Data-Intensive Distributed Computing

CS 451/651 (Fall 2018)

Part 7: Mutable State (2/2)

November 13, 2018

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at http://lintool.github.io/bigdata-2018f/

The Fundamental Problem

We want to keep track of *mutable* state in a *scalable* manner

Assumptions:

State organized in terms of logical records

State unlikely to fit on single machine, must be distributed

MapReduce won't do!

Motivating Scenarios

Money shouldn't be created or destroyed:

Alice transfers \$100 to Bob and \$50 to Carol

The total amount of money after the transfer should be the same

Phantom shopping cart:

Bob removes an item from his shopping cart…

Item still remains in the shopping cart

Bob refreshes the page a couple of times… item finally gone

Motivating Scenarios

People you don't want seeing your pictures:

Alice removes mom from list of people who can view photos
Alice posts embarrassing pictures from Spring Break
Can mom see Alice's photo?

Why am I still getting messages?

Bob unsubscribes from mailing list and receives confirmation
Message sent to mailing list right after unsubscribe
Does Bob receive the message?

Three Core Ideas

Why do these scenarios happen?

Partitioning (sharding)

To increase scalability and to decrease latency

Need distributed transactions!

Replication

To increase robustness (availability) and to increase throughput

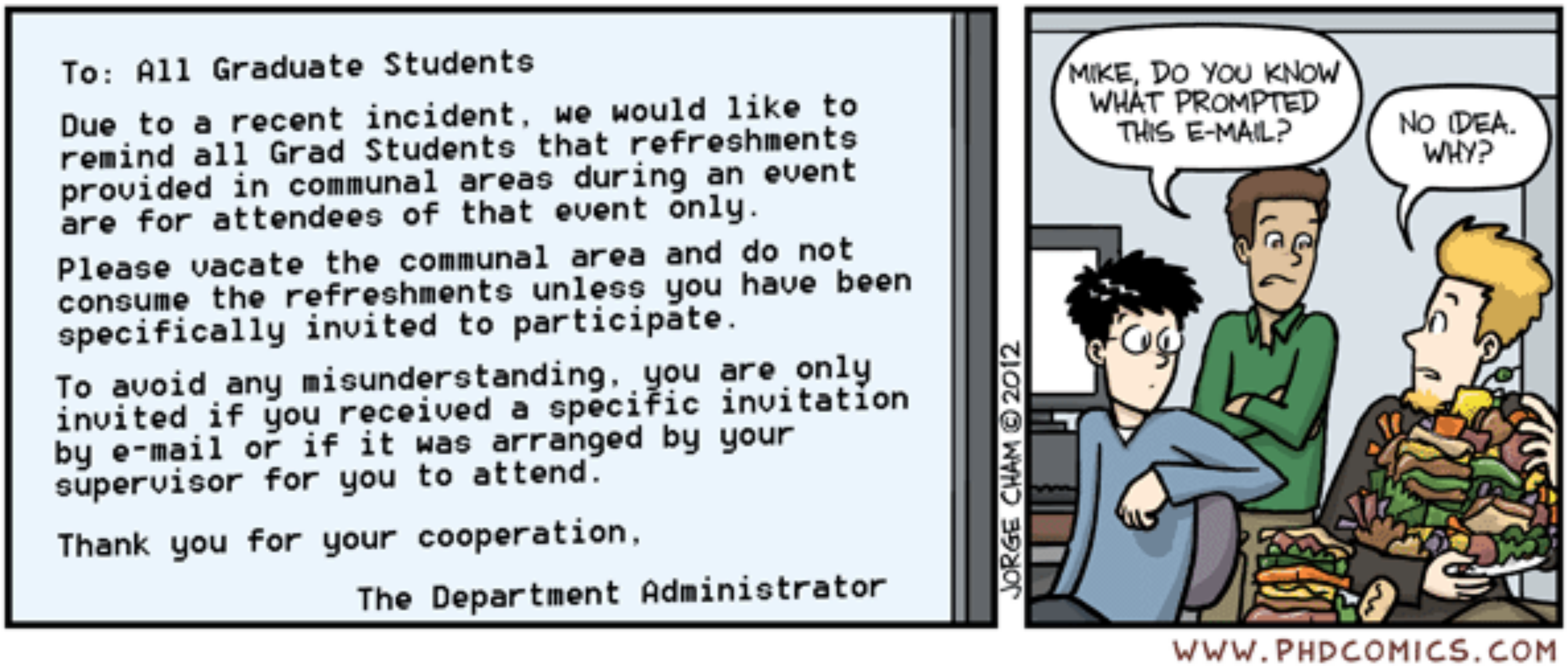
Need replica coherence protocol!

Caching

To reduce latency

Need cache coherence protocol!

Source: Wikipedia (Cake)



Morale of the story: there's no free lunch!

(Everything is a tradeoff)

Source: www.phdcomics.com/comics/archive.php?comicid=1475

Three Core Ideas

Why do these scenarios happen?

Partitioning (sharding)

To increase scalability and to decrease latency

Need distributed transactions!

Replication

To increase robustness (availability) and to increase throughput

Need replica coherence protocol!

Caching

To reduce latency

Need cache coherence protocol!

Source: images.wikia.com/batman/images/b/b1/Bat_Signal.jpg

How do RDBMSes do it?

Transactions on a single machine: (relatively) easy!

Partition tables to keep transactions on a single machine

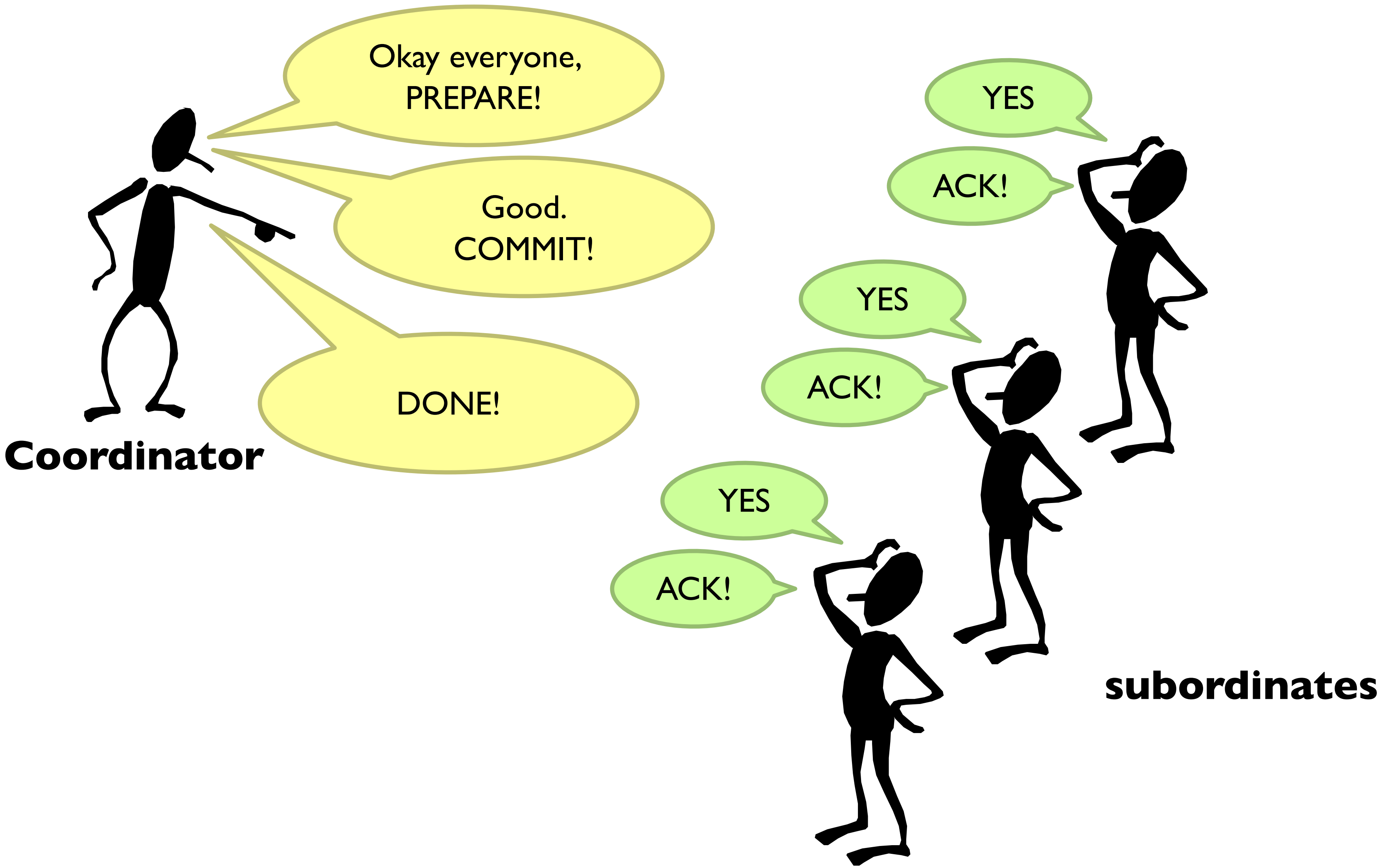
Example: partition by user

What about transactions that require multiple machines?

