CRDTs and Coordination Avoidance (Lecture 8, cs262a)

Ion Stoica & Ali Ghodsi
UC Berkeley
February 12, 2018

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

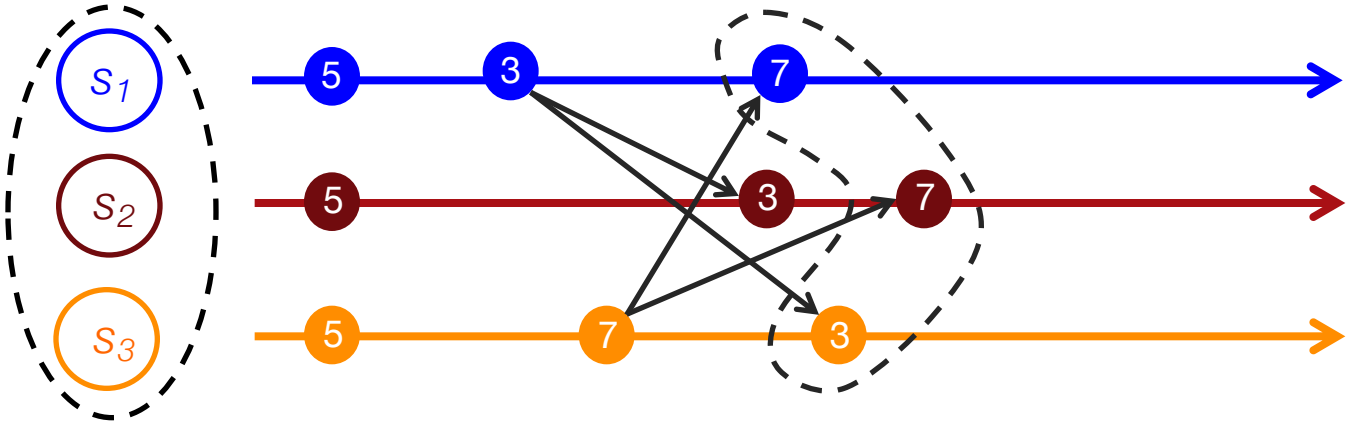
- Performance: local reads
- Fault-tolerance: no data loss unless all replicas fail or become unreachable
- Availability: data still available unless all replicas fail or become unreachable
- Scalability: load balance across nodes for reads

Updates

- Push to all replicas
- Consistency: expensive!

Conflicts

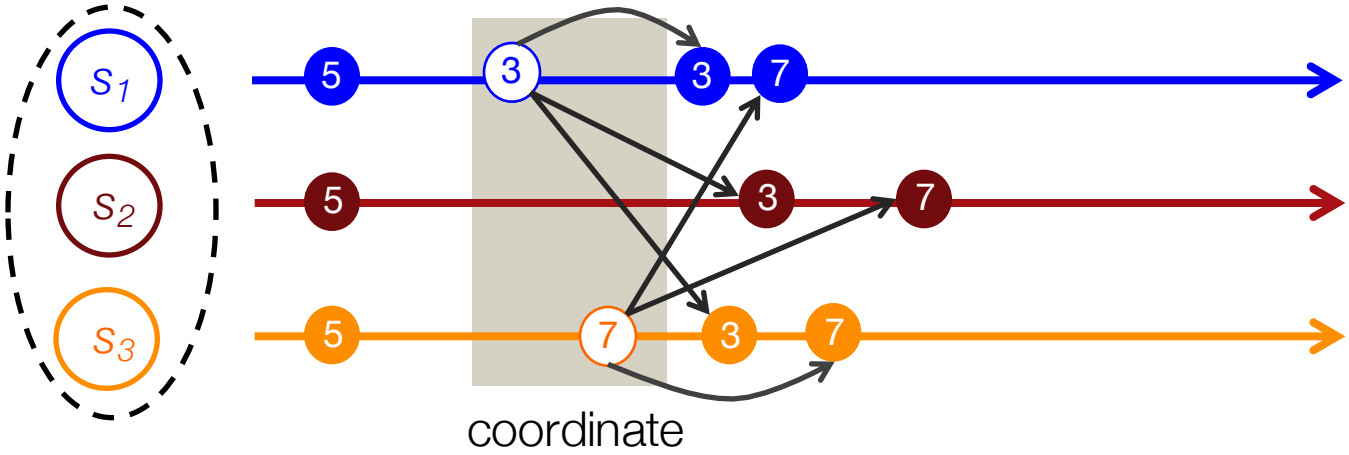
Updating replicas may lead to different results → inconsistent data



