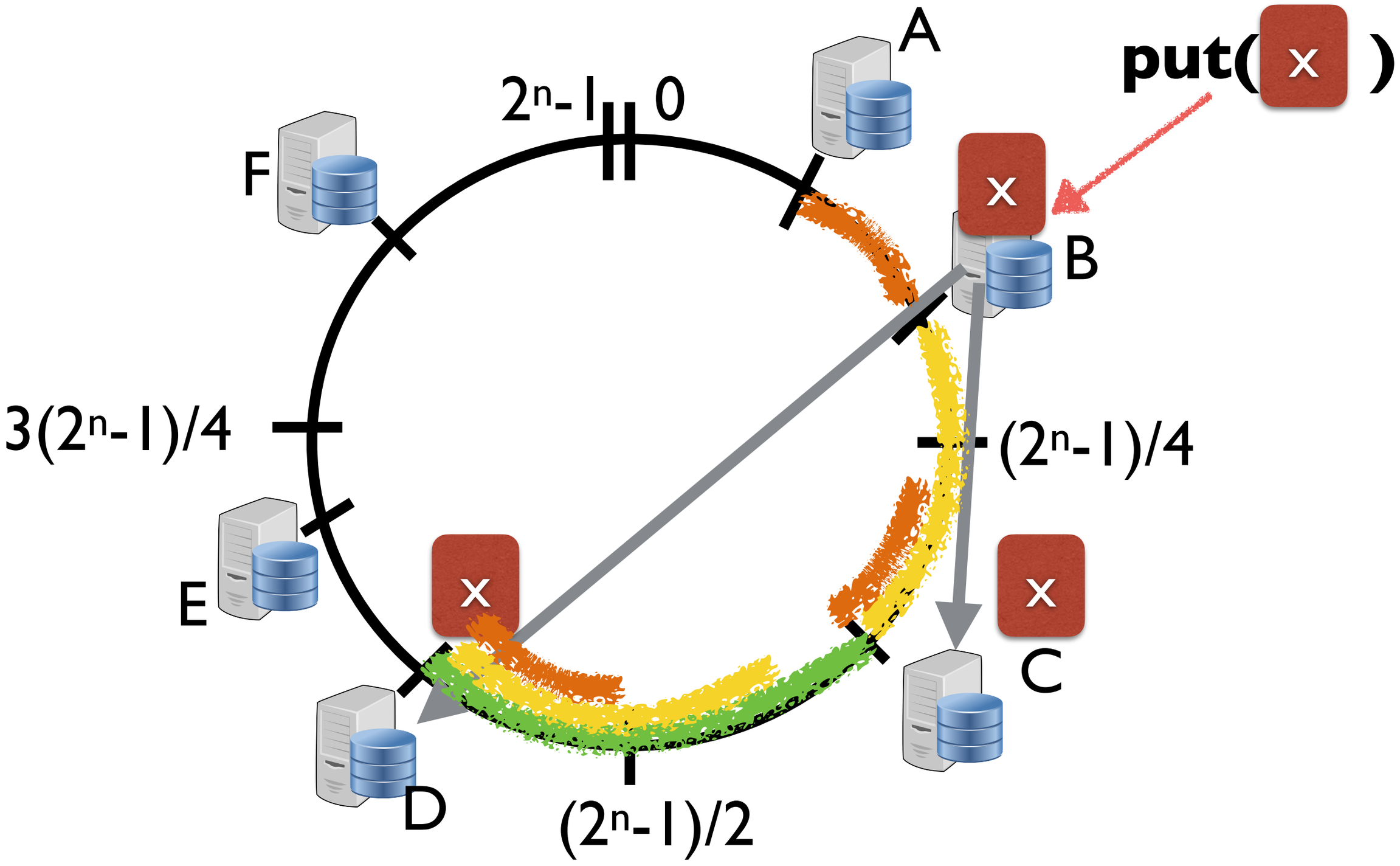


Replication

Assuming N=3



Thus a node is responsible for the region between its id and its N^{th} predecessor



What's the point of this?

. "To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured *"per-instance" "*

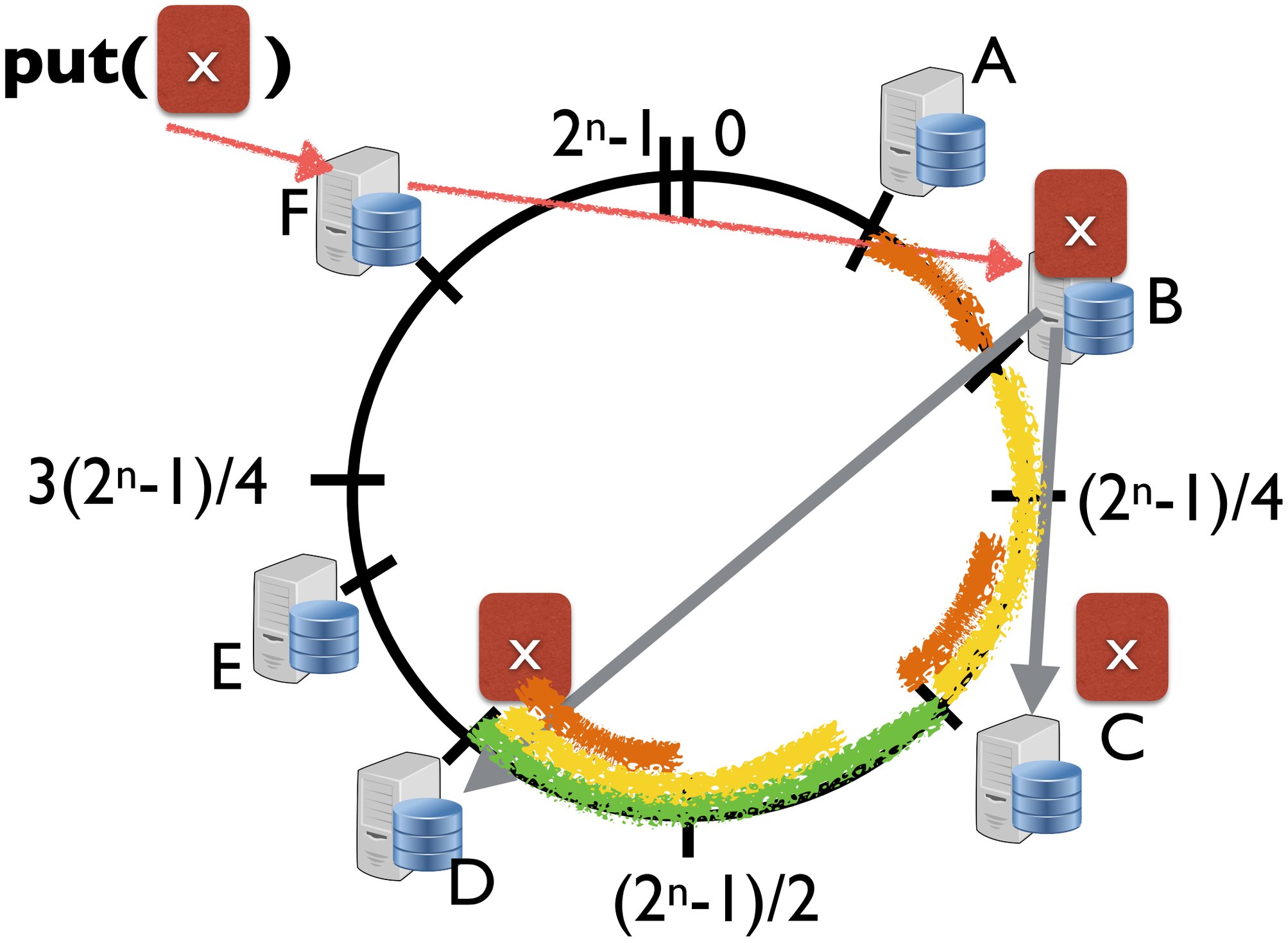
But isn't the writes to disk? Why do we need the replication?

Failures

- Temporary:
  - Keep going around the ring **until you find some who is alive**!
    - ALWAYS WRITABLE — different from PNUTS and TAO
- Permanent :
  - Create a new replica and add it to the ring

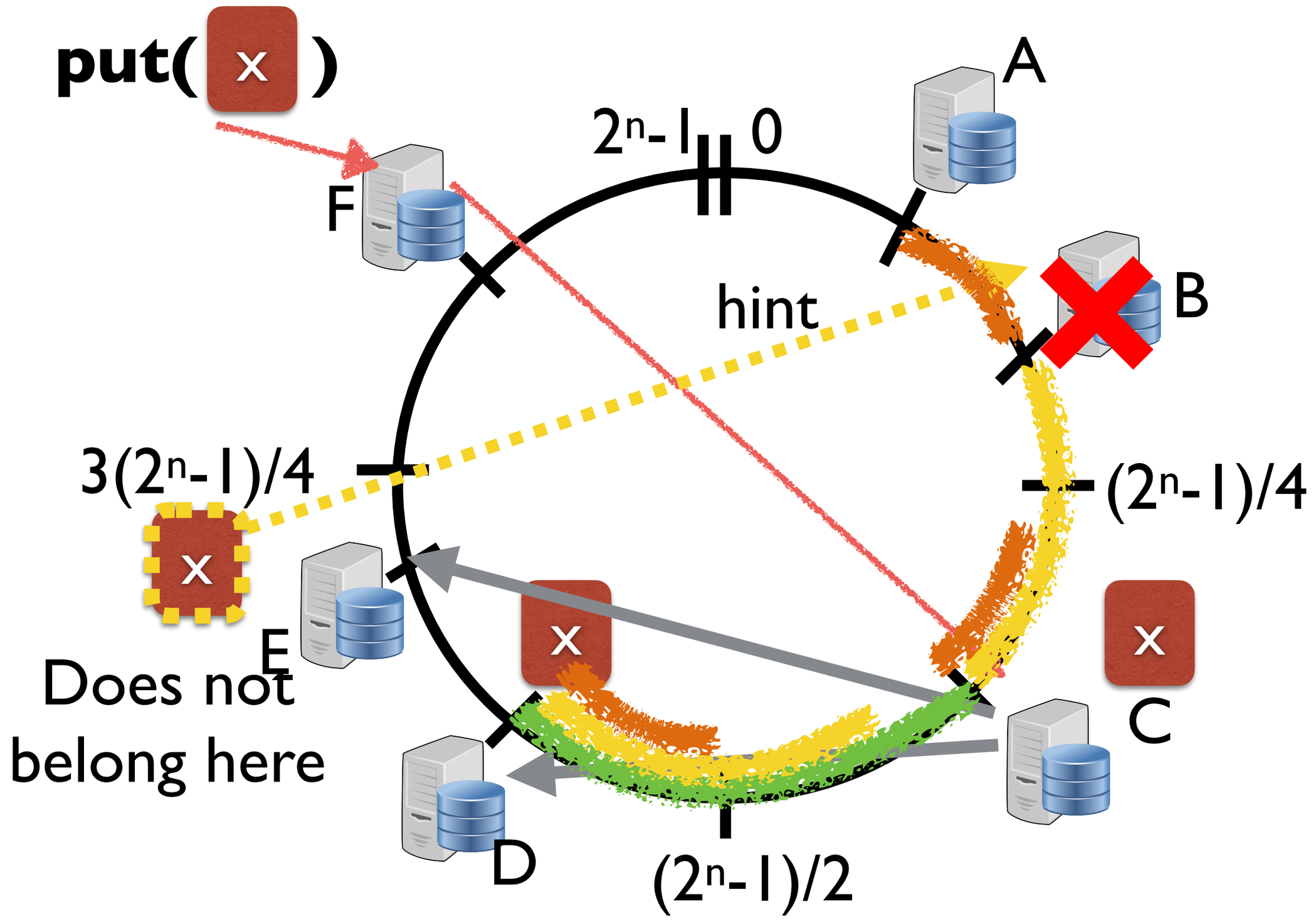
FAILURES

ASSUME W=N=3



FAILURES

ASSUME W=N=3



Consequences of ALWAYS writeable

- Consequence 1:
  - No Master! Look Mom "No Hands"
  - Idea: "Sloppy Quorum"
- Consequence 2:
  - Always writeable + Failures = conflicting versions can arise
  - Idea: "Eventual Consistency" techniques

Sloppy Quorum

- Try to get consistency benefits of single master if no failures
  - But allows progress even if coordinator fails, which PNUTS does not
- When no failures, send reads/writes through single node, the coordinator
  - Causes reads to see writes in the usual case
- But don't insist! allow reads/writes to any replica if failures

N, R, W & QUORUM

“This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that R + W > N yields a ***quorum-like*** system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency. “

What does this mean?

N,R,W

- N : Durability — survive lose of nodes, disks, data centers
- If we don't hear back from R/W nodes then the operations is failed — loss of availability
  - highest availability when W=1 for writes
- Lower values of R/W — increase risk of inconsistencies (return back to client before replicas are written)
  - reduce Durability as persists only at < N locations

Quorum **Like**

- Interesting behavior quorum like when : R + W > N
  - never wait for all N
  - but R and W will overlap
  - cuts tail off delay distribution and tolerates some failures

Sloppy Quorum

- N is first N *reachable* nodes in preference list
  - each node pings successors to keep rough estimate of up/down
  - "sloppy" quorum, since nodes may disagree on reachable
- Sloppy quorum means R/W overlap ***not guaranteed***

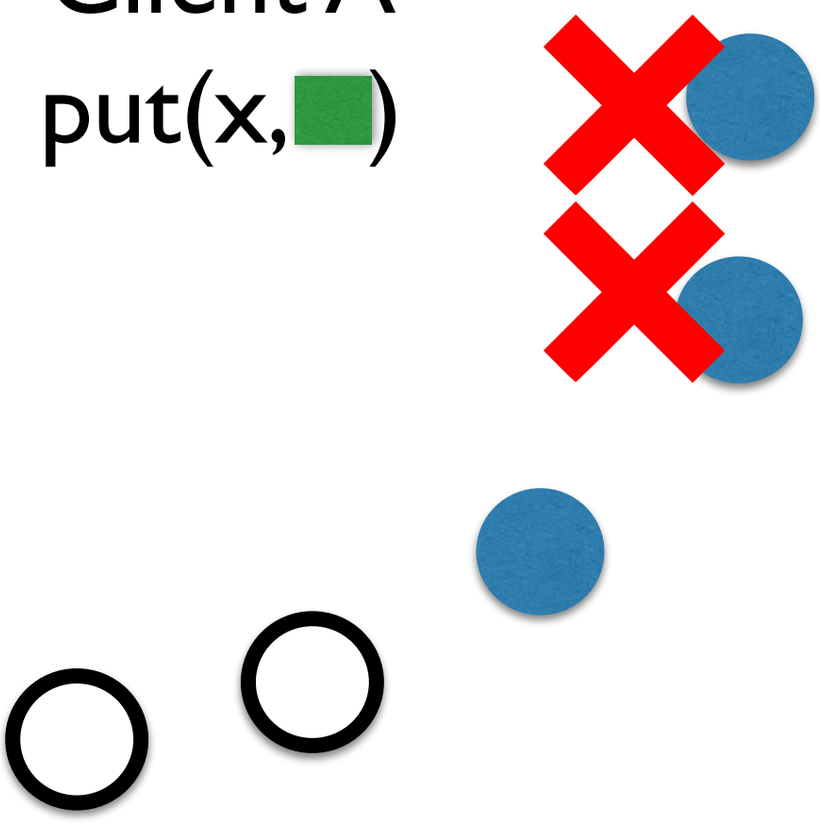
Sloppy Quorum

- Want gets to read most recent writes
- R+W>N works because R and W will overlap BUT Remember we still send it to N nodes we just don't necessarily wait for all N We wait for R on read and W on write

N=3,W=2,R=2

Client A

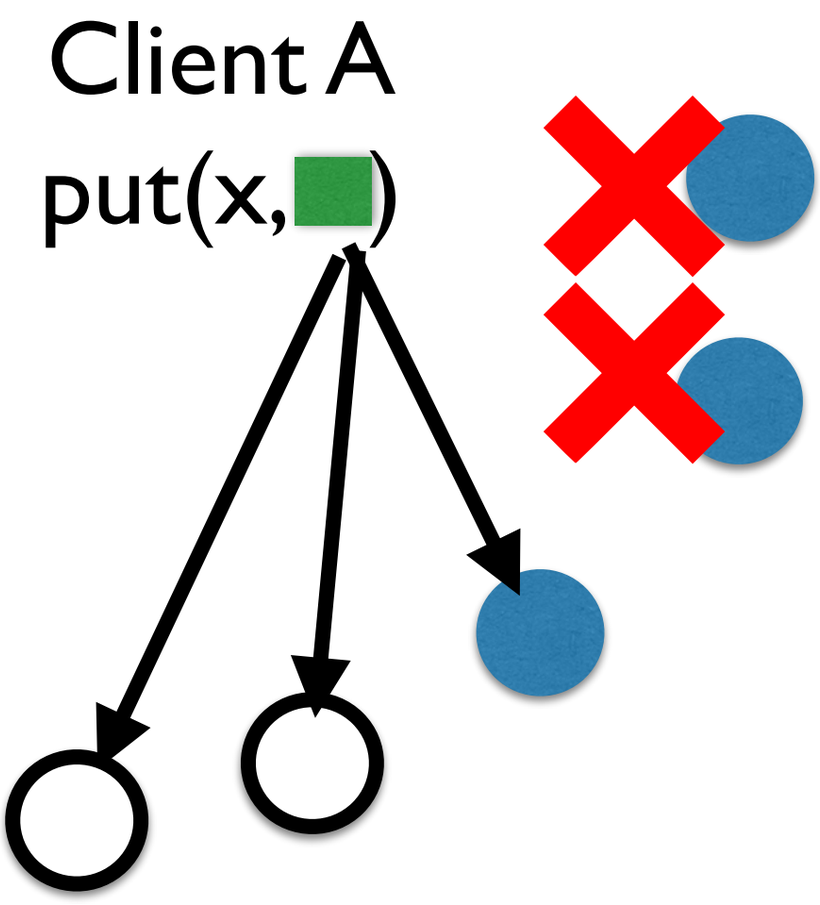
put(x,)



Sloppy Quorum

- Want gets to read most recent writes
- R+W>N works because R and W will overlap BUT Remember we still send it to N nodes we just don't necessarily wait for all N We wait for R on read and W on write

