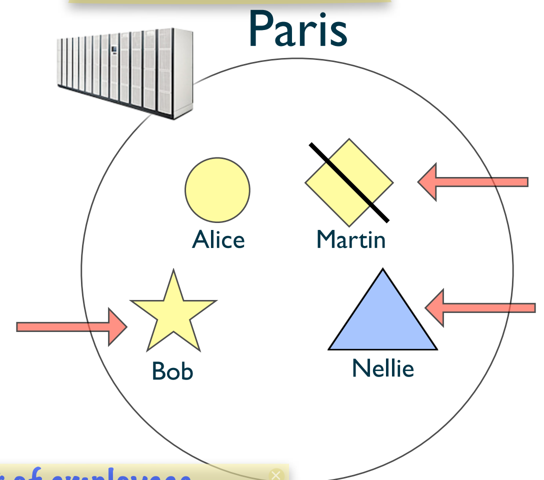
High Concurrency, and Correctness Too

or: How I Stopped Worrying about Eventual Consistency

Marc Shapiro
Nuno Preguiça
Carlos Baquero
Annette Bieniusa
Marek Zawirski

Cloud Computing



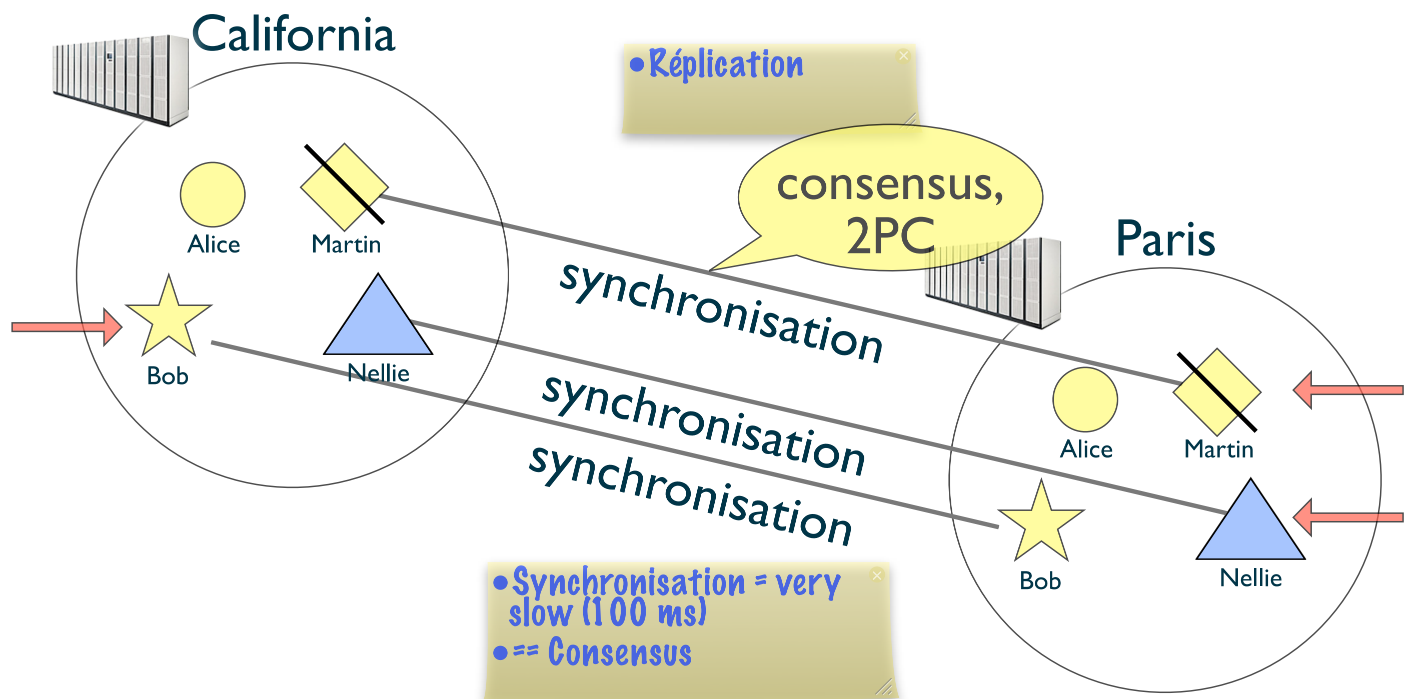
Cloud: powerful compute centre, parallel, fast = O(100.000) transactions/s [Padilha OSDI 2013]

Ex. app: the set of employees of some company

Updates:

- Update Nellie (e.g. increase salary)
- Create Bob
- Remove Martin

Strongly consistent replication

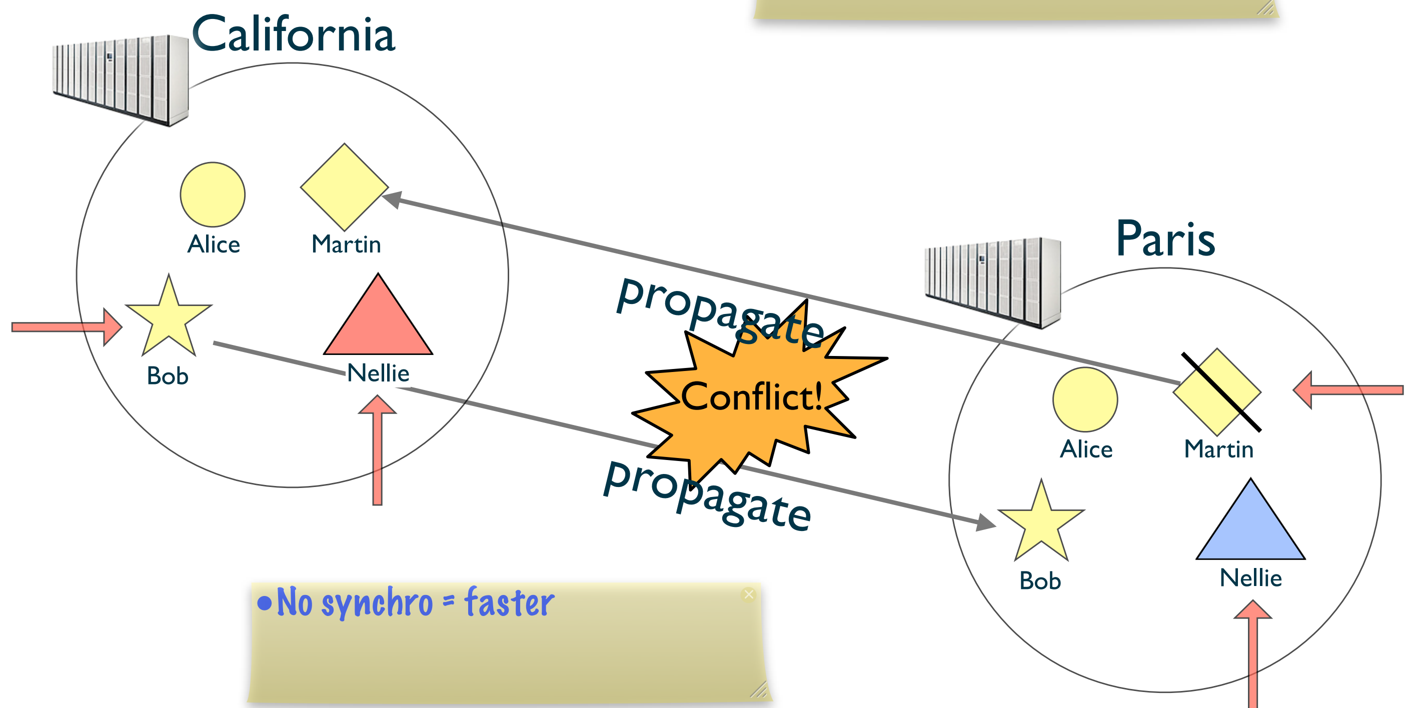


Distributed company:

- slow network ⇒ replicate data

- fast reads
- but synchro for updates is slow; wasting parallelism and compute power

(Synch = 100 ms ⇒ 10 transactions/second!!!)



How about propagating updates asynchronously, rather than synchronising them?

