Microsoft Research

turning ideas into reality

Strong Eventual Consistency and CRDTs

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

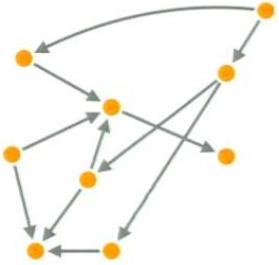
INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche PARIS - ROCQUENCOURT

LIP6
Dpt : Réseaux et Systèmes Répartis

Large-scale replicated data structures

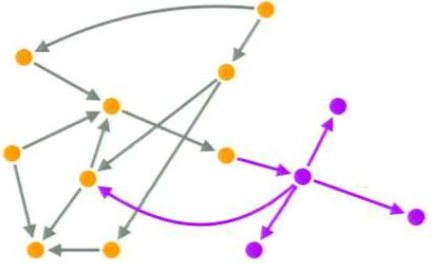


Replication + eventual consistency + fast + simple

Conflict-free objects = no synchronisation whatsoever

Is this practical?

Large-scale replicated data structures

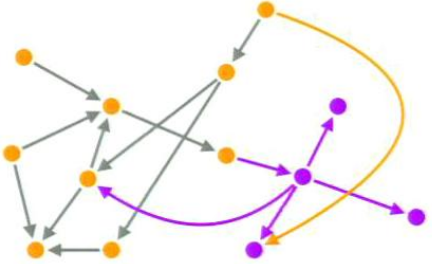


Replication + eventual consistency + fast + simple

Conflict-free objects = no synchronisation whatsoever

Is this practical?

Large-scale replicated data structures



Replication + eventual consistency + fast + simple

Conflict-free objects = no synchronisation whatsoever

Is this practical?

Contributions

Strong Eventual Consistency (SEC)

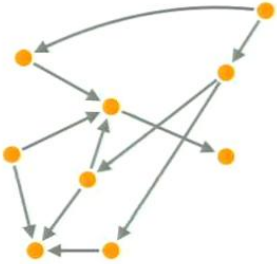
- A solution to the CAP problem
- Formal definitions
- Two sufficient conditions
- Strong equivalence between the two
- Incomparable to sequential consistency

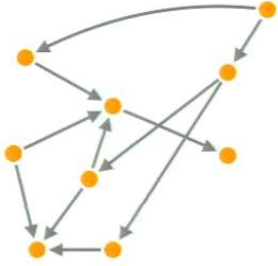
CRDTs = object types satisfying conditions

