Replication and Consistency

Conflict-free Replicated Data Types (CRDTs)

Pedro F. Souto (pfs@fe.up.pt)
(Based on presentation by Ion Stoica & Ali Ghodsi,)

May 31, 2019

Data Replication

Replicate data at many nodes

Performance local reads

Reliability no data-loss unless data is lost in all replicas

Availability data available unless all replicas fail or become unreachable

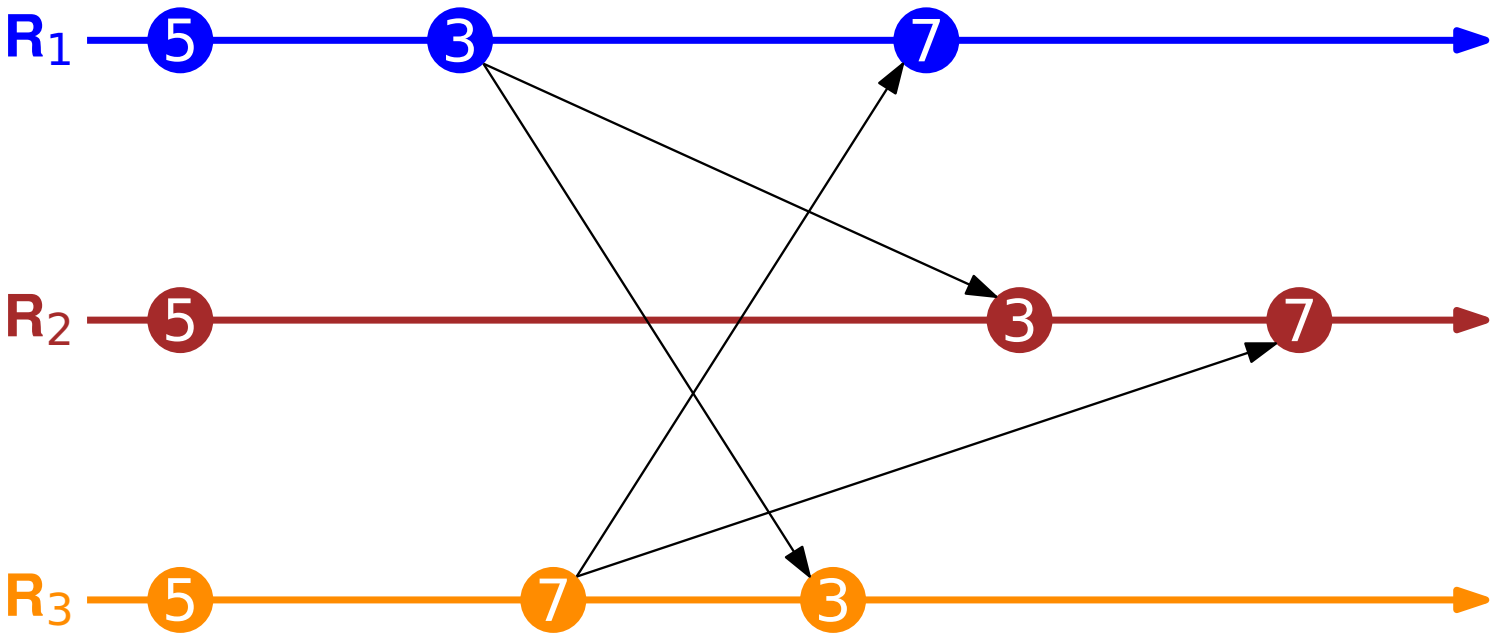
Scalability balance load across nodes for reads

