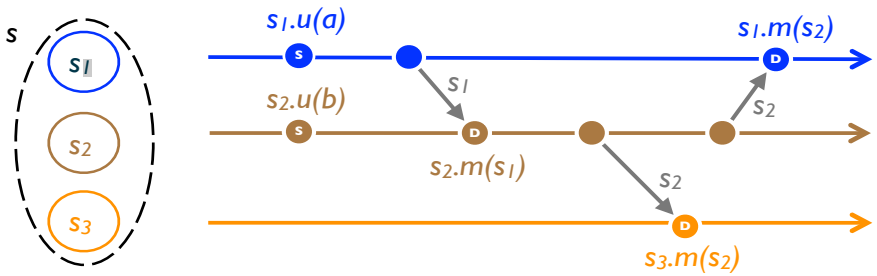
UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Dependable Distributed Systems – 5880V/UE

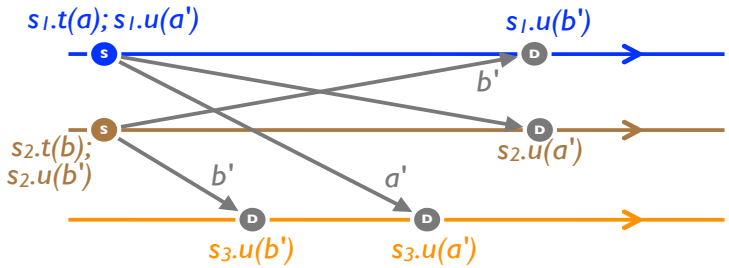
Part 7a. Eventual consistency and conflict-free replicated data types – 2021-10-13

Prof. Dr. Hans P. Reiser | WS 2021/22

UNIVERSITÄT PASSAU



State-base



Operation-based

(source: Shapiro et al.: Conflict-free Replicated Data Types. Research Report RR-7687, 2011, INRIA)

Overview

1. Motivation, CAP theorem
2. Dynamo: Amazon's highly available key-value store
3. Conflict-free replicated data types

Replicated data

Uncoordinated updates

- Fast and easy to implement
- But prone to inconsistent data

Replicated data: strong consistency

Coordinated updates (see previous lectures)

- Deterministic replica behaviour
- Atomic broadcast of requests
- $\rightarrow$ identical state updates

- Consistent but expensive!
  - E.g., Paxos: $O(N^2)$ messages; $\geq \frac{N+1}{2}$ replicas need to be available; slowest of those $\frac{N+1}{2}$ replicas defines latency and throughput

CAP theorem

You can have only two out of three properties in a distributed system

- **Consistency**. “As if there was only one single copy of the data”, also known as “linearizability”
- **Availability**. You always get an answer for your request
- **Partition tolerance**. Continue operating correctly even if the network is split into isolated parts

(see also: Seth Gilbert and Nancy Lynch: “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. ACM SIGACT News, Volume 33 Issue 2, June 2002)

CAP theorem

Assuming that the network is subject to partitioning, you can have only one of the two following properties:

- **Consistency**
- **Availability**

CAP theorem

Assuming that the network is subject to partitioning, you can have only one of the two following properties:

- **Consistency**
- **Availability**

Q: To what extent can you have consistency **and** availability?

Eventual consistency

- Goal: If no new requests arrive, all replicas will eventually converge to a consistent state

- Temporary inconsistencies are allowed
- Update locally first, then propagate (no consensus needed)
- Thus superior availability and scalability for reads and writes

- But: How to handle conflicting update?
  - Application-specific
  - For example, rollback
  - **Usually complicated**

Overview

Dynamo: highly available key-value store

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

"Dynamo: Amazon's Highly Available Key-value Store".

SOSP 2007.

