

Conflict-free Replicated Data Types – A principled approach to Eventual Consistency

Marc Shapiro, *INRIA & LIP6*
Nuno Preguiça, *U. Nova de Lisboa*
Carlos Baquero, *U. Minho*
Marek Zawirski, *INRIA & UPMC*

Replication 101

Replicated data

Share data $\Rightarrow$ Replicate at many locations

- Performance: local reads
- Availability: immune from network failure
- Fault-tolerance: replicate computation
- Scalability: load balancing

Updates

- Push to all replicas:
 - Asynchronous: Reliable Multicast
 - Synchronous: Atomic Multicast
- Consistency?

Strong consistency

State Machine Replication

- Arbitrary sequential, deterministic object
- Globally: total order of updates
- All replicas execute updates in same order

Consensus

- Simultaneous N-way agreement
- Fault-tolerance: *cf* FLP85
- Doesn't scale
- $\equiv$ Atomic Multicast
- building block for Commitment, etc.

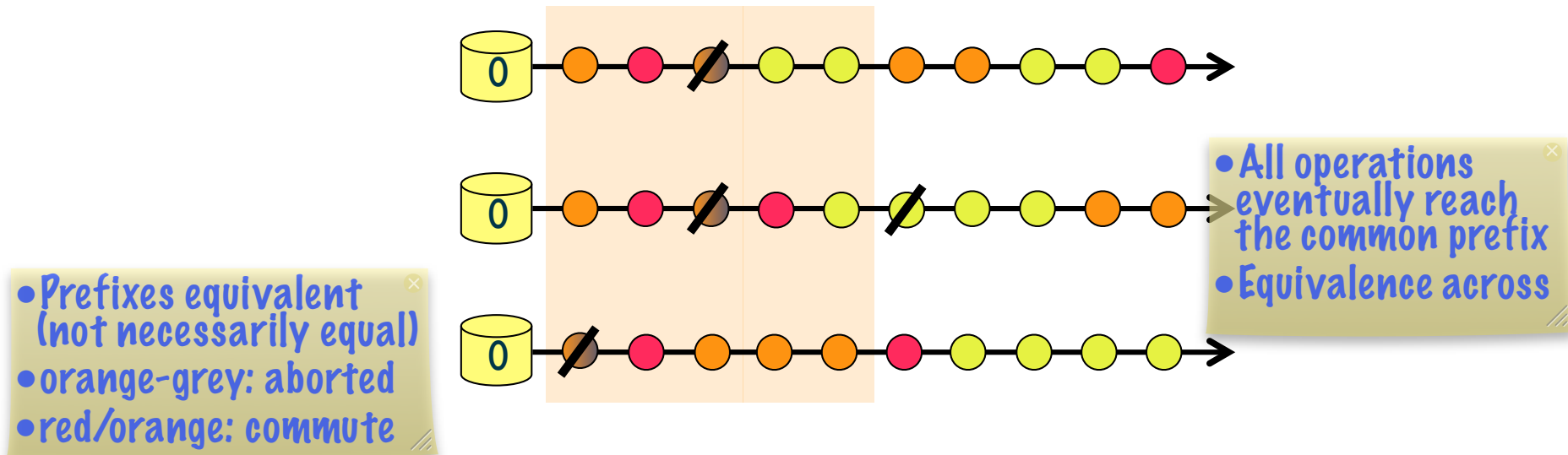
Eventual consistency

Optimistic approach

- Avoid (foreground) synchronisation
- Speculate: replicas diverge
- Reliable broadcast (scalable)
- If no conflicts, merge
- Otherwise, reconcile;
 - arbitrate, merge
 - roll back inconsistent replicas, roll forward

Consistent when all replicas have received all operations & arbitrations

Convergence



Consensus on next extension of prefix

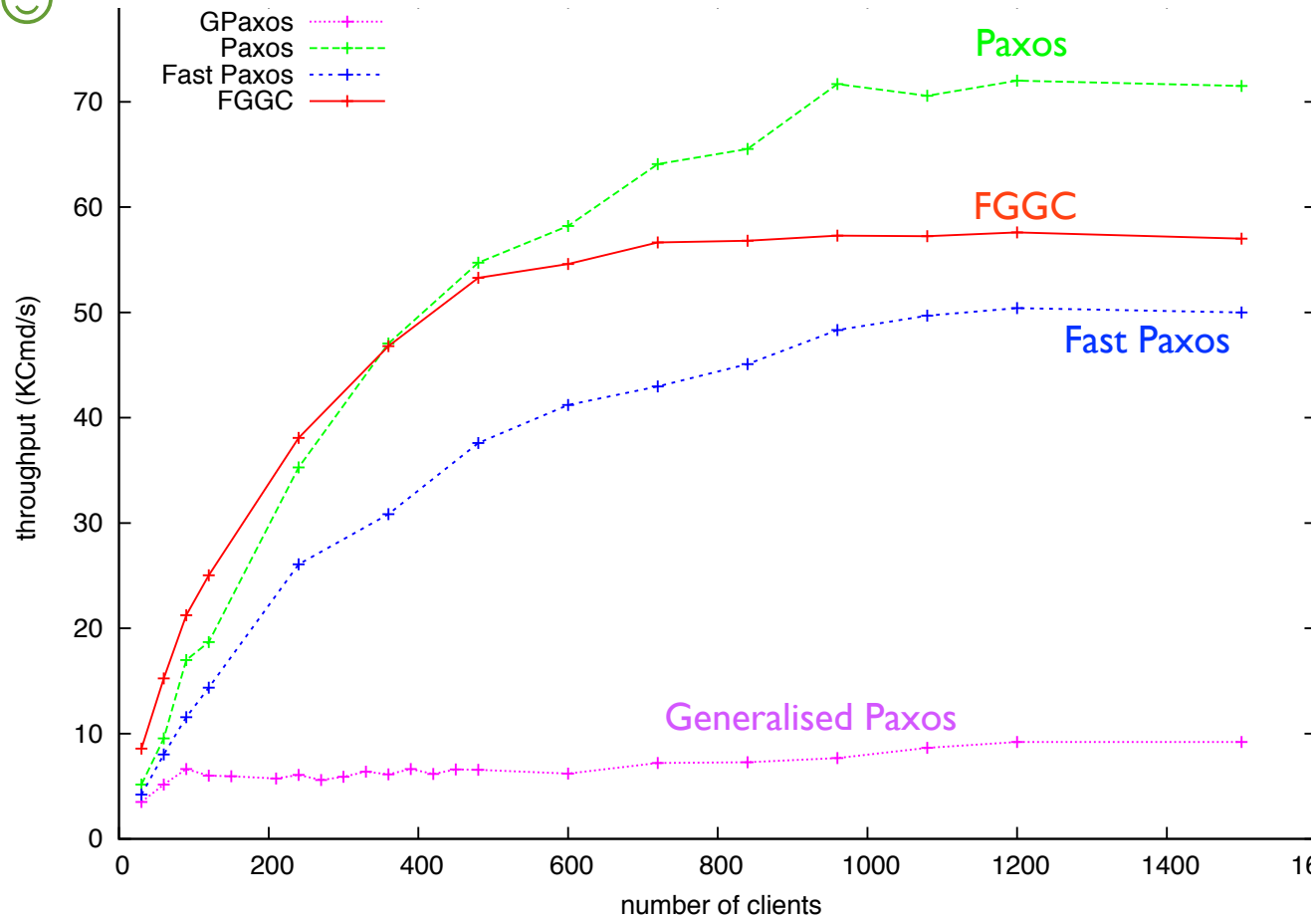
- In the background
- Local progress not blocked
- Conflict $\Rightarrow$ rollback

Improve consensus:

- Leverage future, semantics
- Genuine partial replication

- Minimise rollbacks
- Choose most promising schedule
- Commutativity

Fast Gen.lised Genuine Paxos

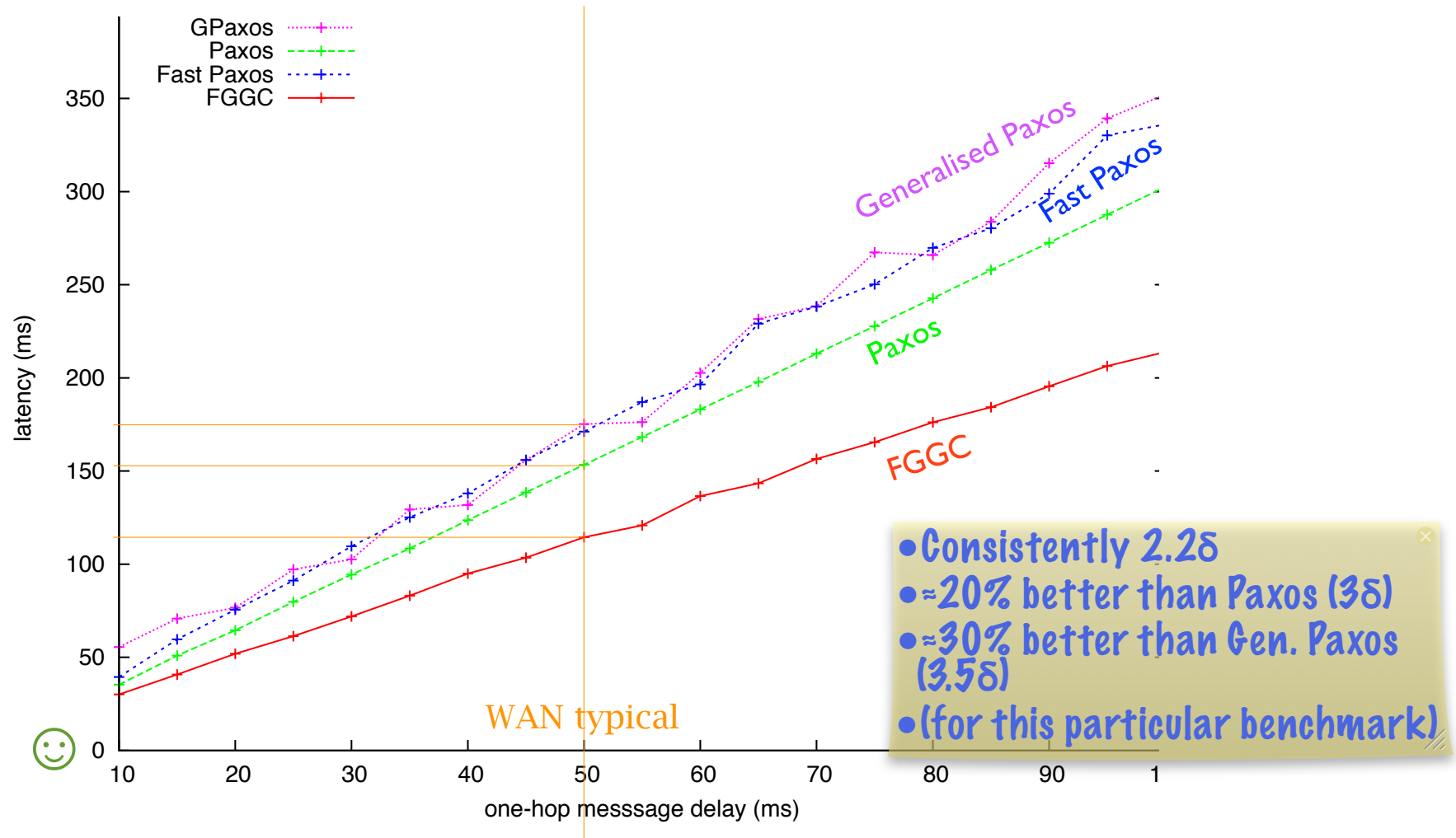


• On a LAN Paxos is fastest (because simplest?)

• Collision recovery has high cost in Generalised Paxos

Throughput on LAN (1024 regs.) [Sutra 2010]

Fast Gen.lised Genuine Paxos



Varying message delay (1024 regs., 360 clients) [Sutra 2010]

Eventual Consistency so far

Eventual consistency

- Moved consensus to background
- Optimistic, speculation
⇒ more available, responsive
- High-level operations: leverage semantics
- Knowledge of “future”

Arbitration/merge on conflict

- Surprisingly complex
- Mostly ad-hoc, error-prone

Improved consensus

- Very complex

Scalability?



• Availability ++
• Latency --

• Complexity ++
• Scalability???

Conflict-free Replicated Data Types (CRDTs)

Intuition:

- *Conflicts* are the problem
- Design data types with no conflicts

CRDTs

- Available, fast
- Reconcile scalability + consistency

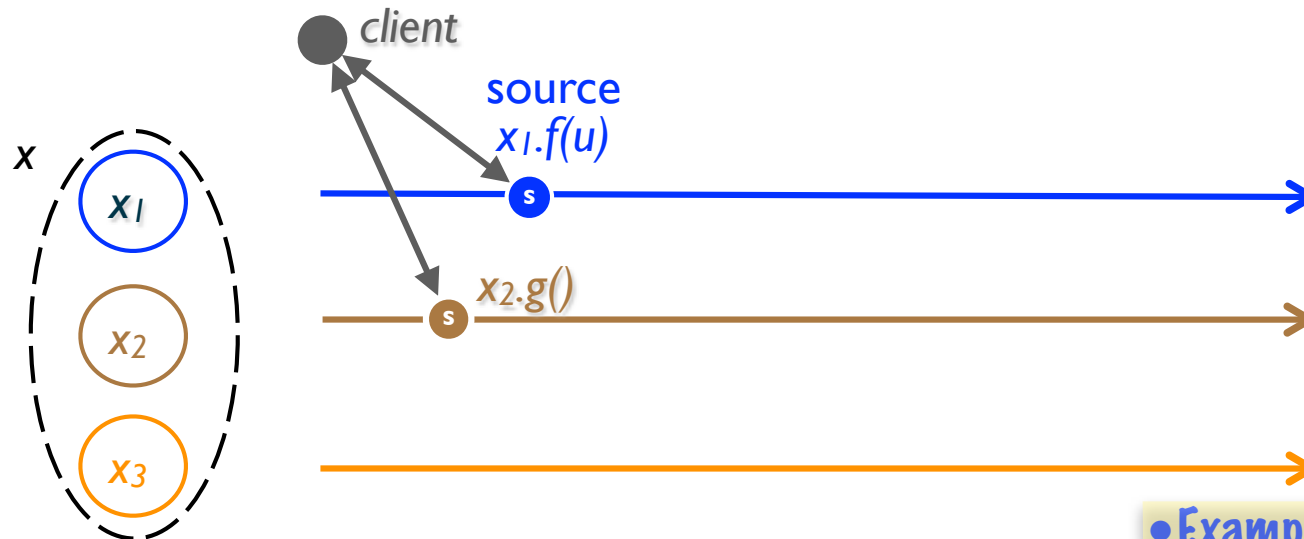
Simple sufficient conditions

- Principled, correct

CRDTs: The theory

Sufficient conditions for
correctness without
synchronisation

Query

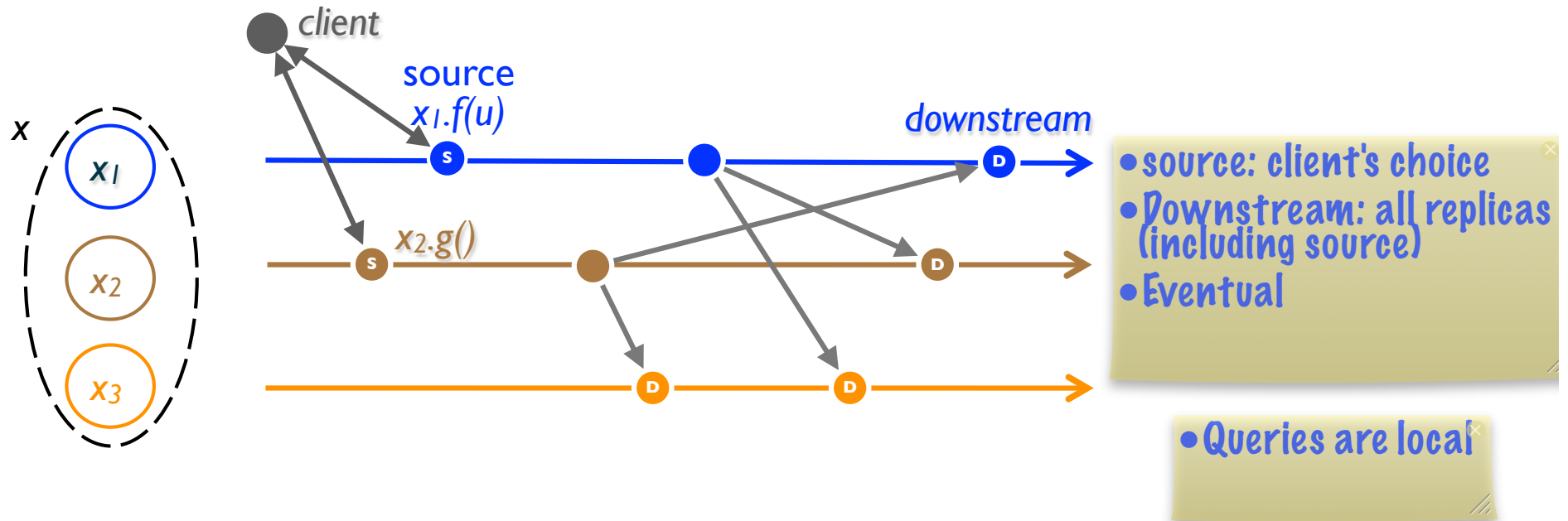


- Example: Amazon shopping cart is replicated
- unspecified client, e.g., Web front-end
- One or more
- load-balancer, failures may direct client to different replicas

Local at source replica

- Client's choice

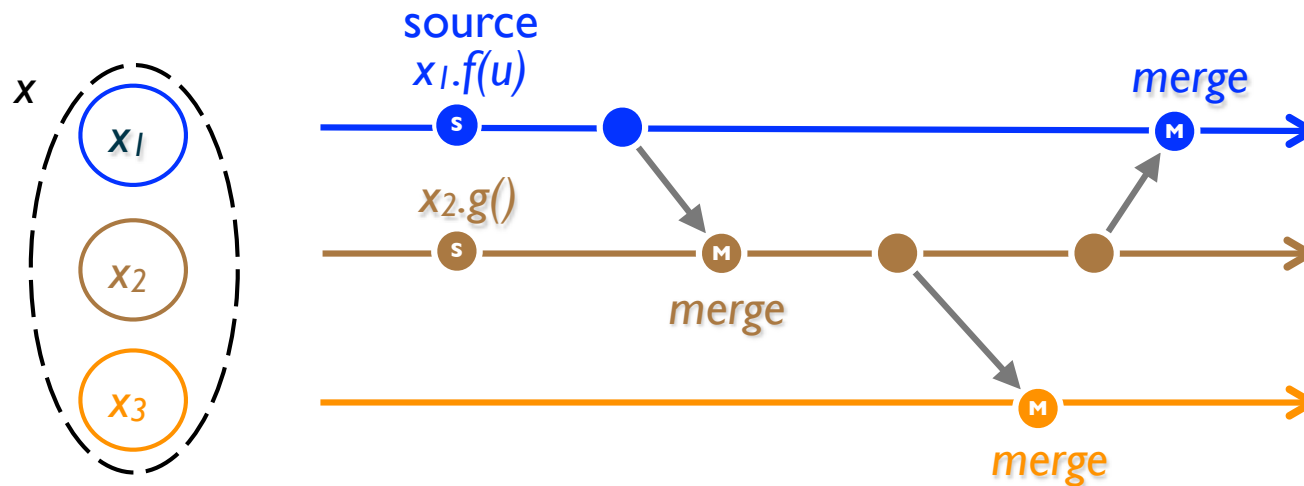
Update



Two-phase updates

- At source:
 - Synchronous
 - Atomic
- Downstream:
 - Asynchronous
 - Atomic

State-based replication



Update at source $x_1.f(u)$, $x_2.g()$, ...

- Precondition, compute
- Assign payload

Convergence:

- Episodically: send x_i payload
- On delivery: *merge* payloads

- merge two valid states
- produce valid state
- no historical info available

State-based specification

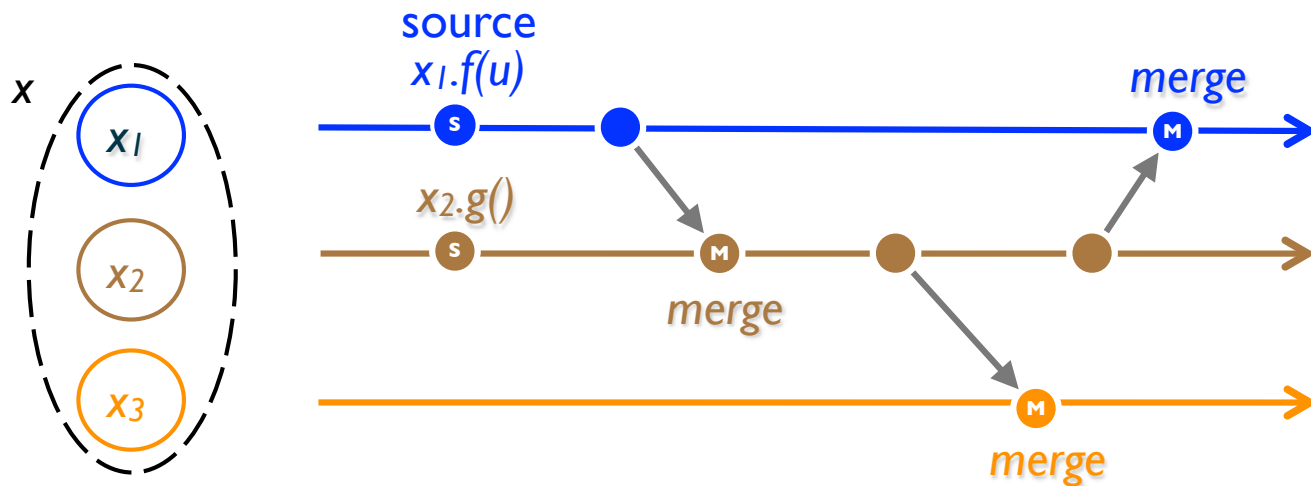
payload Payload type; instantiated at all replicas
initial Initial value
query Query (arguments) : returns
pre Precondition
let Evaluate synchronously, no side effects

update Source-local operation (arguments) : returns
pre Precondition
let Evaluate at source, synchronously
Side-effects at source to execute synchronously

compare (value1, value2) : boolean b
Is $value1 \leq value2$ in semilattice?

merge (value1, value2) : payload mergedValue
LUB merge of value1 and value2, at any replica