Upon an update

- Push data to all replicas
- Challenge: ensure **data consistency**

Conflicts

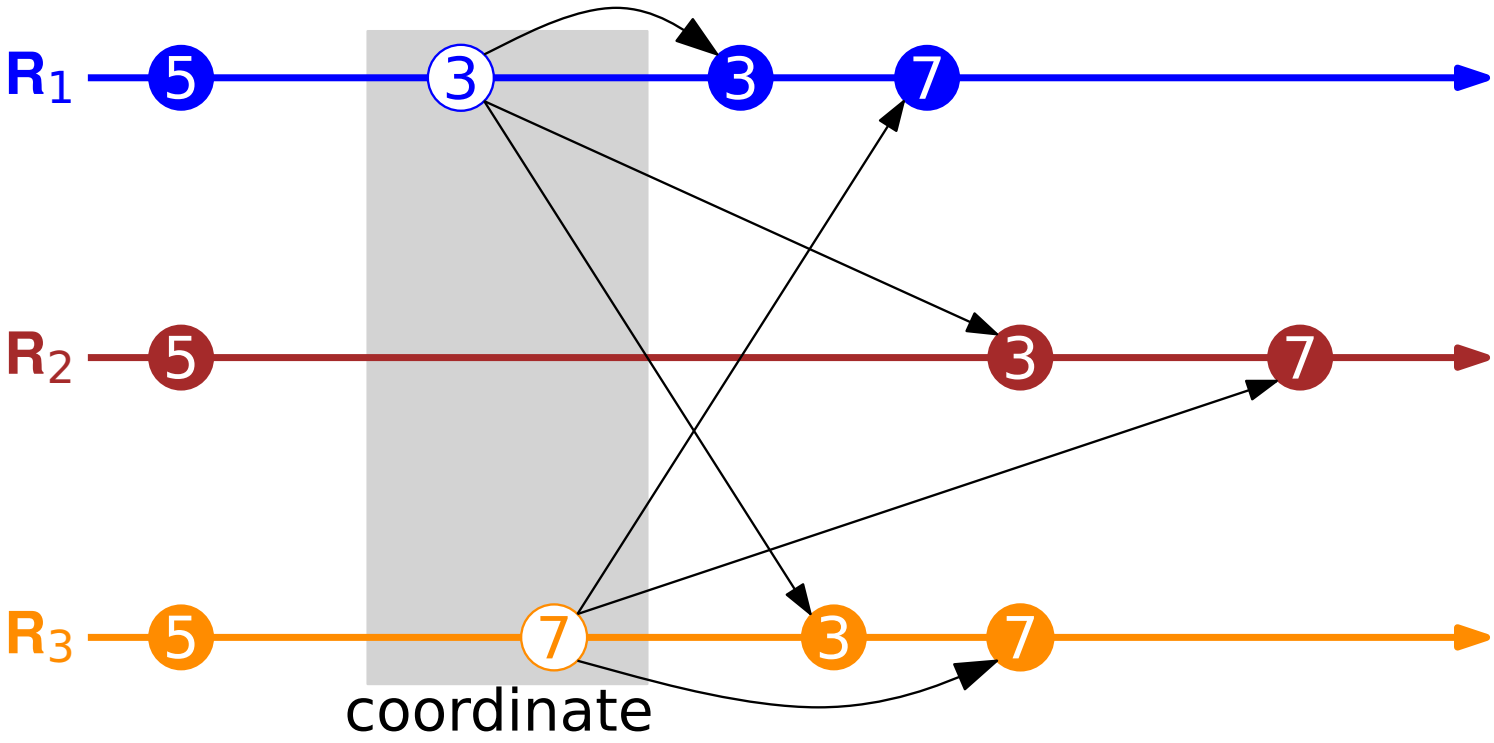
Observation Updating at different replicas may lead to different results, i.e. **inconsistent** data



Strong Consistency

All replicas execute updates in the same order

- Deterministic updates: **same initial state** leads to **same result**



Requires coordination to agree on total order of operations

- Can use Paxos, but ...

Strong Consistency Models

Sequential Consistency

Linearizability

Serializability

Sequential Consistency Model (Lamport 79)

Definition An execution is **sequential consistent iff** it is identical to a sequential execution of all the operations in that execution such that

- all operations executed by any thread, appear in the order in which they were executed by the corresponding thread

Observation This is the model provided by a multi-threaded system on a uniprocessor

Counter-example Consider the following operations executed on two replicas of variables x and y, whose initial values are 2 and 3, respectively

```
Répl. 1          Répl. 2
(2,3)            (2,3)          /* Initial values */
x = y + 2;       y = x + 3;
(5,5)            (5,5)          /* Final values */
```

If the two operations are executed sequentially, the final result cannot be $(5, 5)$

One-copy Serializability (Transaction-based Systems)

Definition The execution of a set of transactions is **one-copy serializable iff** its outcome is similar to the execution of those transactions in a **single** copy

Observation 1 Serializability used to be the most common consistency model used in transaction-based systems

- DB systems nowadays provide weaker consistency models to achieve higher performance

Observation 2 This is essentially the sequential consistency model, when the operations executed by all processors are transactions

- The isolation property ensures that the outcome of the concurrent execution of a set of transactions is equal to some sequential execution of those transactions

Observation 3 (Herlihy . . . sort of) Whereas

Serializability Was proposed for databases, where there is a need to preserve complex application-specific invariants

Sequential consistency Was proposed for multiprocessing, where programmers are expected to reason about concurrency

Linearizability (Herlihy&Wing90)

Definition An execution is **linearizable** iff it is **sequential consistent** and

- if op_1 **occurs before** op_2, according to some **external observer**, then op_1 appears before op_2

Question What is required so that two observers do not order 2 events differently?

- It is known that, because the speed of light is finite, two observers may see 2 events in different order.

Observation If op_1 and op_2 overlap in time, their relative order may be any

CAP Theorem

Can only have two of the following

Consistency Always return a consistent result (linearizable). (As if there was only a single copy of data.)

Avaliability Always return an answer to requests with **acceptable** response-time

Partition-tolerance Continue providing service even if the network partitions

CAP Theorem v2 (1/2)

When the network is **partitioned** you must choose one of the following:

Consistency Always return a consistent result (linearizable). (As if there was only a single copy of data.)

Avaliability Always return an answer to requests with **acceptable** response-time

Paxos chooses **consistency** (safety) over availability (liveness)
But some applications must be available when partitions happen

CAP Theorem v2 (2/2)

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 consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

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 oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. 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.

How to get around CAP?

Eventual Consistency

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

- What are the values observed before convergence is reached?

