CONSENSUS

propose →  → decide

- **Validity** – If a process decides v, then v was proposed by some process
- **Agreement** – No two correct process decide differently
- **Integrity** – No correct process decides twice
- **Termination** – Every correct process eventually decides some value

MESSAGES TAKE TIME

Does it matter how much?

OF COURSE!

AND YET...

Should it matter for
CORRECTNESS?

Assumptions are vulnerabilities!

ASYNCHRONOUS SYSTEMS

NO centralized clock

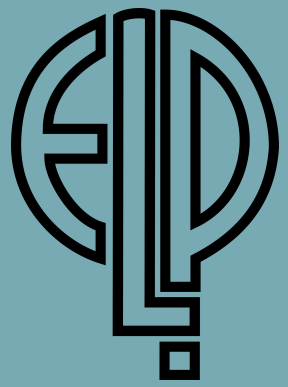
NO upper bound on the relative speed of processes

NO upper bound on message delivery time

CONSENSUS† IS IMPOSSIBLE IN AN ASYNCHRONOUS SYSTEM*

†deterministic

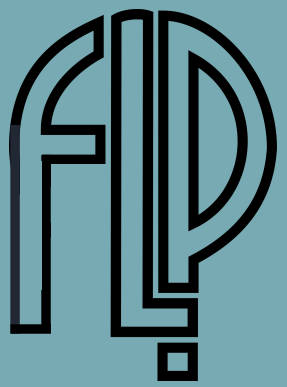
*in the presence of failures

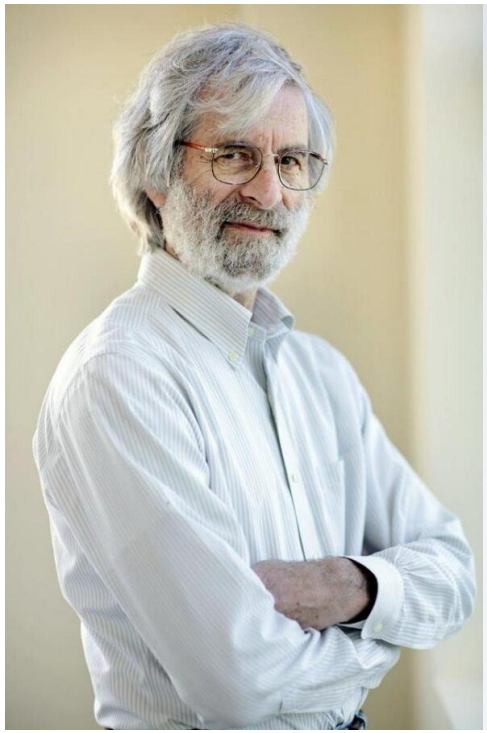


CONSENSUS† IS IMPOSSIBLE IN AN ASYNCHRONOUS SYSTEM*

†deterministic

*in the presence of failures





Paxos

Always safe

Ready to pounce on liveness

GREAT WITHIN DATACENTERS!

Google Cloud

Microsoft Azure

…where, besides, it may be ok to assume more of the network

D. Ports et al.
Designing distributed systems unisng approximate synchrony in datacenter networks
NSDI '15

TAKING STOCK

- We can define a strong notion of correctness for concurrent objects
- We can use consensus to achieve it in a distributed setting
  - Impossible? HA! Nothing is impossible!

TAKING STOCK

- We can define a strong notion of correctness for concurrent objects
- We can use consensus to achieve it in a distributed setting
  - Impossible? HA! Nothing† is impossible!

†Exceptions include:

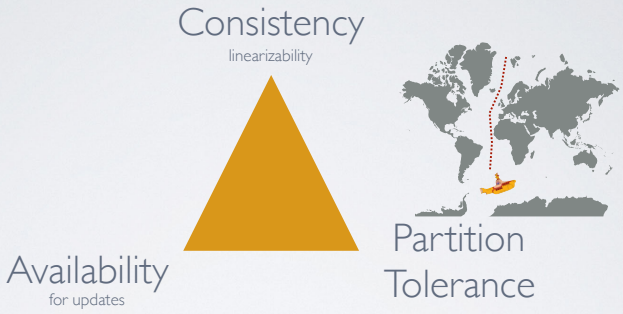
✓ cappuccino after lunch or dinner

✓ actually, asynchronous consensus

✓ England beating Germany at football

… BUT WHAT ABOUT GEO-REPLICATED SYSTEMS?

THE CAP DILEMMA



Eric Brewer's CAP Theorem

"You can have at most two of C, A, and P for any shared data system"

WHAT DOES  MEAN?

Werner Vogels, CTO Amazon

"An important observation is that in larger distributed-scale systems, network partitions are a given; therefore, **consistency and availability cannot be achieved at the same time**."

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

Farewell consistency, we hardly knew ye…

WHAT DOES C A P MEAN?

"No system where P is possible can at all times guarantee both C and A"

i.e.

if your network is highly reliable (and fast), so that P is extremely rare, you can aim for **both** C **and** A

Google Spanner

GEO-REPLICATED SYSTEMS

- Facebook, Twitter, Amazon aim for ALPS
  - Availability
  - low Latency
  - Partition tolerance
  - Scalability
- What about consistency?
  - Tension (you guessed it) between performance and ease of programming

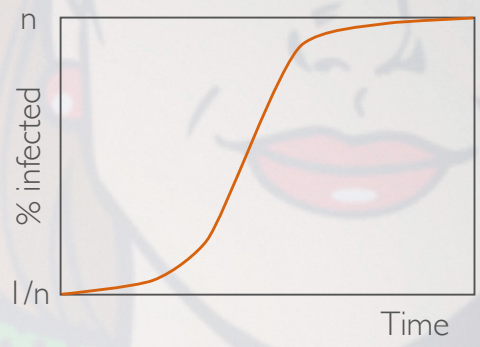
EVENTUAL CONSISTENCY

- Replicas are guaranteed to converge
  - updates performed at one replica are eventually seen by all others
  - if no more updates, replicas eventually reach the same state

If no new updates are made to an object, eventually all accesses will return its last updated value

GOSSIP

- In each round, a replica exchanges what it knows with another replica chosen uniformly at random
- Like an epidemic, it is robust and efficient
- "Infection" completes in O(log n) rounds



WHO'S USING EC?

- Domain Name Service (DNS)
- Facebook
- Amazon
- Twitter
- …
- Bayou (1995)
- Clearinghouse (1987)

WHO'S USING EC?

- Domain Name Service (DNS)
- Facebook
- Amazon
- Twitter
- …
- Bayou (1995)
- Clearinghouse (1987)

BAYOU

Terry et al, SOSP '95

- Replicas keep ordered log of state updates
- Gossip entries in their log
- If no more updates, logs (states) eventually converge
- But Bayou gives you more:

"If the log of R_i contains an update w first performed on R_j, then the log of R_i also contains all the updates accepted by R_j prior to w."

If a replica sees an update w, it has seen all updates that causally precede w!

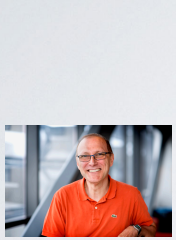
CAUSAL CONSISTENCY

Updates that are causally related should be seen by all replicas in the same order. Concurrent updates may be seen by different replicas in different orders (Hutto & Ahamad, 1990)

Two operations a and b are causally related (a → b) if

1. The same client executes first a then b
2. b reads the value written by a
3. There exists an operation a' such that a → a' and a' → b

WHY CAUSAL CONSISTENCY?



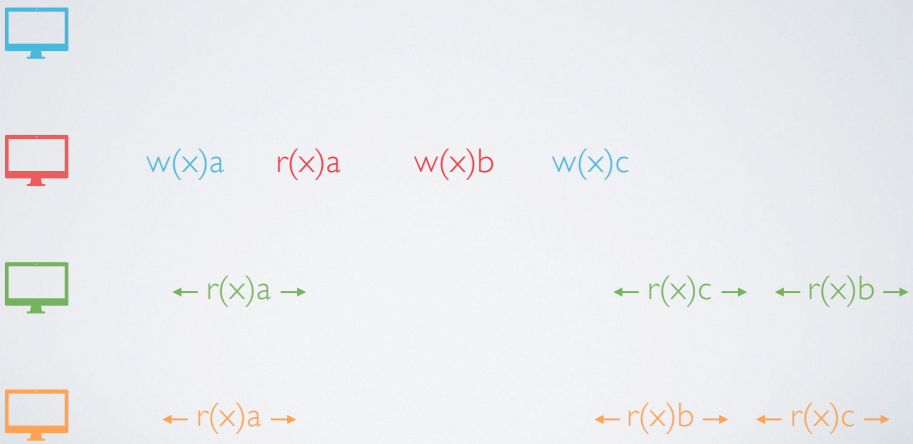
1. Receives selfie (update 2) then defriend request (update 1)
2. Whoops.

