ConcoRDanT

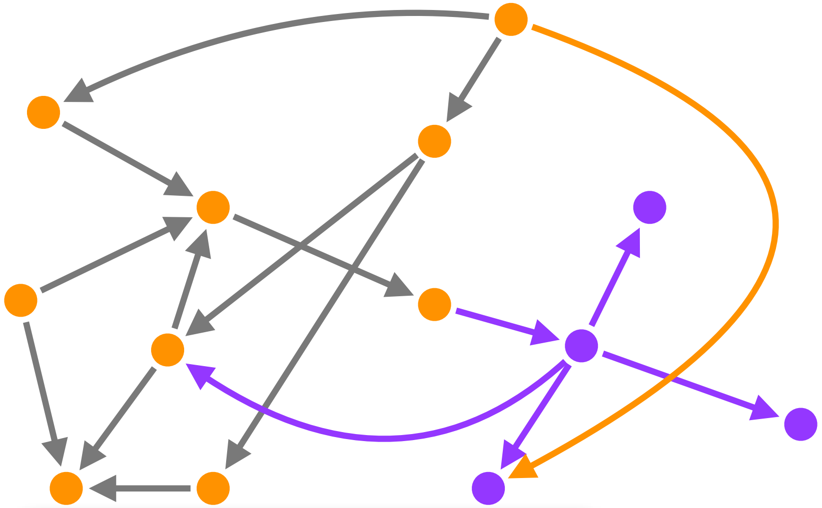
Strong Eventual Consistency and CRDTs

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

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche PARIS - ROCQUENCOURT

LIP6
Dpt : Réseaux et Systèmes Répartis

Large-scale replicated data structures



- Large, dynamic graph
- Incremental, parallel, asynchronous:
  - updates
  - processing

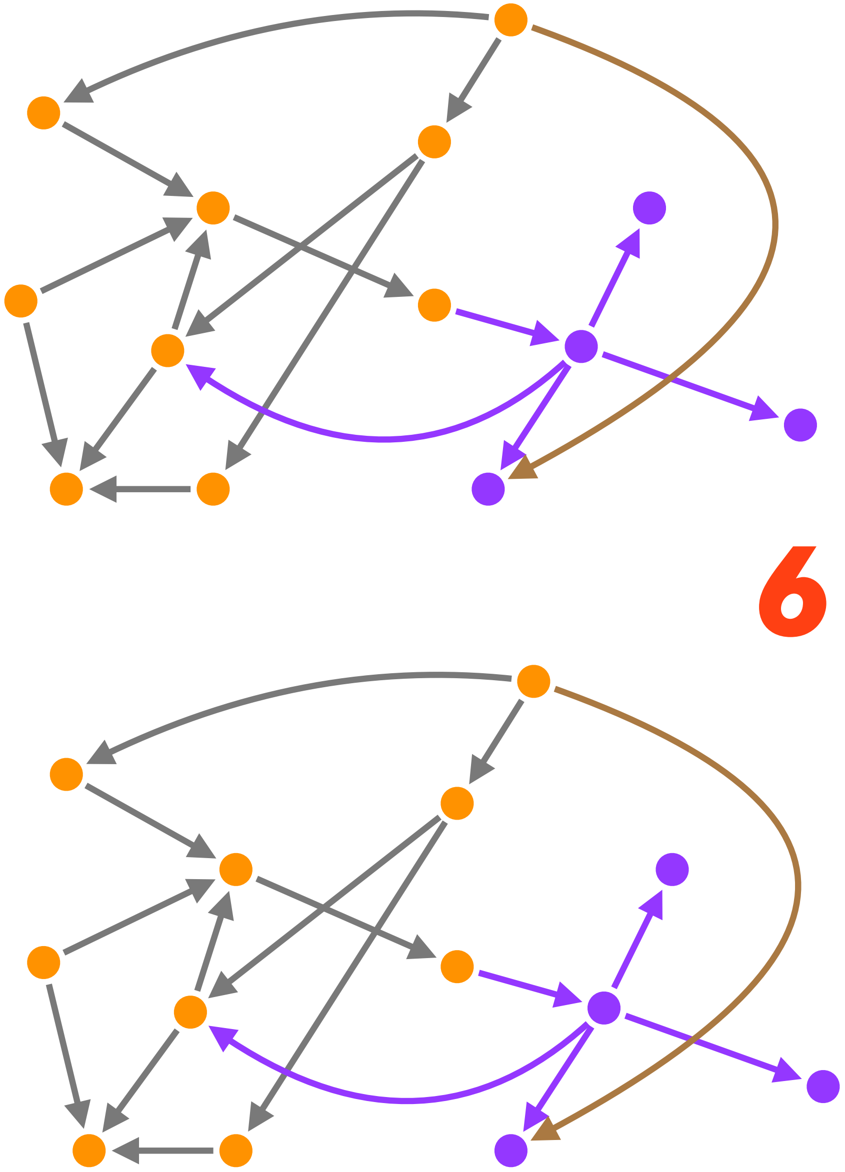
Wish list:

- Mutable
- Incremental
- Fast ⇒ parallel, asynch
- Fault tolerant

Eventual Consistency

- Principles?