Implementation Update locally and propagate

- No coordination upon update
- Assumes **eventual** reliable delivery

Issues

- Need for reconciliation when conflicts occur
  - Complex and ad-hoc
  - May require undoing updates

  Similar to what happens on version control systems such as SVN or git
- Unclear semantics
  - Application may read data that is affected by reconciliation

Conflict-free Replicated Data Types (CRDTs)

Idea Define data types whose operations have semantics that:

- Make clear which intermediate states are observable
- Allow (mostly-)local reconciliation, i.e. without need for expensive coordination among nodes
  - Not exactly conflict-free

Example Use **set**s to implement a shopping cart

Standard CS Trick assume more semantics

- Applicability is more limited
- But can do things that were impossible before

Partial Order Set (poset)

Definition Set S and order relationship $\leq$ between the elements of S, such that for all a, b and c in S:

Reflexive $a \leq a$

Antisymmetric $(a \leq b \wedge b \leq a) \Rightarrow (a = b)$

Transitive $(a \leq b \wedge b \leq c) \Rightarrow (a \leq c)$

(Join) Semi-lattice

Definition of (join) semi-lattice Partial order set $(S, \leq)$ where each pair of elements of S has a least upper bound (LUB), denoted $\sqcup$

Definition of LUB $\ell = x \sqcup y$ is a LUB of $\{x, y\}$ under $\leq$ iff:

$$\forall_m : (x \leq m) \wedge (y \leq m) \Rightarrow (x \leq \ell \wedge y \leq \ell \wedge \ell \leq m)$$

(Nice) Properties of LUBs:

- **Commutativity** $x \sqcup y = y \sqcup x$
- **Idempotentcy** $x \sqcup x = x$
- **Associativity** $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$

(Join) Semi-lattice Example 1

Poset $(\mathbb{N}_0, \leq)$

$\sqcup$ $max()$

Properties

- Commutativity $max(x, y) = max(y, x)$
- Idempotency $max(x, x) = x$
- Associativity $max(max(x, y), z) = max(x, max(y, z))$

(Join) Semi-lattice Example 2

Poset Set of sets, and $\subseteq$

$\sqcup$ $\cup$ (set union)

Properties

- **Commutativity** $A \cup B = B \cup A$
- **Idempotency** $A \cup A = A$
- **Associativity** $(A \cup B) \cup C = A \cup (B \cup B)$

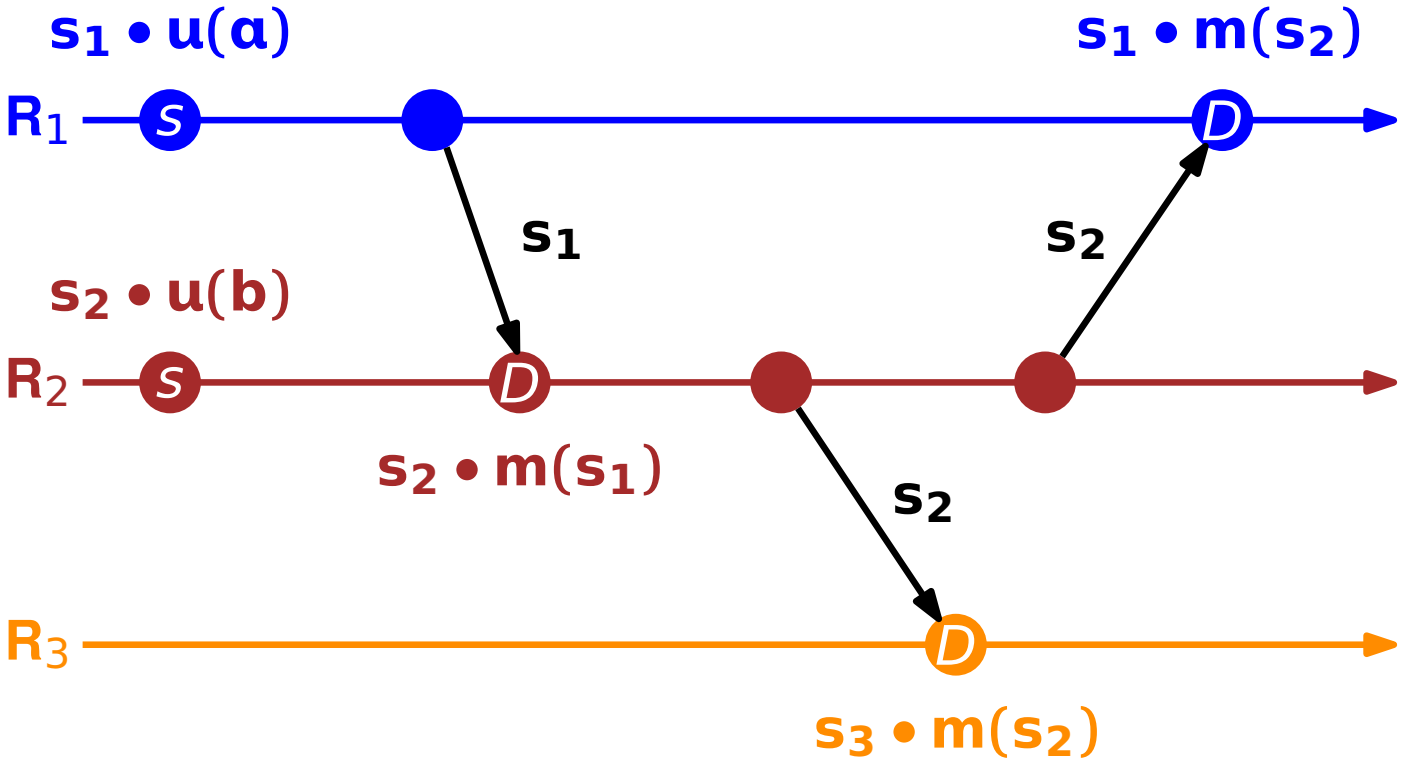
Eureka

Question How can all this math help us in ensuring consistency among replicated objects/services?