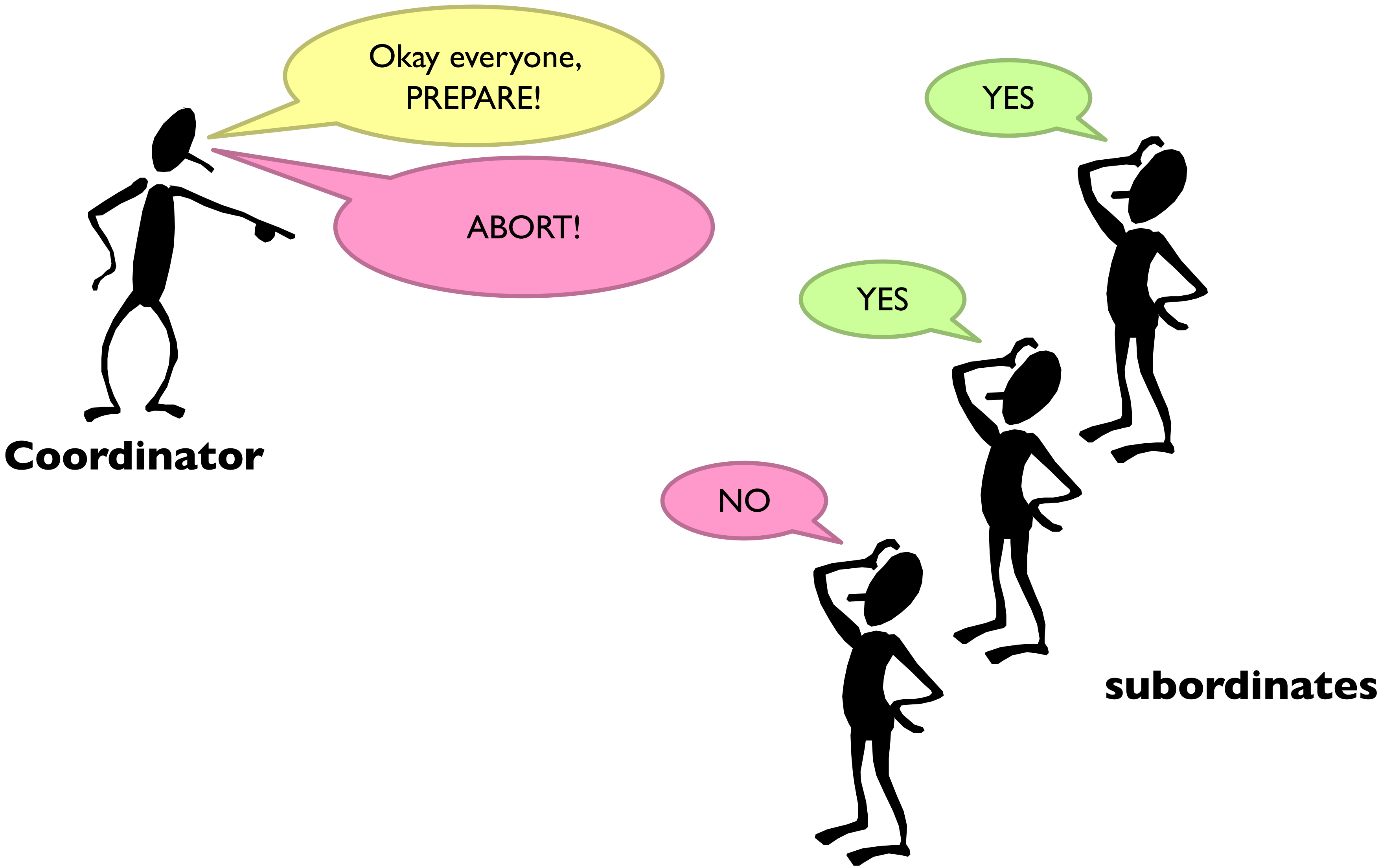
Example: transactions involving multiple users

Solution: Two-Phase Commit

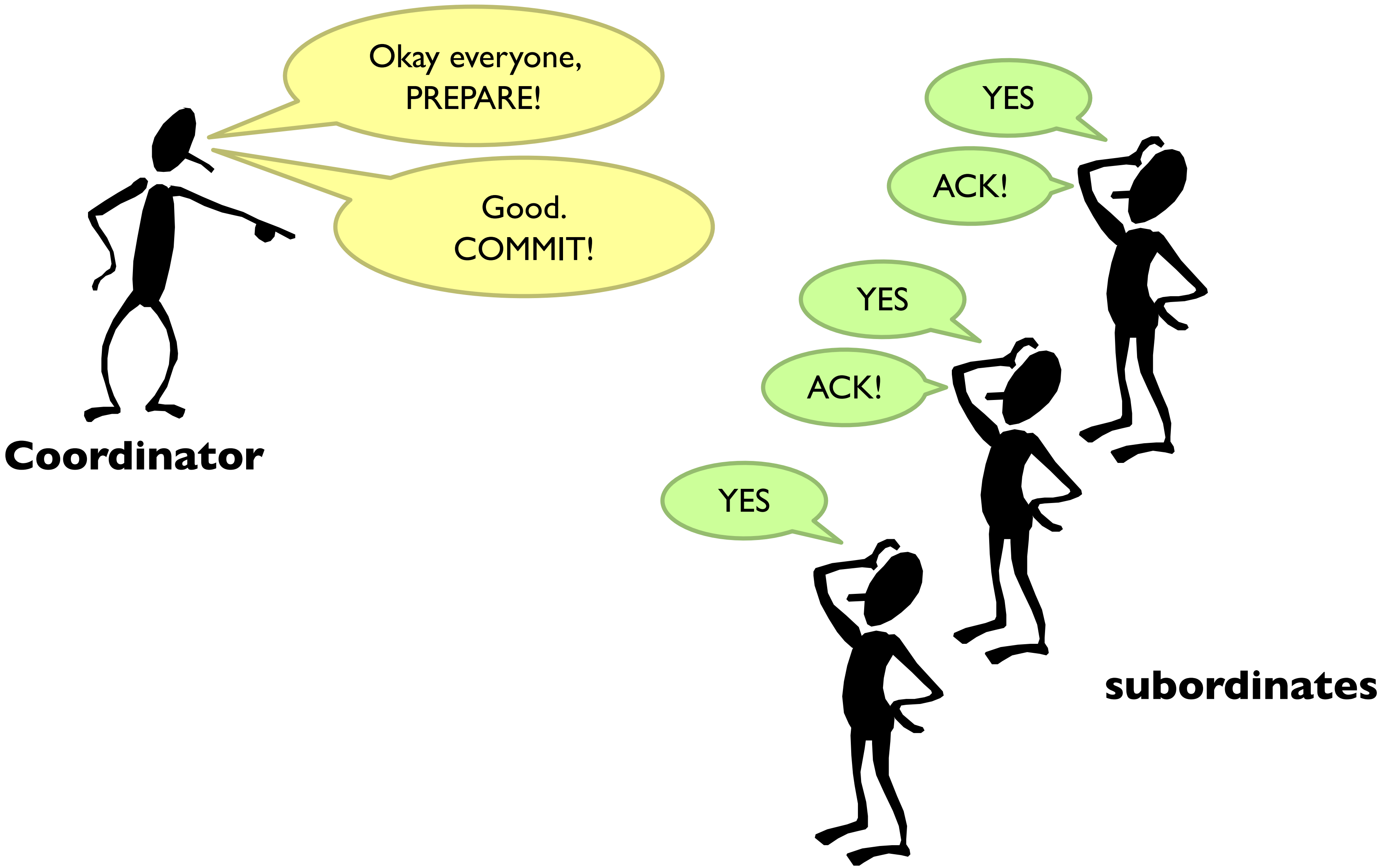
2PC: Sketch


Okay everyone,
PREPARE!
Good.
COMMIT!
DONE!
Coordinator
YES
ACK!
YES
ACK!
YES
ACK!
subordinates

2PC: Sketch


Okay everyone,
PREPARE!
ABORT!
YES
YES
NO
Coordinator
subordinates

2PC: Sketch


Okay everyone,
PREPARE!
Good.
COMMIT!
YES
ACK!
YES
ACK!
YES
Coordinator
subordinates

2PC: Assumptions and Limitations

Assumptions:

Persistent storage and write-ahead log at every node

WAL is never permanently lost

Limitations:

It's blocking and slow

What if the coordinator dies?