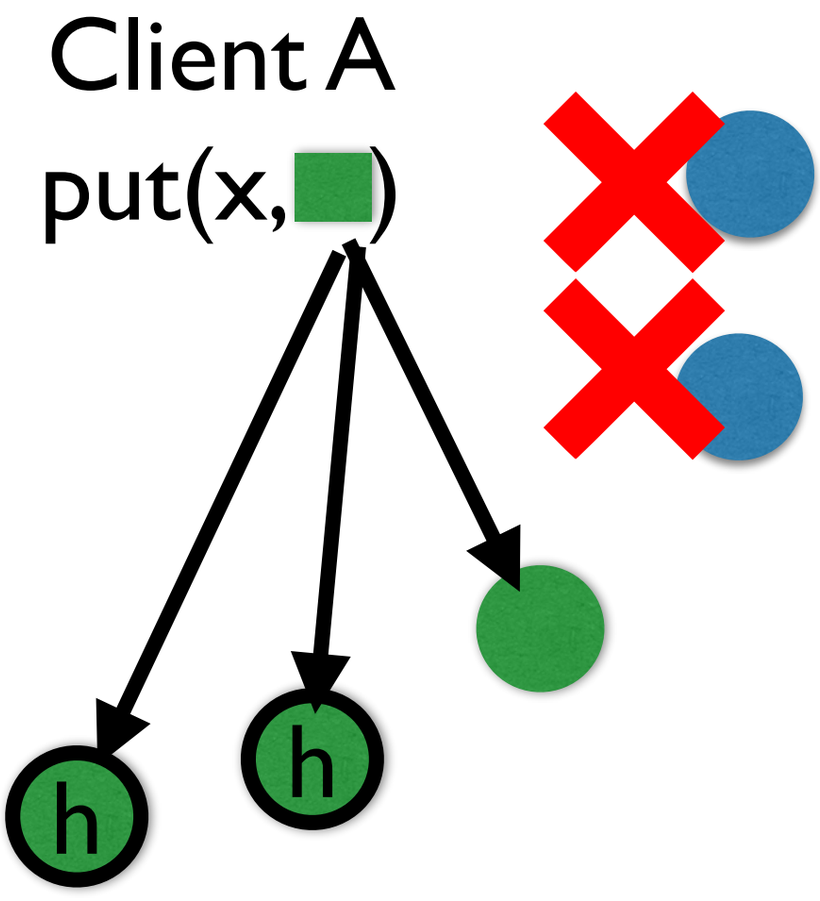
N=3, W=2, R=2



Sloppy Quorum

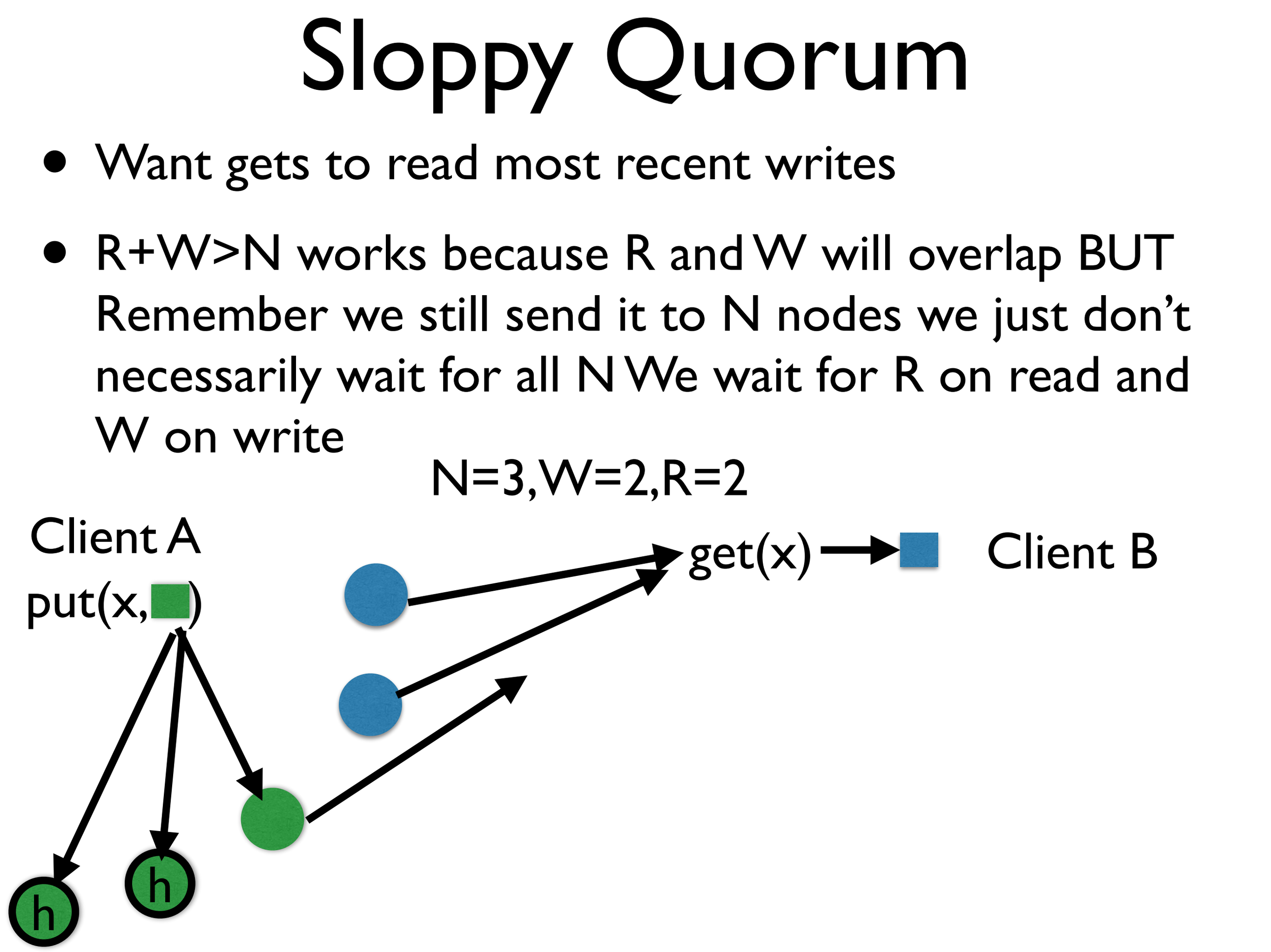
- Want gets to read most recent writes
- R+W>N works because R and W will overlap BUT Remember we still send it to N nodes we just don't necessarily wait for all N We wait for R on read and W on write

N=3, W=2, R=2



Sloppy Quorum

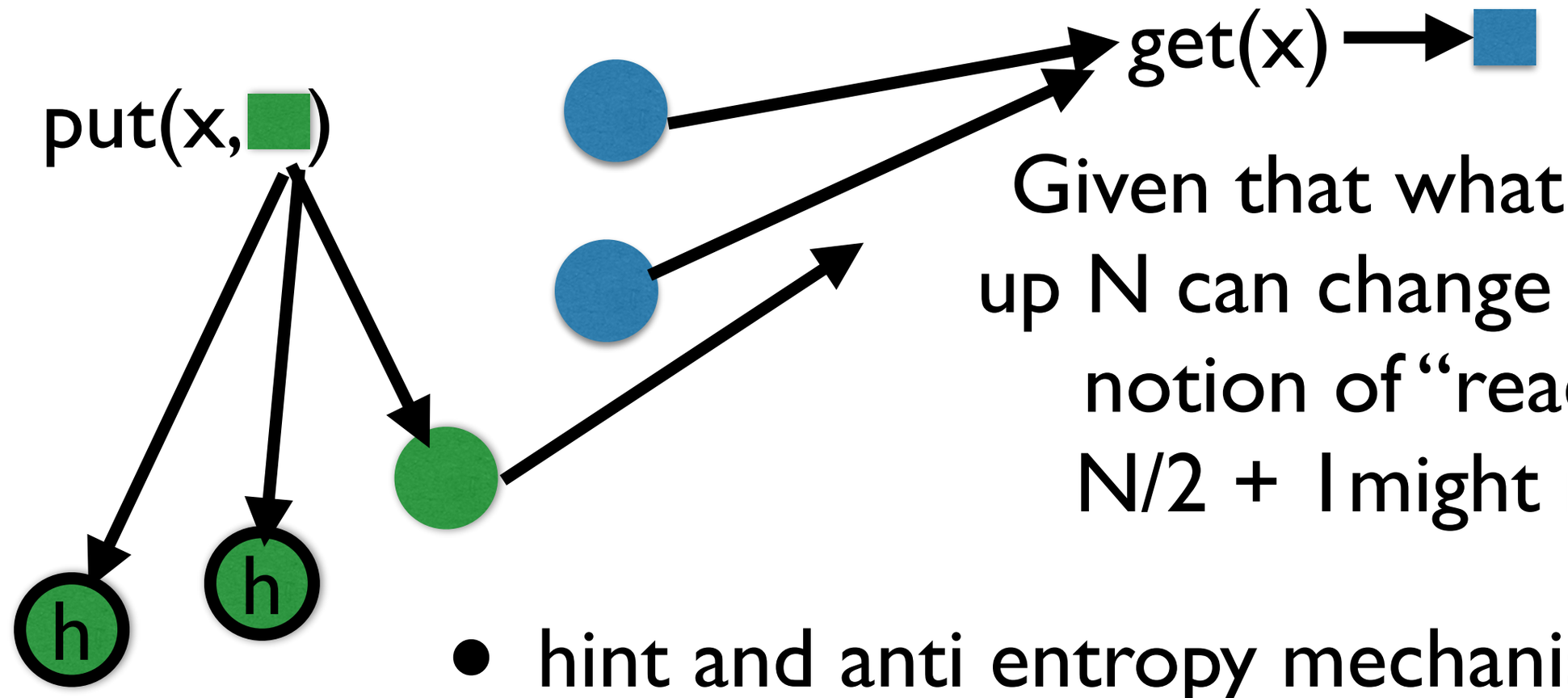
- Want gets to read most recent writes
- R+W>N works because R and W will overlap BUT Remember we still send it to N nodes we just don't necessarily wait for all N We wait for R on read and W on write



Sloppy Quorum

- Want gets to read most recent writes
- R+W>N works because R and W will overlap BUT Remember we still send it to N nodes we just don't necessarily wait for all N We wait for R on read and W on write

N=3,W=2,R=2



Given that what nodes make up N can change based on our notion of "reachable" — N/2 + 1might not overlap

- hint and anti entropy mechanisms try to fix

Eventual Consistency and Versioning

- On a put a coordinator does async writes to its replicas
- Put can return before hearing from all replicas (W<N)
  - Thus replica's can be stale — longer if failures
  - Can get divergent versions — gets and put to stale version can create forked version

Dynamo

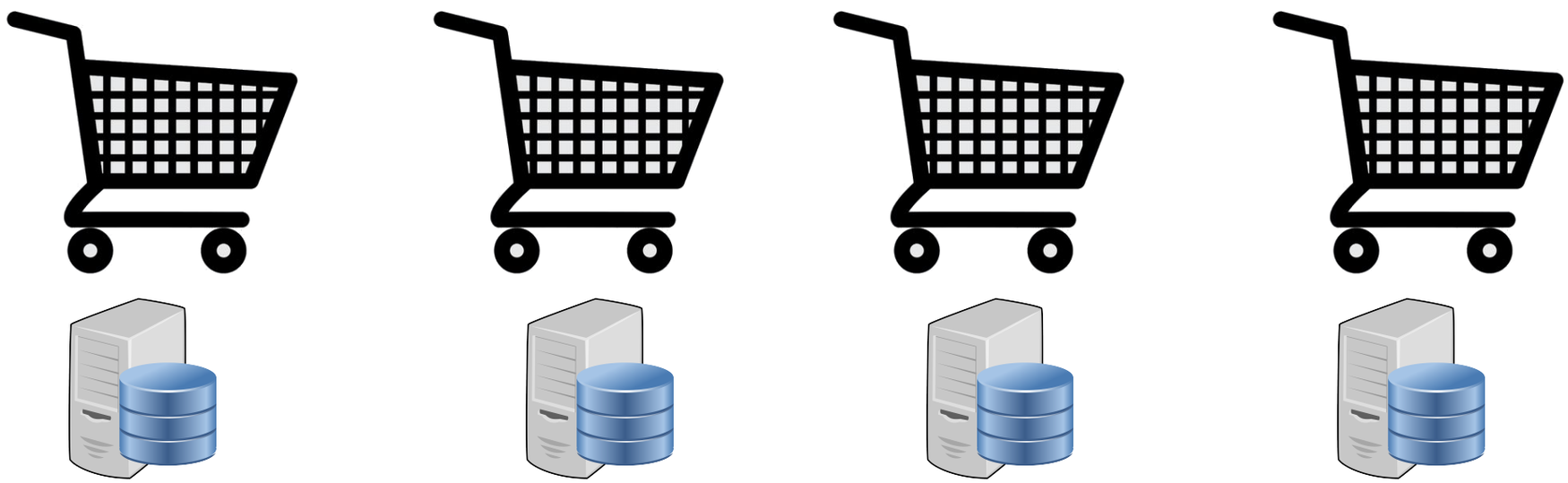
- Accept writes at any replica
- Allow divergent replicas
- Allow reads to see stale or conflicting data
- Resolve multiple versions when noticed
  - latest version if no conflicting updates
  - if conflicts, reader must merge and then write
  - like Bayou -- but in a DB

How can multiple versions arise?

- Maybe a node missed the latest write due to network problem
- So it has old data, should be superseded by newer puts
- get() consults R, will **likely** see newer version as well as old — assuming R+W>N

N=3 R=2 W=2

Carts start empty

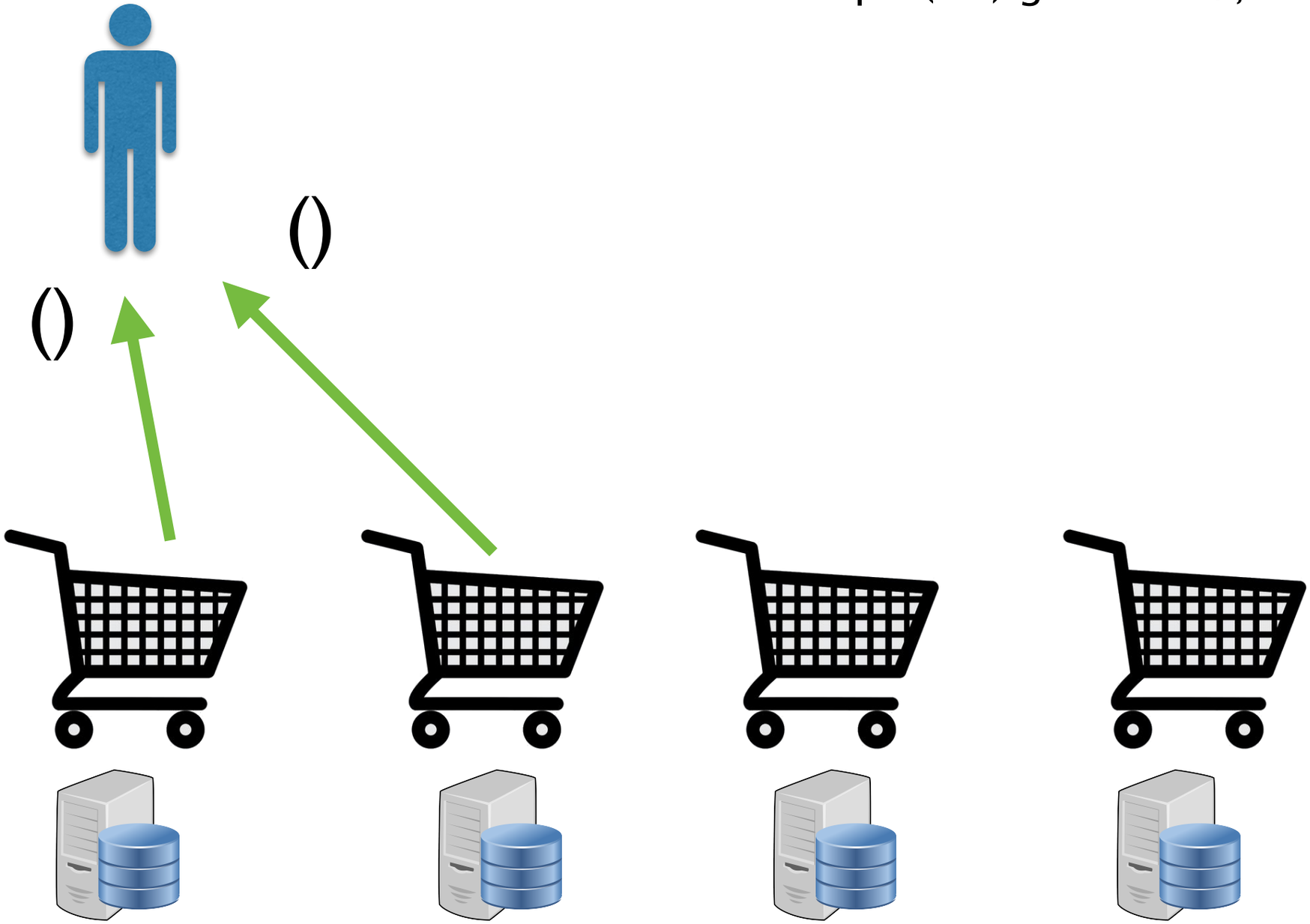


Preference list of nodes

N=3 R=2 W=2

Carts start empty

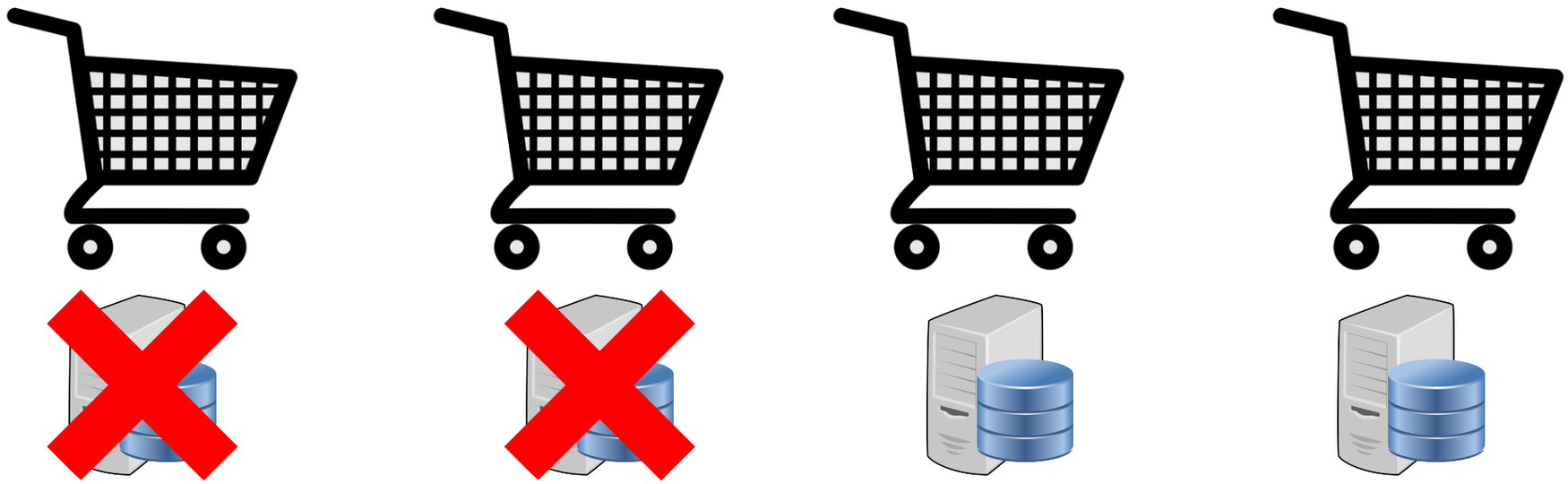
client 1 wants to add item X
get() from n1, n2, yields ""
n1 and n2 fail
put("X") goes to n3, n4