- Counters
- Set
- Directed graph

Consistency

Strong consistency





Preclude conflict: Replicas update in same total order

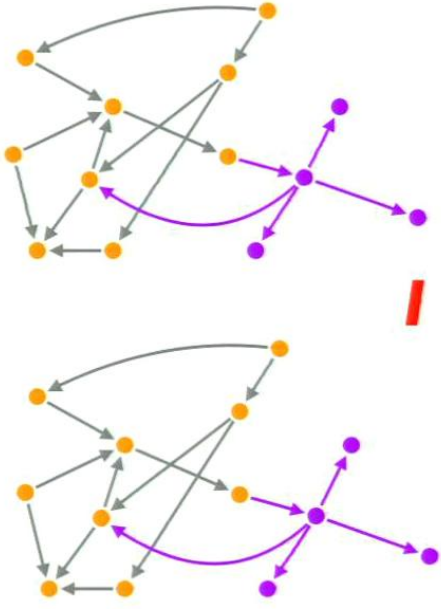
Any deterministic object

Consensus

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

Sequential, linearisable…

Strong consistency



Preclude conflict: Replicas update in same total order

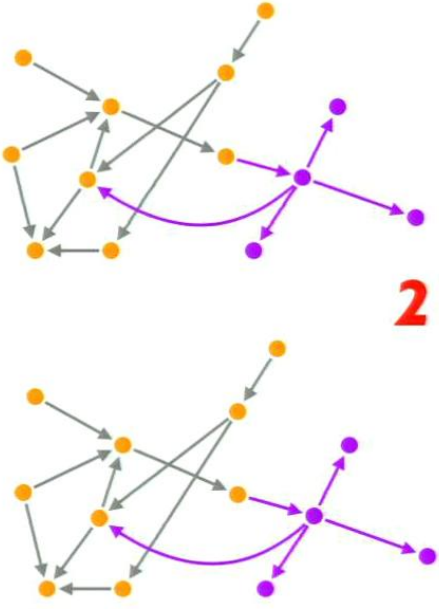
Any deterministic object

Consensus

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

Sequential, linearisable…

Strong consistency



Preclude conflict: Replicas update in same total order

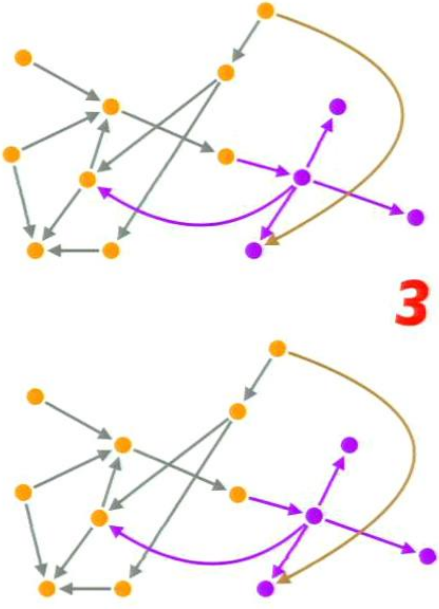
Any deterministic object

Consensus

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

Sequential, linearisable…

Strong consistency



Preclude conflict: Replicas update in same total order

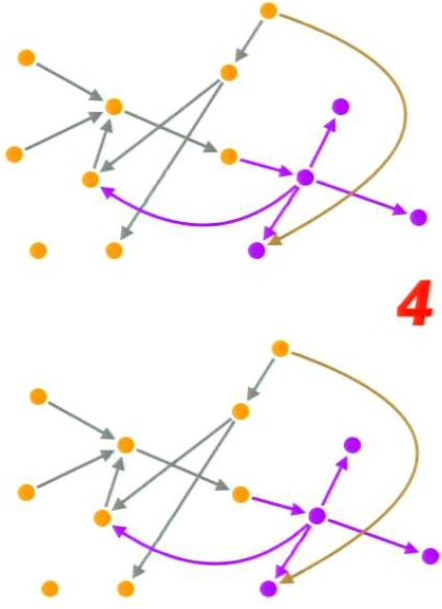
Any deterministic object

Consensus

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

Sequential, linearisable…

Strong consistency



Preclude conflict: Replicas update in same total order

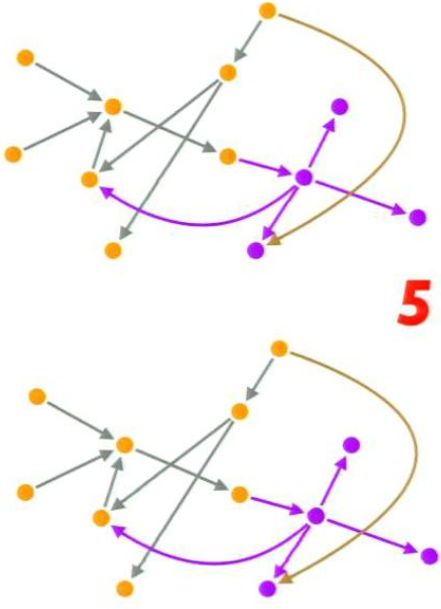
Any deterministic object

Consensus

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

Sequential, linearisable…

Strong consistency



Preclude conflict: Replicas update in same total order

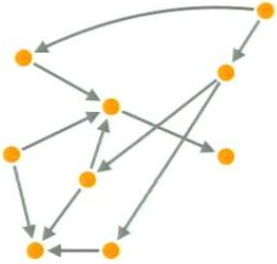
Any deterministic object

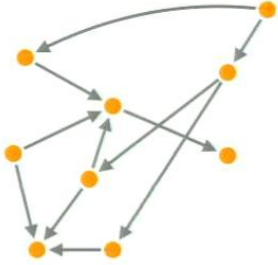
Consensus

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

Sequential, linearisable…

Eventual Consistency



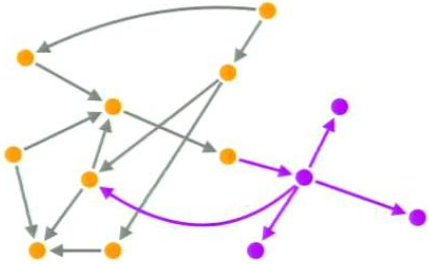


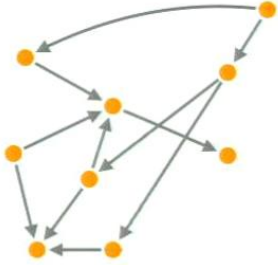
Update local + propagate
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict
- Arbitrate
- Roll back

Consensus moved to background

Eventual Consistency



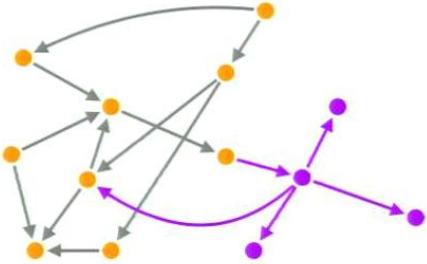


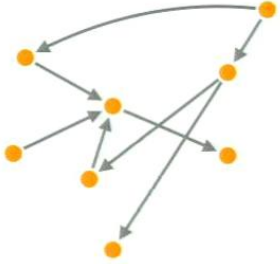
Update local + propagate
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict
- Arbitrate
- Roll back

Consensus moved to background

Eventual Consistency



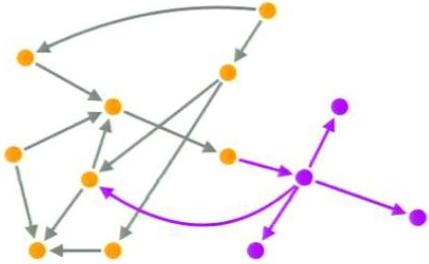


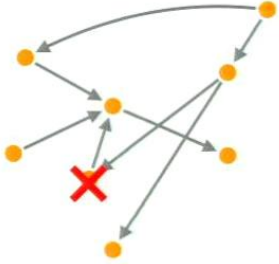
Update local + propagate
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict
- Arbitrate
- Roll back

Consensus moved to background

Eventual Consistency



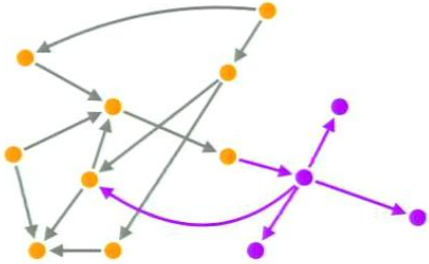


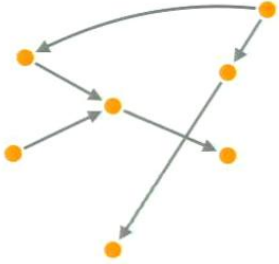
Update local + propagate
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict
- Arbitrate
- Roll back

Consensus moved to background

Eventual Consistency



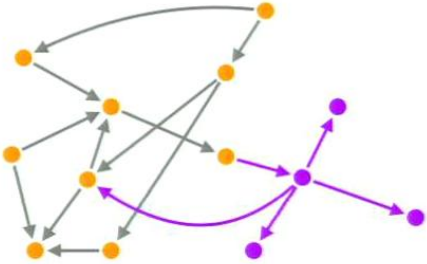


Update local + propagate
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

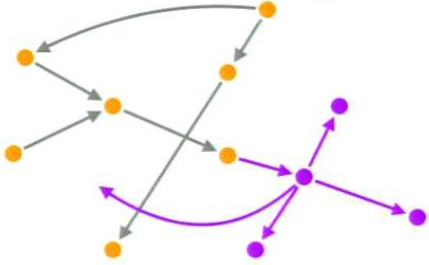
On conflict
- Arbitrate
- Roll back

Consensus moved to background

Eventual Consistency



Conflict!

