From strong to eventual consistency: getting it right

Nuno Preguiça, U. Nova de Lisboa
Marc Shapiro, Inria & UPMC-LIP6

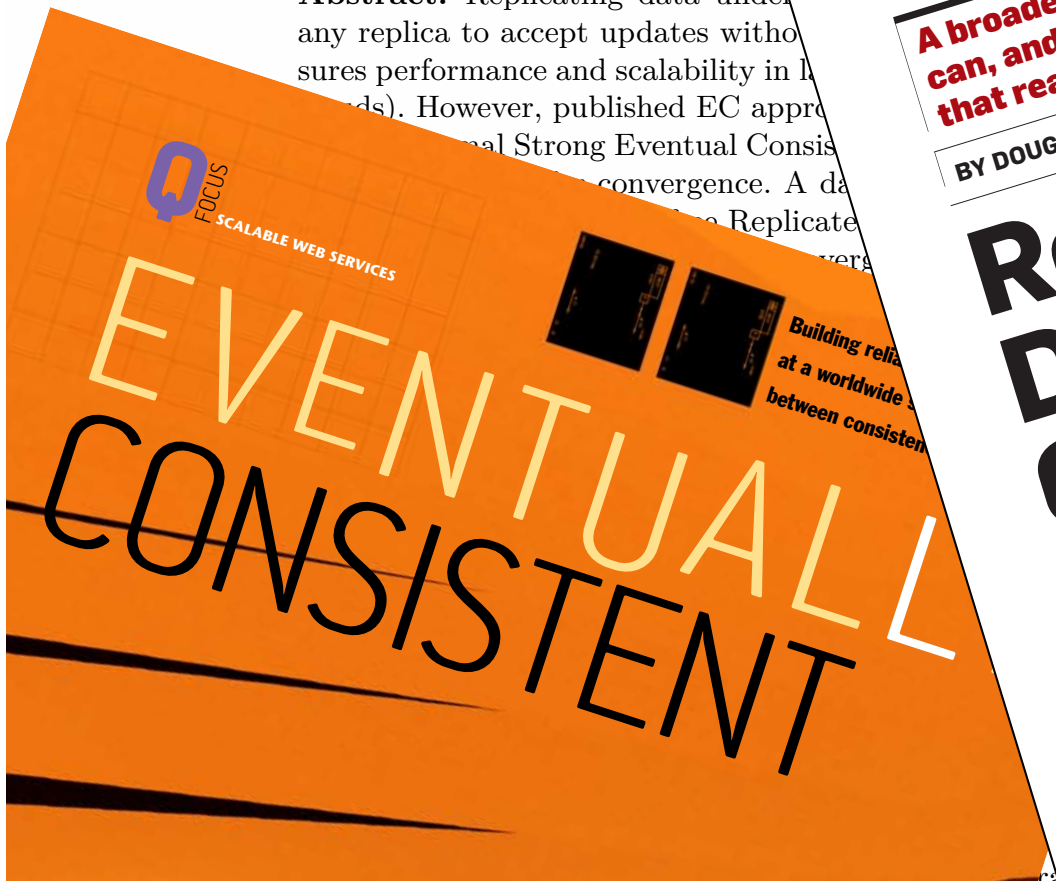
Conflict-free Replicated Data Types ⋆

Marc Shapiro[1,5], Nuno Preguiça[2,1], Carlos Baquero[3], and Marek Zawirski[1,4]

[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
[3] Universidade do Minho, Portug
[4] UPMC, Paris, Fr
[5] LIP6, P

Keywords: Eventual Consistency, I
tributed Systems

Abstract. Replicating data under
any replica to accept updates witho
sures performance and scalability in l
ds). However, published EC appro
al Strong Eventual Consis
convergence. A da
Replicate



review articles

DOI:10.1145/2500500

A broader class of consistency guarantees can, and perhaps should, be offered to clients that read shared data.

BY DOUG TERRY

Replicated Data Consistency Explained Through Baseball

SYSTEMS for the cloud deliver
tees to applications that
rage providers
so that

their applications.[5] These services ensure clients of Windows Azure Storage always see the latest value that was written for a data object. While strong consistency is desirable and reasonable to provide within a datacenter, it raises concerns as systems start to offer geo-replicated services that span multiple datacenters on multiple continents.

Many cloud storage systems, such as the Amazon Simple Storage Service (S3), were designed with weak consistency based on the belief that strong consistency is too expensive in large systems. The designers chose to relax consistency in order to obtain better performance and availability. In such systems, clients may perform read operations that return stale data. The data returned by a read operation is the value of the object at *some past point* in time, not necessarily the latest one. This occurs, for instance, when an update operation is directed to a replica that has not yet received all updates that were accepted by other replicas. Such systems are called *eventually consistent*.

Recent systems, which need to support diverse applications, have begun with a choice of consistency when accessing cloud data. Amazon DynamoDB, for example, offers both *eventually consistent* and *strongly consistent* reads. The latter experience higher latency and lower throughput and offers the same

(/papec/node/2274)

Strong consistency is easy to understand, but it requires synchronisation, which has a high cost and does not tolerate partial system failure or network partitions well. Hence, the past decade has seen renewed interest in an alternative - Eventual Consistency (EC) - where data is accessed and modified without coordination. Many production systems guarantee only EC, including NoSQL databases, key-value stores and cloud storage systems. EC improves responsiveness and availability, updates propagate more quickly, and enables less coordination-intensive and often less complex protocols.

EC is crucial for scalability but represents an unfamiliar design space. Conflicts and divergence are possible, making them hard to program against, while metadata growth and maintaining system-wide invariants can also become problematic. As more developers interact with and program against distributed storage systems, understanding and managing these trade-offs becomes increasingly important.

Although EC remains poorly understood, there has recently been considerable momentum in the research and development community: the past several years have seen the (re)introduction of useful concepts, such as replicated data types, monotonic programming, causal consistency, red-blue consistency, novel proof systems, etc., designed to allow improve efficiency, programmability, and overall operation of weakly consistent stores.

This workshop aims to investigate the principles and practice of Eventual Consistency. It will bring together theoreticians and practitioners from different horizons: system development, distributed algorithms,

Co-located with:

EuroSys 2014
(http://eurosys2014.vu.nl/)

Sponsored by:

(https://syncfree.lip6.fr/)
SyncFree (https://syncfree.lip6.fr/)

(http://concordant.lip6.fr/)
ConcoRDanT (http://concordant.lip6.fr/)

From Strong to Eventual Consistency

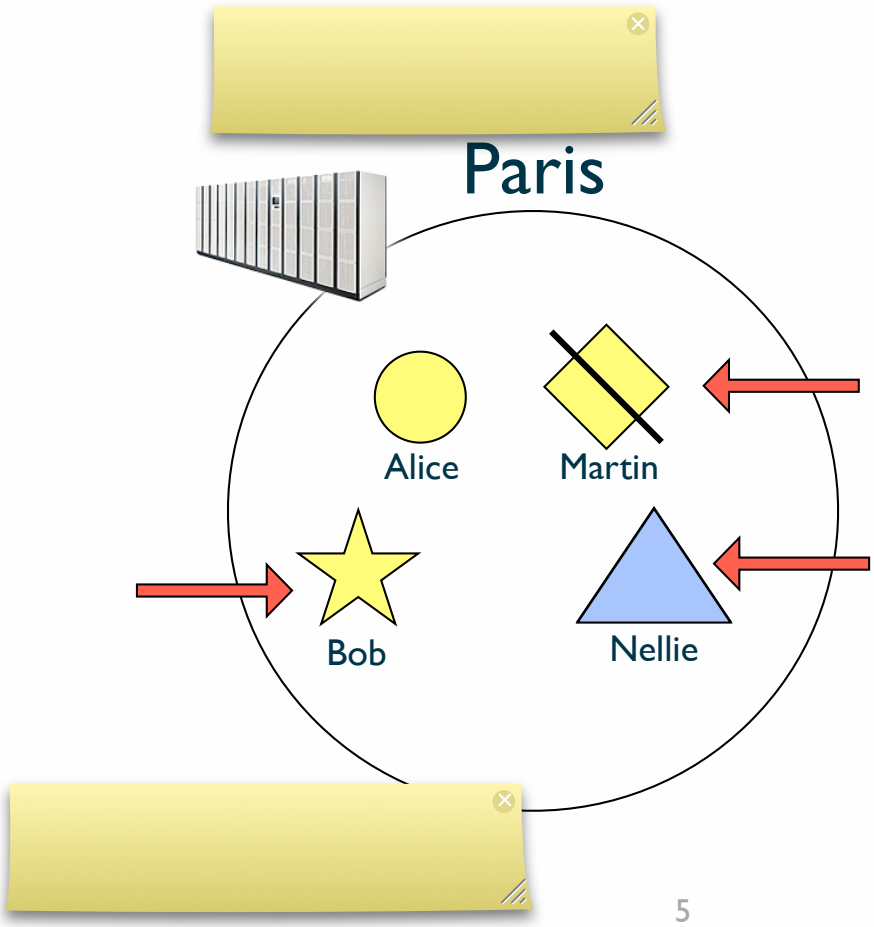
Strong vs. eventual consistency

Pros and cons

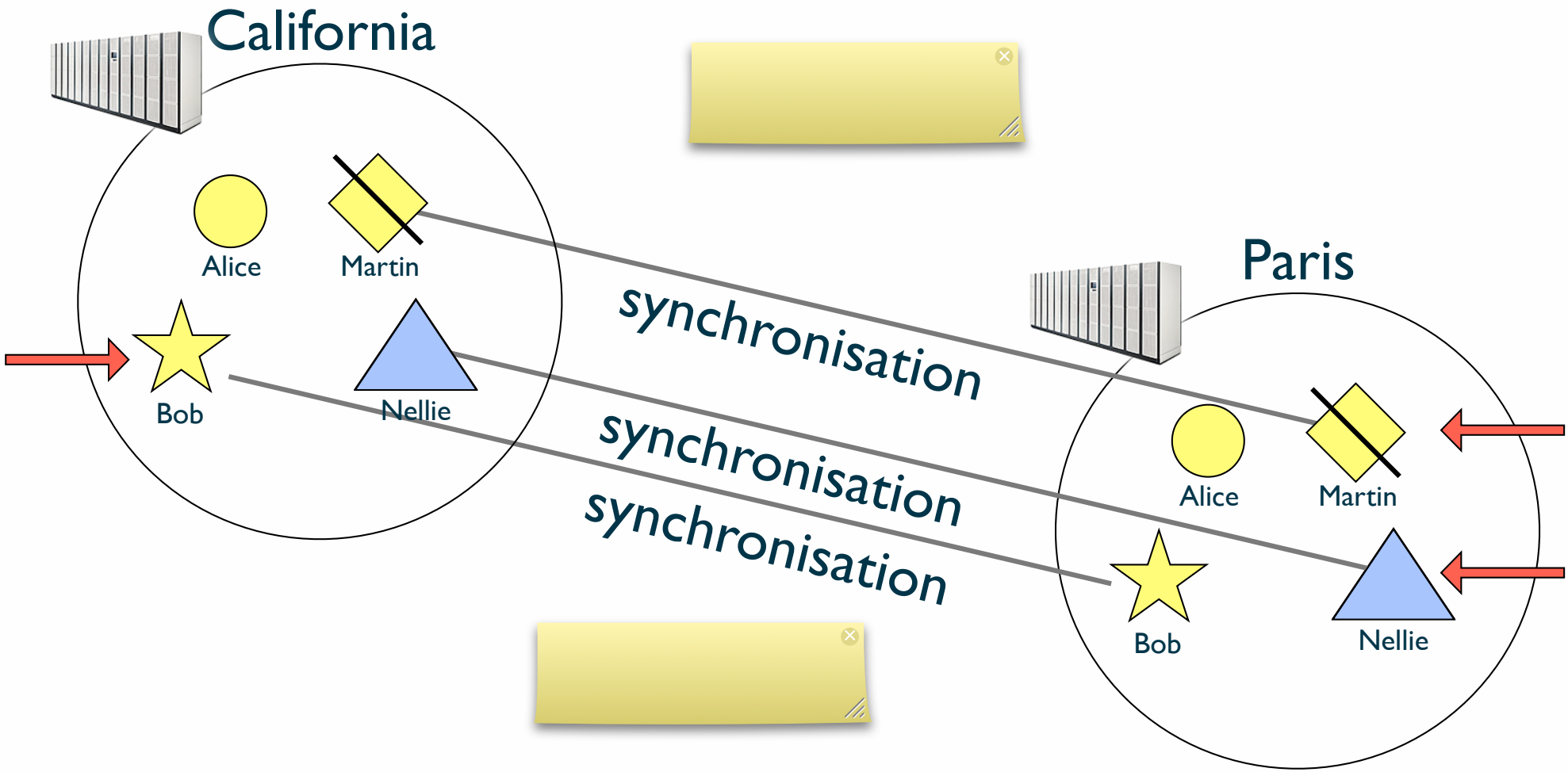
Historical perspective

Examples: Facebook, file systems, Bayou

1 data centre

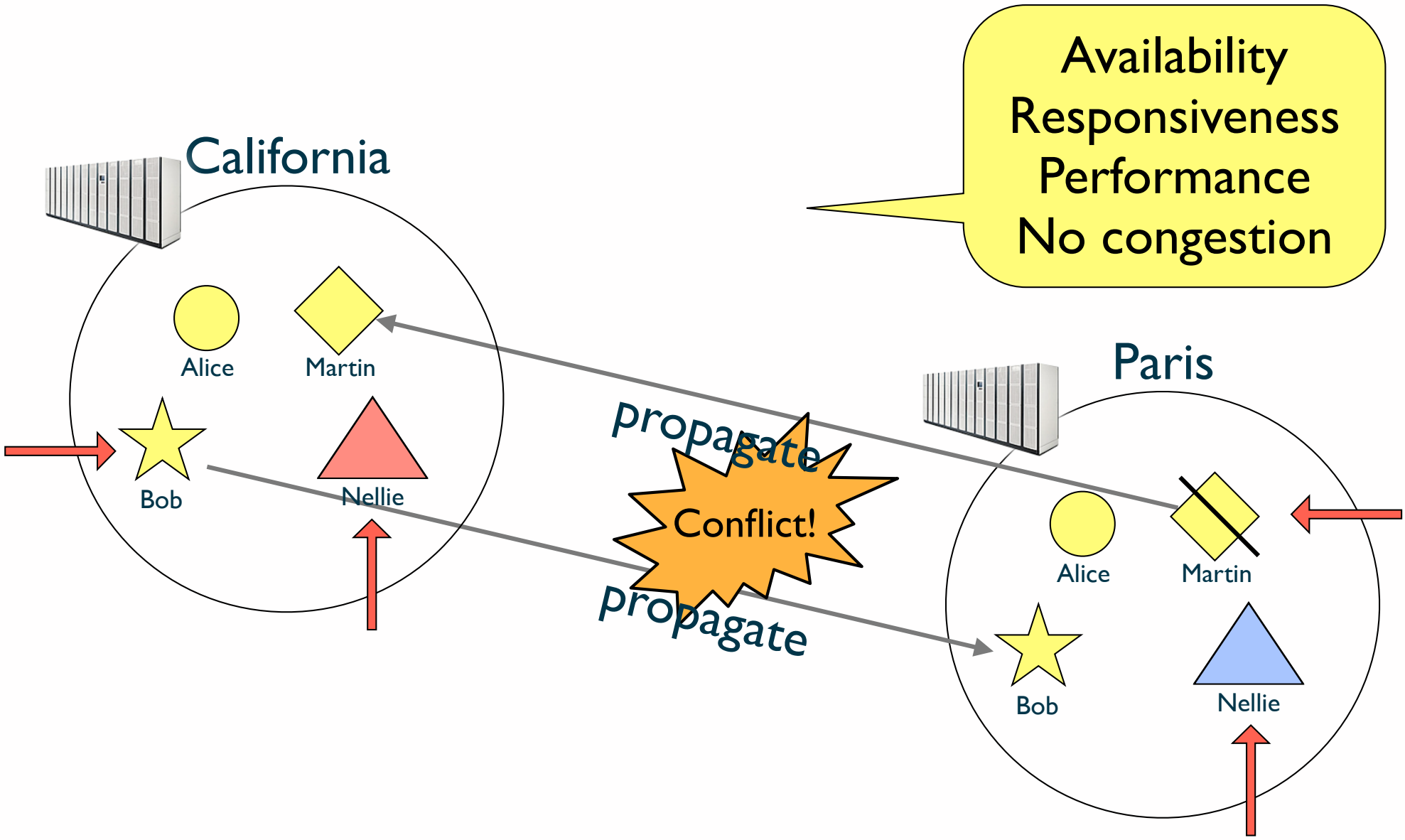


Strongly-consistent replication



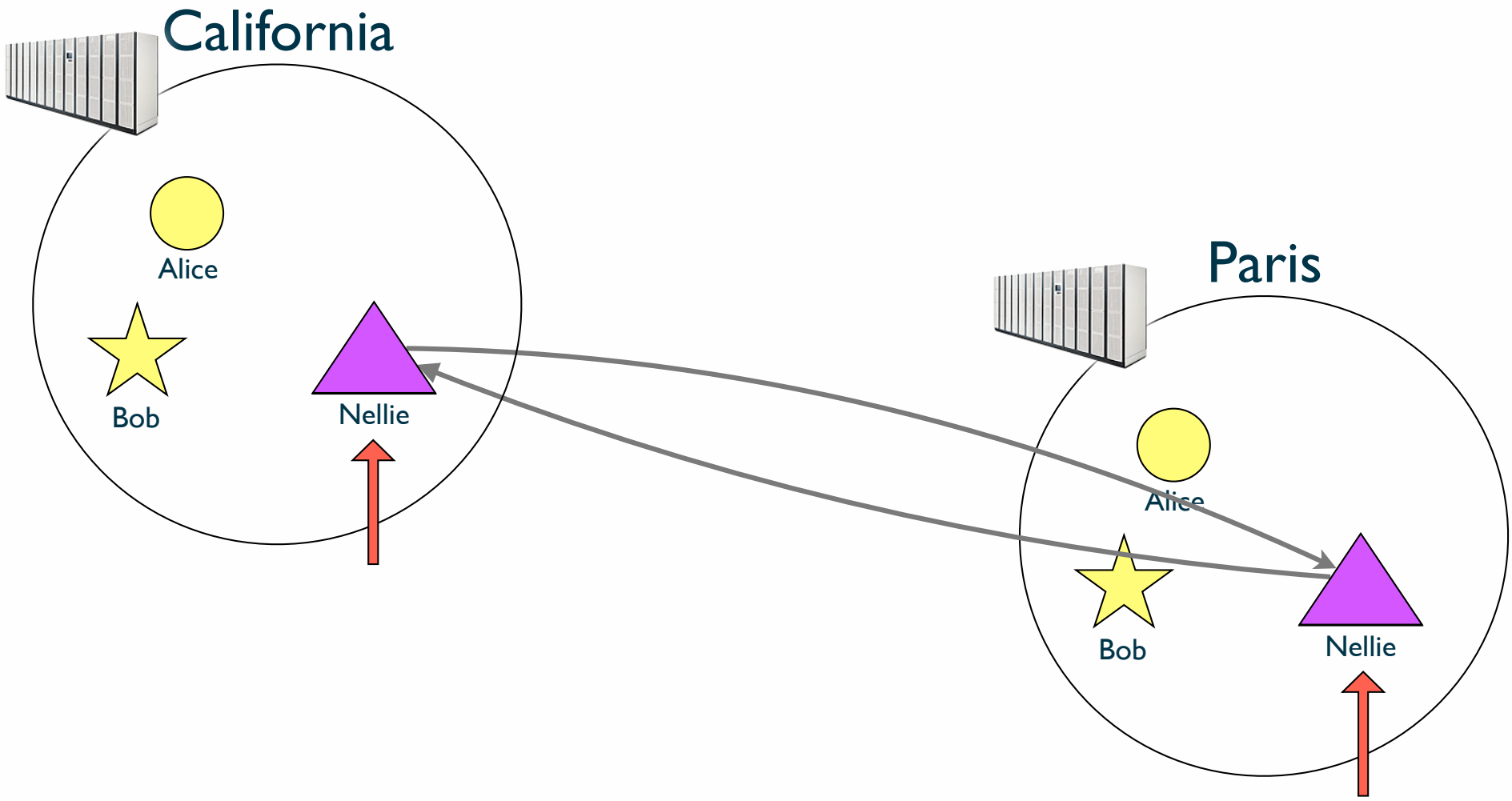
[From strong to eventual consistency: getting it right]