N=3 R=2 W=2

Carts start empty

client 1 wants to add item X
get() from n1, n2, yields ""
n1 and n2 fail
put("X") goes to n3, n4



Preference list of nodes

N=3 R=2 W=2

Carts start empty

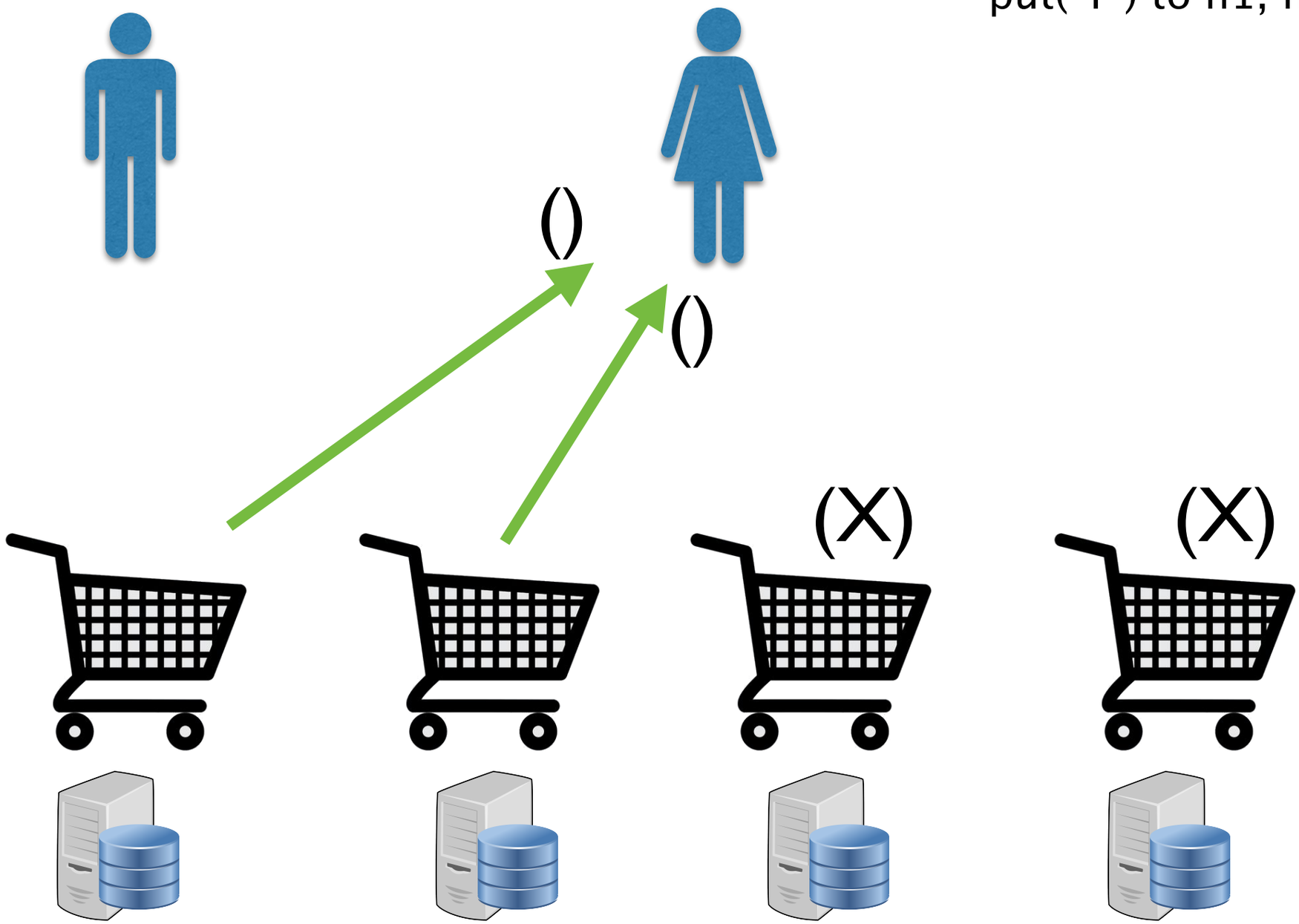
client 1 wants to add item X
get() from n1, n2, yields ""
n1 and n2 fail
put("X") goes to n3, n4

Preference list of nodes

N=3 R=2 W=2

Carts start empty

n1, n2 revive
client 2 wants to add Y
get() from n1, n2 yields ""
put("Y") to n1, n2

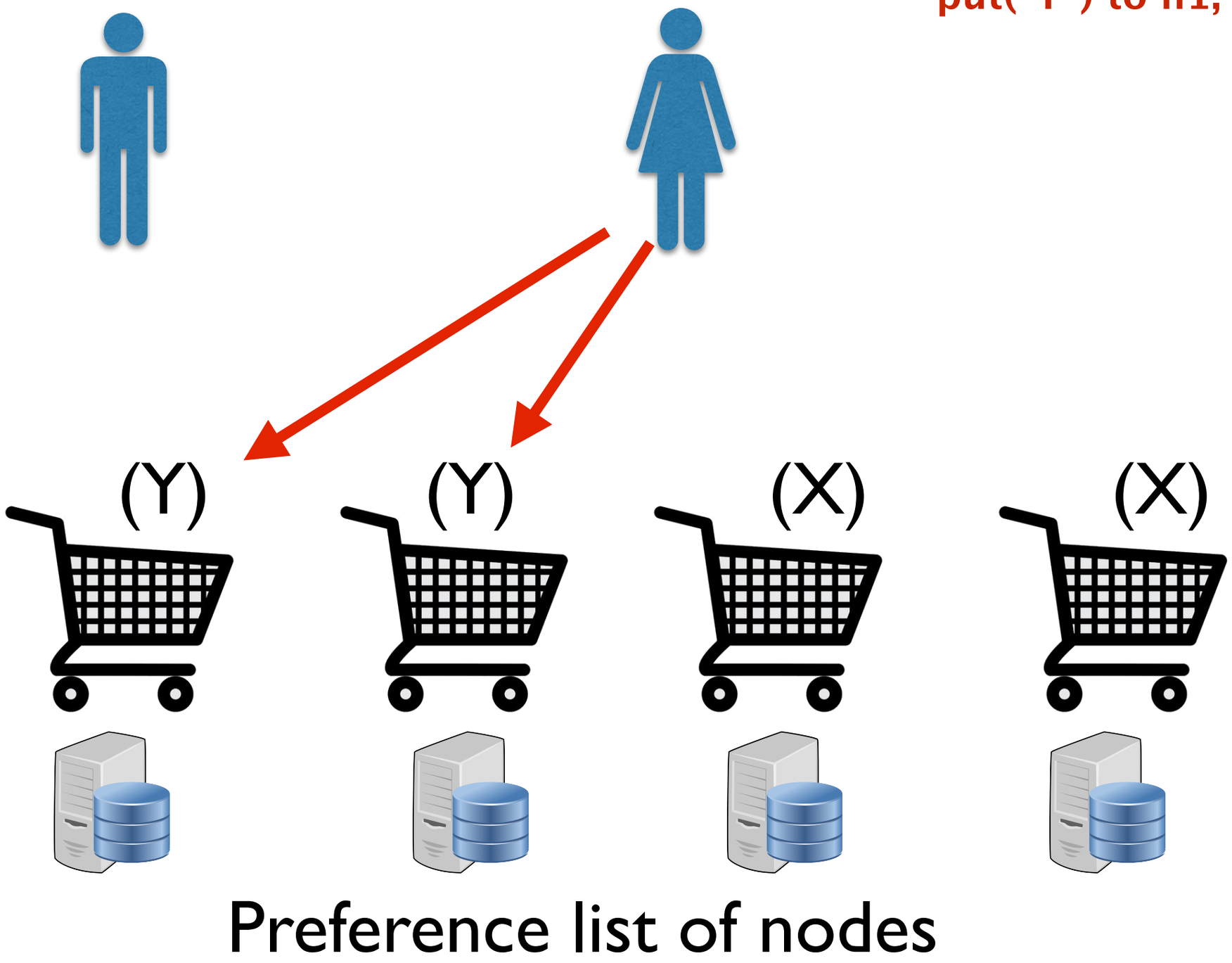


Preference list of nodes

N=3 R=2 W=2

Carts start empty

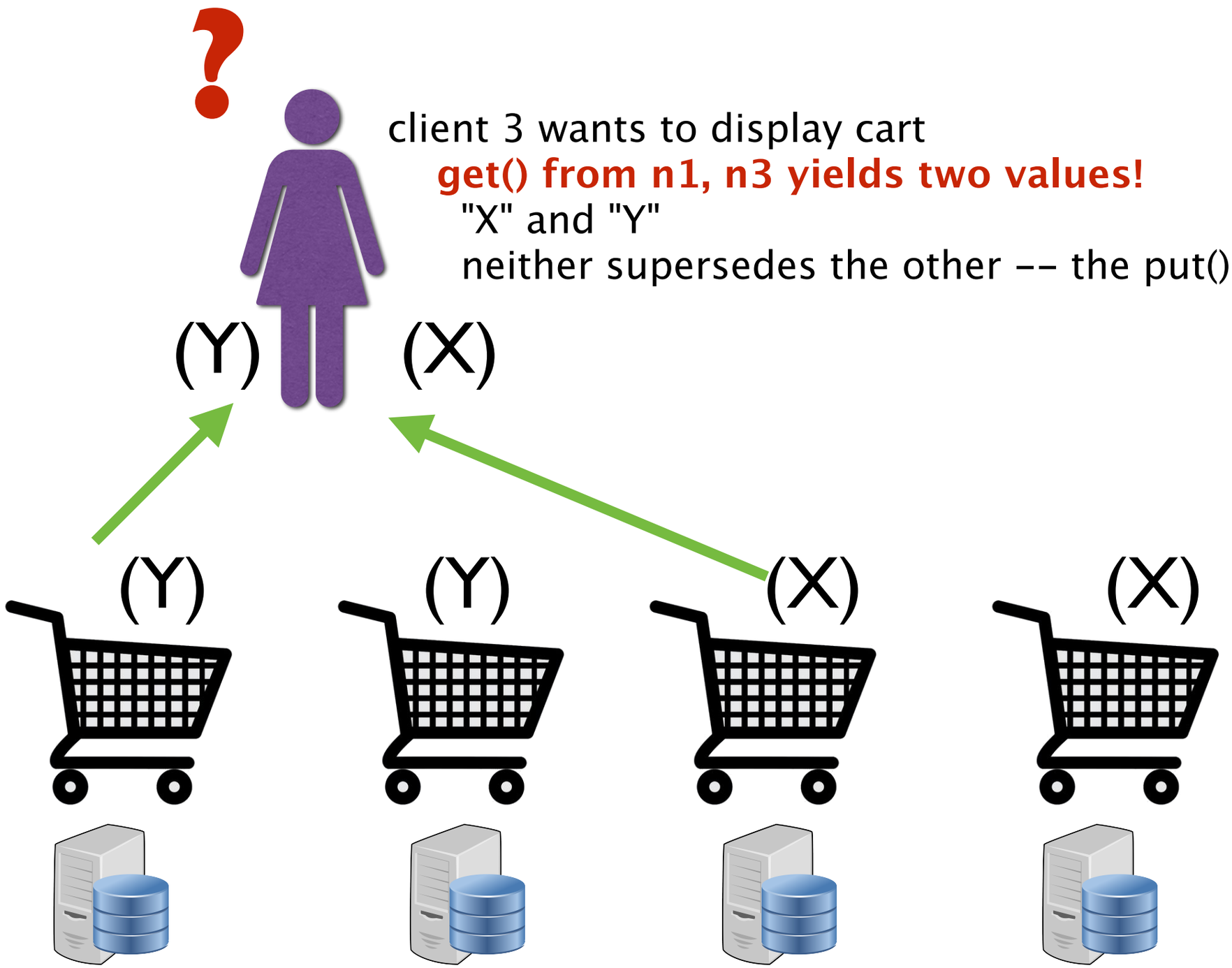
n1, n2 revive
client 2 wants to add Y
get() from n1, n2 yields ""
put("Y") to n1, n2



Preference list of nodes

N=3 R=2 W=2

Carts start empty



client 3 wants to display cart

get() from n1, n3 yields two values!

"X" and "Y"

neither supersedes the other -- the put()s conflicted

How to resolve conflicts on read?

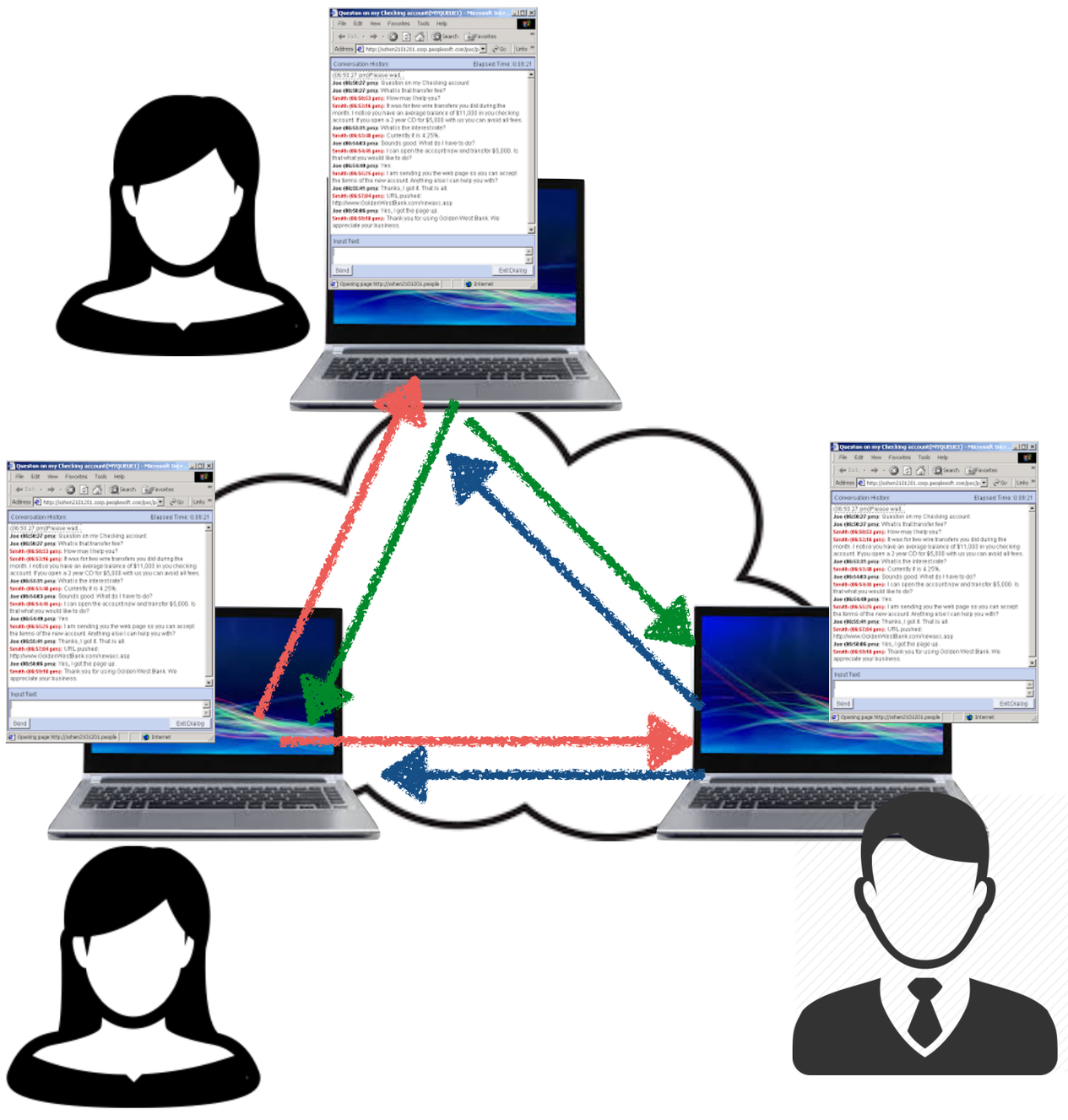
- Application dependent
  - Shopping cart:
    - Merge — union ?
      - might undelete Item X
- Maybe use “latest” for some Apps
- Write merge result back to Dynamo

Detection: Versioning

- Every write to a key produces in a new version that is derived from a prior version (immutable copies)
- Normal case one version is subsumed by the next as it is causally the successor but partitions/failures could result in multiple versions that get independent changes
- vector clocks to the rescue (again) to **detect conflicting versions**

Vector Clocks

Peer to Peer Chat

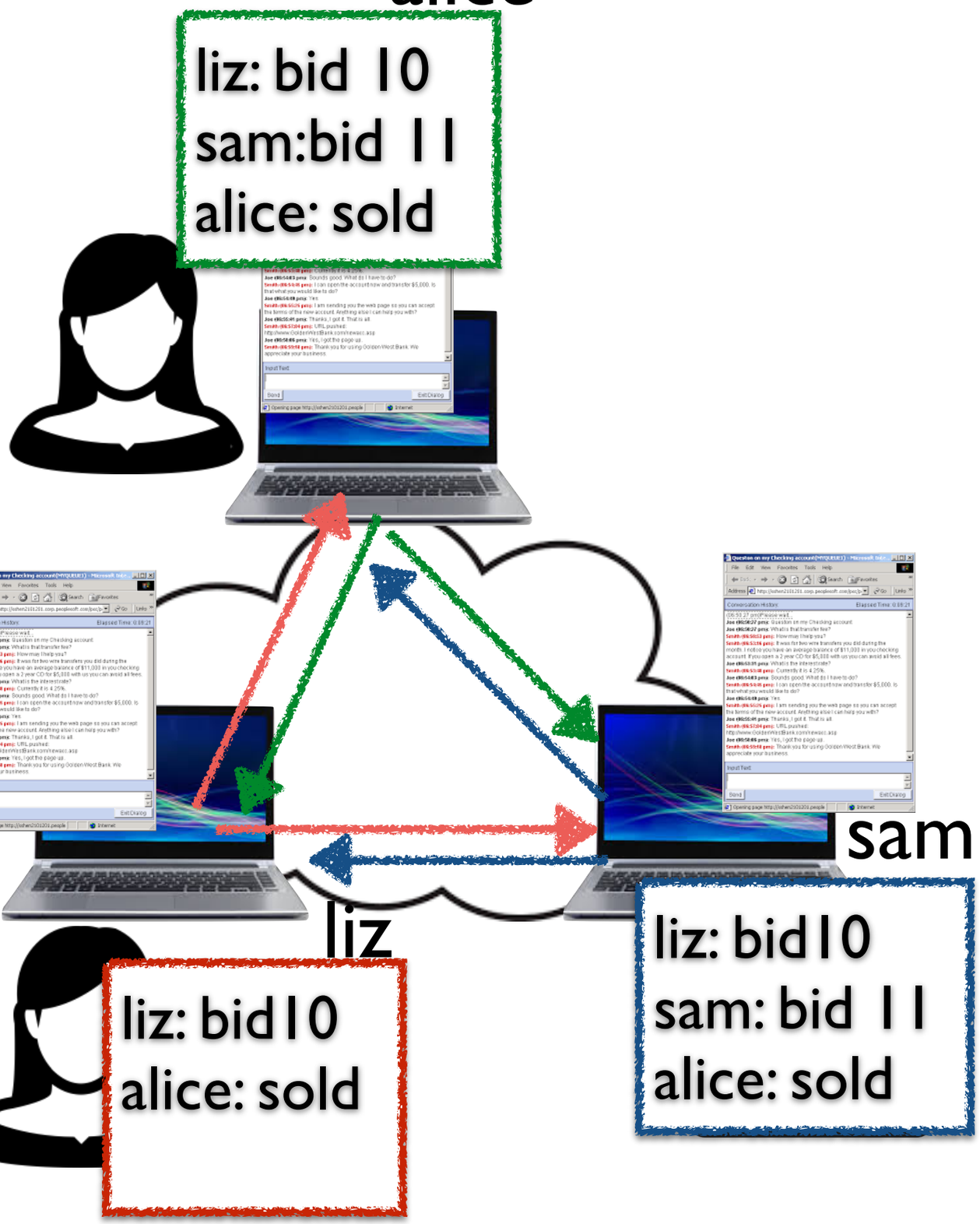


- Anyone can send message anytime — no traffic cop
- Simply receive messages and put to the end of log