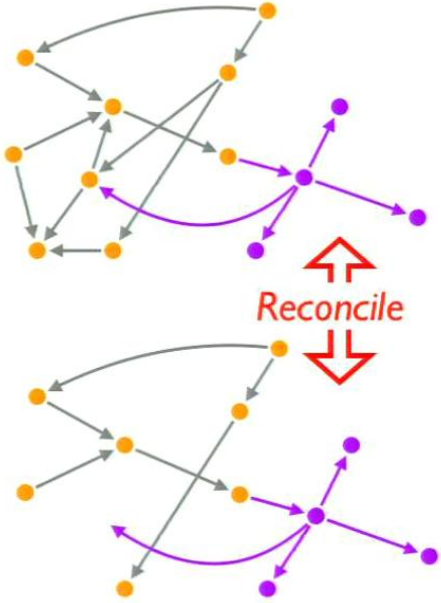


Update local + propagate
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict
- Arbitrate
- Roll back

Consensus moved to background

Eventual Consistency

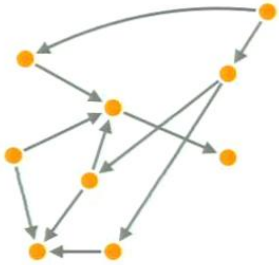


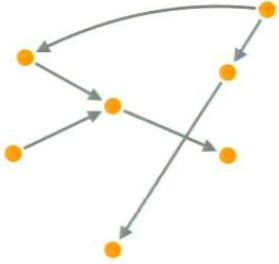
Update local + propagate
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict
- Arbitrate
- Roll back

Consensus moved to background

Eventual Consistency



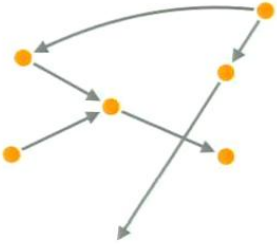


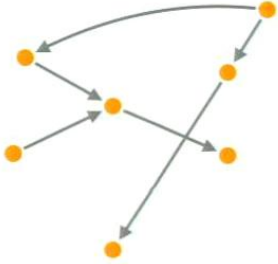
Update local + propagate
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict
- Arbitrate
- Roll back

Consensus moved to background

Eventual Consistency



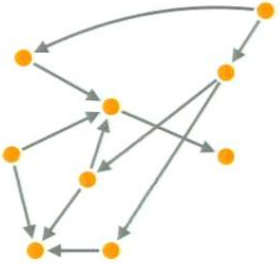


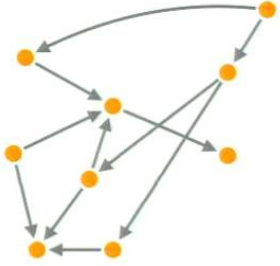
Update local + propagate
- No foreground synch
- Expose tentative state
- Eventual, reliable delivery

On conflict
- Arbitrate
- Roll back

Consensus moved to background

Strong Eventual Consistency





Update local + propagate

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

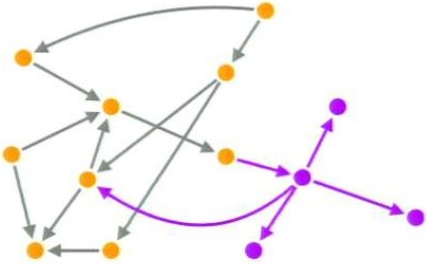
No conflict

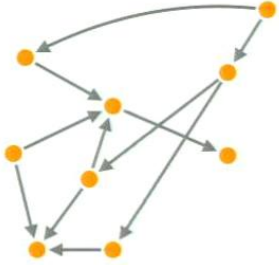
- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

Strong Eventual Consistency





Update local + propagate

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

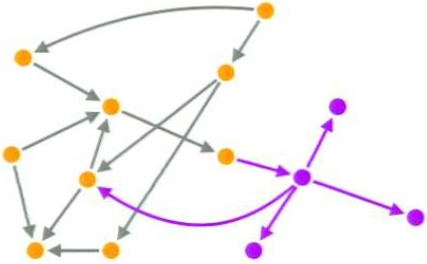
No conflict

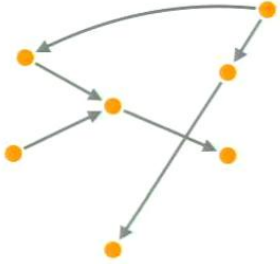
- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

Strong Eventual Consistency





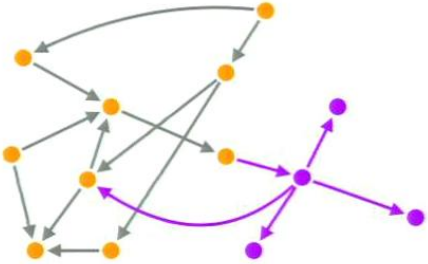
Update local + propagate
- No synch
- Expose intermediate state
- Eventual, reliable delivery

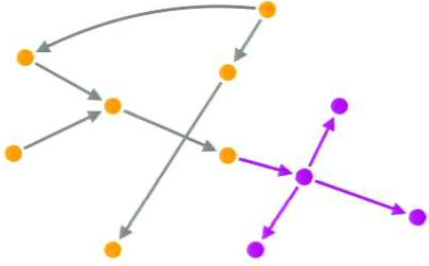
No conflict
- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

Strong Eventual Consistency





Update local + propagate

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

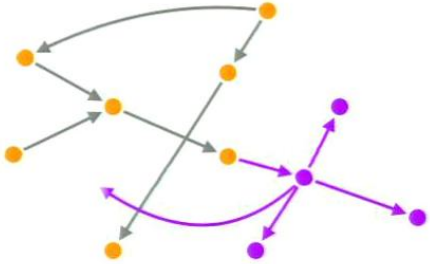
No conflict

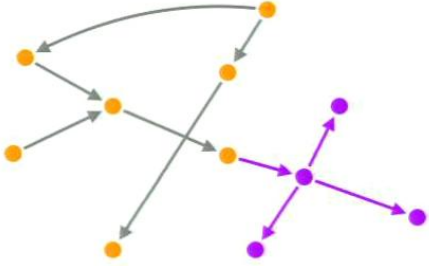
- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

Strong Eventual Consistency





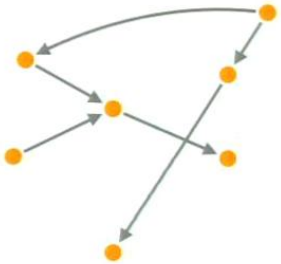
Update local + propagate
- No synch
- Expose intermediate state
- Eventual, reliable delivery