Response Use LUB, $\sqcup$, to reconcile/merge the state of different replicas

- This idea is due to Carlos Baquero and Francisco Moura, Universidade do Minho, published in a 1997 paper
- Similar idea, though not in such a systematic way, by Eugene Wuu and Arthur J. Bernstein, in a 1984 paper

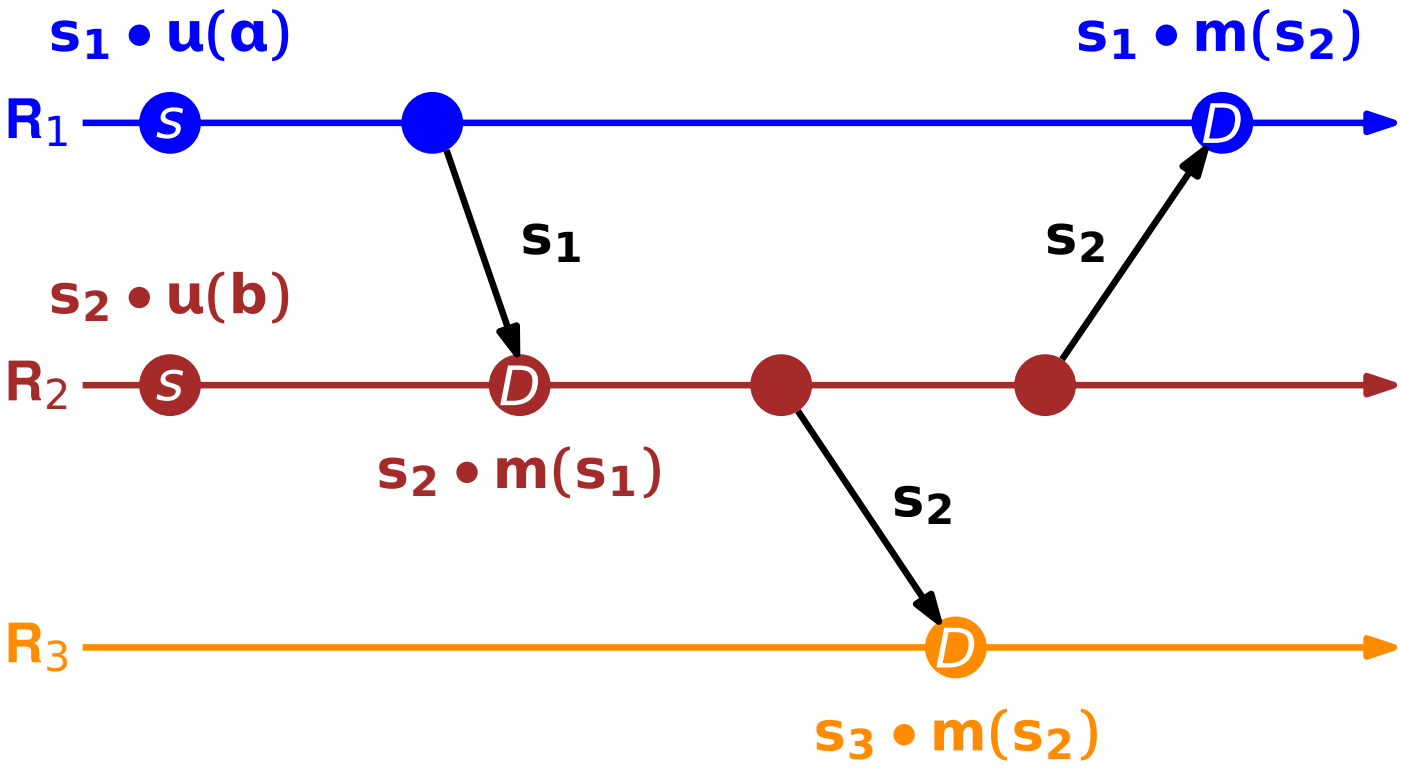
State-Based Replication: Convergent RDT (CvRDT)



(Replicated) State-based object a tuple (S, s^0, q, u, m) where:

- S set of states of object. Replica at process p_i has state $s_i \in S$
- s^0 initial state
- q side-effect free query object's state operation
- u update object's state operation
- m merge state from remote replica operation

State-Based Replication: CvRDT (2/2)



Algorithm

Replica p_i sends periodically its current state to p_j
Replica p_j executes m, merging received state into its local state

So that this algorithm ensures that p_i and p_j have the same state after "merging the same updates", we need to impose some restrictions on the replicated object . . .

Monotonic Semi-lattice Object

Definition a state-based object with partial order $\leq$, denoted $(S, \leq, s^0, q, u, m)$, that has the following properties

1. $(S, \leq)$ is a joint semi-lattice
2. u monotonically increases the local state, i.e. $s \leq s \bullet u$
3. m changes the current local state, s, to the LUB of s and the remote state s':

$$s \bullet m(s') = s \sqcup s'$$

CvRDT and Strong Eventual Consistency

Theorem Assuming eventual delivery and termination of all operations, any state-based object that satisfies the monotonic semi-lattice property is **strong eventually consistent (SEC)**, where

SEC

Guarantees If no new updates are made to an object, all replicas will **eventually** converge to the same value (as in Eventual Consistency) **and**

- Replicas that have applied the same updates (may be in different order) have an equivalent state

Proof Based on LUB properties

- And on the monotonicity of u
  - If u were not monotonic, then using the LUB as m might lead to inconsistent states

CvRDT

Weak assumptions about communication

- Don't care about order
  - Merge is both commutative and associative
- Don't care about delivering more than once
  - Merge is idempotent