1. meditates unspeakable crime
2. defriends me (update 1)
3. posts selfie (update 2) while engaging in unspeakable crime

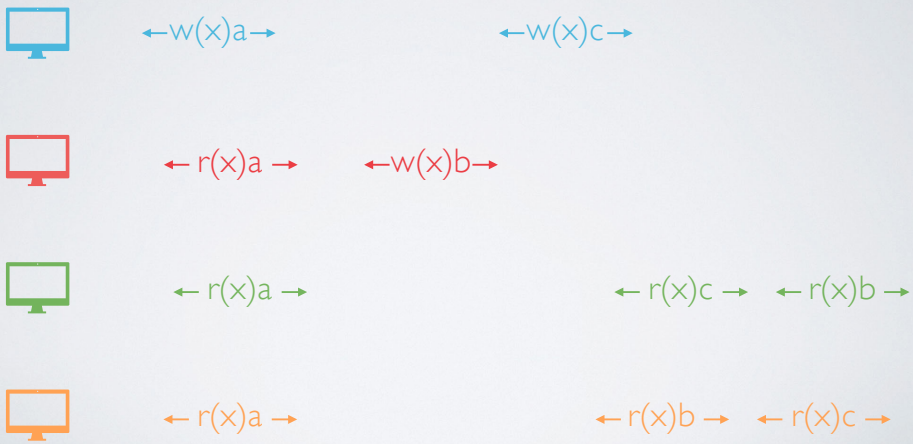
SEQUENTIALLY CONSISTENT?

CAUSALLY CONSISTENT?

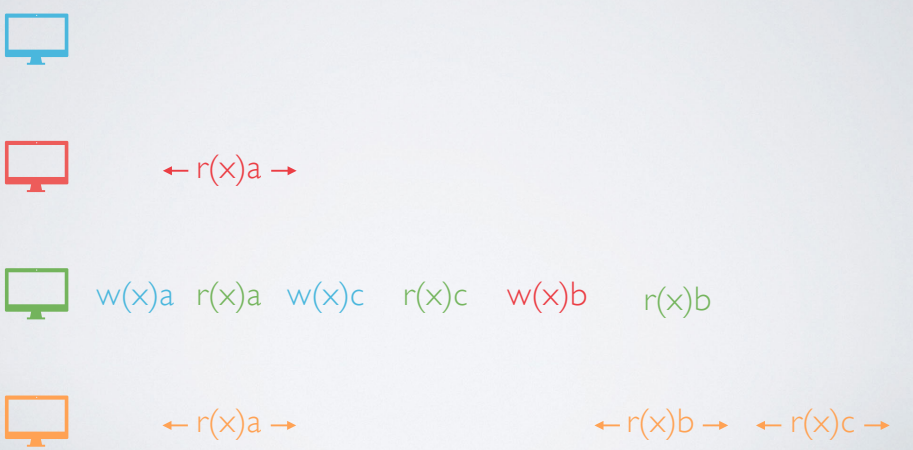
CAUSALLY CONSISTENT?



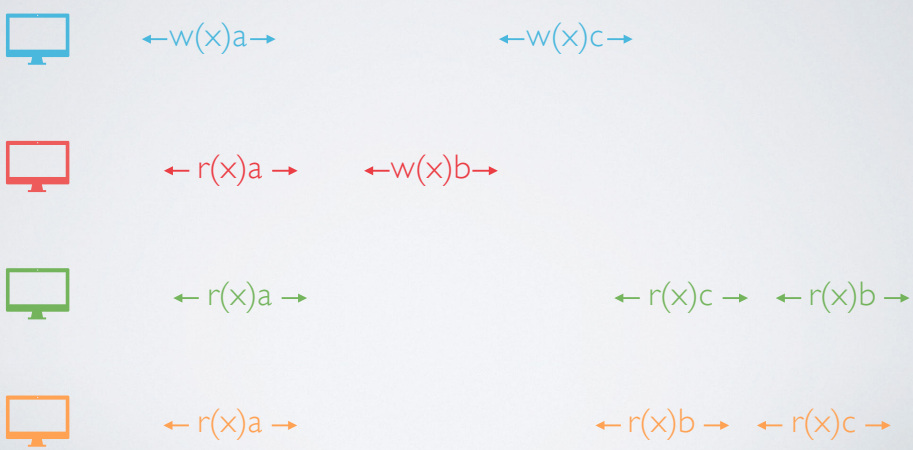
CAUSALLY CONSISTENT?



CAUSALLY CONSISTENT?



CAUSALLY CONSISTENT?



CAUSALLY CONSISTENT?

← r(x)a →

← r(x)a → ← r(x)c → ← r(x)b →

w(x)a r(x)a w(x)b r(x)b w(x)c r(x)c

CAUSALLY CONSISTENT?

←w(x)a→ ←w(x)c→

← r(x)a → ←w(x)b→

← r(x)a → ← r(x)c → ← r(x)b →

← r(x)a → ← r(x)b → ← r(x)c →

CAUSALLY CONSISTENT?

←w(x)a→ ←w(x)c→

← r(x)a →

← r(x)a → ←w(x)b→ ← r(x)c → ← r(x)b →

← r(x)a → ← r(x)b → ← r(x)c →

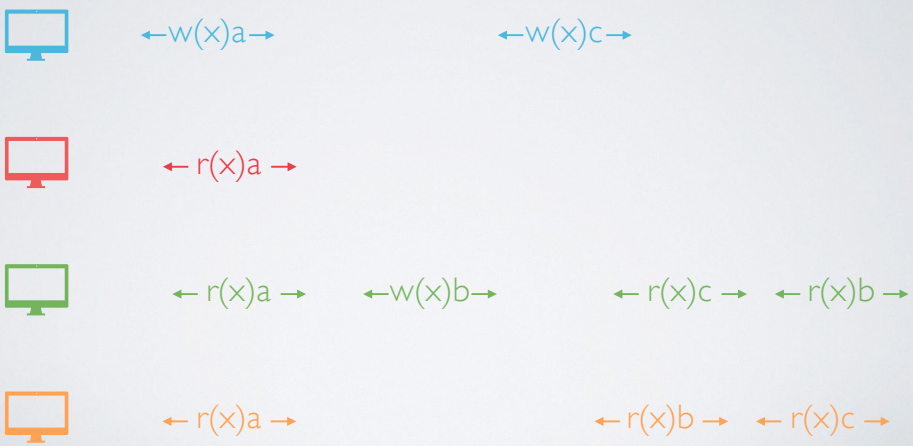
CAUSALLY CONSISTENT?

← r(x)a →

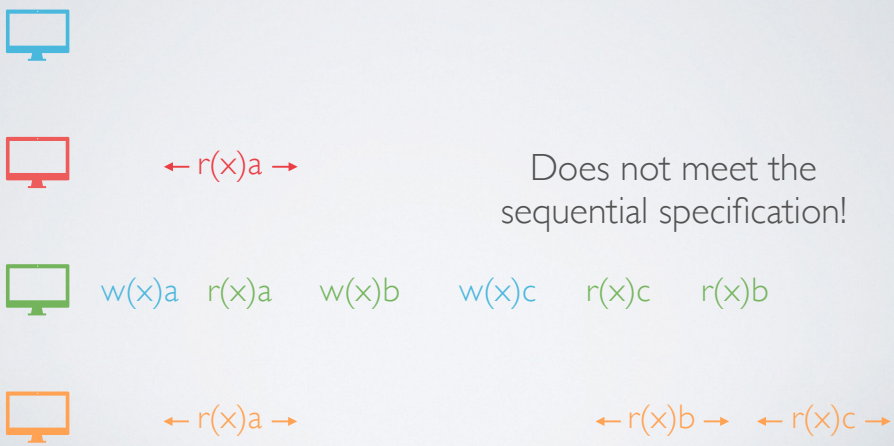
← r(x)a → ← r(x)c → ← r(x)b →

w(x)a r(x)a w(x)b r(x)b w(x)c r(x)c

CAUSALLY CONSISTENT?



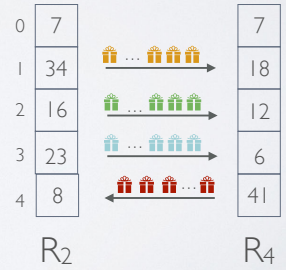
CAUSALLY CONSISTENT?



Does not meet the sequential specification!

CAUSAL CONSISTENCY IN BAYOU

- When replica R_i receives an update from a client, it assigns to it a timestamp (logical $time_i$, i)
- Each replica R_i maintains a version vector $R_i.V[\ ]$
  - $R_i.V[j]$ = highest timestamp of any write logged by R_j and known to R_i
- Replicas learn which updates they need to exchange by comparing version vectors!



BUT DOES IT SCALE?

