CRDTs and Coordination Avoidance (Lecture 16, cs262a)

Ion Stoica,
UC Berkeley
October 19, 2016

Today's Papers

CRDTs: Consistency without concurrency control,
Marc Shapiro, Nuno Preguica, Carlos Baquero, Marek Zawirski
Research Report, RR-6956, INRIA, 2009

(https://hal.inria.fr/inria-00609399v1/document)

Coordination Avoidance in Database Systems,
Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica,
Proceedings of VLDB'14

(http://www.vldb.org/pvldb/vol8/p185-bailis.pdf)

Replicated Data

Replicate data at many nodes

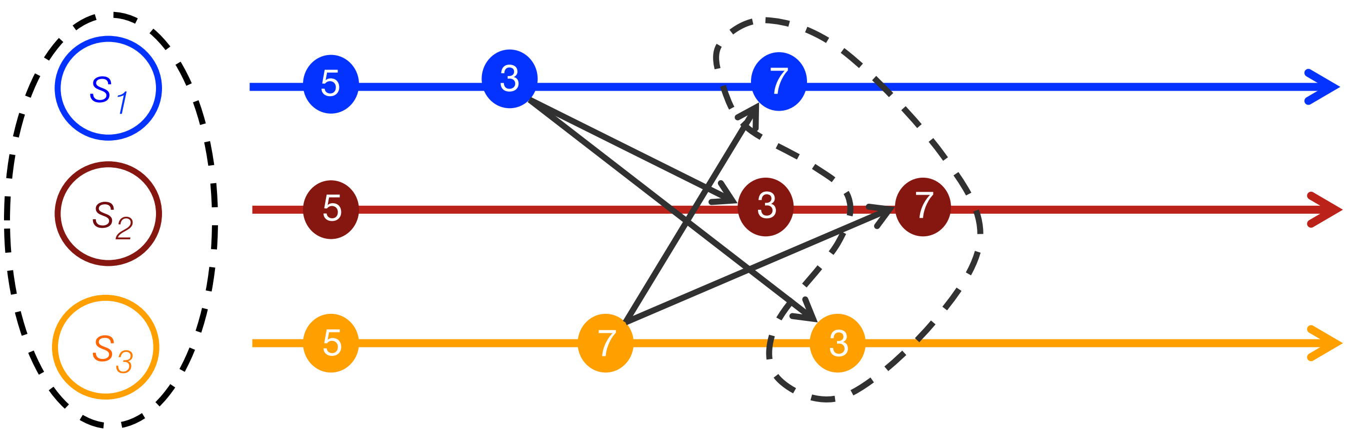
- Performance: local reads
- Fault-tolerance: no data loss unless all replicas fail or become unreachable
- Availability: data still available unless all replicas fail or become unreachable
- Scalability: load balance across nodes for reads

Updates

- Push to all replicas
- Consistency: expensive!

Conflicts

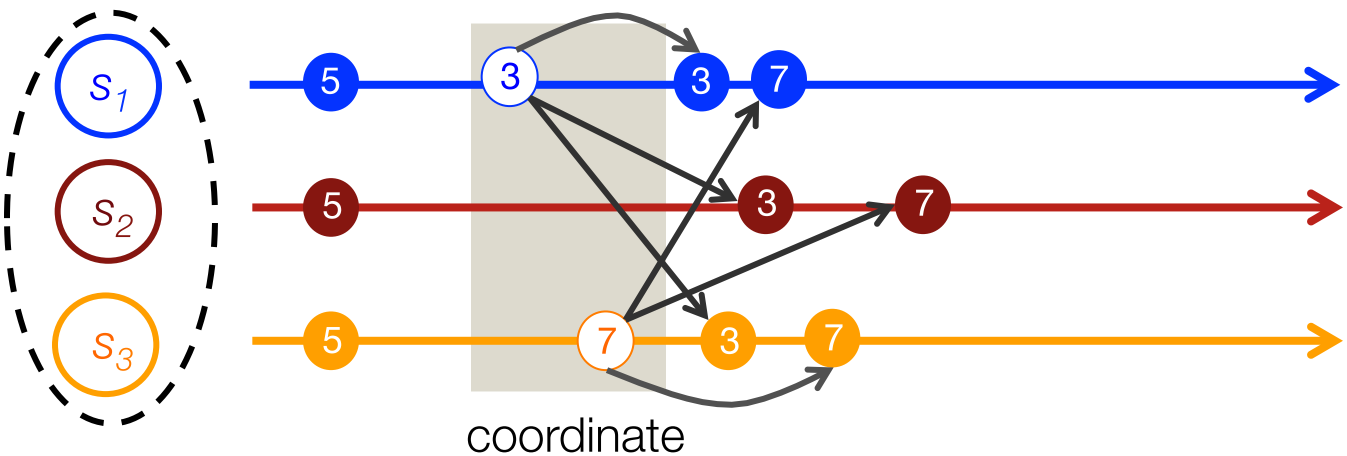
Updating replicas may lead to different results → inconsistent data