Nice …

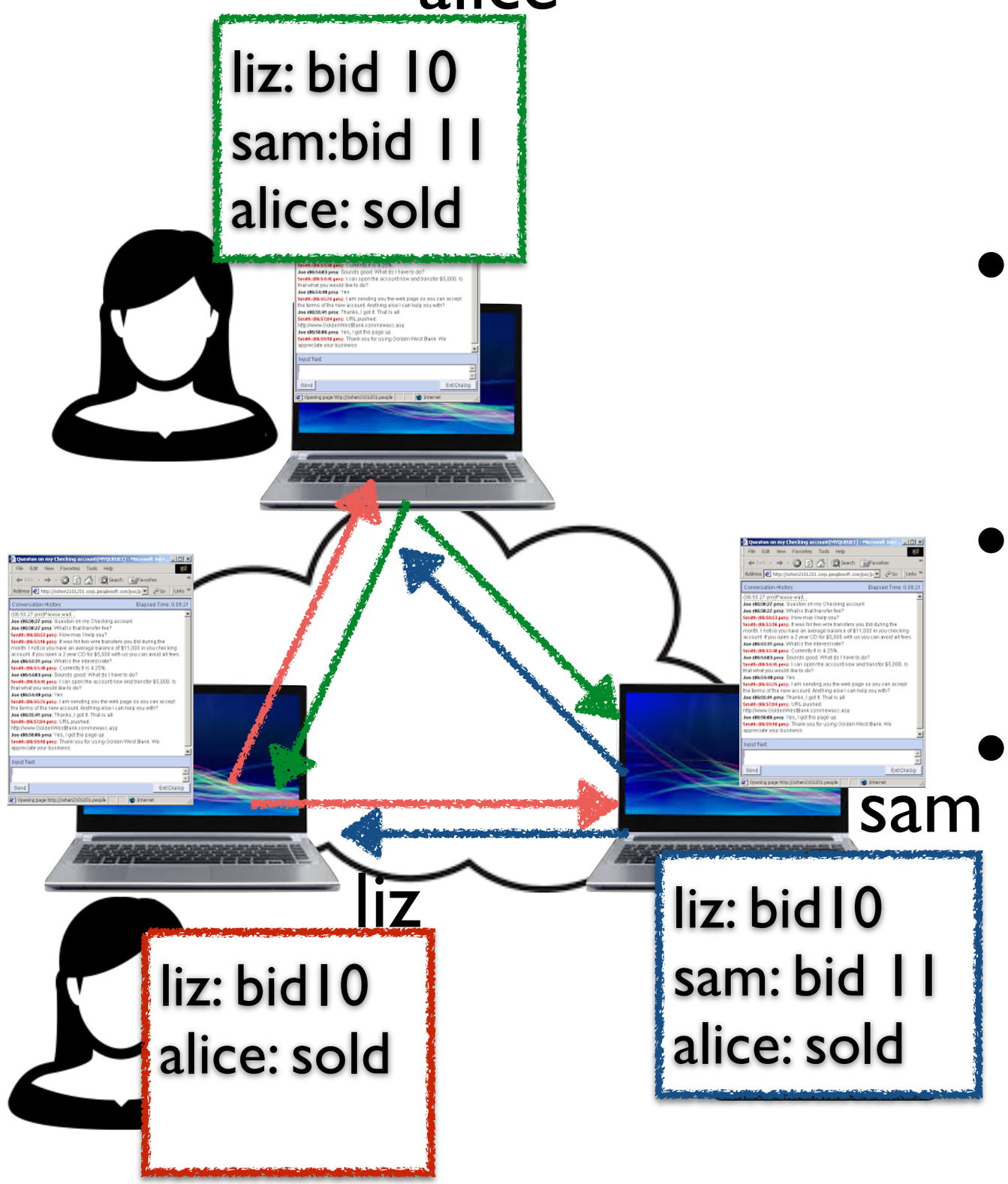
- No centralized ordering
- No centralized control

Do we care about order?



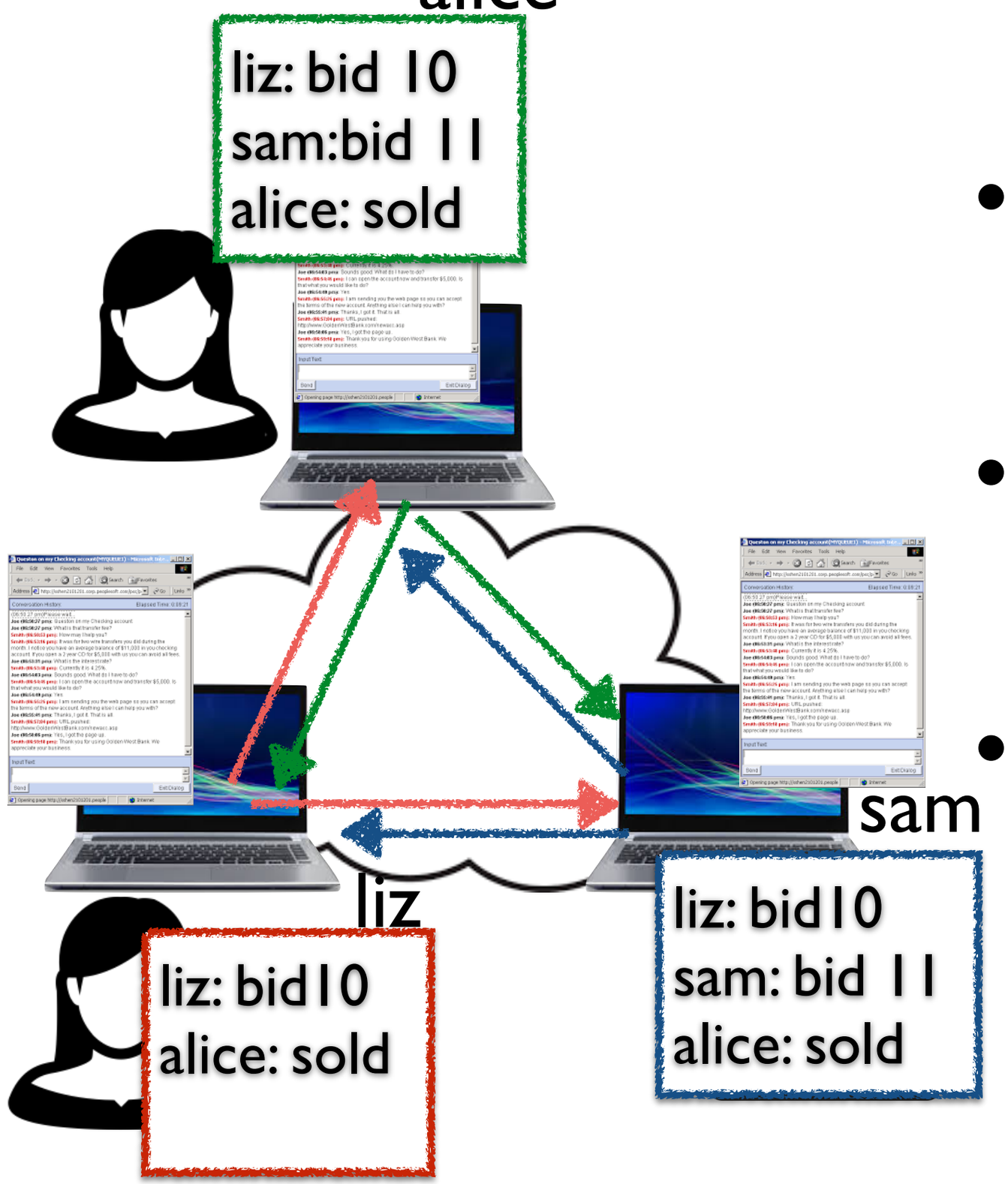
- But the network can deliver messages in arbitrary order to the different clients or lose a message
- suppose we are in an auction
- liz's view of the world is not causally consistent to how alice made her decision

Can we solve this without Centralization?



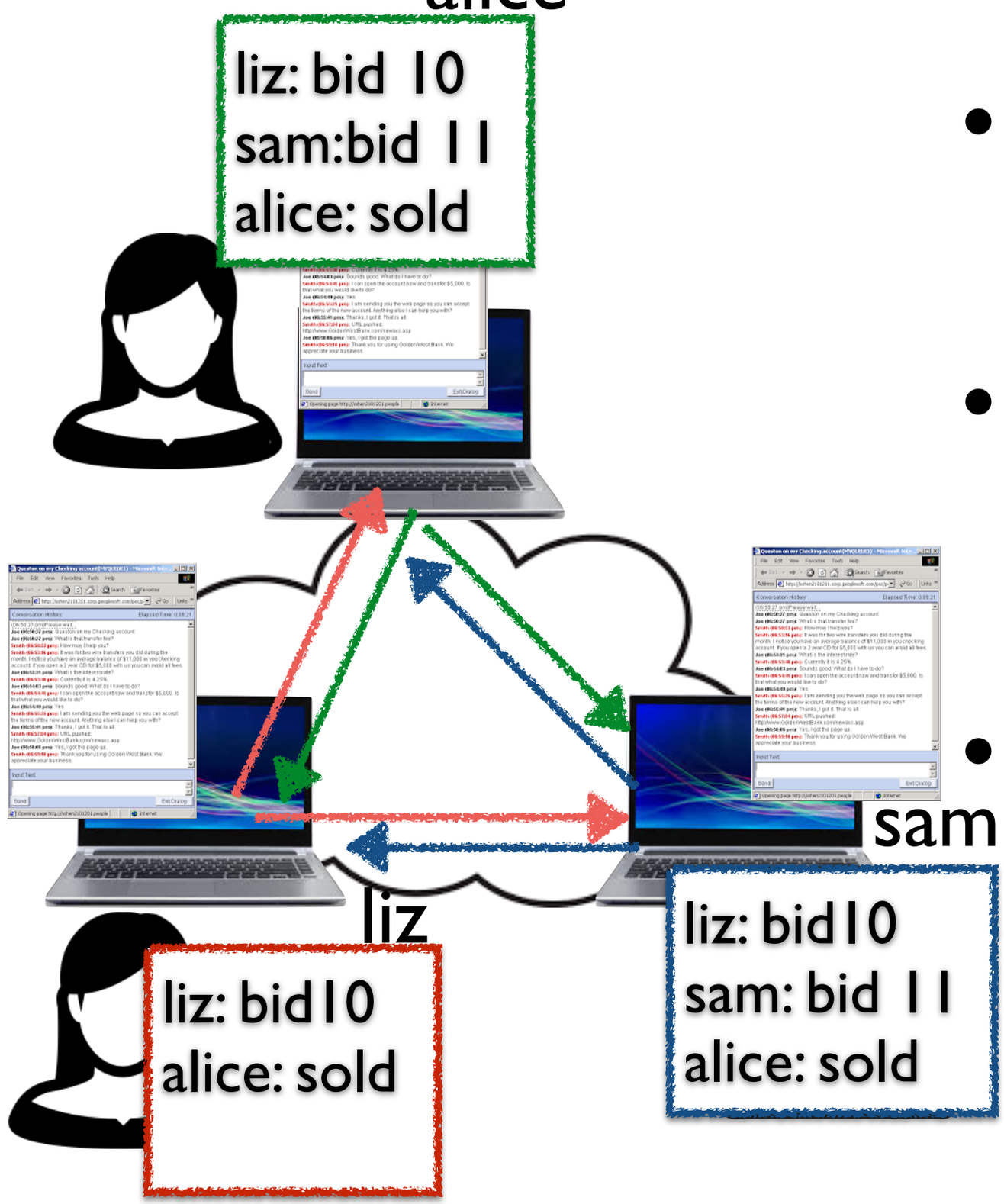
- Alice "computed" her message based on/relative to certain inputs
- Sam can only interpret her results if he sees those inputs
- How can we solve this problem?

Can we solve this without Centralization?



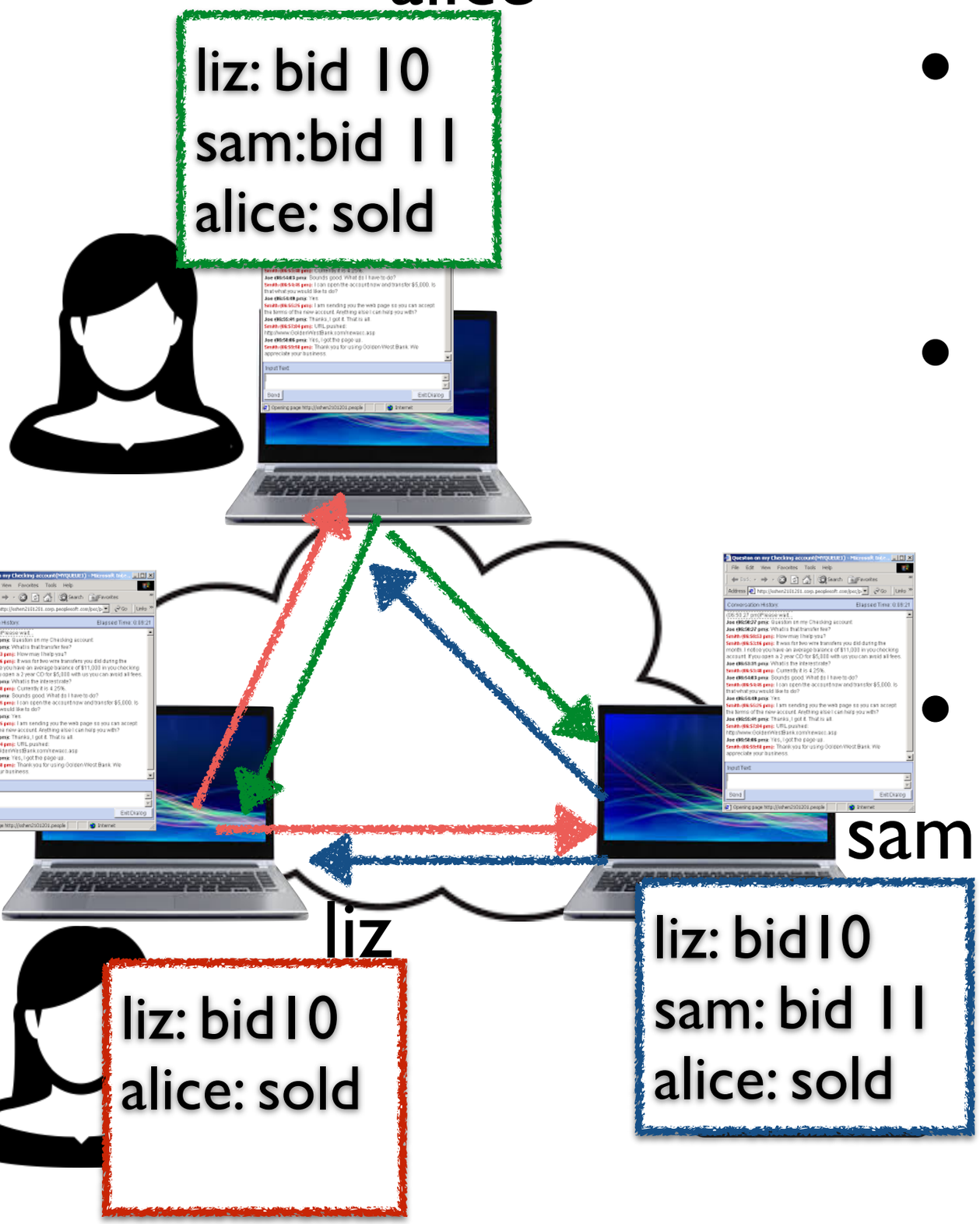
- If Alice includes what she has seen — what has gone into the decision she made
- Sam would know what CAUSED — lead to Alice's message
- If everyone does this we can get a sensible view without centralized control

X Causally Precedes Y



- DEFN: X causally precedes Y if
  - Locally: M0 does X and then M0 does Y — one thing happens before another
  - Globally: M0 does X and then M0 sends a message to M1, M1 receives the message and then does Y — X happens before Y
- Formally: this is transitive closure on local "causality" — A happened before B. B happened before C. Then A happened before C