- Log-exchange requires each replica to serve as serialization point
  - When replica is a datacenter with thousands of shards, some node must serialize across all shards

COPS: CLUSTER OF ORDER PRESERVING SERVERS

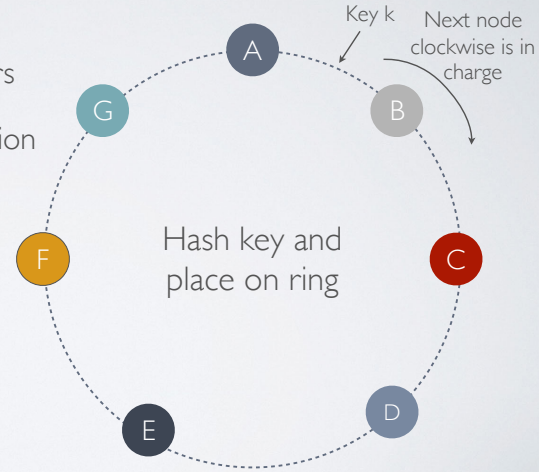
Loyd et al., SOSP'11

- Many clients, few datacenters

COPS: CLUSTER OF ORDER PRESERVING SERVERS

Loyd et al., SOSP'11

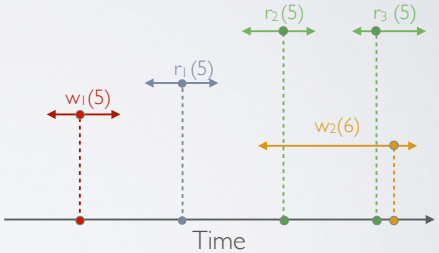
- Many clients, few datacenters
- Consistent hashing to partition the keys
  - in each shard a "primary" node responsible for key



COPS: CLUSTER OF ORDER PRESERVING SERVERS

Loyd et al., SOSP'11

- Many clients, few datacenters
- Consistent hashing to partition the keys
  - in each shard a "primary" node responsible for key
- Each datacenter is linearizable
  - low latency, "no" partitions



COPS: CLUSTER OF ORDER PRESERVING SERVERS

Loyd et al., SOSP'11

- Many clients, few datacenters
- Consistent hashing to partition the keys
  - in each partition a "primary" node responsible for key
- Each datacenter is linearizable
  - low latency, "no" partitions
- Get/Put operations execute at a local datacenter, and then asynchronously replicated

TOWARDS SCALABLE CAUSAL CONSISTENCY

Distributed verification

~~Serialization~~

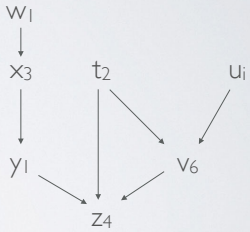
- Each client keeps a context 
  - In principle, it includes all values previously read or written in client's session and what they depend on
- On get, returned key version and its causal dependencies are are added to context

TOWARDS SCALABLE CAUSAL CONSISTENCY

Distributed verification

~~Serialization~~

- Each client keeps a context 
  - In principle, it includes all values previously read or written in client's session and what they depend on
- On get, returned key version and its causal dependencies are are added to context



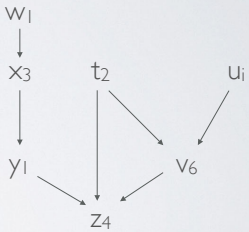
TOWARDS SCALABLE CAUSAL CONSISTENCY

Distributed verification

~~Serialization~~

- Each client keeps a context 
  - In principle, it includes all values previously read or written in client's session and what they depend on
- On get, returned key version and its causal dependencies are are added to context

- On a put, client includes (and replicates) its "nearest dependencies" from context…

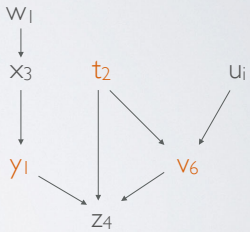


TOWARDS SCALABLE CAUSAL CONSISTENCY

Distributed verification

~~Serialization~~

- Each client keeps a context 
  - In principle, it includes all values previously read or written in client's session and what they depend on
- On get, returned key version and its causal dependencies are are added to context
- On a put, client includes (and replicates) its "nearest dependencies" from context…



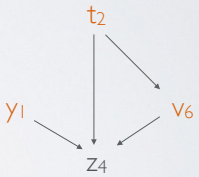
TOWARDS SCALABLE CAUSAL CONSISTENCY

Distributed verification

~~Serialization~~

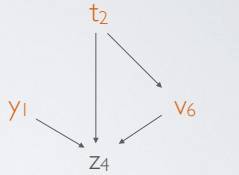
- Each client keeps a context 

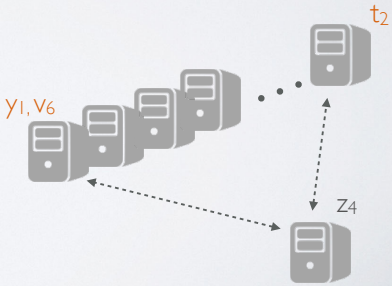
  - In principle, it includes all values previously read or written in client's session and what they depend on
- On get, returned key version and its causal dependencies are are added to context
- On a put, client includes (and replicates) its "nearest dependencies" from context… and resets context to the latest put



TOWARDS SCALABLE CAUSAL CONSISTENCY

- Remote datacenter, before applying z_4, verifies nearest dependencies have already been applied



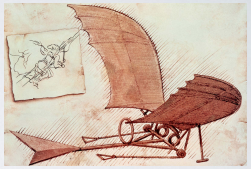


ACADEMIC SYSTEMS EXPLORING CAUSAL CONSISTENCY

- COPS (SOSP '11)
- Bolt-On (SIGMOD '13)
- Chain Reaction (Eurosys '13)
- Eiger (NSDI '13)
- Orbe (SOCC '13)
- GentleRain (SOCC '14)
- Cure (ICDCS 16)
- Tardis (SIGMOD '16)
- Saturn (Eurosys '17)