State-based convergent objects: CvRDT



If

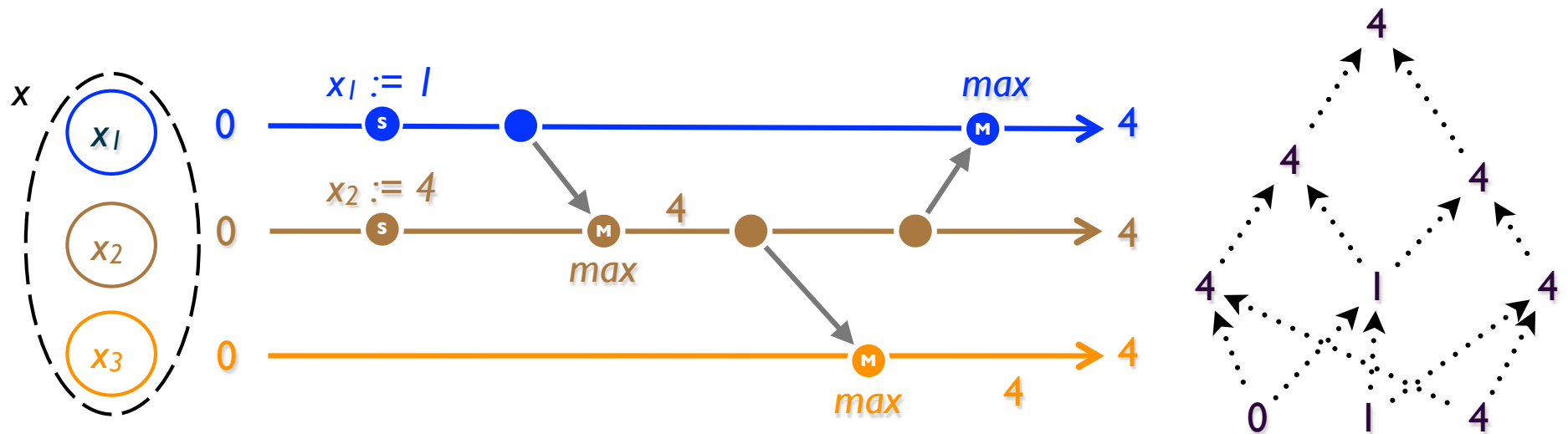
- payload type forms a semi-lattice
 - updates are increasing
 - *merge* computes Least Upper Bound $\sqcup$
- then replicas converge to LUB of last values

Example: Payload = int, *merge* = *max*

• $\sqcup$ = Least Upper Bound
LUB = merge

• no reference to history

Example CvRDT



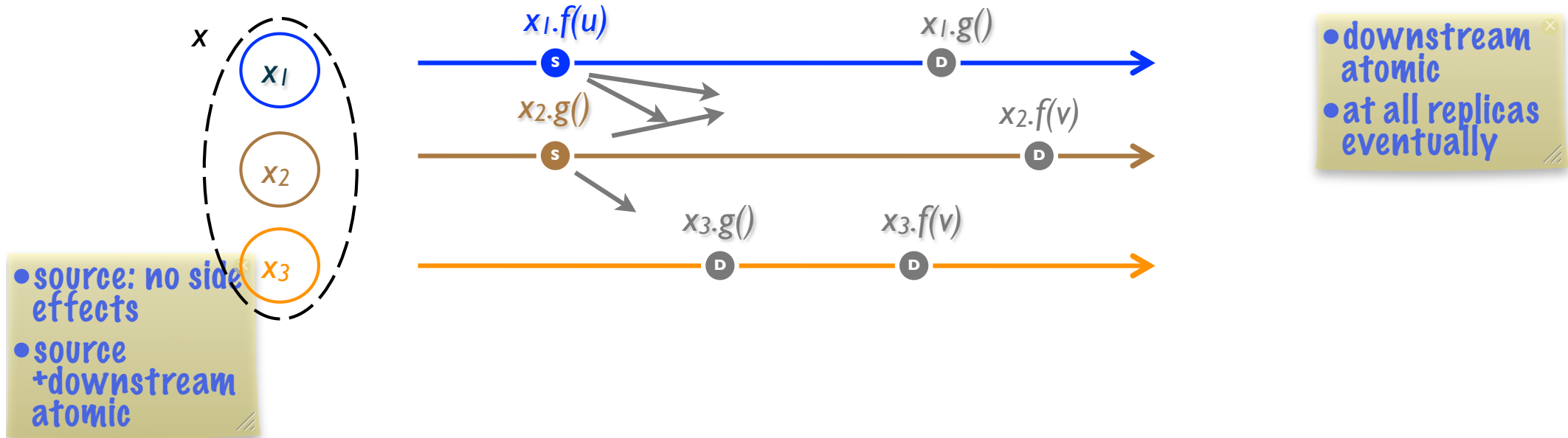
If

- payload type forms a semi-lattice
- updates are increasing
- *merge computes Least Upper Bound* $\sqcup$

then replicas converge to LUB of last values

Example: $f = \text{assign}$, $\text{merge} = \text{max}$

Operation-based replication



At source:

- source precondition, computation
- broadcast to all replicas

Eventually, at all replicas:

- downstream precondition
- Assign local replica

Operation-based specification

payload *Payload type; instantiated at all replicas*

initial *Initial value*

query *Source-local operation (arguments) : returns*

pre *Precondition*

let *Execute at source, synchronously, no side effects*

update *Global update (arguments) : returns*

atSource (arguments) : returns

pre *Precondition at source*

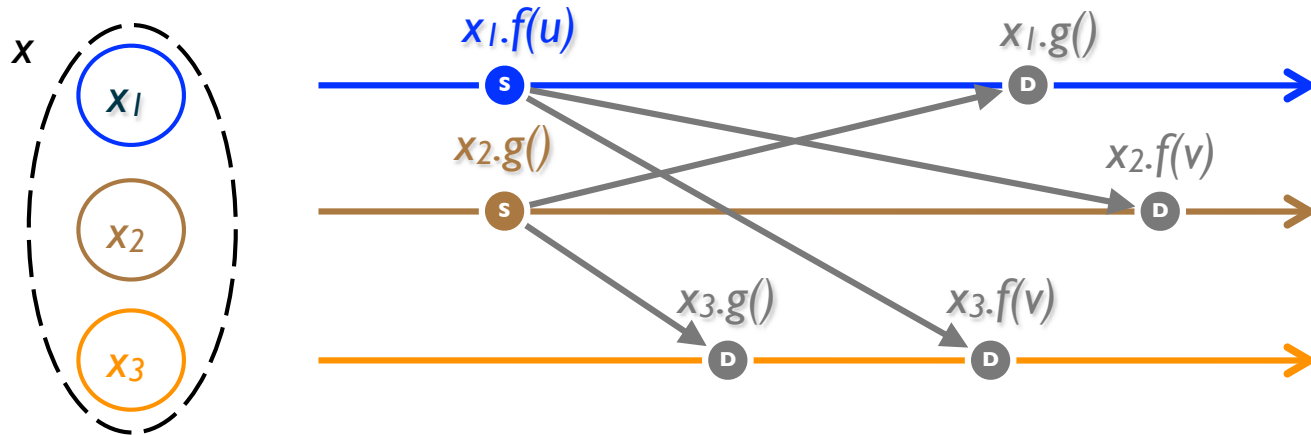
let *1st phase: synchronous, at source, no side effects*

downstream (arguments passed downstream)

pre *Precondition against downstream state*

2nd phase, asynchronous, side-effects to downstream state

Commutative-operation-based objects: CmRDTs



- Delivery order $\approx$ ensures downstream precondition
- happened-before or weaker

If:

- (*Liveness*) all replicas execute all downstreams in precondition order
- (*Safety*) concurrent operations all commute

Then: replicas converge

$C_vRDT \equiv C_mRDT$

Operation-based emulation of state-based object

- At source: apply state-based update
- Downstream: apply state-based *merge*
- Monotonic semi-lattice $\Rightarrow$ commute

State-based emulation of op-based object

- Update: at-source, add op to set of messages
- Merge: union of message sets
- Execute when *dpre* = true
- Live: eventual delivery, eventual execute
- Commute $\Rightarrow$ semi-lattice

- Use state or operations
- as convenient

CRDTs: The challenge

What interesting objects can
we design with no
synchronisation whatsoever?