Dynamo: Motivation

- Reliability at massive scale: big challenge
- Slightest outage
  - $\Rightarrow$ significant financial consequences
  - $\Rightarrow$ impact on customer trust

- Dynamo: used by Amazon's core services to provide "always-on" experience

Dynamo: easy usage

Simple interface:

- `get(key)`
  - returns object *or list of objects with conflicting versions*, along with a context
- `put(key, context, object)`
  - stores (replicas of an) object and context for a key
- Context:
  - Opaque encoding of meta-data, such as version number and vector timestamps

Dynamo: partitioning

System should scale incrementally (dynamically partition data over a variable set of nodes)

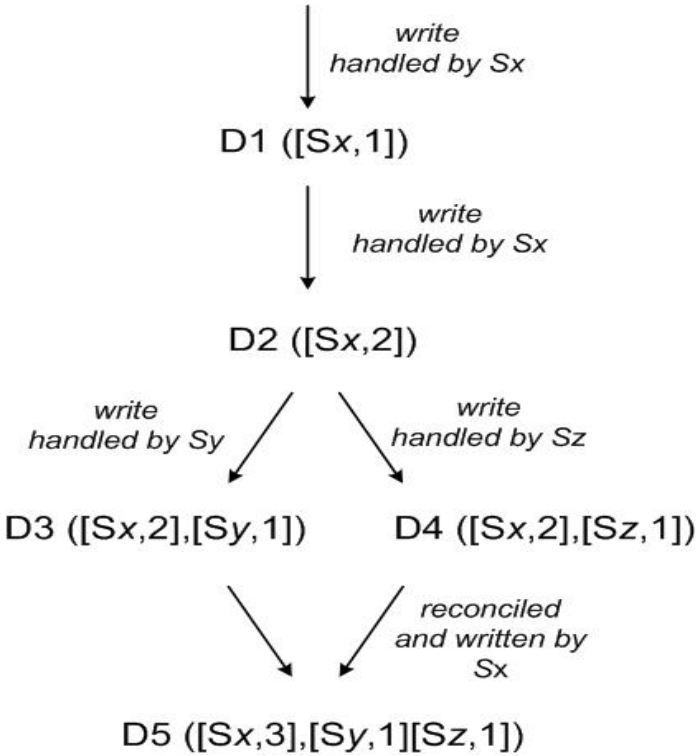
- Consistent hashing: output of hash function treated as a circular space
  - Similar to Chord
  - Nodes have random location on the ring
  - Successor node of key hash responsible for that key
- Challenges:
  - Non-uniform distribution of nodes
  - Non-uniform distribution of data
  - Non-uniform performance of nodes (heterogeneous nodes)
- Solution:
  - Each node has several “virtual” nodes
  - More balanced join/leave load
  - Number of virtual nodes depends on capacity of heterogeneous nodes

Dynamo: replication

- High availability and durability: replication of data on N hosts
- Each key k is assigned to a coordinator node (successor on ring)
- Coordinator in charge of replication on next $N - 1$ successor nodes
  - . . . skipping virtual nodes handled by same physical node
- `get` and `put` with configurable parameters R and W:
  - `get` must involve R nodes, `put` must involve W nodes
  - If $R + W > N$, quorum-like operation, but latency dictated by slowest of R (or W) replicas
  - In practice, usually $R + W < N$ for better latency (but less consistency)
  - Failure handling: using next N **healthy** nodes on ring (not next N nodes)

Dynamo: replication

- Result of replication strategy
  - In best case, consistent replicas
  - But in case of temporary or permanent failures: replicas may diverge
- `put` operation includes vector time stamp indicating causal history
- Semantic reconciliation with application support
  - For example: two versions of a shopping cart, with (a) item A added, and (b) item B added
  - Reconciliation: shopping cart with both A and B added