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

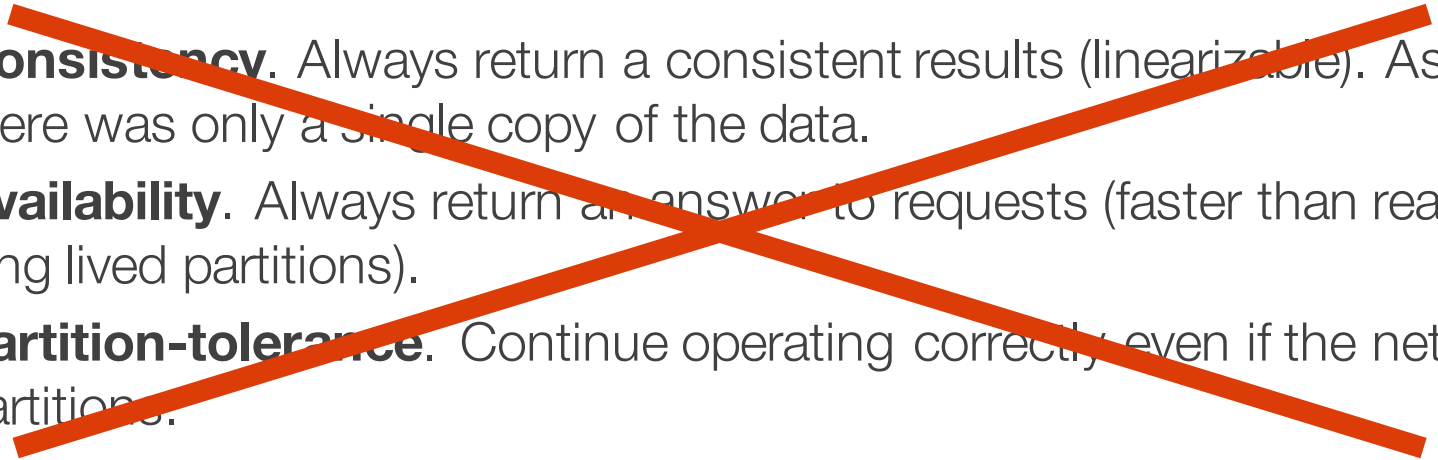
- Deterministic updates: same update on same objects → same result

Requires coordination and consensus to decide on total order of operations

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

CAP theorem

- Can only have two of the three properties in a distributed system
  - **Consistency**. Always return a consistent results (linearizable). As if there was only a single copy of the data.
  - **Availability**. Always return an answer to requests (faster than really long lived partitions).
  - **Partition-tolerance**. Continue operating correctly even if the network partitions.



CAP theorem v2

- When the networked is **partitioned**, you must chose one of these
  - **Consistency**. Always return a consistent results (linearizable). As if there was only a single copy of the data.
  - **Availability**. Always return an answer to requests (faster than really long lived partitions).

How can we get around CAP?

Eventual Consistency to the rescue

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, applications

- Arbitrate & Rollback

Eventual Consistency

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

However

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

Dynamo: Amazon's Highly Available Key-value Store

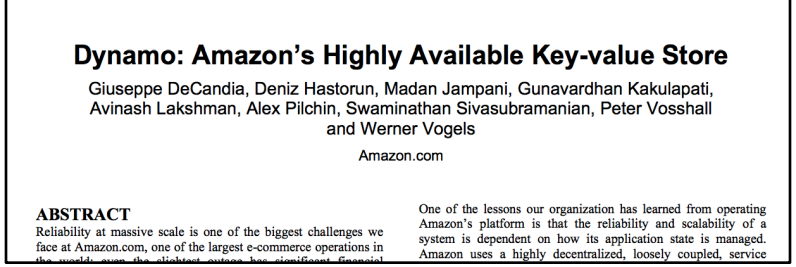
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service



Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

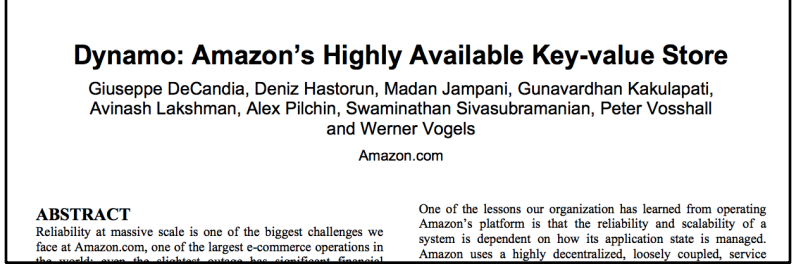
Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service

- Must be available when partitions happen
  - "For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers."
  - Handles 3 million checkouts a day (2009). Availability!



Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

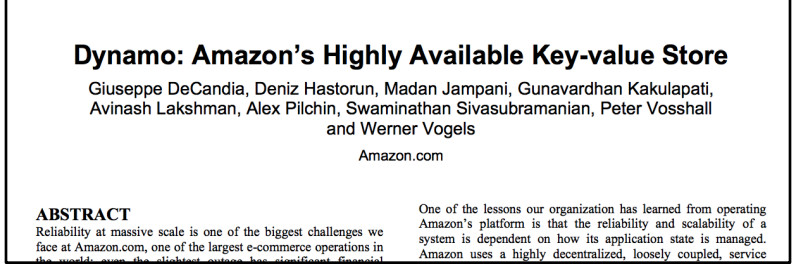
Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service

- Must be available when partitions happen
  - "Many traditional […]. In such systems, writes may be rejected if the data store cannot reach all (or a majority of) the replicas at a given time. On the other hand, Dynamo targets the design space of an "always writeable" data store (i.e., a data store that is highly available for writes). […] For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures. This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected"



Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service

- Must be available when partitions happen
  - "There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions. For example, the shopping cart application requires that an **"Add to Cart" operation can never be forgotten or rejected**. If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. **Note that both "add to cart" and "delete item from cart" operations are translated into put requests to Dynamo**. When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later. ."

Main idea of CRDTs

How does CRDTs get around these consistency problems of eventual consistency?

Create many specialized APIs with custom semantics

- Shopping cart might need a SET instead of PUT/GET
- A search engine might need a distributed DAG

CS Research Trick: assume more semantics. More limited applicability, but can do things that were impossible before!

Strong Eventual Consistency

Strong Eventual Consistency (SEC) is Eventual Consistency with the guarantee that correct replicas that have received the same updates (maybe in different order) have an equivalent correct state!

Like eventual consistency but with deterministic outcomes of concurrent updates

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

Partial Order (poset)

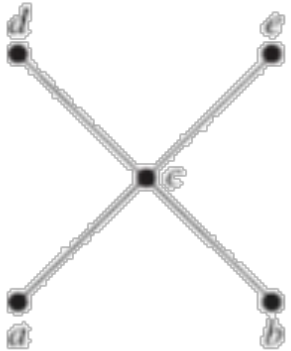
Set of objects S and an order relationship $\leq$ between them, such that for all a, b, c in S

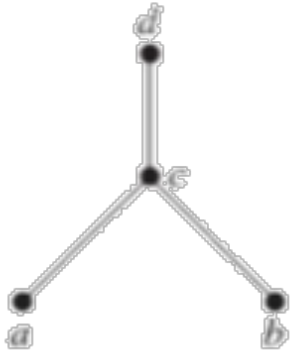
- **Reflexive**: $a \leq a$
- **Antisymmetric**: $(a \leq b \land b \leq a) \Rightarrow (a = b)$
- **Transitive**: $(a \leq b \land b \leq c) \Rightarrow (a \leq c)$

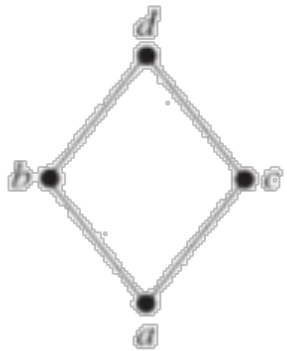
Hesse diagram

Simple way of describing posets, with a graph

- Read bottom to top (smaller to greater), no arrows, just links
- Remove self links
- Remove transitive links







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
  $\forall\ m'\ (x \leq m' \wedge\ y \leq m') \Rightarrow\ (x \leq m \wedge\ y \leq m \wedge\ m \leq m')$

The nice thing about semi-lattices is that 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)

Aha!

How can this help us in building replicated distributed systems?