Counter

Increment / decrement

- Payload: $P = [\text{int}, \text{int}, \dots]$,
 $N = [\text{int}, \text{int}, \dots]$
- $\text{value}() = \sum_i P[i] - \sum_i N[i]$
- $\text{increment} () = P[\text{MyID}]++$
- $\text{decrement} () = N[\text{MyID}]++$
- $\text{merge}(x,y) =$
 $x \sqcup y = ([\dots, \max(x.P[i], y.P[i]), \dots]_i,$
 $[\dots, \max(x.N[i], y.N[i]), \dots]_i)$
- Positive or negative



Counter

Increment / decrement

- Payload: $P = [\text{int}, \text{int}, \dots]$,
 $N = [\text{int}, \text{int}, \dots]$
- $\text{value}() = \sum_i P[i] - \sum_i N[i]$
- $\text{increment} () = P[\text{MyID}()]++$
- $\text{decrement} () = N[\text{MyID}()]++$
- $\text{merge}(x,y) =$
 $x \sqcup y = ([\dots, \max(x.P[i], y.P[i]), \dots]_i,$
 $[\dots, \max(x.N[i], y.N[i]), \dots]_i)$
- Positive or negative

- N masters
- vector of single-master counters
- like vector clock

- can't maintain global invariant such as $x > 0$

Register

Container for a single atom

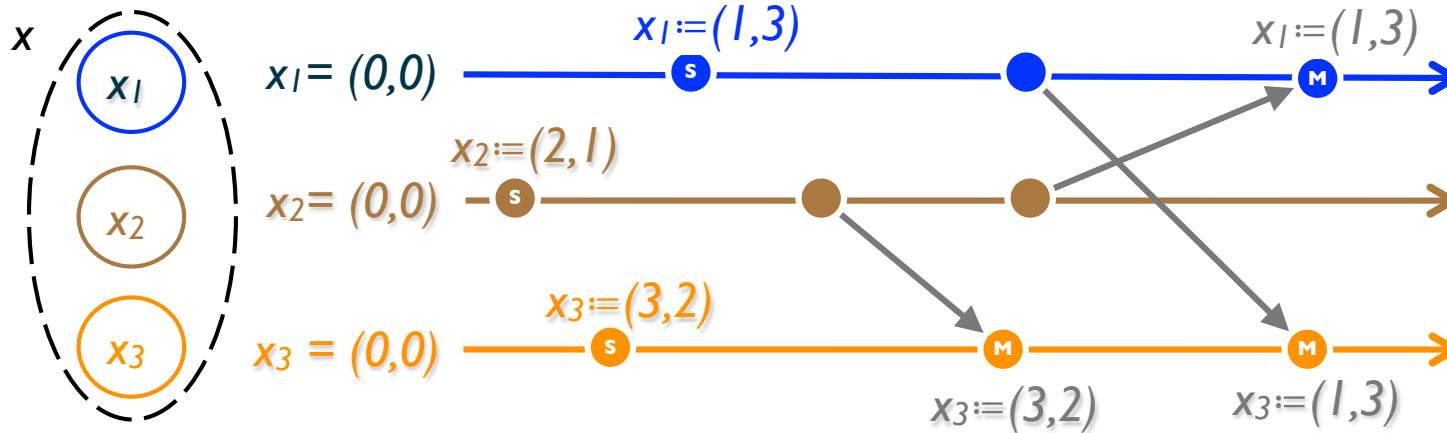
Operations:

- *read: val*
- *assign (val)*
 - Overwrites preceding value

Concurrent *assign*

- Single value, arbitrary choice?
- All concurrent values?

Last Writer Wins Register



- Examples:
- NFS
- shared memory?

CvRDT payload: (*atom value*, *timestamp ts*)

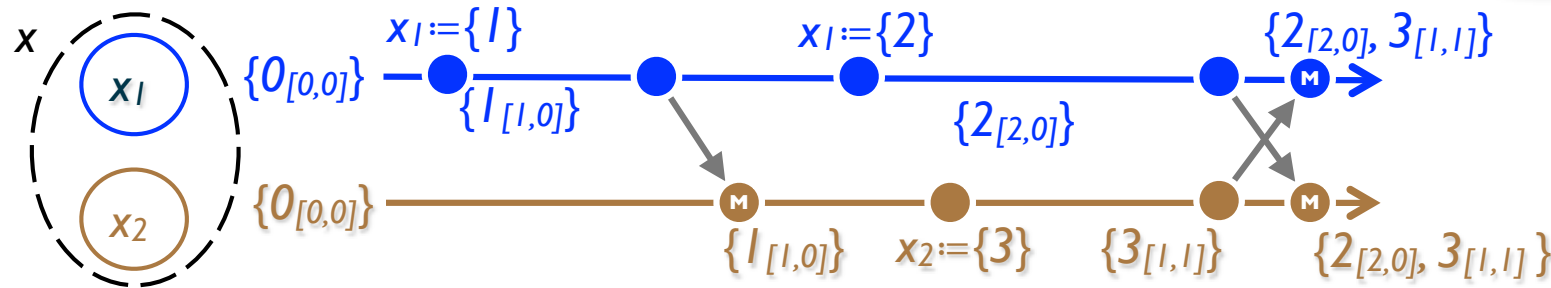
- *assign*: overwrite value, increment *ts*
- Merge takes value with highest timestamp; other is lost
- $x \leq y \stackrel{\text{def}}{=} x.ts \leq y.ts$
- $\text{merge}(x,y) = x.t < y.t ? y : x$

- spec: state-based
- values form a semi-lattice
- no reference to history f_i

- Timestamps implement a *total order*
- Generally $\approx$ real time but could be any total order

MV-Register

- Examples:
- Unison
- CVS, SVN
- Dynamo shopping cart



- A more recent assignment overwrites an older one
- Concurrent assignments are merged by union
- Standard VC merge

$\approx$ LWW-Set Register

- Payload = $\{ (value, VT\ vv) \}$
- *assign*: overwrite value, $vv++$

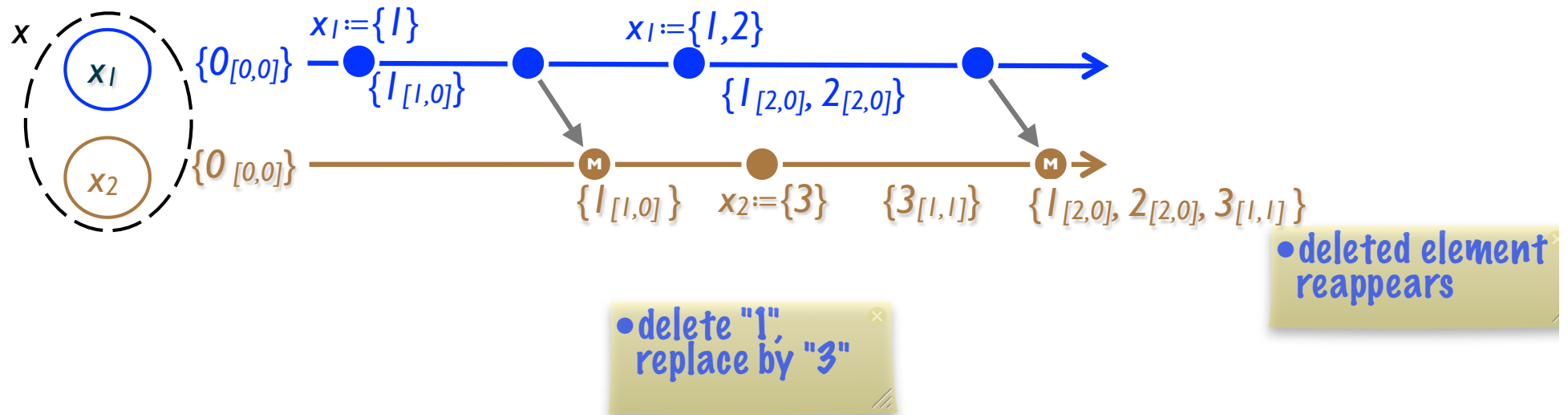
Concurrent updates unioned (no lost updates)

- $merge(X, Y) =$

$$\{ x \in X \mid \nexists y \in Y: x.vv < y.vv \} \cup$$

$$\{ y \in Y \mid \nexists x \in X: x.vv > y.vv \}$$

Bookstore anomalies



“An add operation is never lost. However, deleted items can resurface.” [Dynamo, SOSP 2007]

Preferred approach: Set CRDT

Set

Operations:

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

No duplicates

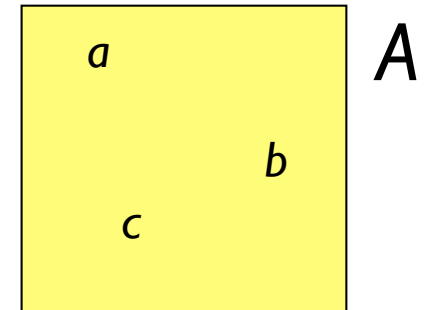
The prototypical CRDT?