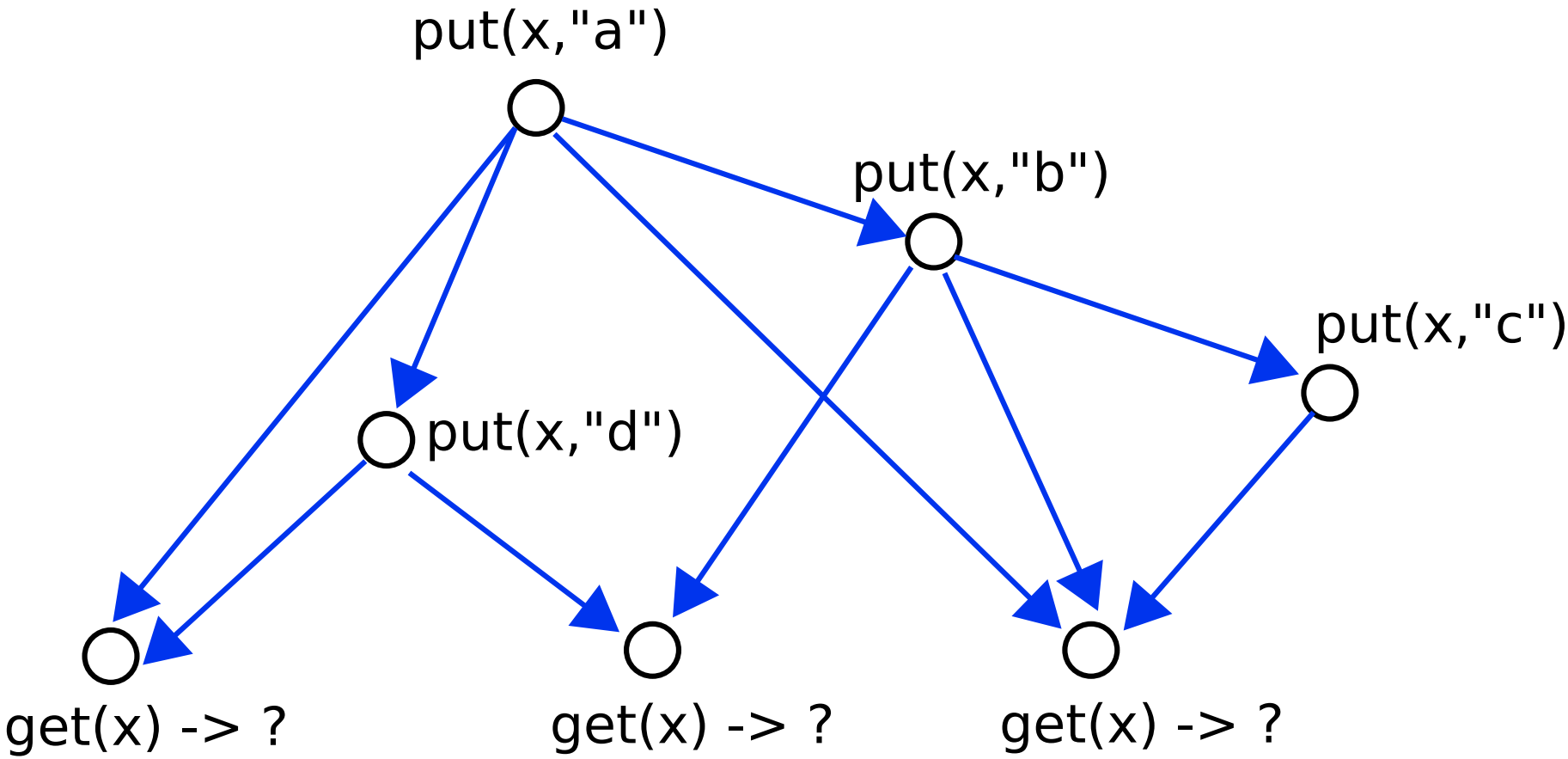


Dynamo: summary

- Some more details on failure handling: see paper
- Reduced consistency guarantees can increase availability and decrease latency
- Requires application support for semantics-aware reconsiliation