A FAIRY TALE ENDING…

Linearizability



Eventual consistency

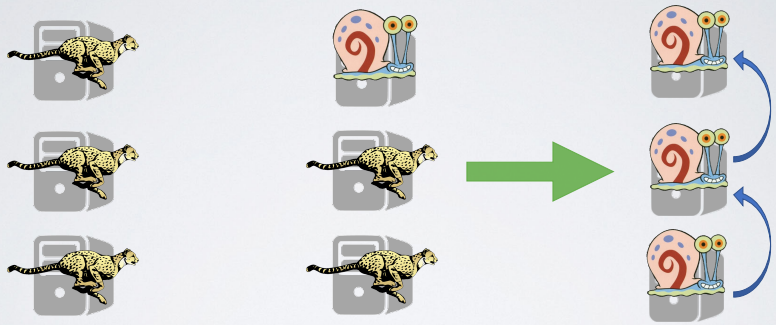
Causal Consistency

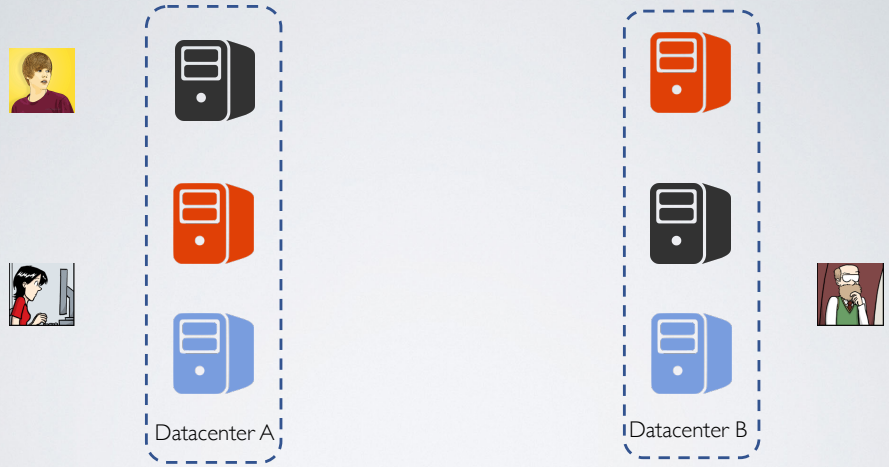
INDUSTRIAL SYSTEMS USING CAUSAL CONSISTENCY

INDUSTRIAL SYSTEMS USING CAUSAL CONSISTENCY

No, seriously now…

SLOWDOWN CASCADES

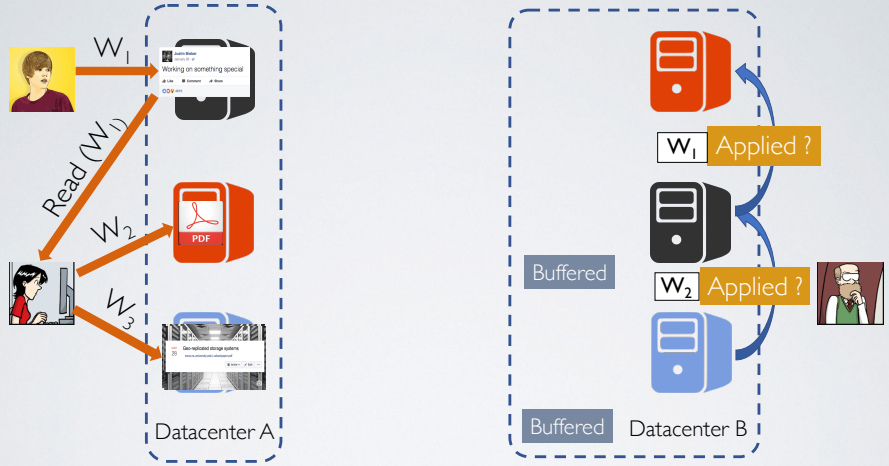




Replicated and sharded storage for a social network

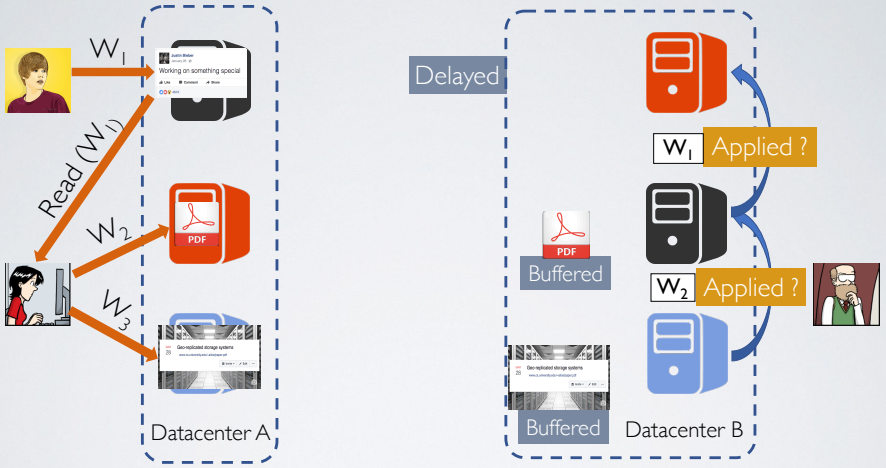
Writes are causally ordered

$W_1 \rightarrow W_2 \rightarrow W_3$



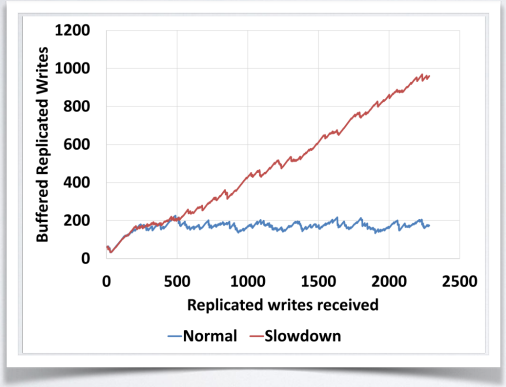
Current causal systems enforce causal consistency as an invariant

Slowdown Cascade



Alice's advisor unnecessarily waits for Justin Bieber's update despite not reading it

SLOWDOWN CASCADES IN EIGER (NSDI '13)



Buffers for replicated writes grow out of control

OCCULT Mehdi et al, NSDI '17

Observable Causal Consistency Using Lossy Timestamps

OCCULT

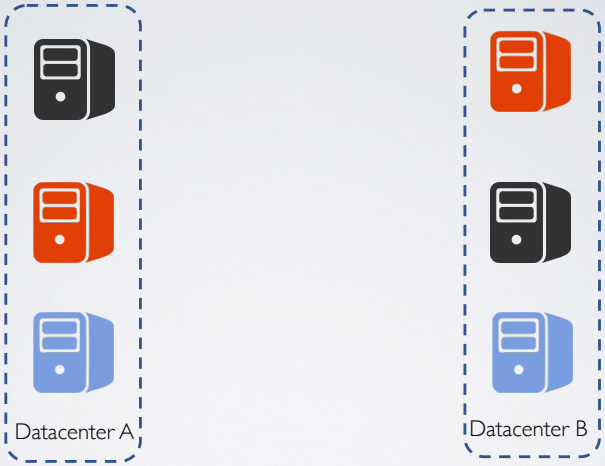
Observable Causal Consistency

OBSERVABLE CAUSAL CONSISTENCY

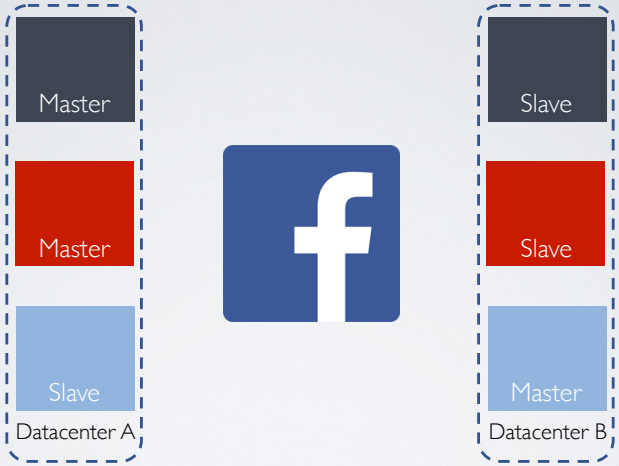
- Causal Consistency

  each client observes a monotonically non decreasing set of updates (including its own) in an order that respects potential causality between operations

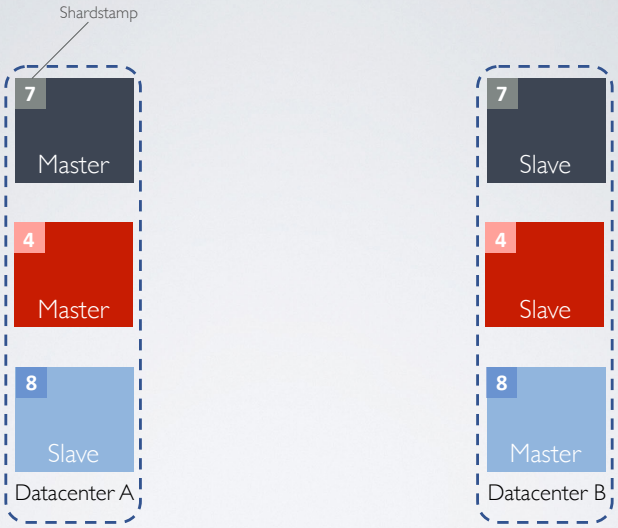
Instead of a causally consistent data store, implement a data store that appears to clients indistinguishable from one



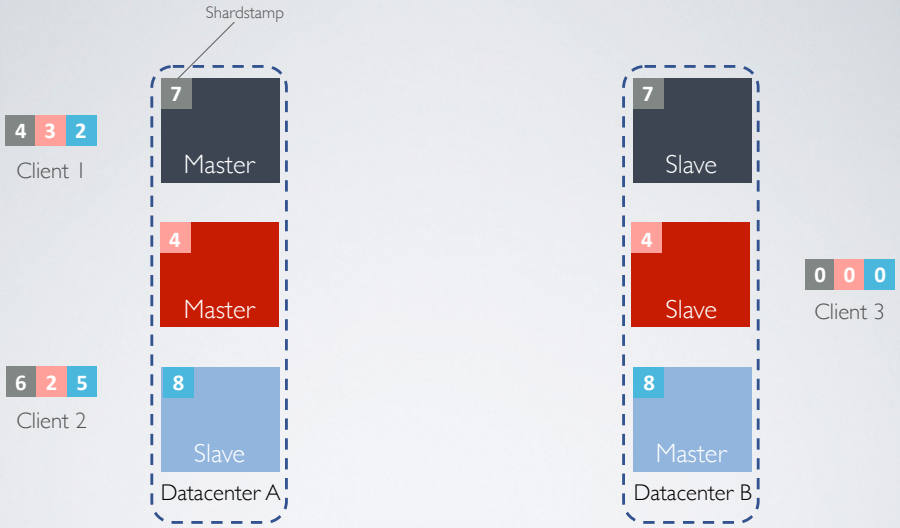
How can I guarantee clients observe
a causally consistent datastore ?