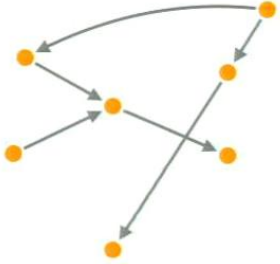
No conflict
- Unique outcome of concurrent updates

No consensus: ≤ n-1 faults
Not universal

Solves the CAP problem

Definition of EC



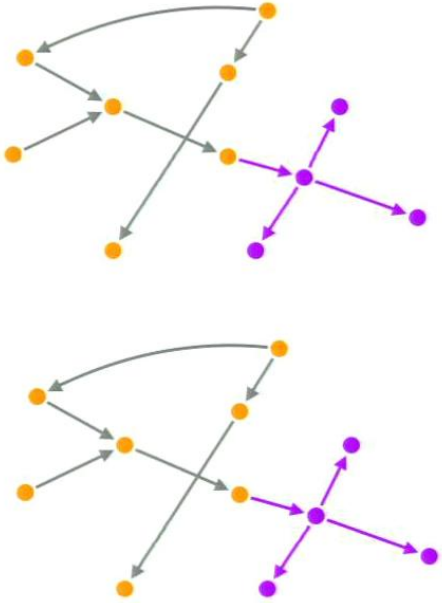


Eventual delivery: An update executed at a correct replica eventually executes at all correct replicas

Termination: All update executions terminate

Convergence: Correct replicas that have executed the same updates eventually reach equivalent state (and stay)

Definition of SEC



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

Strong Eventual Consistency

Local update
- No synchronisation
- Fast, responsive

Solves the CAP problem
- (Strong Eventual) Consistent
- Available
- Partition-tolerant
  - tolerates up to *n-1* failures

Does not require consensus
- Cheap
- Not universal

SEC ≠ Sequential Consistency

Consider Set-like object *S* such that:

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

Satisfies SEC conditions

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

Not sequentially consistent

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

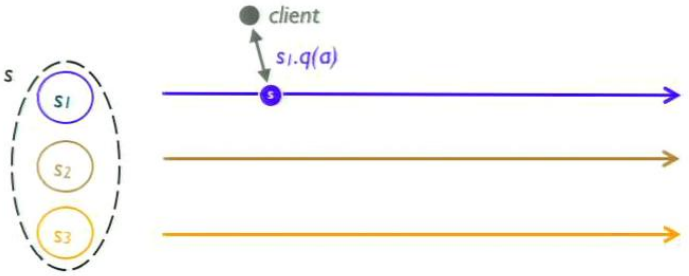
Object model & Sufficient conditions

Query

Local at *source* replica

- Client's choice

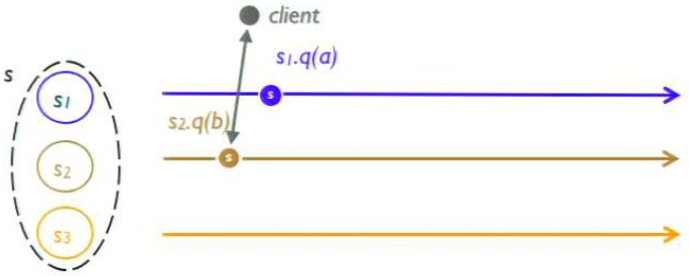
Query



Local at *source* replica

- Client's choice

Query



Local at *source* replica

- Client's choice

Synchronisation approaches

State-based:

- Local queries, local updates
- Send full state; on receive, merge
- Conceptually simple
- *File systems (NFS, Unison, Dynamo)*

Operation-based:

- Local queries
- Replicated updates
- Log, send operations
- Reconcile non-commutative operations
- Vector clock: ≈static network
- *Collaborative editing, Bayou, PNUTS*

State-based replication

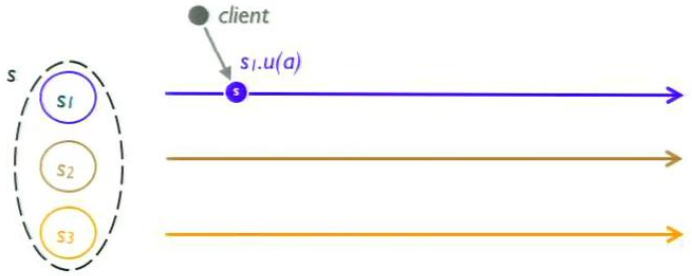
Local at source $s_1.u(a)$, $s_2.u(b)$, …

- Compute
- Update local payload

Convergence:

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

State-based replication



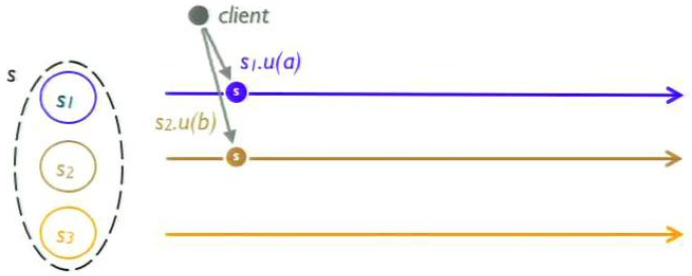
Local at source $s_1.u(a)$, $s_2.u(b)$, …

- Compute
- Update local payload

Convergence:

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

State-based replication



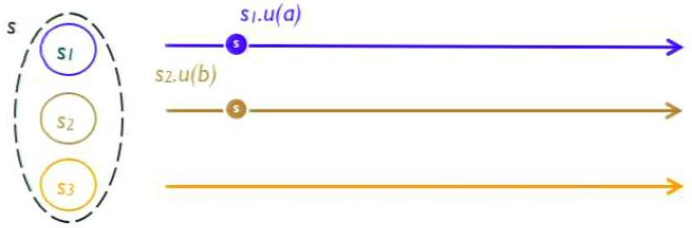
Local at source $s_1.u(a)$, $s_2.u(b)$, …

- Compute
- Update local payload

Convergence:

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

State-based replication



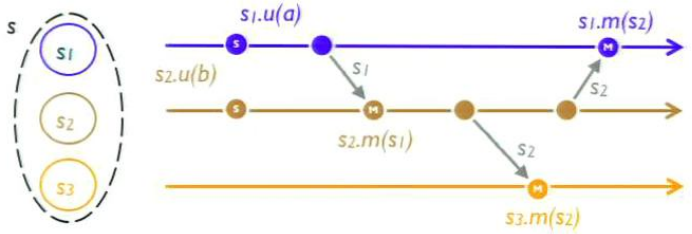
Local at source $s_1.u(a)$, $s_2.u(b)$, …

- Compute
- Update local payload

Convergence:

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

State-based replication



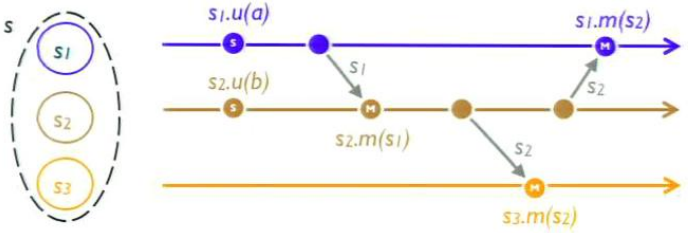
Local at source $s_1.u(a)$, $s_2.u(b)$, …

- Compute
- Update local payload

Convergence:

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

State-based: monotonic semi-lattice ⇒ CRDT



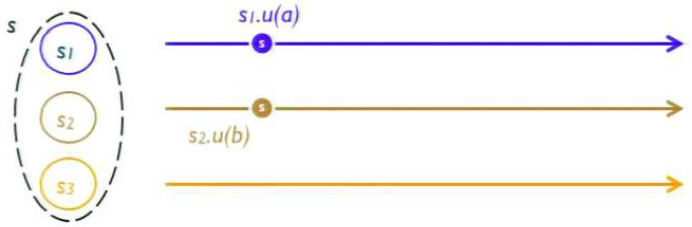
If

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

then replicas converge to LUB of last values

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

Operation-based replication



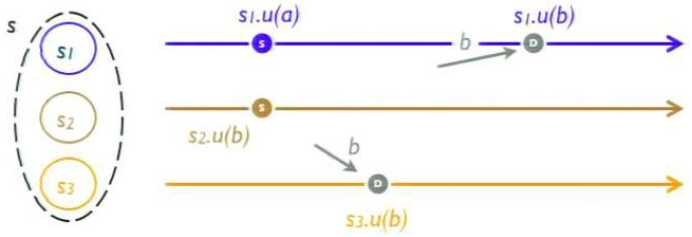
At source:

- prepare
- broadcast to all replicas

Eventually, at all replicas:

- update local replica

Operation-based replication



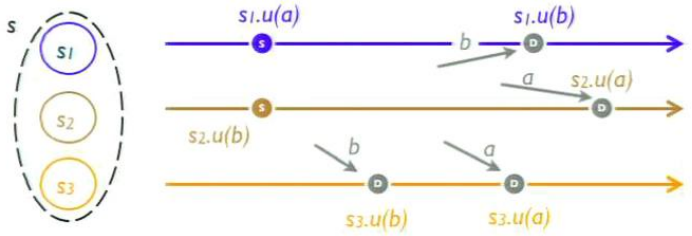
At source:

- prepare
- broadcast to all replicas

Eventually, at all replicas:

- update local replica

Operation-based replication



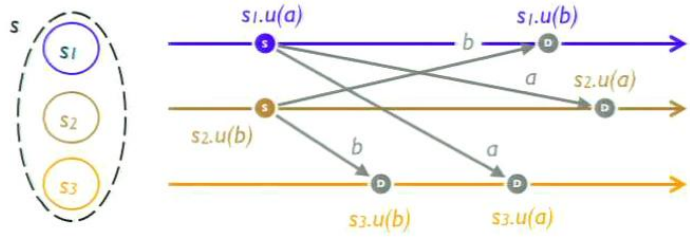
At source:

- prepare
- broadcast to all replicas

Eventually, at all replicas:

- update local replica

Op-based: commute ⇒ CRDT



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

Monotonic semi-lattice ⇔ commutative

1. A state-based object can emulate an operation-based object, and vice-versa
2. State-based emulation of an op-based CRDT is a CRDT
3. Operation-based emulation of a state-based CRDT is a CRDT

Emulating operation-based with state-based

RB_i reliable broadcast emulated as CvRDT

Replica payload at *i*:

- Grow-Set *Msgs*
- Grow-Set *Dlvrd*

Operations:

- *send(m)*: $Msgs := Msgs \cup \{ m \}$
- *deliver (Pre)*:
  choose $m \in Msgs \setminus Dlvrd$: *Pre(m)*;
  $Dlvrd := Dlvrd \cup \{ m \}$;
  m
- $s \leq s' \stackrel{def}{=} s.Msgs \subseteq s'.Msgs$
- *merge(s,s')*: $s.Msgs \cup s'.Msgs$

Comparison

State-based:

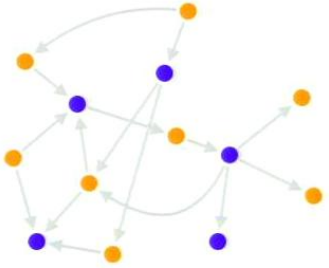
- Update ≠ merge operation
- Simple data types
- State includes preceding updates; no separate historical information

Operation-based:

- Update operation
- Higher level, more complex
- More powerful, more constraining
- Small messages

State-based or op-based, as convenient

Sharded CRDT



A combination of independent CRDTs remains a CRDT

Very large objects

- Independent *shards*
- Static: hash

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

- Payload: P = [int, int, …]
- *value()* = $\sum_i P[i]$
- *increment ()* = *P[MyID]++*
- *merge(s,s')* =
  $s \sqcup s' = [\ldots, max(s.P[i], s'.P[i]), \ldots]_i$

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

Set design alternatives

Sequential specification:

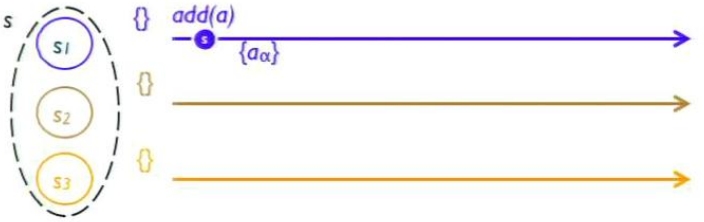
- $\{true\}$ add(e) $\{e \in S\}$
- $\{true\}$ remove(e) $\{e \notin S\}$

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

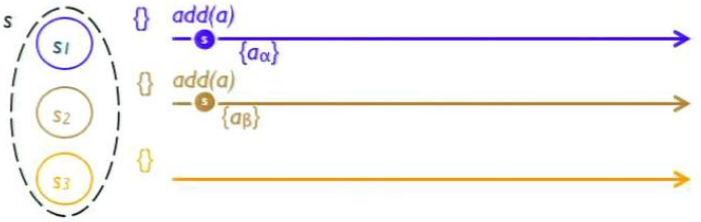
- ~~linearisable?~~
- error state?
- last writer wins?
- add wins?
- remove wins?