- Just use the LUB ⊔ to merge state between replicas

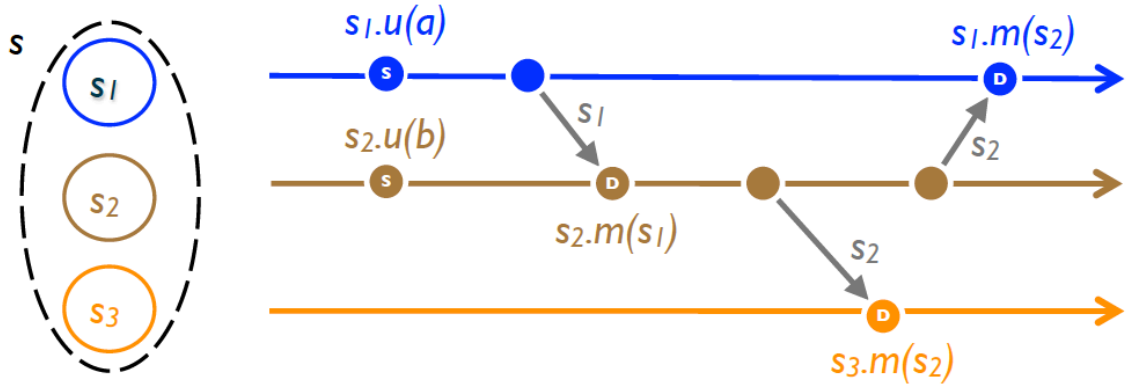
For instance, could build a CRDT using

- Supports add(integer)
- Supports get → returns the maximum integer
- How?

Always correct: available and strongly eventually consistent

Can we support remove(integer)?

State-based Replication



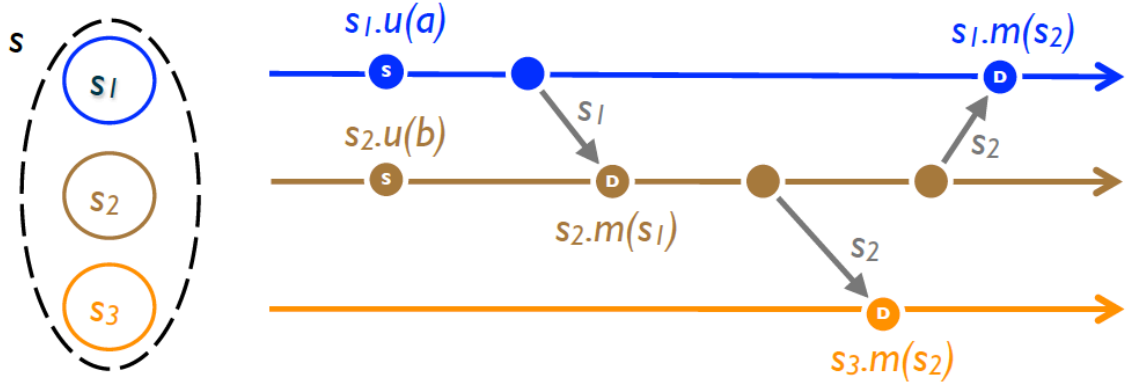
Replicated object: a tuple (S, s_0, q, u, m).

- 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

Monotonic Semi-lattice Object

A state-based object with partial order ≤, noted (S,≤, 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 ≤
2. Merging state s with remote state s′ computes the LUB of the two states, i.e., s •m (s′) = s⊔s′
3. State is monotonically non-decreasing across updates, i.e., s ≤ s • 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