Asynchronous propagation



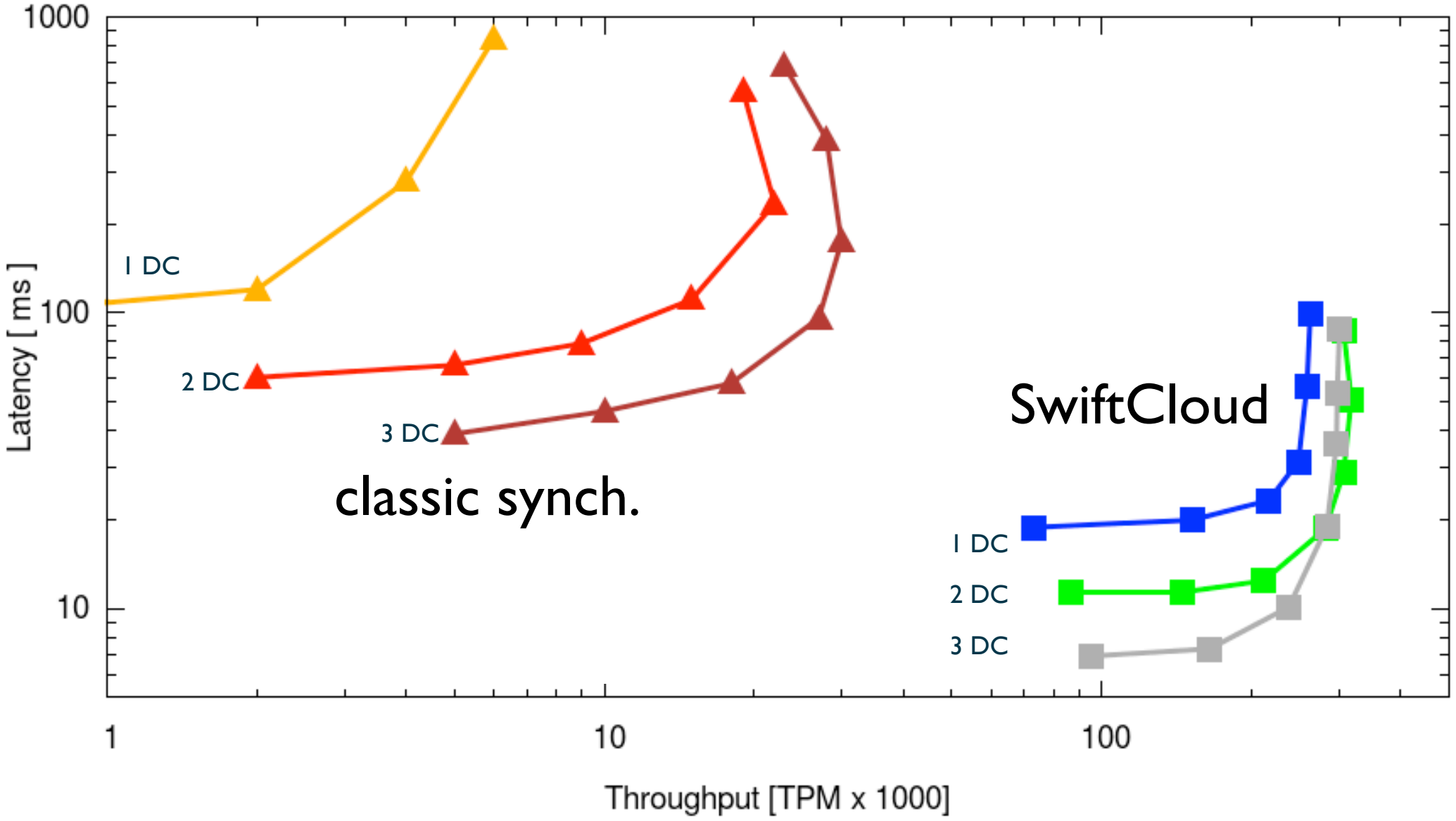
[From strong to eventual consistency: getting it right]

Eventual consistency



[From strong to eventual consistency: getting it right]

Why bother?


1000
100
10
Latency [ms]
1
10
100
Throughput [TPM x 1000]
1 DC
2 DC
3 DC
classic synch.
SwiftCloud
1 DC
2 DC
3 DC

EC Historical perspective (1)

Different backgrounds, requirements, approaches

Slow, error-prone networks

- uunet ⟹ epidemic communication
- DNS: Thomas Write Rule: *LWW*
- Wuu & Bernstein [PODC 1984]

Mobile computing ⟹ opportunistic communication, algorithmic conflict resolution

- File systems: *Coda, Locus/Ficus, resolvers*
- Bayou: *tentative execution & rollback*
- IceCube: *app-independent*

EC Historical perspective (2)

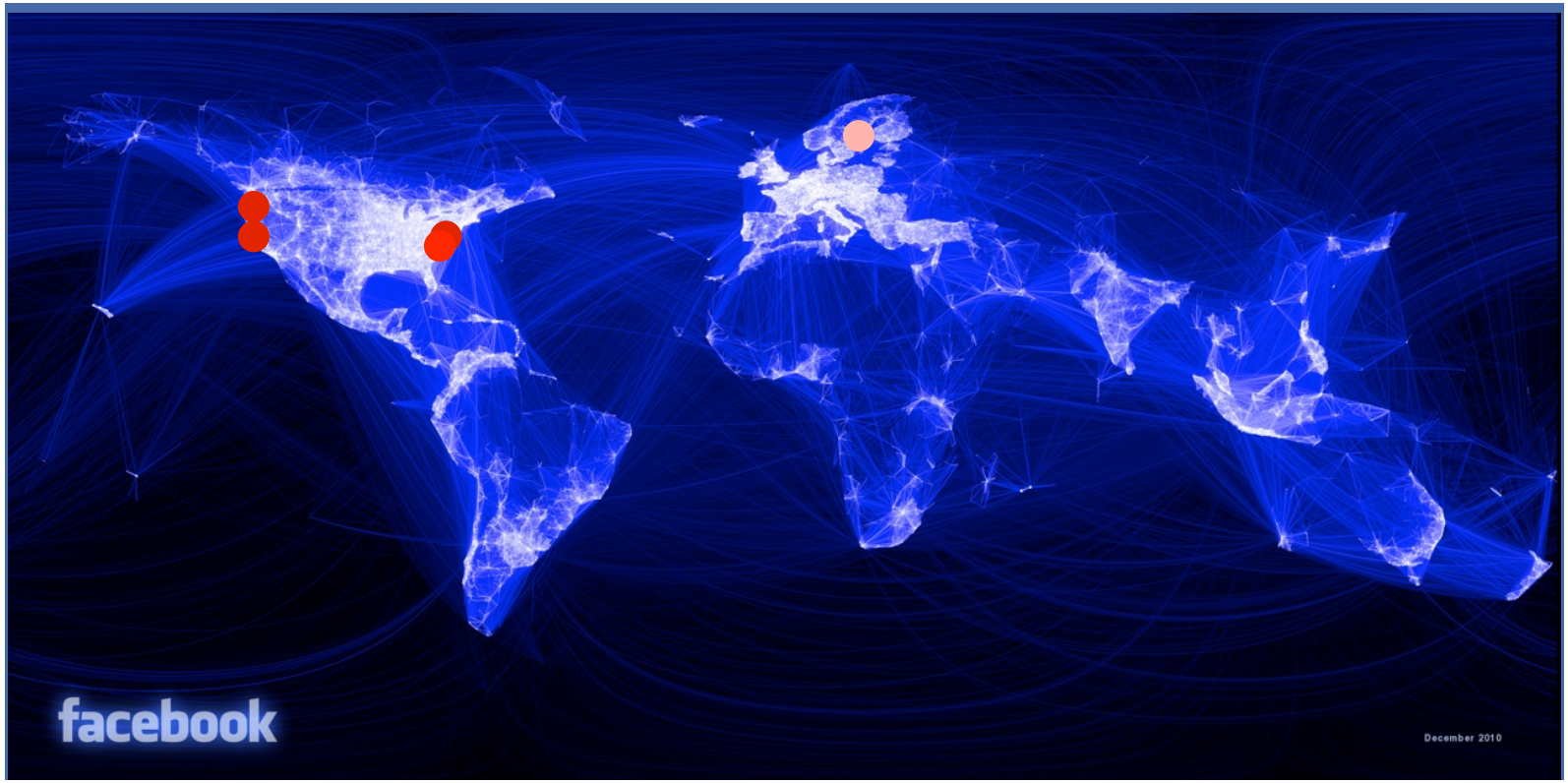
Group editing, source code management

- Offline, textual merge: *CVS, SVN, git*
- Online, operation-based: *Operational Transformation (OT), Google Drive*

P2P and cloud computing: fast, deterministic, program-oriented

- NoSQL, DHTs, key-value stores: *LWW, Dynamo, C-set*
- Causal consistency: *Walter, Cops/Eiger*

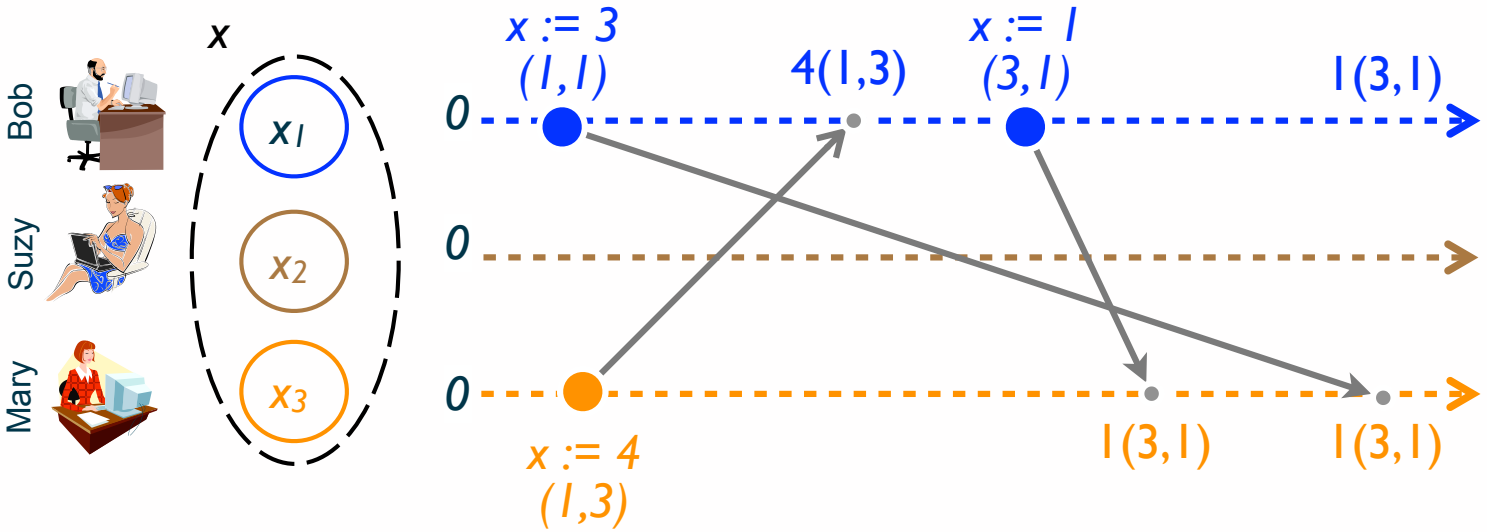
Example: Facebook



- Read from closest server
- Write to California
- Other servers update cache every 15 minutes
- After write: read from CA for 15 minutes

[From strong to eventual consistency: getting it right]

Example: Replicated files



Update: assign, overwrite file

- Transmit file + timestamp *(unique, monotonic* ↗*)*
- Highest timestamp wins

Widely used: file systems, KVSs

[Johnson, Thomas, *The maintenance of duplicate databases*, 1976, RFC 677]

Example: Bayou

Arbitrary database, code

Update = *(dep-check, update-proc, merge-proc)*

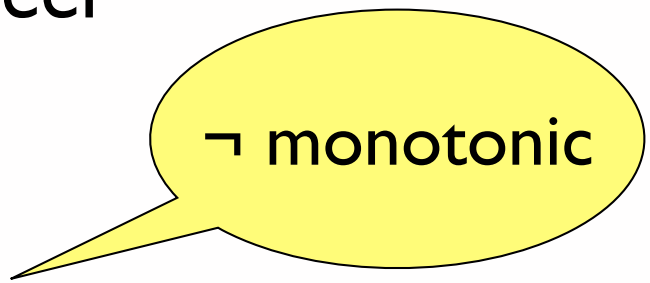
- Execute, append to log

Anti-entropy

- Send *My-Log* \ *Remote-Log* to peer

Receive update

- Order by timestamp
- Possibly: Roll back, re-execute



Convergence: primary site imposes order

- Consensus by dictatorship

Prune logs

Basic Mechanisms

Read and update replica (multi-master)

Transmit later

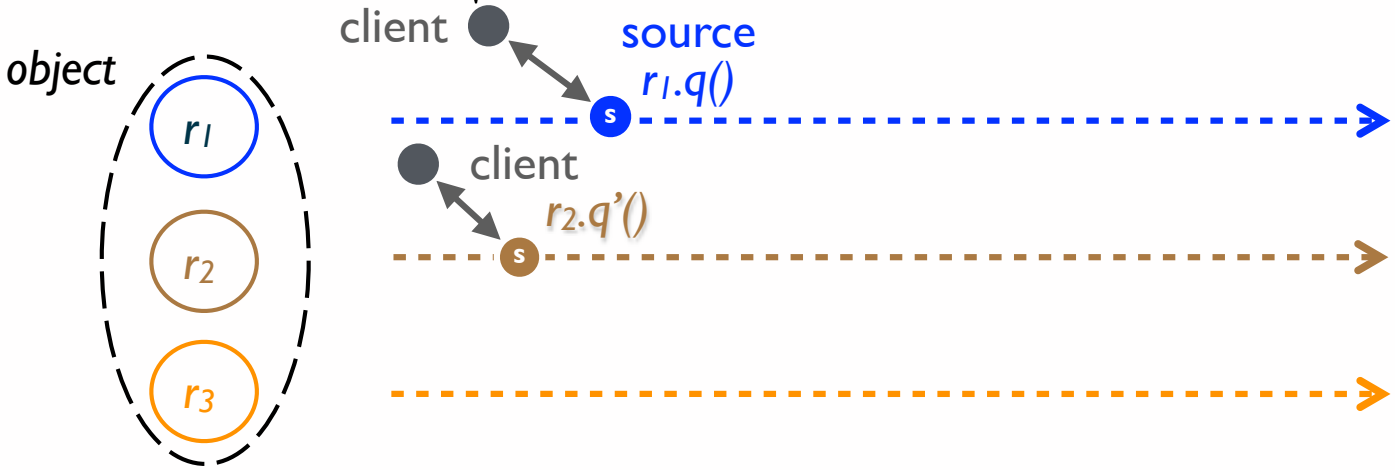
Detect order / concurrency / conflict

Reconcile: converge to a common state

Garbage-collect metadata

Query

multiple replicas:
¬ monotonic reads

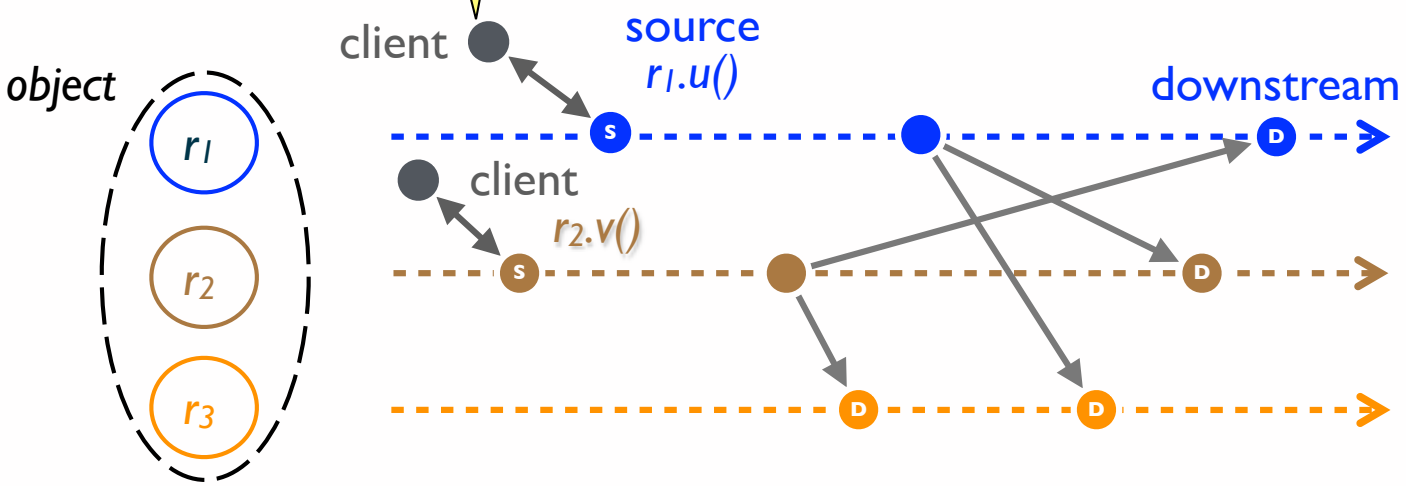


Query local replica

Bayou: read database copy

multiple replicas:
¬ Read My Writes

Update & transmit



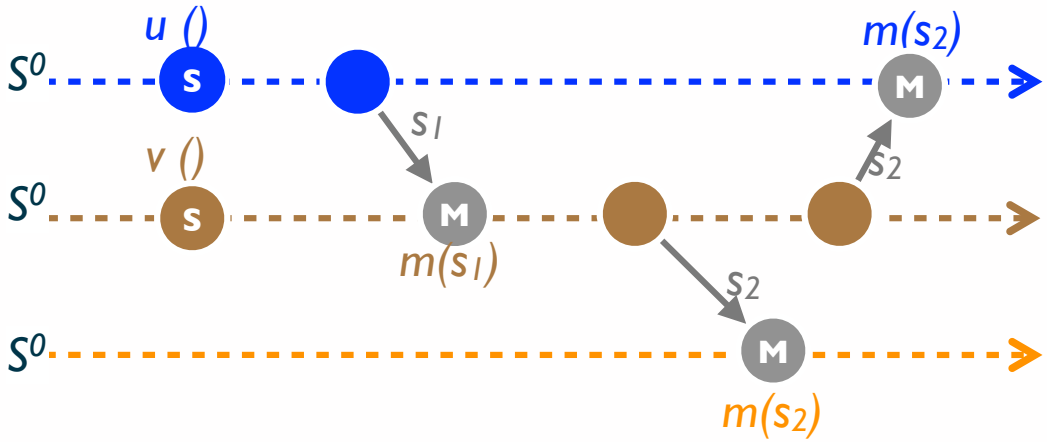
Update *source* replica

Transmit to *downstream* replicas later

- In background
- Flooding, anti-entropy, etc.