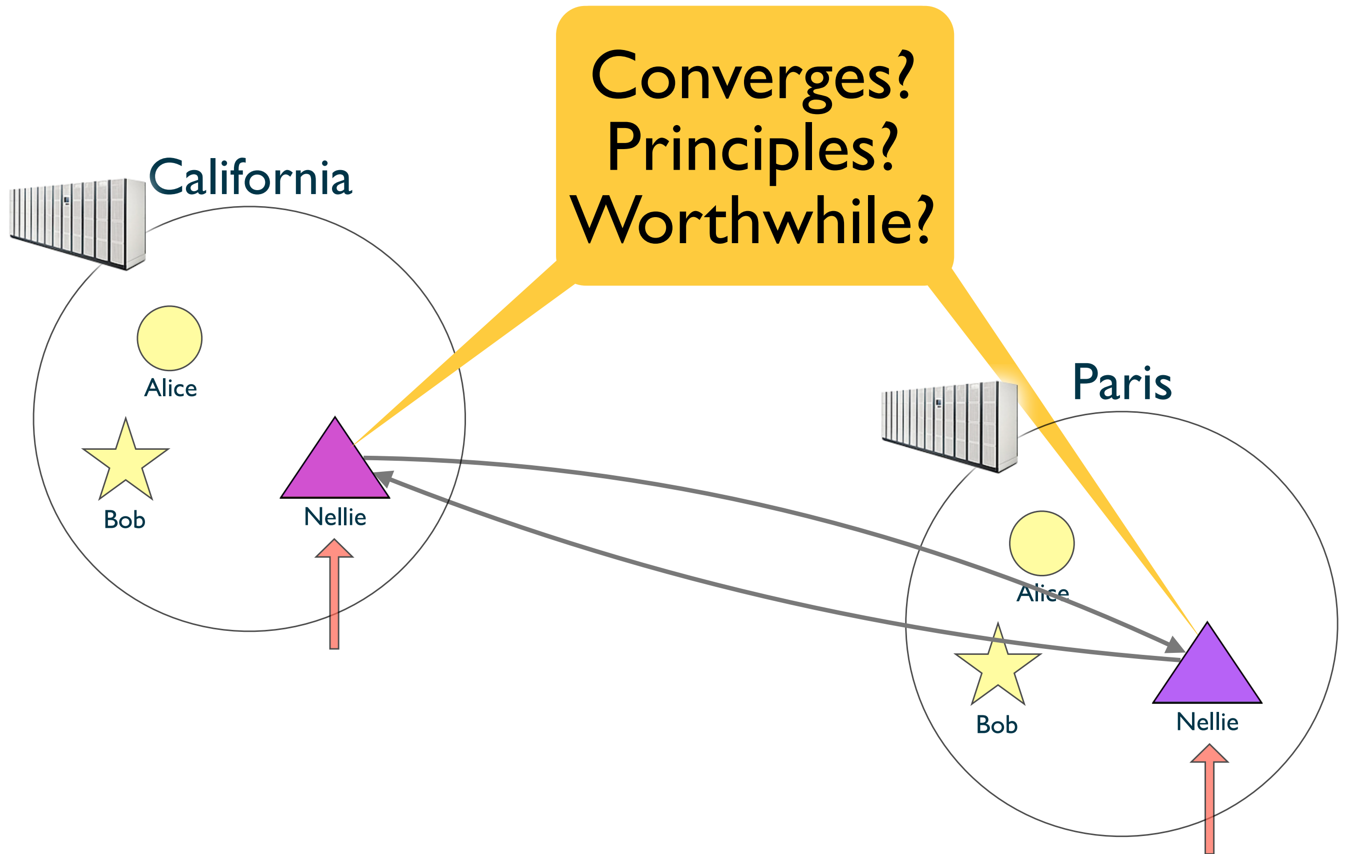
A data centre makes progress without waiting for the other one

Concurrent updates (Martin || Bob)

Concurrent updates to Nellie: conflict????

California: increase Nellie salary

Paris: change Nellie's department



EC:

- asynch propagation
- merge concurrent updates

But how? Up to the user? Synchronise to ensure convergence? Depends on semantics?

Outline

Why Eventual Consistency (EC)?
- What principles?

Conflict-free Replicated Data Types (CRDTs)
- Design principles
- Set

Strengthening EC + CRDT with useful guarantees
- Preserve asynchronous properties

Results, open questions

Conflict-free Replicated Data Types (CRDTs)

What interesting objects can we design with no synchronisation whatsoever?

Register
- Last-Writer Wins
- Multi-Value

Set
- Grow-Only
- 2P
- Observed-Remove

Map

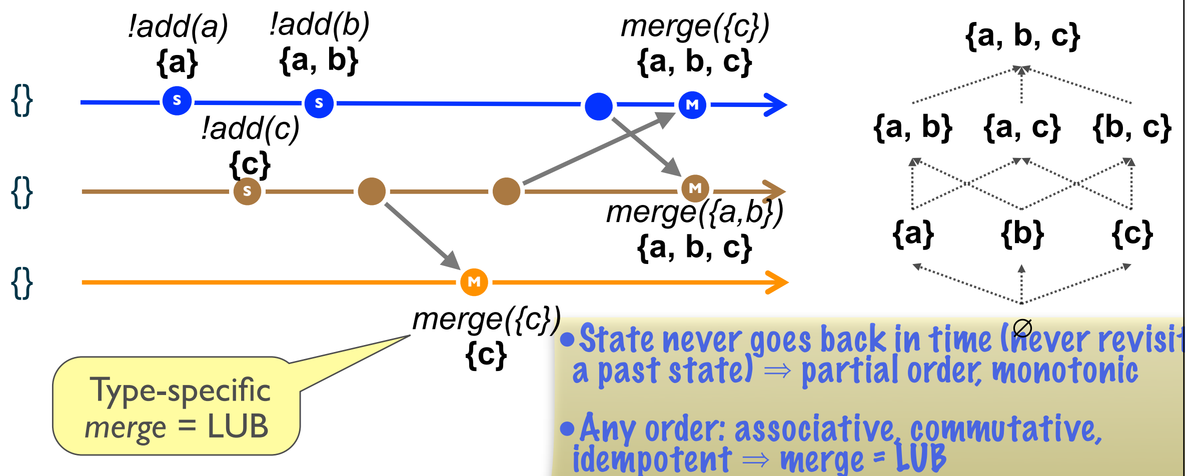
Counter
- Unlimited
- Restricted non-negative

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence

Power of consensus: any sequential specification
What is the power of asynchronous systems?