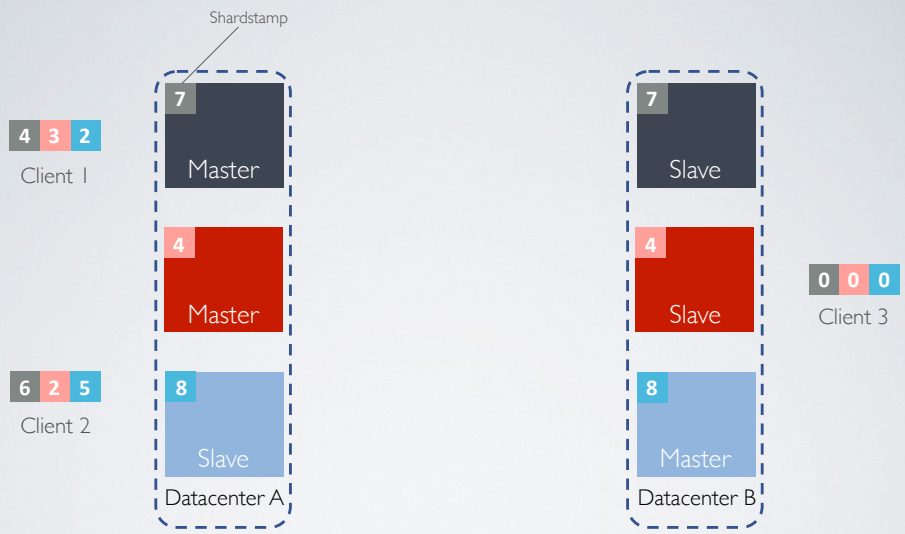
Writes accepted only by master shards and then
replicated asynchronously and in-order to slaves



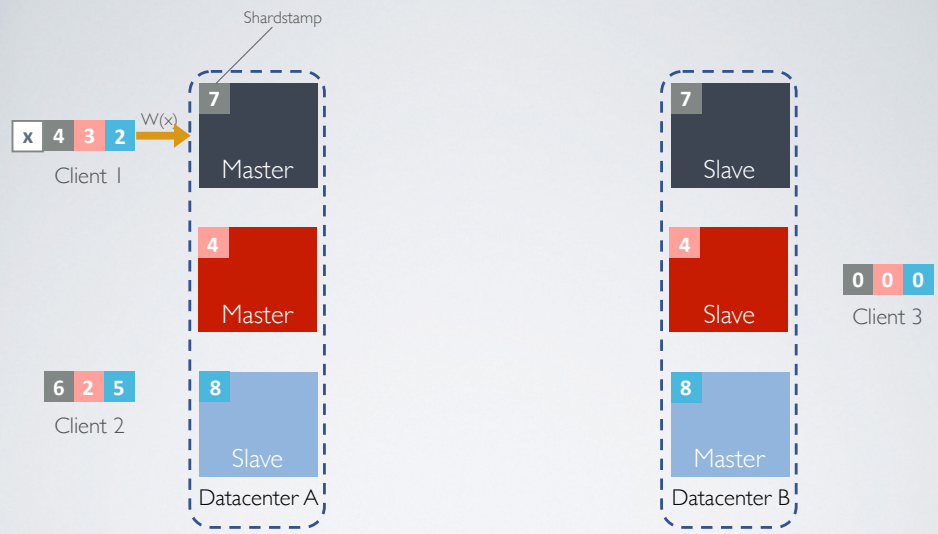
Each shard keeps track of a shardstamp
which counts the writes it has applied



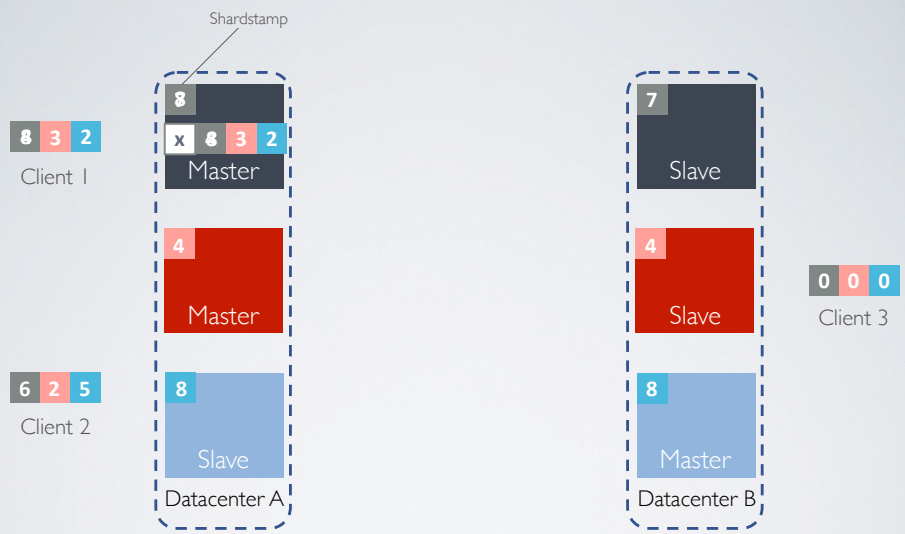
Causal timestamp: vector of shardstamps
identifying the state client knows about



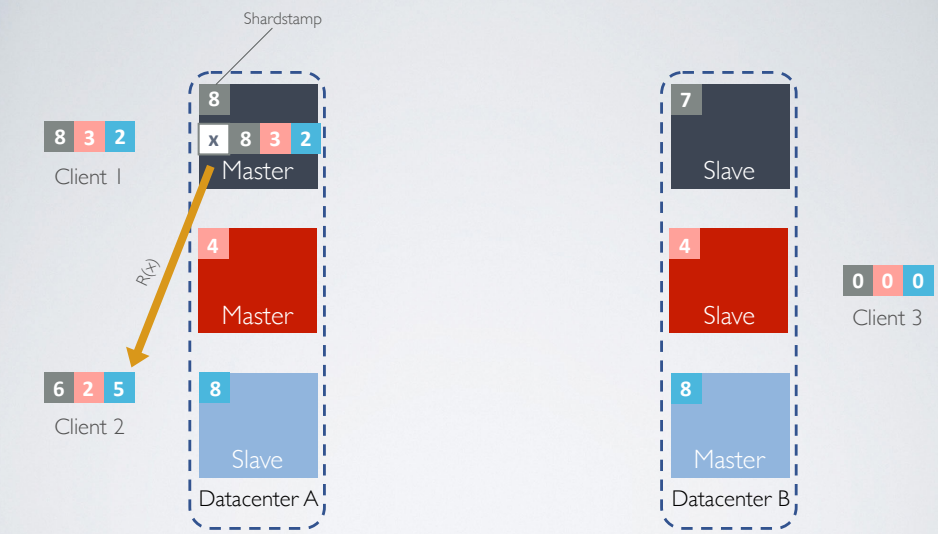
Write protocol: causal timestamps stored with objects to propagate dependencies



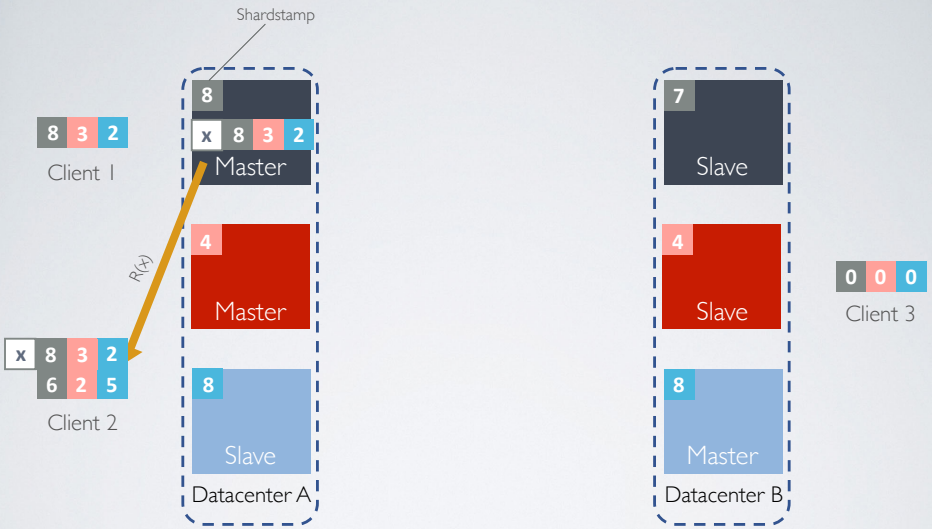
Write protocol: causal timestamps stored with objects to propagate dependencies



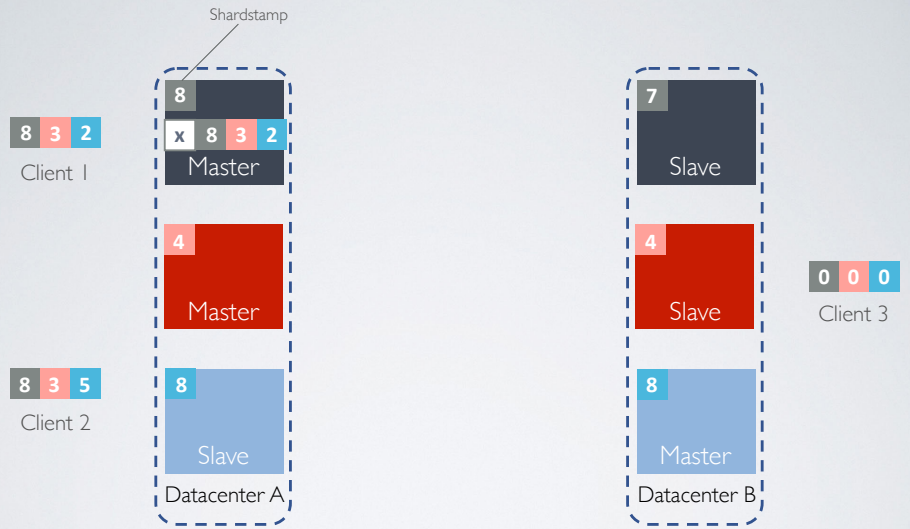
Write protocol: causal timestamps stored with objects to propagate dependencies