Observed-Remove Set

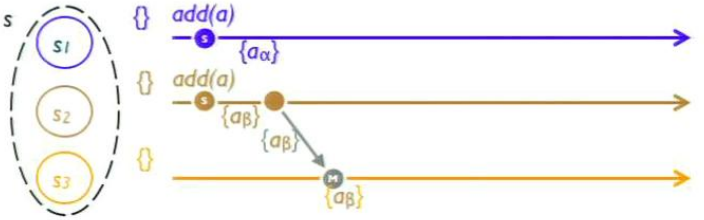
Observed-Remove Set



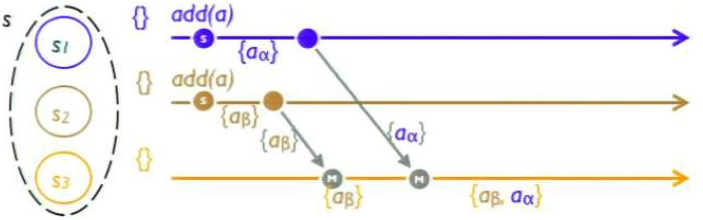
Observed-Remove Set



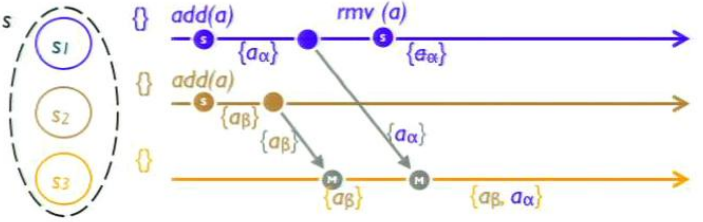
Observed-Remove Set



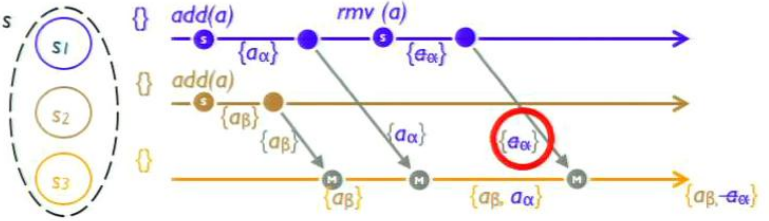
Observed-Remove Set



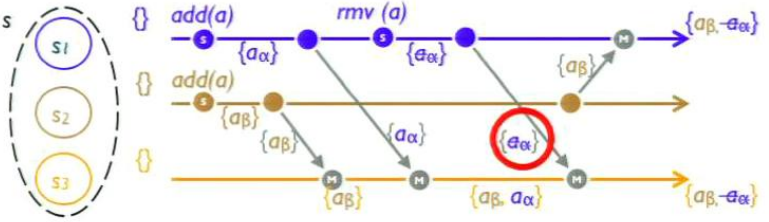
Observed-Remove Set



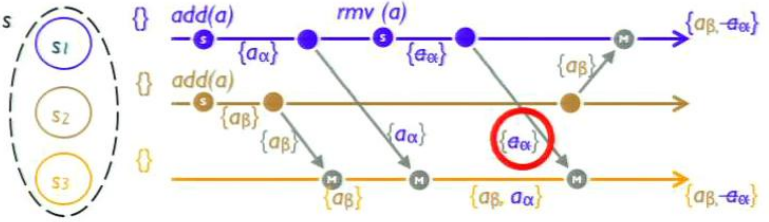
Observed-Remove Set



Observed-Remove Set



Observed-Remove Set



- 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

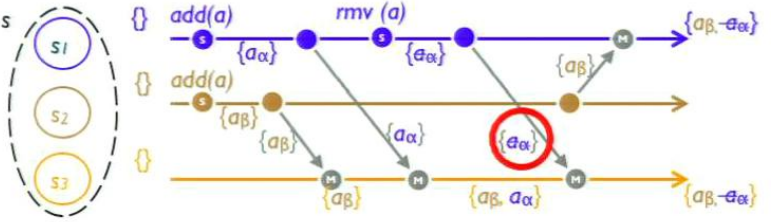
Set: solves Dynamo Shopping Cart anomaly

Optimisations:

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

Observed-Remove Set

Observed-Remove Set



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

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

Graph design alternatives

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

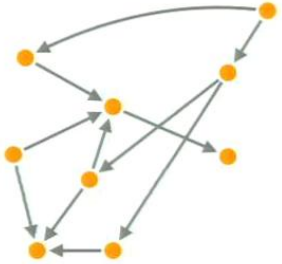
Sequential specification:

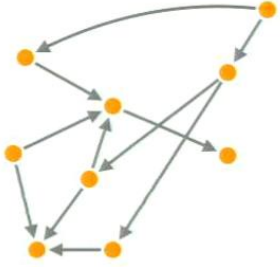
- $\{v,v' \in V\}$ addEdge(v,v') $\{\ldots\}$
- $\{\nexists(v,v') \in E\}$ removeVertex(v) $\{\ldots\}$