Grow-only set



Ensure concurrent updates are deterministic

Sufficient condition: *"Monotonic semi-lattice:"*

- Partial order
- Monotonic updates
- *Merge* computes Least Upper Bound
- *Merge* eventually delivered infinite #times

Epidemic: send state to others; merge received state
Merge is type-specific
monotonic + LUB ⇒

LUB is commutative, associative, idempotent
payload ↗
epidemic propagation
state-based: inefficient for large payload

Asynchronous Set designs

G-Set: Grow-only set + union merge

- No *remove*

2P-Set: [Wuu & Bernstein PODC 1984]

- Grow-only elements + grow-only tombstones
- Can't *add;remove;add*

Dynamo MV-Register [deCandia SOSP 2007]

- Assignment
- Removed elements re-appear

c-set: [Sovran et al., SOSP 2011]

- Count number of *add/remove* operations
- Anomalous *add/remove*

Extending the Set seq. spec.

Sequential specification of Set:

- $\{true\}$!add(e) $\{e \in S\}$
- $\{true\}$!rmv(e) $\{e \notin S\}$

Commutative ($e \neq f$):

- $\{true\}$!add(e)|| !add(e) $\{e \in S\}$
- $\{true\}$!rmv(e)|| !rmv(e) $\{e \notin S\}$
- $\{true\}$!add(e)|| !add(f) $\{e,f \in S\}$
- $\{true\}$!rmv(e)|| !rmv(f) $\{e, f \notin S\}$
- $\{true\}$!add(e)|| !rmv(f) $\{e \in S, f \notin S\}$

Ambiguous:

- $\{true\}$!add(e)|| !rmv(e) {????}

Extending the Set seq. spec.

Sequential specification of Set:

- $\{true\}$!add(e) $\{e \in S\}$
- $\{true\}$!rmv(e) $\{e \notin S\}$

Commutative ($e \neq f$):

- $\{true\}$!add(e) ; !add(e) $\{e \in S\}$
- $\{true\}$!rmv(e) ; !rmv(e) $\{e \notin S\}$
- $\{true\}$!add(e) ; !add(f) $\{e,f \in S\}$
- $\{true\}$!rmv(e) ; !rmv(f) $\{e, f \notin S\}$
- $\{true\}$!add(e) ; !rmv(f) $\{e \in S, f \notin S\}$

What about:

- $\{true\}$!add(e) || !rmv(e) $\{????\}$

“Principle of permutation equivalence”

- All sequential composition have same effect ⇒ let parallel composition have same effect

Extending the Set seq. spec.

Sequential specification of Set:

- $\{true\}$!add(e) $\{e \in S\}$
- $\{true\}$!rmv(e) $\{e \notin S\}$

Commutative ($e \neq f$):

- $\{true\}$!add(e) || !add(e) $\{e \in S\}$
- $\{true\}$!rmv(e)|| !rmv(e) $\{e \notin S\}$
- $\{true\}$!add(e) || !add(f) $\{e,f \in S\}$
- $\{true\}$!rmv(e)|| !rmv(f) $\{e, f \notin S\}$
- $\{true\}$!add(e) || !rmv(f) $\{e \in S, f \notin S\}$

What about:

- $\{true\}$!add(e) || !rmv(e) $\{????\}$

!add(e) || !rmv(e)

{*true*} !add(e) || !rmv(e) {*????*}

- ~~sequential consistency, linearisable?~~
- last writer wins? $\{\ !add(e) < !rmv(e) \Rightarrow e \notin S$ $\wedge\ !rmv(e) < !add(e) \Rightarrow e \in S\ \}$
- error state? $\{\perp_e \in S\}$
- add wins? $\{e \in S\}$
- remove wins? $\{e \notin S\}$

Deterministic

- Independent of order of delivery
- Independent of local state
- No synchronisation

Not an interleaving semantics

Any of these is an acceptable semantics, i.e., ensures convergence

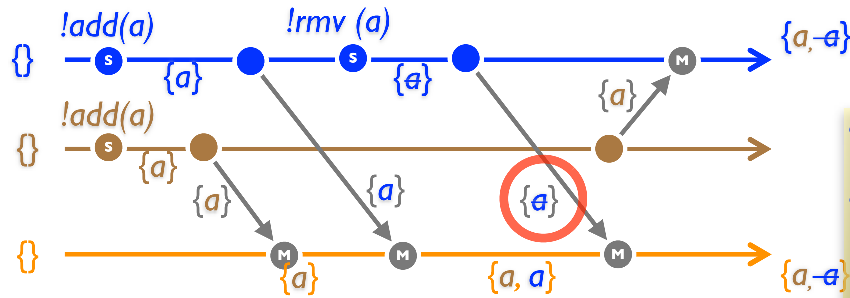
Satisfies the Monotonic Semi-Lattice property

Which one you choose depends on application semantics

Better than previous app-dependent approaches:

- encapsulated in a data type
- no synchronisation
- guaranteed correct

Add-Wins Set (OR-Set)



- Timestamps assumed unique
- Can never remove more tokens than exist
- Op order ⇒ removed

{a}!add(a) = {a, a}	*unique*
{a, b}!rmv(a) = {~~a~~, b}	*mark visible instances of a*
{~~a~~, a}?contains(a) | true	*∃ non-marked instance of a*
{a, b}.merge({~~a~~, c}) = {~~a~~, b, c}	*union + marker*

$\{true\}$!add(e) || !rmv(e) $\{e \in S\}$

In case of !add+remove, OR-Set gives precedence to !add

Garbage-collecting tombstones

!rmv(e)

- Discard information about e?
- Merging *S* (e ∈ *S*) and *S*' (e ∉ *S*'):
  Has e been added to *S*? or removed from *S*'?