Three Core Ideas

Why do these scenarios happen?

Partitioning (sharding)

To increase scalability and to decrease latency

Need distributed transactions!

Replication

To increase robustness (availability) and to increase throughput

Need replica coherence protocol!

Caching

To reduce latency

Need cache coherence protocol!

Replication possibilities

Update sent to a master

Okay, but if the master fails?

Replication is synchronous

Replication is asynchronous

Combination of both

Update sent to an arbitrary replica

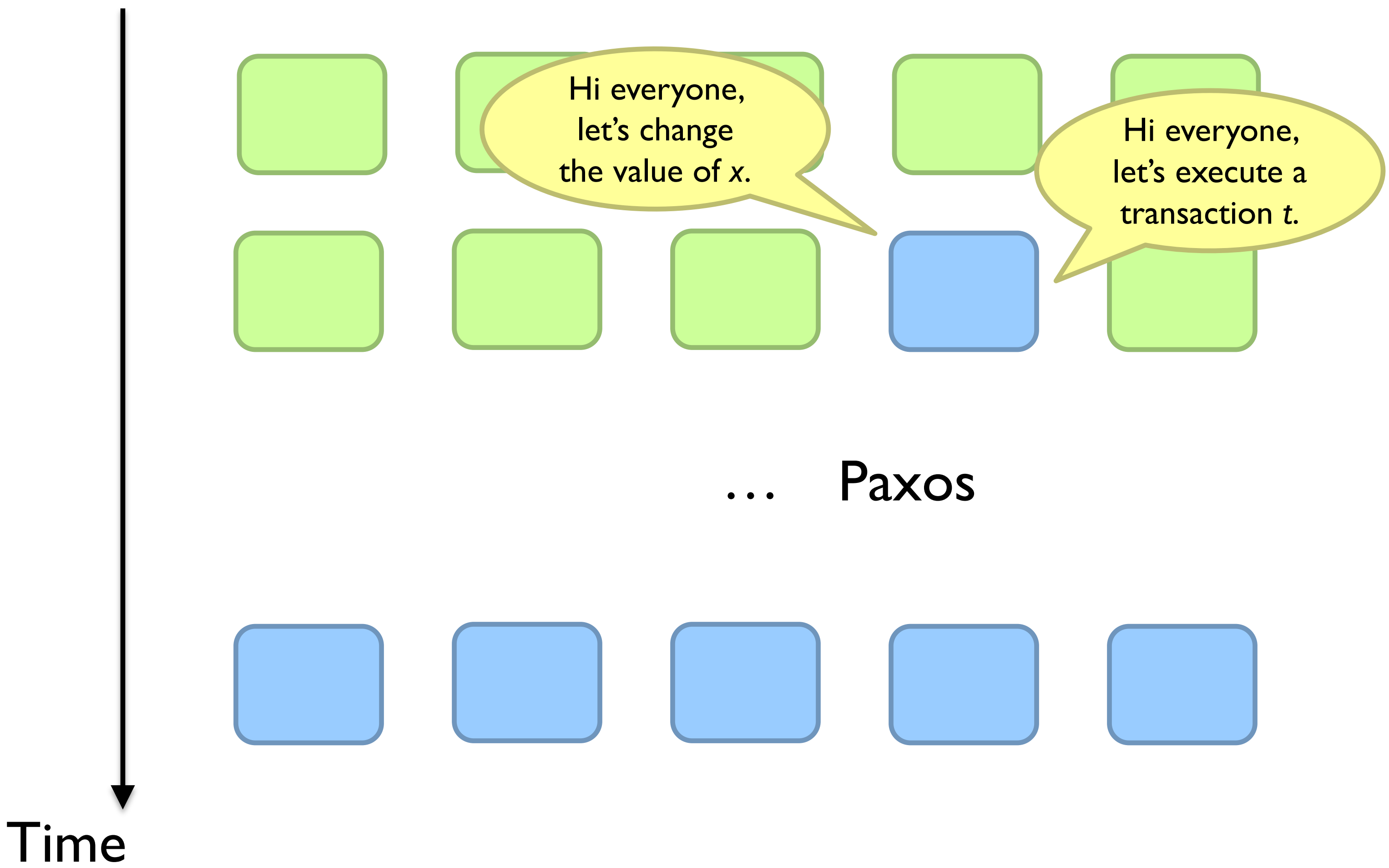
Replication is synchronous(?)

Replication is asynchronous

Combination of both

Distributed Consensus

More general problem: addresses replication and partitioning



Replication possibilities

Okay, but if the master fails?

Update sent to a master

Replication is synchronous

Replication is asynchronous

Combination of both

Update sent to an arbitrary replica

Replication is synchronous(?)

Replication is asynchronous

Combination of both

Guaranteed consistency with a consensus protocol

A buggy mess

"Eventual Consistency"

CAP “Theorem”
(Brewer, 2000)

Consistency

Availability

Partition tolerance

… pick two

CAP Tradeoffs

CA = consistency + availability

E.g., parallel databases that use 2PC

AP = availability + tolerance to partitions

E.g., DNS, web caching

Is this helpful?

CAP not really even a "theorem" because vague definitions

More precise formulation came a few years later

Source: Abadi (2012) Consistency Tradeoffs in Modern Distributed Database System Design. IEEE Computer, 45(2):37-42

Abadi Says…

CP makes no sense!

CAP says, in the presence of P, choose A or C

But you'd want to make this tradeoff even when there is no P

Fundamental tradeoff is between consistency and latency

Not available = (very) long latency

Move over, CAP

PACELC ("pass-elk")

PAC

If there's a partition, do we choose A or C?

ELC

Otherwise, do we choose **L**atency or **C**onsistency?

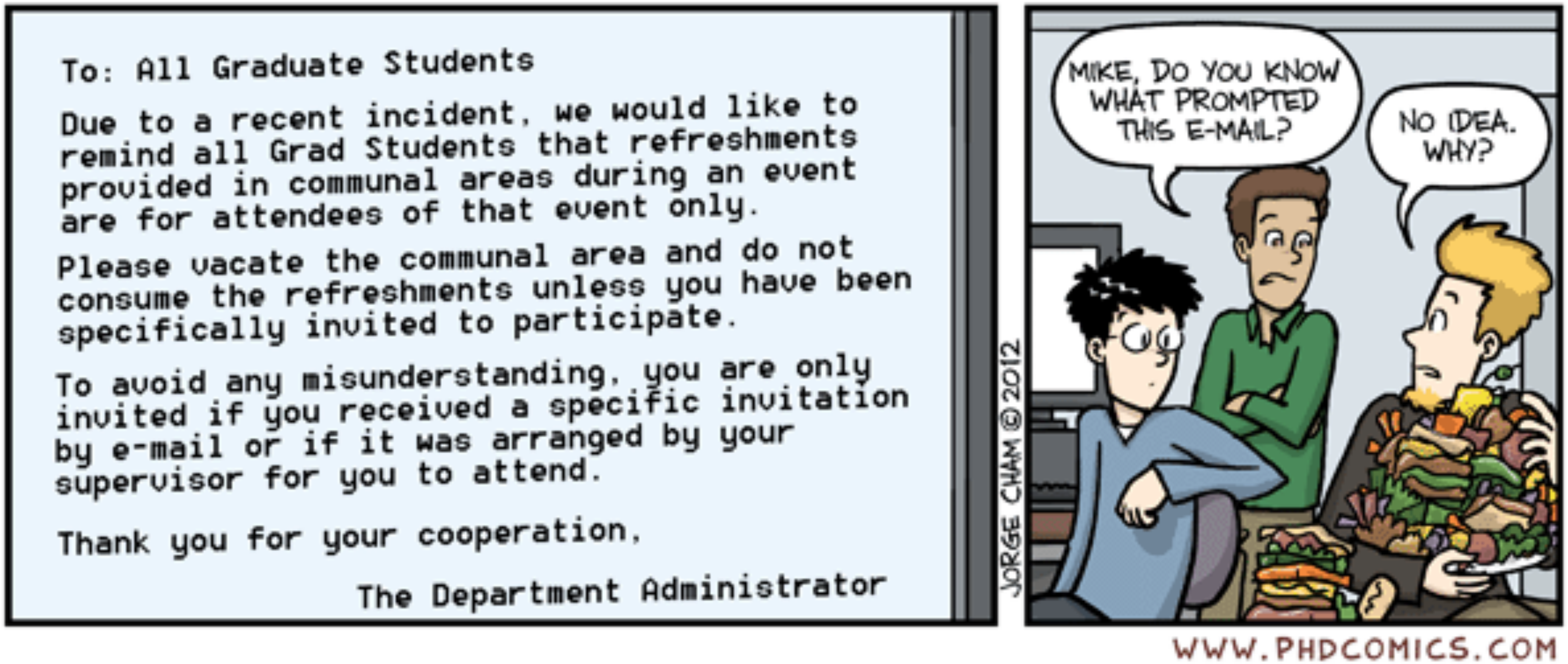
At the end of the day…

Guaranteed consistency with a consensus protocol

A buggy mess

“Eventual Consistency”

Sounds reasonable in theory…
What about in practice?