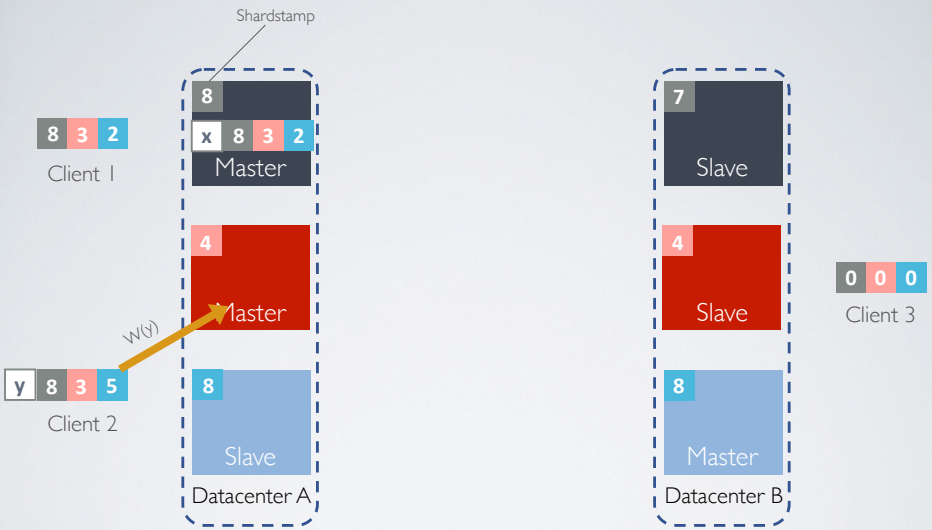
Read protocol: always safe to read from the Master



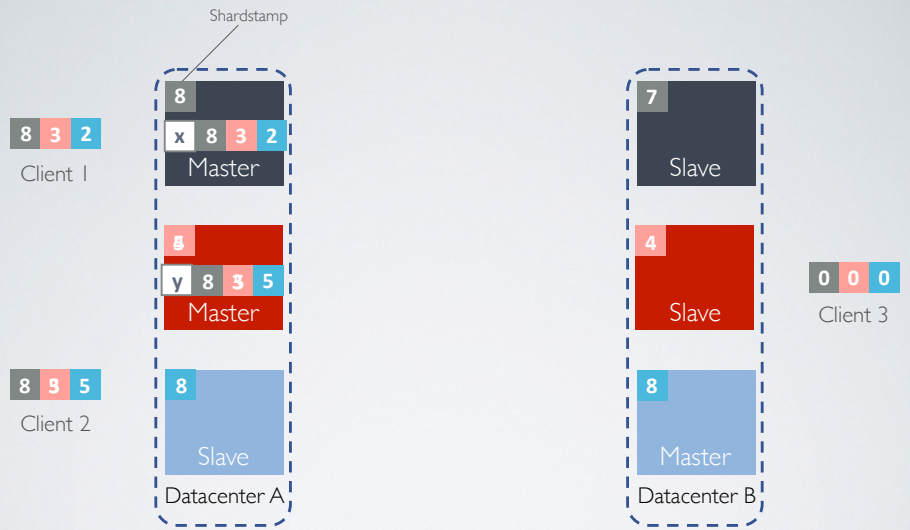
Read protocol: object's causal timestamp merged into client's causal timestamp



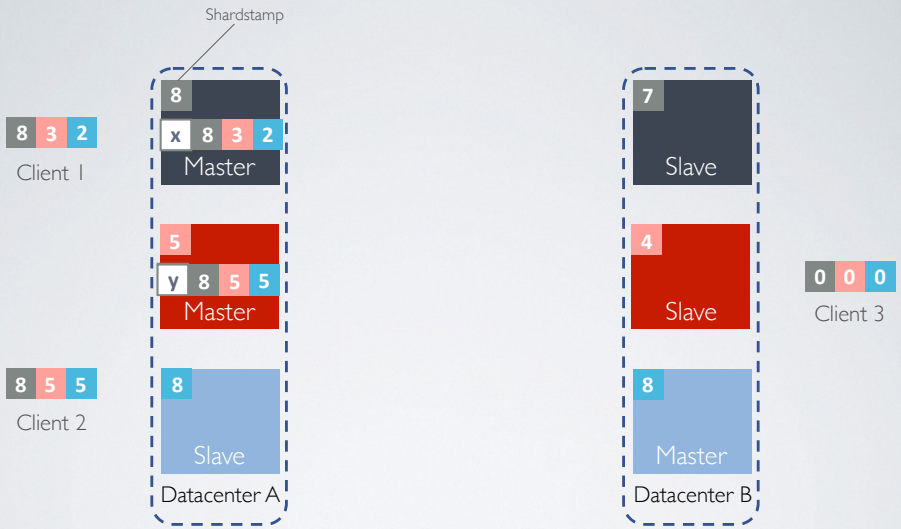
Later writes reflect causal timestamp of read object



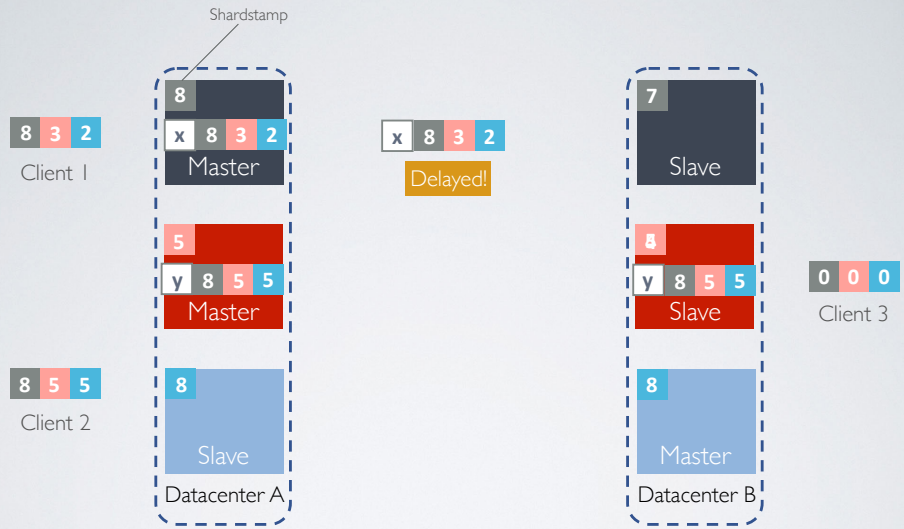
Later writes reflect causal timestamp of read object



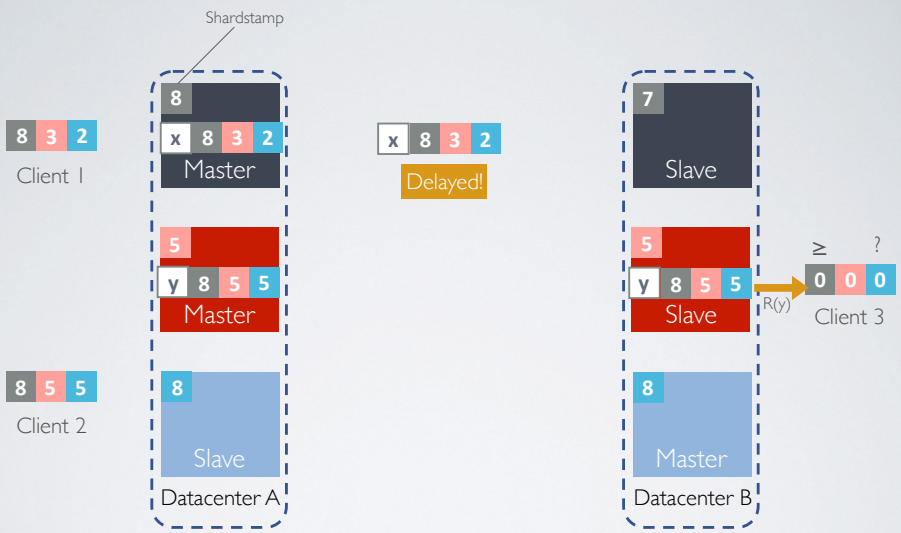
Later writes reflect causal timestamp of read object