Strong consistency



Preclude conflict: Replicas update in same total order

Any deterministic object

Consensus

- Serialisation bottleneck
- Tolerates < $n/2$ faults

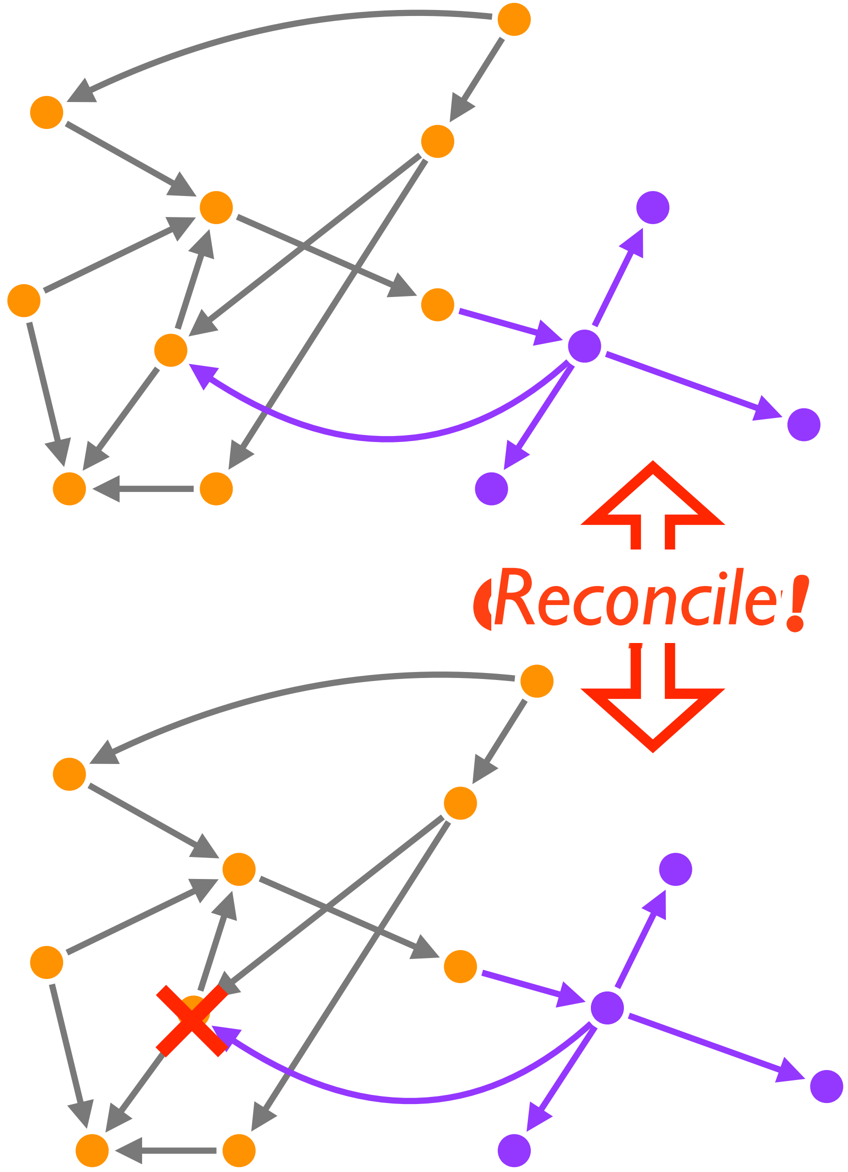
- Simultaneous N-way agreement

Sequential, linearisable…

Universal

- Very general
- Correct
- Doesn't scale

Eventual Consistency



Update local + propagate

- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict

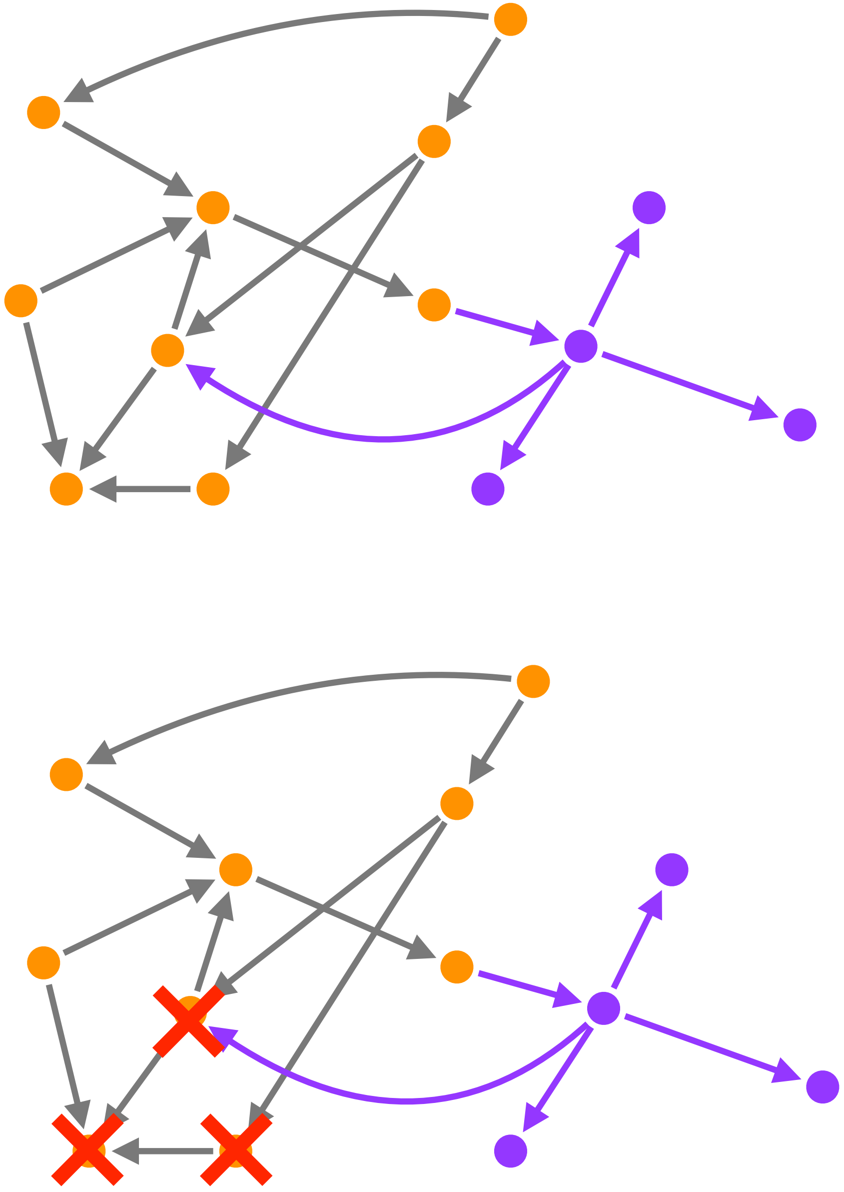
- Arbitrate
- Roll back

- Availability ++
- Parallelism++
- Latency --
- Complexity ++
- Consensus (in background)
- Designed for human collaboration