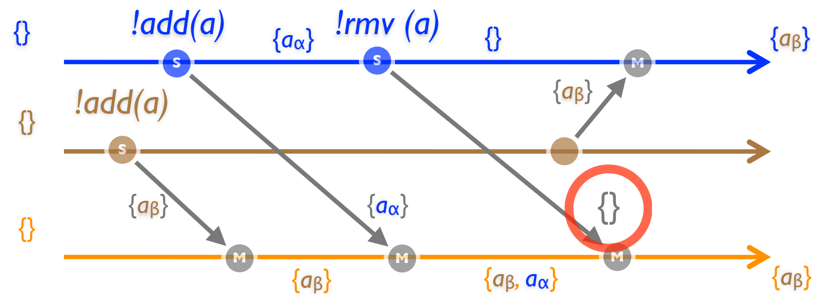
Insight:

- Maintain summary of UIDs generated
- Compare to summary

Solution:

- UID = Lamport *(process, counter)*
- Summary = vector clock *[process → counter]*

OR-Set: no tombstones (1)

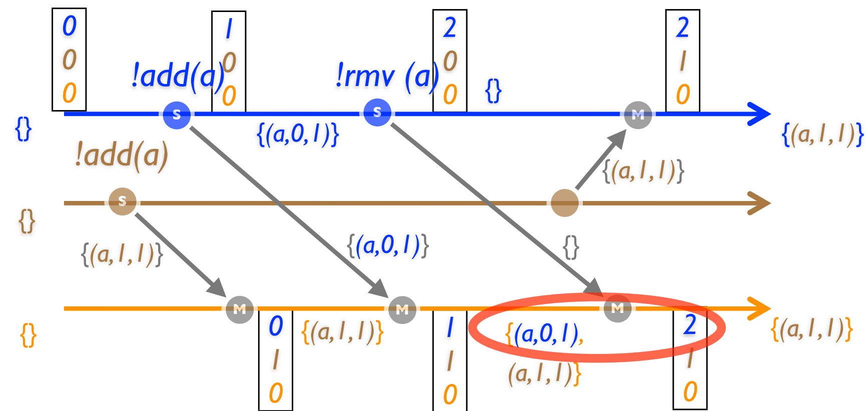


Assume (for now) delivery in happens-before order

Replica 3: $\{a_\alpha\}$ followed by $\{\}$ $\Rightarrow$ a_α was removed

No tombstone required

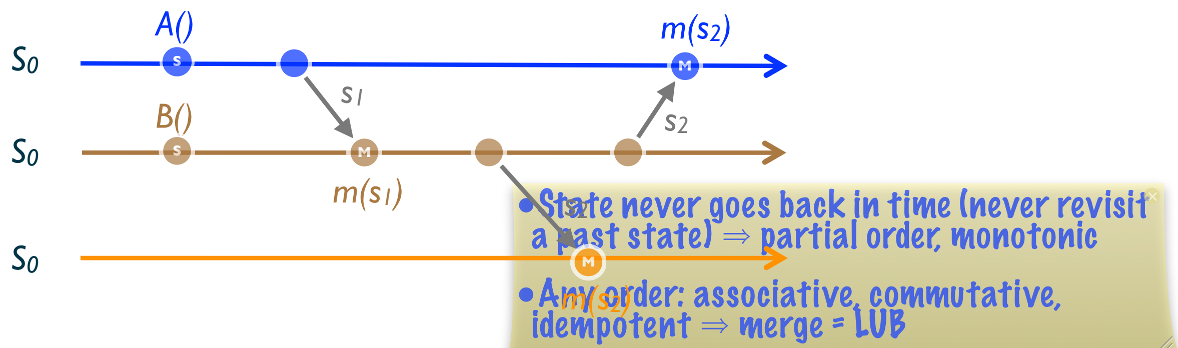
OR-Set: no tombstones (2)



State-based + unordered delivery

- Unique identifier = replica *i* + clock $V_i[i]$
- If element missing & clock dominates = removed

State-based CRDT



Updates have effect
No rollback

Sufficient condition: "*Monotonic semi-lattice*"