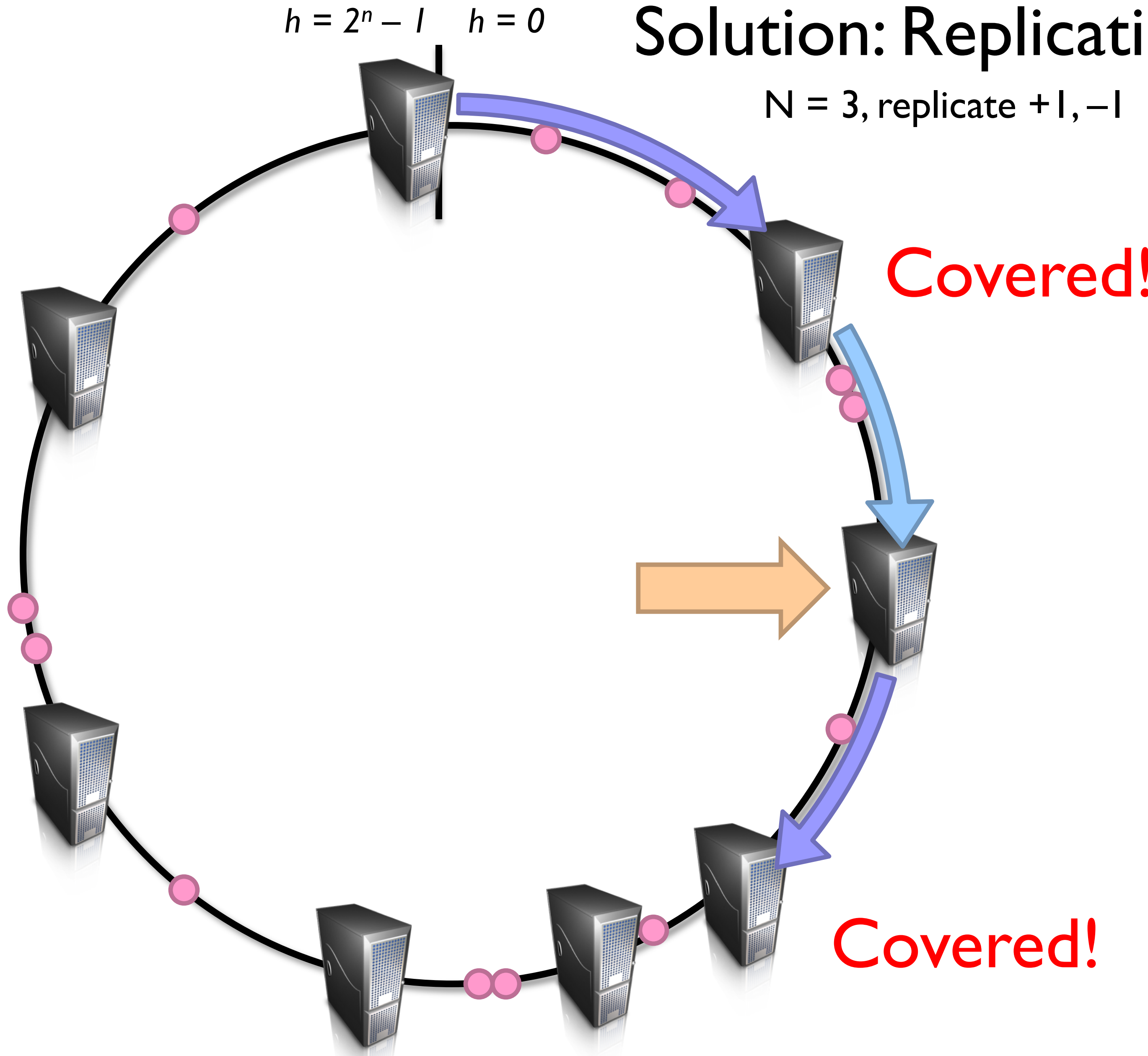


Morale of the story: there's no free lunch!
(Everything is a tradeoff)

Source: www.phdcomics.com/comics/archive.php?comicid=1475

Solution: Replication

N = 3, replicate +1, –1



Machine fails: What happens?

HBase

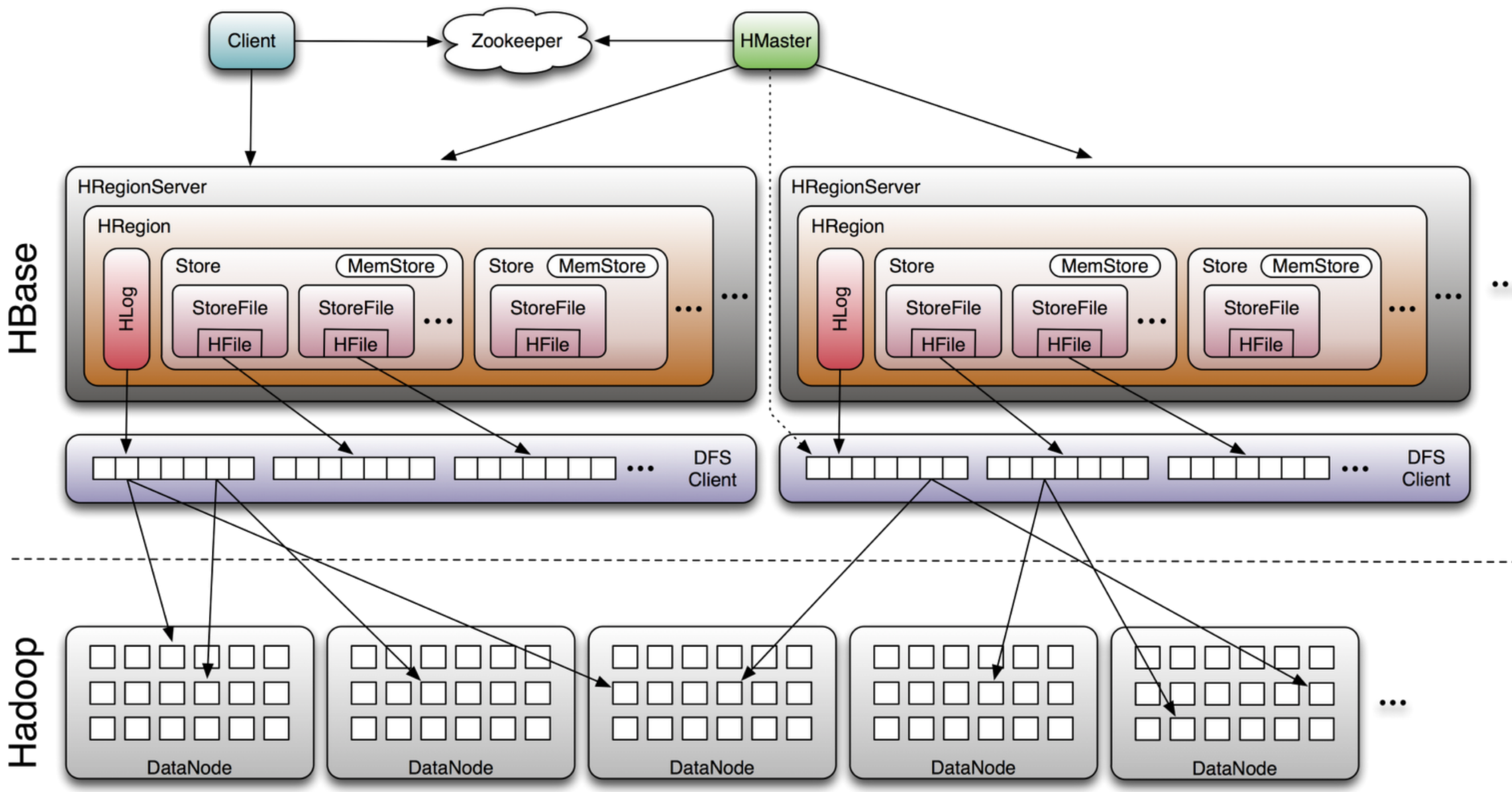


Image Source: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

Three Core Ideas

Why do these scenarios happen?

Partitioning (sharding)

To increase scalability and to decrease latency

Need distributed transactions!