Consensus moved to background

Strong Eventual Consistency

- Available, responsive
- More parallelism
- No conflicts
- No rollback



Update local + propagate

- No synch
- Expose intermediate state
- Eventual, reliable delivery

No conflict

- Unique outcome of concurrent updates

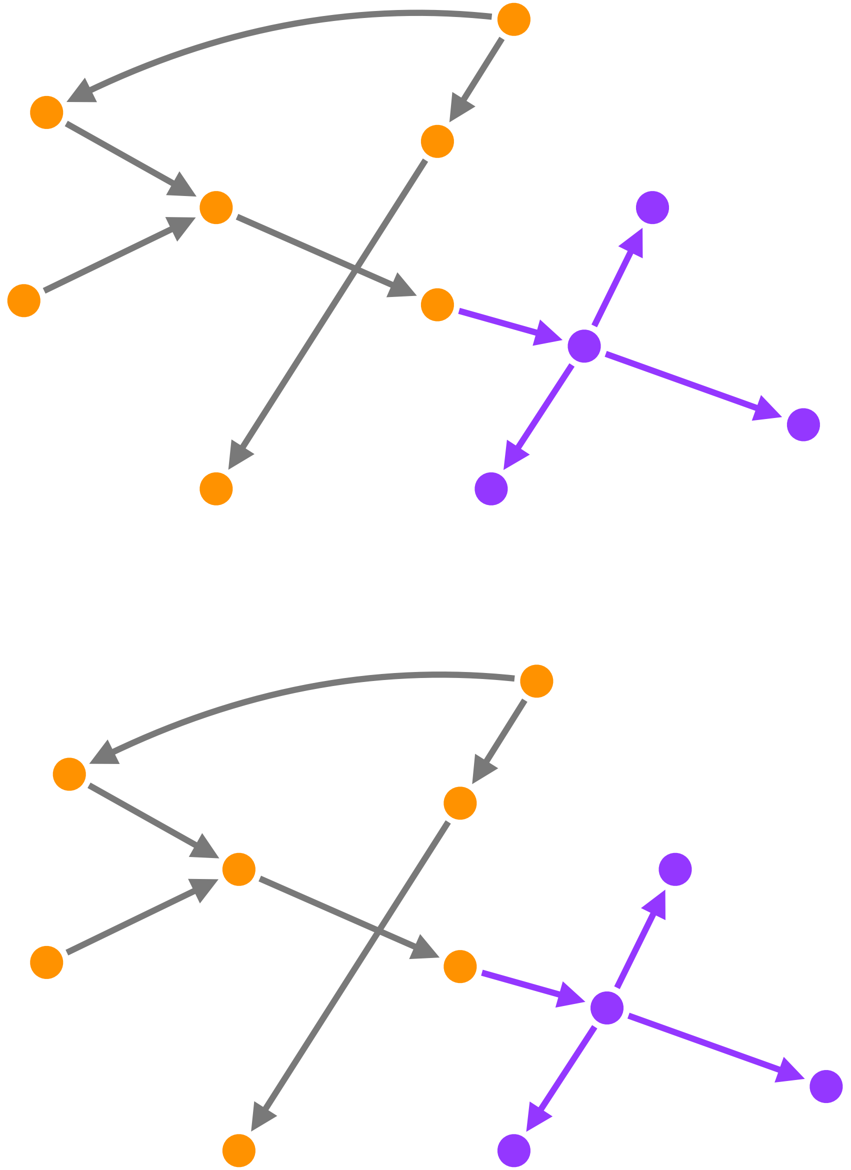
No consensus: ≤ n-1 faults

Not universal

Fast, responsive

Solves the CAP problem

Strong Eventual Consistency



Eventual delivery: An update executed at some correct replica eventually executes at all correct replicas

Termination: All update executions terminate

Strong Convergence: Correct replicas that have executed the same updates have equivalent state

- No conflicts
- No rollback
- No consensus
- Limited

Conflict-free Replicated Data Types (CRDTs)

Intuition:

- *Conflict resolution* requires synchronisation
- Conflict-freedom satisfies SEC