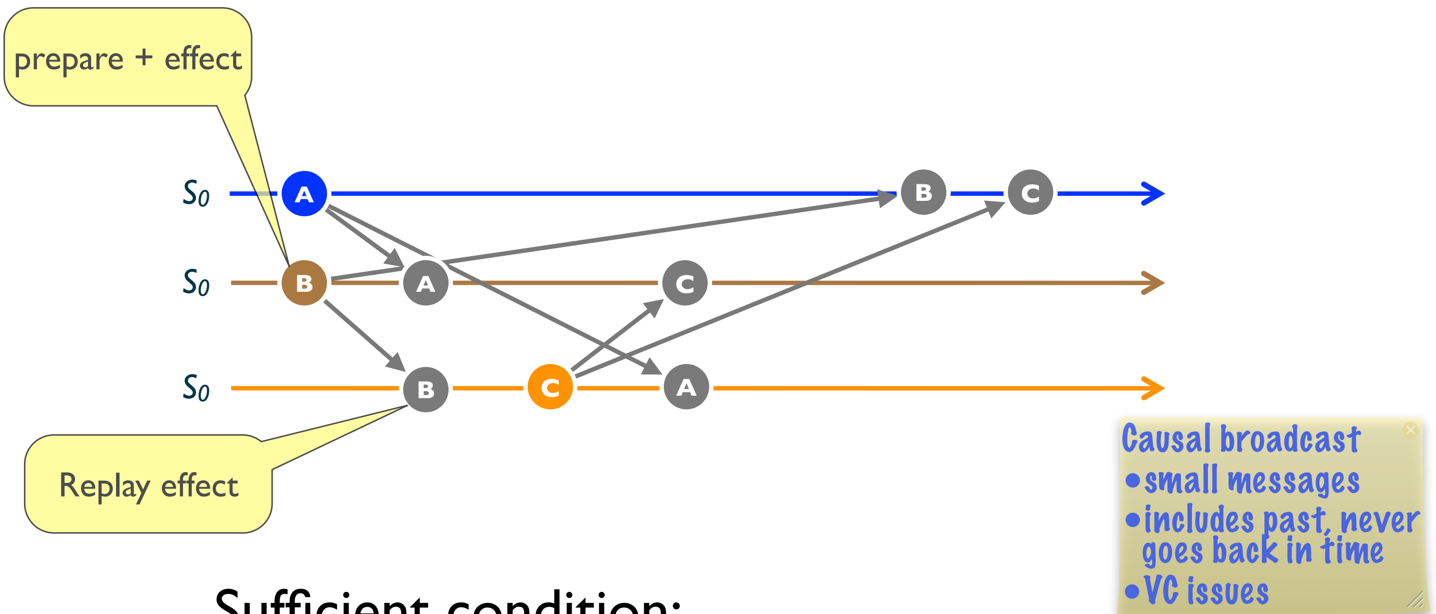
- Partial order
- Monotonic
- *Merge* computes Least Upper Bound
- *Merge* eventually delivered

Delivery order unimportant

monotonic + LUB ⇒

LUB is commutative, associative, idempotent
payload ↗
epidemic propagation
state-based: inefficient for large payload

Op-based CRDT



Sufficient condition:

- Reliable causal delivery
- Concurrent operations "commute"

Design principles

Mergeable Eventual Consistency:

- Parallel, unsynchronised updates
- Deterministic *merge*

Conflict-Free Replicated Data Types:

$\vee$ Epidemic + monotonic semi-lattice
$\vee$ Causal + "commutative" concurrent updates

Semantics:

- Sequential histories behave as expected
- Commute $\Rightarrow$ principle of equivalence
- Otherwise: deterministic, context-independent

Prepare vs. effect

Pruning state: must not violate monotonicity

Conflict-free Replicated Data Types (CRDTs)

What interesting objects can we design with no synchronisation whatsoever?

Register
- Last-Writer Wins
- Multi-Value

Set
- Grow-Only
- 2P
- Observed-Remove

Map

Counter
- Unlimited
- Restricted non-negative

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence

Power of consensus: any sequential specification
What is the power of asynchronous systems?

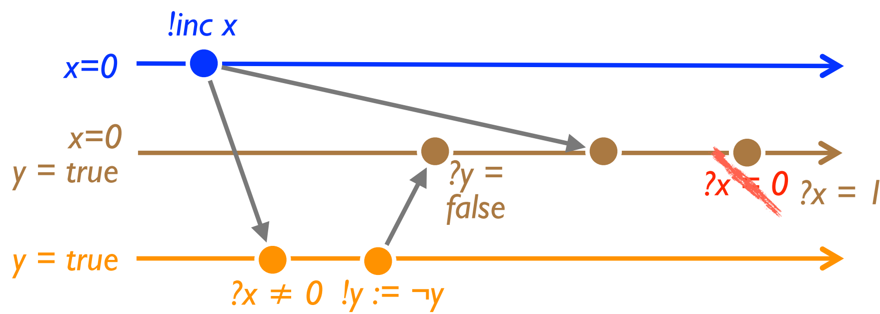
Extra guarantees — but no consensus

Without synchronisation:

- Causal consistency
- Transactions
- Mixed asynchronous + synchronous
- Fault tolerance

- No synchronisation for the blue operations
- Obviously the red ones require consensus among themselves

Causal consistency



Knowledge (pre- / post-conditions) ¬decreasing

Nothing missing: transitively closed

Vector clock = O (|Masters|)