Replication

To increase robustness (availability) and to increase throughput

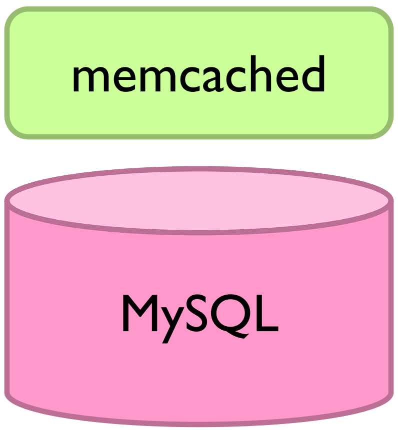
Need replica coherence protocol!

Caching

To reduce latency

Need cache coherence protocol!

Facebook Architecture



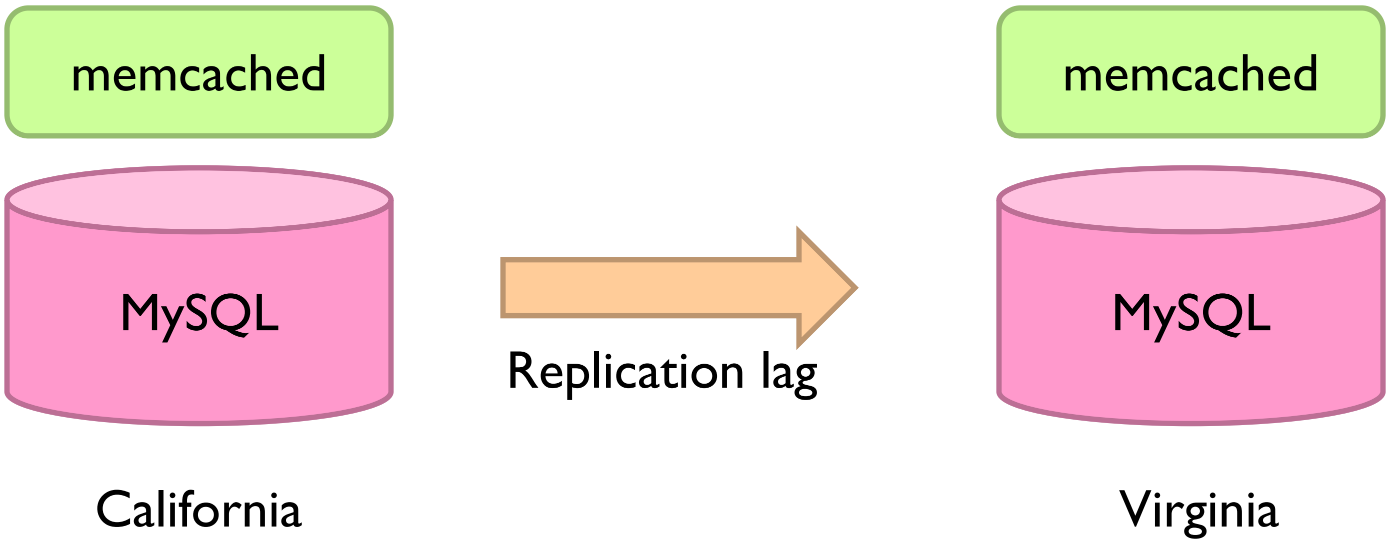
Read path:
Look in memcached
Look in MySQL
Populate in memcached

Write path:
Write in MySQL
Remove in memcached

Subsequent read:
Look in MySQL
Populate in memcached

Source: www.facebook.com/note.php?note_id=23844338919

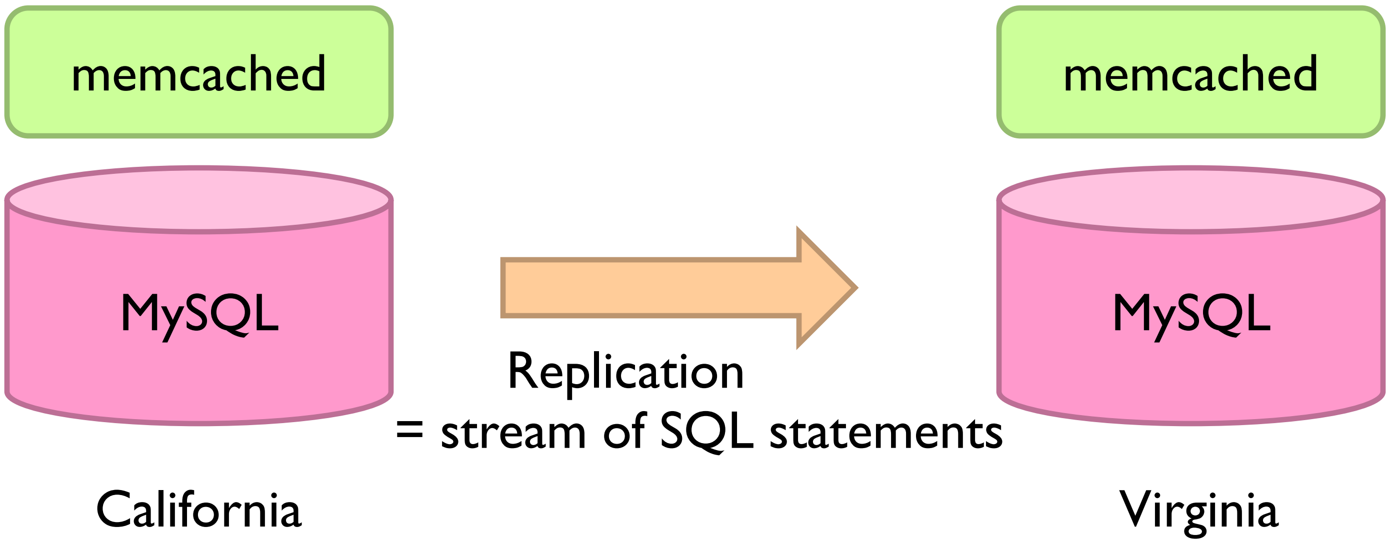
Facebook Architecture: Multi-DC



1. User updates first name from “Jason” to “Monkey”.
2. Write “Monkey” in master DB in CA, delete memcached entry in CA and VA.
3. Someone goes to profile in Virginia, read VA replica DB, get “Jason”.
4. Update VA memcache with first name as “Jason”.
5. Replication catches up. “Jason” stuck in memcached until another write!

Source: www.facebook.com/note.php?note_id=23844338919

Facebook Architecture: Multi-DC



Solution: Piggyback on replication stream, tweak SQL

```
REPLACE INTO profile (`first_name`) VALUES ('Monkey’)
WHERE `user_id`='jsobel' MEMCACHE_DIRTY 'jsobel:first_name'
```

Source: www.facebook.com/note.php?note_id=23844338919

Three Core Ideas

Why do these scenarios happen?

Partitioning (sharding)

To increase scalability and to decrease latency

Need distributed transactions!

Replication

To increase robustness (availability) and to increase throughput

Need replica coherence protocol!

Caching

To reduce latency

Need cache coherence protocol!

Source: Google

tl;dr -

Single row transactions

Easy to implement, obvious limitations

Implement a global consensus protocol for every transaction

Guarantee consistency, but slow

Eventual consistency

Who knows?

tl;dr -

Single row transactions

Easy to implement, obvious limitations

Implement a global consensus protocol for every transaction

Guarantee consistency, but slow

Eventual consistency

Who knows?

Can we cheat a bit?