Strong Consistency

All replicas execute updates in same total order

- Deterministic updates: same update on same objects → same result



Strong Consistency

All replicas execute updates in same total order

- Deterministic updates: same update on same objects → same result

Requires coordination and consensus to decide on total order of operations

- N-way agreement, basically serialize updates → **very expensive!**

Eventual Consistency

If no new updates are made to an object all replicas will eventually converge to the same value

Update local and propagate

- No consensus in the background → scale well for both reads and writes
- Expose intermediate state
- Assume, eventual, reliable delivery

On conflict

- Arbitrate & Rollback

Eventual Consistency

If no new updates are made to an object all replicas will eventually converge to the same value

Move consensus to background

However:

- High complexity
- Unclear semantics if application reads data and then we have a rollback!

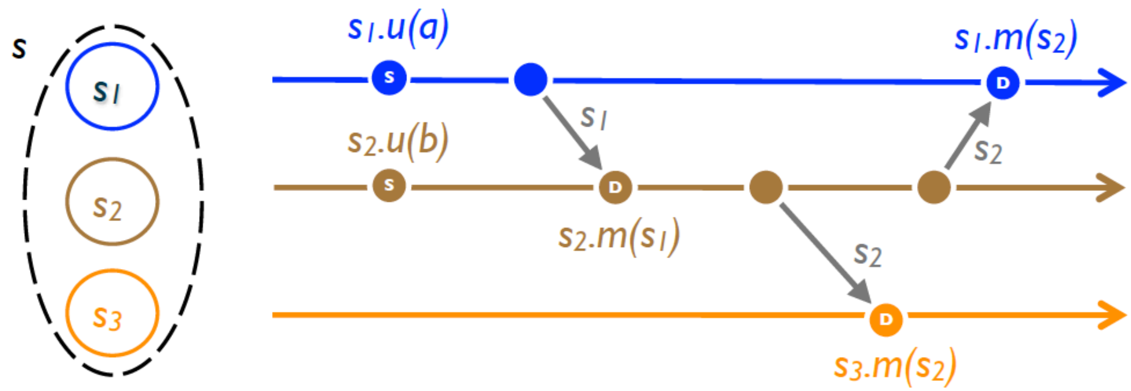
Strong Eventual Consistency

Like eventual consistency but with deterministic outcomes of concurrent updates

- No need for background consensus
- No need to rollback
- Available, fault-tolerant, scalable

But not general; works only for a subset of updates

State-based Replication



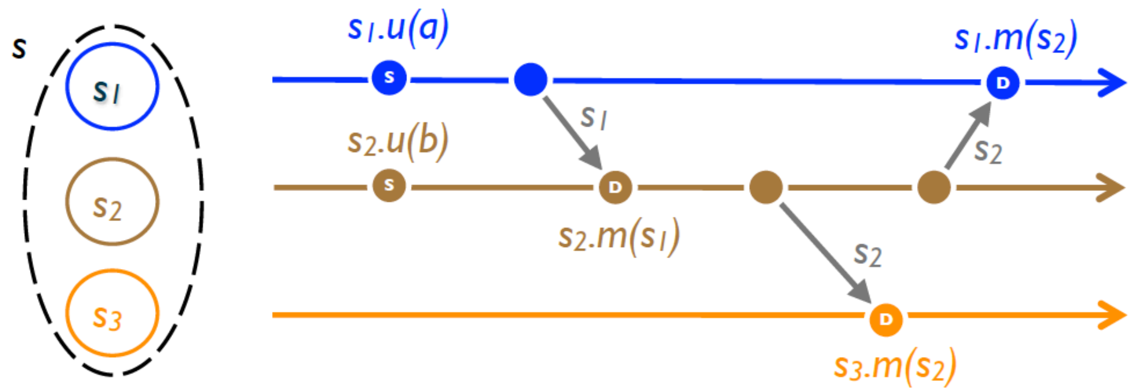
Replicated object: a tuple (S, s_0, q, u, m).