No consensus

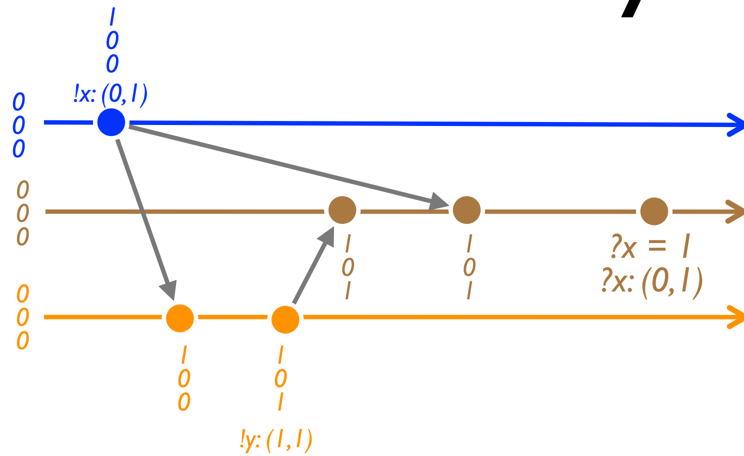
Minimal, natural extension of Program Order
(total order would require consensus)

Implementing causal consistency: classical

Delay delivery of message until its full causal context is ensured

Assumes all data is read

Implementing causal consistency



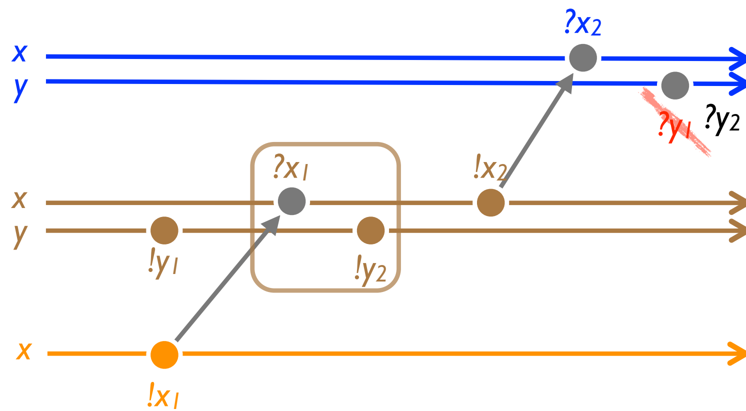
Tag data with update timestamp

Detect & wait for missing version

VC = not new

Per-object causal order = new (I think)

Causal transactions



Read a causally-consistent state

All-or-nothing, isolated updates

No consensus (not ACID)

Transaction: group of queries and updates

- Atomic: all-or-nothing updates
- Read snapshot: causally-consistent cut
- Atomicity extends causality across objects
- Session guarantees: snapshot includes prior updates
- No aborts: asynchronous

Atomic, Locally Consistent, Isolated, Durable; no synchronisation ⇒ ¬serialisable, ¬SI

Red/Blue consistency

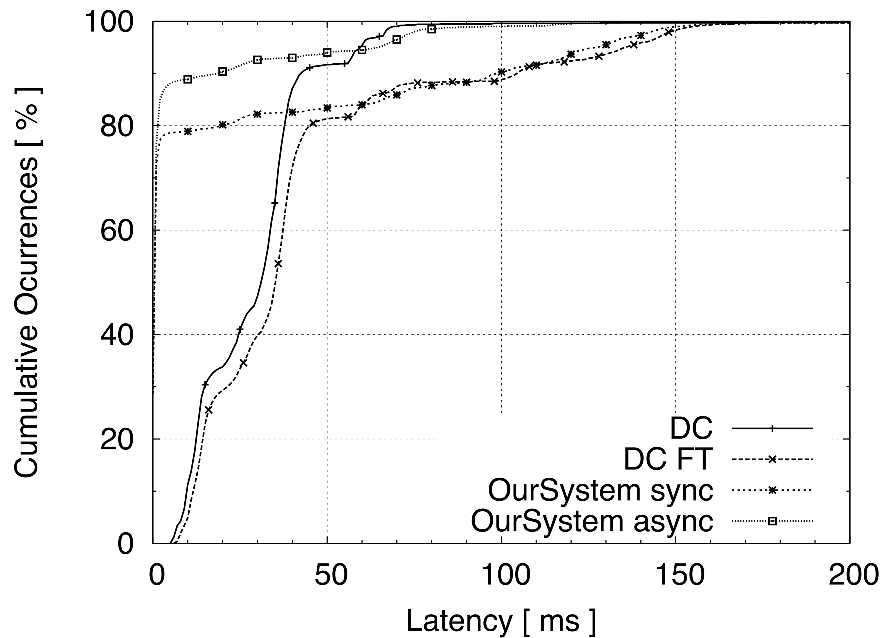
Mix synchronous (red) and asynchronous (blue) updates