Receiver applies update

State-based / data shipping



Merge two (or three) states

- How to ensure convergence?

Delivery:

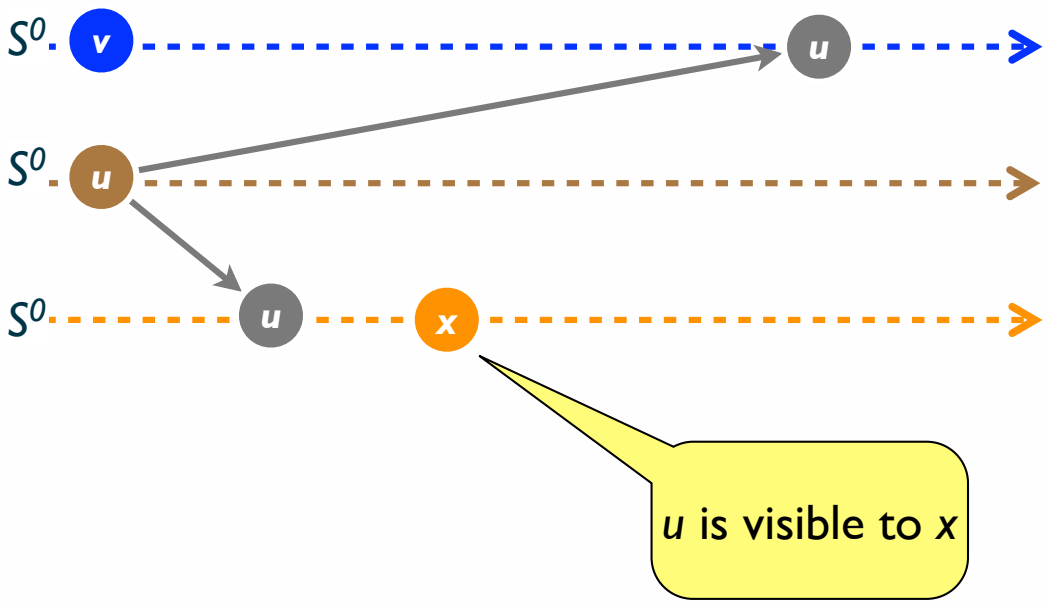
- P2P "epidemic," probabilistic
- Star

File systems: dropbox, SVN

Inefficient for large payload
Convergence?

epidemic ⇒ probabilistic
eventual delivery

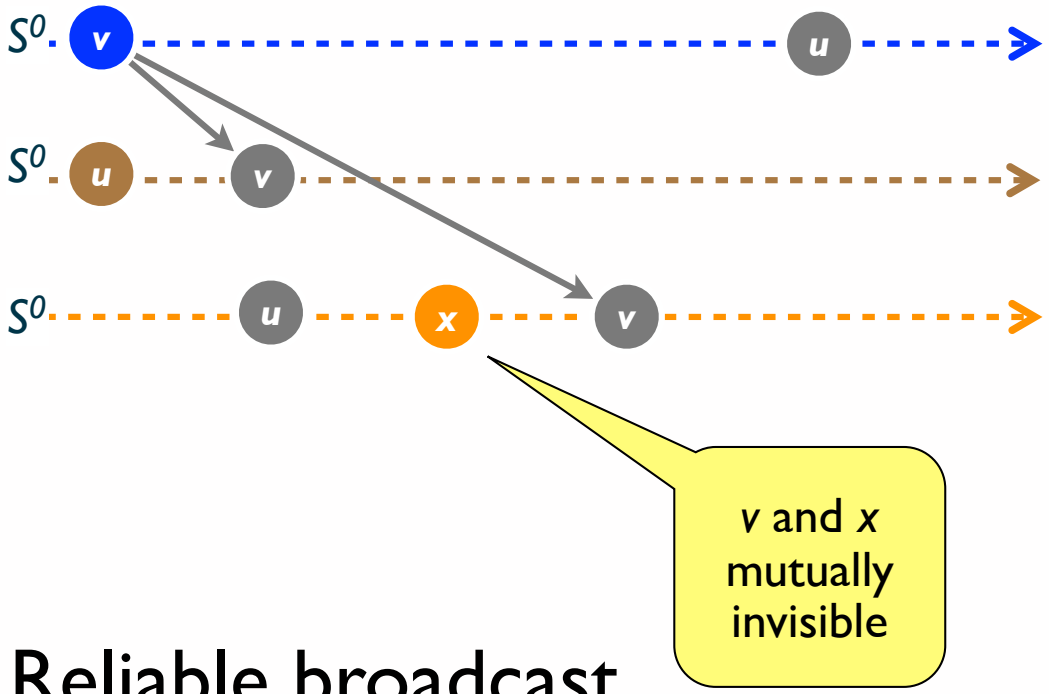
Op-based / function shipping



- Reliable broadcast
- Different execution orders
  - How to ensure convergence?

Bayou, IceCube, OT

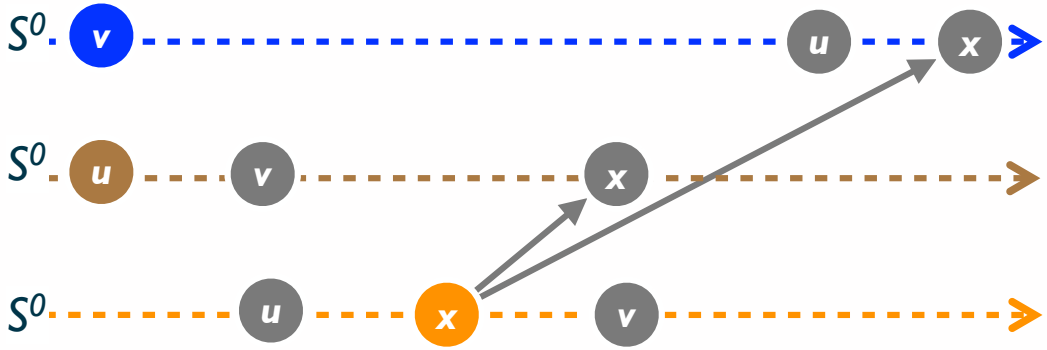
Op-based / function shipping



- Reliable broadcast
- Different execution orders
  - How to ensure convergence?

Bayou, IceCube, OT

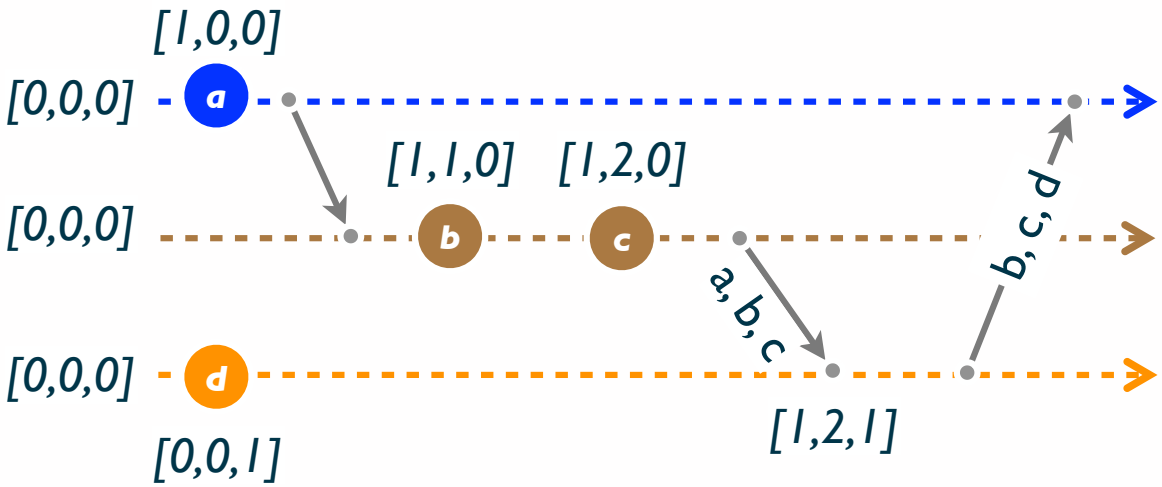
Op-based / function shipping



- Reliable broadcast
- Different execution orders
  - How to ensure convergence?

Bayou, IceCube, OT

Vector clocks & anti-entropy



$V_i[j]$ = #events of j observed by i

$V_i[j] - V_{i'}[j]$ = events of j that i' should transmit to i

$V(a) < V(b)$ = a happened before b

$V(a) \not< V(d) \wedge V(d) \not< V(a)$ = a concurrent with b

[From strong to eventual consistency: getting it right]

Detect concurrent updates

Two updates:

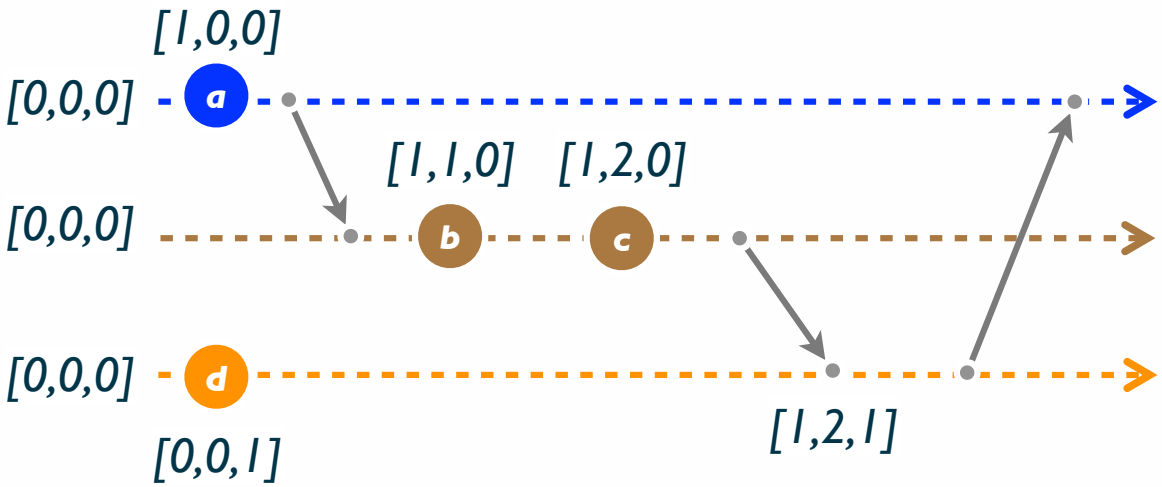
- If $u < v$, then v
- If $(u \not< v \wedge v \not< u)$, then *????*

Detect concurrency

- Explicit history/dependence (*Isis, Cops/Eiger)*
- Vector clocks *(Isis 2, SwiftCloud)*
- Programmed check *(LWW, Bayou, Sinfonia)*

Requires meta-data

Vector clocks & concurrency



$V_i[j]$ = #events of j observed by i

$V_i[j] - V_{i'}[j]$ = events of j that i' should transmit to i

$V(a) < V(b)$ = a happened before b

$V(a) \not< V(d) \wedge V(d) \not< V(a)$ = a concurrent with b

Resolve concurrent updates

Resolve:

- No conflict
- Dynamic total order *(Bayou, IceCube)*
  - Consensus!
- Static total order: arbitration *(LWW)*
- Resolver program *(SVN, Bayou)*
- Ask user

Convergence?

- All replicas make equivalent

Garbage-collect metadata

At least-once delivery:

- An update *u* must be available for forwarding until received by all replicas

Wuu & Bernstein 1984

- Receiver *i* sends *ack(u,i)* to all other replicas
- *j* received *ack(u,i)* from $\forall i$: discards *u*
- Not live if partitioned

Bayou

- Truncate log arbitrarily
- If necessary, recover with full-state transfer
- May lose updates

Distributed resolution

Bayou/IceCube conflict resolution:

- Convergence $\Longrightarrow$ unique $\Longrightarrow$ consensus
- Off critical path: availability OK
- Roll back: expensive, confusing

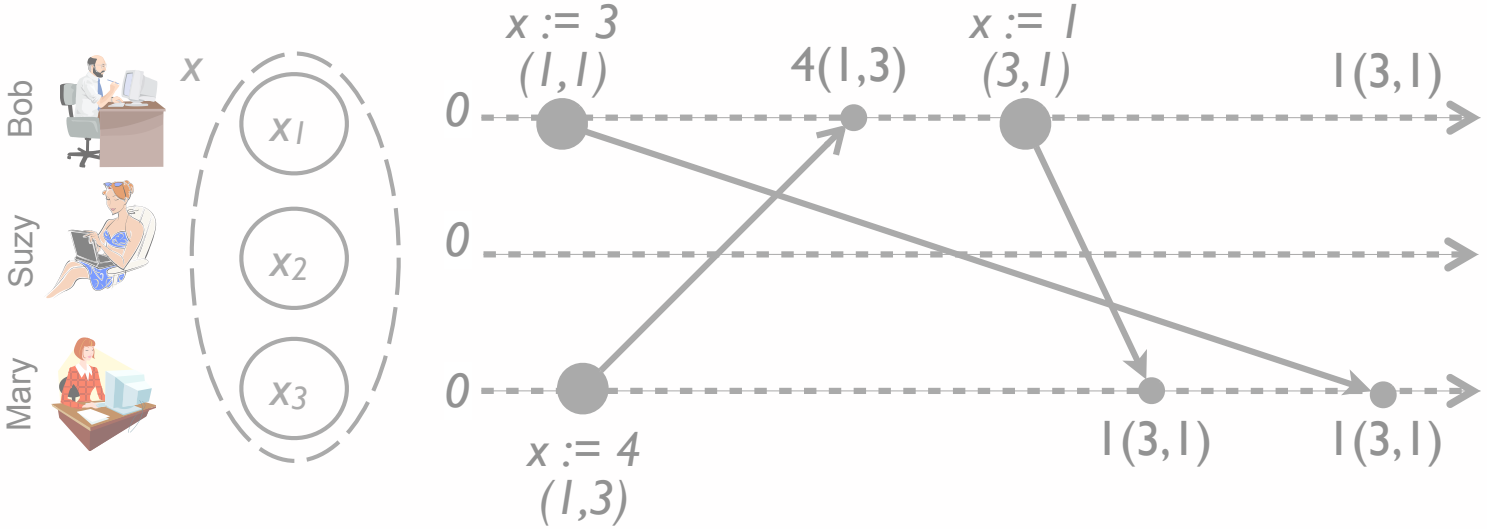
Alternative: decentralised resolution

- No synchronisation
- Convergence conditions:
  - Deterministic
  - Dependent on delivery set
  - Not on delivery order, local info

[From strong to eventual consistency: getting it right]

Decentralised resolution:

Highest timest. wins (LWW)



Update: assign, overwrite file

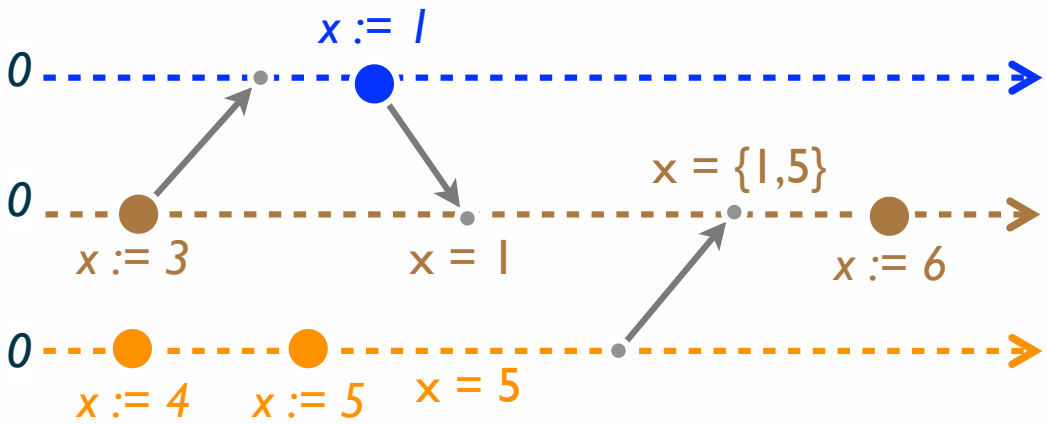
- Transmit file + timestamp *(unique, monotonic ↗)*
- Highest timestamp wins

Widely used: file systems, KVSs

[Johnson, Thomas, *The maintenance of duplicate databases*, 1976, RFC 677]

Decentralised resolution:

Multi-Value Register



Not concurrent: usual register semantics

Concurrent assignments: return *set* of values

Decentralised resolution:

2P-Set

Representation:

- *A*: set of added items
- *T*: set of removed items (tombstones)

Operations:

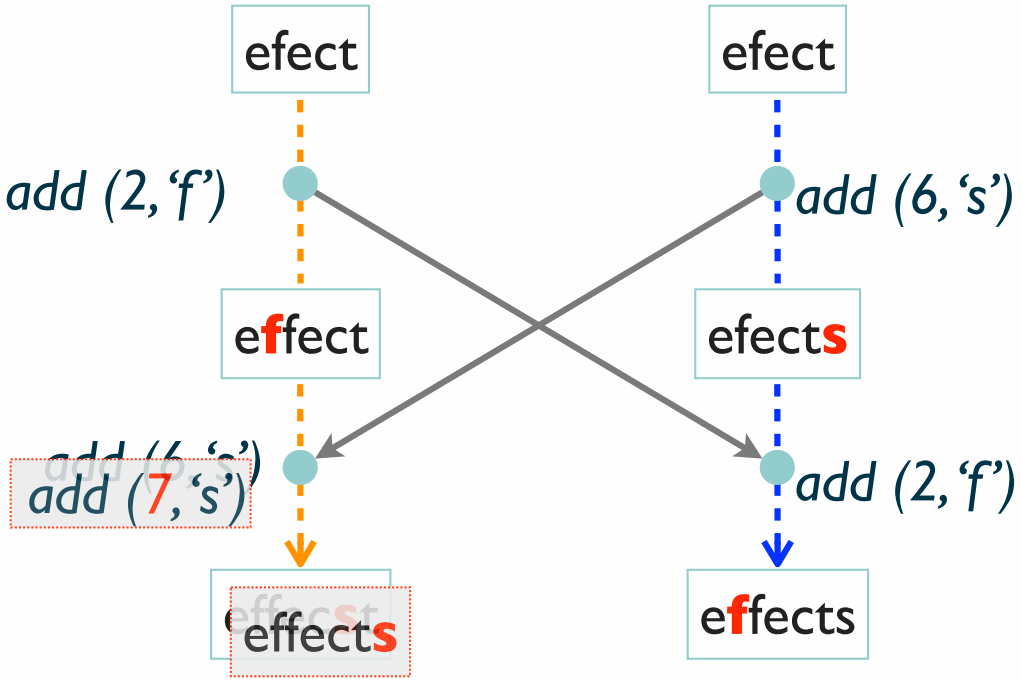
- *add (e)* $A := A \cup \{ e \}$
- *rmv (e)* $T := T \cup \{ e \}$
- *lookup (e)* $e \in A \setminus T$

Resolving concurrent updates:

- $A := A_1 \cup A_2$
- $T := T_1 \cup T_2$

Decentralised resolution:

Operational Transformation



$$\frac{add\ (6, 's')}{add\ (left, 1\,char)} = add\ (\mathbf{7}, 's')$$

$$\frac{add\ (6, 's')}{rm\ (left, 1\,char)} = add\ (\mathbf{5}, 's')$$

Ensure convergence in any order

Transform received operations according to previous concurrent updates

OT correctness conditions

TP1: $f \bullet \frac{g}{f} \equiv g \bullet \frac{f}{g}$

TP2: $\frac{h}{f \bullet \frac{g}{f}} = \frac{h}{f \bullet \frac{g}{f}}$

Star topology: TP1

P2P topology: TP1 + TP2

TP1 ≡ commutativity

[From strong to eventual consistency: getting it right]

Basic Mechanisms

Read and update replica (multi-master)

Transmit later

Detect order / concurrency / conflict

Reconcile: converge to a common state

Garbage-collect metadata

Formal definitions

System model

Define EC

Partial Order model

Stronger Guarantees

System model

Assume some shared objects + specification

Objects support query and update operations.
Operations terminate.