tl;dr -

Per partition

Implement a ~~global~~ consensus protocol for every transaction

Guarantee consistency, ~~but slow~~

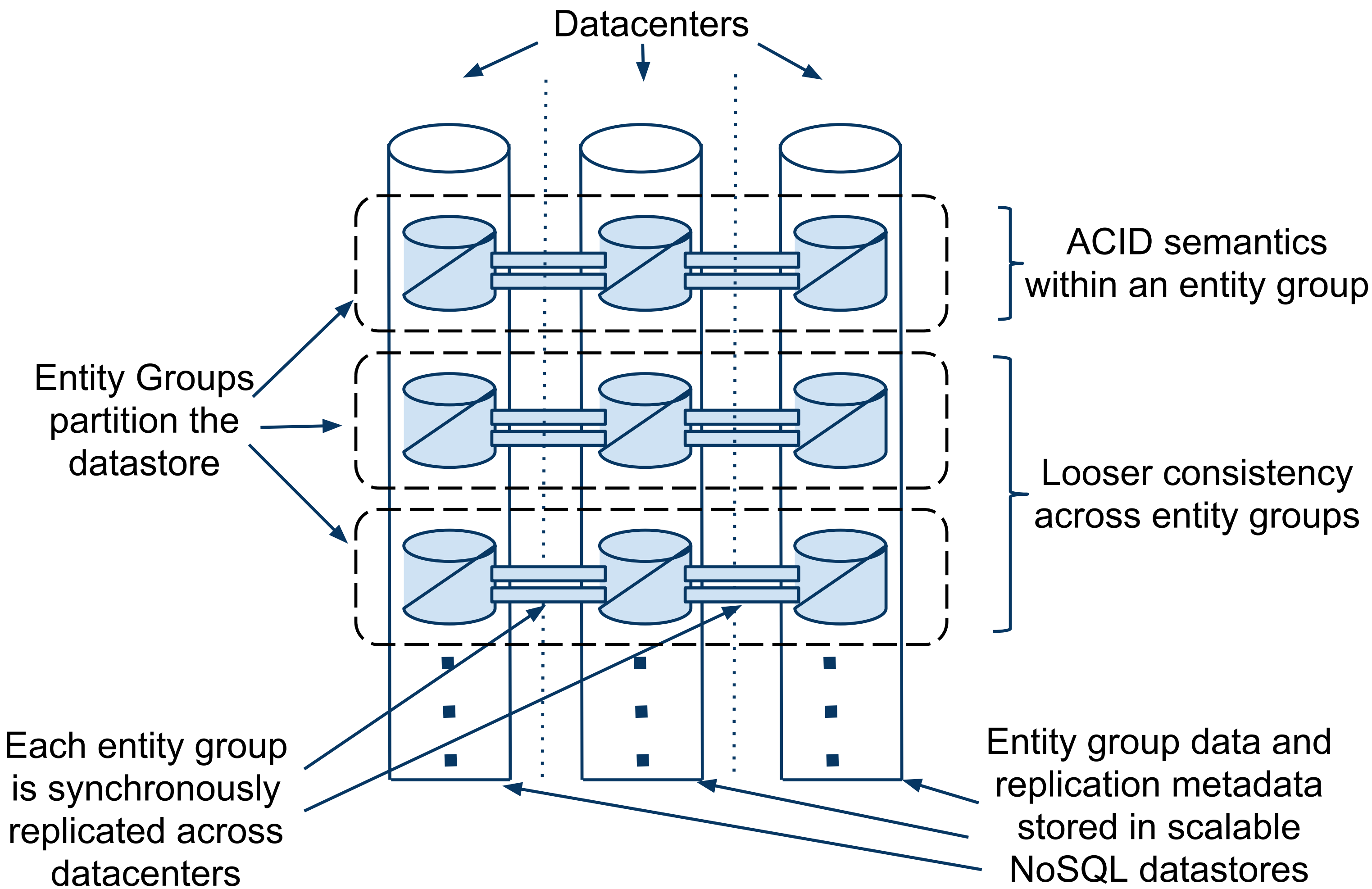
And fast!

Entity groups

Groups of entities that share affinity

Example: user + user's photos + user's posts etc.

