Commutative and Convergent Replicated Data Types

Marc Shapiro, INRIA & LIP6
Nuno Preguiça, U. Nova de Lisboa
Carlos Baquero, U. Minho
Marek Zawirski, INRIA & UPMC
Principled approach to Eventual Consistency

CAP: consistency vs. scalability

Eventual Consistency:
- Avoid (foreground) synchronisation
- Diverge, detect conflicts, repair
- Consistent if/when all replicas have received all operations
- Ad-hoc ⇒ error-prone

CRDT: Provable convergence guarantees
- Principled, correct
- No concurrency control: available, fast
- Reconcile scalability + consistency

So far, only ad-hoc approaches
Handful of CRDTs known

Study CRDTs:
- Expose underlying principles, limits
- Expand knowledge of CRDTs
- Catalogue of composable CRDTs

Long-term objective:
- Push the limits
- Radically simplify the design of cloud software
The theory
Sufficient conditions for correctness without synchronisation
State-based replication

Update $f(u)$
- **pre** $u > x$
- $x := \frac{(x+u)}{2}$

Update at source $x_1.f(u)$, $x_2.g()$, ...
- **Precondition, compute**
- **Assign payload**

Convergence:
- **Episodically:** send $x_i$ payload
- **On delivery:** merge payloads

merge $x, y$
- max($x, y$)

merge two valid states
produce valid state
no historical info available
A principled approach to eventual consistency

Semi-lattice

A poset \((S, \leq)\) is a \textit{join-semilattice} if:

- for all \(x, y\) in \(S\) a LUB exists
  \[\forall x, y \in S, \exists z: x \leq z \land y \leq z \land \nexists z': x, y \leq z' < z\]

LUB = Least Upper Bound

- Associative: \(x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z\)
- Commutative: \(x \sqcup y = y \sqcup x\)
- Idempotent: \(x \sqcup x = x\)

Examples:

- \((\text{int}, \leq)\) \(\quad x \sqcup y = \max(x, y)\)
- \((\text{sets}, \subseteq)\) \(\quad x \sqcup y = x \cup y\)
A principled approach to eventual consistency

If
• payload type forms a semi-lattice
• updates are increasing
• merge computes \( \sqcup \)

then replicas converge to LUB of last values

Example: Payload = int, merge = max
A principled approach to eventual consistency

Example CvRDT

If

- payload type forms a semi-lattice
- updates are increasing
- *merge* computes \( \sqcup \)

then replicas converge to LUB of last values

Example: \( f = \text{assign}, \text{merge} = \text{max} \)
A principled approach to eventual consistency

Operation-based replication

At source:
- source precondition, computation
- broadcast to all replicas

Eventually, at all replicas:
- downstream precondition
- Assign local replica

Update \( f(u) \) atSource \( (u) : v \)
\[ \text{spre } u > x \]
\[ v = (x+u)/2 \]

downstream \( (v) \)
\[ \text{dpre } x > 0 \]
\[ x := v \]
Commutative-operation-based objects: CmRDTs

If:
• (Liveness) all replicas execute all dowstreams in precondition order
  • (Safety) concurrent operations all commute

Then: replicas converge
CvRDT $\equiv$ CmRDT

Operation-based emulation of state-based object
- At source: apply state-based update
- Downstream: apply state-based *merge*
- Monotonic semi-lattice $\Rightarrow$ commute

State-based emulation of op-based object
- Update: at-source, add op to set of messages
- Merge: union of message sets
- Execute when $dpre = true$
- Live: eventual delivery, eventual execute
- Commute $\Rightarrow$ semi-lattice

Use state or operations 
as convenient
The challenge:
What interesting objects can we design with no synchronisation whatsoever?
Single-master counter

Increment / decrement

- Payload = int $p, n$
- $\text{increment()} \triangleq [\text{myID()}=42] p++$
- $\text{decrement()} \triangleq [\text{myID()}=42] n++$
- $\text{value()} \triangleq p–n$
- $x \leq y \triangleq x.p \leq y.p \land x.n \leq y.n$
- $\text{merge}(x,y) = (\max(x.p, y.p), \max(x.n, y.n))$
Single-master counter

Increment / decrement

- Payload = int \( p, n \)
- \( \text{increment}() \overset{\text{def}}{=} [\text{myID()}=42] \ p++ \)
- \( \text{decrement}() \overset{\text{def}}{=} [\text{myID()}=42] \ n++ \)
- \( \text{value}() \overset{\text{def}}{=} p–n \)
- \( x \leq y \overset{\text{def}}{=} x.p \leq y.p \land x.n \leq y.n \)
- \( \text{merge } (x,y) = (\max (x.p, y.p), \max (x.n, y.n)) \)
Multi-master counter