X Causally Precedes Y

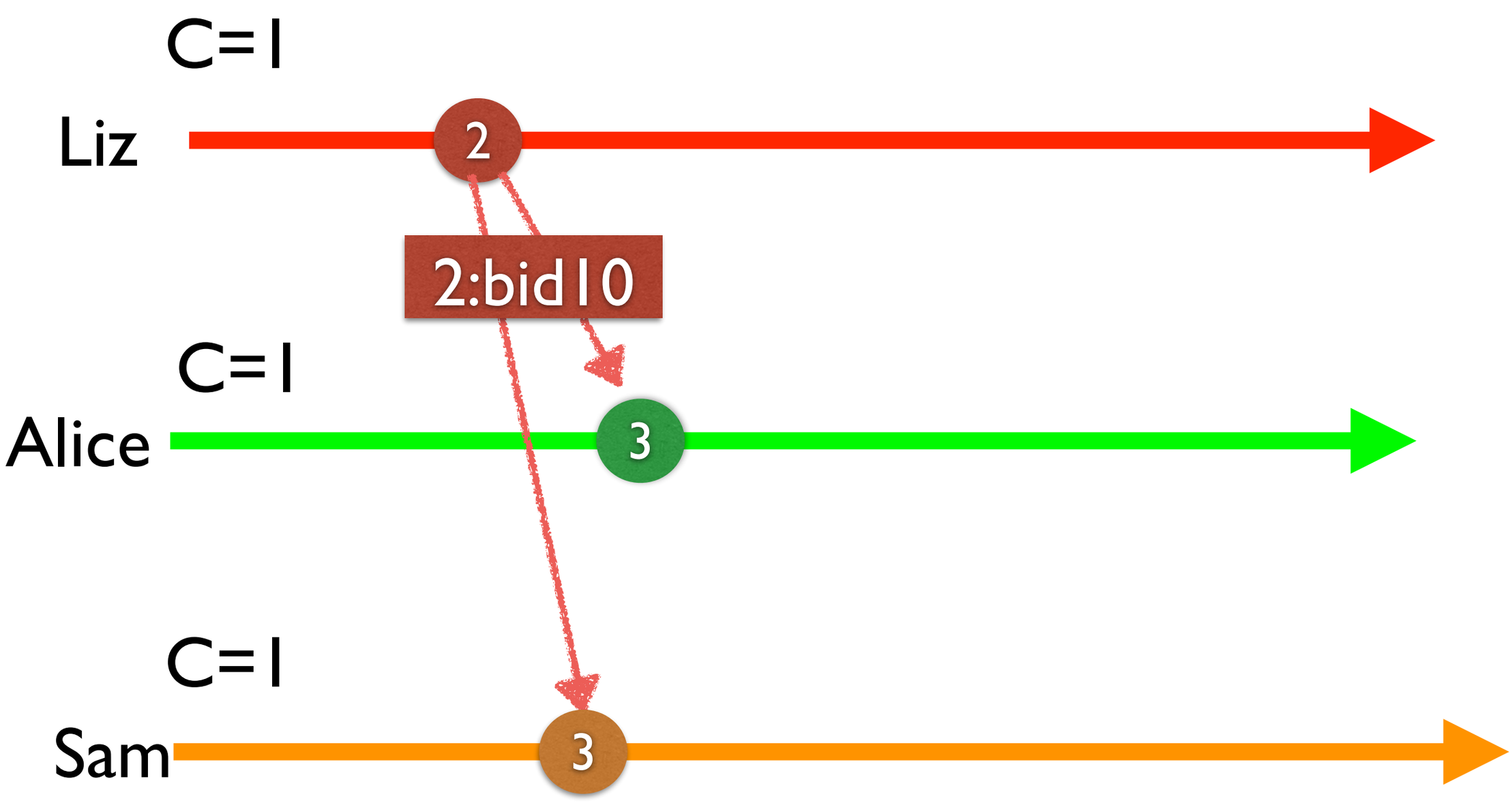


- if I do something (X) and then send a message to a remote machine
- If the remote system is going to do something (Y) that depends on the receipt of the message it can't do it before it received the message
- So if you do something after a message is received we say that it depends on something that preceded the send of the message from the source host. Thus X causally precedes Y

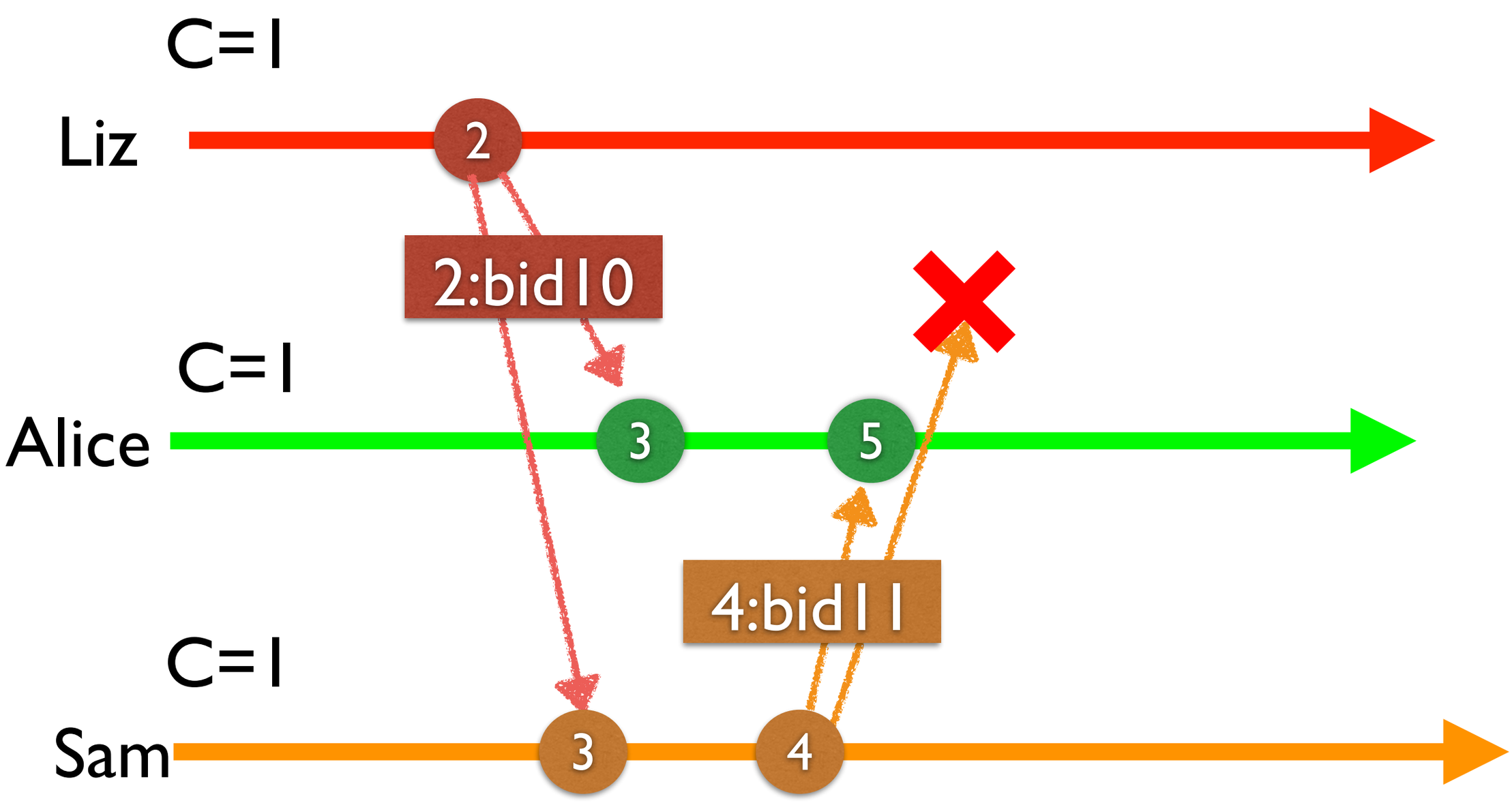
Causal Consistency

- Given the idea of Causally Precedes we can now come up strategies for capturing it (ordering events)
- if x causally precedes y, everyone should see x before y

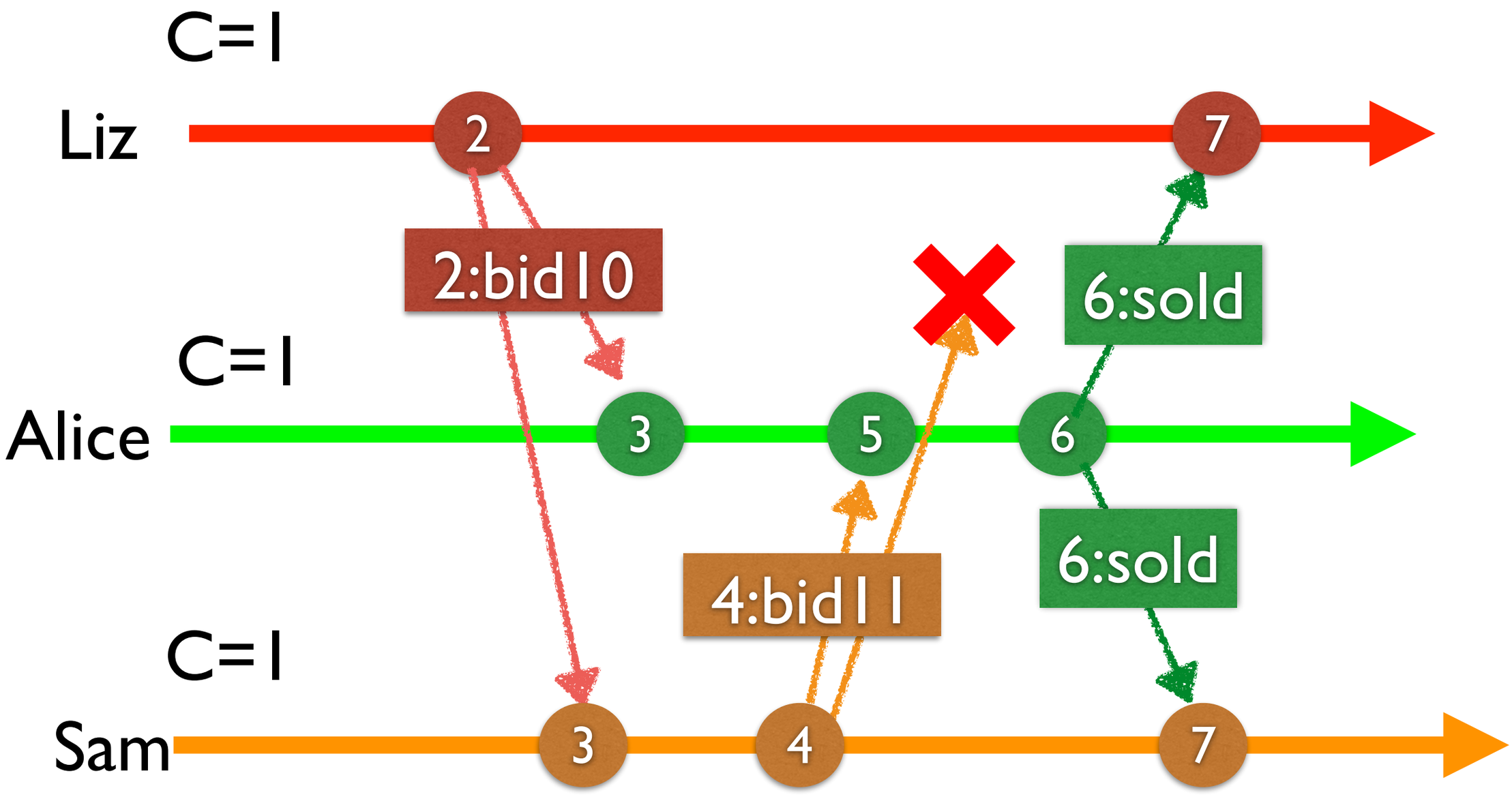
Lamport Logical Clocks not quite enough



Lamport Logical Clocks not quite enough

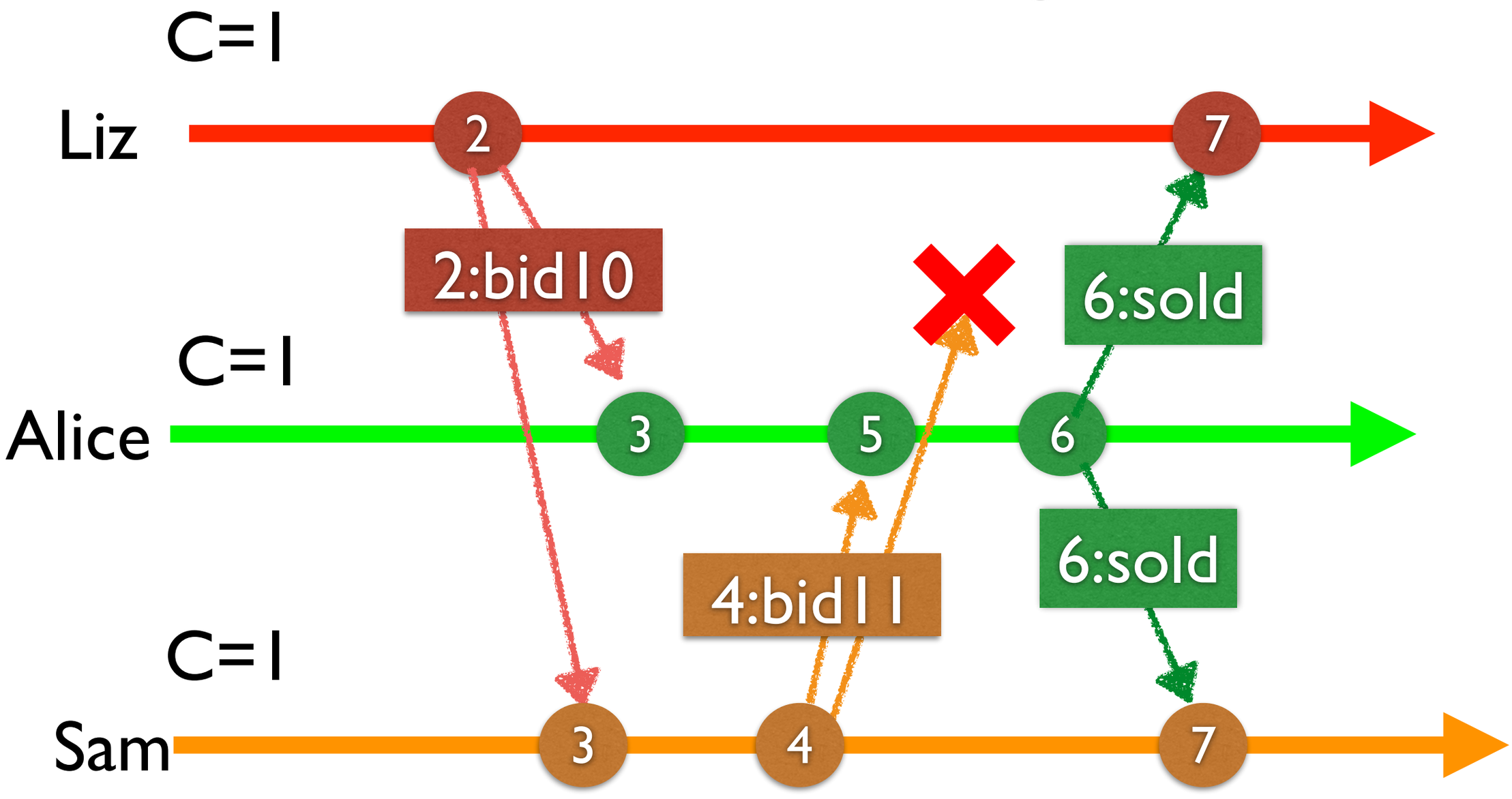


Lamport Logical Clocks not quite enough



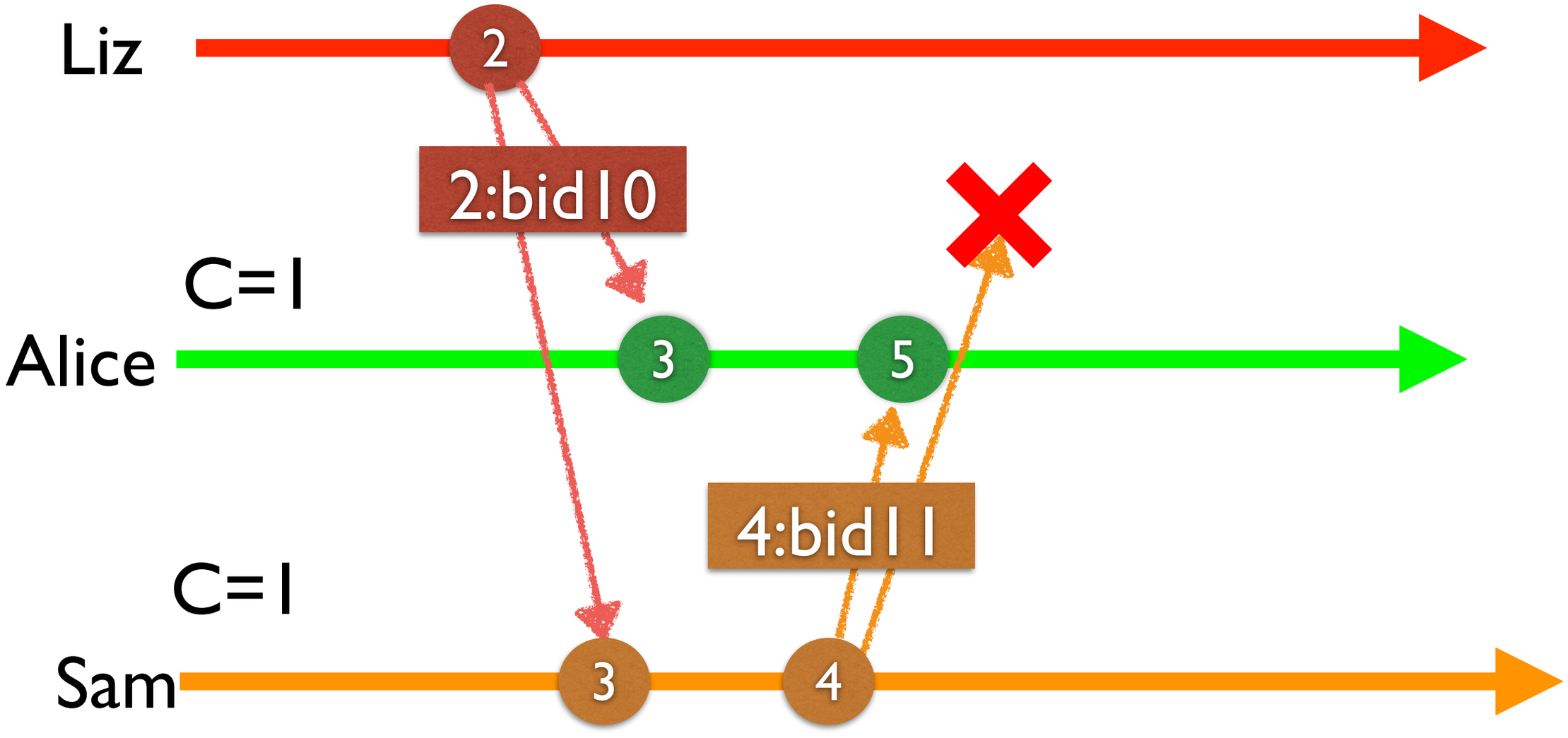
Lamport Logical Clocks not quite enough

What went wrong?