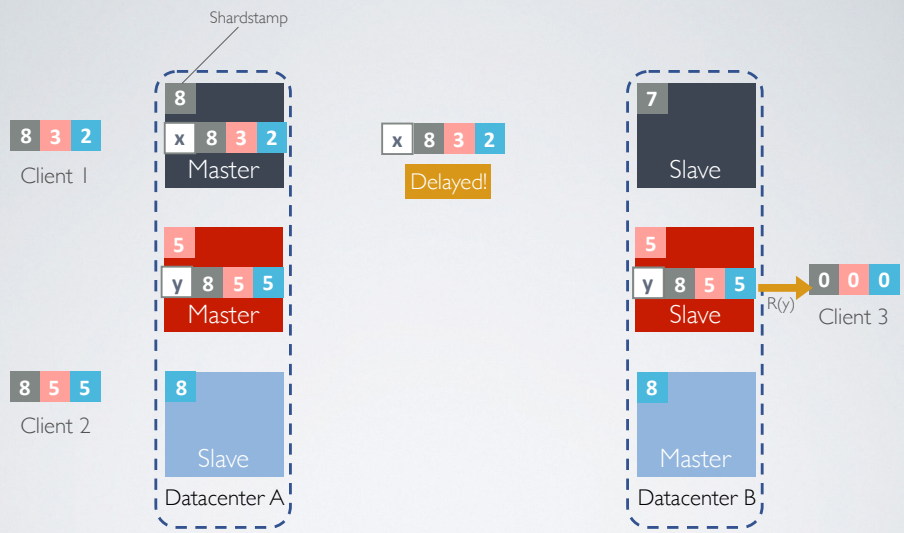
Replication: As in eventual consistency, asynchronous, unordered writes are applied immediately



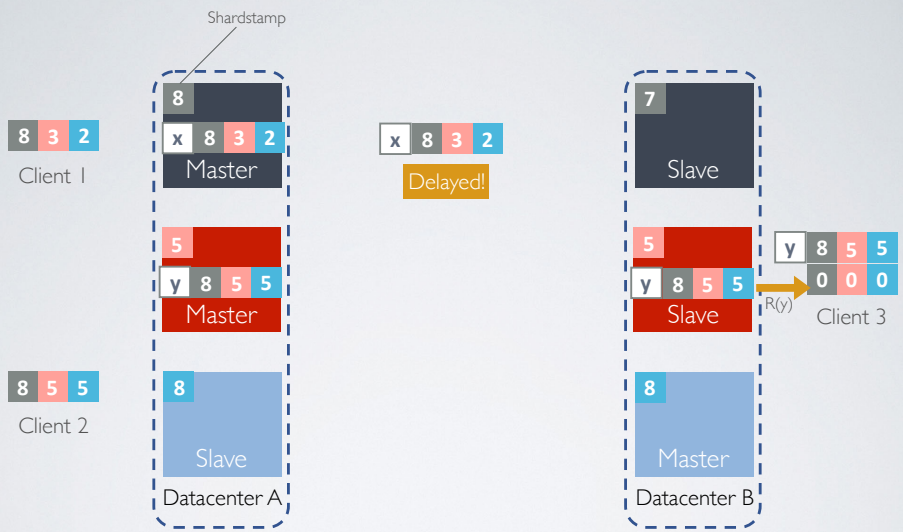
Replication: Slave increment its shardstamp using causal timestamp of replicated write



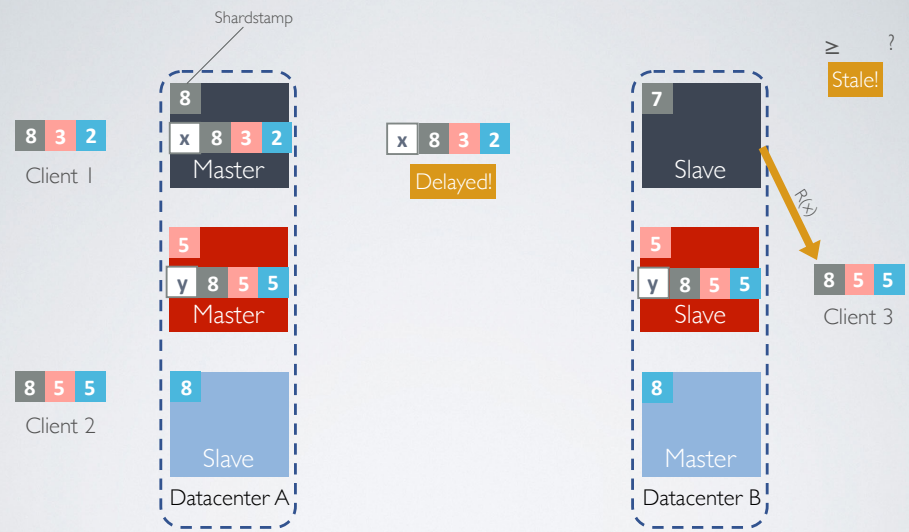
Read protocol: Clients run consistency checks when reading from slaves



Read protocol: Clients run consistency checks when reading from slaves



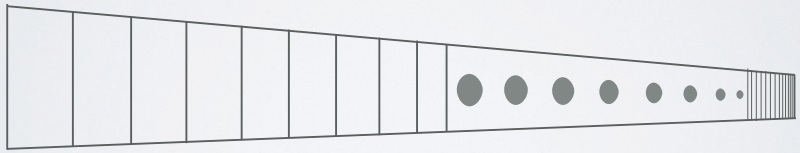
Read protocol: Though x is delayed, we can read y anyway!



Read protocol: Though x is delayed, we can read y anyway!

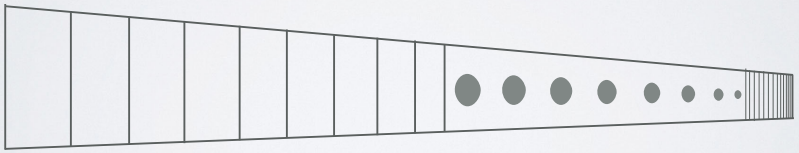
A TIMESTAMP! MY KINGDOM FOR A TIMESTAMP!

- What happens to causal timestamps at scale?
  - datacenters have tens of thousands of shards…



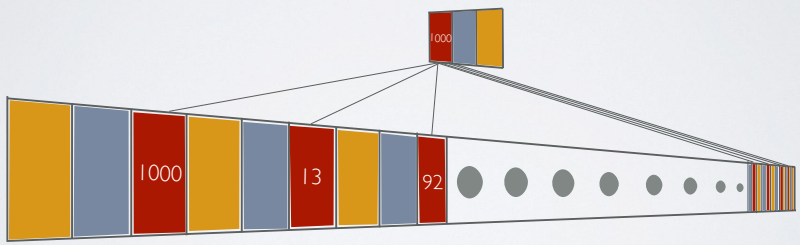
COMPRESSING TIMESTAMPS

- Conflate shardstamps with the same index mod N



COMPRESSING TIMESTAMPS: STRUCTURAL COMPRESSION

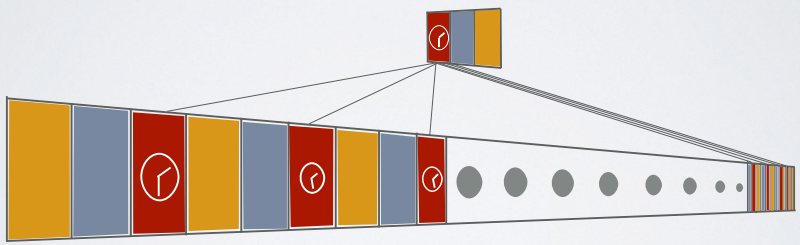
- Conflate shardstamps with the same index mod N



False dependencies

COMPRESSING TIMESTAMPS: STRUCTURAL COMPRESSION

- Use loosely synchronized rather than logical clocks



Fewer false dependencies: staleness bound by clocks' offset, independent of write rate

COMPRESSING TIMESTAMPS: TEMPORAL COMPRESSION

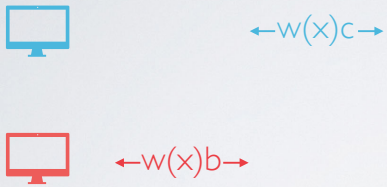
- False dependencies arise when recent and old timestamps are conflated
  - ✓ Use high resolution to track recent updates
  - ✓ Conflate the rest!

					Catchall
Shardstamps	4000	3989	3880	3873	3642
Shard Ids	1789	44	1815	1571	*

4	Shardstamps
16K	Shards
0.01%	False dependencies

CONVERGENCE

Mahajan et al.; Lloyd et al. (2011)



Causal consistency does not guarantee eventual consistency!