⇒ Design data types with no conflicts

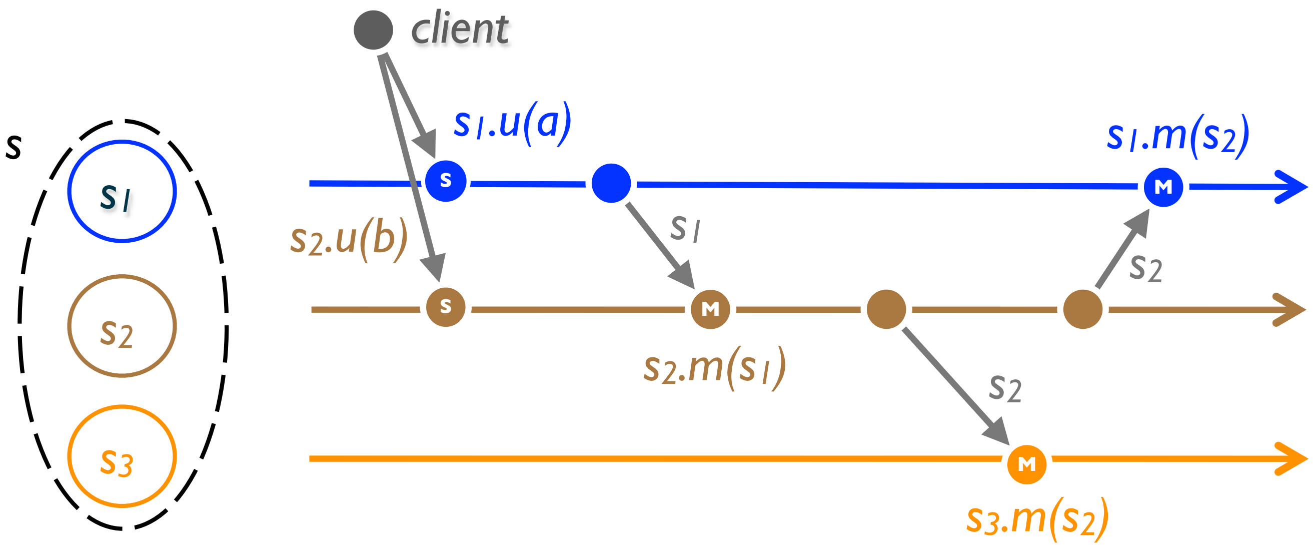
CRDTs

- Available, fast
- Reconcile scalability + consistency

Simple sufficient conditions

- Principled, correct

State-based replication



Local at source $s_1.u(a)$, $s_2.u(b)$, …

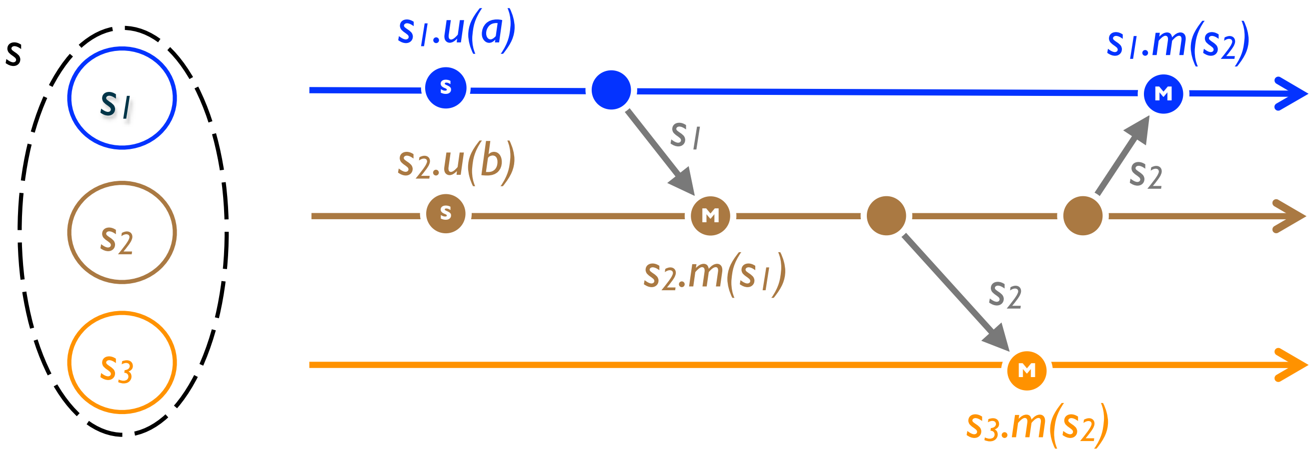
- Compute
- Update local payload

Convergence:

- Episodically: send s_i payload
- On delivery: *merge payloads m*

- merge two valid states
- produce valid state
- no historical info available
- Inefficient if payload is large

State-based: monotonic semi-lattice ⇒ CRDT



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

If

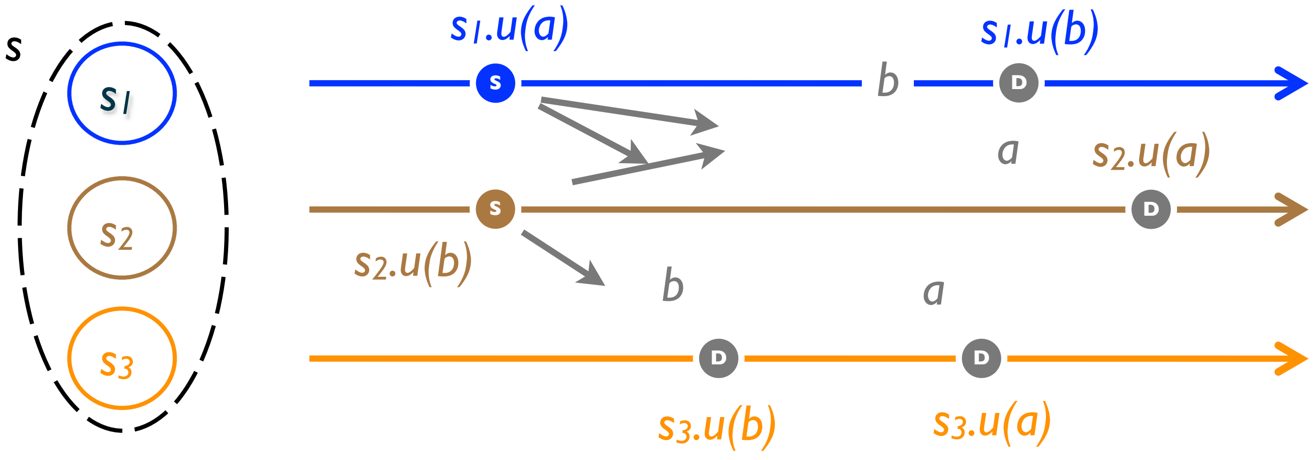
- payload type forms a <u>semi-lattice</u>
- updates are increasing
- *merge* computes Least Upper Bound

then replicas converge to LUB of last values

- no reference to history

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

Operation-based replication



- push to all replicas eventually
- push small updates
  - more efficient than state-based