Causality

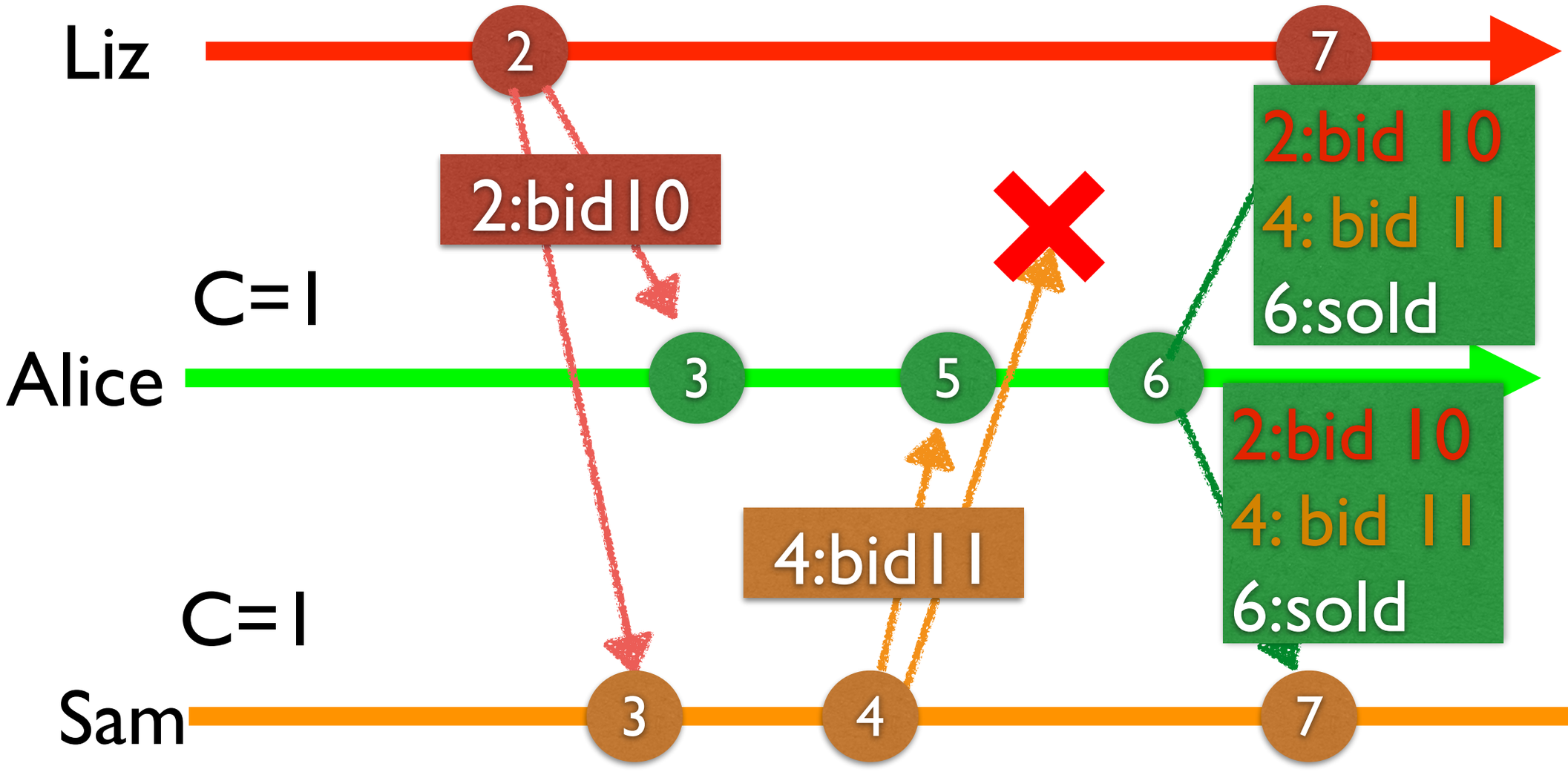
- NAIVE: Send entire history of all events with every message — thus you are sure to see all events I have seen



Causality

- NAIVE: Send entire history of all events with every message — thus you are sure to see all events I have seen

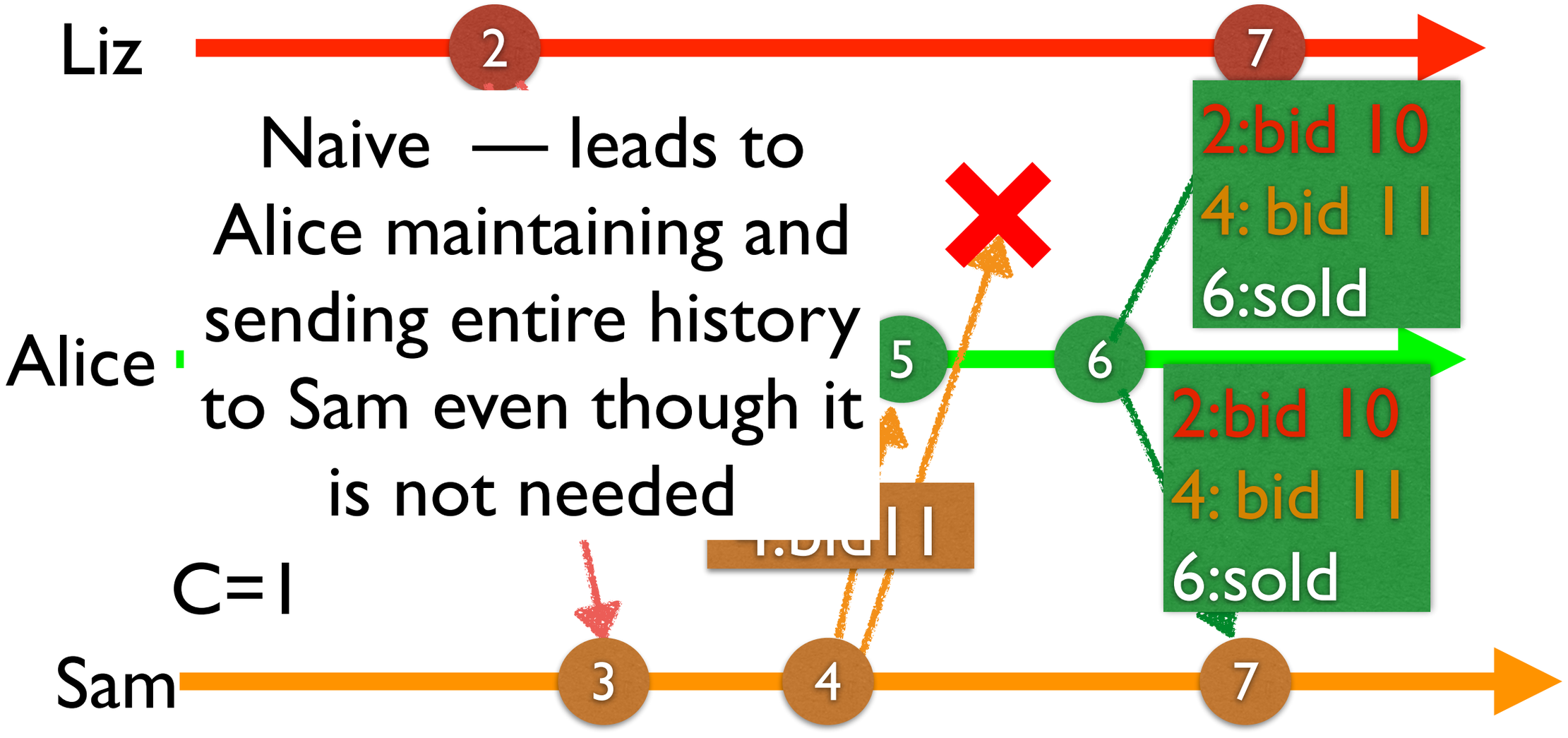
C=1



Based on these messages Alice sends a inclusive message

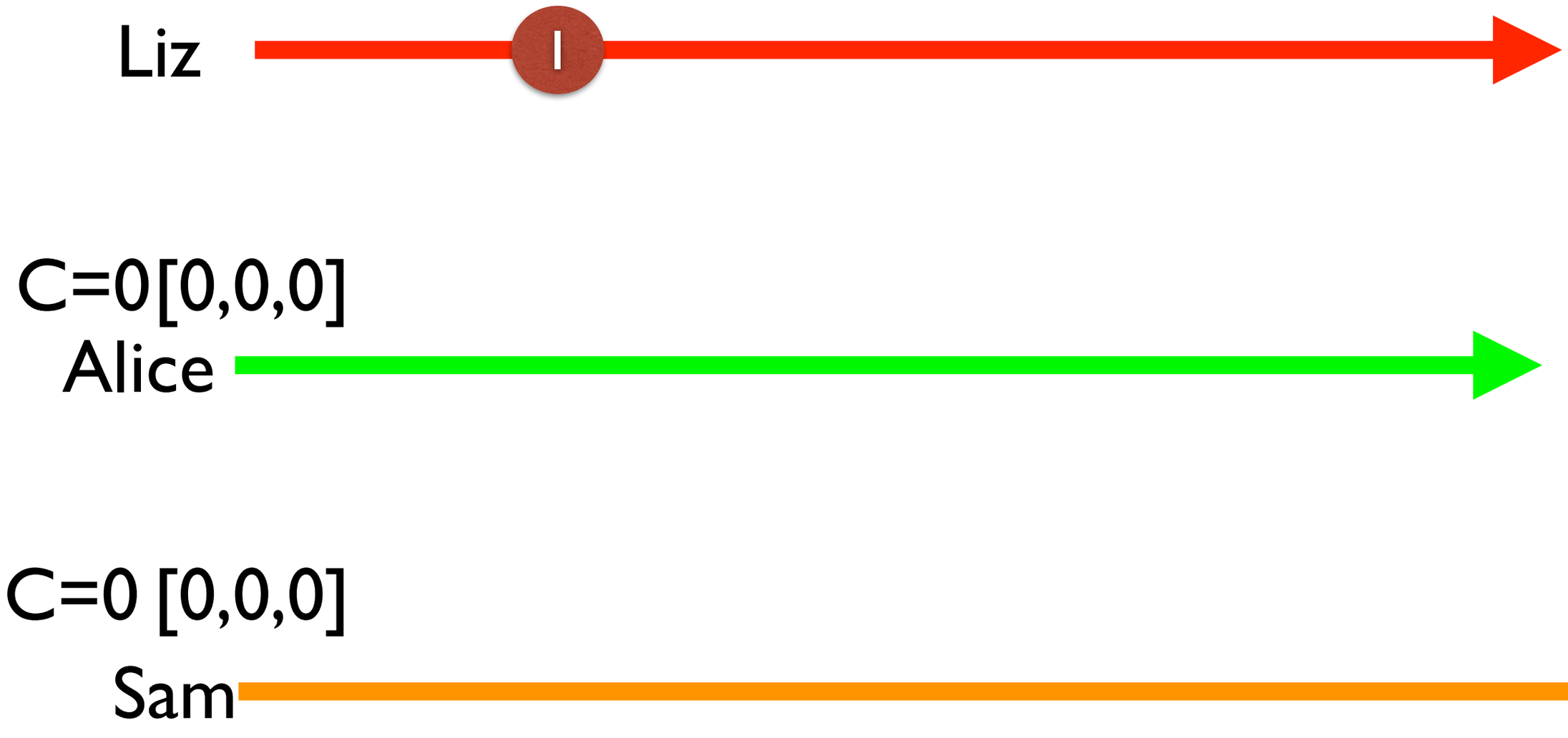
Causality

- NAIVE: Send entire history of all events with every message — thus you are sure to see all events I have seen



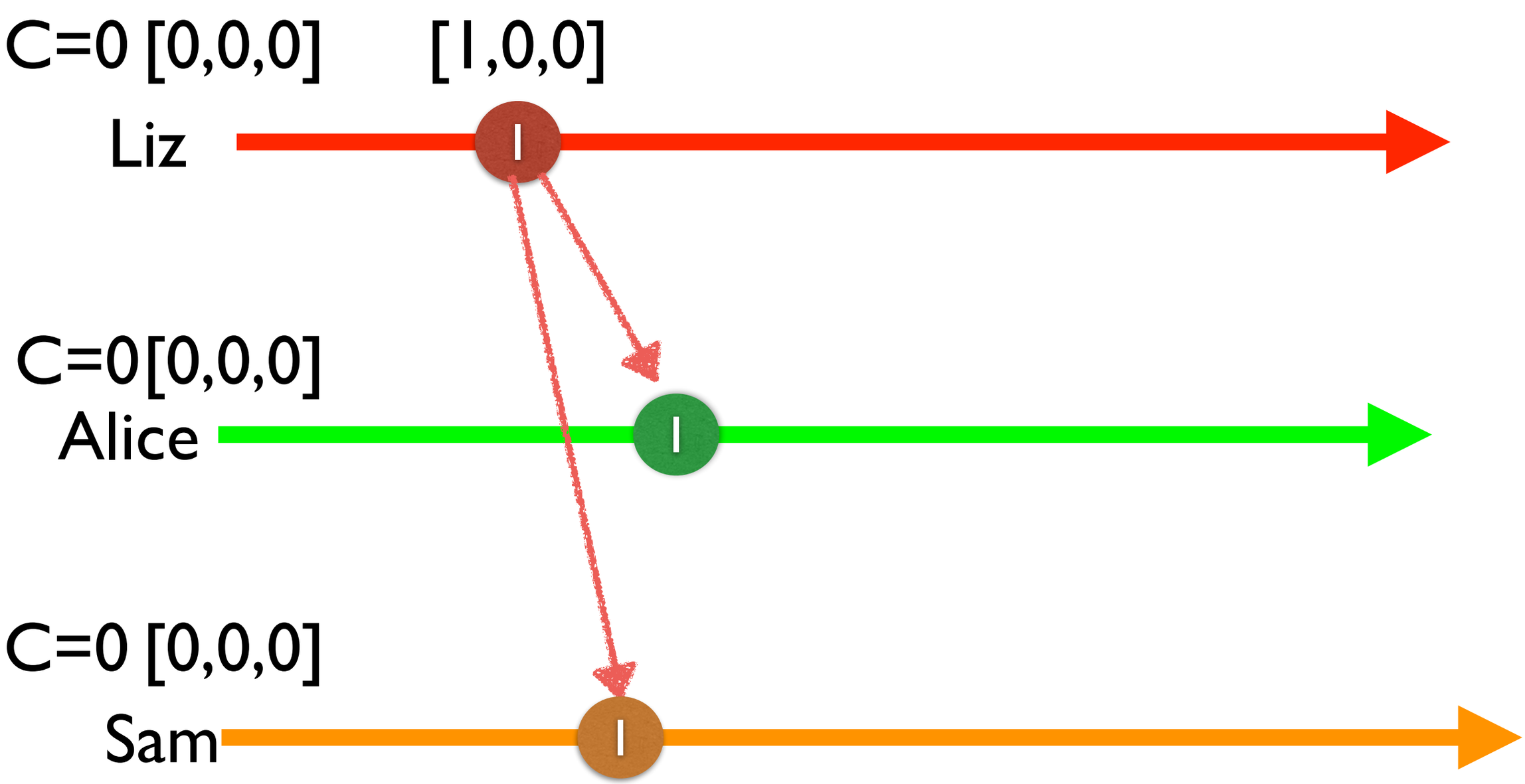
“Vector”

maintain a vector of logical clocks that are sent along with your messages — indicates what you have seen and from whom



“Vector”

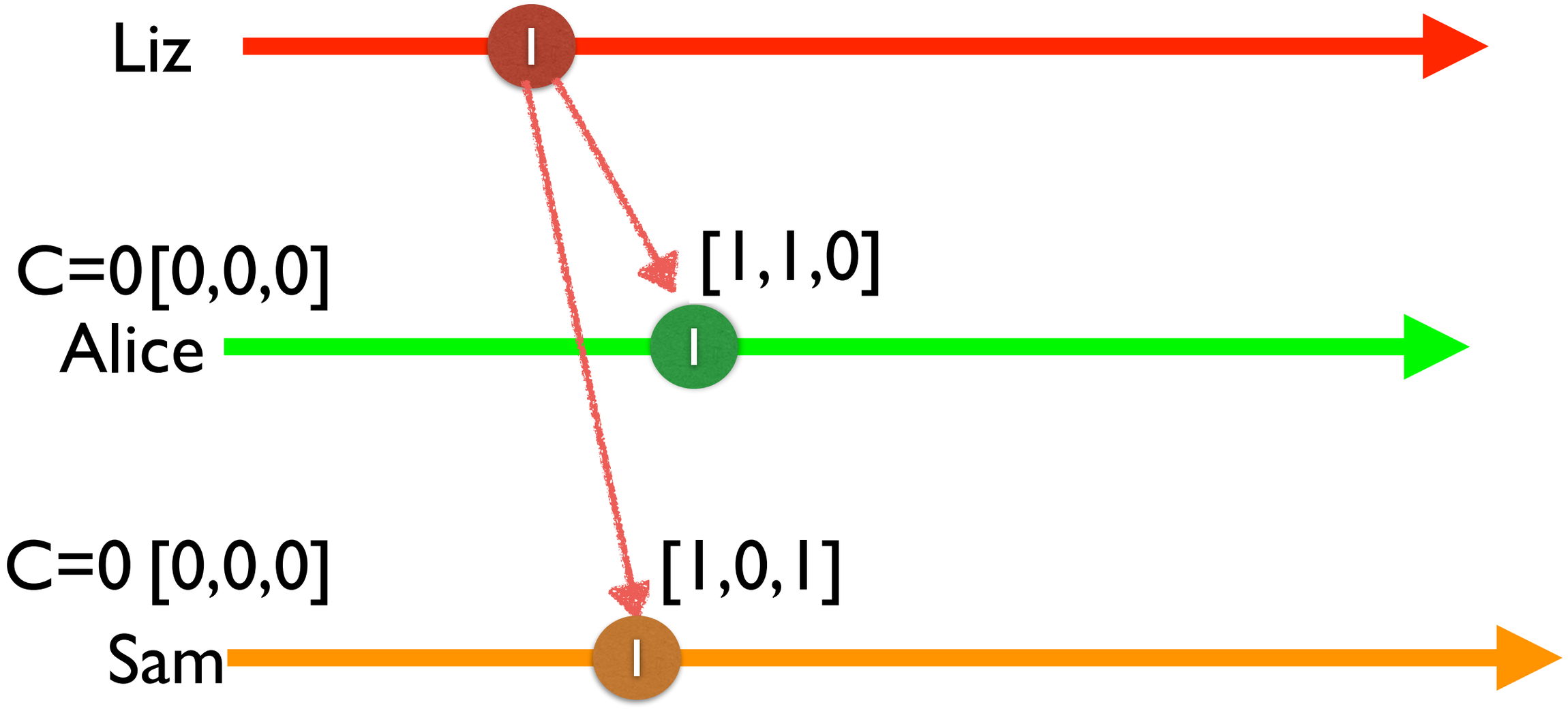
each node maintains a vector of clocks one for each node in the system and updates its own logical clock as usual and add sets vector TS for the messages



"Vector"