Objects are replicated at a number of processes.

A client may update a replica:

- Update invocation returns
- Update is sent to other replicas
- Receiving replica applies the update
- Subject to correctness conditions hereafter

Eventual Consistency?

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

Captures intuition, but

- Informal
- Assumes update = overwrite
- What happens if updates don't stop?

[From strong to eventual consistency: getting it right]

Eventual Consistency

- Every update eventually reaches every replica at least once
- An update has effect on a replica at most once
- At all times, the state of a replica satisfies the objects' specifications
- Two replicas that received the same set of updates eventually reach the same state

Transactional: replace "update" with "transaction comprising one or more updates".

Asynch ∩ stronger guarantees

Session guarantees
- Read Your Writes
- Monotonic Reads, Monotonic Writes
- Writes Follow Reads

Causal consistency: "If *v* is visible, and *u* precedes *v*, then *u* is visible"
- Implies session guarantees
- Available ⟹ Strongest possible [MAD 2011]

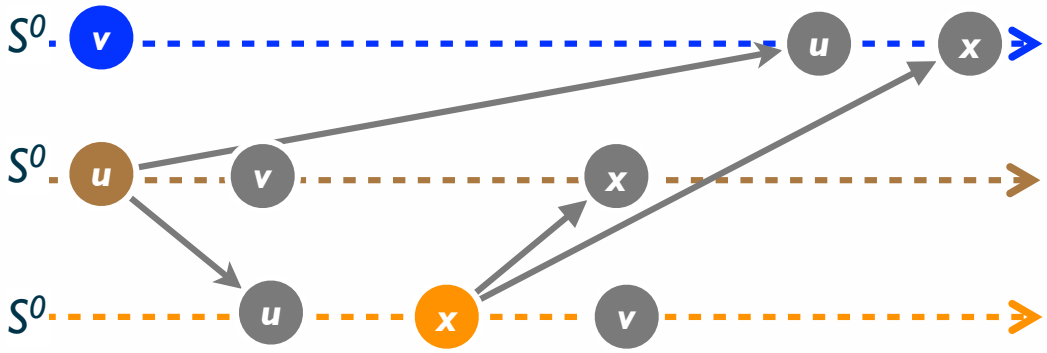
Mergeable transactions

Fault tolerance

How to implement at scale?

[From strong to eventual consistency: getting it right]

Causal consistency

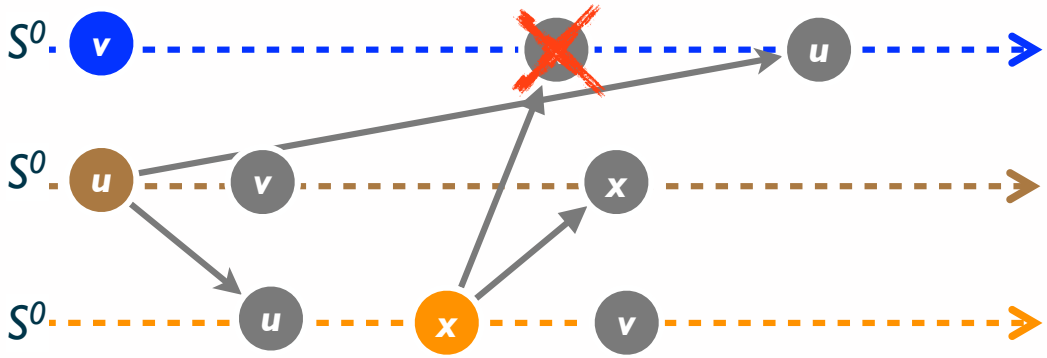


x observed effects of *u* at source
$\Longrightarrow$ *x* to be delivered after *u*

Challenge: implement at scale

- Meta-data explosion

Causal consistency



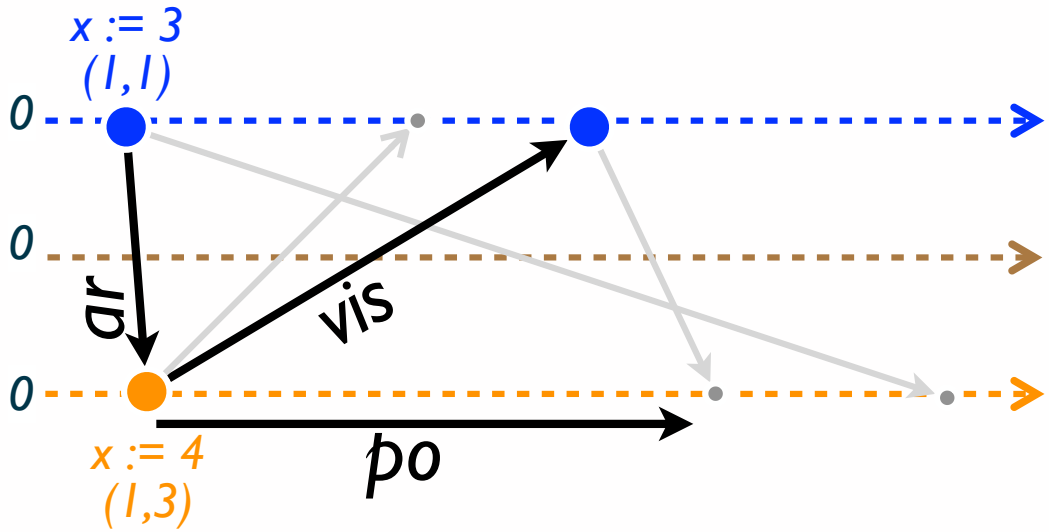
x observed effects of *u* at source
$\Longrightarrow$ *x* to be delivered after *u*

Challenge: implement at scale

- Meta-data explosion

Partial-Order model

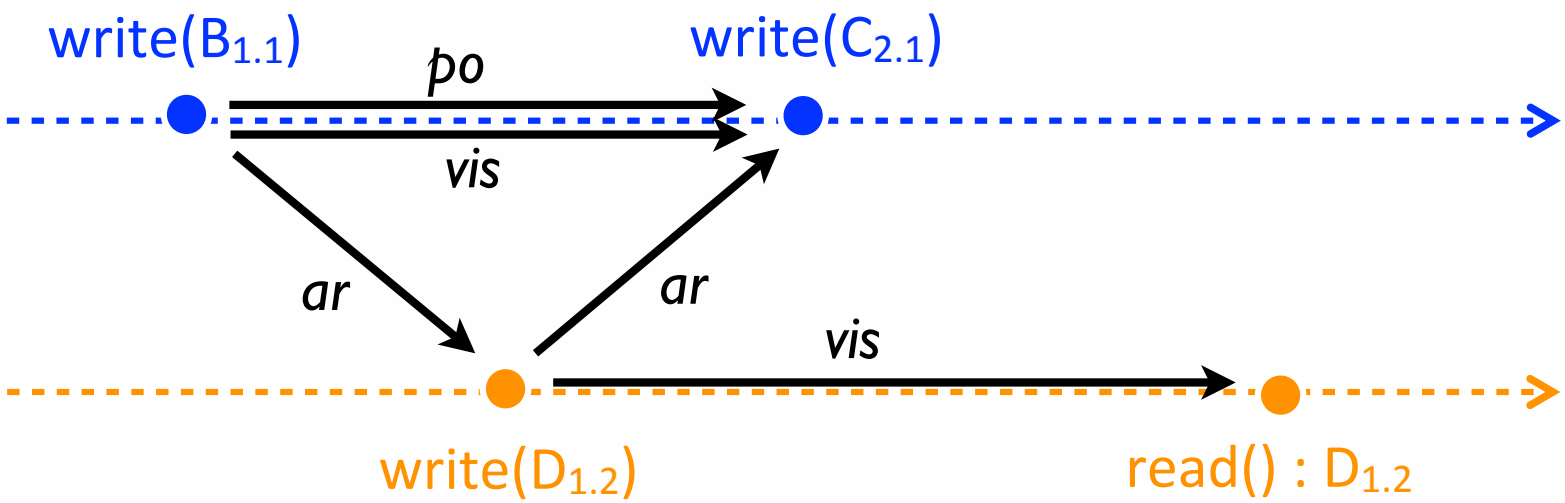
[Burckhardt, Gotsman, Yang, "Understanding Eventual Consistency", MSR-TR-2013-39]



Effect of *op* = *F(op, events, po, vis, ar)*

- vis^{-1} : Recent updates
- *ar* : Arbitrate concurrent updates
- *F* : Deterministic function

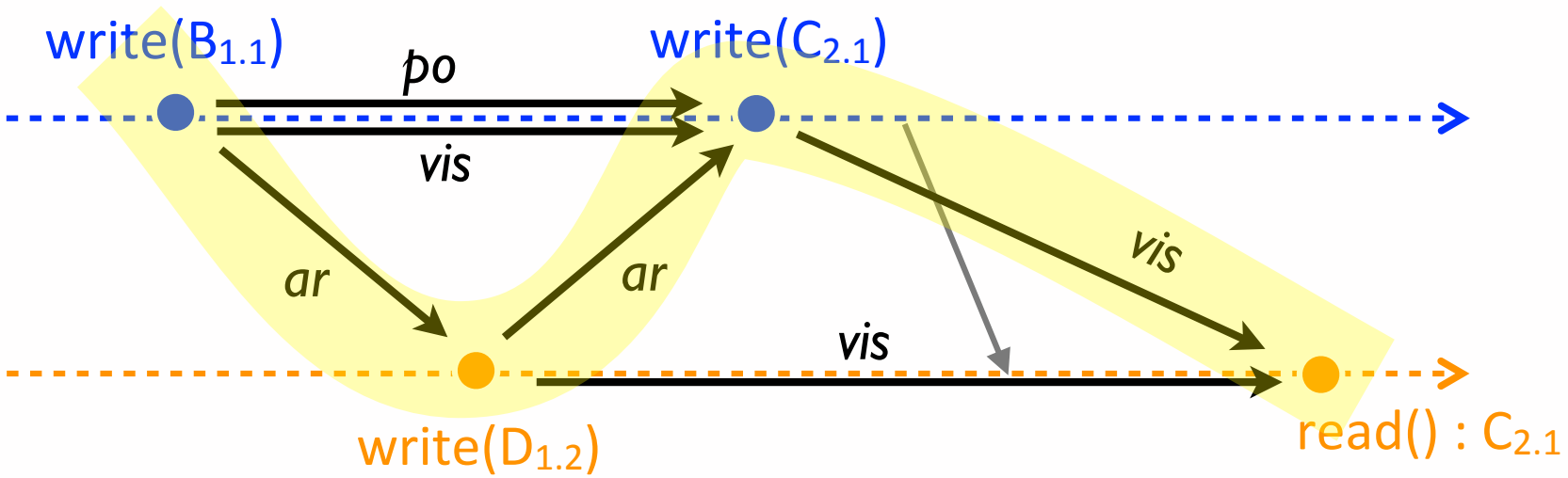
P-O model: LWW



Result of an action?

Only visible actions matter

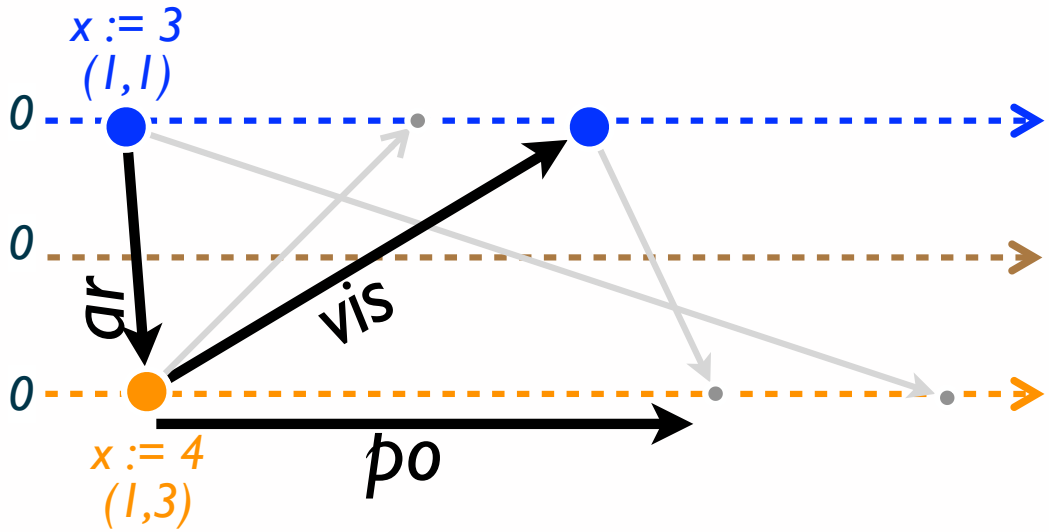
P-O model: LWW



Result of an action?

Only visible actions matter

P-O model: Guarantees



$hb \stackrel{\text{def}}{=} (po \cup vis)^+$

Read Your Writes:	$po \subseteq vis$
Monotonic Reads:	$(vis; po) \subseteq vis$
Causal:	$hb \subseteq vis \wedge hb \subseteq ar$
Atomic + Isolated:	???

$\Longrightarrow$ RYW, MR, ...

Formal definitions

System model

Define EC

Partial Order model

Stronger Guarantees

CRDTs: Conflict-Free Replicated Data Types

Definition

Convergence conditions: State- vs. Op.-based

Concurrency semantics

Designing CRDTs

Beyond CRDTs

Conflict-Free Replicated Data Type: key concepts

Replicated at multiple nodes

Conflict-free: update without coordination, decentralised resolution

Data type: Encapsulation, well-defined interface

Eventual convergence by construction

Library of CRDTs

Register
- Last-Writer Wins
- Multi-Value

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

Map

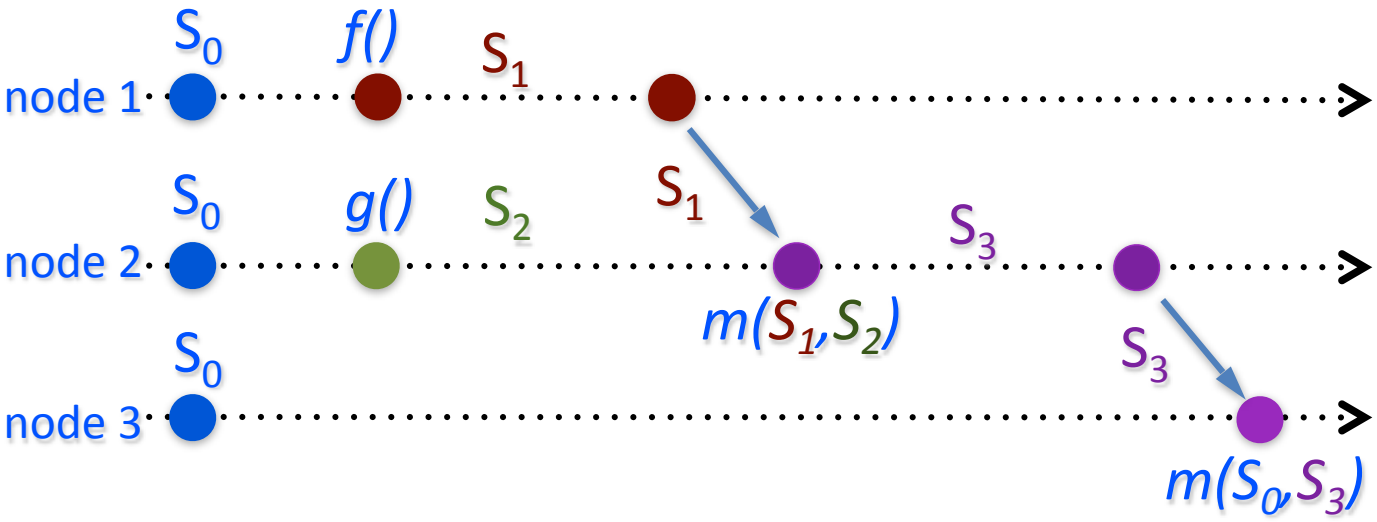
File-system tree

Counter
- Unlimited
- Non-negative

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence

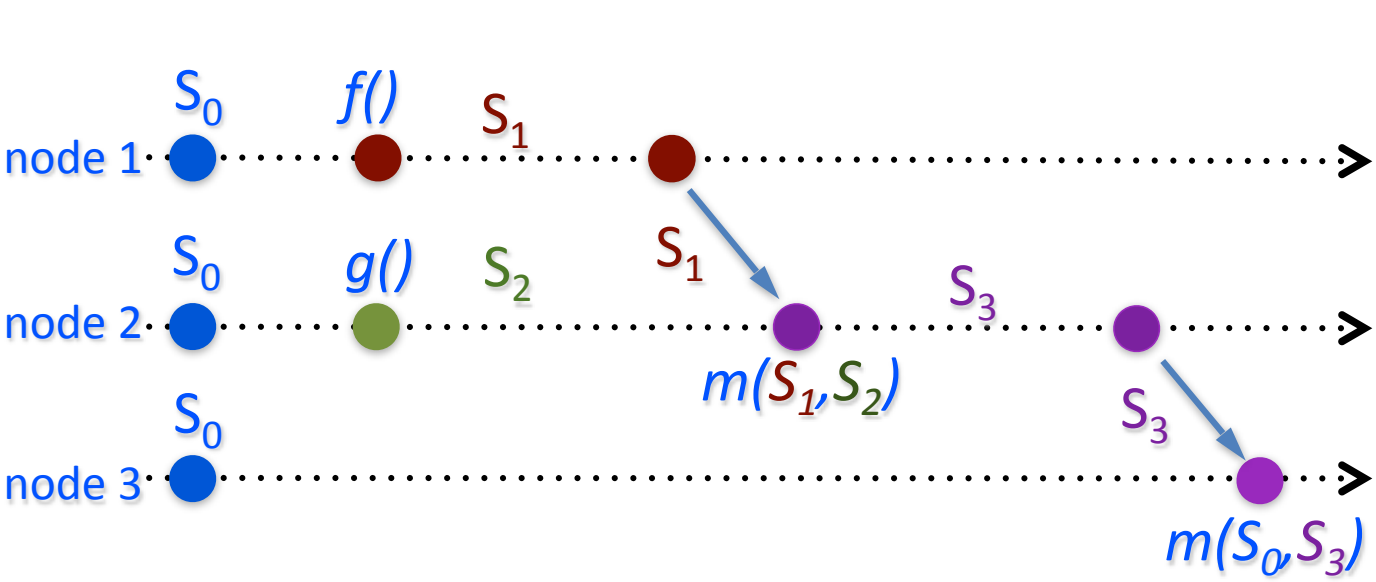
State-based CRDT

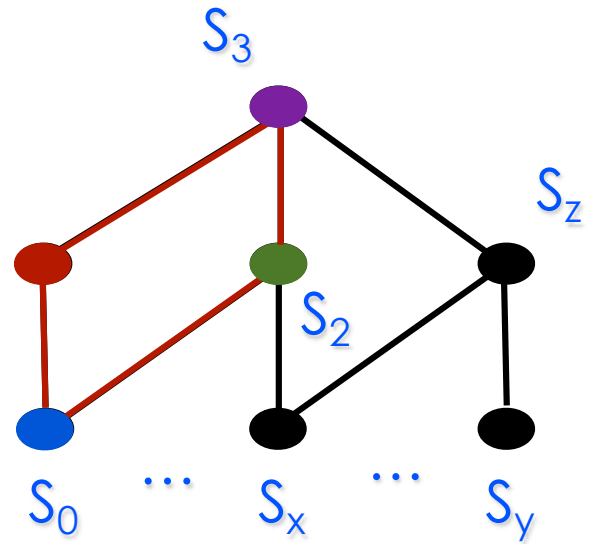


Convergence?