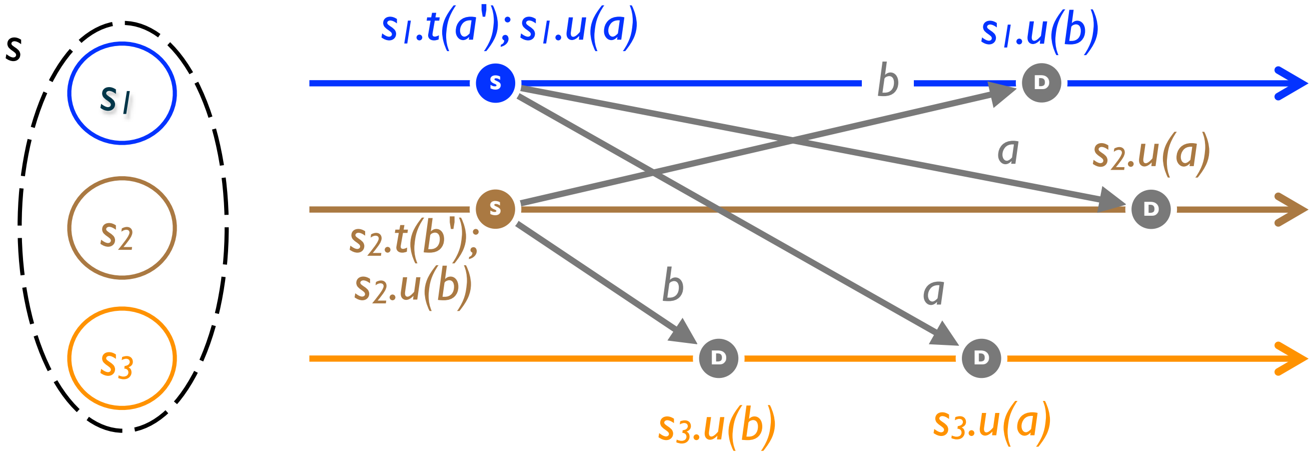
At source:

- prepare
- broadcast to all replicas

Eventually, at all replicas:

- update local replica

Operation-based replication



- push to all replicas eventually
- push small updates
  - more efficient than state-based

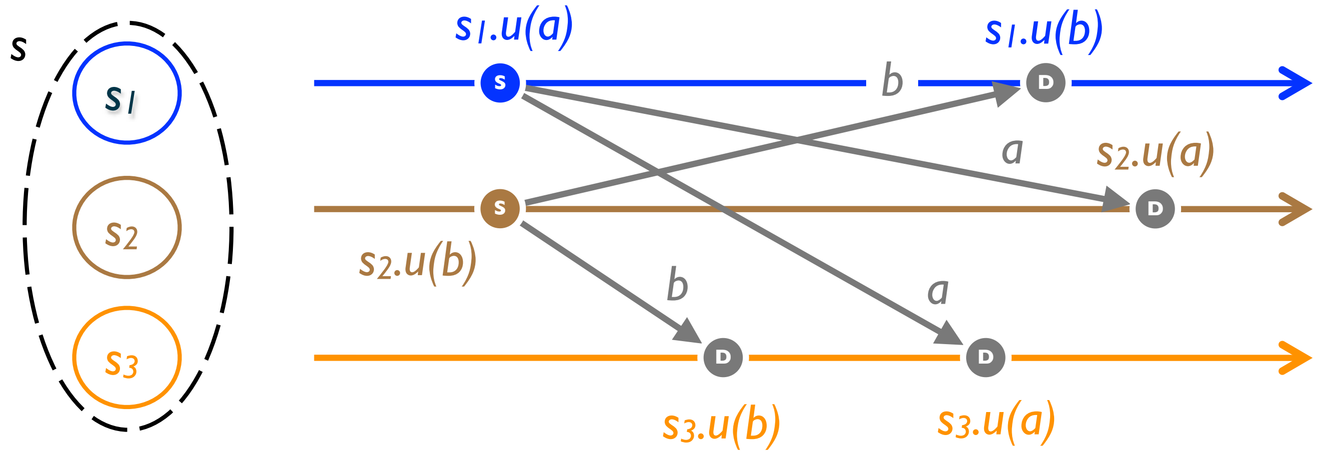
At source:

- prepare
- broadcast to all replicas

Eventually, at all replicas:

- update local replica

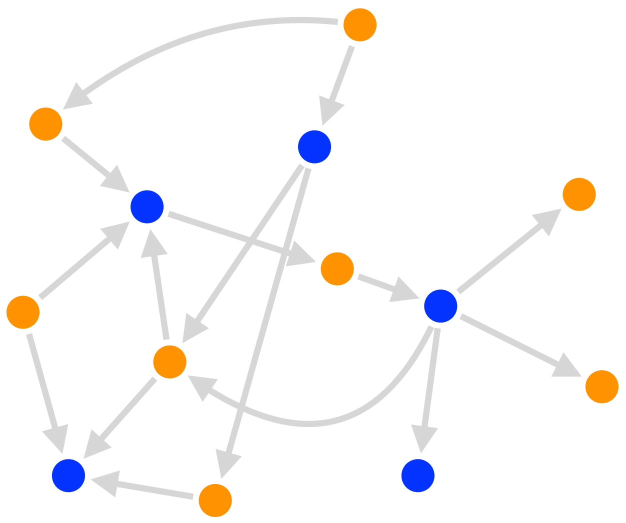
Op-based: commute ⇒ CRDT



- Delivery order ≃ ensures downstream precondition
- happened-before or weaker

If: • (*Liveness*) all replicas execute all operations in delivery order
• (*Safety*) concurrent operations all commute
Then: replicas converge

Composition and sharding



A composition of independent CRDTs is a CRDT

Very large objects
- Independent *shards*
- Static: hash

- (Dynamic: requires consensus to rebalance)

Statically-Sharded CRDT
- Each shard is a CRDT
- Update: single shard
- No cross-object invariants

The challenge:

What interesting objects can we design with no synchronisation whatsoever?

Portfolio of CRDTs

Register
- Last-Writer Wins
- Multi-Value

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

Map
- Set of Registers

Counter
- Unlimited
- Non-negative

Graphs
- **Directed**
- Monotonic DAG
- Edit graph

Sequence
- Edit sequence

Multi-master counter

Increment / decrement

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

- like vector clock

Multi-master counter

Increment / decrement

- Payload: P = [int, int, …], N = [int, int, …]
- $value() = \sum_i P[i] - \sum_i N[i]$
- $increment() = P[MyID]{+}{+}$
- $decrement() = N[MyID]{+}{+}$
- $merge(s,s') =$
  $$s \sqcup s' = ([\ldots, max(s.P[i], s'.P[i]), \ldots]_i, [\ldots, max(s.N[i], s'.N[i]), \ldots]_i)$$
- Positive or negative

- like vector clock

- can't maintain global invariant such as s>0

Set design alternatives

- linearisable: sequential order
- equivalent to real-time order
- Requires consensus

Sequential specification:

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

$\{true\}$ add(e) || remove(e) $\{????\}$

- ~~linearisable?~~
- error state?
- last writer wins?
- add wins?
- remove wins?