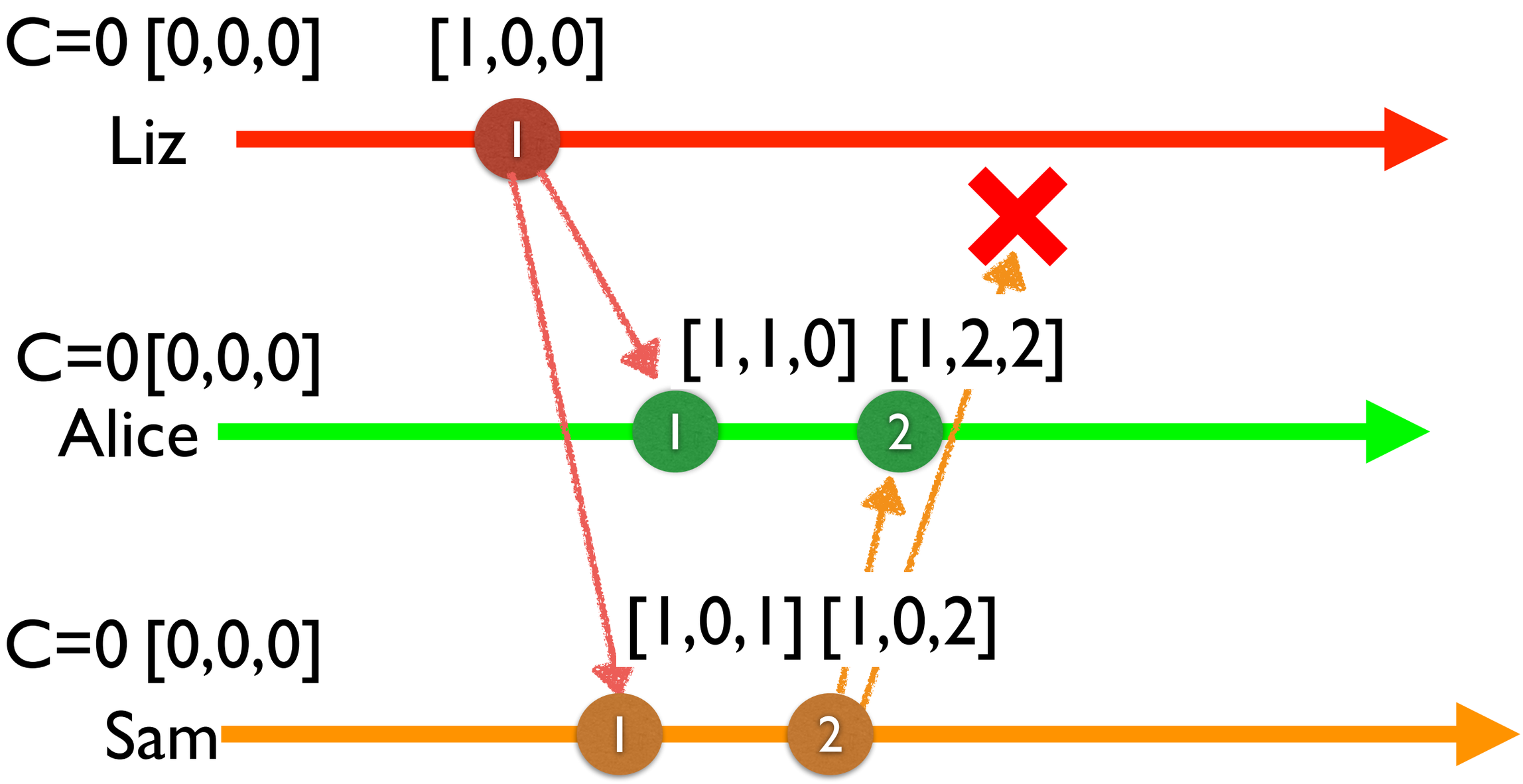
on receipt of message "merge" your vector to the element wise max of msg TS and your own vector

— updating your logical clock as usual

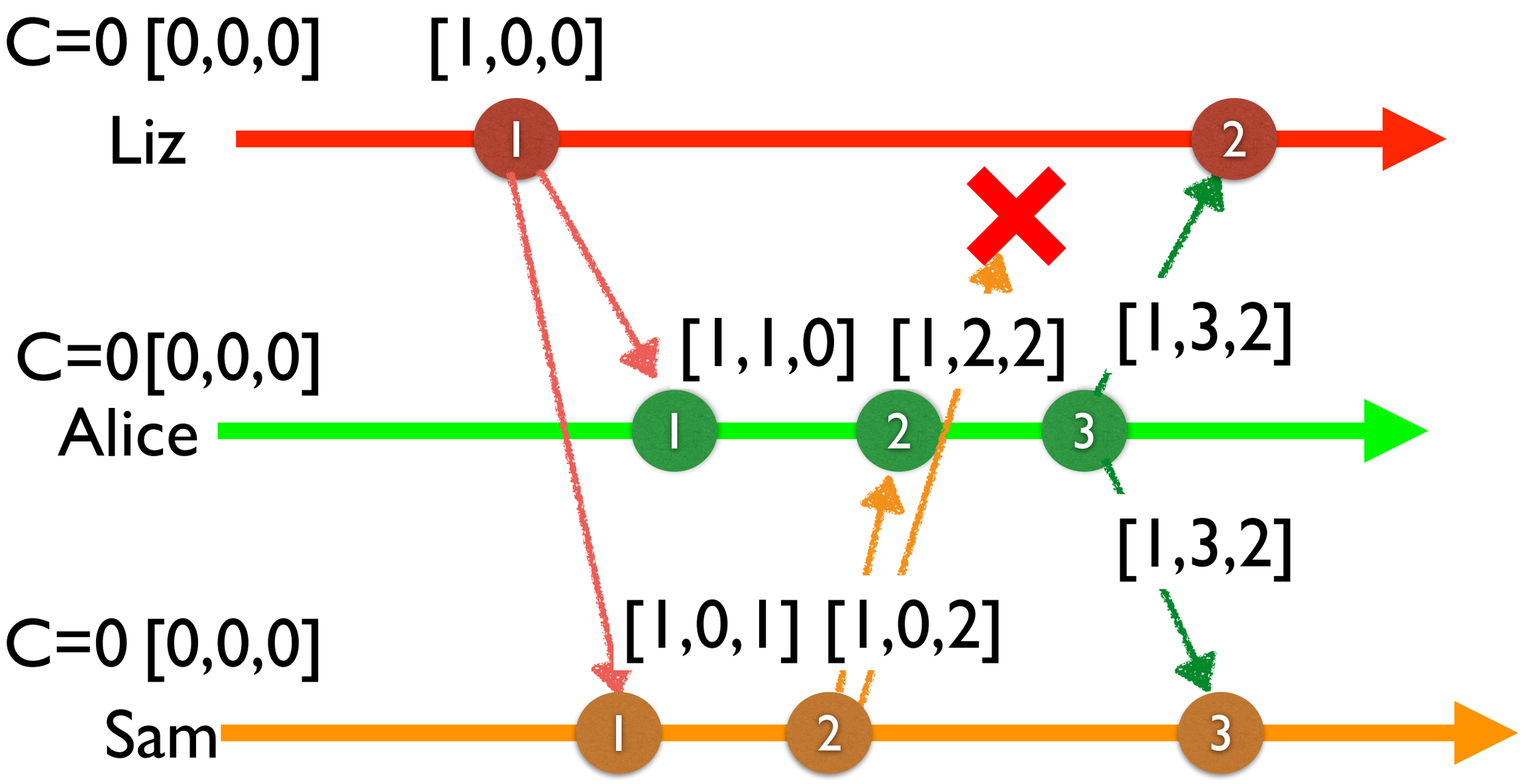


“Vector”

on receipt of message update your vector to the element wise max of incoming and your own



Liz can now see that she is missing some by inspecting her vector [1,0,**0**] with [1,3,**2**] to see that Alice sent her message with a different view of prior messages



Vector Clocks

- We can take this idea and formalize it into a Vector Clock.
- On which we can define a comparison operator that lets us detect if things are causally related.
  - are our histories derived from each other or are the parallel and forks of each other

Vector Clocks

$\underline{a}$ causally precedes $\underline{b}$ ($\underline{a}<\underline{b}$) :
iff for all i : $\underline{a}$[i] < $\underline{b}$[i]

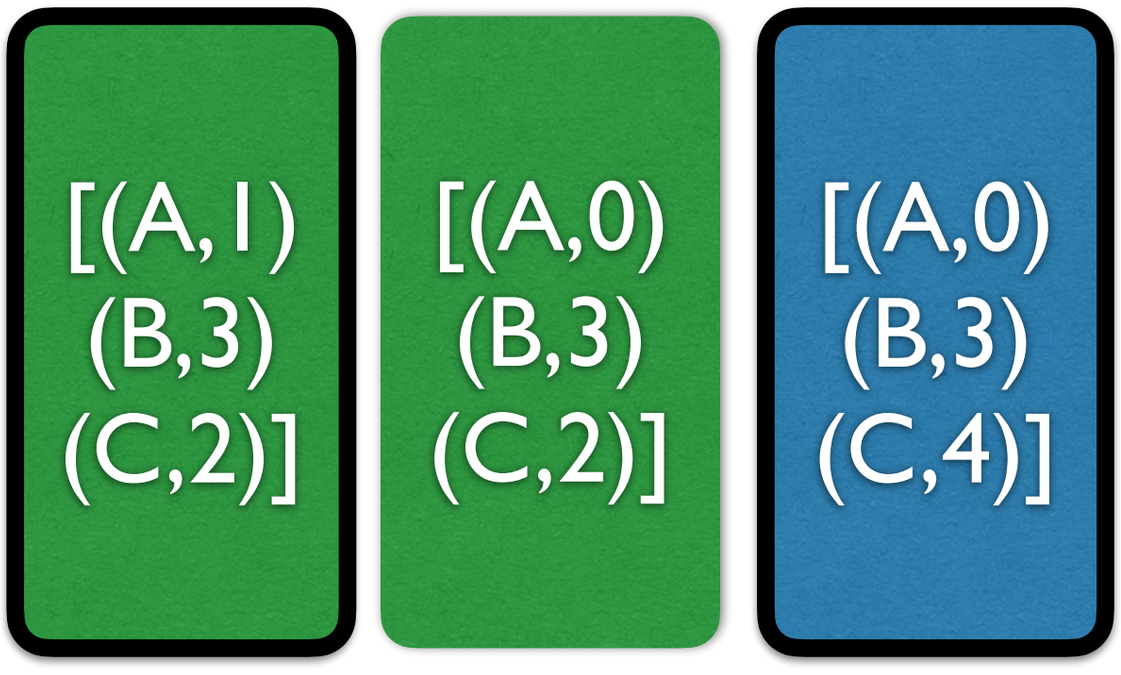
Neither $\underline{a}$ or $\underline{b}$ causally precedes each other ($\underline{a}||\underline{b}$) :
if there exists i,j such that
$\underline{a}$[i]<$\underline{b}$[i] and $\underline{a}$[j] > $\underline{b}$[j]

Dynamo use of VCs

A little more subtle

(not a fixed vector and set things up so that element wise <= indicates causally ordered)

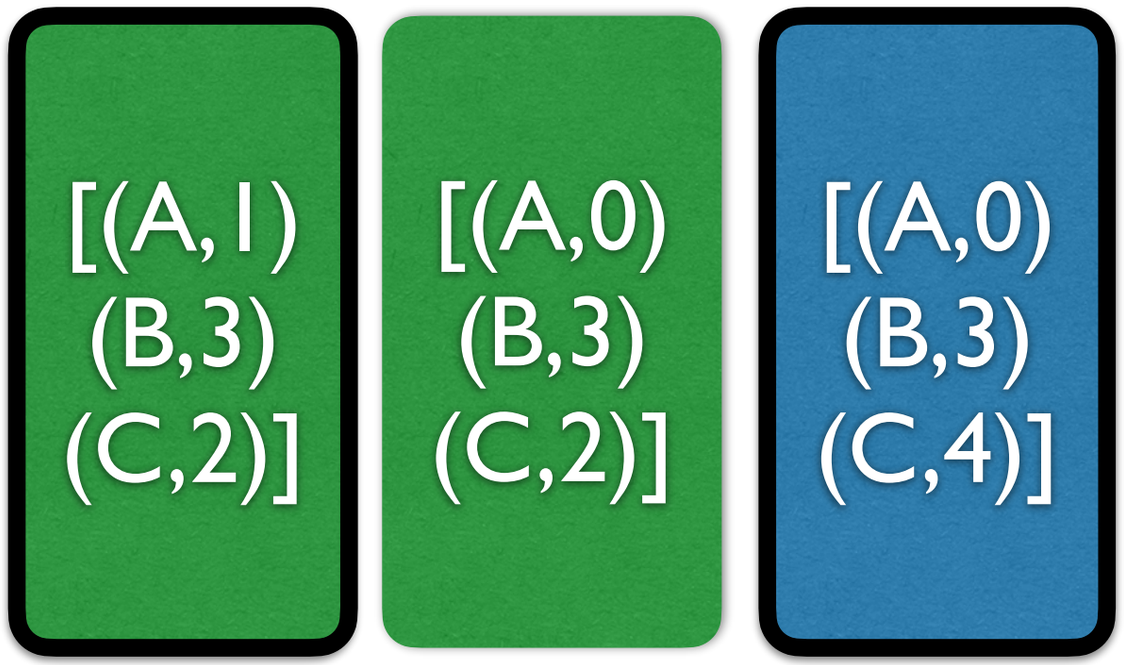
- Each replica of a data item has a vector clock
  - list of node,counter pairs
- Used to determine if versions are causally ordered or are parallel branches



<= Causally dependent else parallel branches

Dynamo use of VCs

- **put** to a an object pass in version (context) that the **put** is relative too eg version returned from **get**
- The **put** with this context is considered to resolve the conflicts and collapses back to a single branch