Increment / decrement
• Payload: \( P = [\text{int}, \text{int}, \ldots] \),
  \( N = [\text{int}, \text{int}, \ldots] \)
• \( \text{value}() = \sum_i P[i] - \sum_i N[i] \)
• \( \text{increment}() = P[\text{MyID}]++ \)
• \( \text{decrement}() = N[\text{MyID}]++ \)
• \( \text{merge}(x,y) = x \sqcup y = ([\ldots, \max(x.P[i], y.P[i]), \ldots], \ldots, \max(x.N[i], y.N[i]), \ldots)]_i \)
• Positive or negative
Multi-master counter

Increment / decrement

- Payload: \( P = [\text{int}, \text{int}, \ldots], \)
  \( N = [\text{int}, \text{int}, \ldots] \)
- \( \text{value}() = \sum_i P[i] - \sum_i N[i] \)
- \( \text{increment} () = P[\text{MyID}()]++ \)
- \( \text{decrement} () = N[\text{MyID}()]++ \)
- \( \text{merge}(x,y) = x \sqcup y = ([\ldots,\max(x.P[i],y.P[i]),\ldots], \[
\ldots,\max(x.N[i],y.N[i]),\ldots]\)) \)
- Positive or negative
Register

Container for a single atom

Operations:
  • *read: val*
  • *assign (val)*
    - Overwrites preceding value

Concurrent *assign*
  • Single value, arbitrary choice?
  • All concurrent values?
Last Writer Wins Register

CvRDT payload: \((atom \ value, \ timestamp \ ts)\)
- assign: overwrite value, increment ts
- Merge takes value with highest timestamp; other is lost
- \(x \leq y \equiv x.ts \leq y.ts\)
- \(\text{merge} (x,y) = x.t < y.t \ ? \ y : x\)

A principled approach to eventual consistency
A principled approach to eventual consistency

MV-Register

\[ x_1 \approx \text{LWW-Set Register} \]

- Payload = \{ (value, VT \ vv) \ }
- assign: overwrite value, \ vv++

Concurrent updates unioned (no lost updates)

\[ \text{merge } (X, Y) = \]
\[ \{ x \in X \mid \nexists y \in Y: x.vv < y.vv \} \cup \]
\[ \{ y \in Y \mid \nexists x \in X: x.vv > y.vv \} \]
Bookstore anomalies

"An add operation is never lost. However, deleted items can resurface." [Dynamo, SOSP 2007]

Preferred approach: to design a proper Set CRDT
Set

Operations:
• \( add \) (atom \( a \))
• \( remove \) (atom \( a \))
• \( lookup \) (atom \( a \)) : boolean

No duplicates

The prototypical CRDT?
• \( remove \) does not commute with \( add \)
• Approximations: modify semantics
Grow-only Set, state-based

- Build intuition
- Simple examples
- What state do I need to store and transmit?

Payload = set $A$

$add$ (atom $a$)

$merge (x,y) = x \cup y$

- Assume: state eventually delivered
- Why not remove()?
- Trial and error...
- Hmm, let's move on to something else
A principled approach to eventual consistency

2P-Set (state)

Add, remove: 2P-set

- **Payload** = \((\text{Grow-Set} \ A, \ \text{Grow-Set} \ R)\)
- **add** (atom \(a\))
  \[\text{remove} \ (\text{atom} \ a) \ [ \text{spre:} \ a \in A] \]
  \[\text{lookup} \ (a) = a \in A \land a \notin R\]
- \(x \leq y \overset{\text{def}}{=} x.A \subseteq y.A \land x.R \subseteq y.R\)
- **merge** \((x,y) = (x.A \cup y.A, x.R \cup y.R)\)

- \(A=\text{added}\)
- \(R=\text{removed} \ (\text{tombstones})\)
- Once removed, an element cannot be added again
- Remove has precedence over add (absorbing)

In many distr. sys., uses of Set, add creates a unique element, so this is not a limitation
U-Set = no tombstones

2P-Set
Special, common case: a unique
• Never add again
• No tombstones
Correct shopping cart
**Observed-Remove Set (state)**

- **Payload**: Map $M$: element to 2P-Set of tokens
- **Make add unique**:
  
  $$\text{add}(a) = M.\text{add} (a, \text{unique-token})$$
- **Remove the unique elements observed**
  
  $$\text{remove}(a) = M.\text{removeAll} (a)$$
- **lookup**($a$) = $a \in M \land a.\text{tokens}$ not empty
- **merge** ($x,y$) = merge token sets