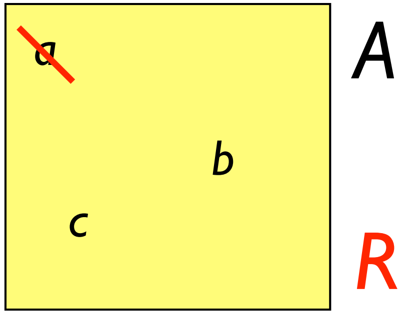
2P-Set

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

Payload = (*Grow-Set A*, *Grow-Set R*)

- $add(e) = A := A \cup \{e\}$
- $remove(e) = e \in A \; ? \; R := R \cup \{e\}$
- $lookup(e) = e \in A \land e \notin R$
- $s \leq s' \stackrel{\text{def}}{=} s.A \subseteq s'.A \land s.R \subseteq s'.R$
- $merge(s,s') = (s.A \cup s'.A, s.R \cup s'.R)$

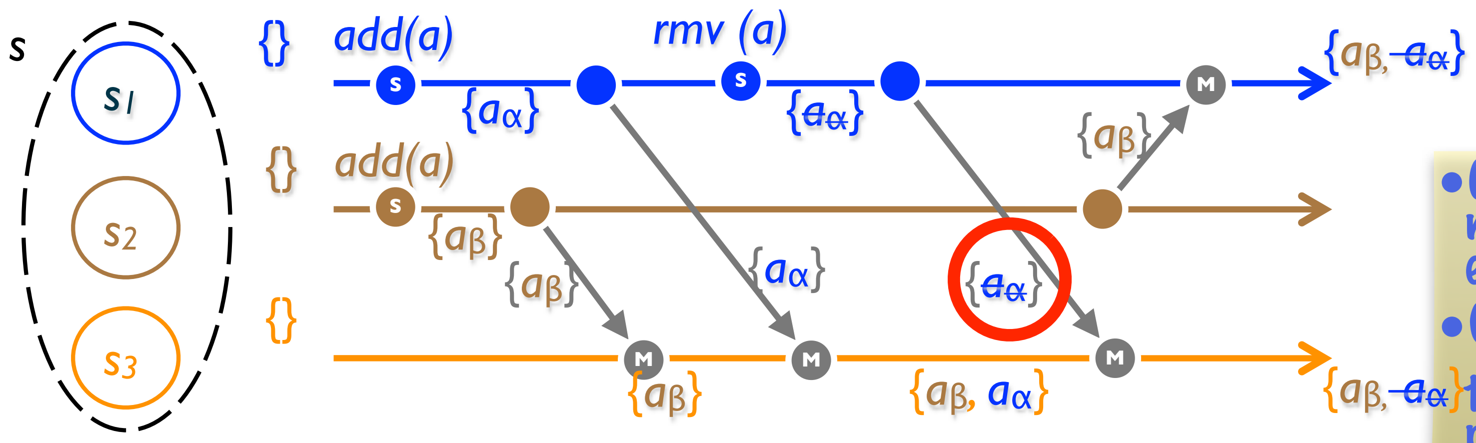
$\{true\}$ add(e) || remove(e) $\{e \notin S\}$



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

Observed-Remove Set



- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

- Payload: added, removed (*element, unique-token*)
  $add(e) = A := A \cup \{(e, \alpha)\}$
- Remove: all unique elements observed
  $remove(e) = R := R \cup \{ (e, -) \in A\}$
- $lookup(e) = \exists\ (e, -) \in A \setminus R$
- $merge\ (S, S') = (A \cup A', R \cup R')$
- $\{true\}$ add(e) || remove(e) $\{e \in S\}$

OR-Set

Set: solves Dynamo Shopping Cart anomaly

Optimisations:

- No tombstones
- Operation-based approach
- Snapshots
- Sharded

OR-Set + Snapshot

Read consistent snapshot

- Despite concurrent, incremental updates

Unique token = time (vector clock)

- α = Lamport *(process i, counter t)*
- UIDs identify snapshot version
- Snapshot: vector clock value
- Retain tombstones until not needed

$lookup(e, t) = \exists\ (e, i, t') \in A : t' > t \wedge \nexists\ (e, i, t') \in R : t' > t$

OR-Set + Snapshot (2)

- A_i = added elements + unique timestamp
- R_i = tombstones + timestamp

- Payload: vector clock V_i
  set $A_i = \{ (e, j, t), \ldots \}$
  set $R_i = \{ (e, j, c, j', t'), \ldots \}$
- $add(e): V_i[i]{+}{+}; A_i \coloneqq A_i \cup \{ (e, i, V_i[i]) \}$
- $remove(e): V_i[i]{+}{+}; R_i \coloneqq R_i \cup \{ (e, j, t, i, V_i[i]) \}$
- $merge(V,A,R)$:
  $\forall j, V_i[j] \coloneqq max(V_i[j], V[j]); A_i \coloneqq A_i \cup A; R_i \coloneqq R_i \cup R$
- $lookup(e,V)$: $(e,j,c) \in A_i \wedge (e,j,c,-,-) \notin R_i \wedge V[j] \geq c$
  $\vee\ (e,j,c,j',c') \in R_i \wedge V[j] \geq c \wedge V[j'] < c$

- lookup w.r.t. a snapshot vector V
- e in set if added, and not removed, and within snapshot
- or if added before snapshot and removed after snapshot

Graph design alternatives

Graph = (V, E) where $E \subseteq V \times V$

Sequential specification:

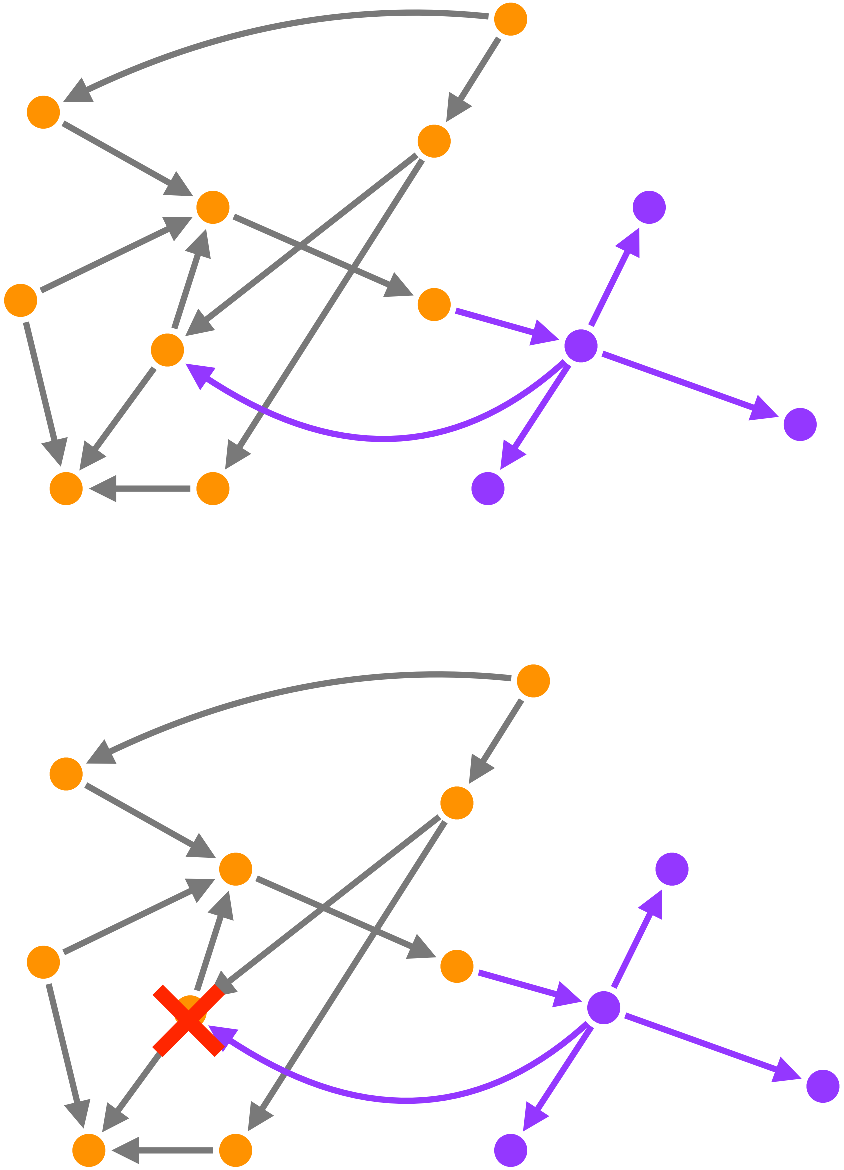
- $\{v, v' \in V\}$ addEdge(v,v') $\{\ldots\}$
- $\{\nexists (v, v') \in E\}$ removeVertex(v) $\{\ldots\}$

Concurrent: removeVertex(v') || addEdge(v,v')

- ~~linearisable?~~
- last writer wins?
- addEdge wins?
- removeVertex wins?
- etc.

- for our Web Search Engine application, removeVertex wins
- Do not check precondition at add/remove