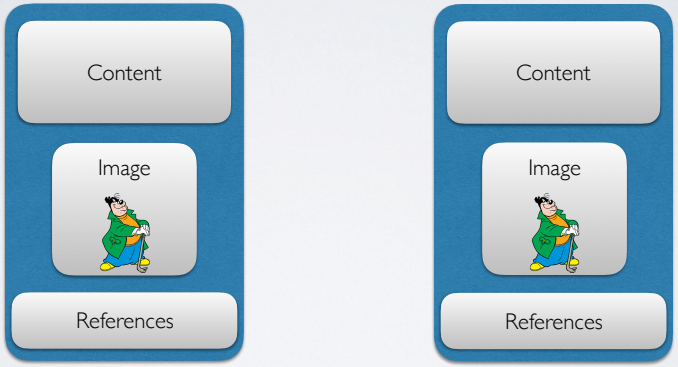
- Merging is either blunt (last writer wins) or hard:
  - requires knowledge of application semantics

LIVING THE DREAM

- Not about preventing anomalies
- About how to provide system support for efficiently resolving anomalies

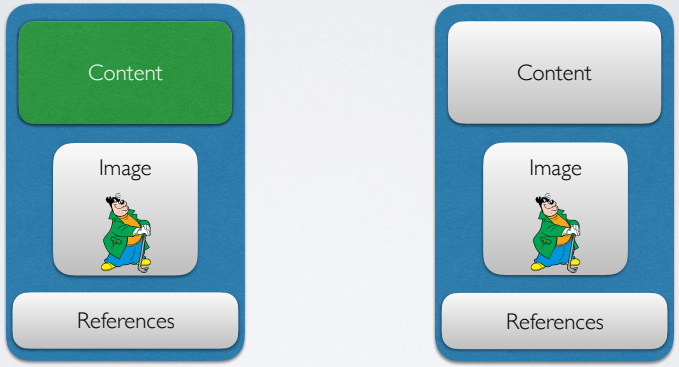
MEET MR. PRUNT

MR. PRUNT'S WIKIPEDIA



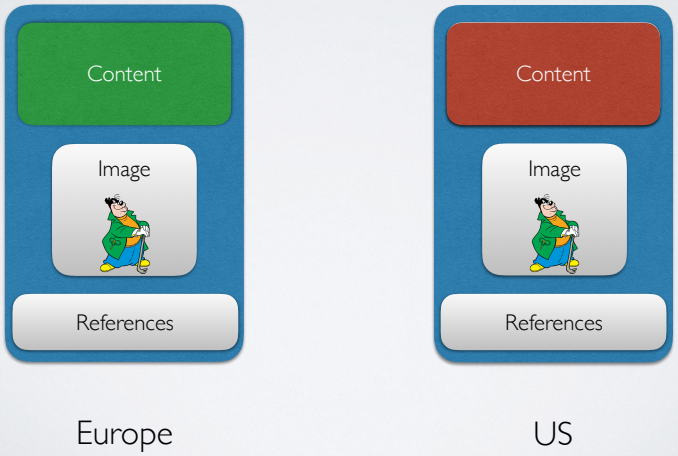
Europe US

ALICE UPDATES CONTENT

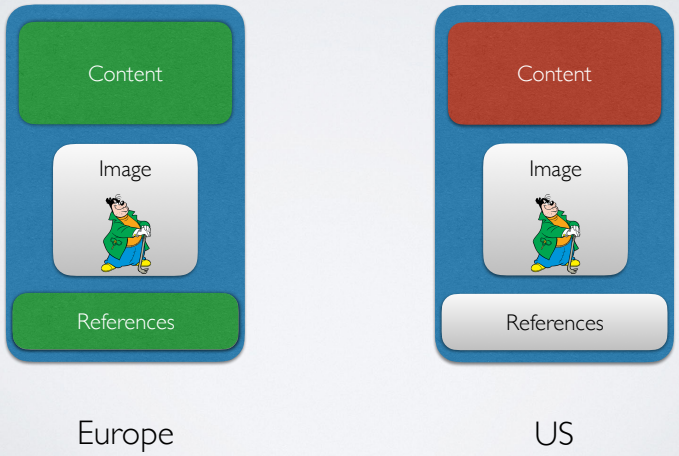


Europe US

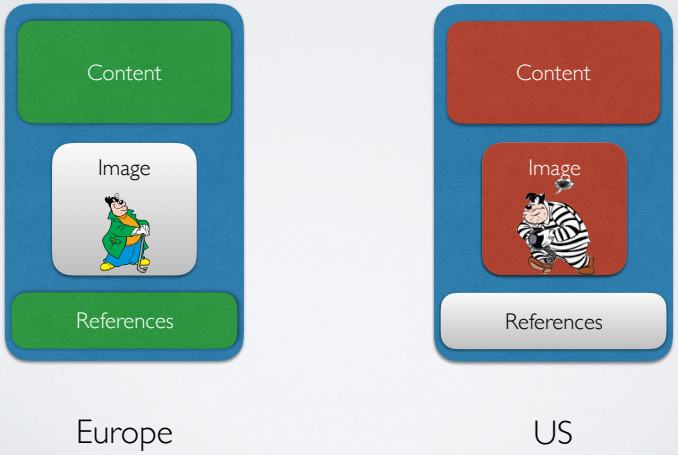
BOB TOO, CONCURRENTLY, UPDATES CONTENT



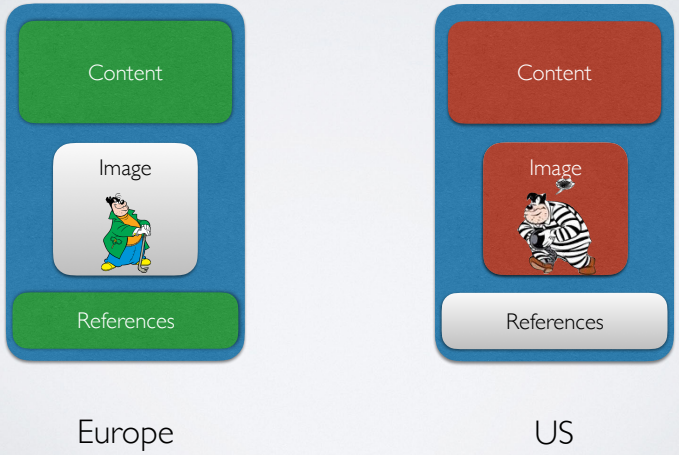
CHARLIE READS ALICE AND UPDATES REFERENCES



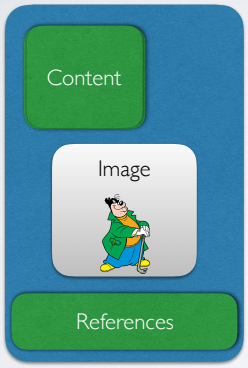
DAVE READS BOB AND UPDATES IMAGE



INCONSISTENT FINAL STATE



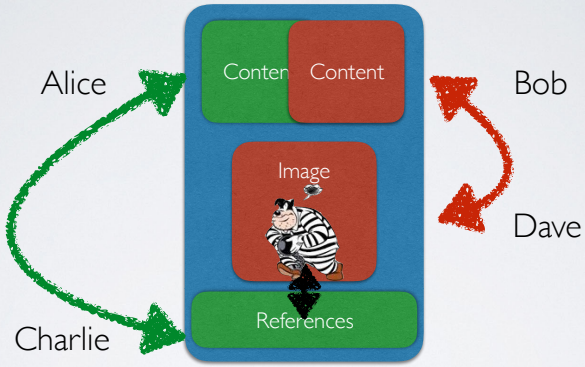
INCONSISTENT FINAL STATE



Europe

US

WHY IS MERGING HARD?

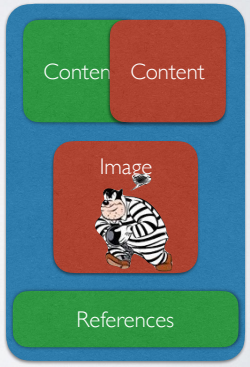


- Conflicts hinge on semantics
- Conflicts are indirect

WHAT WOULD PRUNT SAY?

@realdolandprunt

- Syntactic conflict resolution is sad
  - can't handle semantic conflicts
  - creates the Potemkin abstraction™ of a sequential view
- Lack of cross-object semantics is pathetic
  - a single write-write conflict can affect the entire system state



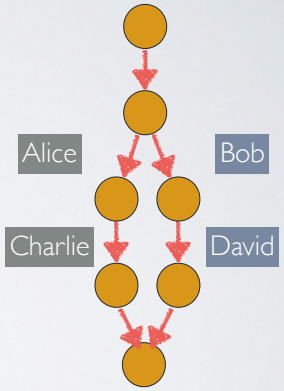
TARDIS

Crooks et al, SIGMOD '16

If you can't hide conflicts from applications, make them truly visible!

THE WORLD ACCORDING TO GIT?

- Branch-on-conflict
  - conflicts create distinct branches
- Branch Isolation
  - branches track linear evolution
- Atomically merge branches (not objects!) when desired
  - expose fork/merge points



PLEASE VISIT OUR LOCAL BRANCH

- TARDiS branches-on-conflict locally for performance

PLEASE VISIT OUR LOCAL BRANCH

- TARDiS branches-on-conflict locally for performance

- No increase in complexity as
  - abstraction of sequential store not preserved end-to-end
  - applications already built to handle merges