- Can never remove more tokens than exist
- Op order $\Rightarrow$ removed tokens have been previously added

- Better shopping cart
- What anomalies?
Map

Set of (key, value) pairs

Payload: \( S = \{ (k, v), \ldots \} \)

- \( \text{lookup} \ (k) = \{ v: (k, v) \in S \} \)
- \( \text{add} \ (k, v) = S := S \cup \{ (k,v) \} \)
- \( \text{remove} \ (k, v) = S := S \setminus \{ (k,v) \} \)
- \( \text{removeAll} \ (k) = S := S \setminus \{ (k, _) \} \)

CRDT approximations

- 2P-Map
- PN-Map
- LWW Map
- Observed-Remove Map
Graph

Graph = (V, E)  
where V = set of atoms  
E ⊆ V×V  

addVertex (v) → addEdge (v, w)  
  → removeEdge (v, w)  
  → removeVertex (v)

Any of the set-like CRDTs is OK  
• e.g. 2P-Set ⇒ 2P-Graph

In the general case, cannot enforce global property,  
e.g. acyclic
Tombstone

• 2P-Set: forbid \textit{add-remove-add}
• Graph: \textit{addEdge}(u,v) || \textit{removeVertex}(u)
• Discard when all concurrent \textit{addEdge} delivered
  – i.e. when \textit{removeVertex} stable
  – Wuu, Bernstein/Golding algorithm
• No consensus
• Not live in presence of crash
A principled approach to eventual consistency

Monotonic DAG

- add: between already-ordered elements
- remove: preserves existing order
- Monotonic between remaining elements [restrictive meaning]
- Typical application: concurrent text editing

Causal order too strong for add
Too weak for delete

add-between \((x, y, z)\)
- dpre: \(x, z \in V \land x < z\)
- effect: \(y \in V \land x < y < z\)

remove \((y)\)
- effect: \(y \notin V \land x < z\)
Sequence

Sequence of elements of type $T$

- Co-operative edit buffer: sequence of atoms
- \textit{add-at-location, remove}

Two approaches:
- Linked list
- Continuum
Elements of type \((\text{atom } v, \text{LTS } ts)\)

- Explicit (total order) graph \(x < y < z\)

**add-after \((x, y)\):**

- \(\text{dpre}: \text{add-after}(..., x) \rightarrow \text{add-after} \ (x, ...)\)
- Sequential: \(\text{add-after} \ (x,y) \rightarrow \text{add-after} \ (x,z)\)
  \[\Rightarrow y.ts < z.ts \land x < z < y\]
- Concurrent: \(\text{add-after} \ (x,y) \parallel \text{add-after} \ (x,z)\)
  \[\land y.lts < z.lts \Rightarrow x < z < y\]
Elements of type \((atom \, v, \, LTS \, ts)\)

- **Explicit (total order) graph** \(x < y < z\)

**add-after** \((x, y)\):

- **dpre**: \(add-after(..., x) \rightarrow add-after (x, ...)\)
- **Sequential**: \(add-after (x,y) \rightarrow add-after (x,z)\) \(\Rightarrow y.ts < z.ts \land x < z < y\)
- **Concurrent**: \(add-after (x,y) \parallel add-after (x,z)\) \(\land y.ts < z.ts \Rightarrow x < z < y\)
Assign each element a unique real number
  • position

Real numbers not appropriate
  • approximate by tree
Layered Treedoc

sparse $8^6$-ary tree

binary tree

Edit: Binary tree

Concurrency: Sparse tree

A principled approach to eventual consistency
Rebalance

Tree has nice logarithmic properties

Wikipedia, CVS experiments:
- Lots of removes
- Unbalanced over time

Rebalancing changes IDs:
- Strong synchronisation (commitment)
- In the background
- Liveness not essential
- Core-Nebula: small-scale consensus
Take aways

Principled approach to eventual consistency

Two sufficient conditions:
  • State: monotonic semi-lattice
  • Operation: commutativity

Useful CRDTs
  • Register: Last-Writer-Wins, Multi-Value
  • \( \approx \) Set: 2P (remove wins), OR (add wins)
  • Map \( \approx \) Set + Register
  • Graph \( \approx \) (Set, Set) + \( E \subseteq V \times V \)
  • Monotonic DAG
  • Sequence: list, continuum
CRDTs for cloud computing

ConcoRDanT: ANR 2010–2013

• Systematic study, explore design space
• Characterise invariants
• Library of data types: multilog, K-V store + composition

When consensus required:

• Mix commutative / non-commutative semantics
• Move off critical path, non-critical ops
• Speculation + conflict resolution