Correctness

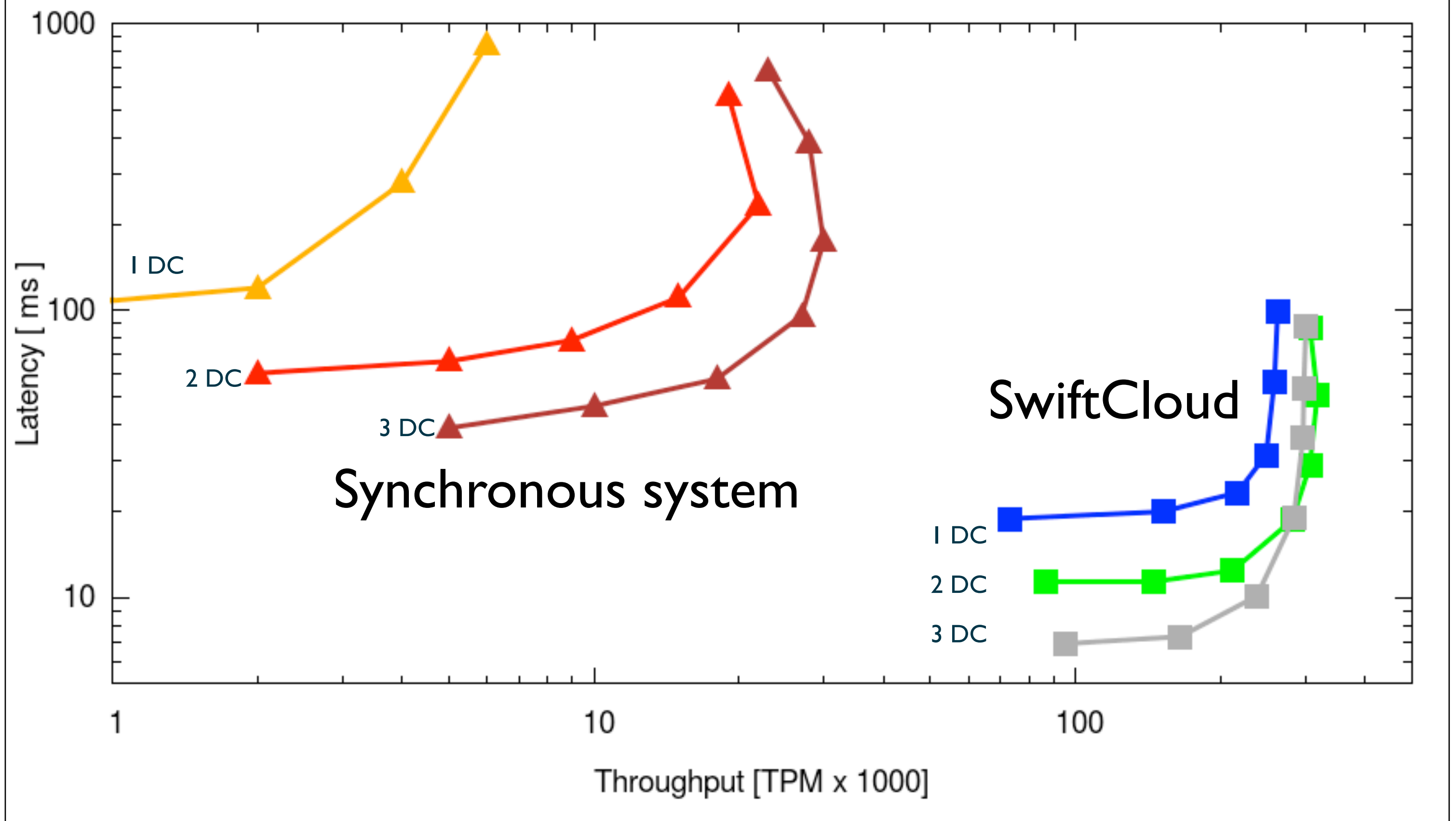
- All reads and updates are causally ordered
- Blue updates commute with both red and blue ones
- Red updates are totally ordered with red ones (but not with blue ones)

[Li et al. OSDI 2012]

Operation response time



Eventual =? “Sometime in a distant future”?
You get your answers much more quickly than with the synchronous approach



EC = “sometime in the distant future”?? Responds faster than synchronous.

SwiftCloud performance [both latency + throughput] 10x classical geo-replication, even though SwiftCloud is fault tolerant and SOA-noFT is not!
⇒ HW cost /10

Explanation: SwiftCloud absorbs 90% of transactions (read + write!) in cache

(Tables in next slide: "(do not show"))
- 1 DC: TPM ratio ∈ [12, 250]; throughput ratio ∈ [2, 45]
- 2 DC: TPM ratio ∈ [16, 43]; throughput ratio ∈ [5, 11]
- 3 DC: TPM ratio ∈ [10, 20]; throughput ratio ≈ 5

Summary

CRDT

- Available, fast
- Reconcile scalability ∪ consistency
- Principled, correct by design
- Always mergeable
- High-level: Set, Tree, Sequence

SwiftCloud

- Extreme geo-replication
- Updates at the edge
- Conflict-free transactions
- Pruning maintains correctness

Open questions

What is possible without any consensus?
- Characterise invariants?
- Local invariants ⇒ global?

Bounded memory

Bounded ID space

State-operation duality: program transformations?

Mixed weak & strong consistency

No consensus = no synchronisation at all << “wait-free synchronisation"