The burden is on designing the CvRDTs

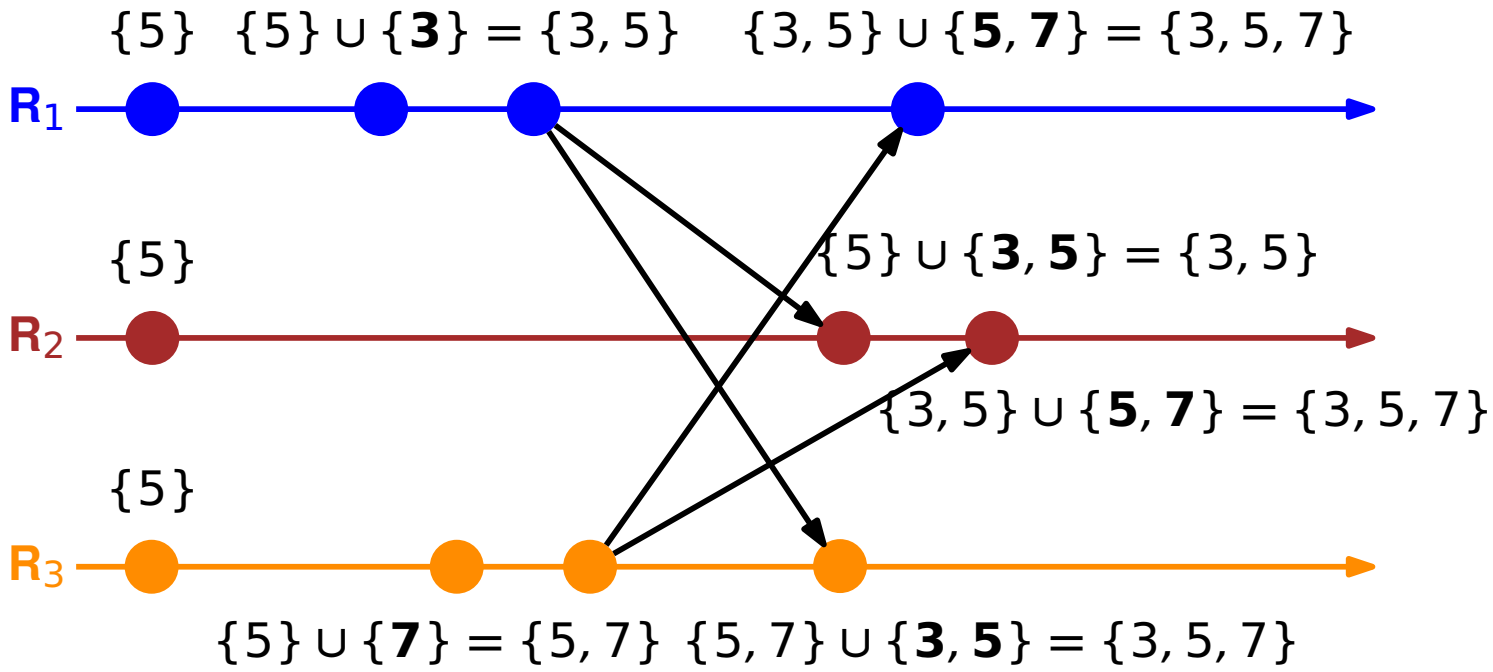
- Sometimes requires real ingenuity

CvRDT Example: Grow-only Set

u add new element to local replica

q return entire set

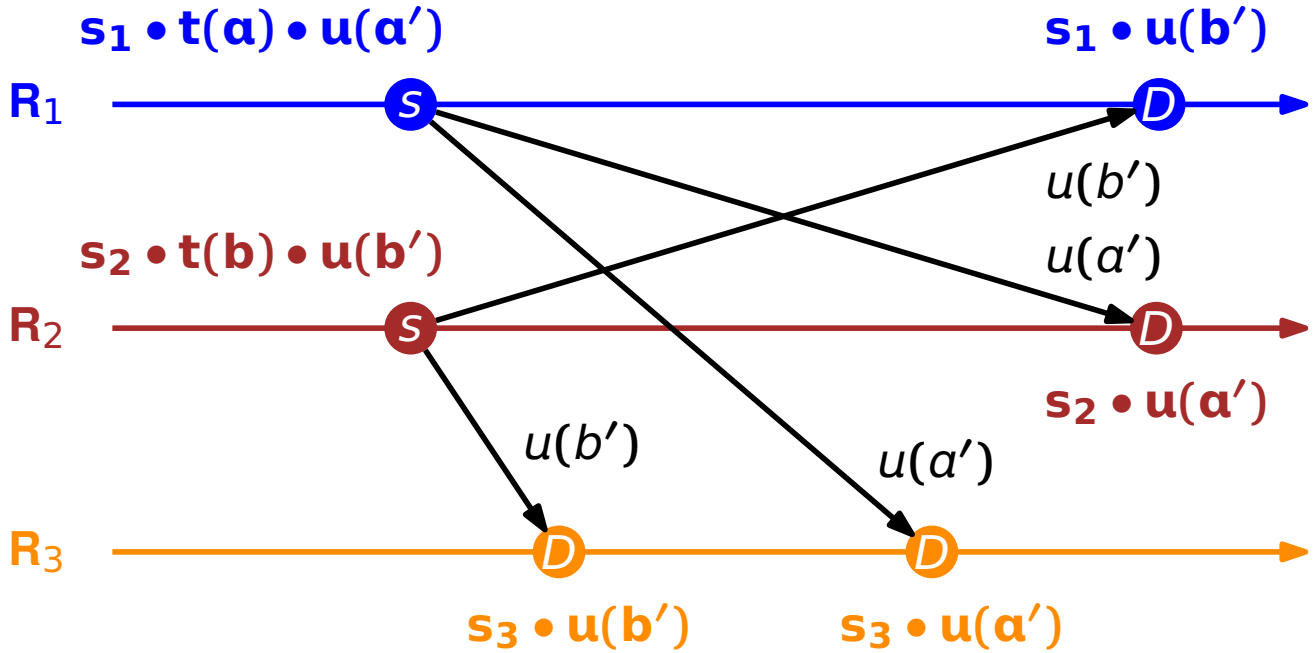
m union of local set with remote set



Question 1 what if *u* also allowed to remove an element?

Question 2 how can we implement a set that also allows to remove elements?

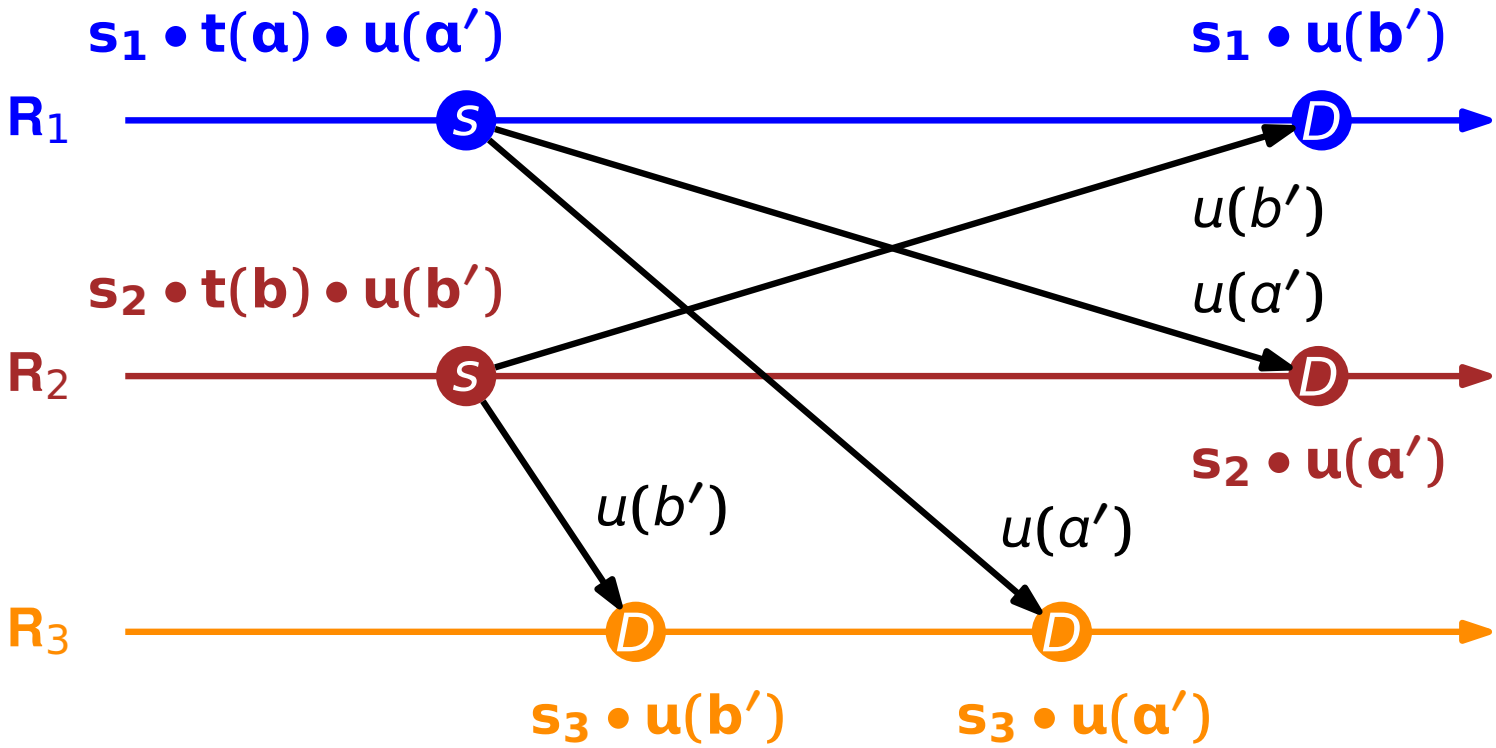
Operation-Based Replication: Commutative RDT



(Replicated) Op-based object a tuple (S, s^0, q, t, u, P) where S, s^0 and q have the same meaning (state domain, initial state and query operation) as for CvRDTs:

- t side-effect-free *prepare-update* operation at **local copy (source)**
- u *update* operation at **all copies**
  - u is executed immediately after the respective *prepare-update* operation at the source
- P delivery precondition

Operation-Based Replication (2/2)



Algorithm use causal-order broadcast for propagating **update** operations

Reliability all updates are delivered by all replicas exactly once

Causal order if $u \rightarrow u'$ and a replica delivers both u and u' , then u is delivered before u' (this is the delivery precondition, P)

- Assume updates are applied in the order they are delivered

Commutative Replicated Data Type (CmRDT)

Enabled update An update u is enabled in state s, if the delivery precondition P holds in s