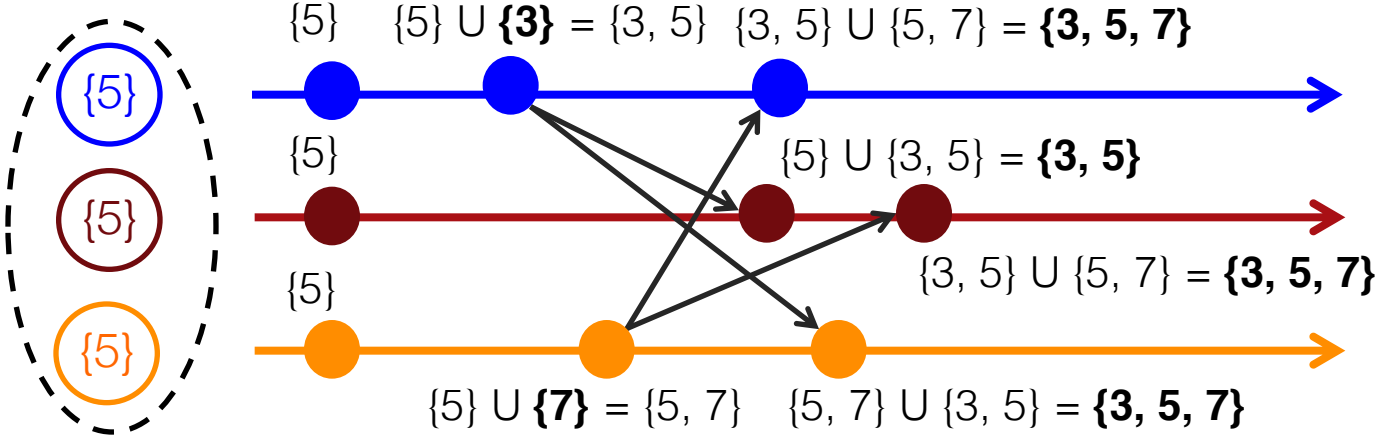
- Merge is **idempotent**

Numerical Example: Union Set

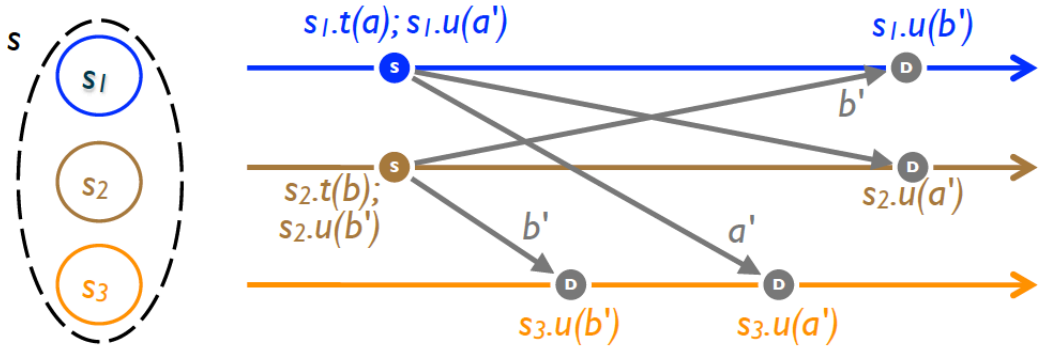
u: add new element to local replica

q: return entire set

merge: union between remote set and local replica



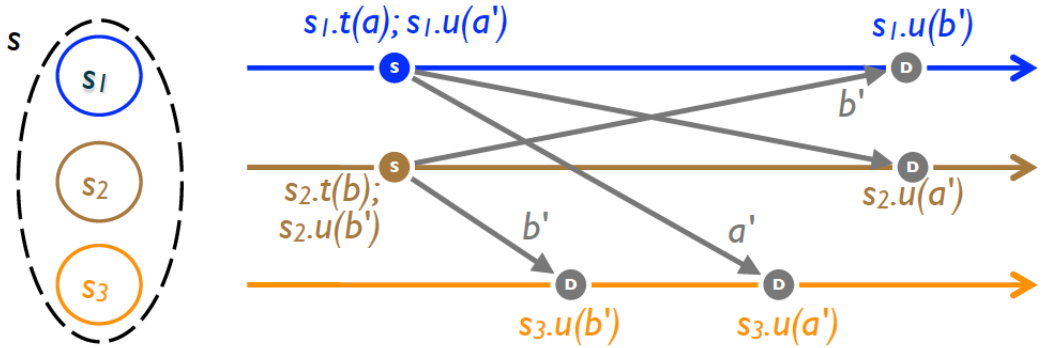
Operation-based Replication



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 • u′ (resp. s • u)
- s • u • u′ ≡ s • u′ • u

Commutativity holds for concurrent updates

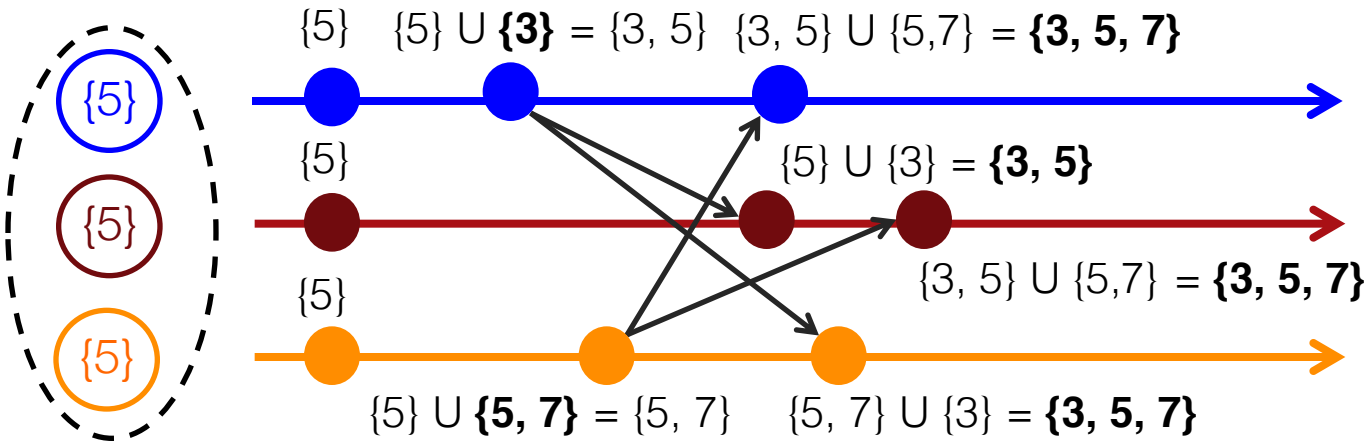
Commutative Replicated Data Type (CmRDT)