[From strong to eventual consistency: getting it right]

State-based CRDT





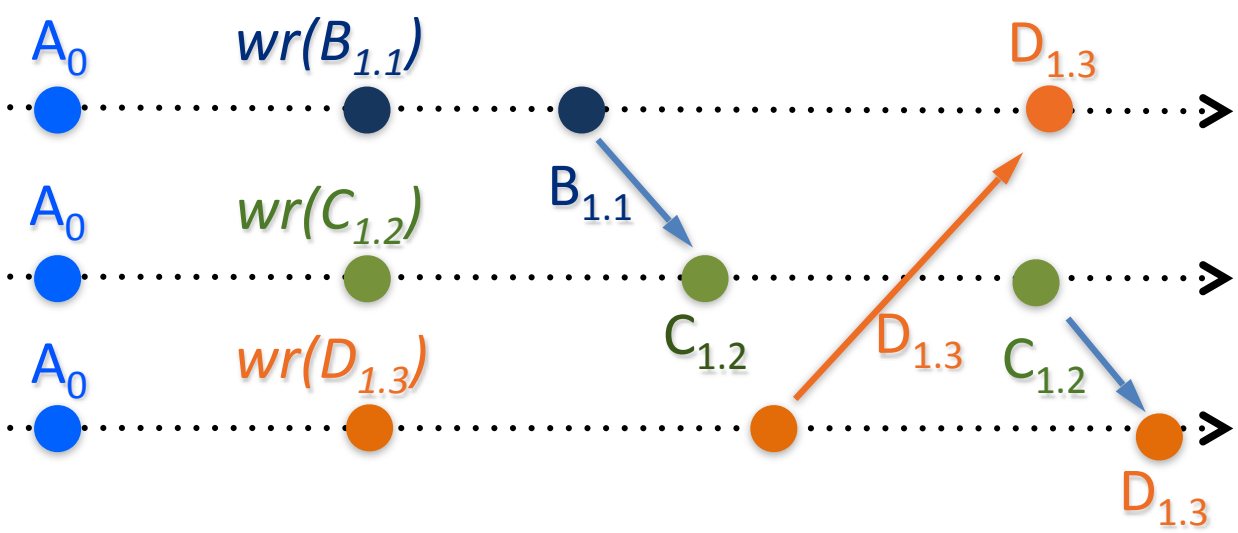
Safety requirements:

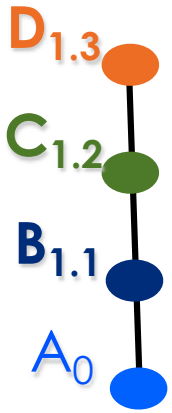
- Don't go backwards ⟹ partial order, monotonic
- Apply each update once: *m* idempotent
- Merge in any order: *m* commutative
- Merge contains several updates: *m* associative

Monotonic semi-lattice + merge = Least Upper Bound

[From strong to eventual consistency: getting it right]

Example: highest timestamp wins register





State:

V_{ts} : value *V*, timestamp *ts*

Operations:

write: overwrite *v*, increment *ts*

merge: *value* with highest *ts* wins

$s \leq s' \stackrel{\text{def}}{=} s.ts \leq s'.ts$

$merge\ (s,s') = s.t < s'.t\ ?\ s' : s$

CRDTs: Conflict-Free Replicated Data Types

Definition

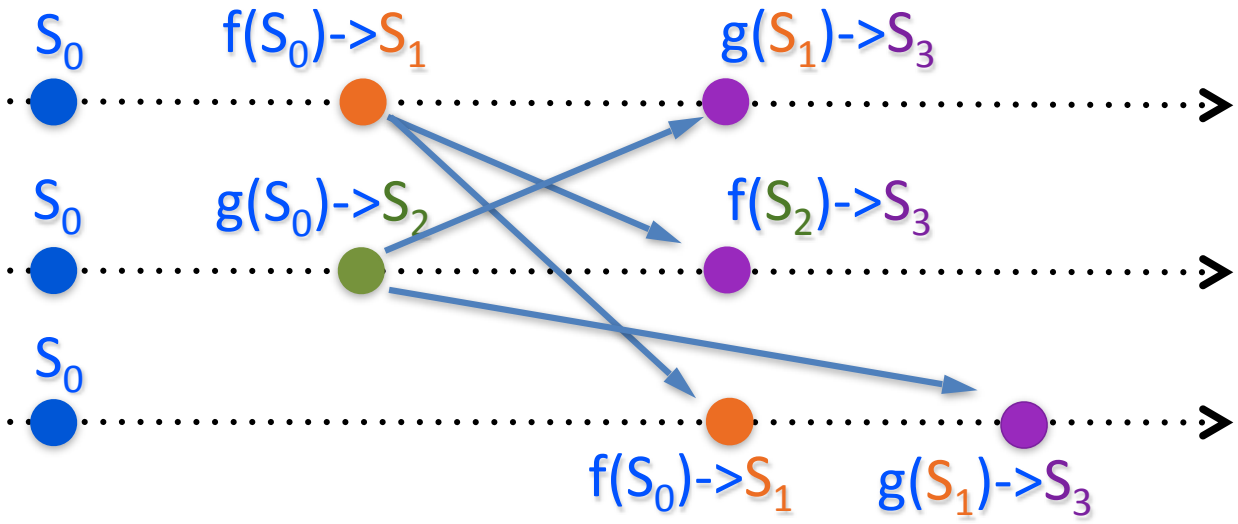
Convergence conditions: State- vs. Op.-based

Concurrency semantics

Designing CRDTs

Beyond CRDTs

Operation-based CRDT



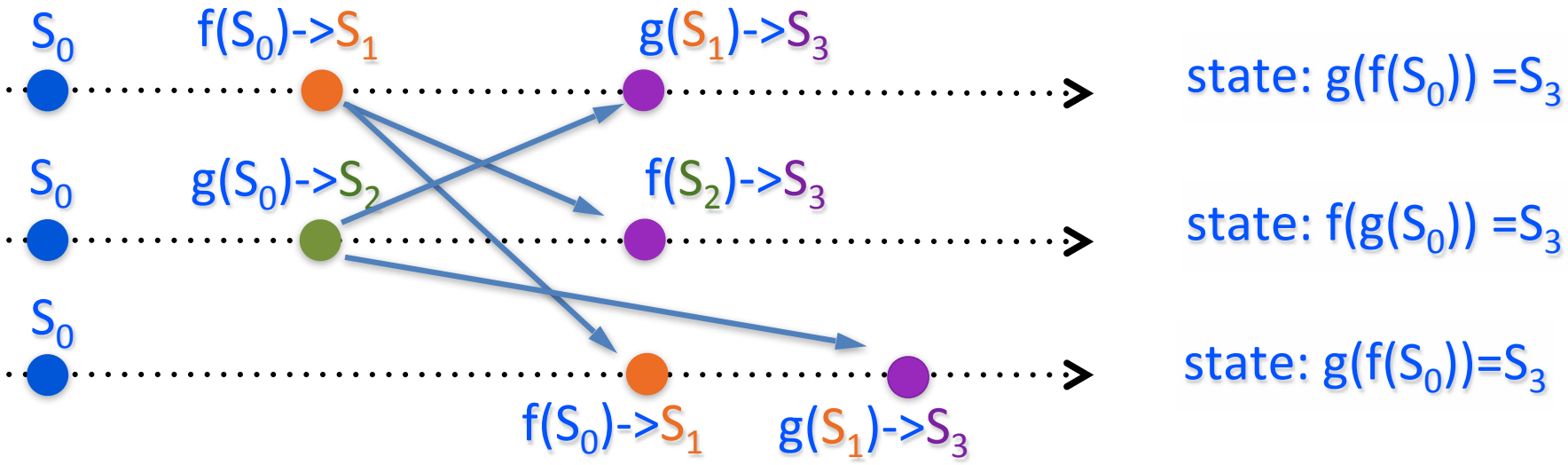
Operation/function shipping

Reliable broadcast

Convergence?

[From strong to eventual consistency: getting it right]

Operation-based CRDT



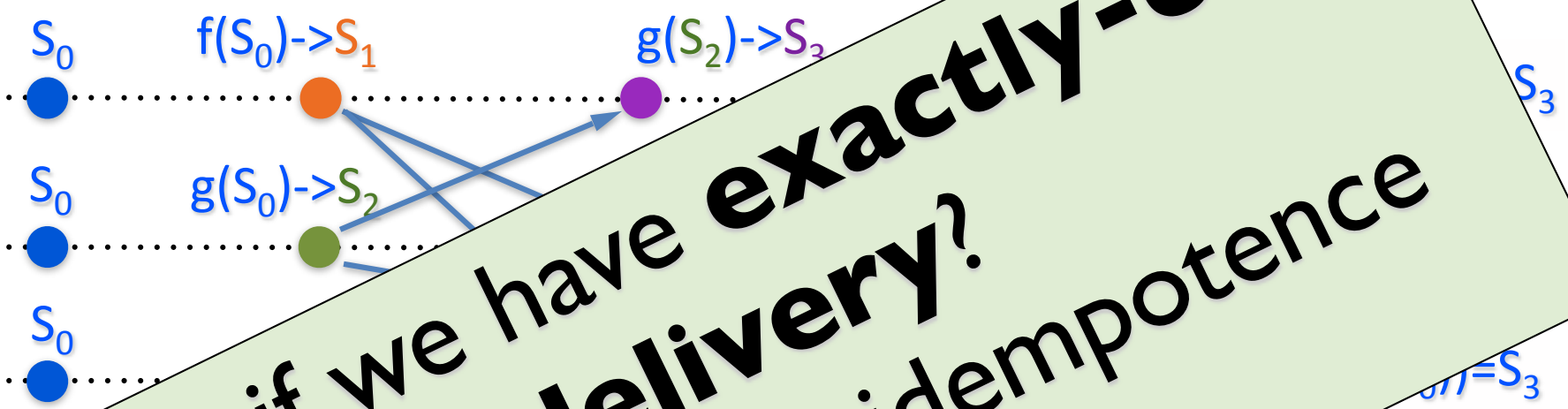
Convergence: sufficient condition

Reliable delivery

Operations must commute and be idempotent

[From strong to eventual consistency: getting it right]

Operation-based CRDT



What if we have **exactly-once delivery**?

We could drop idempotence requirement.

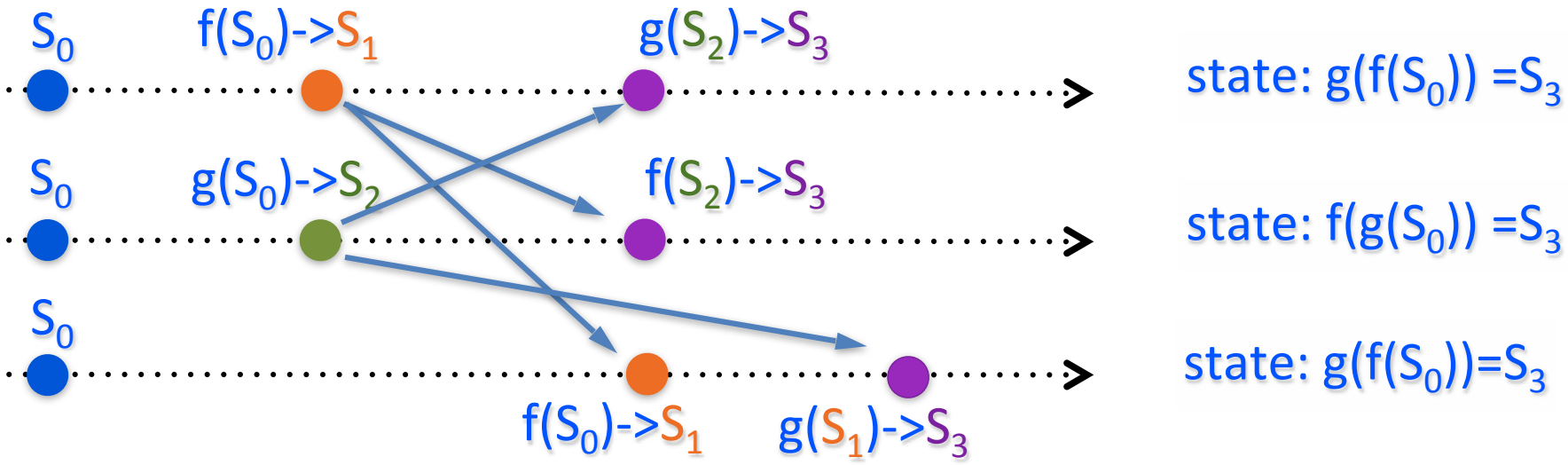
…ient condition

…iable delivery

… must commute and be idempotent

[From strong to eventual consistency: getting it right]

Operation-based CRDT



Convergence: sufficient condition

Reliable exactly-once delivery

Operations must commute ~~and be idempotent~~

[From strong to eventual consistency: getting it right]

Operation-based CRDT

What if we have **causal delivery**?

Only concurrent operations need to commute.

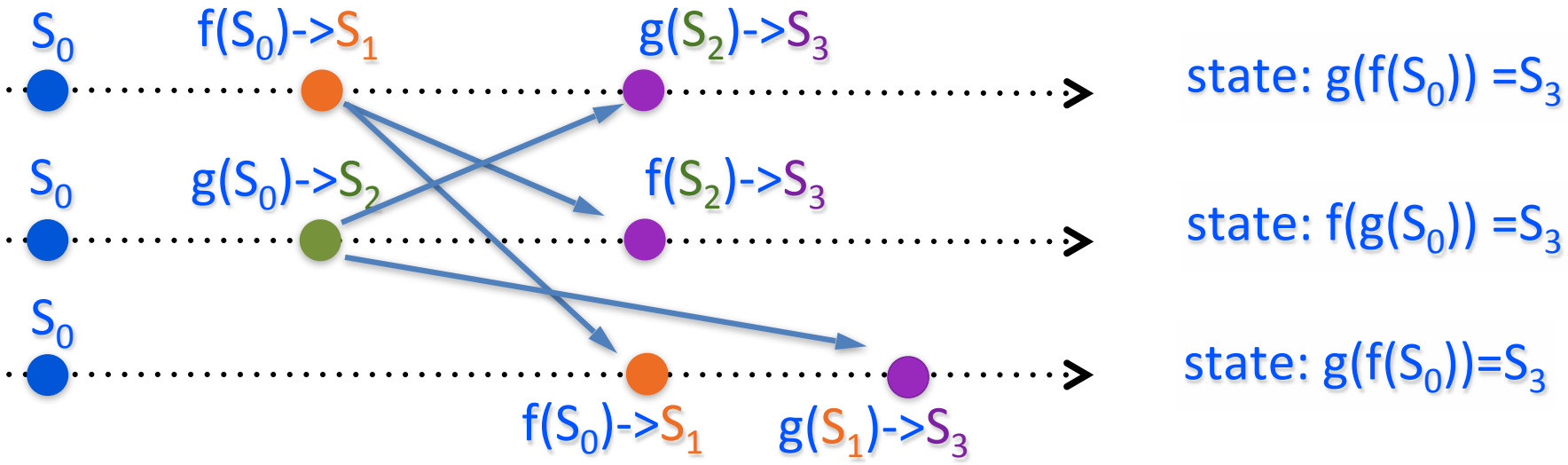
ce: sufficient condition

Reliable exactly-once delivery

perations must commute and ~~be idempotent~~

[From strong to eventual consistency: getting it right]

Operation-based CRDT



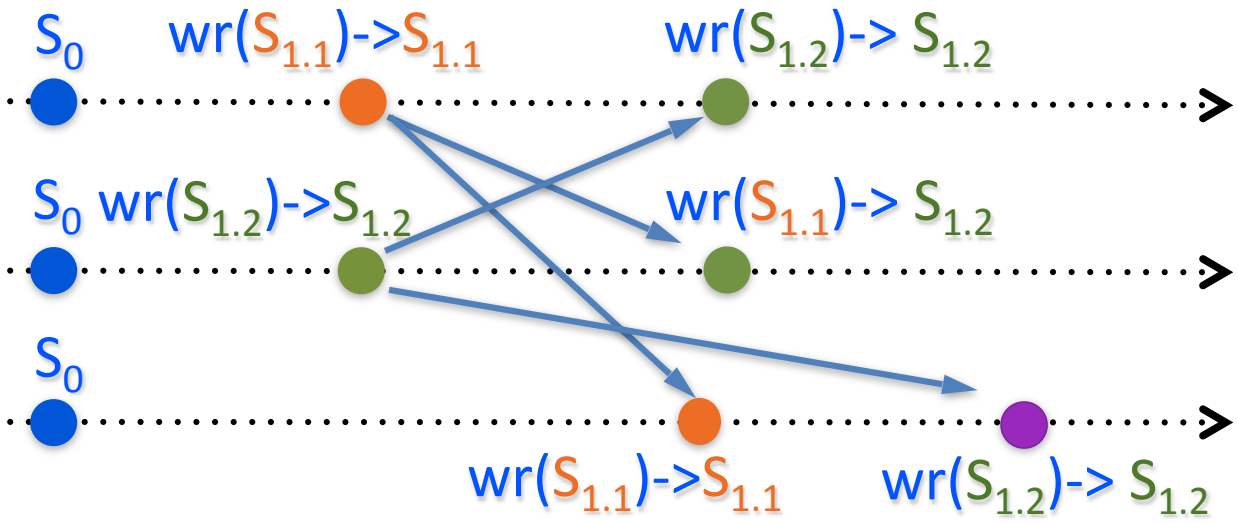
Convergence: sufficient condition

Reliable causal exactly-once delivery

Concurrent Operations must commute ~~and be idempotent~~

[From strong to eventual consistency: getting it right]

Example: highest timestamp wins register



State:

V_{ts} : *value V, timestamp ts*

Operations:

write(V'$_{ts'}$) : if ts' > ts then V_{ts}:= $V'_{ts'}$

[From strong to eventual consistency: getting it right]

Unified CRDT model

State-based:

- Payload type forms a semi-lattice
- Updates are increasing
- *Merge* computes Least Upper Bound
- *Merge* eventually delivered

Operation-based:

- ~~Operations eventually delivered~~
- Concurrent operations commute

Combined:

- Operations are idempotent

[From strong to eventual consistency: getting it right]

CRDTs: Conflict-Free Replicated Data Types

Definition

Convergence conditions: State- vs. Op.-based

Concurrency semantics

Designing CRDTs

Beyond CRDTs

CRDTs: the unfortunate truth

Many data types **do not** naturally commute

Need to **resolve concurrent** operations

[From strong to eventual consistency: getting it right]

Set

Operations:

- *add (*atom *a)*
- *remove (*atom *a)*
- *lookup (*atom *a)* : boolean

No duplicates

- union and intersection commute
- not set difference

[From strong to eventual consistency: getting it right]

Extending the Set seq. spec.