Update Commutativity updates (t, u) and (t', u') commute, iff for any reachable replica state, s, where both u and u' are enabled:

1. u remains enabled in state $s \bullet u'$
2. same for u', i.e. u' remains enabled in state $s \bullet u$
3. $s \bullet u \bullet u' = s \bullet u' \bullet u$

Commutative Replicated Data Type (CmRDT) is an operation-based replicated object whose update operations commute

Commutativity must hold for concurrent updates:

- Causal-order broadcast does not ensure a totally ordered delivery of messages

Eventual enabled delivery must be proven, if P is different from causal delivery

- By setting P to causal delivery, there is no proof obligation, if we use causal-order broadcast

CmRDT and Strong Eventual Consistency

Theorem Assuming causal-oder delivery of updates and termination of all operations, any op-based object that satisfies the commutativity property for all concurrent updates, and whose delivery precondition P is satisfied by causal delivery, is **strong eventually consistent (SEC)**, where

SEC

Guarantees If no new updates are made to an object, all replicas will **eventually** converge to the same value (as in Eventual Consistency) **and**

- Replicas that have applied the same updates (may be in different order) have an equivalent state

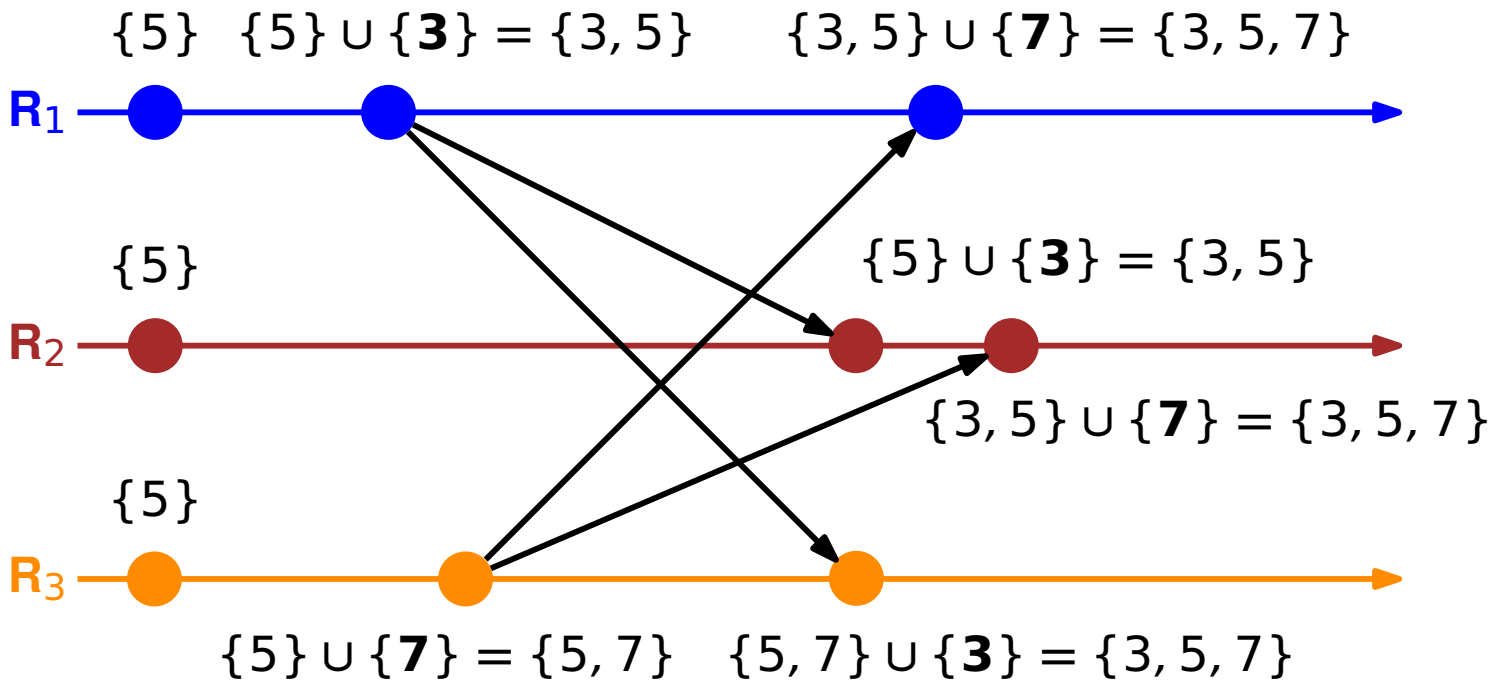
Proof Based on causal-order broadcast

- And on the commutativity of concurrent updates
  - If updates were not commutative, then using causal broadcast to deliver updates might lead to inconsistent states