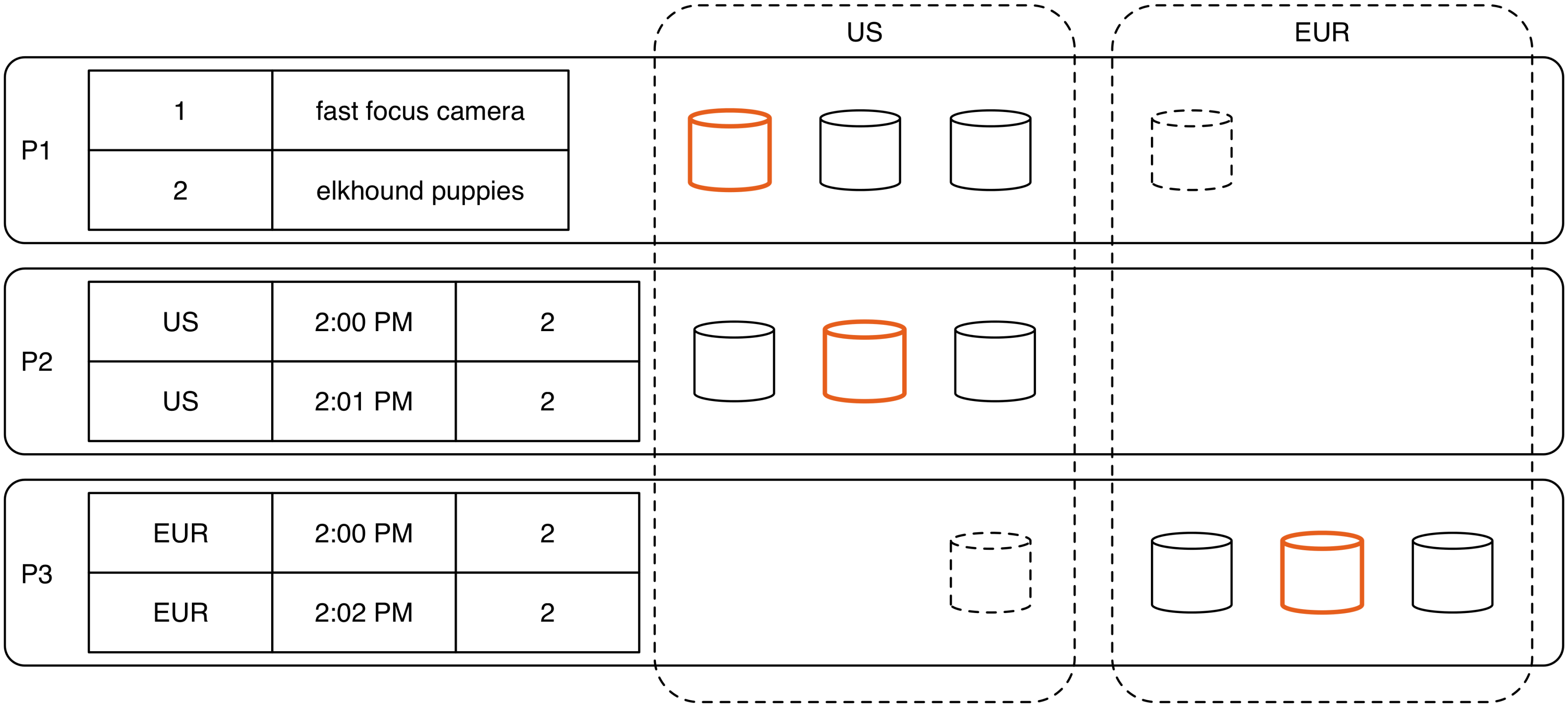
Google's Megastore



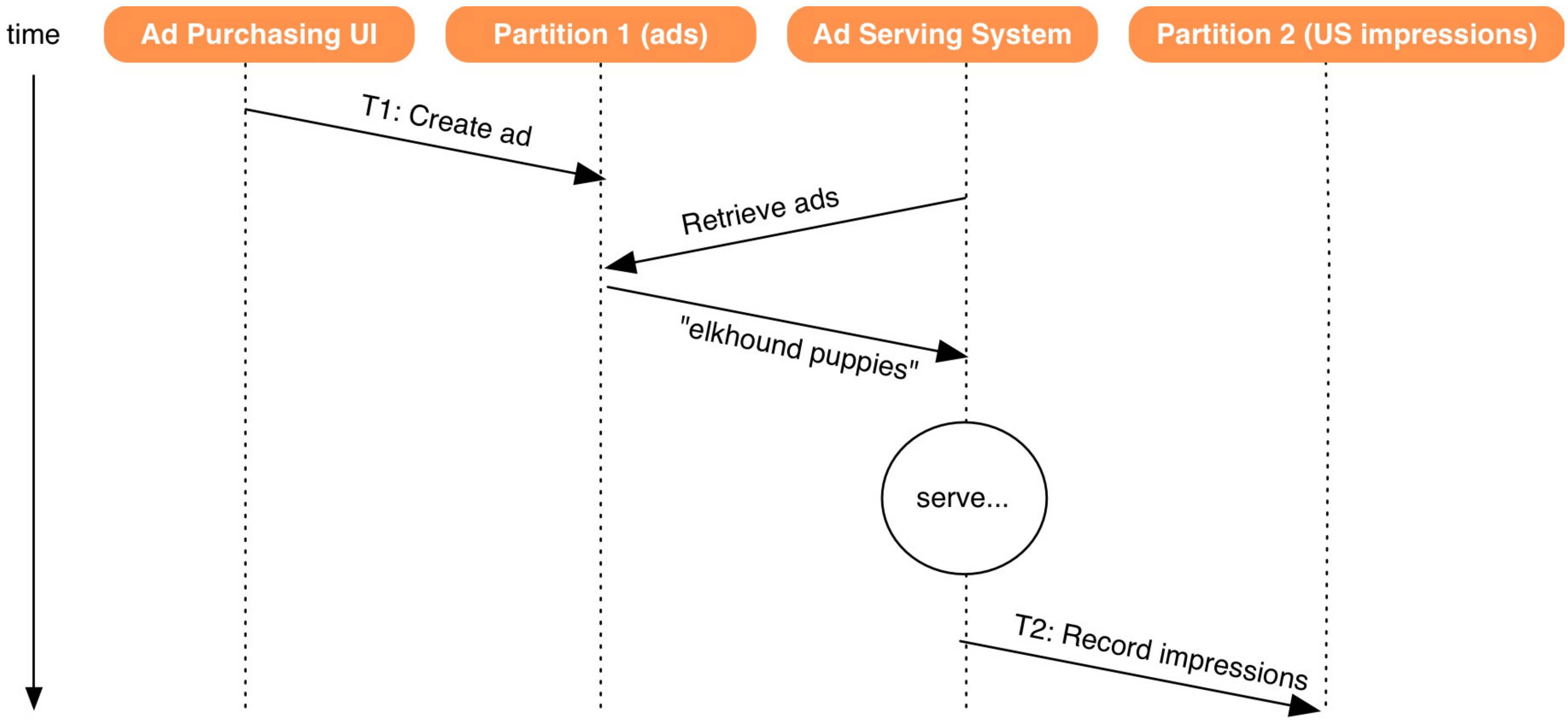
Source: Baker et al., CIDR 2011

But what if that’s not enough?

Preserving commit order: example schema



Preserving commit order



Source: Llyod, 2012

Snapshot MapReduce and queries

Initial state	
T1@ts1	INSERT INTO ads VALUES (2, “elkhound puppies”)
T2@ts2	INSERT INTO impressions VALUES (US, 2PM, 2)

Source: Llyod, 2012

Google's Spanner

Features:

Full ACID translations across multiple datacenters, across continents!

External consistency (= linearizability):

system preserves *happens-before* relationship among transactions

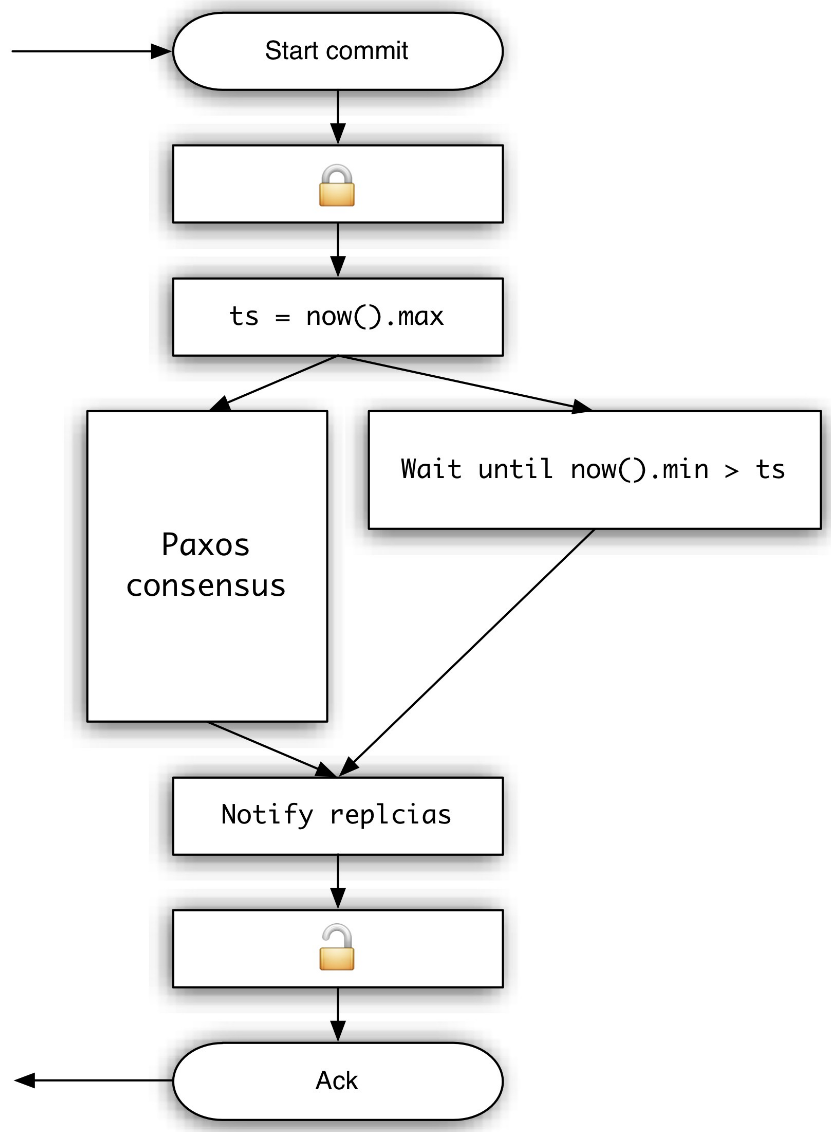
How?

Given write transactions A and B, if A *happens-before* B, then

timestamp(A) < timestamp(B)