- *remove* does not commute with *add*
- Approximations: modify semantics

- union and intersection commute
- not set difference

Grow-only Set, state-based

- Build intuition
- Simple examples
- What state do I need to store and transmit?



Payload = set A
add (atom a)
merge (x,y) = $x \cup y$

add (a)
add (b)
add (c)
add (b)

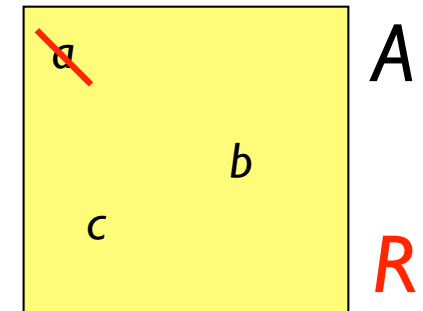
- Assume: state eventually delivered
- Why not remove()?
- Trial and error...
- Hmm, let's move on to something else

2P-Set (state)

- A =added
- R = removed (tombstones)
- Once removed, an element cannot be added again
- Remove has precedence over add (absorbing)

Add, remove: 2P-set

- Payload = (Grow-Set A , Grow-Set R)
- $add(atom\ a)$
 $remove(atom\ a)$ [spre: $a \in A$]
 $lookup(a) = a \in A \wedge a \notin R$
- $x \leq y \stackrel{\text{def}}{=} x.A \subseteq y.A \wedge x.R \subseteq y.R$
- $merge(x,y) = (x.A \cup y.A, x.R \cup y.R)$



$add(a)$
 $add(b)$
 $remove(a)$
 $add(c)$
 $add(b)$
 $add(a)$

• In many distr. sys., uses of Set, add creates a unique element, so this is not a limitation

U-Set = no tombstones

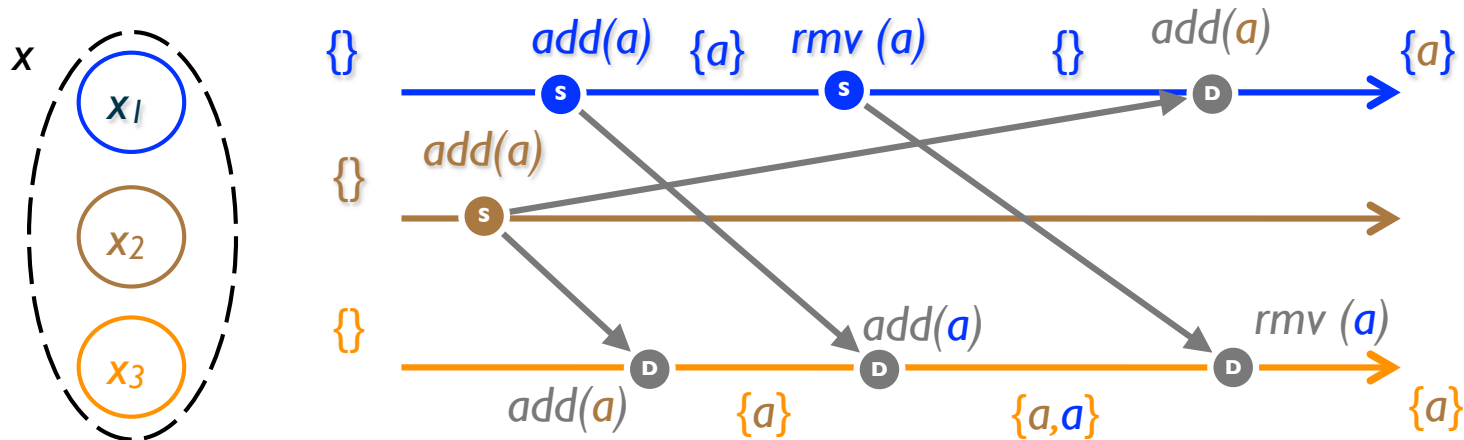
2P-Set

Special, common case: a unique

- Never add again
- No tombstones

Correct shopping cart

Observed-Remove Set (state)



- Can never remove more tokens than exist
- Op order $\Rightarrow$ removed tokens have been previously added

- Payload: Map M : element to 2P-Set of tokens
- Make add unique:
 $add(a) = M.add(a, \text{unique-token})$
- Remove the unique elements observed
 $remove(a) = M.removeAll(a)$
- $lookup(a) = a \in M \wedge a.tokens$ not empty
- $merge(x, y) = \text{merge token sets}$

- Better shopping cart
- What anomalies?

Map

Set of (key, value) pairs

Payload: $S = \{ (k, v), \dots \}$

- $lookup(k) = \{ v: (k, v) \in S \}$
- $add(k, v) = S := S \cup \{ (k, v) \}$
- $remove(k, v) = S := S \setminus \{ (k, v) \}$
- $removeAll(k) = S := S \setminus \{ (k, _) \}$

CRDT approximations

- 2P-Map
- PN-Map
- LWW Map
- Observed-Remove Map

Graph

Graph = (V, E)