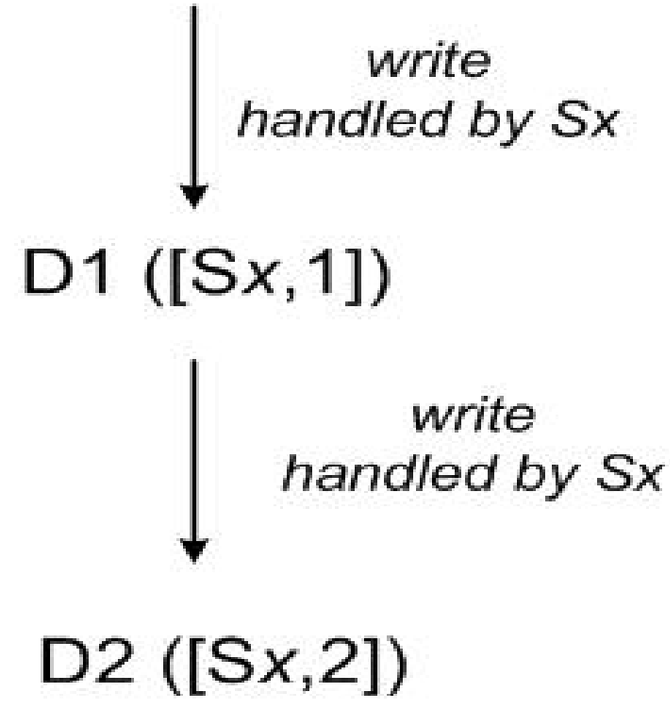


two versions returned: greens are “syntactically” collapsed

Vector clocks in Dynamo

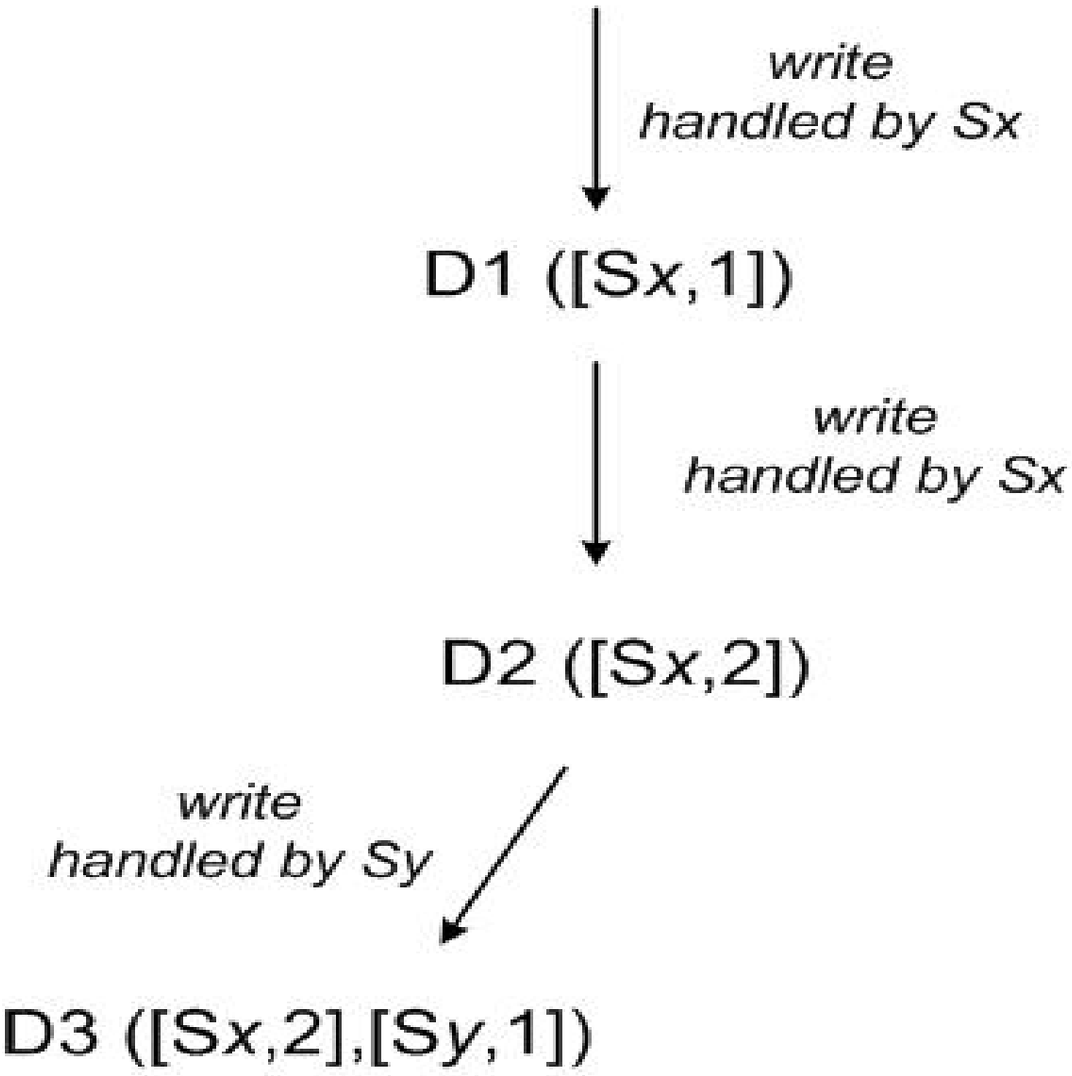
C1: Add item 1 to cart

Vector clocks in Dynamo



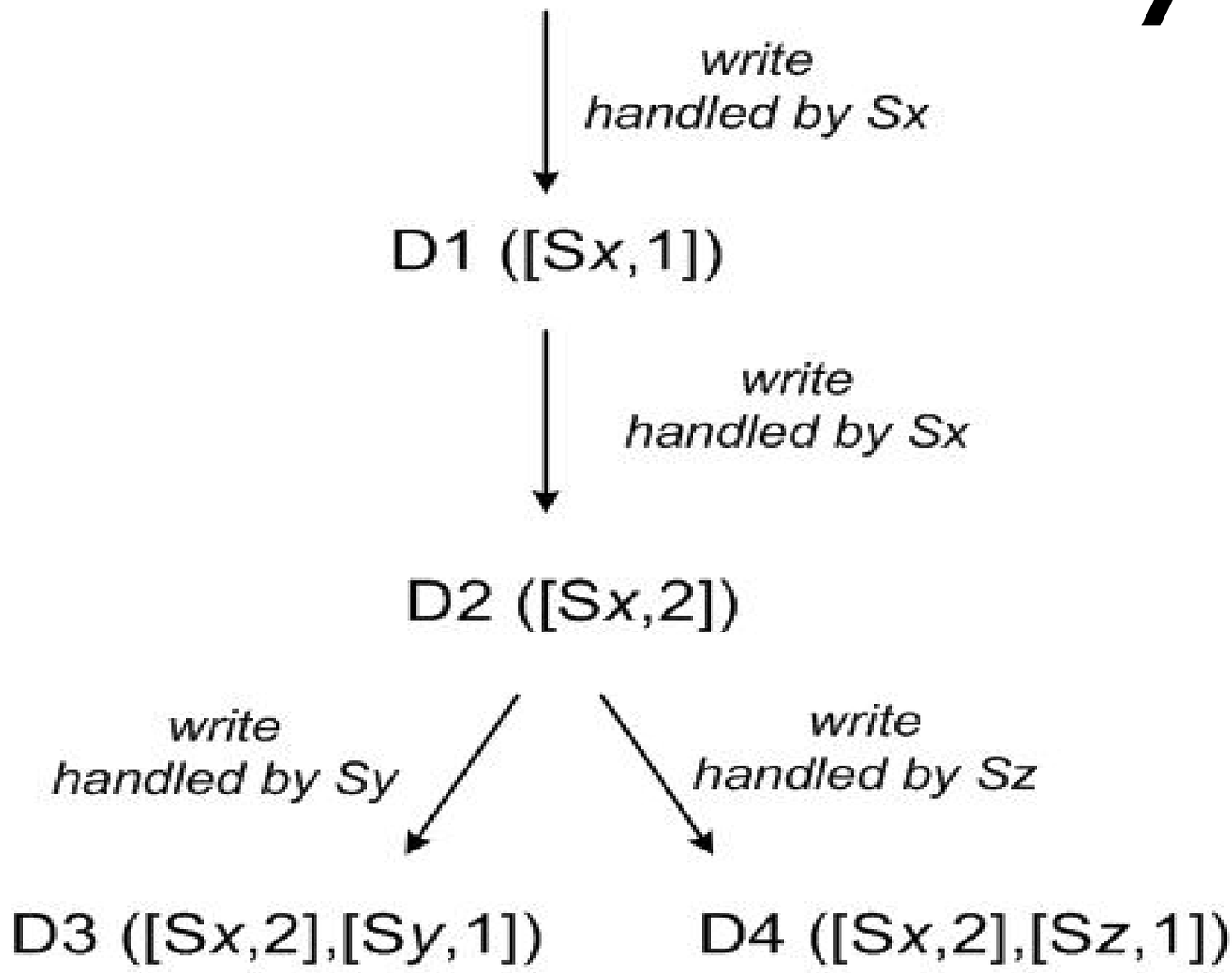
C1: another item (same coordinator)
this version overrides old version but there could be replicas who don't receive update

Vector clocks in Dynamo



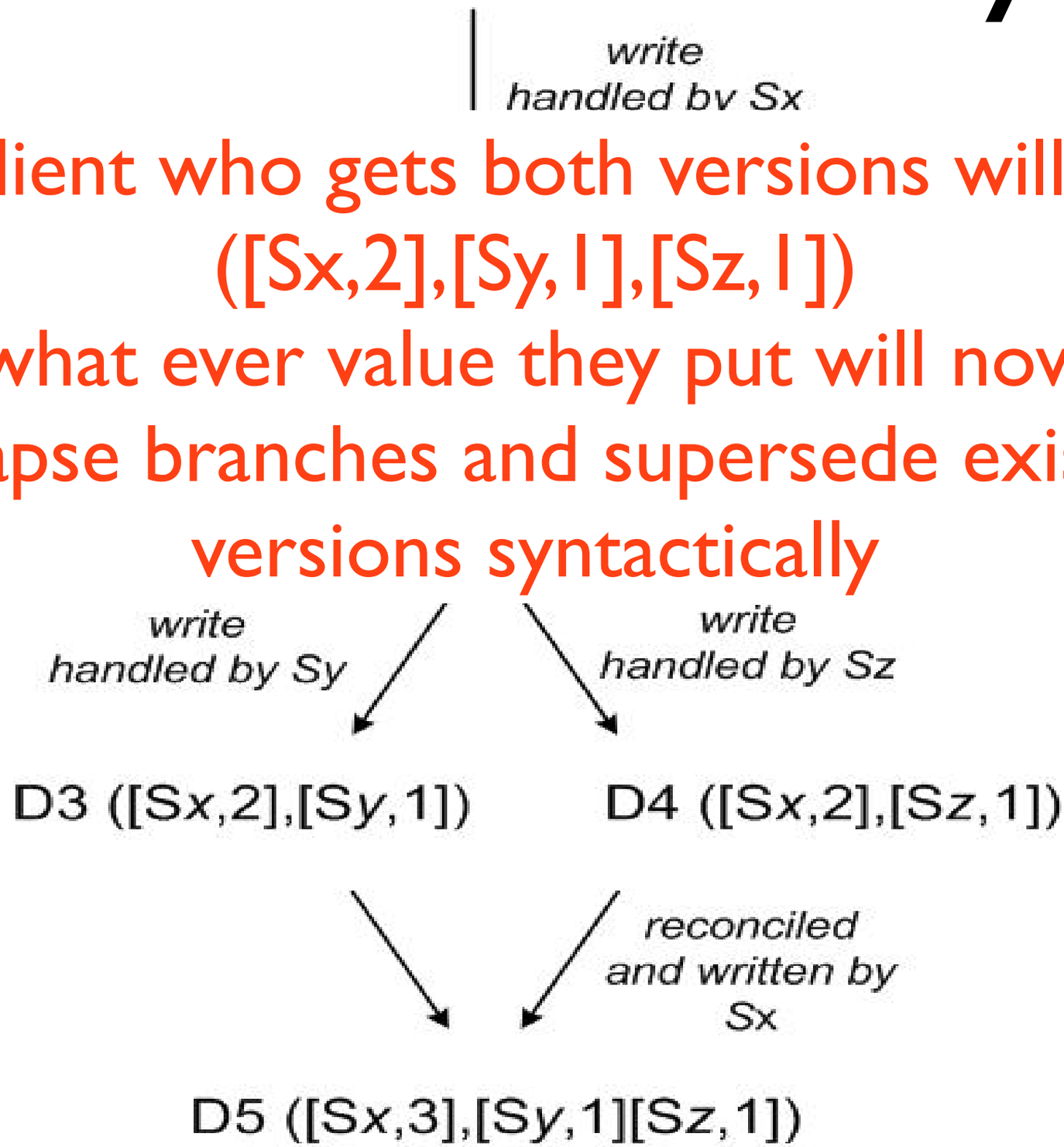
C1: another update
but different server
handles put

Vector clocks in Dynamo



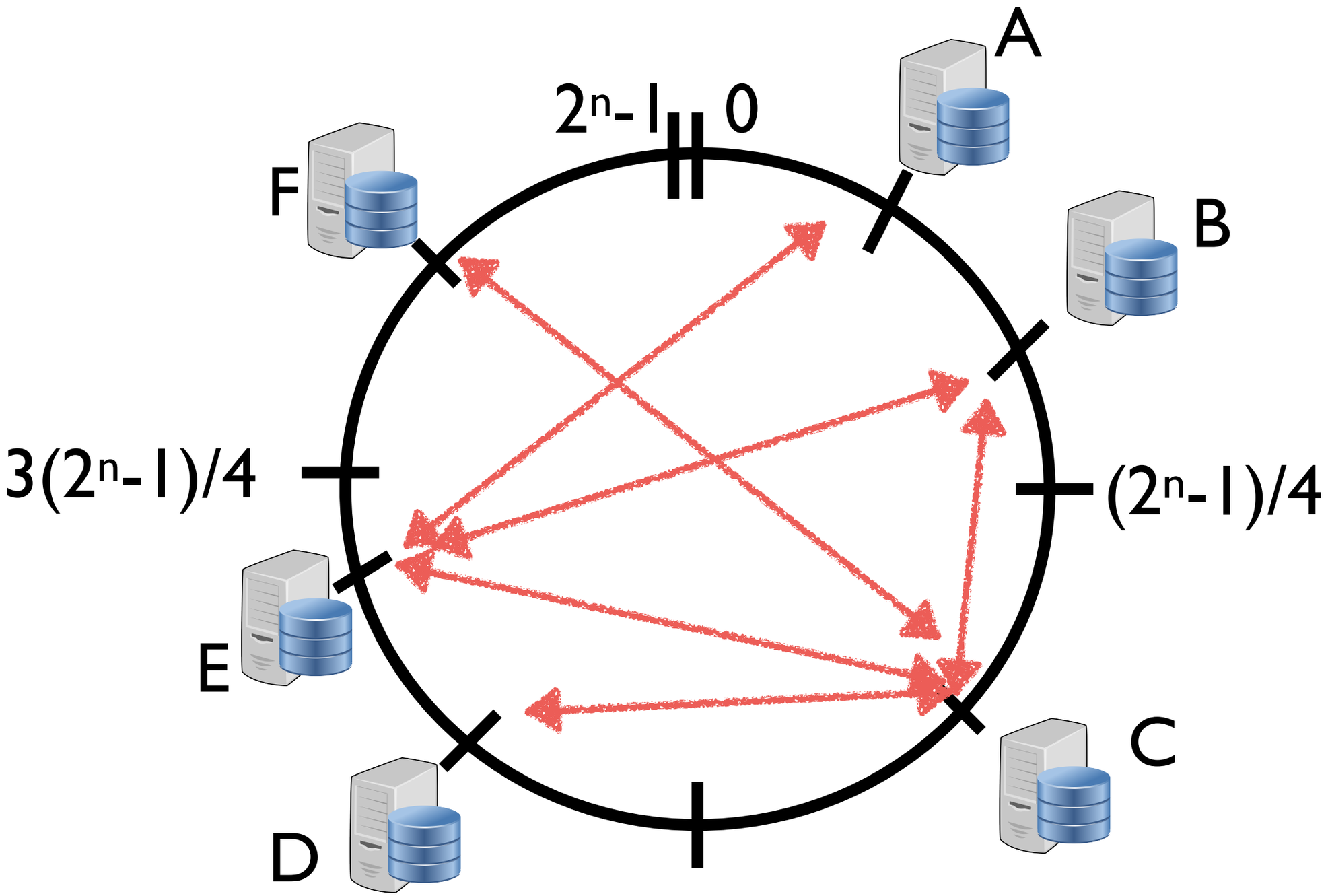
C2: different client reads D2 and then put is handled by yet another server

Vector clocks in Dynamo

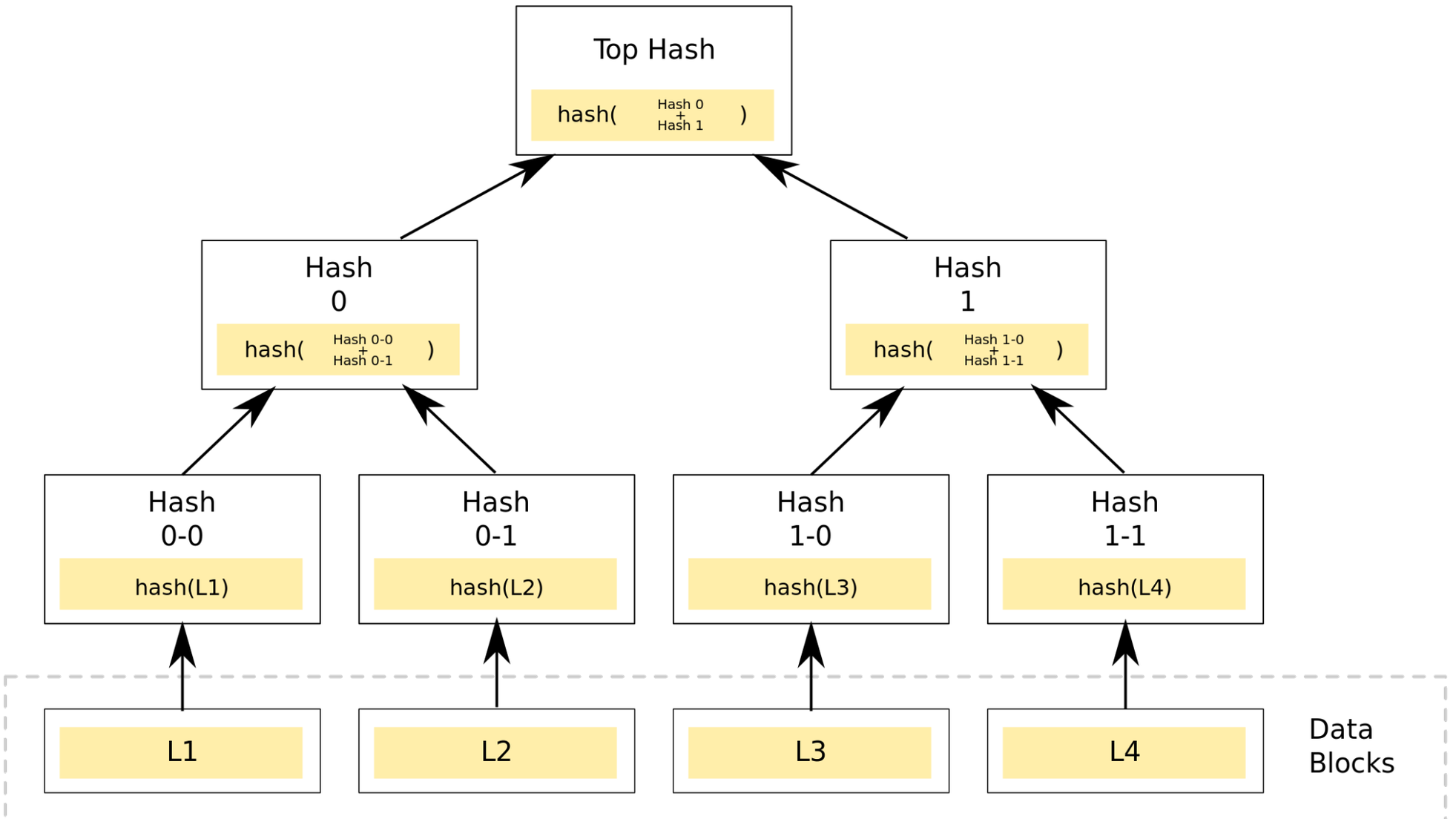


A client who gets both versions will get
([Sx,2],[Sy,1],[Sz,1])
what ever value they put will now
collapse branches and supersede existing
versions syntactically

Anti-Entropy Protocol

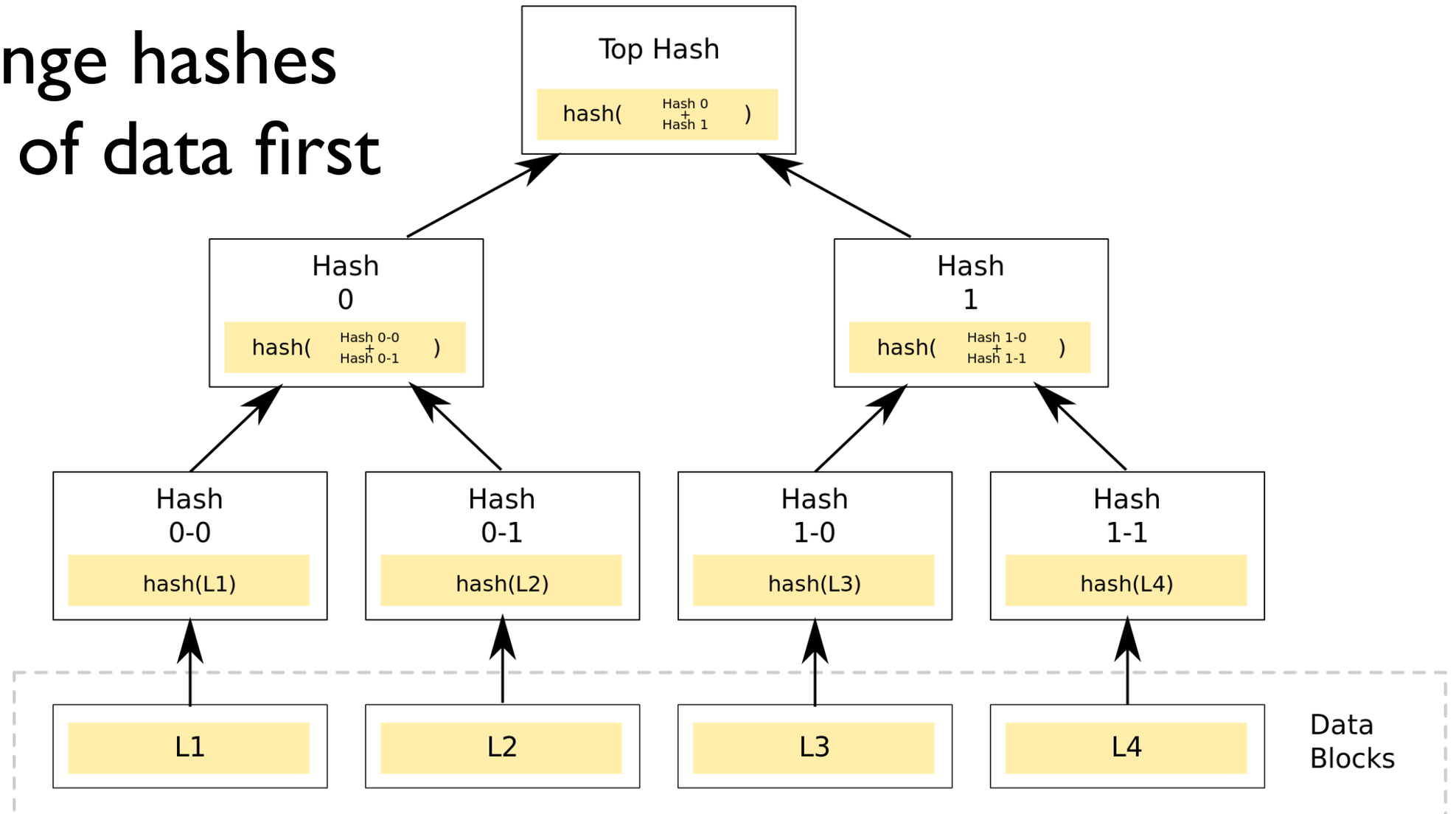


Merkle Tree


Top Hash
hash(Hash 0 + Hash 1)
Hash 0
hash(Hash 0-0 + Hash 0-1)
Hash 1
hash(Hash 1-0 + Hash 1-1)
Hash 0-0
hash(L1)
Hash 0-1
hash(L2)
Hash 1-0
hash(L3)
Hash 1-1
hash(L4)
L1
L2
L3
L4
Data Blocks

Merkle Tree

Exchange hashes
instead of data first



Exchange data only if hash values are different!

Data Center failure?

Data Center failure?

Preference list constructed to cross datacenter & High speed links