Sequential specification of Set:

- {*true*} add(e) {e ∈ *S*}
- {*true*} rmv(e) {e ∉ *S*}

Commutative ($e \neq f$):

- {*true*} add(e) ; add(e) {e ∈ *S*}
- {*true*} rmv(e) ; rmv(e) {e ∉ *S*}
- {*true*} add(e) ; add(*f*) {e,*f* ∈ *S*}
- {*true*} rmv(e) ; rmv(*f*) {e, *f* ∉ *S*}
- {*true*} add(e) ; rmv(*f*) {e ∈ *S*, *f* ∉ *S*}

Ambiguous:

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

[From strong to eventual consistency: getting it right]

Extending the Set seq. spec.

Sequential specification of Set:

- {*true*} add(e) {e ∈ *S*}
- {*true*} rmv(e) {e ∉ *S*}

"Principle of permutation equivalence"

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

Commutative (e ≠ *f*):

- {*true*} add(e) ; add(e) {e ∈ *S*}
- {*true*} rmv(e) ; rmv(e) {e ∉ *S*}
- {*true*} add(e) ; add(*f*) {e,*f* ∈ *S*}
- {*true*} rmv(e) ; rmv(*f*) {e, *f* ∉ *S*}
- {*true*} add(e) ; rmv(*f*) {e ∈ *S*, *f* ∉ *S*}

What about:

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

[From strong to eventual consistency: getting it right]

Extending the Set seq. spec.

Sequential specification of Set:

- {*true*} add(e) {e ∈ *S*}
- {*true*} rmv(e) {e ∉ *S*}

Commutative (*e* ≠ *f*):

- {*true*} add(e) || add(e) {e ∈ *S*}
- {*true*} rmv(e) || rmv(e) {e ∉ *S*}
- {*true*} add(e) || add(*f*) {e,*f* ∈ *S*}
- {*true*} rmv(e) || rmv(*f*) {e, *f* ∉ *S*}
- {*true*} add(e) || rmv(*f*) {e ∈ *S*, *f* ∉ *S*}

What about:

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

[From strong to eventual consistency: getting it right]

add(e) || rem(e)

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

- ~~linearisable?~~
- last writer wins? { add(e)<rmv(e) ⇒ e ∉ S ∧ rmv(e)<add(e) ⇒ e ∈ 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

[From strong to eventual consistency: getting it right]

Concurrent ≠ Sequential Interleaving

Consider Set-like object S such that:

- $\{true\}$ add(e) $\{e \in S\}$
- $\{true\}$ remove(e) $\{e \notin S\}$
- $\{true\}$ add(e) || remove(e) $\{e \in S\}$

- OR-Set
- add wins

Satisfies correctness conditions:

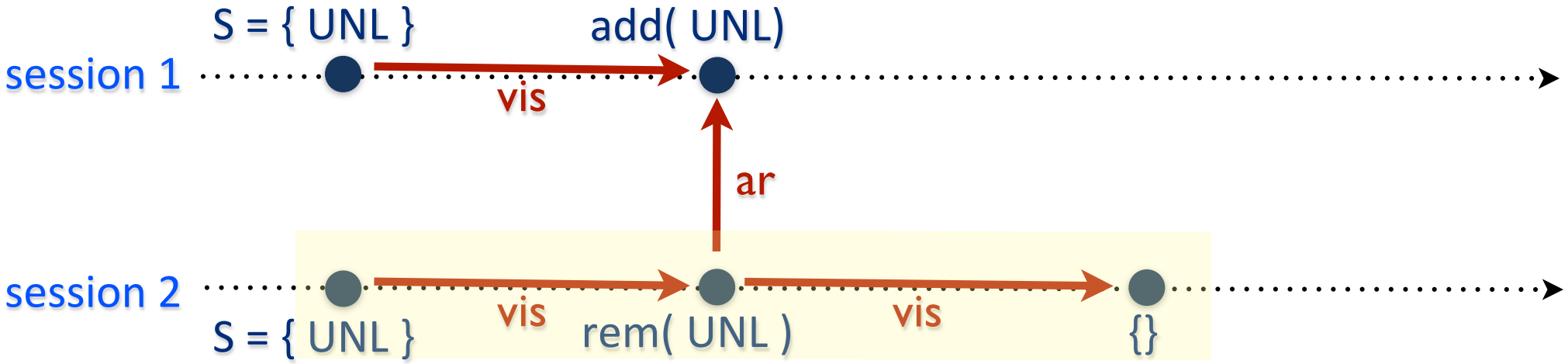
$$\{true\} \left| \begin{array}{c} \text{add(e); remove(e')} \\ \| \\ \text{add(e'); remove(e)} \end{array} \right| \{e, e' \in S\}$$

- Conversely, SC is not SEC
- requires consensus

Cannot be explained by any sequential interleaving

[From strong to eventual consistency: getting it right]

P-O Semantics of add-wins set

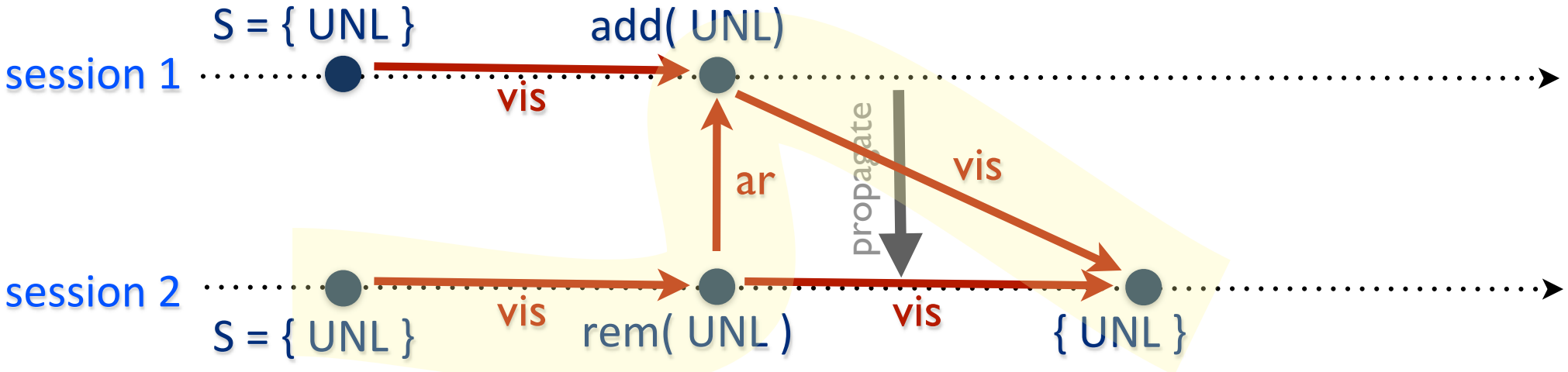


Set

On concurrent add/rem, add wins

[From strong to eventual consistency: getting it right]

P-O Semantics of add-wins set



Set

On concurrent add/rem, add wins

[From strong to eventual consistency: getting it right]

CRDTs: Conflict-Free Replicated Data Types

Definition

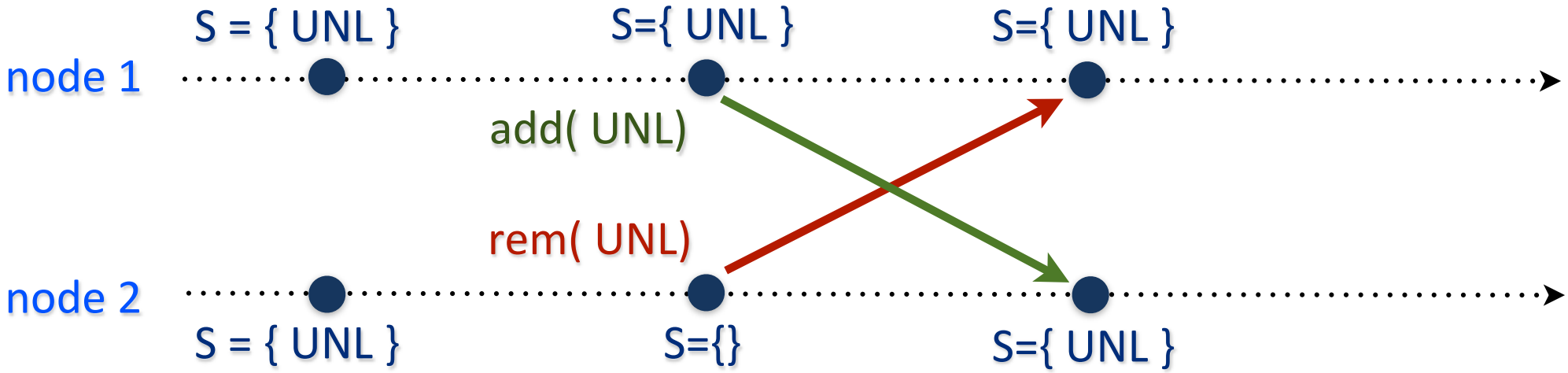
Convergence conditions: State- vs. Op.-based

Concurrency semantics

Designing CRDTs

Beyond CRDTs

Add-wins set (ORSet): operation-based based CRDT

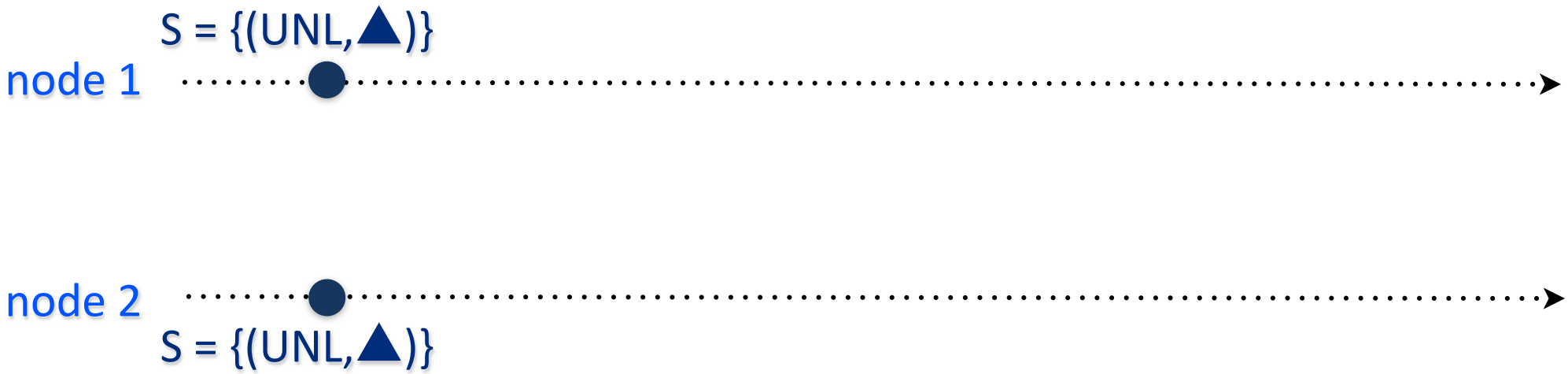


Problem: how to guarantee that:

rem(UNL) || add(UNL) = { UNL }

[From strong to eventual consistency: getting it right]

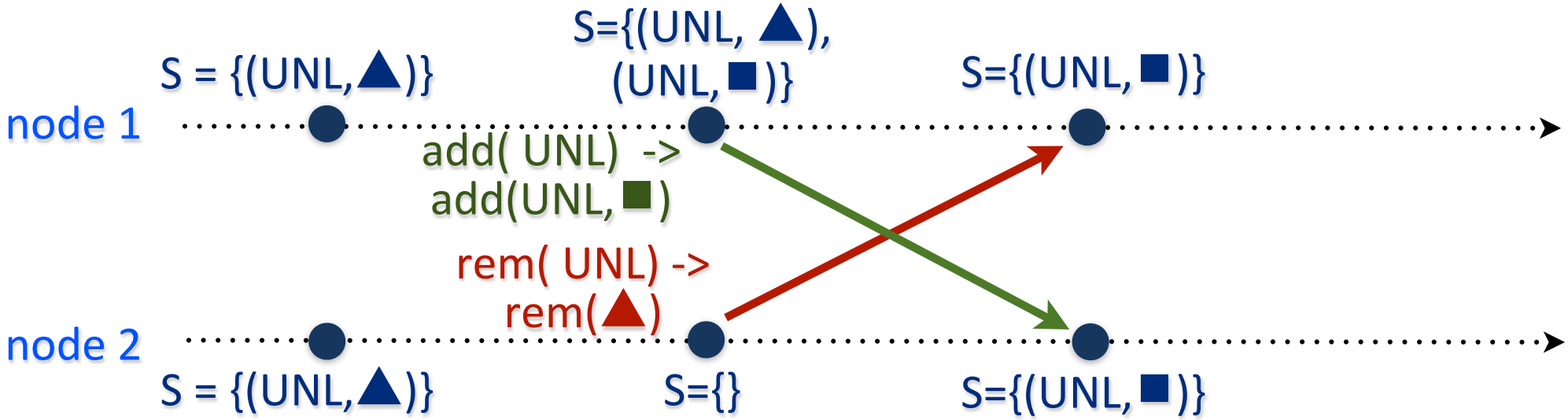
Add-wins set (ORSet): operation-based CRDT



Key ideas:

(1) Tag information with **uids** (▲,Ỳ,■,...)

Add-wins set (ORSet): operation-based CRDT



Key ideas:

(1) Tag information with **uids** (▲, ■,...)

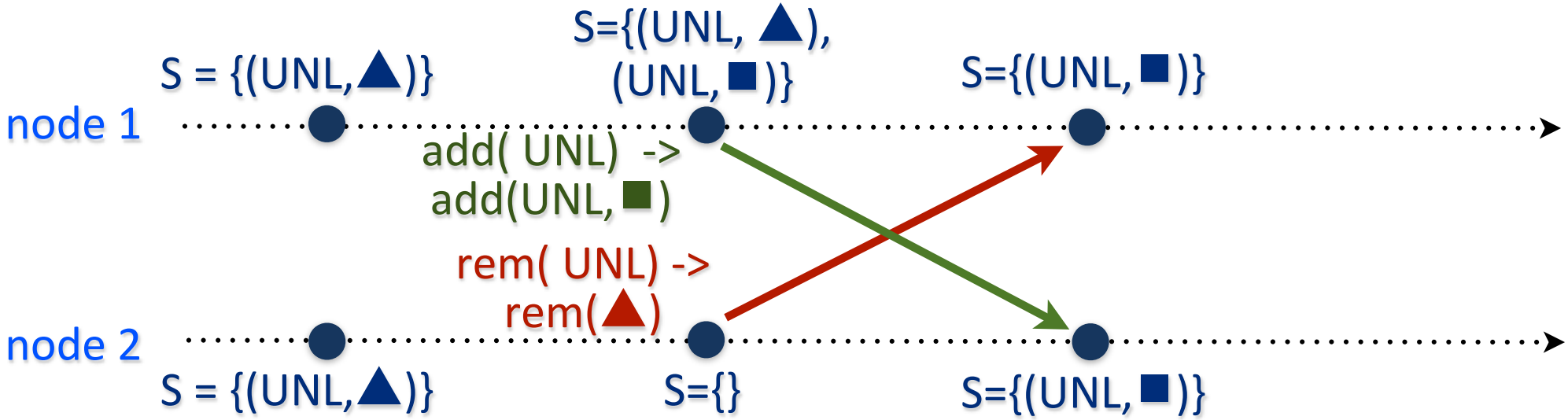
(2) Divide operations into:

prepare: with no side effect

effect: apply side-effects

[From strong to eventual consistency: getting it right]

Add-wins set (ORSet): operation-based CRDT



UID life-cycle

(1) non-existent
(2) created
(3) removed
(4) garbage-collected

Can we garbage-collect after remove operation?