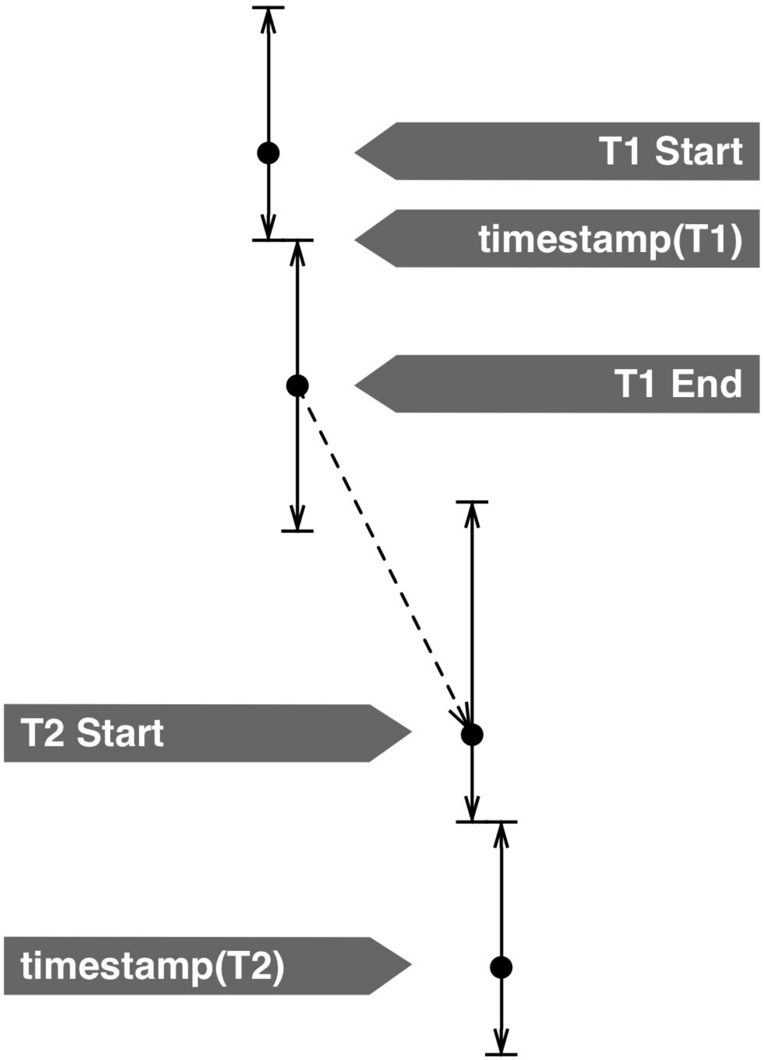
Source: Llyod, 2012

TrueTime → write timestamps



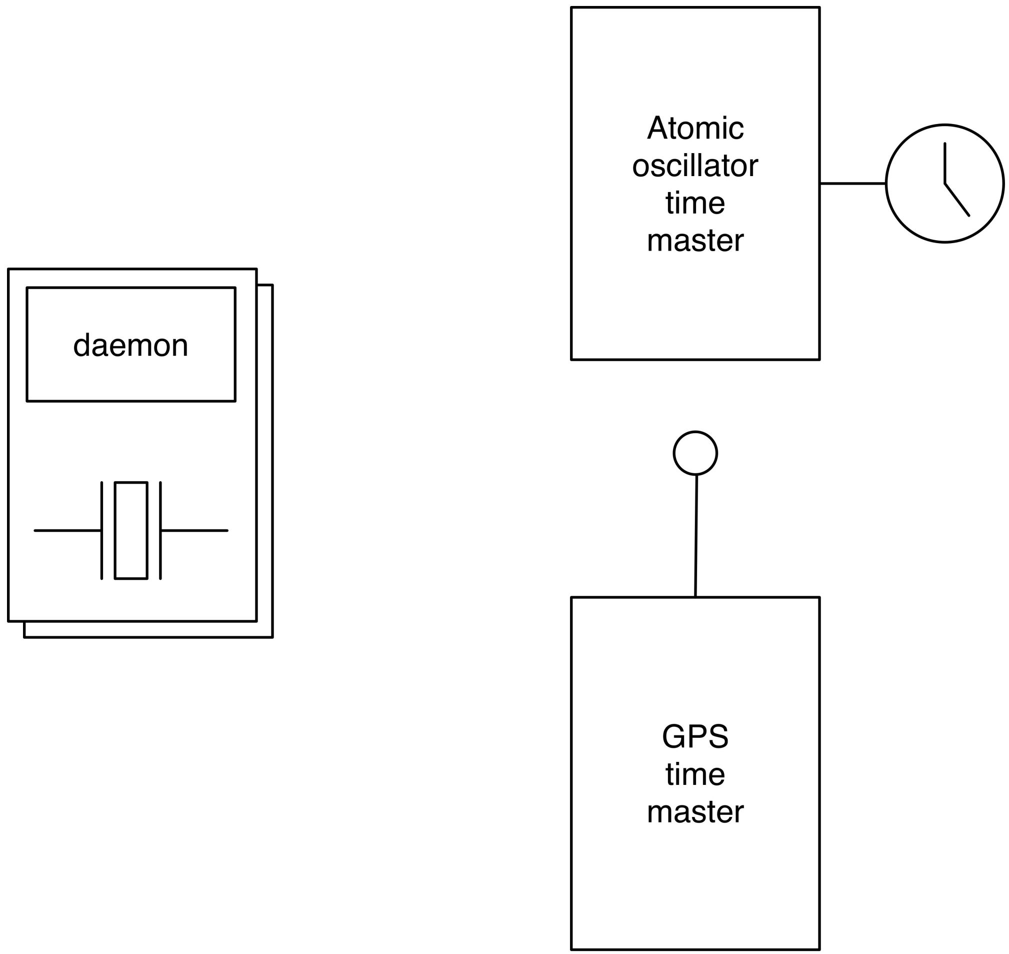
Source: Llyod, 2012

Why this works

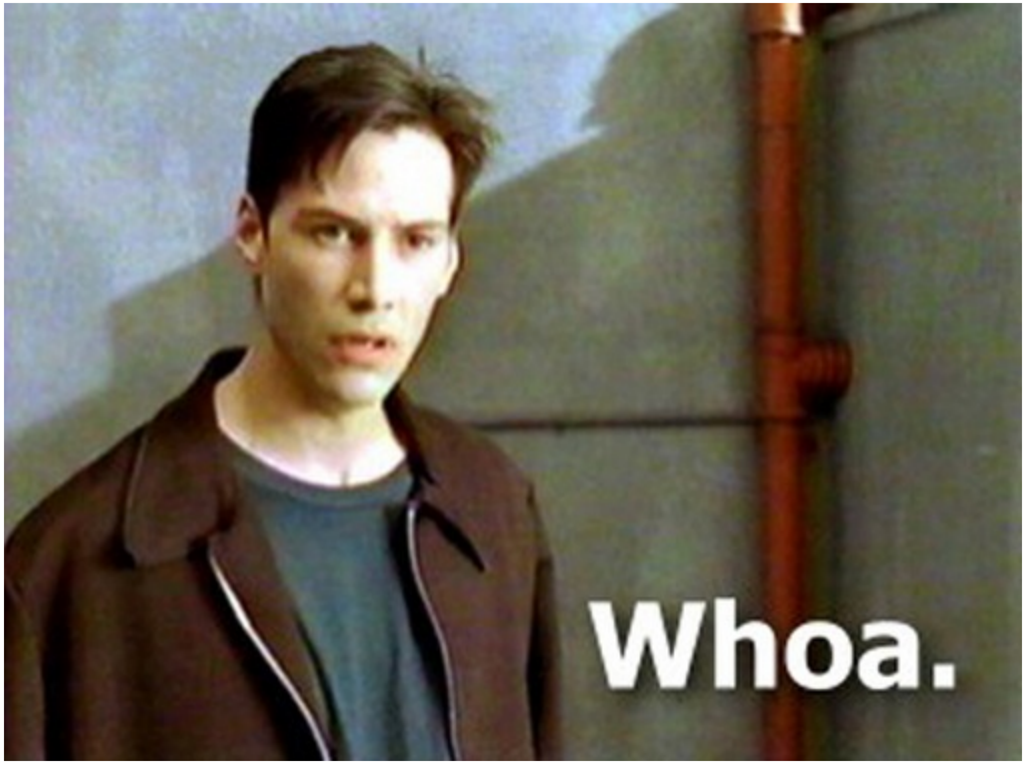


Source: Llyod, 2012

TrueTime



Source: Llyod, 2012



What's the catch?

Source: The Matrix

Three Core Ideas

Partitioning (sharding)

To increase scalability and to decrease latency

Need distributed transactions!

Replication

To increase robustness (availability) and to increase throughput

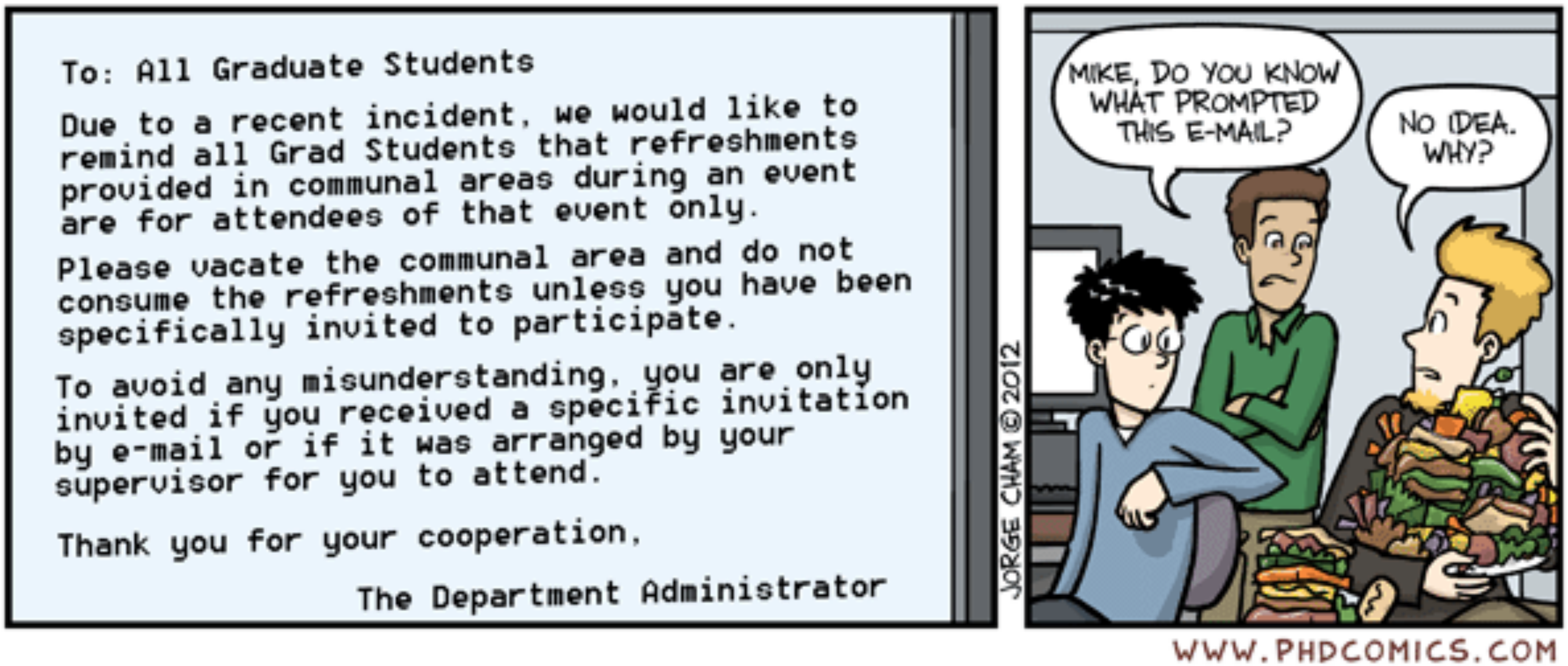
Need replica coherence protocol!

Caching

To reduce latency

Need cache coherence protocol!

Source: Wikipedia (Cake)



Morale of the story: there's no free lunch!

(Everything is a tradeoff)

Source: www.phdcomics.com/comics/archive.php?comicid=1475