- Replica at process p_i has state $s_i \in S$
- s_0: initial state

Each replica can execute one of following commands

- q: query object's state
- u: update object's state
- m: merge state from a remote replica

State-based Replication



Algorithm

- Periodically, replica at p_i sends its current state to p_j
- Replica p_j merges received state into its local state by executing m

After receiving all updates (irrespective of order), each replica will have same state

Semi-lattice

Partial order $\leq$ set S with a least upper bound (LUB), denoted $\sqcup$

- $m = x \sqcup y$ is a LUB of $\{x, y\}$ under $\leq$ iff
  $\forall m', x \leq m' \wedge y \leq m' \Rightarrow x \leq m \wedge y \leq m \wedge m \leq m'$

It follows that $\sqcup$ is:

- **commutative**: $x \sqcup y = y \sqcup x$
- **idempotent**: $x \sqcup x = x$
- **associative:** $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$

Example

Partial order $\leq$ on set of integers

$\sqcup$: max( )

Then, we have:

- **commutative**: max(x, y) = max(y, x)
- **idempotent**: max(x, x) = x
- **associative**: max(max(x, y), z) = max(x, max(y, z))

Example

Partial order $\subseteq$ on sets

$\sqcup$: U (set union)

Then, we have:

- commutative: A U B = B U A
- idempotent: A U A = A
- associative: (A U B) U C = A U (B U C)

Monotonic Semi-lattice Object

A state-based object with partial order $\leq$, noted $(S, \leq, s_0, q, u, m)$, that has following properties, is called a monotonic semi-lattice:

1. Set S of values forms a semi-lattice ordered by $\leq$
2. Merging state s with remote state s' computes the LUB of the two states, i.e., $s \bullet m(s') = s \sqcup s'$
3. State is monotonically non-decreasing across updates, i.e., $s \leq s \bullet u$

Convergent Replicated Data Type (CvRDT)

Theorem: Assuming eventual delivery and termination, any state-based object that satisfies the monotonic semi-lattice property is SEC

Why does it work?

Don't care about order:

- Merge is both **commutative** and **associative**

Don't care about delivering more than once

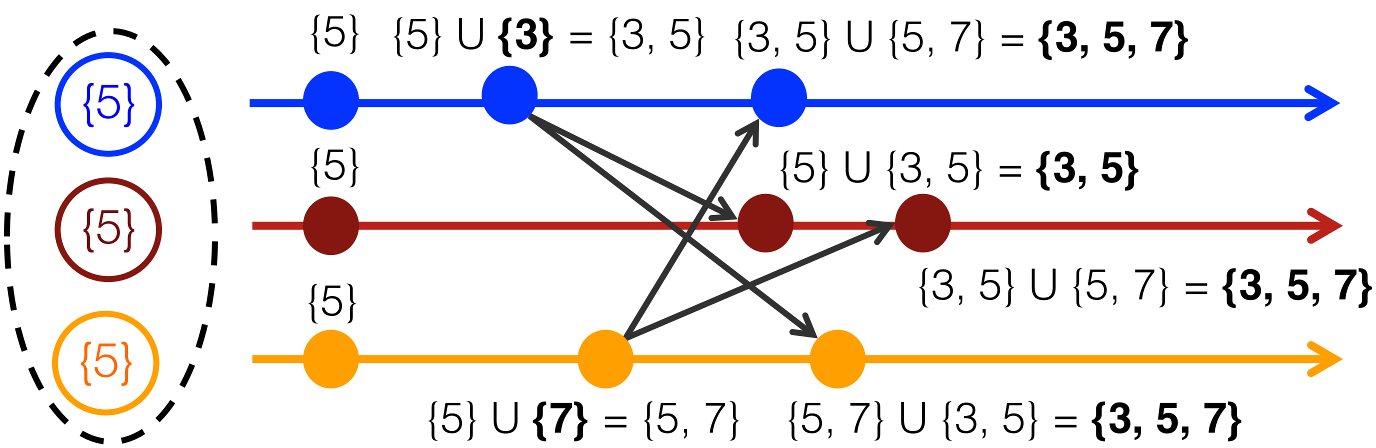
- Merge is **idempotent**

Numerical Example: Union Set

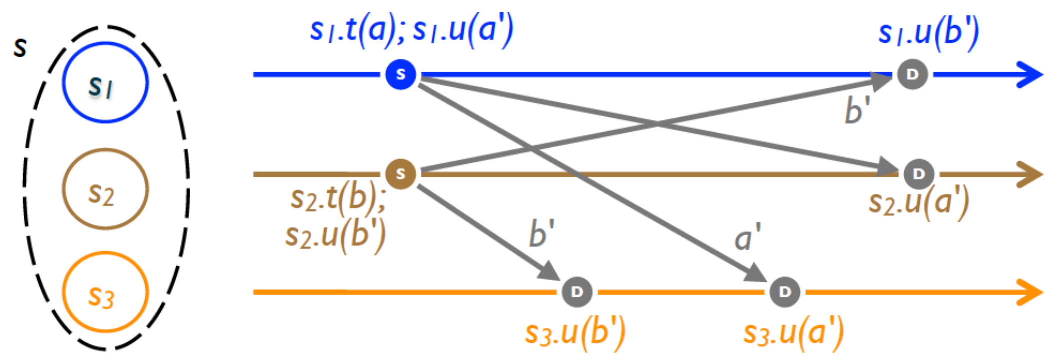
u: add new element to local replica

q: return entire set

merge: union between remote set and local replica



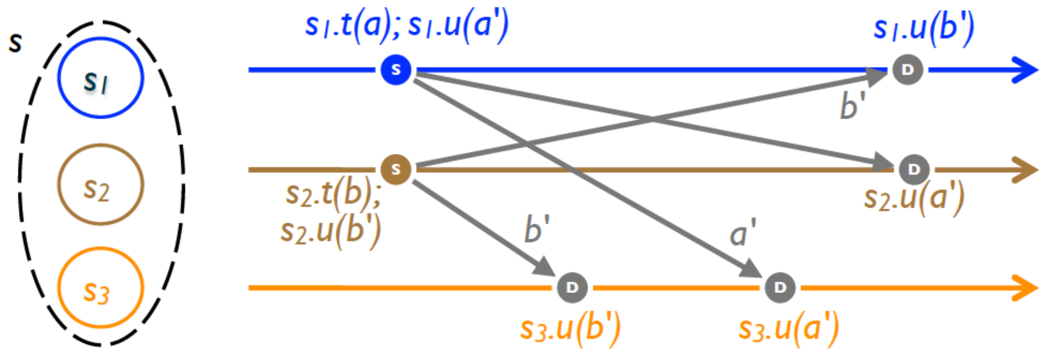
Operation-based Replication



An op-based object is a tuple (S, s_0, q, t, u, P), where S, s_0 and q have same meaning: state domain, initial state and query method

- No merge method; instead an update is split into a pair (t, u), where
- t: side-effect-free prepare-update method (at local copy)
- u: effect-free update method (at all copies)
- P: delivery precondition (see next)

Operation-based Replication



Algorithm

- Updates are delivered to all replicas
- Use causally-ordered broadcast communication protocol, i.e., deliver every message to every node exactly once, consistent with happen-before order
- Happen-before: updates from same replica are delivered in the order they happened to all recipients (effectively delivery precondition, P)
- Note: concurrent updates can be delivered in any order

Commutativity Property

Updates (t, u) and (t', u') commute, iff for any reachable replica state s where both u and u' are enabled

- u (resp. u') remains enabled in state $s \bullet u'$ (resp. $s \bullet u$)
- $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$