Yes, causal delivery guarantees for any uid ◆:

add(◆) → ***rem***(◆)*

[From strong to eventual consistency: getting it right]

Operation-based OR-Set specification

State: set S of pairs (a,uid_a)

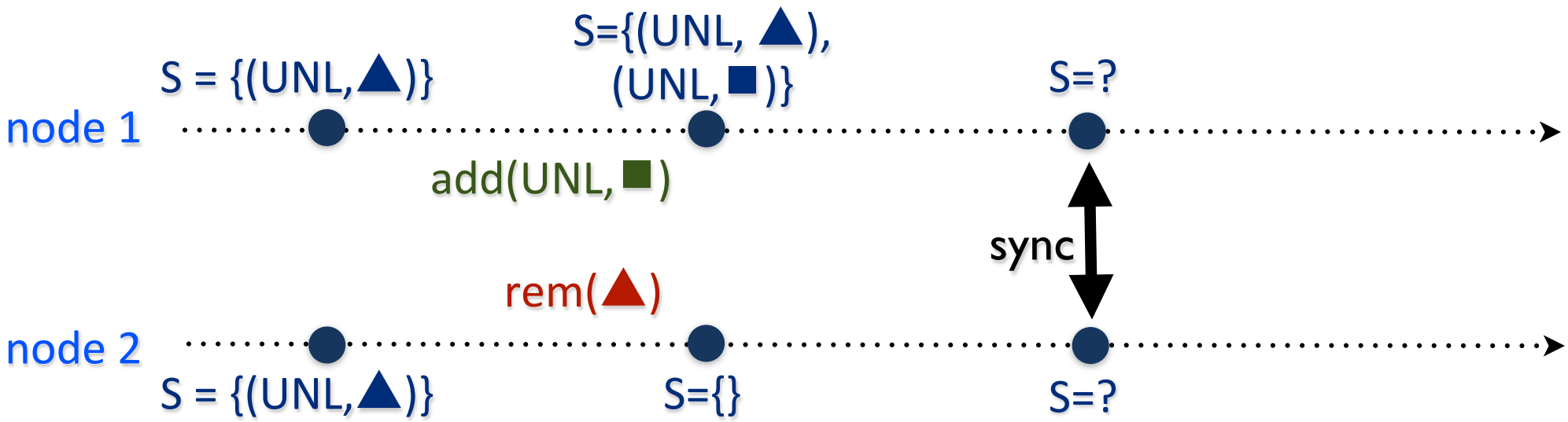
add(a) -> add((a,uid_a))
$S := S \cup \{(a,uid_a)\}$

rem(a) -> rem(old_{uids}) : $old_{uids}=\{u:(a,u) \in S\}$
$S := S \setminus old_{uids}$

lookup() : returns $\{a:(a,_) \in S\}$

[From strong to eventual consistency: getting it right]

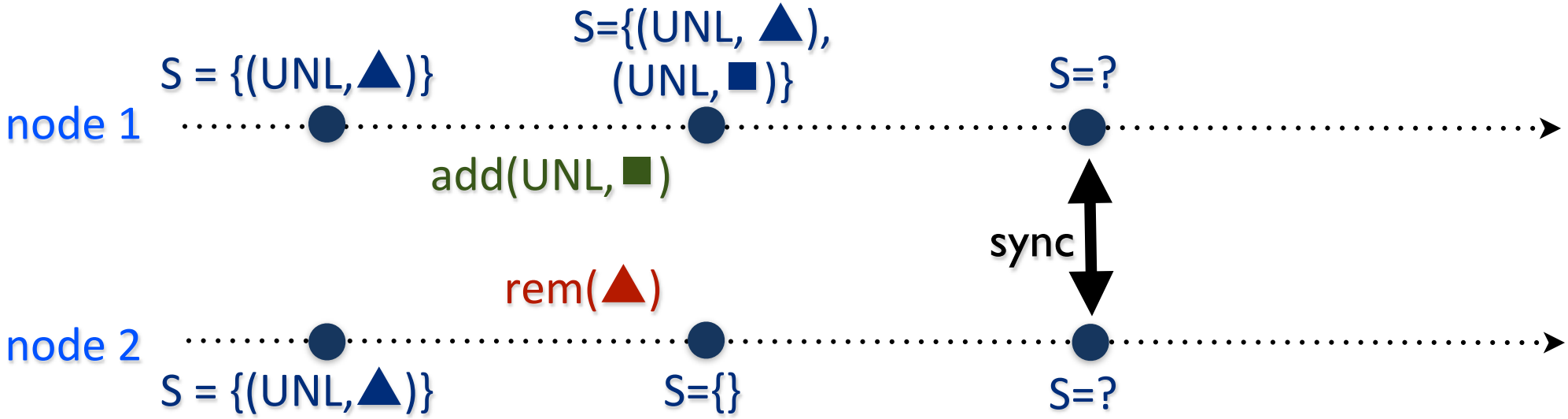
Add-wins set (ORSet): state-based CRDT



Problem: what to do when the state of one replica includes u and the other does not?

[From strong to eventual consistency: getting it right]

Add-wins set (ORSet): state-based CRDT



UID life-cycle

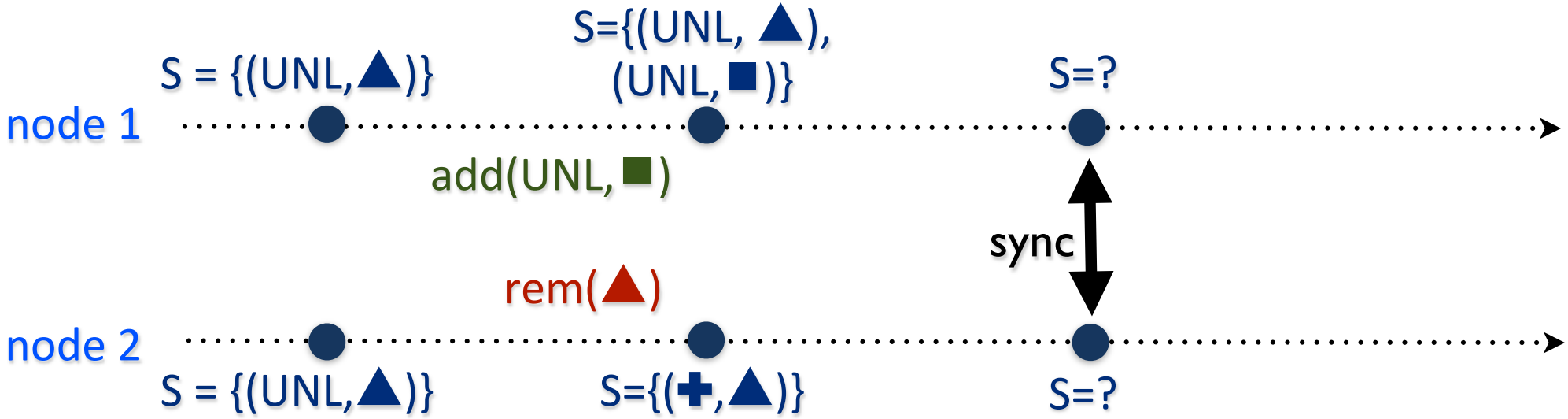
(1) non-existent
(2) created
(3) deleted
(4) garbage-collected

Can we garbage-collect after remove operation?

No - need to keep tombstones.

[From strong to eventual consistency: getting it right]

Add-wins set (ORSet): state-based CRDT

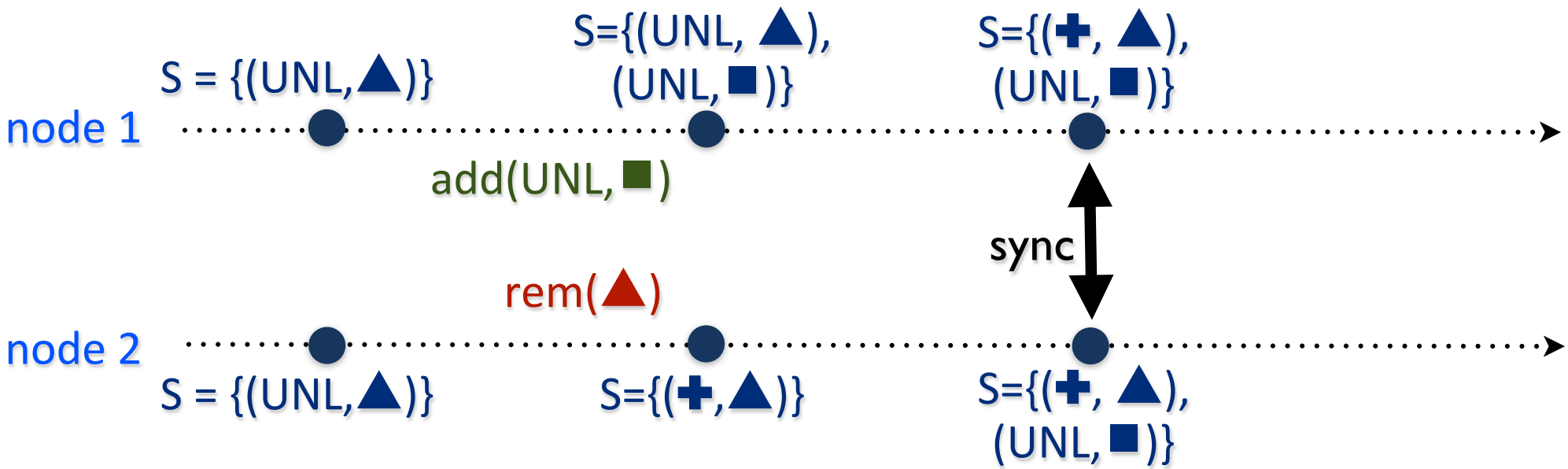


UID life-cycle

(1) non-existent
(2) created
(3) deleted
(4) garbage-collected

[From strong to eventual consistency: getting it right]

Add-wins set (ORSet): state-based CRDT



UID life-cycle

(1) non-existent
(2) created
(3) deleted
(4) garbage-collected

[From strong to eventual consistency: getting it right]

State-based OR-Set specification

State, S

S.A = set of pairs (a,uid_a)

S.R = set of tombstone uids

add (a) -> add((a,uid_a))

S.A := S U {(a,uid_a)}

rem (a) -> rem(old_{uids}) : *old_{uids}={u:(a,u) ∈ S}*

S.A := S.A \ {(a,u): u ∈ old_{uids} } ; S.R = S.R U old_{uids}

lookup() : returns *{a:(a,_) ∈ S.A}*

CRDTs: Conflict-Free Replicated Data Types

Definition

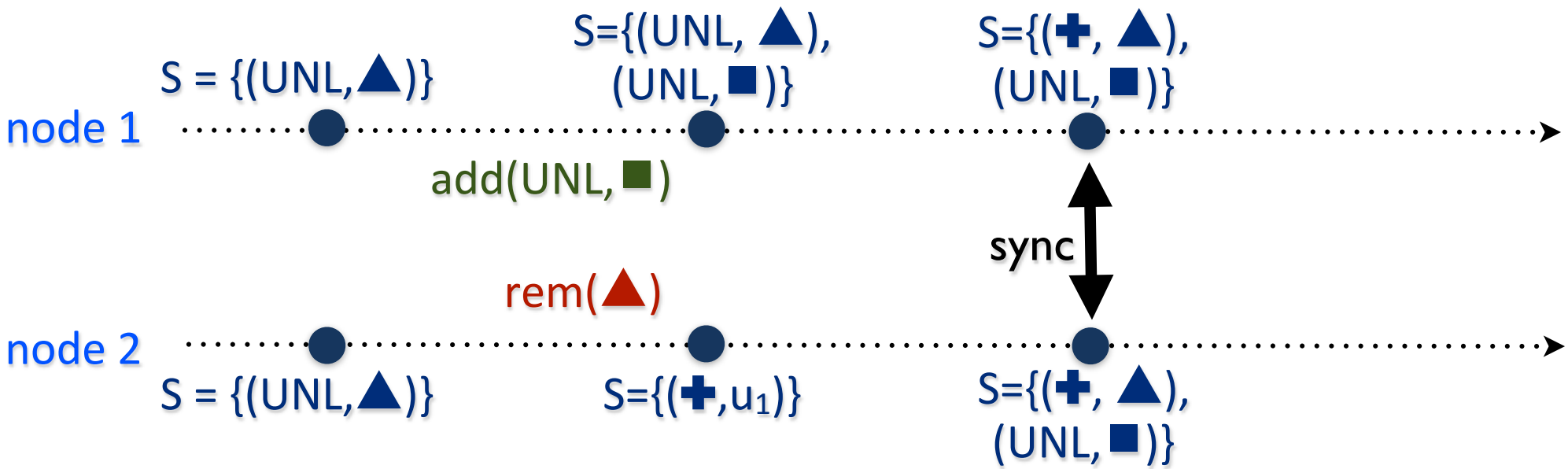
Convergence conditions: State- vs. Op.-based

Concurrency semantics

Designing CRDTs: Optimizing Metadata

Beyond CRDTs

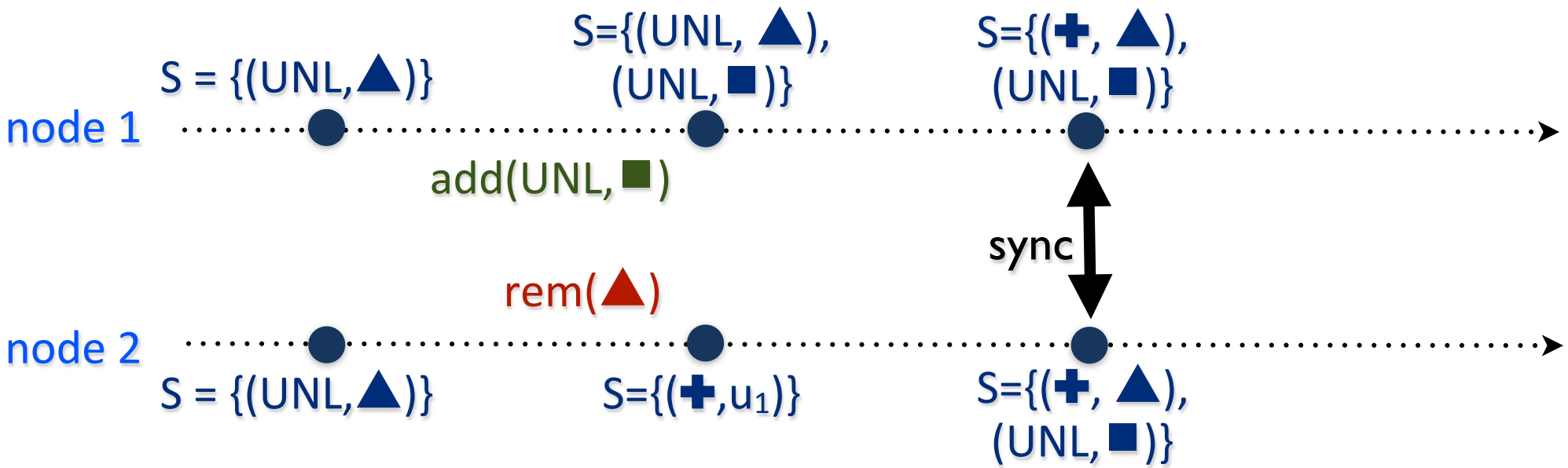
Add-wins set (ORSet): state-based CRDT



Problem: can we add information to support garbage collecting removed *uids*?

[From strong to eventual consistency: getting it right]

Add-wins set (ORSWOT): state-based CRDT

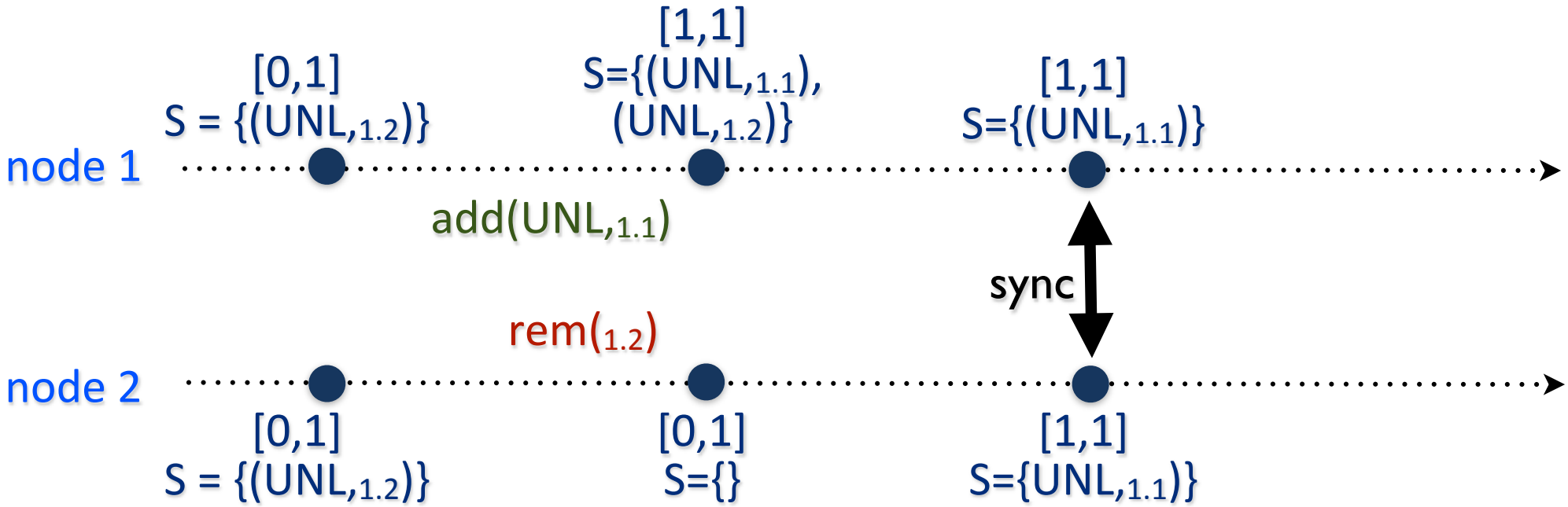


Key idea: summarise removed information

Tag element with logical timestamp
Vector summarises adds received
Merge: check vector: element is new?
Preserves monotonic semi-lattice, ≡ OR-Set

[From strong to eventual consistency: getting it right]

Add-wins set (ORSWOT): state-based CRDT



Key idea: summarise removed information

Tag element with logical timestamp

Vector summarises adds received

Merge: check vector: element is new?

Preserves monotonic semi-lattice, ≡ OR-Set

***u*1.2** *was removed because (1.2) ∈ [0,1]*

***u*1.1** *was created because (1.1) ∉ [0,1]*

[From strong to eventual consistency: getting it right]

Library of CRDTs

Register
- Last-Writer Wins
- Multi-Value

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

Map

File-system tree

Counter
- Unlimited
- Non-negative

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence

CRDTs: Conflict-Free Replicated Data Types

Definition

Convergence conditions: State- vs. Op.-based

Concurrency semantics

Designing CRDTs

Beyond CRDTs: Transactions & Invariants

Transactions

Snapshot isolation (conceptually)

Begin: take database snapshot

Read/Write: execute on database snapshot

Commit: success if no write/write conflict in concurrent transaction

[From strong to eventual consistency: getting it right]

Transactions

Parallel Snapshot isolation (conceptually)

Begin: take database snapshot from a single replica

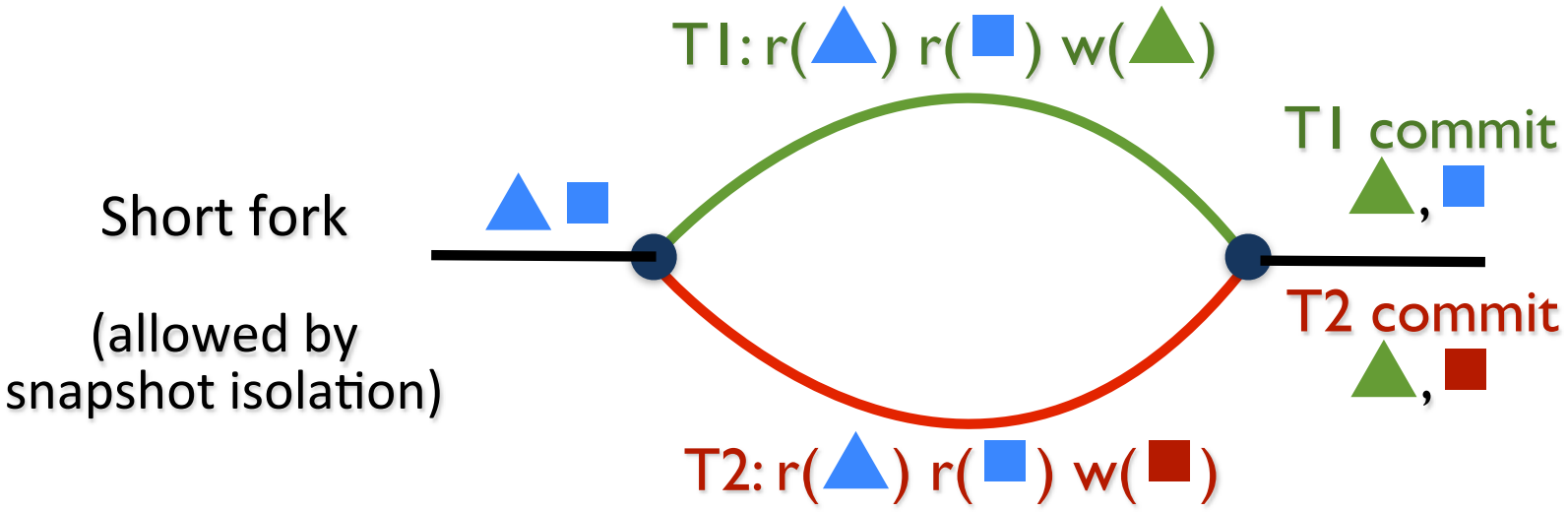
Read/Write: execute on database snapshot

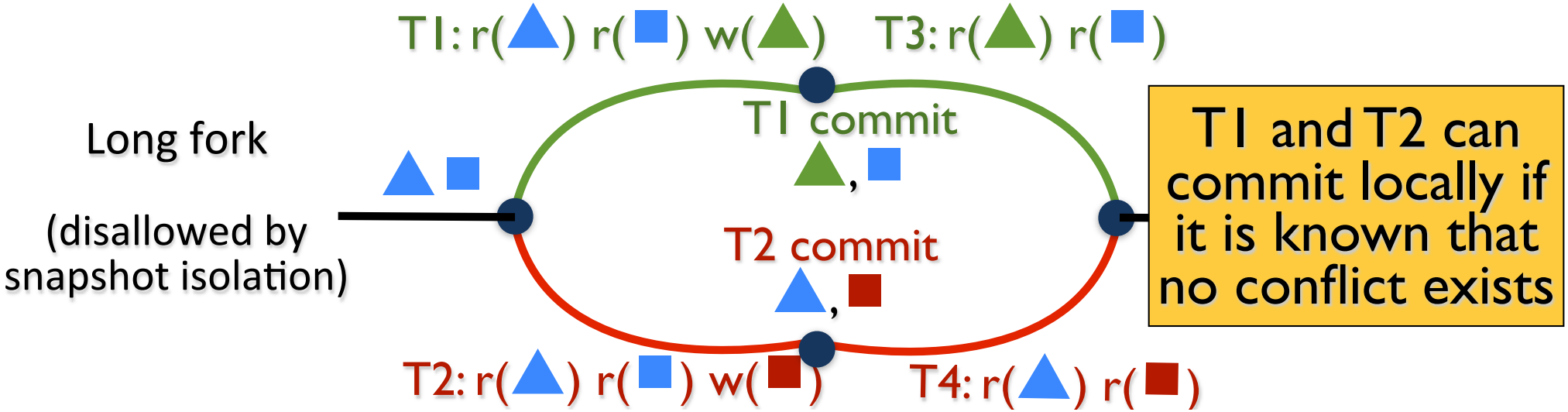
Commit: success ~~if no write/write conflict in concurrent transaction~~ even on concurrent write/write in a CRDT

From: Y Sovran, et.al. Transactional storage for geo-replicated systems. In SOSP'11.