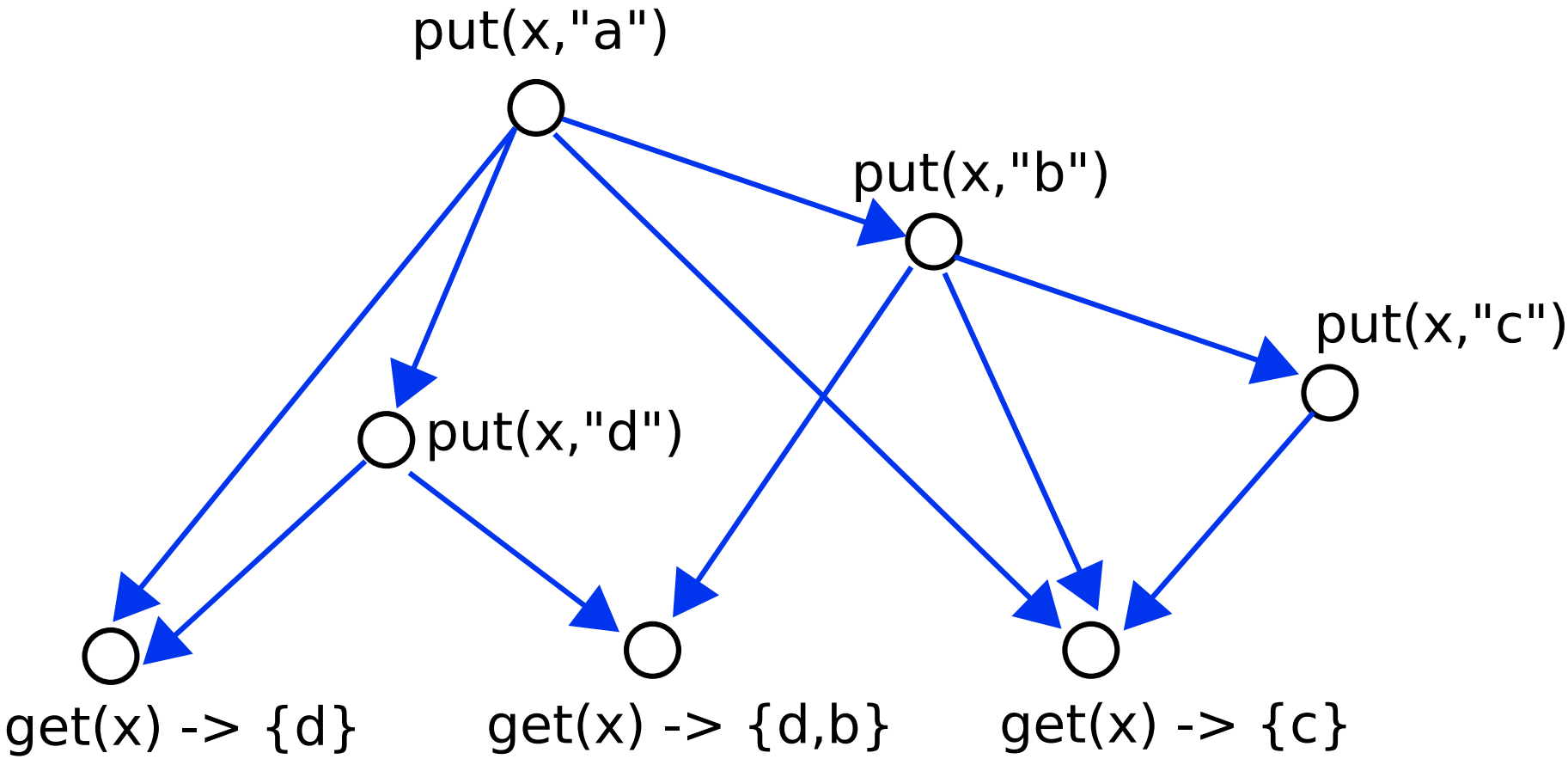
Additional example: read/write register

Assume you have the following interactions with a read/write register.
What do the read operations at the bottom return?



Additional example: read/write register

Assume you have the following interactions with a read/write register.
What do the read operations at the bottom return?



Overview

Conflict-free replicated data types

- Eventual consistency has advantages
  - Availability, partition-tolerance
  - Low latency
- Can we define “conflict-free objects” that do not require application support for semantic-aware reconciliation?

Conflict-free replicated data types

- Eventual consistency has advantages
  - Availability, partition-tolerance
  - Low latency
- Can we define “conflict-free objects” that do not require application support for semantic-aware reconciliation?
- *Strong eventual consistency:* Two nodes (replicas) that have received the same (unordered) set of updates will be in the same state.

Conflict-free replicated data types

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski:

“Conflict-free replicated data types”

SSS’11. Proceedings of the 13th Int. Conf. on Stabilization, Safety, and the Security of Distributed Systems, 2011

Conflict-free replicated data types

- Solves CAP problem
- No consensus required: tolerates $n - 1$ (crash) faults
- How/when is this feasible?