where V = set of atoms

$$E \subseteq V \times V$$

$addVertex(v) \rightarrow addEdge(v, w)$

$\rightarrow removeEdge(v, w) \rightarrow removeVertex(v)$

• and similarly
for w

Any of the set-like CRDTs is OK

- e.g. 2P-Set $\Rightarrow$ 2P-Graph

• delay
concurrent
removes

In the general case, cannot enforce global property,
e.g. acyclic

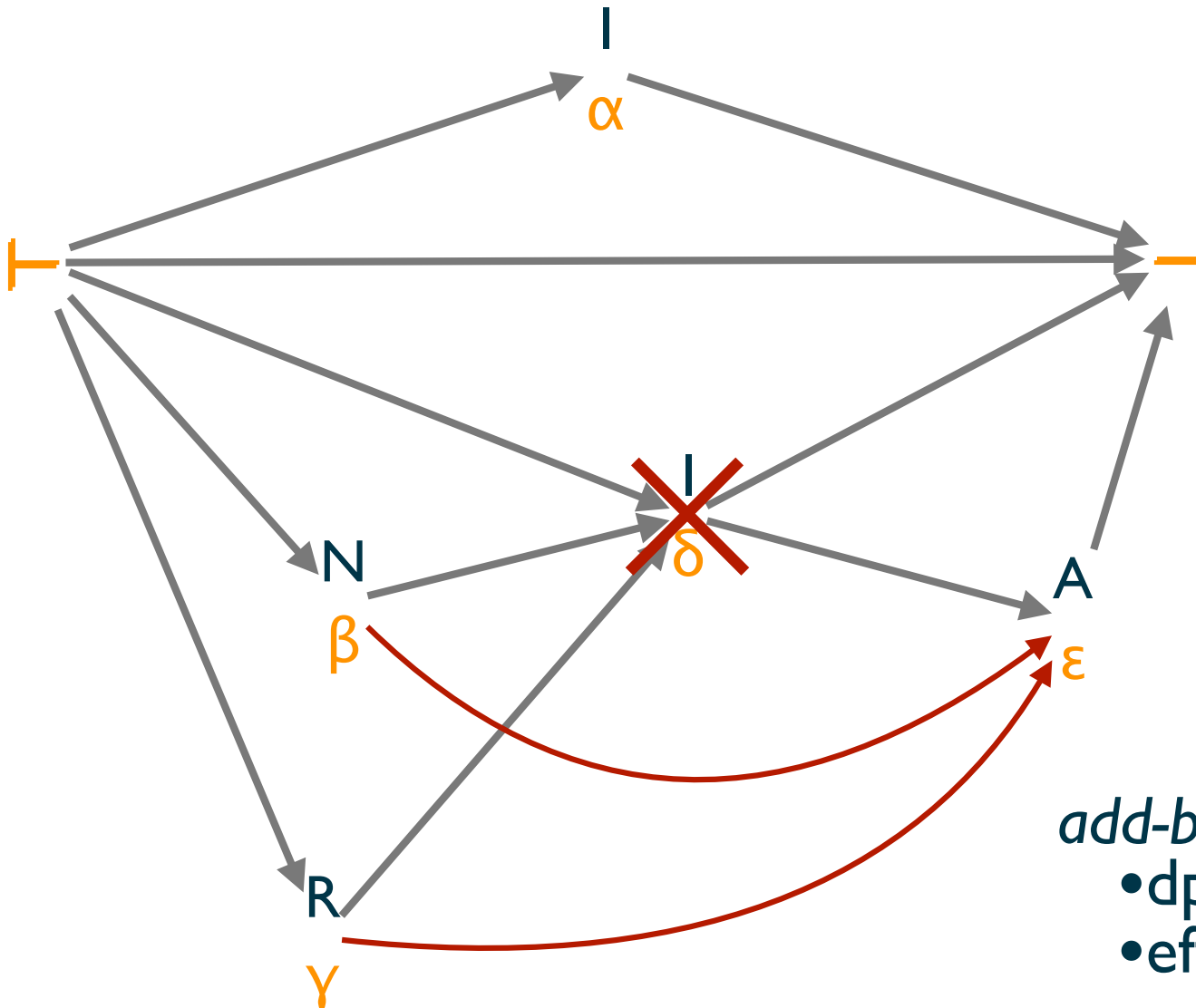
• Counter-
examples
next

GC

Tombstone

- 2P-Set: forbid *add-remove-add*
- Graph: *addEdge(u,v) || removeVertex(u)*
- Discard when all concurrent *addEdge* delivered
 - i.e. when *removeVertex* stable
 - Wu, Bernstein/Golding algorithm
- No consensus
- Not live in presence of crash

Monotonic DAG



- add: between already-ordered elements
- remove: preserves existing order
- Monotonic between remaining elements [restrictive meaning]
- Typical application: concurrent text editing

- Causal order too strong for add
- Too weak for delete

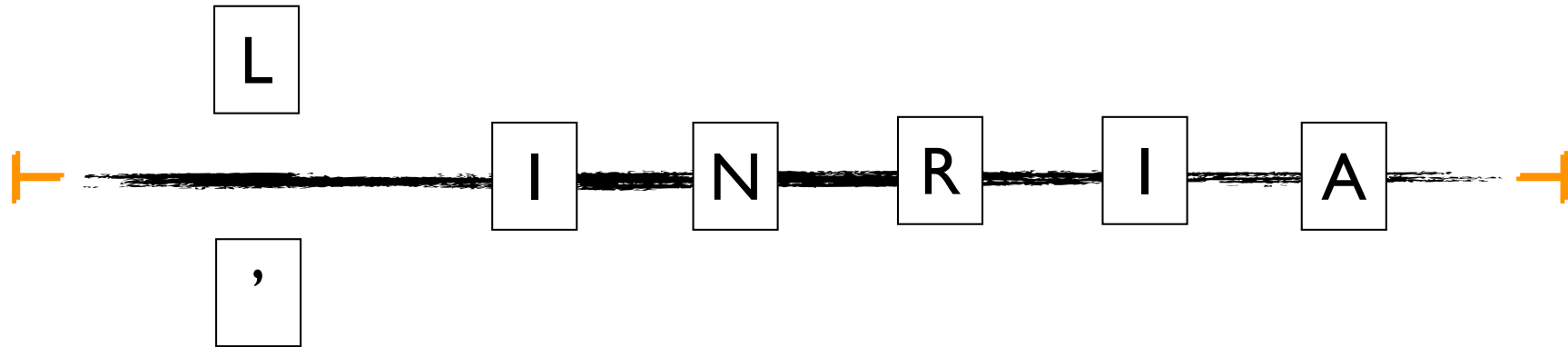
add-between (x, y, z)

- dpre: $x, z \in V \wedge x < z$
- effect: $y \in V \wedge x < y < z$

remove (y)

- effect: $y \notin V \wedge x < z$

Sequence



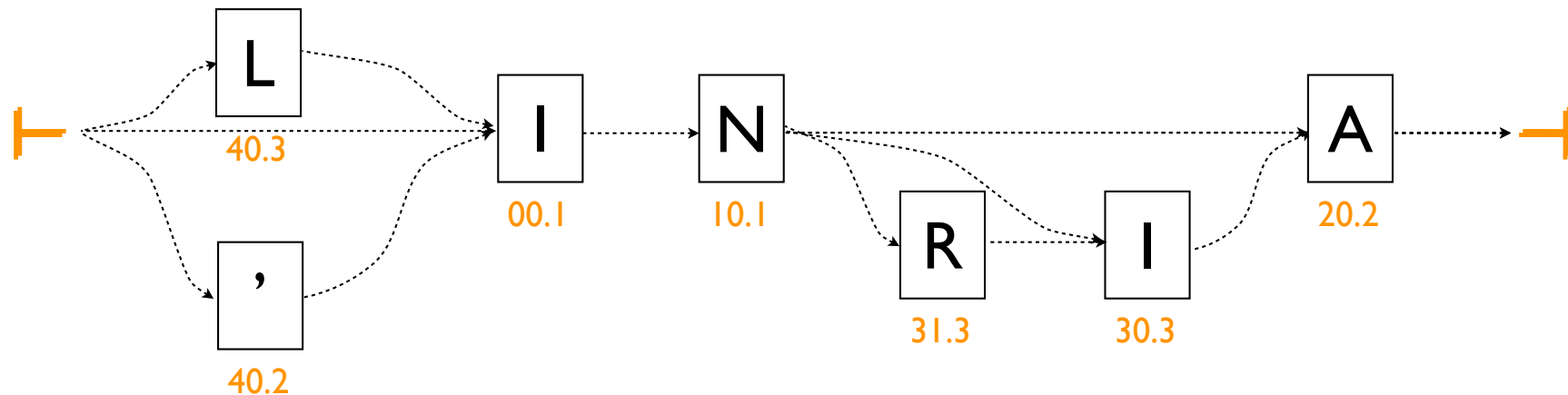
Sequence of elements of type T

- Co-operative edit buffer:
sequence of atoms
- *add-at-location, remove*

Two approaches:

- Linked list
- Continuum

Roh's RGA



Elements of type $(atom\ v, LTS\ ts)$

- Explicit (total order) graph $x < y < z$

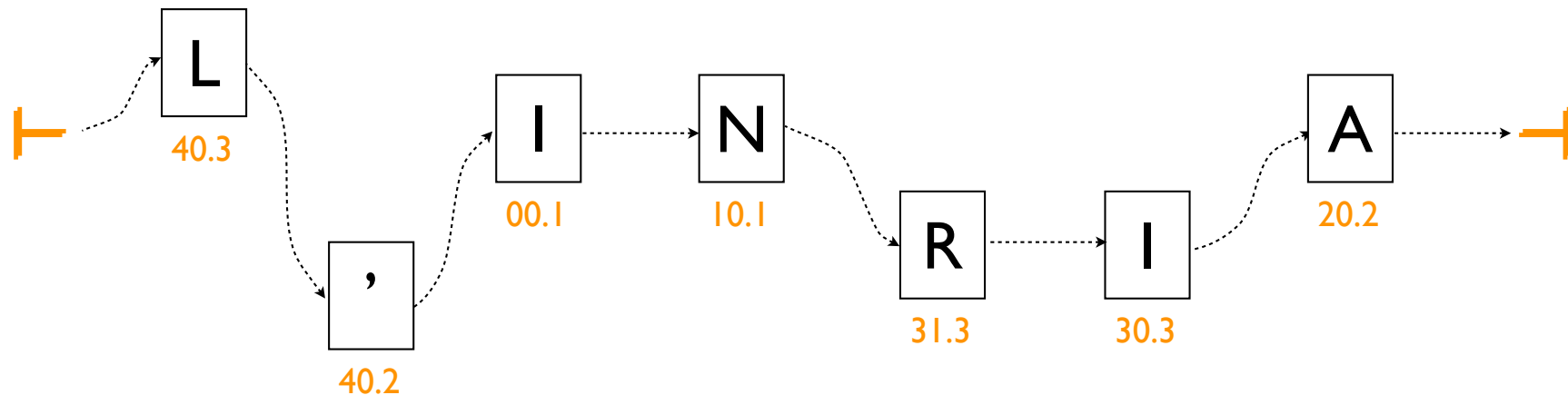
• Lamport
timestamp

add-after (x, y) :

- dpre: $add-after(..., x) \rightarrow add-after(x, ...)$
- Sequential: $add-after(x, y) \rightarrow add-after(x, z)$
 $\Rightarrow y.ts < z.ts \wedge x < z < y$
- Concurrent: $add-after(x, y) \parallel add-after(x, z)$
 $\wedge y.ts < z.ts \Rightarrow x < z < y$

• Concurrent
behaviour
consistent

Roh's RGA



Elements of type $(atom\ v, LTS\ ts)$

- Explicit (total order) graph $x < y < z$

• Lamport
timestamp

add-after (x, y) :

- dpre: $add-after(..., x) \rightarrow add-after(x, ...)$
- Sequential: $add-after(x, y) \rightarrow add-after(x, z)$
 $\Rightarrow y.ts < z.ts \wedge x < z < y$
- Concurrent: $add-after(x, y) \parallel add-after(x, z)$
 $\wedge y.lts < z.lts \Rightarrow x < z < y$

• Concurrent
behaviour
consistent

Continuum



Assign each element a unique real number

- *position*

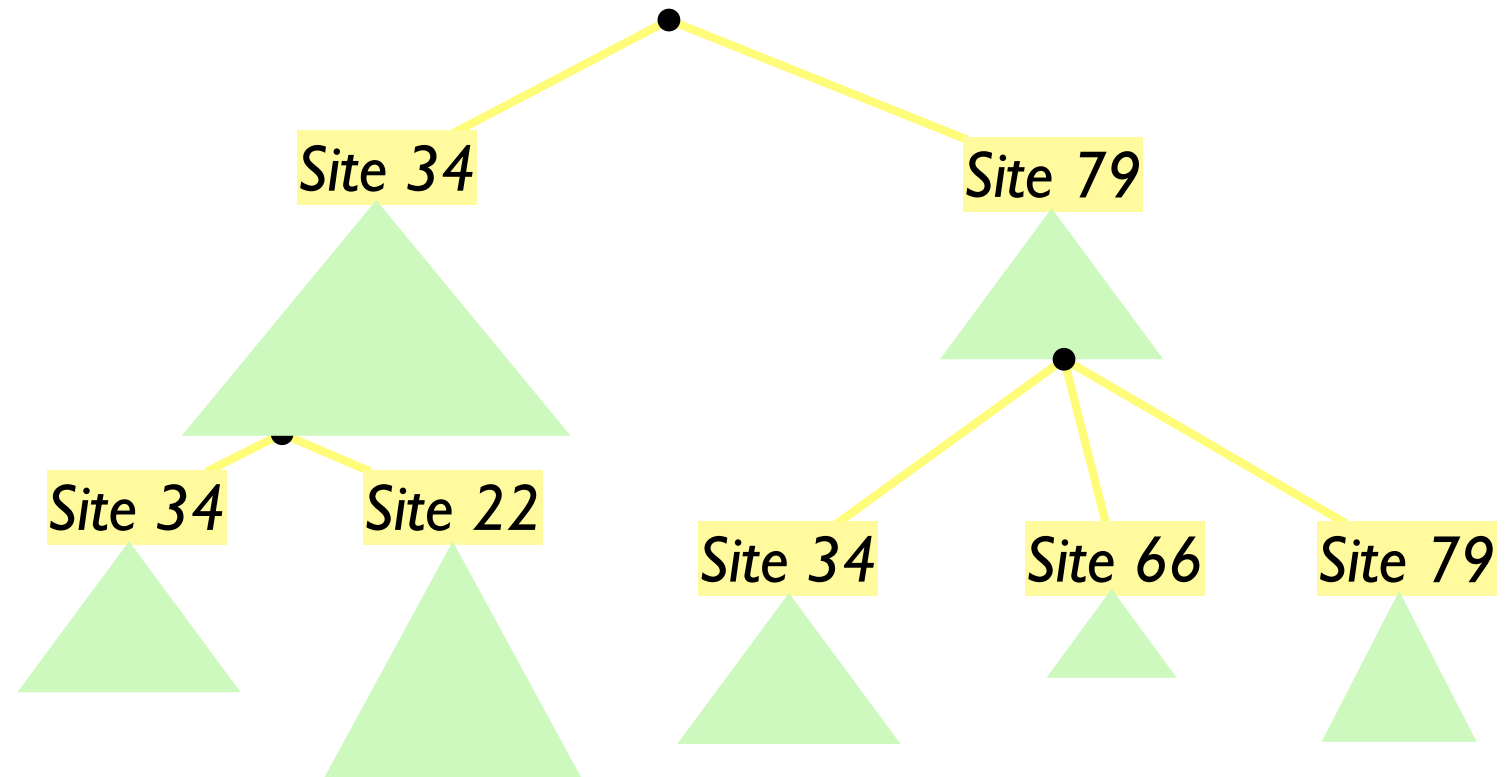
Real numbers not appropriate

- approximate by tree

Layered Treedoc

sparse 8^{64} -
-ary tree

binary tree



Edit: Binary tree

Concurrency: Sparse tree

Rebalance



Tree has nice logarithmic properties

Wikipedia, CVS experiments:

- Lots of removes
- Unbalanced over time

Rebalancing changes IDs:

- Strong synchronisation (commitment)
- In the background
- Liveness not essential
- Core-Nebula: small-scale consensus

Take aways

Principled approach to eventual consistency

Two sufficient conditions:

- State: monotonic semi-lattice
- Operation: commutativity

Useful CRDTs

- Register: Last-Writer-Wins, Multi-Value
- $\approx$ Set: 2P (remove wins), OR (add wins)
- Map $\approx$ Set + Register
- Graph $\approx$ (Set, Set) + $E \subseteq V \times V$
- Monotonic DAG
- Sequence: list, continuum

Future work

CRDT-based cache for cloud

Strong invariants

- $counter \geq 0$
- $graph \in DAG, tree, XML\ schema$

Approaches:

- Restricted problems
- Consensus
- Eventual conformance: diverge + fix, probabilistic guarantees

• exponential
backoff

Quasi-CRDTs

- Common operations commute
- Occasional consensus

CRDTs for cloud computing

ConcoRDanT: ANR 2010–2013

- Systematic study, explore design space
- Characterise invariants
- Library of data types: multilog, K-V store
+ composition

When consensus required:

- Mix commutative / non-commutative semantics
- Move off critical path, non-critical ops
- Speculation + conflict resolution