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

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

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')*

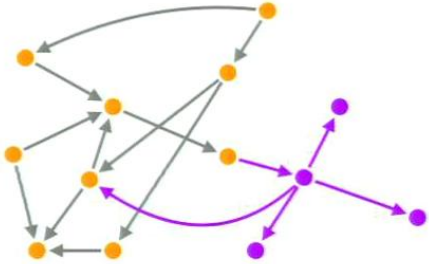
Do not enforce invariant a priori

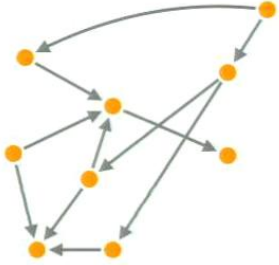
- lookupEdge(*v,v'*) = $(v,v') \in E \wedge v \in V \wedge v' \in V$

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

- remove wins

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')*

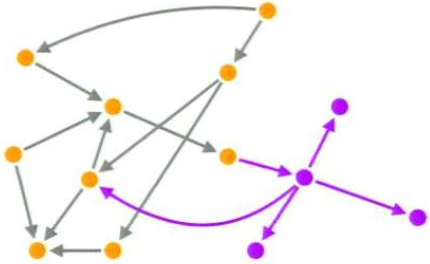
Do not enforce invariant a priori

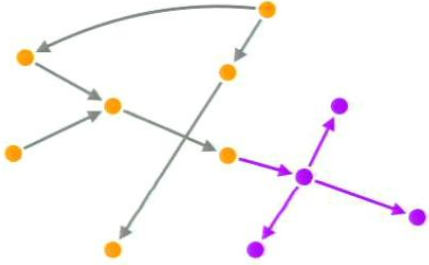
- lookupEdge(*v,v'*) = $(v,v') \in E \wedge v \in V \wedge v' \in V$

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

- remove wins

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')*

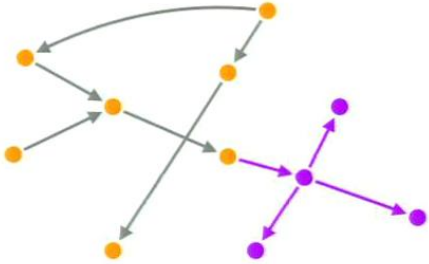
Do not enforce invariant a priori

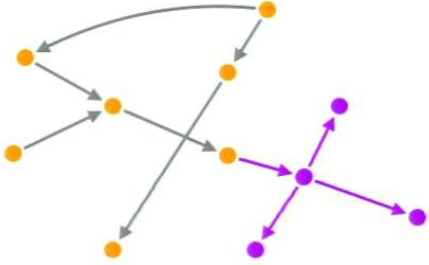
- lookupEdge(*v,v'*) = $(v,v') \in E \wedge v \in V \wedge v' \in V$

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

- remove wins

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')*

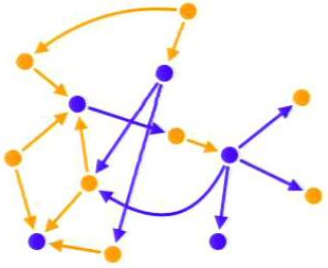
Do not enforce invariant a priori

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

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

- remove wins

Graph + shards + snapshots



Snapshot

- see OR-Set

Sharding

- See OR-Set
- Do not enforce invariant *a priori*

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

Ongoing work

CRDTs for P2P & Cloud Computing

ConcoRDanT: ANR 2010–2013

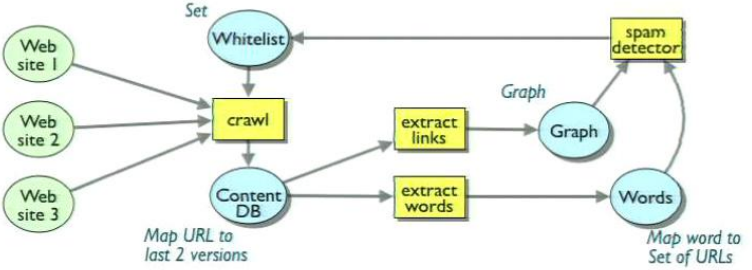
Systematic study of conflict-free design space

- Theory and practice
- Characterise invariants
- Library of data types

Not universal

- Conflict-free vs. conflict semantics
- Move consensus off critical path, non-critical ops

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

More scalability

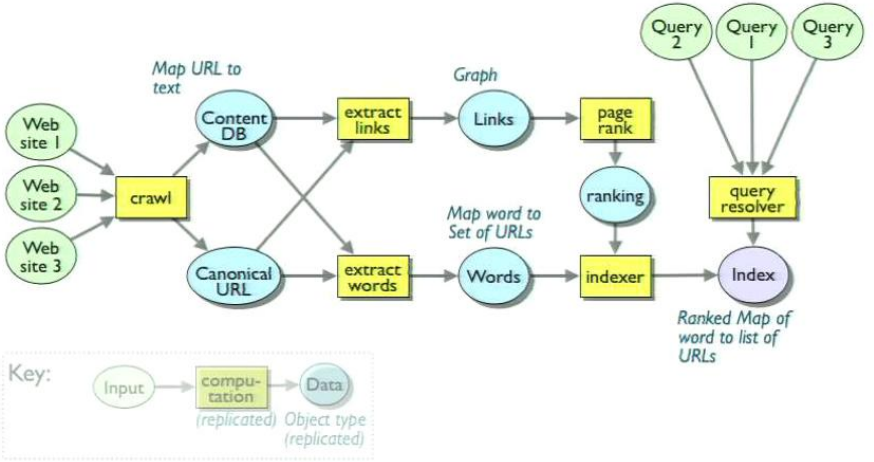
Incremental, asynchronous processing
- Operation-based: send small updates
- Dataflow-style notification network
- Push/throttle according to required QoS