Commutativity holds for concurrent updates

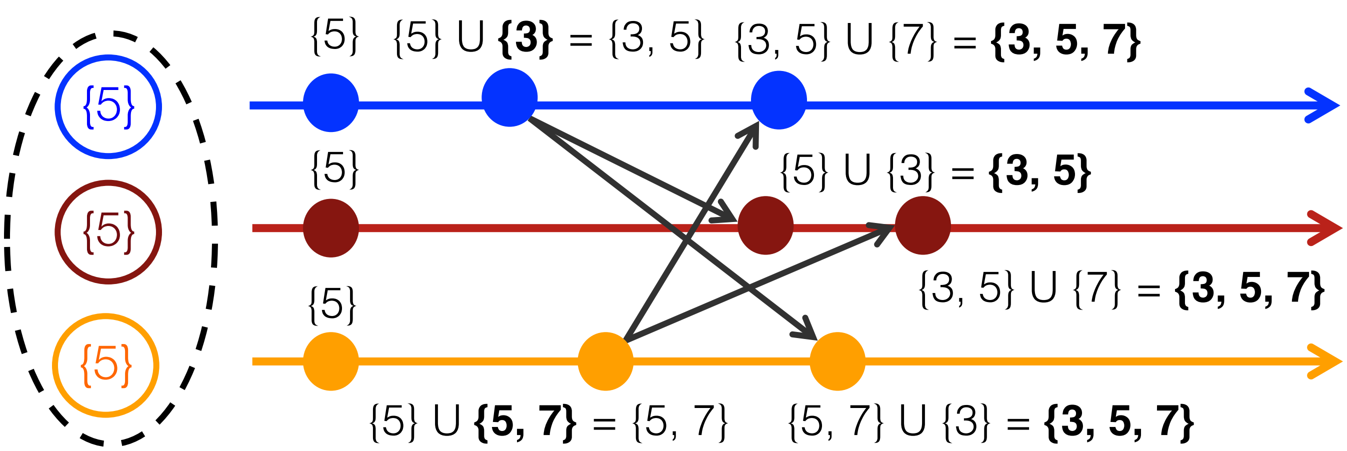
Commutative Replicated Data Type (CmRDT)

Assuming causal delivery of updates and method termination, any op-based object that satisfies the commutativity property for all concurrent updates is SEC

Numerical Example: Union Set

t: add a *set* to local replica

u: add delta to every remote replica



State-based vs Operation-based Replication

Both are equivalent!

- You can use one to emulate the other

Operation-base

- More efficient since you can ship only small updates

State-based

- Just requires reliable broadcast; causally-ordered broadcast much more complex!

CRDT Examples (cont'd)

Integer vector (virtual clock):

- u: increment value at corresponding index by one
- m: maximum across all values, e.g., m([1, 2, 4], [3, 1, 2]) = [3, 2, 4]

Counter: use an integer vector, with query operation

- q: returns sum of all vector values (1-norm), e.g., q([1, 2, 4]) = 7

Counter that decrements as well:

- Use two integer vectors:
  - I updated when incrementing
  - D updated when decrementing
- q: returns difference between 1-norms of I and D

CRDT Examples (cont'd)

Add only set object

- u: add new element to set
- m: union between two sets
- q: return local set

Add and remove set object

- Two add only sets
  - A: when adding an element, add it to A
  - R: when removing an element, add it to R
- q: returns A\R

CAP Theorem

You cannot achieve simultaneously

- Strong consistency
- Availability
- Partition tolerance

Why?

SEC a Solution for CAP?

Availability: a replica is always available for both reads and writes
Partition tolerance: any communicating subset of replicas of eventually converges

- even if partitioned from the rest of the network. SEC is weaker than

Fault tolerance: n-1 nodes can fail!

Almost a solution: SEC weaker than Strong Consistency, though good enough for many practical situations

Summary

Serialization, strong consistency

- Easy to use by applications, but don’t scale well due to conflicts

Two solutions to dramatically improve performance:

- CRDTs: eliminate coordination by restricting types of supported objects for **concurrent updates**
- Coordination avoidance: rely on application hints to avoid coordination for **transactions**

Question: what do these model mean for applications?