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

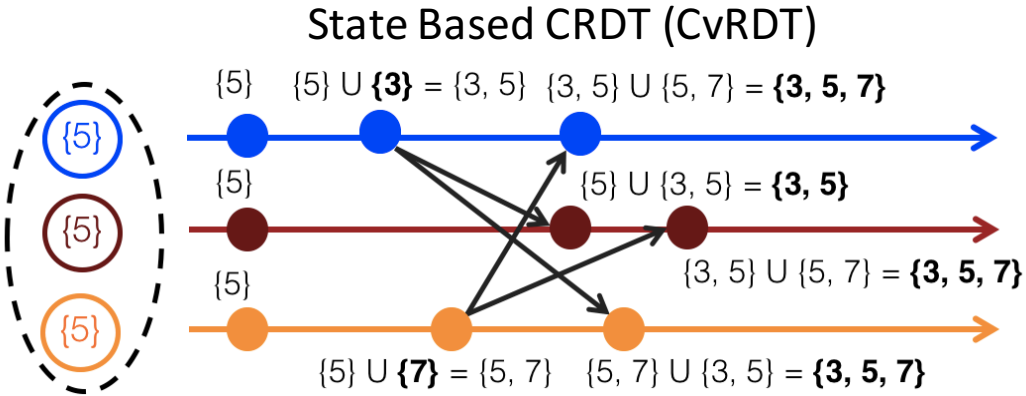
Numerical Example: Union Set

t: add a *set* to local replica

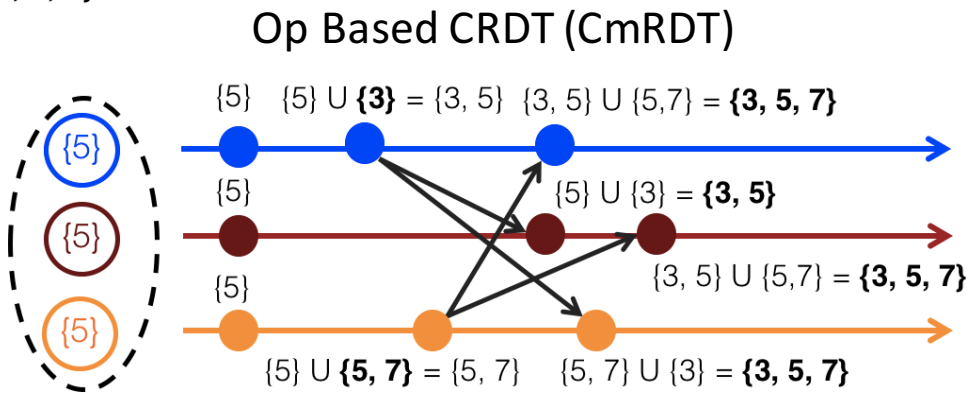
u: add delta to every remote replica



State-based vs Op-based



What is the differences and why might it matter?



State-based vs Operation-based Replication

Both are equivalent!

- You can use one to emulate the other

Operation-based

- More efficient since you can ship only small updates, but requires causally-ordered broadcast

State-based

- Just requires reliable broadcast; causally-ordered broadcast much more complex! But requires sending all state

CRDT Examples (cont'd)

Integer vector (virtual clock):

- u: increment value at corresponding index by one, inc(i)
- 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 (only supports adding an element at most once)

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.

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**

Discussion

- What's the main contribution of this paper?
- What do these models mean for applications?
- What's the relationship between transactions and CRDTs?