Sharding
- Composability

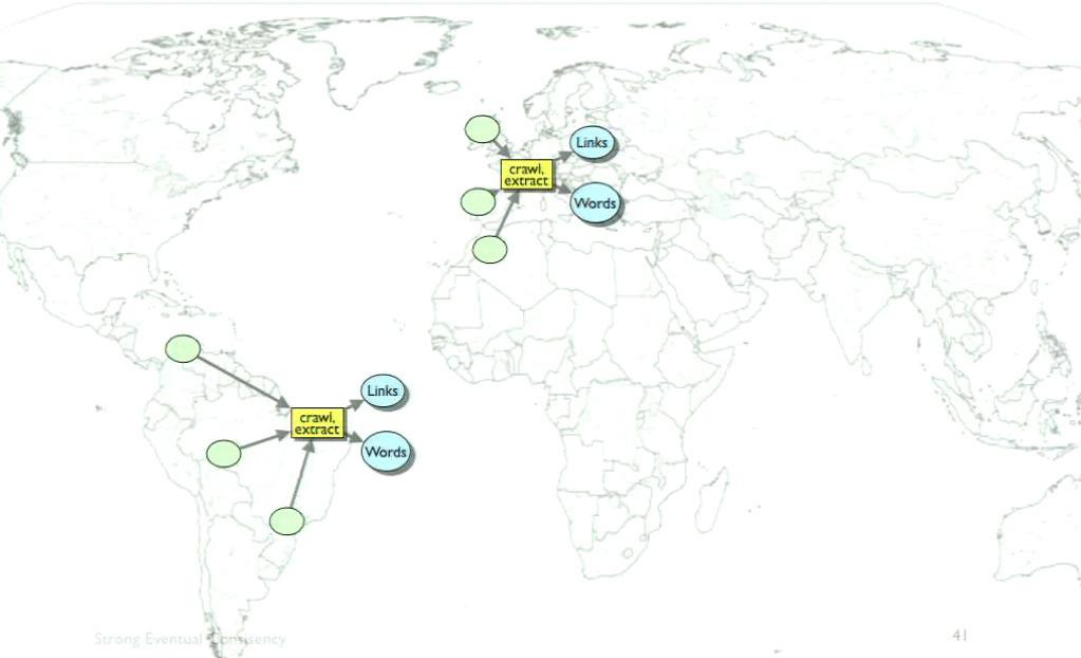
Large-scale transactional processing
- Snapshot Isolation

CRDTs appear ideal

Thought experiment (1)



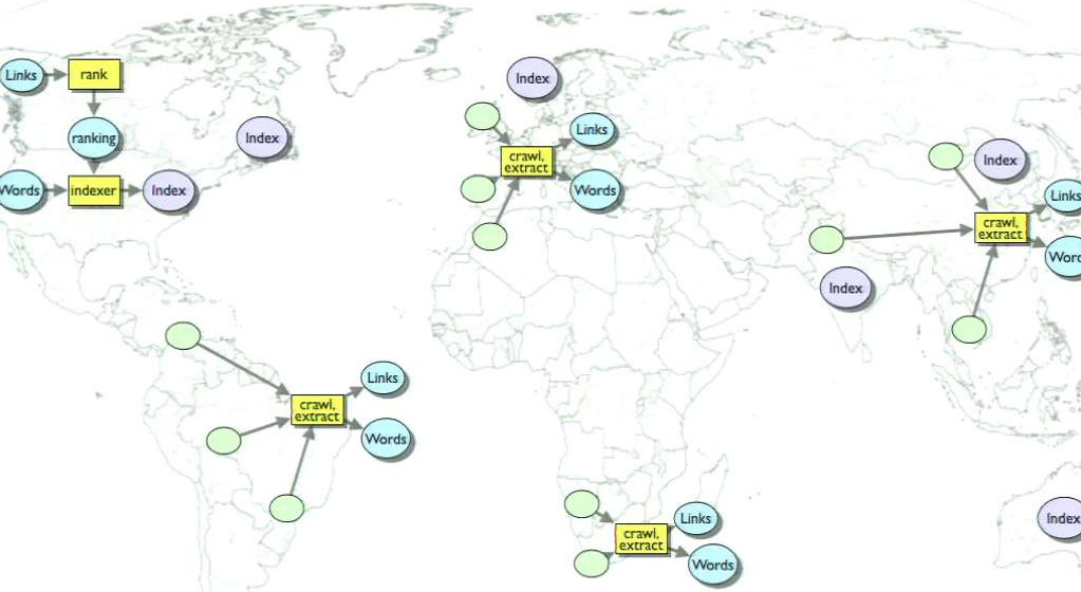
Thought experiment (2)



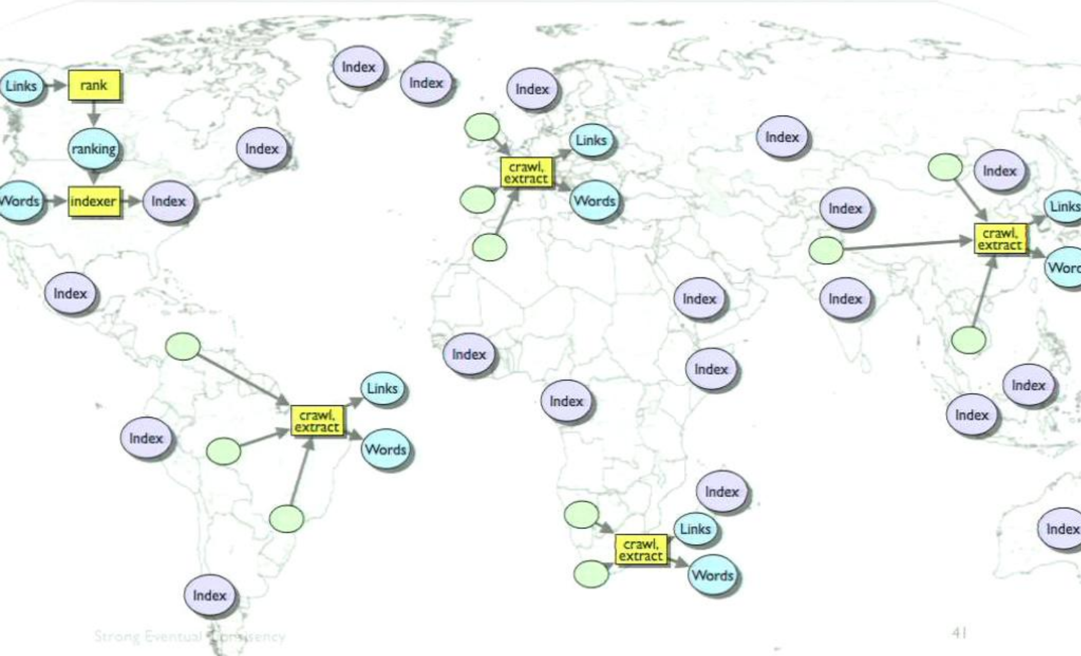
Thought experiment (2)

Thought experiment (2)

Thought experiment (2)



Thought experiment (2)



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