Eventual consistency

Definition of eventual consistency (EC)

- *Eventual update:* 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)
  - This may include rollback or other reconciliation steps

Strong eventual consistency

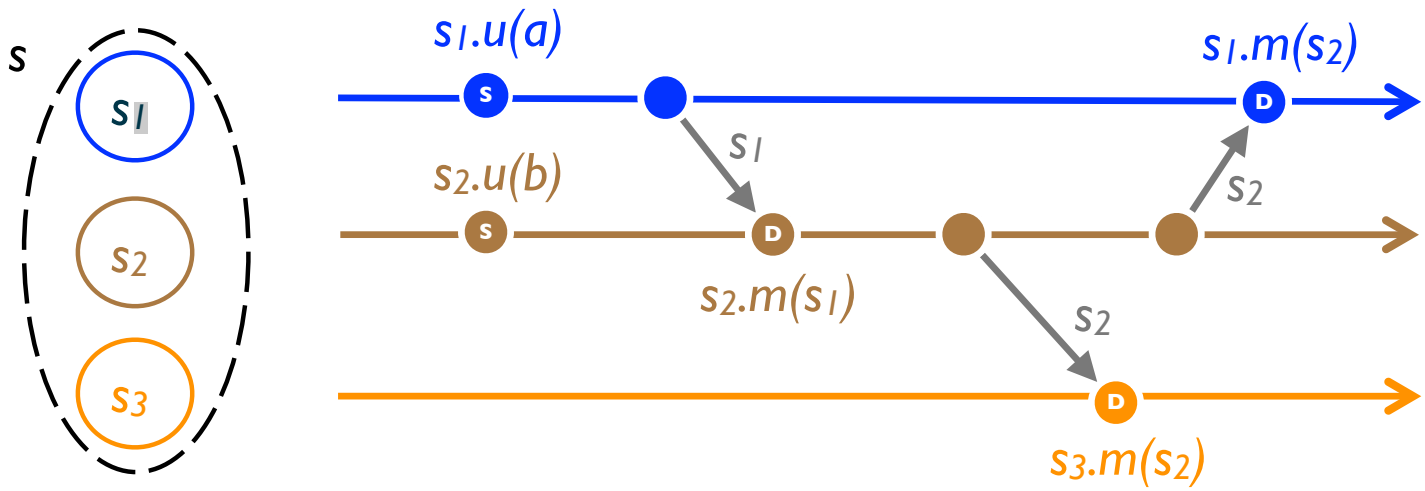
- Same *eventual update* and *termination* as EC
- *Convergence:* Correct replicas that have executed the same updates reach an equivalent state
  - By local update only
  - No rollback, no potentially complicated reconciliation

Conflict-free replicated data types

Two possible styles of specification

- State based
  - Replicas send full state to each other
  - Merge state on receive
- Operation based
  - Log, send operations to all replicas
  - Operations (independent of order) result in same state