[From strong to eventual consistency: getting it right]

Parallel snapshot isolation: anomalies





From: Y Sovran, et.al. Transactional storage for geo-replicated systems. In SOSP'11.

SwiftCloud: geo-replication right to the client machine

Extreme numbers of clients

Large database

Full replication?

Causal consistency for servers and clients

Fault tolerance

Switching servers?

Metadata proportional to # clients?

[From strong to eventual consistency: getting it right]

SwiftCloud: geo-replication right to the client machine

Low latency

… for **reads**

Partial replication/caching in clients

… for **writes**

Weak consistency

Mergeable writes => CRDT

+ Transactions with parallel snapshot isolation (PSI transactions)

[From strong to eventual consistency: getting it right]

Causal consistency in clients

Problem

Metadata of size O(#clients) does not scale

Key idea:

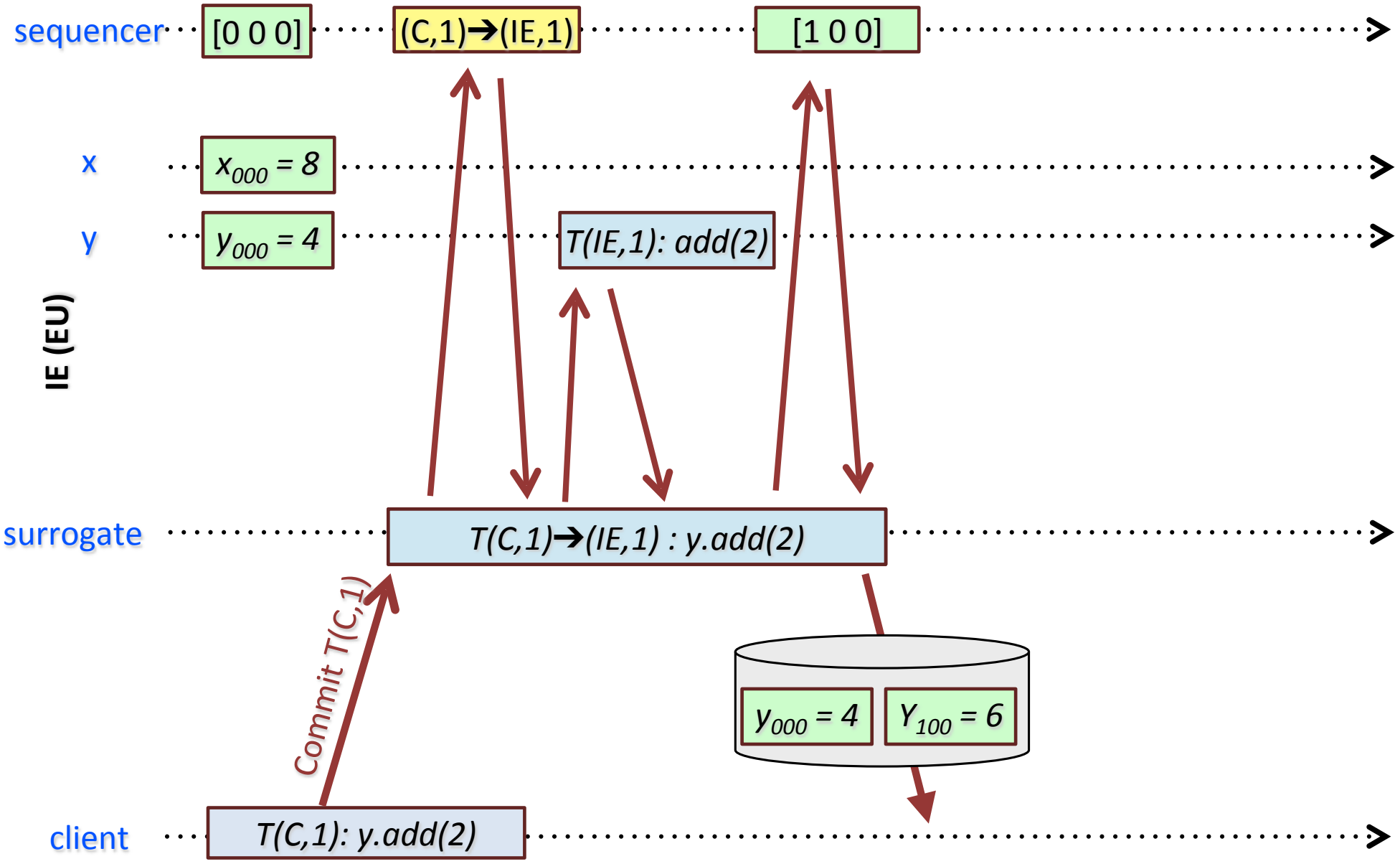
Assign server timestamp when operation received in DC

Use version vector of size O(#DCs) for tracking causal dependencies

Use client timestamp to filter duplicates

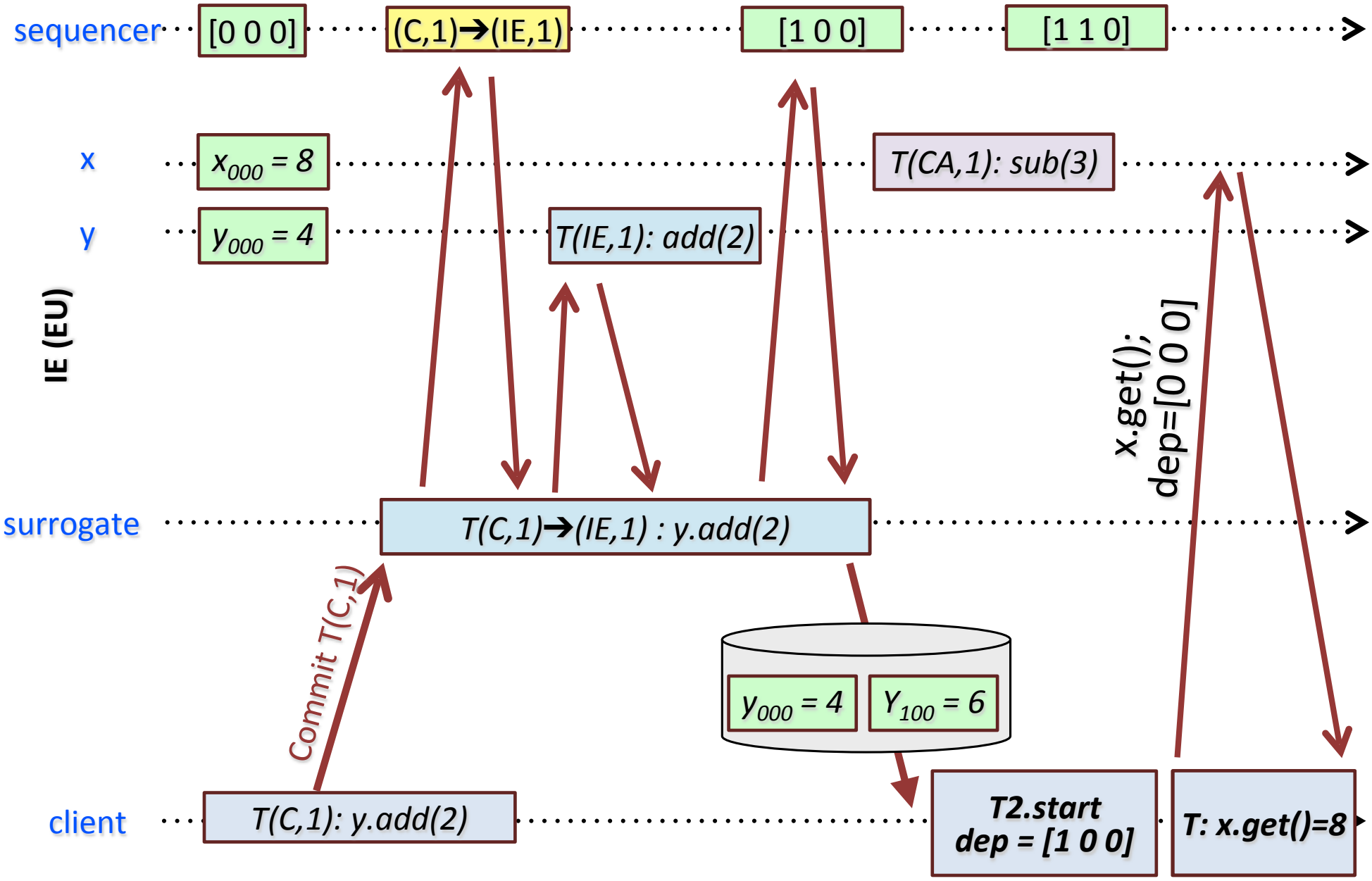
[From strong to eventual consistency: getting it right]

Transaction execution on server



[From strong to eventual consistency: getting it right]

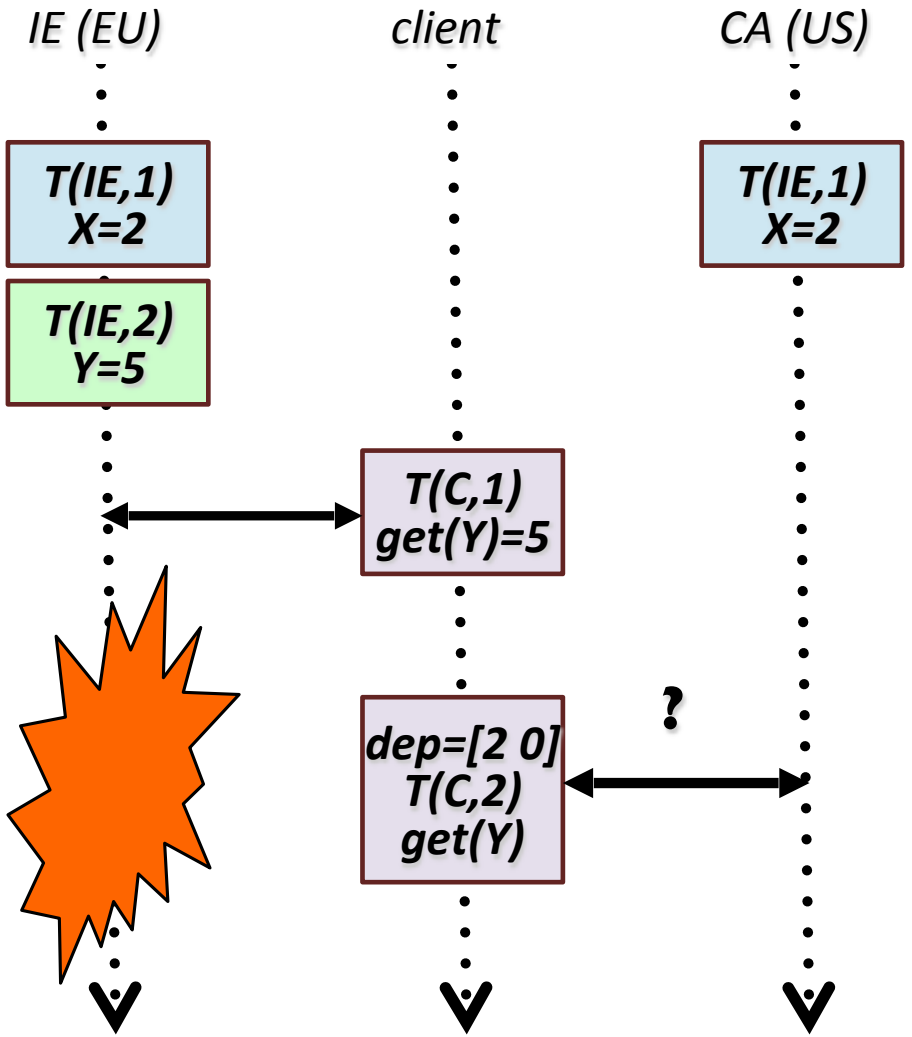
Transaction execution on server



[From strong to eventual consistency: getting it right]

Support DC-failures or Client/DC network failure

- Problem: on fail-over, new DC may not know some observed updates
  - Leads to blocking or breaking session guarantees
- Potential solutions
  - Operations only complete after being stable in f+1 DCs
    - Slow writes
  - Involve clients in fail-over: clients keep enough information to ensure session guarantees



[From strong to eventual consistency: getting it right]

Client-assisted failover

Own updates

- Keep a log of own updates

Observable state

- Union of own updates and stable updates

On fail-over

- Replay log of own updates => may lead to double delivery
- Guarantee idempotence – rely on CRDT properties; use client identifiers in operation execution

[From strong to eventual consistency: getting it right]

CRDTs: Conflict-Free Replicated Data Types

Definition

Convergence conditions: State- vs. Op.-based

Concurrency semantics

Designing CRDTs

Beyond CRDTs: Transactions & Invariants

Limitations of PSI transactions

Similar to snapshot isolation

- Not serializable
- Cannot maintain (all) invariants across data items
- E.g.: *invariant*: $\forall$ x, x $\in$ groups $\Rightarrow$ x $\in$ participants
- participants.rem("Nuno") || groups.add("Nuno","Marc")

Unlike snapshot isolation

- Cannot maintain (some) invariants for a single data item
- E.g.: *invariant*: stock $\geq$ 0
- stock.dec(1) || stock.dec(1)

[From strong to eventual consistency: getting it right]

Invariant-preserving CRDTs (InvCRDT)

Strong consistency/**serializability** is often used to guarantee that an **invariant** is **not broken**

E.g.: $stock \geq 0$

Could rely on **weak consistency** if **invariant-preservation** is guaranteed

[From strong to eventual consistency: getting it right]

InvCRDT: e.g.: Non-negative counter

Key idea: local invariants that imply global invariants

Global invariant : $x \geq 0$

E.g. $x = 100$

x can be decremented by 100

Split rights by the replicas

for 5 replicas, each replica can decrement x by 20

each replica can additionally decrement x by the value it incremented x

Operations that breaks local invariant needs to obtain additional rights

[P. O'Neil, The escrow transactional method, PODS'86]

When InvCRDTs are not enough...

...combine CRDTs with strong consistency

Walter [SOSP'11]

Gemini - red/blue consistency model [OSDI'12]

[From strong to eventual consistency: getting it right]

Road ahead

New data types

Composing CRDT

Dynamic sharing and slicing of CRDTs

Understand semantics (PLV community)

- E.g. several papers at POPL addressing weak consistency

Building effective EC databases with better guarantees

- Causal consistency
- Invariant-preservation
- Transactions
- Security

Final remarks

Principled approach

Two sufficient conditions

- State: monotonic semi-lattice
- Operation: commutativity

Useful CRDTs

- Register, Counter, Set, Map (KVS), Graph, Monotonic DAG, Sequence

Beyond simple CRDTs

- Transactions, invariant-preserving CRDT, etc.

[From strong to eventual consistency: getting it right]

CRDTs in the wild

Walter [SOSP 2011]: c-set

SwiftCloud: CRDTs + transactions + FT session guarantees

Riak currently supports counters; version 2.0 will support additional CRDTs (set, map)

Third-party implementations: StateBox (Erlang), KnockBox (Clojure), meangirls (Ruby), riak-java-crdt (Java)

[From strong to eventual consistency: getting it right]

Contributors

Annette Bieniusa
Carlos Baquero
Joan Marquès
Mahsa Najafzadeh
Marek Zawirski
Marc Shapiro
Mihai Leţia
Nuno Preguiça
Sérgio Duarte
Valter Balegas

(/papec/node/2274)

Strong consistency is easy to understand, but it requires synchronisation, which has a high cost and does not tolerate partial system failure or network partitions well. Hence, the past decade has seen renewed interest in an alternative - Eventual Consistency (EC) - where data is accessed and modified without coordination. Many production systems guarantee only EC, including NoSQL databases, key-value stores and cloud storage systems. EC improves responsiveness and availability, updates propagate more quickly, and enables less coordination-intensive and often less complex protocols.

EC is crucial for scalability but represents an unfamiliar design space. Conflicts and divergence are possible, making them hard to program against, while metadata growth and maintaining system-wide invariants can also become problematic. As more developers interact with and program against distributed storage systems, understanding and managing these trade-offs becomes increasingly important.

Although EC remains poorly understood, there has recently been considerable momentum in the research and development community: the past several years have seen the (re)introduction of useful concepts, such as replicated data types, monotonic programming, causal consistency, red-blue consistency, novel proof systems, etc., designed to allow improve efficiency, programmability, and overall operation of weakly consistent stores.

This workshop aims to investigate the principles and practice of Eventual Consistency. It will bring together theoreticians and practitioners from different horizons: system development, distributed algorithms,

Co-located with:

EuroSys 2014
(http://eurosys2014.vu.nl/)

Sponsored by:

(https://syncfree.lip6.fr/)
SyncFree (https://syncfree.lip6.fr/)

(http://concordant.lip6.fr/)
ConcoRDanT (http://concordant.lip6.fr/)