Directed Graph



Payload = OR-Set V, OR-Set E

Updates add/remove to V, E

- *addVertex(v)*, *removeVertex(v)*
- *addEdge(v,v')*, *removeEdge(v,v')*

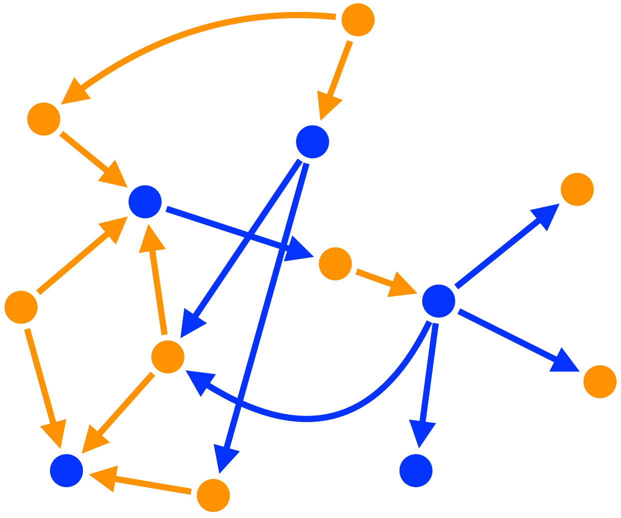
Do not enforce invariant a priori

- lookupEdge(v,v') = $(v,v') \in E$
  $\land\ v \in V \land v' \in V$

removeVertex(v') || *addEdge(v,v')*

- removeVertex wins

Graph + shards + snapshots



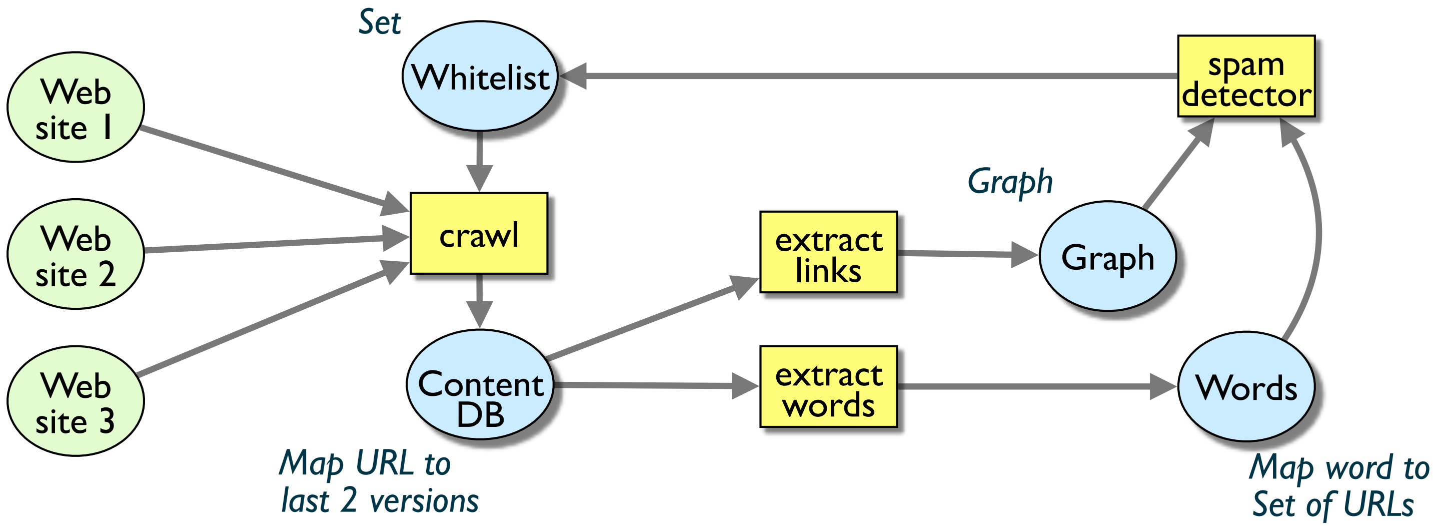
Snapshot

- see OR-Set

Sharding

- See OR-Set
- Do not enforce invariant *a priori*
  lookupEdge$(v,v') = (v,v') \in E$
  $\wedge\ v \in V \wedge v' \in V$

CRDT + dataflow



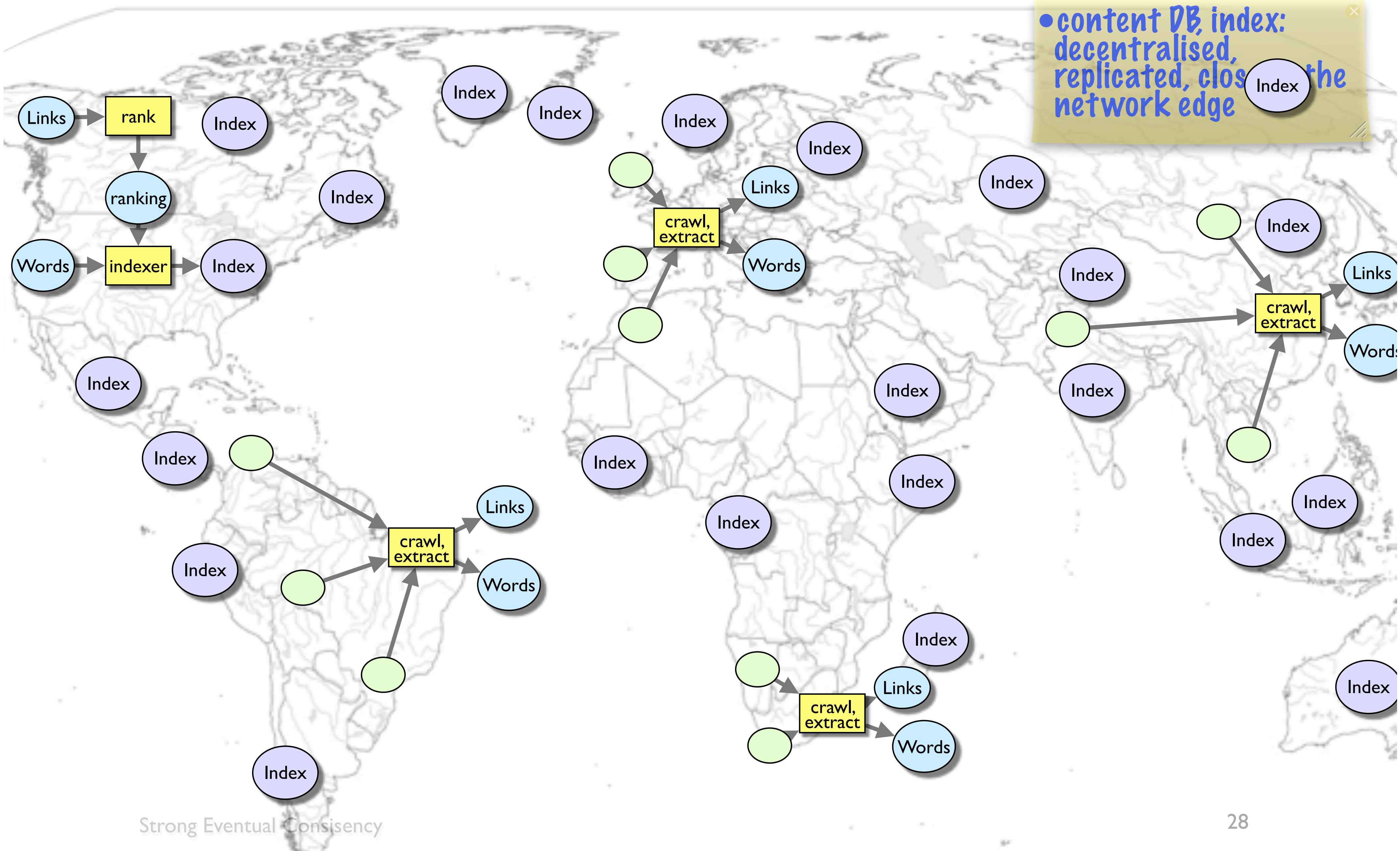
Incremental, asynchronous processing

- Replicate, shard CRDTs near the edge
- Propagate updates ≈ dataflow
- Throttle according to QoS metrics (freshness, availability, cost, etc.)

Scale: sharded

Synchronous processing: snapshot, at centre

Thought experiment (2)



- content DB, index: decentralised, replicated, clos the network edge

Contributions

Strong Eventual Consistency (SEC)

- A solution to the CAP problem
- Formal definitions
- Two sufficient conditions
- Strong equivalence between the two
- SEC shown incomparable to sequential consistency

CRDTs

- integer vectors, counters
- sets
- graphs