CmRDT Example: Grow-only Set

t add new element to local replica (set union)

u add delta (of state) to every remote replica

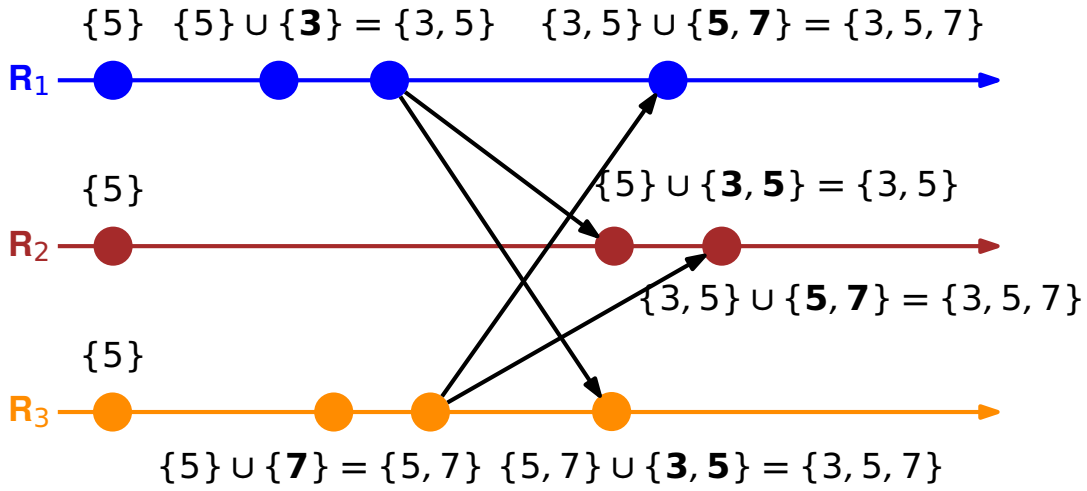


Question 1 what if *u* also allowed to remove an element?

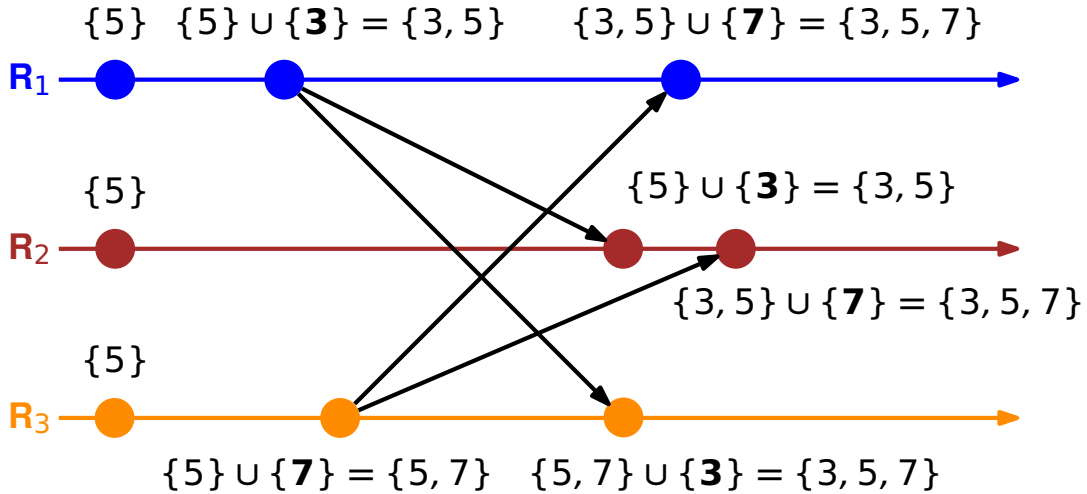
Question 2 how can we implement a set that also allows to remove elements?

State-based vs. Op-based Replication (1/2)

State-based CRDT (CvRDT)



Operation-based CRDT (mvRDT)



What is the difference, and why might it matter?

State-based vs. Op-based Replication (2/2)

Equivalence

- It is possible to emulate one with the other
  - Paper provides a theoretic construction
  - But this is not necessarily the best way to design a CRDT

Operation-based

More efficient Need send only small updates rather than sending all state

State-based

Needs reliable broadcast only which is simpler than causally-ordered broadcast

CvRDT Examples: Counters

Up-only Counter use an integer vector

- u increment value at corresponding index by one
- m element-wise maximum, e.g., $m([1,2,4],[3,1,2]) = [3,2,4]$
- q return sum of all vector values (known as *1-norm*), e.g. $q([1,3,4]) = 8$

Up and Down Counter

- Use two integer vectors:
  - I updated when operation *inc* is invoked
  - D updated when operation *dec* is invoked
- q returns difference between 1-norms of I and D

CvRDT Examples: Sets

Grow-only set

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

Quasi-standard set

- Use two add-only sets

  A upon *add* operation, add it to *A*
  R upon *remove* operation, add it to *R*
  - Should not allow removing an element before adding it

q returns $A \setminus R$
- Allows adding an element only once, i.e. cannot add an element after removing it

CAP Theorem v2

When the network is **partitioned** you cannot achieve both:

Consistency Always return a consistent result (linearizable). (As if there was only a single copy of data, i.e. strong consistency.)

Avaliability Always return an answer to requests with **acceptable** response-time

CRDT's a Solution for CAP?

Avaliability a replica is always available for both "reads" and "writes"

Consistency eventual consistency without the need for undoing operations, i.e. "strong" eventual consistency (SEC)

- SEC is weaker than strong consistency, but it is good enough for many applications, e.g. the shopping cart

Challenge Design suitable CRDTs

Further Reading

- Ion Stoica & Ali Ghodsi, *CRDTs and Coordination Avoidance*, Lecture 8, cs262a, UC Berkeley, February 12, 2018
- Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zirwiski, *Conflict-free Replicated Data Types*, [Research Report] RR-7681, 2011, pp 18. (inria-00609399v1)
- Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zirwiski, *A comprehensive study of Convergent and Commutative Replicated Data Types*, [Research Report] RR-7506, Inria – Centre Paris-Rocquencourt; INRIA. 2011, pp.50. (inria-00555588)