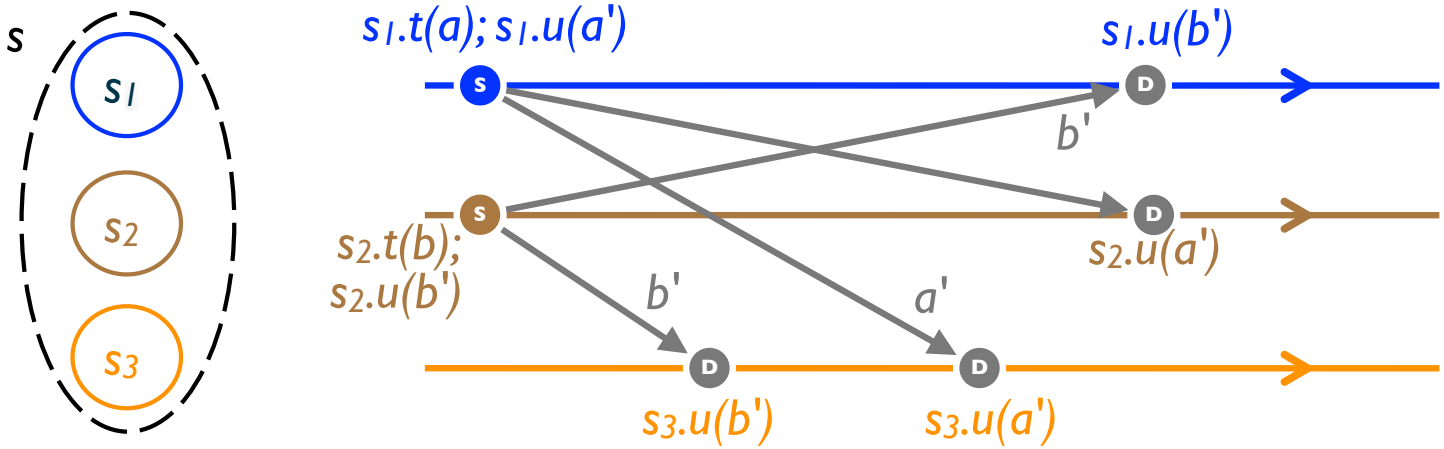
State-based replication



(also called CvRDTs – convergant replicated data types)

- Update operation at one replica
  - Local state update at replicas
  - Periodically, replicas send state updates to others
  - Merge state on delivery
- Sufficient condition:
  - Set of states forms a semi-lattice
    (partially ordered set with least upper bound)
  - Updates are increasing
  - Merge computes least upper bound

Operation-based replication



(also called CmRDTs – commutative replicated data types)

- Update operations sent to *all* replicas
  t: local "prepare" operation; u: distributed "update" operation
  - Local state update at all replicas
  - Usually: causal order of updates (cbcast)
- Sufficient condition:
  - Update are commutative

Operation-based vs state-based

. . . you can prove that state-based and operation-based are equivalent

CRDT examples: G-Counter

Grow-only counter

- CmRDT: broadcast increment operations (easy)
- CvRDT:

```
payload integer[n] P
   initial [0,0,...,0]

update increment()
   P[myID]++

query value() : integer v
   let v = ∑_i P[i]

merge (X, Y) : Z
   let ∀ i∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

CRDT examples: PN-Counter

Positive-negative counter

- CmRDT: broadcast increment/decrement operations (easy)
- CvRDT:

```
payload integer[n] P, integer[n] N
   initial [0,0,...,0], [0,0,...,0]

update increment()
   P[myID]++
update decrement()
   N[myID]++

query value() : integer v
   let v = ∑_i P[i] - ∑_i N[i]

merge (X, Y) : Z
   let ∀ i∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
   let ∀ i∈ [0, n - 1] : Z.N[i] = max(X.N[i], Y.N[i])
```

CRDT examples: G-Set

G-Set: Append-only set (“grow only set”)

- CmRDT: broadcast add operations (easy)
- CvRDT:
  - Set inclusion as order relation
  - Adding elements to set is increasing
  - Set union as least upper bound (merge operation)

CRDT examples: set with add and remove

Can we construct a set data type where we can add and remove elements?

- Problem: Usually “add” and “remove” do not commute

CRDT examples: 2P-Set

Solution 1: 2P-Set: Two Phases Set

Constraints:

- An element may be added only once
- An element may be removed only after it was added

Implementation as CvRDT:

- Maintain two sets A and R, add adds to A, remove adds to R
- query returns $A \setminus R$

Implementation as CmRDT:

- Ensure causal delivery of operations (and operation constraints)

CRDT examples: LWW-Set

Solution 2: LWW-Set: Last Writer Wins Set

Each set element has a timestampt *t* and a visibility flag (*true/false*)
Query returns all elements with visibility flag *true*

Implementation as CvRDT:

- Create union of all set elements
- For each element, use (timestamp, visibility) pair with larger timestamp

Implementation as CmRDT:

- Execute update only if update has larger timestamp than local timestamp of element

CRDT examples: C-Set

Solution 3: C-Set: Counter Set

Each set element has a counter k
Local add: possible only if $k \leq 0$, sets counter to 1 ($\delta = -k + 1$)
Local remove: possible only if $k > 0$, sets counter to 0 ($\delta = -k$)

Implementation as CmRDT:

- Update operation is adding δ to the element's counter
- Elements with $k == 0$ can be removed
- Query returns elements with $k > 0$

More CRDT's

More complex CRDT's are possible

See `https://arxiv.org/pdf/1201.1784.pdf` for examples

For example: graphs

- Useful for, e.g., collaborative editing